The Search for a Cost Matrix to Solve Rare-Class Biological Problems

Mark J. Lawson

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Liqing Zhang, Chair
Lenwood S. Heath
Naren Ramakrishnan
G. Alan Wang
Weiguo Fan

11/16/2009
Blacksburg, Virginia

Keywords: Classification, Machine Learning, Local Search, Bioinformatics

Search for a Cost Matrix to Solve Rare-Class Biological Problems

Mark J. Lawson

(ABSTRACT)

The rare-class data classification problem is a common one. It occurs when, in a dataset, the class of interest is far outweighed by other classes, thus making it difficult to classify using typical classification algorithms. These types of problems are found quite often in biological datasets, where data can be sparse and the class of interest has few representatives. A variety of solutions to this problem exist with varying degrees of success.

In this paper, we present our solution to the rare-class problem. This solution uses MetaCost, a cost-sensitive meta-classifier, that takes in a classification algorithm, training data, and a cost matrix. This cost matrix adjusts the learning of the classification algorithm to classify more of the rare-class data but is generally unknown for a given dataset and classifier. Our method uses three different types of optimization techniques (greedy, simulated annealing, genetic algorithm) to determine this optimal cost matrix. In this paper we will show how this method can improve upon classification in a large amount of datasets, achieving better results along a variety of metrics. We will show how it can improve on different classification algorithms and do so better and more consistently than other rare-class learning techniques like oversampling and undersampling. Overall our method is a robust and effective solution to the rare-class problem.

Dedication

To my friends and family, who have supported me throughout

To Liqing, the best advisor a grad student could have

To Christina, my love and inspiration

GO HOKIES!

Acknowledgments

I would like to acknowledge all of the professors of my committee for their guidance and assistance in the creation of this dissertation.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

A common problem in machine learning is that of classification. Based on a vector of attributes that describe a data member, a classification model determines what label to assign to a given data member. This classification model is generated through training a classification algorithm on a set of training data. While many issues can lead to a classification model that performs poorly in identifying the correct label for a data member, a common issue presents itself to most types of classification algorithms in the form of "rare-class" data.

Basically stated, rare-class data occurs when one class of data is far outweighed by the other (the assumption being that we are dealing with a binary or two-class problem), so that we have a majority class and a minority class. However, no real ratio threshold can be given and there can be cases in which one class is barely outweighed by another (60% majority, 40% minority) and it is a rare-class example while there can be cases where the majority class far outweighs the minority class and yet no issues exist. The latter example has been shown to be true when the two classes are linearly separable, as then no matter how many more majority class examples exist, a classifier will behave well [28].

An attempt at a definition is the following: **Rare-class data is data in which an unaltered classifier performs poorly in terms of identifying the minority class, thus leading to a high number of false negatives and a low number of true positives**. So, in essence, we are not aware that the given dataset is a rare-class set until we have "seen" how the classifier performs on it. If the unaltered classifier does not identify any true positives (the assumption here is that true positives represent a correct identification of a minority class data member), then we are dealing with a rare-class problem.

In a fashion that seems almost ironically phrased, rare-class data is not an uncommon occurrence [6]. A few examples are: security in a web-based system (majority: normal users, minority: intruder) [19], identification of oil slicks on map data (majority: normal land/water, minority: oil) [33], identifying flaws in manufacturing (majority: functional, minority: non-functional) [45], and identifying calcifications in mammogram scans (majority:

no calcification, minority: calcification) [58].

Another common example of rare-class data happens when dealing with multi-class data. Due to the fact that the majority of classification algorithms are designed for two-class classification, a common approach is to break up a multi-class problem into multiple two-class problems. This is done by creating a classifier for each class and then determining if a data member belongs to this class or not. While this is sometimes easier than using a multi-class classification method, it can create instances of rare-class data. If many classes exist, then the identification of one class versus all remaining ones will in some instances create a rare-class problem.

In this paper, we will address the rare-class problem and present our solution to it. First we will define important definitions and metrics that will be used throughout this dissertation (Chapter 2). Then we shall show an in depth look at previous solutions and what their advantages and disadvantages are (Chapter 3). This is then followed by a look at our solution methods (Chapter 4) and a brief look at their implementation (Chapter 5). Finally, we show and discuss the results of our approach by applying it to a variety of datasets (Chapter 6) and present our conclusions (Chapter 7).

Chapter 2

Definitions

2.1 Classification terms

Classification is an often-studied and important aspect of machine learning and computer science in general [57, 53]. In essence the idea is to determine what class a data member belongs to based on prior knowledge. A data member consists of attributes that are meant to adequately define it and which are then used to determine its label (or class). Given a set of data members and their attributes and labels, collectively referred to as training data, a classification algorithm is trained to create a classification model. This classification model then consists of a set of rules that allows it to determine what class a set of attributes (data member) belongs to. To determine the effectiveness of the classification model, it is then tested on a set of test data, typically data that was withheld from training. The attributes are given to the model and the subsequent generated label is compared to the known label in order to determine its performance. The most basic performance metric to be used is accuracy, in which a percentage is given based on how many data members were correctly labeled. But as we shall see in the next section, this is not the only metric.

2.2 Rare-class metrics

The problem with rare-class classification stems from the use of accuracy as the evaluation metric for a classifier. Commonly, classification methods use accuracy to guage their progress in correctly identifying data members. However, when one class far outweighs another, most classifiers tend to classify all data members as belonging to the majority class. In fact, most classifiers abide by the concept of Occam's razor and will therefore focus on a simple hypothesis [37] to maximize accuracy. In this case, the simplest hypothesis that produces the highest accuracy is to state that all data members belong to the majority class. If 99%

of the data members belong to the majority class, then a classifier seems to be performing well on the surface as this means the classifier has an accuracy of 99% (a highly desirable accuracy level). However no minority class members are being identified, as can be clarified in an example.

Say we have created a classification program called *DiagnoseBot*. *DiagnoseBot* takes a given set of symptoms for a patient (temperature, white cell count etc.) and then will tell the user if the patient has a specified disease (the user inputs the name of the disease and the program essentially determines if the patient is "positive" or "negative" for the disease). Assume we are looking for a disease (*DiseaseA*) that has an occurrence rate of 3%, so that 97% of patients will not have it. Now we have created two versions of the program *DiagnoseBotA* and *DiagnoseBotB*. *DiagnoseBotA* classifies all data members as belonging to the majority class. *DiagnoseBotB* correctly identifies those with a disease, but is not especially accurate, giving off false positives.

DiagnoseBotA says that 100 patients are all healthy and therefore has an accuracy of 97%. *DiagnoseBotB* says 90 patients are healthy and 10 have *DiseaseA* (within these 10 are the 3 that actually do have it) and therefore has an accuracy of 93%. According to the accuracy, *DiagnoseBotA* seems to be the better program. However, it does not (and will not) identify anyone who has the disease, which is especially important in medical diagnosis. *DiagnoseBotB* is preferable as false positives (incorrectly stating that a patient has the disease) are not as dangerous as false negatives (incorrectly stating that a patient does not have the disease). So we clearly need better alternate metrics to evaluate these two versions of the program. We will continue to use these examples throughout our discussion.

Before we begin with discussing these new metrics, it is important to first define some terminology. Virtually all metrics stem from one source in evaluating rare-class classification methods: the confusion matrix. As seen in Table 2.1, the matrix consists of all possible types of classifications for a two-class dataset. In our examples, the confusion matrices of our programs would be:

$$DiagnoseBotA = \begin{bmatrix} 0 & 0 \\ 3 & 97 \end{bmatrix}$$

Table 2.1: Confusion Matrix

True Positive (TP) (Correctly Identified as Minority Class Member)	**False Positive (FP)** (Incorrectly Identified as Minority Class Member)
False Negative (FN) (Incorrectly Identified as Majority Class Member)	**True Negative (TN)** (Correctly Identified as Majority Class Member)

$$DiagnoseBotB = \begin{bmatrix} 3 & 7 \\ 0 & 90 \end{bmatrix}$$

While this gives us a better outlook on how they performed than accuracy, a more detailed analysis is desirable.

The first metric we shall look at is called recall. Recall is defined by the following formula:

$$\mathbf{R}ecall = \frac{TP}{TP + FN} \tag{2.1}$$

Put simply, recall gives a percentage of how many of the actual minority class members the classifier correctly identified (FN + TP represent a total of all minority members). Applying this to our example, we can already see how this metric is more advantageous than accuracy as *DiagnoseBotA* has a recall of 0 and *DiagnoseBotB* has a recall of 1. However, this metric is not a perfect indicator, as a version of the program that stated all patients had the disease would also have a recall of 1 (*DiagnoseBotC*).

$$DiagnoseBotC = \begin{bmatrix} 3 & 97 \\ 0 & 0 \end{bmatrix}$$

To balance this shortcoming in recall, another metric is used in conjunction with it: precision. Precision is defined by:

$$\mathbf{P}recision = \frac{TP}{TP + FP} \tag{2.2}$$

Precision gives a percentage of how many of the minority class members as determined by the classifier do in fact belong to the minority class (TP + FP represent a total of positive predictions by the classifier). So our additional classifier (*DiagnoseBotC*) that stated all patients had the disease would have a precision of 0.03. Meanwhile, our *DiagnoseBotB* has a precision of 0.3 which is far more desirable. *DiagnoseBotA* has an undefined precision (due to the fraction being 0/0). Much like recall, precision is not a perfect metric either. Another classifier (*DiagnoseBotD*) could diagnose exactly two patients as having the disease (one false, one correct) and therefore have a precision of 0.5.

$$DiagnoseBotD = \begin{bmatrix} 1 & 1 \\ 2 & 96 \end{bmatrix}$$

So an ideal classifier would have a 1 value for both recall and precision. In such an ideal situation it would be quite evident that this is the ideal classifier. But what if one classifier

has a high recall and a low precision and another has these values reversed? Which classifier is then better?

To simplify this analysis, an additional metric is used: F-Measure.

$$F - Measure = \frac{2}{\frac{1}{R} + \frac{1}{P}} \tag{2.3}$$

F-Measure is the harmonic mean between recall and precision, essentially an average between the two percentages. It greatly simplifies the comparison between classifiers. For instance, it is quite evident that the best version of the *DiagnoseBot* is *DiagnoseBotB*. However its precision is lower than *DiagnoseBotD* and its recall is equal to *DiagnoseBotC*. When we look at the F-Measures of the programs, it becomes clear which program performs best. *DiagnoseBotB* has an F-Measure of 0.46, while *DiagnoseBotC* has one of 0.06 and *DiagnoseBotD* has one of 0.4.

An additional metric that is commonly used is referred to as the G-mean [33]. As opposed to the F-measure, the G-mean calculates a geometric mean between recall and the True Negative Rate (TNR).

$$TrueNegativeRate(TNR) = \frac{TN}{TN + FP} \tag{2.4}$$

$$G - mean = \sqrt{Recall * TNR} \tag{2.5}$$

The idea here is that a balance is struck between the identification of each class. Because *DiagnoseBotA* and *DiagnoseBotC* only identify one class, they both have a G-mean of 0. Again, the best G-mean is given to *DiagnoseBotB*, 0.96. *DiagnoseBotD* fairs worse with a G-mean of 0.57. Interesting to note here is that a larger discrepancy exists between B and D in their G-mean values than in their F-Measures.

The benefit of these metrics is that they represent single values that can be used to compare the performance of multiple classifiers (especially F-measure and G-mean). Much in the way accuracy is normally used, they represent a simple way to state "Classifier A performs better than Classifier B." In our example, we explained why *DiagnoseBotB* is preferable to the other classifiers as it identifies all infected patients at the cost of making some false positive claims. This is reflected in both F-measure and G-mean.

But sometimes, using a single scalar value does not "show the whole picture." In order to see a more detailed, visual representation of classifier performance, an ROC graph can be used (a great review and how-to-guide can be found in [18]). Receiver Operating Characteristics (ROC) originally stem from the evaluation of classifiers in signal detection theory and are now commonly accepted as a method of evaluating classifiers in rare-class situations. An

ROC graph is one that plots the false positive rate of a classifier against the recall (in the graph this is listed as "true positive rate"). An empty ROC graph can be seen in Figure 2.1.

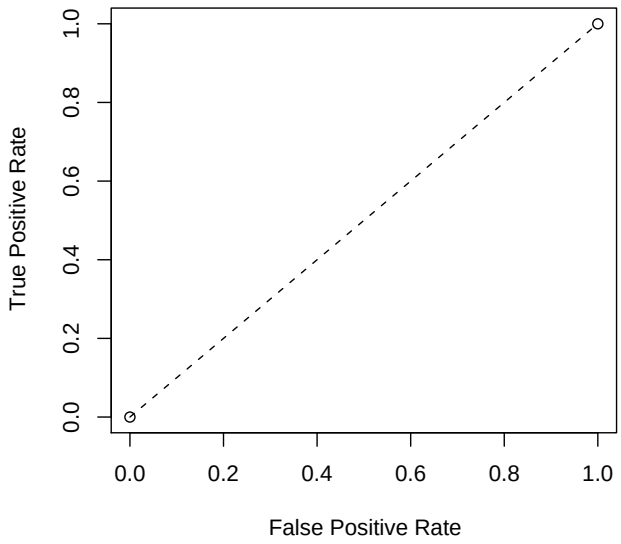

Figure 2.1: ROC Space

The dashed line represents values that result from randomly guessing. So if a random-guessing classifier guesses the positive class 60% of the time, it will have a recall and false positive rate of 0.6. A perfect classifier on the other hand would have a recall of 1 and a false positive rate of 0, which is located in the upper left corner. So the closer a classifier is to the upper left corner, the better it performs. Interestingly, the worst outcome for a classifier is to lie directly on the random diagonal. While technically a classifier that is in the lower right hand corner has worse performance, its evaluations can be negated and its value will be in the upper left.

In Figure 2.2, we have placed the four DiagnoseBot classifiers into the ROC graph. *DiagnoseBotA* and *DiagnoseBotC* both fall on the diagonal, showing them again to have the worst performance. *DiagnoseBotA* makes no positive predictions, so while its false positive rate is 0, its true positive rate is also 0. *DiagnoseBotC* makes only positive predictions so both true positive rate and false positive rate are 1. As in all previous metrics, *DiagnoseBotB* is shown to perform best as it is closest to the upper left corner.

As our classifier evaluations are discrete (with only one value for both true positive and false positive rates), looking at the points and evaluating their distance from the upper left are all we can use to evaluate them. Alternatively we can connect each point to (0,0) and (1,1) and evaluate the size of their "curves" (triangles). There exist classifiers (such as NaiveBayes)

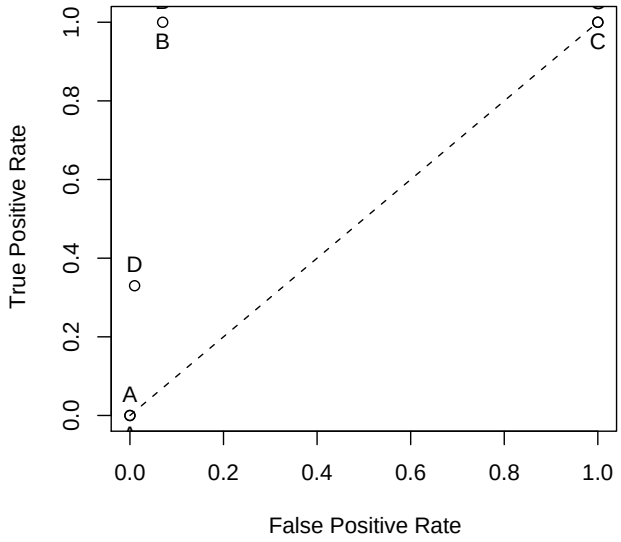

Figure 2.2: ROC of DiagnoseBot

that give a more detailed description of their ratio between both rates and a more detailed curve can be created. So the closer the curve comes to the upper left corner, the better it performs, and with classifiers like NaiveBayes we can even state under what conditions it performs best. The results of an ROC analysis can also be summarized as a scalar by taking the area under the curve known as the AUC (Area Under the ROC Curve). The larger the area, the better the classifier performance.

Additional graphical methods include Precision-Recall Curves [12] (which can also be simplified into a scalar value by taking the area under the curve) and Cost Curves [14], which give additional information as to which classifier performs best and under what conditions.

2.3 Cost Matrix

Based on the confusion matrix, a cost matrix C allows us to punish misclassifications and reward correct classifications.

$$C = \begin{bmatrix} C_{0,0} & C_{1,0} \\ C_{0,1} & C_{1,1} \end{bmatrix} \tag{2.6}$$

So $C_{1,0}$ and $C_{0,1}$ represent punishments for misclassifications and $C_{0,0}$ and $C_{1,1}$ represent

rewards for correct classifications. To differentiate between rewards and punishments we use positive values for punishments and negative values for rewards. This cost matrix is then used to alter the learning of a classification algorithm according to these rewards/punishments. All values are integers as this allows for simpler manipulations, as we shall show in the Methods section.

Chapter 3

Literature Review

Now that we have explained the problems with rare-class data and shown ways to evaluate a classifier's performance on them, it is time to approach the subject of how one deals with this type of data. Rare-class data is a common problem and a variety of approaches have been developed to deal with them. A comprehensive review of all methods is beyond the scope of this work, but fortunately the approaches can be categorically defined and further elicited through generalized descriptions of these approaches and recent examples of their implementations.

The most basic way to break down the methods on dealing with rare-class data is based on what is being altered: the algorithm of the classifier (algorithm-level) or the data being learned/classified (data-level). An algorithm-level approach will alter the way an algorithm creates a classifier, basing it on values or metrics (sometimes user-determined, other times determined through repeated learning iterations) that are different from just determining a classifier with the highest accuracy. This is done by either using a wrapper around the classification method that alters the way it learns and processes the training data (sometimes referred to as a "metaclassifier") or by altering the classifier itself (for instance, pruning branches in a decision tree).

The other main method is the data-level approach. As opposed to the algorithm-level, here the training data is changed to fit the classification method. The issue that is addressed is the fact that the ratio is skewed towards the majority class. So if the ratio was more manageable (such as 50:50 or similar), a normal classification method could be used. In order to achieve this, there are two main methods (and it is conceivable that they can be combined): undersampling (minimizing the amount of the majority class) and oversampling (maximizing the amount of the minority class). These methods will be explored in further detail in the following sections.

3.1 Oversampling

Oversampling attempts to solve the rare-class problem by increasing the amount of minority class members (what we have referred to as the positive class) in the training data. Generally this is done by reusing minority class members in the training/learning process or by creating new, synthetic minority class members based on existing ones.

The advantage of this approach is that a normal classification method can be used. The minority class members can be increased to the point where their amount is balanced against the initially majority class members. Furthermore, unlike what we will see in undersampling, no data is lost. The classifier uses all of the original training data (plus potentially some new data).

There are two main ways to go about oversampling (reusing existing or creating new data members) and each has its own disadvantages. Reusing will count the minority class members multiple times during the learning process, which has a high likelihood of leading to overfitting. The classifier will become too familiar with these minority class members and be unable to identify potential minority class members that it has not seen before. Creating new data members on the other hand, will hopefully address this issue by giving the classifier a varied set of minority class members. However, the creation of these new members is a tricky process. Due to the fact that the new minority class members are synthetic, there exists the very real possibility that these new data members are "wrong" (unrepresentative of the actual minority class) thus giving the classifier false information.

Despite these problems, oversampling remains a popular approach in dealing with rare-class data. When the ratio is not too drastic, it can lead to good performance results, especially if the reuse of minority class examples is randomized. Janpowicz et al. [28] had the best results with oversampling with replacement when evaluating multiple methods of dealing with rare-class data. Furthermore, it is rather simple to implement.

3.1.1 Example: SMOTE

An example of an oversampling method is SMOTE which stands for Synthetic Minority Oversampling TEchnique [7]. As the name already suggests, SMOTE creates new minority class data members. It does this based on the known minority data members. Taking all minority class members, the algorithm determines the 5 nearest neighbors for each data member. Then, using each minority class member as a root, it randomly picks a nearest neighbor and calculates the attribute differences between the two data members. A new data member is created by adding each attribute difference times a random number between (0,1) to the root data member. Thus a new data member is created that is "between" the two existing data members, and an equal amount of synthetic data is created using each existing data member as the root. Of course an implicit requirement of this algorithm is

that every attribute is numeric and continuous.

SMOTE has been shown to be a useful method for oversampling data, achieving better performance in "normal" classifiers. It has been shown to be extremely effective when combined with C4.5 (a decision-tree classifier) and somewhat effective with RIPPER (a rule-based learner) and NaiveBayes. It can also be used to create ROC curves for discrete classifiers (as in our examples in the previous section) by creating different data amounts for the classifier to learn/train on.

There are a variety of implementations and variations of SMOTE. The authors/creators of SMOTE have demonstrated its effectiveness with two separate methods: SMOTEBoost [8] and Wrapper SMOTE [5]. SMOTEBoost combines SMOTE with AdaBoost, a popular boosting method. Boosting works by performing multiple learning iterations (creating a classifier for each one) and focusing on those data members the classifier learned incorrectly in the next iteration. SMOTEBoost takes those minority class members the previous classifier guessed incorrectly and increases their amount for the next iteration. It also has a way to deal with discrete values by using an approach called the Values Distance Metric. It was quite effective with RIPPER and shown to be better than AdaCost [17], a method that combines cost values of misclassification with boosting.

Wrapper SMOTE is an attempt at creating a program that will automatically create an "ideal" amount of training data for a classification method. Using a greedy-based search that evaluates and improves upon the results of a classifier based on the rare-class evaluations metrics we have already discussed, the wrapper automatically generates an ideal amount of data for a given classifier. The authors show this to be a very effective approach.

Another variation is Borderline-SMOTE [23]. As opposed to merely oversampling all minority class members, this approach focuses on oversampling those minority class members that are "on the borderline" between minority and majority class. These borderline members may include nearest neighbors in the SMOTE algorithm that are actually a part of the majority class. The idea here (which we shall revisit with the concept of undersampling) is that the differentiation between majority and minority class members is emphasized and thus better discernible by a "normal" classifier.

SMOTE can also be quite effective when dealing with multi-class problems such as protein classification [61]. In this research, the authors had m classes (proteins) to classify and thus created m classifiers. The majority class was split into equally sized groups for training for each classifier and the same amount of minority class members was created for each classifier using SMOTE. A simple approach, it proved nonetheless effective according to the authors.

3.2 Undersampling

Undersampling attempts to solve the rare-class problem by decreasing the amount of majority class members (what we have referred to as the negative class) in the training data. Generally this is done by filtering the majority class (according to some guidelines) or by merely randomly eliminating majority class members from the training data.

The advantage of this approach (in the same way as oversampling) is that a normal classification method can be used. The majority class members can be decreased to the point where their amount is balanced against the initially minority class members. But unlike in oversampling, no new data needs to be created and no data is repeated. Only original data is included in the training data.

A disadvantage in this approach should be quite evident. In data classification a general rule of thumb is that the more data, the better the classifier can learn. However, here one is minimizing the data, essentially ignoring observations that were made about the majority class. While this could be compared to eliminating "outliers", generally a large number of majority class members need to be eliminated in order to balance out the ratio of majority to minority. Furthermore, what data should be eliminated? Eliminating the wrong ones could cause the classifier to ignore specific instances that may become more important and then become misclassified. Quite often this will then lead to a large amount of false positives and in general, undersampling has not been shown to be as effective (or as easy to implement) as oversampling [28].

So what data should be eliminated in order for undersampling to be effective? As stated, the elimination of data can be treated in a similar fashion to identifying outliers. However, here we need to eliminate more than the usual approaches allow us to. The ideal set of training data would be a situation in which we were able to pinpoint the exact values where majority and minority class differentiate. So we need to filter out all those majority class members that are "furthest away" in terms of euclidean distance from the minority class.

3.2.1 Example: Undersampling through Clustering

A recent approach on this was used in the classification of two different biological, rare-class problems (splice site prediction and protein subcellular localization) [60]. The authors used a clustering approach to undersample the data. Using hierarchical clustering, they partitioned the data (both majority and minority class members) into clusters based on similarity (Euclidean distance). This created pure (only one class) and non-pure (both classes) clusters. For each non-pure cluster, a classifier is created. The classifiers are then combined using weighted voting.

This approach is further illustrated in Figures 3.1 and 3.2. In Figure 3.1, we see a graphical representation of minority and majority class members (in this figure and in subsequent

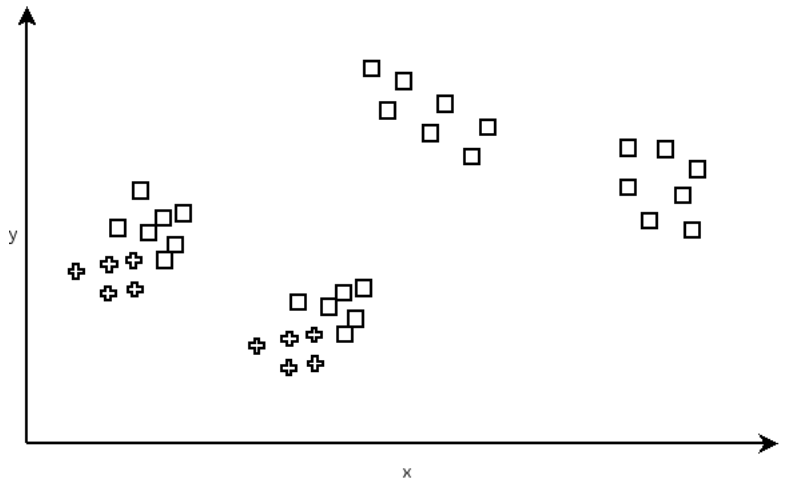

Figure 3.1: Minority (crosses) and Majority Class (squares) members

ones, minority class members are denoted with a cross and majority class members are denoted with a square). As is quite evident, some majority class members are closer (i.e. smaller Euclidean distance) than others from the minority members. Furthermore, there are definable groups that are similar (between the two classes), and groups that are outliers. Looking then at Figure 3.2 we can see how these data members could potentially be clustered. The pure clusters contain only majority class members and the non-pure clusters contain both data members. The non-pure clusters are then used in the next step.

In Figure 3.3 we can see how these non-pure clusters are used to learn the dataset. Each non-pure cluster has its own classifier trained on it (the classification method used is Multi-Layer Perceptron, which was shown by Japkowicz et al. to perform relatively well on rare-class data [28]). These classifiers are then combined using a weighted majority vote (in which each cluster is assigned a weight based on the ratio of majority to minority class members in the cluster).

3.3 Cost-sensitive classification

Cost-sensitive classification attempts to solve the rare-class problem by adjusting the learning of the classification algorithm being used. Generally this is done by creating costs associated with the misclassification of the minority class (what we have referred to as the positive class) and adjusting the learner based on a punishment-reward system.

The advantage of this approach is that the training data remains unaltered. No data is added/repeated (oversampling) and no data is filtered out (undersampling). Furthermore, the ratio between majority and minority class members could factor into the learning of the classifier and is preserved in this methodology.

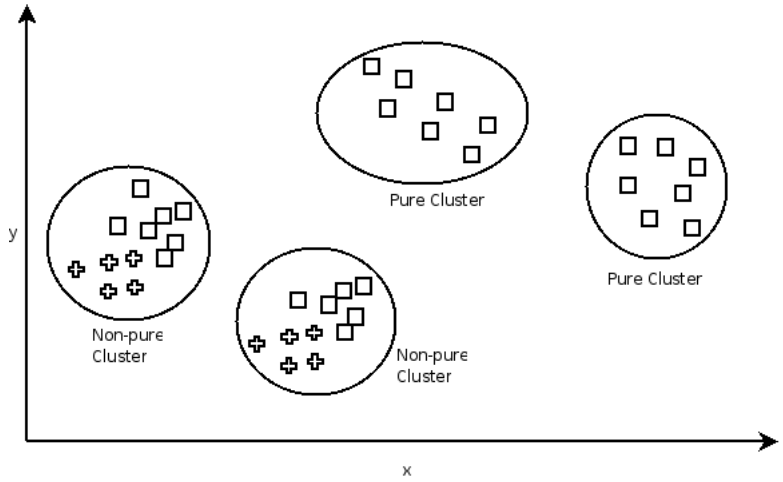

Figure 3.2: Clusters of Minority and Majority Class members

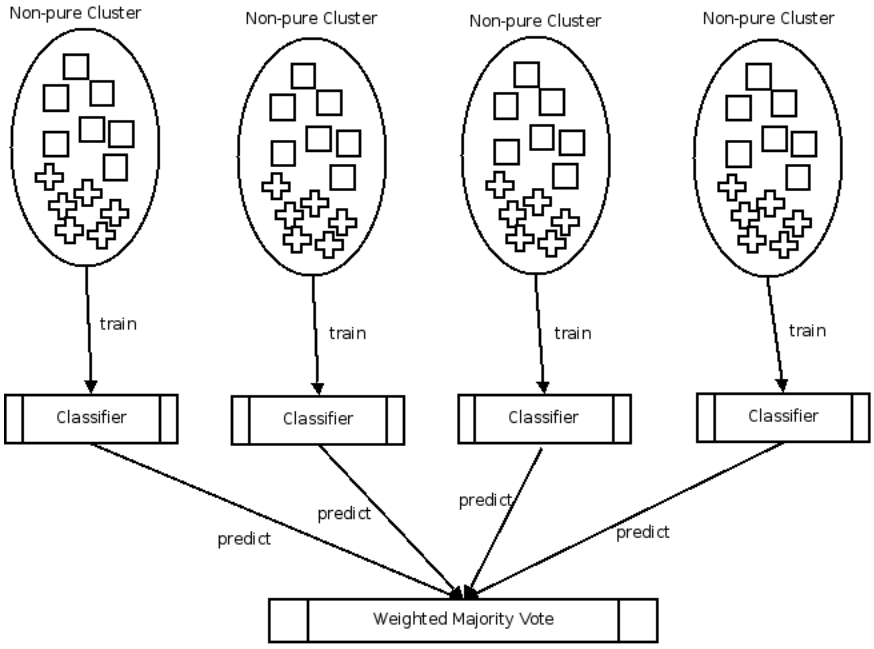

Figure 3.3: Clustering Flowchart

As mentioned earlier, a cost-sensitive classifier uses cost values to alter the learning of the classifier. However, these cost values must be given in advance and are generally unknown for a given dataset. This is usually solved by experimenting with many cost values, which can prove to be time-consuming. Furthermore, once ideal cost values have been determined they are rarely, if ever, transferable to a different dataset.

A meta-classifier, cost-sensitive classification comes in many types. Typically, they employ cost values to punish a classifier for misclassifying a data member. Some research states it is better to punish misclassification of both class members [30] as just focusing on the minority class will result in an over-emphasis on recall. Others, however, have shown to produce good results by focusing on punishing misclassification of the minority class [15, 50].

3.3.1 Example: MetaCost

An attempt that is focused on approximating an optimal classification model, especially in regards to rare-class problems, is MetaCost [13], shown in Algorithm 1. The basic idea of MetaCost is to take a normal, unaltered classifier and adjust the learning with a cost matrix. This is done through a series of steps. The first step is to take the training data and create multiple bootstrap samples of the data. These bootstrap samples are then used for training to create an ensemble of classifiers. The ensemble of classifiers are then combined through a majority vote to determine the probability of each data object x belonging to each class label. Next, each data object in the training data is relabeled based on the evaluation of a *conditional risk* function (Equation 3.1), and a final classifier is then produced after applying the classification algorithm to the relabeled training data.

The key aspect in the MetaCost learning process is to minimize *conditional risk*.

$$R(i|x) = \sum_{j} P(j|x)C_{i,j}. \tag{3.1}$$

The conditional risk $R(i|x)$ defines the cost of predicting that data object x belongs to class label i instead of class label j, $P(j|x)$ is the probability that data object x belongs to class label j, and $C_{i,j}$ is the cost for making such a classification. $C_{i,j}$ corresponds to entries in the cost matrix (Equation 2.6), essentially a variant of the confusion matrix, where $i \in \{0, 1\}$ and $j \in \{0, 1\}$. The cost matrix allows one to punish misclassifications and reward correct classifications, for example, by negative and positive values, respectively. Clearly, the success of the evaluation of the *conditional risk* function and thereby the performance of the MetaCost prediction rests on the cost matrix. Imaginably a "bad" cost matrix can distort the learning and produce a "bad" classifier. Therefore, it is imperative to identify a "good" cost matrix.

Algorithm 1 Adapted MetaCost Algorithm

Input:
S is the training set
L is a classification algorithm
C is a cost matrix
m is the number of bootstrap samples to generate
n is the number of data objects in each bootstrap sample
p is $True$ if L produces class probabilities

{0,1} is the set of classes

Function MetaCost (S, L, C, m, n, p)

$S' \leftarrow \emptyset$

for $k = 1$ to m **do**
  Let S_k be a bootstrap sample of S with n data objects
  Let M_k = Model produced by applying L to S_k
end for

for $x \in S$ **do**
  for $j \in \{0,1\}$ **do**
    Let $P(j|x) = \frac{1}{m} \sum_{k=1}^{m} P(j|x, M_k)$
    where
      if p **then**
        $P(j|x, M_k)$ is produced by M_k
      else
        $P(j|x, M_k) = 1$ for the class predicted by M_k for x, and 0 for the other class
      end if
  end for
  Let the class of x = $\text{argmin}_i \sum_j P(j|x)C_{i,j}$
  $S' \leftarrow S' \cup \{x\}$
end for
Let M = Model produced by applying L to S'

return M

Chapter 4

Cost Matrix Optimization

In order for MetaCost to be effective, we need to find an optimal cost matrix. This chapter focuses on our approaches to finding this cost matrix.

4.1 Basic Idea: Optimization of a Cost Matrix

In MetaCost, the input values m, n, and p are essentially tweaks or givens of the algorithm once the type of classification algorithm is determined. To simplify the function call, we can assume some default values for them, thus, the call to the MetaCost algorithm becomes a function of S, L, and C.

Let us define an evaluation function $\text{Eval}(M, T)$ that takes as input a generated model M based on a cost matrix C and produces an evaluation on how it performs on test set T. Assuming that we have access to the set of all possible cost matrices $(C_i, i \in N^*)$, we can then search for the cost matrix that achieves the highest evaluation value,

$$C_{best} = \arg\max_{C_i} \text{Eval}(M_{C_i}, T), \tag{4.1}$$

and denote C_{best} as the *optimal* cost matrix for the given data and classification algorithm. This search can also be seen in Figure 4.1.

So the problem is how to find the *optimal* cost matrix computationally. While an exhaustive search of all possible cost matrices can guarantee that we find the *optimal* cost matrices, it is far from efficient if not impossible. As such, we propose three search methodologies to find a "best" cost matrix: a greedy search, a simulated annealing search, and a genetic algorithm search.

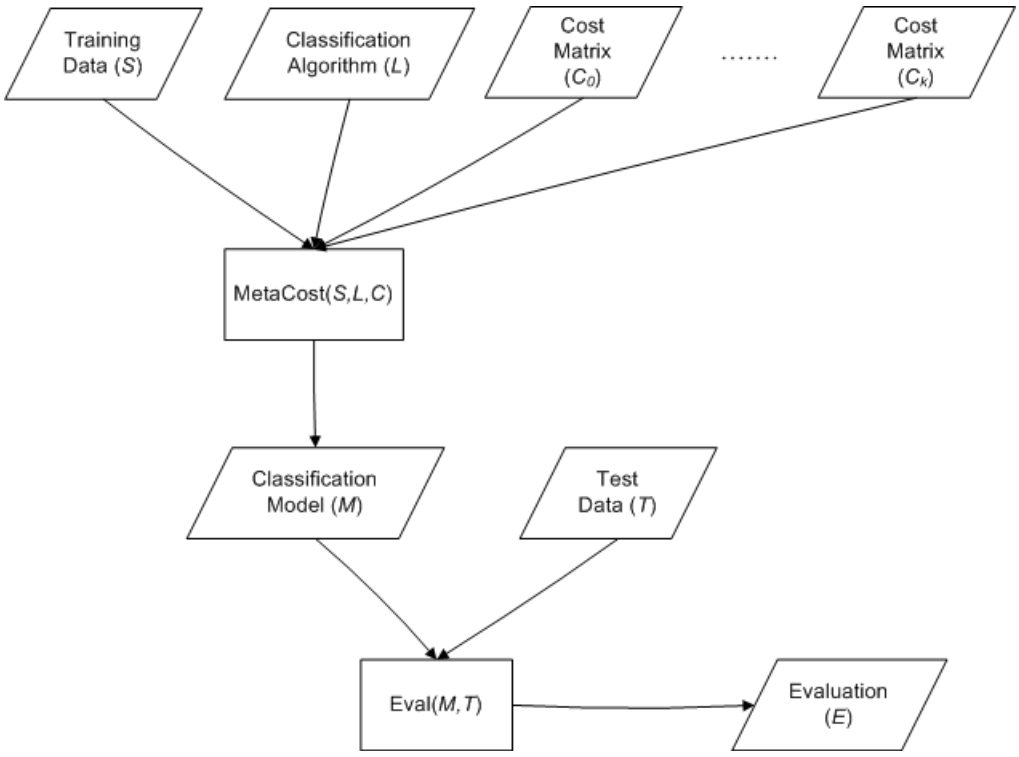

Figure 4.1: Basic Cost Matrix Search Process

4.2 Greedy Search

4.2.1 Process

A greedy search is typically a very simple search method. The idea is to take a current solution to a problem, generate new solutions based on this solution, and then replace the current solution with a new, better solution. If no new, better solution can be found, then the process is halted. Our application of this approach to the search for cost matrices can be seen in Figure 4.2. The main difference in this approach is that we are tracking two solutions: a current best and an overall best.

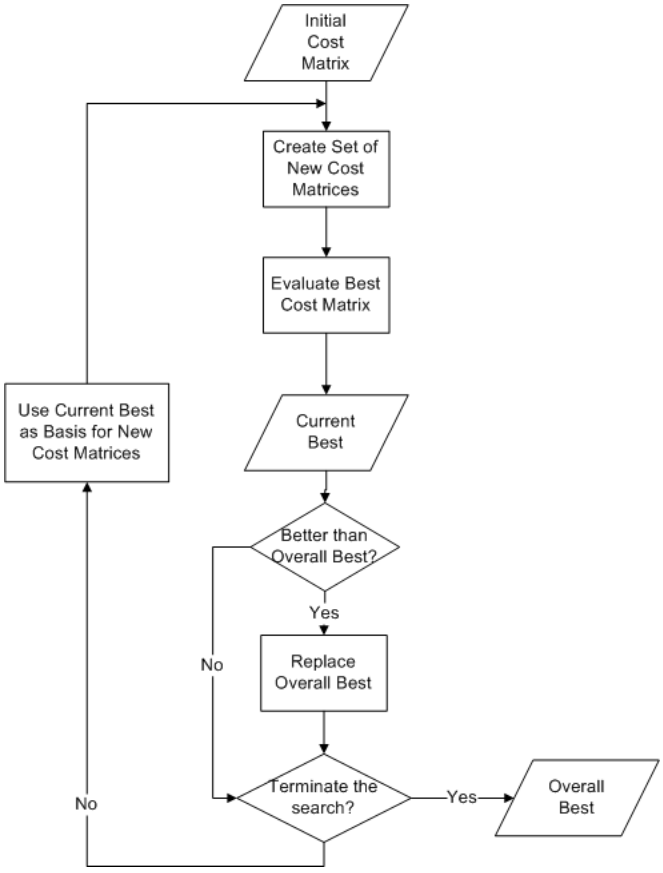

Figure 4.2: Greedy Search Flowchart

Shown in Algorithm 2, the basic idea of the greedy search is to start with an initial cost matrix and to increment its costs to find a cost matrix that achieves a better evaluation value. An initial cost matrix is typically a cost matrix that will create a model that is the same as the model created by an unaltered classifier. Our positive class is the minority class.

Starting with this initial cost matrix, the method creates seven new ones. Each of these cost matrices represents a different combination of incrementing/decrementing the costs (correct classifications are decremented by one and misclassifications are incremented by one). Of note here is that the cost for correct classifications of the majority class (negative class) is not adjusted and left at zero. This is due to the fact that typically a poor classifier will order most (if not all) data members as belonging to the majority class. Therefore, there is no need to "reward such behavior" and our method has fewer cost matrices to test. After creating these seven new cost matrices, each one is used to create a new model through MetaCost, using the given training data S and classification algorithm L. After these models have been created, they are evaluated on the given test set T using the evaluation function $\text{Eval}(M, T)$. The model that has the highest evaluation value is kept and its cost matrix is used to initialize the next iteration of cost matrix creation.

As can be seen in the algorithm, the method keeps track of two cost matrices, a "current best" cost matrix and an "overall best" cost matrix. This was done in order to overcome one of the common problems with greedy searches, that of finding a local maximum that is lower than the global maximum. So when a potential poor local maximum is reached, it can be stored as the overall best and the method can essentially "look ahead" to see if a better cost matrix can be found. If a better one is found, the overall best cost matrix is updated.

The parameter n is passed into the function to give a count of how many times the creation of new cost matrices occurs. A simple check of whether the overall best matrix is the same as the current best cost matrix serves as an indication of whether a maximum (local or global) has been reached. If not, the number of iterations can be increased.

Two aspects are worth noting. First, the evaluation function $\text{Eval}(M, T)$ can be based on any of the metrics for rare-class predictions such as F-measures, ROC curves, and G-mean. Second, the search for the best cost matrix can only improve upon a base classifier. At worst, the method will work as well as an unaltered classifier. This is due to the fact that a model that is built by the MetaCost algorithm with the initial cost matrix is identical to a model that was built using only the base classification algorithm. So if no better cost matrix is found, the initial cost matrix will be returned as the best.

Algorithm 2 Greedy-Based Search

Input:
S is the training set
T is the test set
L is a classification algorithm
n is the number of iterations to run the algorithm

{0,1} is the set of classes
Let Eval(M, T) return an evaluation value on how Model M performed on test set T

Function GreedyCost(S, T, L, n)

Let I be the initial cost matrix where all punishments/rewards are 0
Let C be the current best cost matrix, initialized to I
Let M_C be the current best model, initialized to MetaCost(S, L, C)
Let O be the overall best cost matrix, initialized to I
Let M_O be the overall best model, initialized to M_C

for i = 1 to n **do**
  Let A be a set of cost matrices
  where
    $A_0 \leftarrow C + \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
    $A_1 \leftarrow C + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$
    $A_2 \leftarrow C + \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix}$
    $A_3 \leftarrow C + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
    $A_4 \leftarrow C + \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}$
    $A_5 \leftarrow C + \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix}$
    $A_6 \leftarrow C + \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$
  Set C and M_C to *null*
  for j = 0 to 6 **do**
    M = MetaCost(S, L, A_j)
    if Eval(M, T) > Eval(M_C, T) **then**
      $C = A_j$ and $M_C = M$
    end if
  end for

  if Eval(M_C, T) > Eval(M_O, T) **then**
    $O = C$ and $M_O = M_C$
  end if
end for

return O, M_O

4.2.2 Example Run

We demonstrate the effectiveness of our search method by showing how it can be used to assist in the classification of a simple simulated dataset. For the classification algorithm in this demonstration, we chose NaiveBayes as implemented by John and Langley [29]. This classifier uses estimator classes whose numeric estimator precision values are chosen based on analysis of the training data. NaiveBayes has been often shown to be an effective (if basic) classification technique. As our evaluation metric, we used the F-Measure, as detailed in our definitions section. Recall, the F-Measure ranges from 0 to 1, and the higher it is, the better performance a classification algorithm has. Additionally, in order to focus exclusively on specific instances that were misclassified, one dataset will be used for both training and testing of the classifiers. This way we can see what adjustments to the cost matrix affect specific data instances.

Basic Data

We created a two-dimensional square where the corners are (in x,y coordinates) (0,0) and (20,20). This square was then divided into two sections by the line x = 10, with all points that have x-values less than 10 belonging to class 0 and all points that have x-values greater than 10 belonging to class 1. With these restrictions in place, we randomly created 20 data points for class 0 and 20 data points for class 1. Important to note is that none of these data points had x-values of exactly 10.

Since this data is linearly separable, it should be rather simple for a classifier to learn the data. As expected, NaiveBayes classifies all data members correctly. It achieves an F-Measure of 1.

More Advanced Data

Next, we altered the dataset to make it more challenging for the classification method. We added 6 new datapoints that all have x-values of 10, but different y-values (which were chosen arbitrarily). In addition, three belong to class 0 [(10,1), (10,3), (10,4)] and three belong to class 1 [(10,2), (10,5), (10,6)].

After training and testing on this new dataset, the classifier had 3 misclassifications. These 3 misclassifications were the three additions to class 1 [(10,2), (10,5), (10,6)], which the classifier predicted belonged to class 0. In fact, the classifier determined that all data members on the x=10 line belong to class 0. The F-Measure for this classifier was 0.939.

Greedy Search

Since the classifier made some misclassifications on the more advanced data, we applied the MetaCost algorithm with our greedy-based search for an "optimal" cost matrix to see whether we could improve upon this classification. What follows are the detailed results of this search.

Initializations. To provide an adequate starting point, we must first create a base model to work from. For this we used an initial cost matrix in which no reward is given for correct classifications and both misclassification punishments are 1. As expected, the results are the same as the basic, unaltered classifier. So both the Overall Best Model (M_O) and Current Best Model (M_C) were then set to this new model.

Iteration 1. Using this initial cost matrix, we then created 7 new cost matrices as detailed in Algorithm 2. These were in turn used to create 7 new models, which were then compared based on their generated F-Measures. The best results were achieved through the cost matrix

$$C_1 = \begin{bmatrix} -1 & 2 \\ 2 & 0 \end{bmatrix}. \tag{4.2}$$

The F-Measure achieved through this cost matrix was 0.957. The generated model made only two misclassifications, one being the same as before (predicting that (10,2) has the class label of 0 instead of 1) and one new misclassification (predicting that (10,4) has the class label of 1 instead of 0). Both misclassifications are part of the advanced data, but the number of misclassifications has been lowered from 3 to 2.

Iteration 2. Using the cost matrix C_1 as a base cost matrix, we again created 7 new cost matrices and created 7 new models using MetaCost. The resulting models were then evaluated and the best F-Measure was achieved through the cost matrix

$$C_2 = \begin{bmatrix} -1 & 3 \\ 3 & 0 \end{bmatrix}. \tag{4.3}$$

This cost matrix achieved an F-Measure of 0.979 and made only one misclassification (predicting that (10,2) has the class label of 0 instead of 1). All other data members were classified correctly.

Iteration 3. Again 7 new cost matrices were created using the cost matrix C_2 as a base cost matrix, which in turn created 7 new models. After evaluation, the best F-Measure was achieved through

$$C_3 = \begin{bmatrix} -1 & 4 \\ 4 & 0 \end{bmatrix}. \tag{4.4}$$

However, the F-Measure is the same as the F-Measure of C_2 (0.979). So while the Current Best Cost Matrix (C) is updated, the Overall Best Cost Matrix (O) was not. Further iterations produced no F-Measure improvement.

Evaluation of Results

The best cost matrix found was C_2, which raised the F-Measure from 0.939 to 0.979 and decreased the amount of misclassifications from 3 to 1. Clearly, our method can improve upon a basic, unaltered classifier.

As mentioned in the explanation of the MetaCost algorithm, the algorithm relabels the data to better reflect a learning methodology more in-line with the given cost matrix. Therefore, it is highly likely that the misclassified data member was assigned the wrong class so that the final classification model could learn the other data members better, in essence sacrificing the correct classification of this data member. So while it may not achieve 100% on the training set, our approach does improve upon the initial classifier with an arguably acceptable loss.

4.3 Simulated Annealing

4.3.1 Process

Based on annealing in metallurgy in which a metal is cooled incrementally to achieve a minimum energy crystalline structure, simulated annealing has become a commonly used optimization method. Kirkpatrick et al. [32] and Cerny [4] independently applied this idea to optimization, showing it to be an effective method. Today many implementations exist for both single and multi-objective optimization [49].

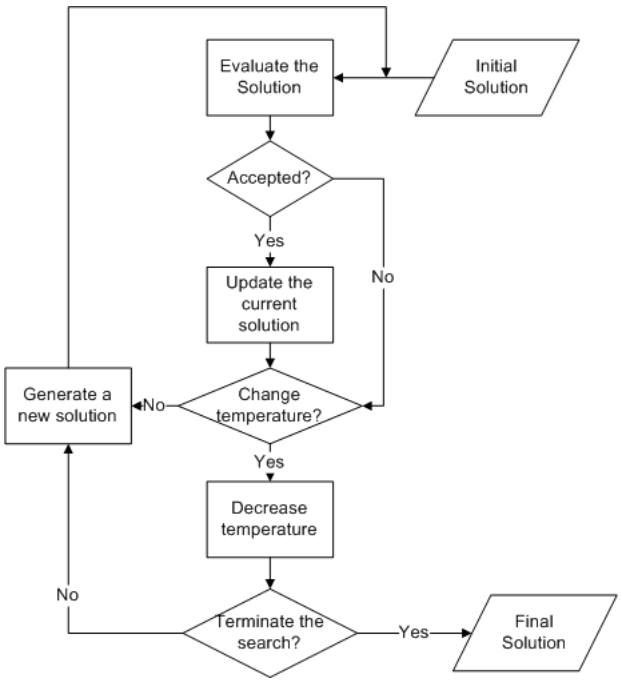

Figure 4.3: Simulated Annealing Flowchart

In Figure 4.3 we show a basic flowchart that is typical of most simulated annealing algorithms [40]. Starting with an initial solution, we evaluate the solution (in our case using the Eval function) and determine how well it performs, setting it as our current solution. Using this solution as a starting point, we create a new solution and evaluate how it performs. If we accept the new solution (based on either it performing better than the current solution or if it passes the replacement rule) then we update this solution to be the new current solution. We then repeat this process using the current solution.

Throughout this process we keep track of a temperature value which is altered according to a cooling schedule. Once the final temperature is reached, the search is terminated. This is advantageous as it provides a concrete stopping point for the algorithm.

There are many implementations for simulated annealing and many decisions need to be made as to how to implement one for a specific problem. We will tackle these in the following section.

- **Creating a Neighbor**
  Using the current solution (cost matrix), we need to create a new solution. This is no trivial task as this new solution needs to be "in the neighborhood" of the current solution and must also be capable of creating all possible solutions [49]. This criteria allows us to incrementally search the entire search space.

  In Algorithm 3, we can see the Neighbor function. This function takes in a cost matrix (typically the current best solution) and increments/decrements the various rewards/punishments with a random value in between the range of 0 and c. c is a constant integer that can be specified by the user. Important for this function is that true positive rewards ($C_{0,0}$) are always equal to or lower than 0 while the false positive ($C1, 0$) and false negative punishments ($C_{0,1}$) are kept equal to or higher than 0. This ensures that correct classifications are not punished and misclassifications are not rewarded, respectively. True negative rewards ($C_{1,1}$) are kept at 0.

Algorithm 3 Neighbor Function

Input:
X is a cost matrix (the current best solution)

Let c be a constant integer that represents the maximum punishment/reward increment
Let Rand(c) return a random integer between 0 and c

Function Neighbor(X)

Let Y be a cost matrix, initialized to X

r_1 = Rand(c)
if $Y_{0,0} + r_1 > 0$ **then**
  $Y_{0,0} = Y_{0,0} - r_1$
else
  $Y_{0,0} = Y_{0,0} + r_1$
end if

r_2 = Rand(c)
if $Y_{0,1}$ - $r_2 < 0$ **then**
  $Y_{0,1} = Y_{0,1} + r_2$
else
  $Y_{0,1} = Y_{0,1} - r_2$
end if

r_3 = Rand(c)
if $Y_{1,0}$ - $r_3 < 0$ **then**
  $Y_{1,0} = Y_{1,0} + r_3$
else
  $Y_{1,0} = Y_{1,0} - r_3$
end if

return Y

- **Replacement Rule**

  There are two ways a new solution is accepted in a typical simulated annealing algorithm. One is when the evaluation function returns a higher value than the value generated with the current solution. In this instance, the current solution is simply replaced with this new, better performing solution.

  If however, the new solution does not perform better than the current best, it still has a chance to replace it. At this point, the algorithm determines a probability value that can be seen in the following equation:

$$P = e^{-\Delta E/K} \tag{4.5}$$

  This is the probability with which the new solution will replace the current best solution. ΔE is the difference between the evaluation value of the current best solution and the evaluation value of the new solution. K is the current temperature value. So the likelihood that the new solution will replace the current best solution is increased if the difference is very low and/or the temperature value is very high. This is done to avoid local minima. At high temperatures, the likelihood of replacement is high. But as this value is lowered (cooled) it becomes less and less likely.

- **Cooling Schedule**

  One of the most important decisions in implementing a simulated annealing algorithm is what type of cooling schedule to have. While many schedules exist [49], we chose the geometric cooling rule as seen in the following equation:

$$K_{i+1} = g * K \tag{4.6}$$

  In this equation, the constant g is a value that is close to but less than 1 (it can vary between 0.8 and 0.99). So with each update, the temperature K is gradually lowered. This update occurs after a set number of new solutions have been created.

- **Termination**

  One of the benefits of simulated annealing is that a finite termination point exists (unlike with our greedy search and the genetic algorithm). The algorithm keeps track of two temperature values, a current temperature and a final temperature. When the current temperature is equal to the final temperature, the algorithm stops and the global best solution is returned to the user.

Algorithm 4 Simulated Annealing Cost Matrix Search

Input:
S is the training set
T is the test set
L is a classification algorithm
K_I is the initial temperature
K_F is the final temperature
n is the number if iterations per temperature value

Let Eval(M, T) return an evaluation value E on how Model M performed on test set T
Let Neighbor(X) return a cost matrix Y that is a neighbor of cost matrix X
Let Rand() return a (pseudo)random number between 0 and 1

Function SimAnnealCost(S, T, L, K_I, K_F, n**)**

Let I be the initial cost matrix
Let C be the current best cost matrix, initialized to I
Let M_C be the current best model, initialized to MetaCost(S, L, C)
Let O be the overall best cost matrix, initialized to I
Let M_O be the overall best model, initialized to M_C
Let K_C be the current temperature, initialized to K_I

while $K_C > K_F$ **do**
  for $i = 1$ to n **do**
    Y = Neighbor(C)
    M_Y = MetaCost(S, L, Y)

    if (Eval(M_Y, T) > Eval(M_C, T)) **then**
      $C = Y$ and $M_C = M_Y$
    else
      Let ΔE = Eval(M_C, T) - Eval(M_Y, T)
      if Rand() $\leq e^{-\Delta E/K}$ **then**
        $C = Y$ and $M_C = M_Y$
      end if
    end if

    if Eval(M_C, T) > Eval(M_O, T) **then**
      $O = C$ and $M_O = M_C$
    end if
  end for
  $K_C = g * K_C$ where $g < 1$
end while

return O, M_O

In Algorithm 4, we can see our implementation of the simulated annealing search for a best cost matrix. As with the greedy approach, it takes in a training set S, a classification algorithm L, and a test set T for evaluation purposes. Unique to this search are the need for an initial K_I and final K_F temperature and a number for the amount of iterations n to be performed per temperature. The temperature values are generally kept low with the defaults being 2 and 0.1 for the initial and final temperature, respectively. In addition to the input we specify three functions: the aforementioned Eval function, the Neighbor function (seen in Algorithm 3), and a function named Rand() that returns a pseudo-random floating point value between 0 and 1.

In lines 15-20 we can see the first part of the algorithm, namely the initializations. These are the same as the greedy based search, with the addition of initializing the current temperature K_C. As in the greedy search, we keep track of two cost matrices and their corresponding models to ensure that at worst we perform as well as an unaltered classifier.

In line 22 the actual search begins in the form of a nested loop. The outer loop is dependent on the temperature; it will continue iterating until the current temperature K_C is no longer higher than the final temperature K_F. The inner loop will iterate n times. After these iterations are completed, the temperature is lowered.

Inside these loops is the search for a better cost matrix through the creation of neighbor cost matrices. The current best cost matrix C is passed into the Neighbor function and the generated cost matrix Y is used to create a new model. If this model produces a better evaluation value than the current best model, it then replaces the current best. If not, then the replacement probability function shown in Equation 4.5 is used. If the generated random value is less than or equal to this probability value, then the Y and M_Y replace C and M_C, respectively. Finally, the current best model is compared to the overall best M_O, replacing it only if the evaluation value is higher.

After the final temperature is reached, the overall best cost matrix O and model M_O are returned to the user.

4.4 Genetic Algorithm

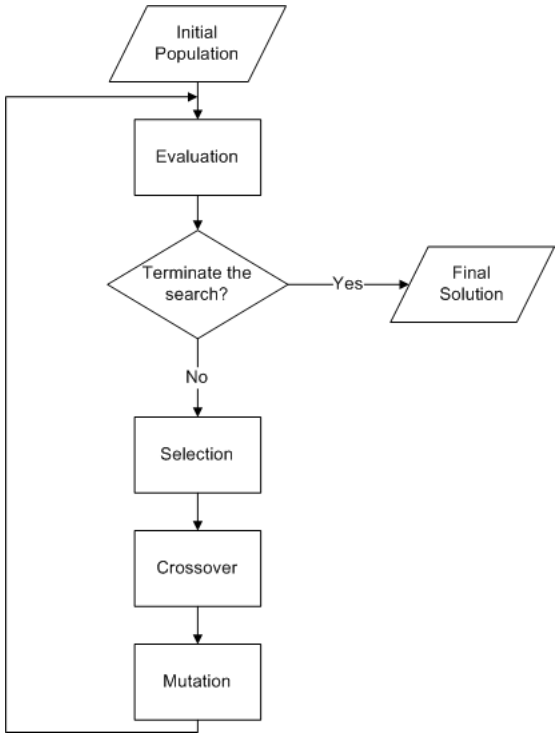

Figure 4.4: Genetic Algorithm Flowchart

4.4.1 Process

The basic idea (as seen in Figure 4.4) behind a genetic algorithm is to use the concepts of natural selection to find a best solution for a problem. The idea was popularized in a book by Holland [24], who came up with the name "genetic algorithm", even though research on the topic had existed for a while. Using a pool of potential solutions, the algorithm will "evolve" these solutions to find a best one, using the genetic concepts of recombination, mutation, and survival of the fittest. While many implementations exist [44], with differing terminology, there are common aspects to all that we shall address in this section, as well as our decisions and solutions to solving the search for a cost matrix problem.

- **Representation**
  Representation deals with how these potential solutions are internalized within the algorithm. Typically we deal with two concepts: genotype and phenotype. A genotype is usually a way to basically state a solution, for instance as a collection of integers. The phenotype on the other hand can be something more abstract and is linked in

some way to this genotype (an implementation of these integers). For instance, in our problem the genotype is the cost matrix and the phenotype is the classification model that is created with this cost matrix.

In keeping with the biological roots of genetic algorithms, a genotype can also be referred to as a chromosome which is composed of genes. A chromosome is equivalent to an array and the genes are the individual positions in the array. The values of the genes are referred to as alleles. Applying these definitions to our problem, a chromosome would be equivalent to a cost matrix. The main change that would have to be made, is that this 2x2 matrix needs to be represented as a one- dimensional array/set. This can be seen in Equation 4.7. Here all punishments/rewards are represented as genes and their assigned integer values are their alleles.

$$C - Chromosome = \{C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}\} \tag{4.7}$$

- **Evaluation/Fitness Function**
  The main concept of natural selection is the survival of the fittest. In nature, this can be determined by a variety of factors with the end result that the fittest species is the one that survives based on its ability to, for instance, adapt to the environment or evade predators. For our solution space, we need to create a way to determine which solution is the best, i.e. the fittest. This is done through a fitness function, a function that a genotype/phenotype can be plugged into and it will return an evaluation of its fitness. Specifically, we need an evaluation of how well the cost matrix does in creating a classification model.

  To do this, we use the previously declared Eval function. The Eval function takes in a generated classification model (that we have generated using MetaCost with the given classification algorithm, the given training data, and the current cost matrix as input) and a test dataset and returns an evaluation measure on how it performed. This measure consists of a value between 0 and 1, with 1 being optimal performance. Since the focus is on rare-class data, this evaluation metric is one that is usually associated with rare-class classifiers, such as the F-Measure (a harmonic mean between precision and recall) or the AUC (area under the ROC curve).

- **Population**
  The population is the set of all possible solutions that are currently being evaluated. Typically, the size is kept constant with new, better solutions replacing old, poor solutions. Of importance here is to decide how large a pool of solutions to create with larger populations taking longer to evaluate and smaller populations potentially less likely to find a best solution.

  A variety of thought exists on this problem and determining the optimal population size depends on many factors such as the length of the chromosomes as well as what variation operators are being used. For instance, if the variation operators do not

generate any new alleles, then we must have a large enough population to encompass all possible alleles for each gene. This can lead to very large populations and can prove impossible with continuous gene values.

Further thought must also be given to what the initial population will be composed of. A common approach is to create a set of random chromosomes, initialized to random alleles. In our case, random integers would be selected for the punishments/rewards in the cost matrix. Heuristics can also be used to create a more focused initial population. For instance, typically the punishment for false negatives is higher than the false positive punishments in the found best cost matrices. So we could include a restriction that all generated cost matrices must have false negative punishments that are higher than their false positive punishments. However, it is debatable if such heuristics are necessary because presumably our methodology would filter out poor solutions after a few iterations anyway. At best, such heuristics would then be a time-saving technique and reduce the amount of iterations.

Another idea is to seed the population with known good solutions. This concept, sometimes referred to as "inoculation", can be useful in that it will already have good solutions from which to create new solutions. In our situation, we could seed the population using an initial cost matrix, to make sure that the most basic case is included in the search and therefore improved upon. However, the benefits are debatable and recent research has shown that seeding tends to reduce the quality of the best solution [52].

- **Selection**

  One of the most important tenets of evolution is the concept of selection. While many selection methods exist for genetic algorithms, they typically all rely to some degree on the fitness value. The idea is to eliminate the poor solutions and to keep the good solutions. Selection is broken down into two basic types: parent selection and survivor selection. In parent selection, one or more chromosomes are selected to generate new solutions, also called offspring. In survivor selection, those offspring with poor performance are weeded out. Typically, both selection types are used but parent selection can be replaced with a random choice with much success.

  The most basic selection method is to rank the solutions based on fitness value and to take the highest ranked solutions. While this makes sense as the basic idea is to generate good solutions, it does eliminate the possibility that a better solution can be achieved by combining a good solution with a poor solution. This is addressed in the commonly used "roulette-wheel" method. In this method, choosing which solutions to use and/or keep can be equated to spinning a roulette wheel. However, the choice of solution is not completely random as those solutions with higher fitness values have a larger surface on the wheel. This way, better solutions are more likely to be chosen but worse solutions also have a chance of being selected. Additional methods include tournament selection (in which a subset is evaluated to determine the best solution,

which is then chosen as parent) and survivor selection methods which take into account the age of solutions (the older the solution, the more likely it is to be eliminated).

- **Variation Operators**
  After the parents have been selected, operators need to be defined in which new solutions (offspring) are created based on these parent solutions. Typically they fall into two types: mutation and recombination. Mutation takes in a single solution (genotype) and creates an offspring by randomly changing one or more parts of the genotype based on a mutation probability. In our example, this would involve taking in a cost matrix and randomly incrementing/decrementing one or more of the punishments/rewards. Recombination involves taking two solutions (genotypes) and combining them to create an offspring that contains aspects of both. There are many ways to do this and oftentimes selecting the right one is based on the current problem and what the specific genotype consists of. Since our genotype consists of set of integers (a cost matrix), we can use a method like cross-over, in which we take values from both cost matrices and create a new one.

  As mentioned in the discussion of the population, it is possible to only use recombination such as cross-over as our variation operator. The stipulation for this is that representatives of every possible allele value must then be part of the initial population in order to have a complete search. For instance, if we set the stipulation that punishments/rewards can be between 0 and 100 for every cost in the cost matrix, we would need a population size of at least 404 to include all possible allele values. But if we combine cross-over with mutation, we can start with a smaller population size, with the confidence that new alleles will be introduced into the population through mutation.

- **Termination Condition**
  With each iteration of the genetic algorithm, new solutions are created and if our variation operators, fitness functions, and selection methods are adequate, better solutions are being created. So how do we know if we have created enough new solutions? In some situations, a perfect fitness value is defined and the generation of offspring can be halted. However, in situations such as ours, a perfect fitness value is unknown or at least most likely unattainable (it would be unrealistic to expect a cost matrix to create a classification model that perfectly classifies all test data).

  There are various thoughts on how to create a termination condition. For one, an absolute number of generations can be created or the algorithm can be halted after a threshold value for the fitness values has been reached. Another idea is to stop when no more improvement occurs in the creation of a new generation. Related to this is the concept of minimal population diversity. If the population reaches a set where the solutions are highly similar (in terms of phenotypes or fitness values) one can presume that the algorithm has converged on a best solution. More thoughts on the termination condition will be addressed after discussing the algorithm implementation.

Algorithm 5 Genetic Algorithm Cost Matrix Search

Input:
S is the training set
T is the test set
L is a classification algorithm
n is the number of iterations to run the algorithm
s is the size of the solution population

Let Eval(M, T) return an evaluation value E on how Model M performed on test set T
Let Cross(X_1,X_2) return two new cost matrices by crossing over cost matrices X_1 and X_2
Let Mutate(X) return a cost matrix that has been mutated based on a determined mutation rate

Function GeneticCost(S, T, L, n, s**)**

Let I be the initial cost matrix
Let C be the current best cost matrix, initialized to I
Let M_C be the current best model, initialized to MetaCost(S, L, C)
Let O be the overall best cost matrix, initialized to I
Let M_O be the overall best model, initialized to M_C

Create population of random Cost Matrices P with size s
for all X in P **do**
  Model M = MetaCost(S, L, X)
  E = Eval(M, T)
  Assign E to X
end for

for i = 1 to n **do**
  Create new population of offspring Cost Matrices F using Cross(X_1,X_2) and Mutate(X)
  for all X in F **do**
    Model M = MetaCost(S, L, X)
    E = Eval(M, T)
    Assign E to X
  end for
  Replace Cost Matrices in P with low E values with Cost Matrices in F with high E values

  Set C to the Cost Matrix in P with the highest E value
  Set M_C to the model generated by MetaCost(S, L, C)

  if Eval(M_C, T) > Eval(M_O, T) **then**
    $O = C$ and $M_O = M_C$
  end if
end for

return O, M_O

In Algorithm 5, we show the pseudocode for the genetic algorithm search for a cost matrix. The initial input is the same as the greedy-based search with the one exception that a population size s must be passed into the algorithm. Additional input parameters include both training and test data (which must be of the same type and ideally consist of different examples), a classification algorithm (this can be of any type such as a decision-tree learner, rule-based learner or SVM), and the number of iterations to run the algorithm. We also define three functions that facilitate the GeneticCost algorithm. The first is the Eval function which returns an evaluation value on how well the given model predicts the given test data. The other two are variation operators: Recomb, which takes in two cost matrices and creates two new ones through recombination (cross-over), and Mutate, which takes in one cost matrix and mutates it by changing the values of it.

In lines 14-19 we see the initializations at the beginning of the algorithm. We start off with an initial cost matrix and train a model using it. Using the initial cost matrix is equivalent to training a classification algorithm unaltered, i.e., without MetaCost. Both the current best cost matrix and the overall best cost matrix are initialized to this initial cost matrix and their corresponding models are initialized to the resulting model. This is done so that *at worst* the resulting model will perform as well as an unaltered classifier. A further initialization occurs in lines 20-25, where we initialize the population of cost matrices. Using the initial cost matrix as a template we create a total of s random cost matrices. We then use these cost matrices to create a series of models and in turn evaluate these models. The evaluation values are then assigned to each cost matrix in the population.

In line 27 the search part of the algorithm starts. The first step, implemented in lines 28-34, is to evolve the population of solutions (cost matrices). To do this, we create a population of offspring. These offspring are created through the functions Cross and Mutate which cross two random cost matrices and mutate cost matrices based on a mutation rate, respectively. After these offspring cost matrices have been created, they are each in turn used to create classification models and evaluated on the test data. The resulting E values are then assigned to the cost matrices that generated them. Those cost matrices with higher E values are used to replace the cost matrices with lower E values in the population P, keeping the size of this population constant.

After this evolution and natural selection step have completed, the cost matrix with the highest E value is set to be the current best cost matrix and its resulting model is then set to the current best model. This is then compared to the overall best cost matrix, replacing it and its model if the E value is better. The main reason for this step is to only keep track of the best solution if it actually improves on an unaltered classification model.

After the number of iterations has been reached, the algorithm returns the overall best cost matrix and the overall best model.

4.4.2 Termination

Our algorithm allows for a variety of termination conditions to be employed by the user. The first one is simple, the algorithm can be halted after a set number of iterations n. Due to the fact that the genetic algorithm should be consistently eliminating bad solutions while generating good new ones, it is unlikely that an iteration number will be reached in which the algorithm starts to generate worse solutions. At worst, the algorithm will reach a point where no improvement can be reached any more and subsequent iterations are superfluous.

Based on this, it would be prudent to set as a termination condition whether improvement has been reached after the last iteration. Since we are keeping track of the overall best classifier and the current best classifier, we can refer to improvement having occurred if the current best classifier is the same as the overall best classifier. However, since we are using a genetic algorithm that starts with random chromosomes (cost matrices), it might take a few iterations before improvement can be seen.

Another direction we can take termination in is to have a fitness value threshold. Applied to a specific case, termination can occur when a certain F-Measure value is reached. The difficulty however lies with determining this threshold. Ideally, we would want perfect classification, equivalent to an F-Measure of 1. However, this is an unrealistic goal to have and would most likely result in an infinite loop. So we would have to choose an F-Measure carefully based on the expectations for the particular dataset and classification algorithm.

A final type of termination condition would be an indication of how diverse the solution set (population) is. This could be done after each iteration by extracting all fitness values and seeing how diverse they are through establishing a standard deviation or variance value for the set or by simply seeing the difference between the highest and the lowest fitness value. After a certain threshold is reached, the algorithm can terminate. The benefit of using an approach like this is that termination can be reached when the pool of solutions seems to "converge" towards an optimal one.

The default method it to run the algorithm for 20 iterations.

Chapter 5

Implementation

This chapter is a description of the software that implements the various cost matrix optimization algorithms as well as the underlying classification algorithms. It presents the class layout, the underlying architectures, and sample usage.

The preceding pseudocode was all implemented in Java. Java was chosen for two main reasons:

1. It allows for the flexibility and clarity of object-oriented programming to be utilized in the implementation of the presented algorithms.
2. It allows for us to tie into other Java-based software packages such as weka and JGAP.

Each of the cost matrix optimization algorithms was implemented as their own java class: the greedy-based search is the class **GreedyCost**, the simulated annealing search is the class **SimAnnealCost**, and the genetic algorithm search is the class **GeneticCost**. Classification algorithms were implemented with the help of the weka package, which is an open-source suite of java classes that implement a large number of commonly used classification algorithms [57]. These algorithms range in complexity from basic stump trees to SVMs (Support Vector Machines). In addition, the suite contain other classes that facilitate the classification process such as cost matrix classes and classes for the handling of input data. The genetic algorithm was implemented with the help of JGAP (Java Genetic Algorithms Package) [16]. JGAP is a suite of programs that implement most commonly used genetic algorithms for optimization and other uses. The following sections will detail the classes that were created (and ultimately utilized) for this project.

5.1 Basic Components

These classes represent the basic components in the cost matrix search. Many of them facilitate the use of weka classes, simplifying their implementations and adjusting them to better suit our methods.

- **MyCostMatrix**
  This class serves as a wrapper for the weka **CostMatrix** class. It ensures that the cost matrix is a 2x2 matrix and provides easier methods to adjust the punishments and rewards. In addition it provides methods to create pseudo-random cost matrices and to "scalar jump" a cost matrix, in which all values in the cost matrix are multiplied by a given scalar value.

- **MyMetaClassifier**
  MyMetaClassifier is the interface for a meta-classification method that uses a cost matrix to adjust the learning of an underlying classification algorithm. It is implemented through two classes: **MyMetaCost**, which implements the weka **MetaCost** class, and **MyMetaCostAdaBoost**, which implements an altered form of the weka **MetaCost** class that uses boosting instead of bagging to determine class probabilities.

- **MyClassifier**
  This class is the essential class for classification. It consists of three main objects: a **MyCostMatrix** object, a **MyMetaClassifier** object, and a weka **Classifier** object, that implements virtually any classification algorithm in weka. The MyClassifier object then uses the given cost matrix, with the given meta-classifier to train and subsequently test the given classification algorithm. It also allows for the classification algorithm to be used without the meta-classifier, to provide base results on how an unaltered classifier performs. After testing, the MyClassifier class provides methods to obtain various statistics, such as F-Measure, AUC, G-Mean and the overall confusion matrix.

- **InputData**
  This class implements a method for uploading and using input data for training/testing. Given a source file, the class will automatically read in and store each data member for classification. Additional methods allow for the split into training and test data. The class also contain methods for oversampling and undersampling in which a percentage number is given and the data is expanded/shrunk to correspond to this value.

- **Results**
  This class consists of an **InputData** object and an array of classification values that were assigned to it by a classification model. It is returned after a **MyClassifier** object has been tested on a set of test data. This allows for a detailed look at what data members a classifier correctly/incorrectly classified.

- Additional Classes
  Additional classes consist of a class that automatically logs the search for a cost matrix, a class that prepares a dataset for multiple training/test splits, and a class that automates the running of experiments.

5.2 Search Classes

From a user-based perspective, all the search classes can be viewed as having highly similar functionality. Given a set of training/test data and a classification algorithm, the search method will return a classification model with the best found cost matrix. This is represented by a **MyClassifier** object. Due to this conformity, all search methods can be used in an identical manner. We still however, allow for methods to be used that take into account the specific benefits of each search method.

5.2.1 Common Methods

These are the methods common to the classes **GreedyCost**, **SimAnnealCost**, and **GeneticCost**.

- Constructor
  Three main things are passed into the search method for instantiation: a **MyClassifier** object, which consists of an initial cost matrix, and both training and test data, which are both **InputData** objects and can be the same.

- run()
  The run method instantiates one run of the search algorithm. Variations of this include passing in an iteration value (much like the n value in the shown algorithms), running the search algorithm until a certain minimum metric value is reached, and running the algorithm until a certain number of classification models have been created.

- improve()
  The improve method gives a boolean value on whether the last run of the search algorithm garnered improvement. Improvement is reached anytime the overall best model M_O is replaced. So if the overall best model is equal to the current best, the last run resulted in improvement and a "true" value is returned.

- setMetric()
  This method allows one to set the evaluation metric for the cost matrix search. Supported metrics are: F-Measure (which is the default), AUC (the Area Under the ROC Curve), and G-Mean.

- getCurrentBest(), getOverallBest()
  These methods return the current best and overall best **MyClassifier** objects. This gives the user then the ability to access and use both the best found cost matrix and the best generated classification model.

5.2.2 Unique Methods

While the conformity in the search methods allows for a certain amount of ease of use, it is also important to allow the user to take advantage of the specific aspects of each method.

- **GreedyCost**
  The greedy-based method is pretty straight forward. As such, it can be used with only the common methods and the user will be taking advantage of virtually all of its functionality.
- **SimAnnealCost**
  To fully utilize the benefits of simulated annealing, adjustment methods exist for both the current and final temperature as well as a method to determine if the algorithm has "cooled down" i.e. reached the point where the current temperature is equal to the final temperature. This also allows for a function called runComplete() in which the run method is invoked until the algorithm has "cooled down." A method also exists to alter the adjust range in the neighbor function (the default is 20).
- **GeneticCost**
  The main method that can be invoked is part of the constructor: the user can specify the size of the sample population (default is 30). Additional adjustments can be done to the maximum punishment/reward for the cost matrix (default 100).

5.3 Sample Implementation

5.3.1 GreedyCost

Algorithm 6 Sample GreedyCost Implementation

```
InputData training = new InputData(``training_data.txt''); // initialize training data
InputData test = new InputData(``test_data.txt''); // initialize test data

MyCostMatrix cm = new MyCostMatrix(); // create default, initial cost matrix
MyMetaClassifier meta = new MyMetaCost(); // create default meta-classifier

// create a NaiveBayes classifier using the cost matrix and the meta-classifier, with no new options (null)
MyClassifier classifier = new MyClassifier(``NaiveBayes'', cm, meta, null);

// create a GreedyCost object
GreedyCost greedy = new GreedyCost(classifier, training, test);

// run the search until improvement stops
while (greedy.improve()) {
  greedy.run();
}
// run an additional 5 iterations
greedy.run(5);

MyClassifier optimal = greedy.getOverallBestClassifier(); // get the final best classifier
```

5.3.2 SimAnnealCost

Algorithm 7 Sample SimAnnealCost Implementation

```
InputData training = new InputData(``training_data.txt'');  // initialize training data
InputData test = new InputData(``test_data.txt'');  // initialize test data

MyCostMatrix cm = new MyCostMatrix();  // create default, initial cost matrix
MyMetaClassifier meta = new MyMetaCost();  // create default meta-classifier

// create a JRip classifier using the cost matrix and the meta-classifier, with no new options (null)
MyClassifier classifier = new MyClassifier(``JRip'', cm, meta, null);

// create a SimAnnealCost object
SimAnnealCost simAnneal = new SimAnnealCost(classifier, training, test);

simAnneal.setCurrentTemp(2);  // set current/initial temperature to 2
simAnneal.setFinalTemp(0.1);  // set final temperature to 0.1
simAnneal.setMetric(SimAnnealCost.AUC);  // set metric to be AUC

// run the complete search
simAnneal.runComplete()

MyClassifier optimal = simAnneal.getOverallBestClassifier();  // get the final best classifier
```

5.3.3 GeneticCost

Algorithm 8 Sample GeneticCost Implementation

```
InputData training = new InputData(``training_data.txt'');  // initialize training data
InputData test = new InputData(``test_data.txt'');  // initialize test data

MyCostMatrix cm = new MyCostMatrix();  // create default, initial cost matrix
MyMetaClassifier meta = new MyMetaCost();  // create default meta-classifier

// create a PART classifier using the cost matrix and the meta-classifier, with no new options (null)
MyClassifier classifier = new MyClassifier(``PART'', cm, meta, null);

// create a GeneticCost object with population size of 30 and using the G-mean metric
GeneticCost genetic = new GeneticCost(classifier, training, test, 30,
GeneticCost.GMean);

// run the search until 500 models have been created
genetic.runModelCount(500);

MyClassifier optimal = genetic.getOverallBestClassifier();  // get the final best classifier
```

Chapter 6

Experimental Results and Discussion

In this chapter, we focus on the results we achieved with our detailed methods on five different datasets. In addition to presenting these results, we also detail what these results mean, giving further insight into the rare-class classification process.

6.1 Gene Conversion Simulation

As part of our experiments with rare-class data, we initially developed the greedy-based approach while creating a classification method for gene conversions. The following section details those preliminary results [34].

6.1.1 Predicting Gene Conversions

Due to the significant role that gene conversion plays in both short-term (genetic diversity and diseases) and long-term (gene conservation) evolution, predicting and identifying gene conversions in sequences is an important task in molecular evolution [9]. However, our recent study shows that the handful of gene conversion prediction programs that exist exhibit poor performance [35]. Because gene conversion only occurs between two sequences in a multiple sequence alignment, it falls into the realm of rare-class prediction problems. Therefore, we can apply our method of searching for an optimal cost matrix to improve the performance of gene conversion prediction programs.

For our experiments, we used two gene conversion prediction programs, GENECONV [47] and Partimatrix [27]. GENECONV was developed specifically for the identification of gene conversions. Given an aligned file of DNA sequences, it will give a prediction of what sequence fragments have the highest, unique similarity between two sequences. It then presents p-values that show the significance of these paired sequence fragments. These p-values are

determined both globally and pairwise. Global p-values are determined by comparing each fragment with all other possible fragments, thus establishing the overall uniqueness of the paired fragments. Pairwise compares each fragment with the maximum score that would be expected had gene conversion not occurred. Within both global and pairwise, two p-values are calculated. The first, "SIM," is based on permutations, whereas the second is based on a method developed by Karlin and Altschul [31], "KA."

Partimatrix was designed to identify cases of reticulate evolution, i.e., cases of gene conversion and recombination, and analyze anomalous phylogenetic history in a multiple sequence alignment. Specifically, in a multiple sequence alignment, a column is *parsimoniously informative* if it contains at least two different nucleotides, each occurring in at least two sequences; this method considers only parsimoniously informative columns. Any phylogenetic tree on the s nucleotides in a column specifies $s - 1$ bipartitions of the sequences the nucleotides are from, one bipartition for each edge of the tree. For a column containing exactly two different nucleotides, the column *supports* the bipartition of the sequences given by the nucleotide bipartition. For a column containing two or more different nucleotides, the column is *consistent* with any bipartition that is found in at least one most parsimonious tree for those nucleotides. A column *conflicts* with a bipartition if it is not consistent with the bipartition. Any bipartition that is supported by at least one column has a *support score* equal to the number of columns that support it and a *conflict score* equal to the number of columns that conflict with it. Those bipartitions that have a high support score can be analyzed by various statistical means. If the support score is statistically higher (or the conflict score is statistically lower) in one region of the alignment compared to another, then there is evidence for recombination. If a number of bipartitions provide evidence for recombination, then the partition matrix approach infers that a recombination event is necessary to explain the alignment.

Our analysis on simulated gene data shows that both GENECONV and Partimatrix perform poorly under certain conditions in predicting gene conversions. However, they do not misclassify in the same manner. Specifically, GENECONV appears to be better in predicting more recent gene conversion events whereas Partimatrix is better in correctly predicting "older" gene conversions events. So our research was focused on combining their predictions and thus achieving a better ensemble classifier.

6.1.2 Gene Conversion Data and Classification

Because actual gene conversion data is difficult to obtain, we created simulated gene conversion data similar to an approach developed by Marais [36]. Essentially we simulated the creation of a gene family from a root sequence (through mutation along a simulated phylogenetic tree) and inserted a gene conversion event between two of the genes. This way we could create recent gene conversion events (by having mutations take place mostly before the gene conversion event) and more ancient gene conversion events (by having more mutations occur

after the event). We then created two sets of data: SET1 which consisted of multiple recent gene conversions and SET2 which consisted of multiple ancient gene conversions. Each of these datasets consisted of multiple gene families (consisting of six genes each) and one (or no) gene conversion event.

For classification, we represented each pair of genes within a gene family through a feature vector. In this representation, we can see that gene conversion is a rare-class data problem. A set of six genes represents 15 gene pair combinations and at most one of these gene pairs will have a gene conversion event. Each of these feature vectors consists of the following attributes: average GC content, overall sequence similarity, GENECONV prediction global and pairwise p-values, and Partimatrix conflict and support scores.

Classification was done through the greedy-based search for a cost matrix that we detailed in the methods section. We used the following classification algorithms as the underlying classifiers: NaiveBayes (as implemented by John and Langley [29]), J4.8 (an implementation of the C4.5 decision tree learner [42]), PART (a combination of rule-based learning and C4.5 [21]), and JRip (a rule-based learner based on RIPPER [10]). All classification algorithms were implemented in weka [57], a collection of machine learning algorithms.

6.1.3 Results

Table 6.1: Gene Conversion Simulation SET1 Classification Results

Classifier	TP	FP	TN	FN	Accuracy	Recall	Precision	F-measure
Perfect	139	0	2051	0	1	1	1	1
Just Say No	0	0	2051	139	0.937	0	UNDEF	UNDEF
GENECONV Strict	102	4	1448	35	0.975	0.745	0.962	0.840
GENECONV LP	123	57	1395	14	0.955	0.898	0.683	0.776
Partimatrix	9	137	1315	128	0.833	0.066	0.062	0.064
G-or-P	128	191	1261	9	0.874	0.934	0.401	0.561
NaiveBayes	122	58	1394	15	0.954	0.891	0.678	0.770
PART	107	5	1447	30	0.978	0.781	0.955	0.859
J4.8	109	11	1441	28	0.975	0.796	0.908	0.848
JRip	111	9	1443	26	0.978	0.810	0.925	0.864

This table represents the performance of the various classification methods on dataset SET1. The upper half represents the basic classifiers that do not use the greedy-based approach. *Perfect* represents a theoretical optimal classifier and is included for comparison. *Just Say No* represents a classifier that classifies all data elements as majority class. *GENECONV Strict* uses only global p-values for predictions, whereas *GENECONV LP* uses local pairwise p-values (with 0.05 being used as the threshold for positive classification). *Partimatrix* represents a prediction based on the lowest conflict score between a gene pair within a gene family. *G-or-P* is a basic unification of *GENECONV LP* and *Partimatrix* predictions. The lower half represents the classification algorithms predictions after using the greedy-based search for a cost matrix.

In Tables 6.1 and 6.2, we can see the classification results. For our purposes the positive class

Table 6.2: Gene Conversion Simulation SET2 Classification Results

Classifier	TP	FP	TN	FN	Accuracy	Recall	Precision	F-measure
Perfect	150	0	2250	0	1	1	1	1
Just Say No	0	0	2100	150	0.933	0	UNDEF	UNDEF
GENECONV Strict	1	8	2092	149	0.930	0.007	0.111	0.014
GENECONV LP	5	68	2032	145	0.905	0.033	0.068	0.045
Partimatrix	15	135	1965	135	0.880	0.100	0.100	0.100
G-or-P	19	197	1903	131	0.854	0.127	0.088	0.104
NaiveBayes	8	75	2025	142	0.904	0.053	0.096	0.069
PART	35	214	1886	115	0.854	0.233	0.141	0.175
J4.8	23	160	1940	127	0.872	0.153	0.126	0.138
JRip	40	265	1835	110	0.833	0.267	0.131	0.176

This table represents the performance of the various classification methods on dataset SET2. The upper half represents the basic classifiers that do not use the greedy-based approach. *Perfect* represents a theoretical optimal classifier and is included for comparison. *Just Say No* represents a classifier that classifies all data elements as majority class. *GENECONV Strict* uses only global p-values for predictions, whereas *GENECONV LP* uses local pairwise p-values (with 0.05 being used as the threshold for positive classification). *Partimatrix* represents a prediction based on the lowest conflict score between a gene pair within a gene family. *G-or-P* is a basic unification of *GENECONV LP* and *Partimatrix* predictions. The lower half represents the classification algorithms predictions after using the greedy-based search for a cost matrix.

is when a gene pair has a gene conversion and the negative class is when it does not. Not shown are the results of unaltered classifiers as they were unable to make any true positive classifications due to the class imbalance. Their results were equivalent to the *Just Say No* classifier. For the learning of each set, we created separate training and test data and then evaluated the final model on a second set of unique test data.

In SET1, one can see that GENECONV performs quite well. "GENECONV Strict" has a high accuracy, even higher than the "Just Say No approach". However, through our method we are able to increase the amount of true positives, increase the accuracy, and most importantly, increase the F-measure. The best performers are JRip and PART, which is not surprising as they are rule-based classifiers and rule-based classifiers are known to perform well on rare-class data [53]. Both have a higher F-measure than "GENECONV Strict", a higher accuracy, and both identify more true positives. J4.8 does well too and identifies more true positives than PART, but more false positives as well. Of all the cost matrix classifiers, NaiveBayes identifies the most true positives, but is hindered by the number of false positives it identifies.

In SET2, one can see that ancient gene conversions are far more difficult to accurately detect, as the mutations after the conversion makes some difficult to differentiate. GENECONV performs quite poorly, both in Strict and LP. Partimatrix identifies more gene conversions and G-or-P has the best F-measure of these basic classifiers. This set also shows the shortcoming of using accuracy as a metric as the "Just Say No" approach would appear to be the best classifier. Among the cost matrix classifiers, the NaiveBayes classifier performs quite

poorly. It has an F-measure lower than G-or-P, so it shows no improvement over a basic classifier (it does not identify more gene conversions correctly either). But the rule-based classifiers again perform quite well, with both identifying more gene conversions and having higher F-measures than any of the basic classifiers. In fact, aside from NaiveBayes, all classifiers exhibit both a higher recall and a higher precision than the basic classifiers, showing a definite improvement.

6.2 UCI Data

To further test our approach we used four known biological datasets from the UCI data repository [39]. Since we are focusing on rare-class datasets, we were limited in which ones we could select. In that regard, the PIMA dataset seemed to be the only binary dataset that was both rare-class and biological. However, a multi-class dataset can easily be made to be rare-class when a "one-against-rest" approach is used. In this approach a classifier is created for each class and this classifier determines if a data member belongs to the current class or not. So given enough classes, it is quite easy to create a rare-class problem. This is a commonly used approach due to the fact that most classification algorithms are meant for binary datasets.

The following datasets were used without any modifications (other than the split into "one-against-rest"). None of them have any missing attributes and all attributes are numeric.

6.2.1 Pima Indians Diabetes Dataset

The PIMA Indians Diabetes Dataset (henceforth referred to as PIMA) was generated by the National Institute of Diabetes and Digestive and Kidney Diseases and was originally used to test the ADAP learning algorithm [48]. It consists of a variety of attributes that are meant to be used to determine if a person, in this case a female of Pima Indian heritage at least 21 years of age, does or does not have diabetes, and thus is a binary dataset (a woman is either "positive or "negative" for having diabetes. The attributes are the following:

1. Number of times pregnant
2. Plasma glucose concentration at 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-Hour serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)squared)

7. Diabetes pedigree function
8. Age (years)

The dataset has 768 total data members, in which 268 are "positive" for diabetes and 500 are "negative."

6.2.2 *E. coli* Protein Localization Sites Dataset

The *E. coli* Protein Localization Sites Dataset (henceforth referred to as E. coli) was created to predict the cellular localization sites of proteins in *E. coli* using a probabilistic classification system [25]. Each data member consists of seven measurements associated with a protein with an assigned label indicating one of eight different cellular locations. The seven measurements/attributes are the following:

1. mcg: McGeoch's method for signal sequence recognition.
2. gvh: von Heijne's method for signal sequence recognition.
3. lip: von Heijne's Signal Peptidase II consensus sequence score. Binary attribute.
4. chg: Presence of charge on N-terminus of predicted lipoproteins. Binary attribute.
5. aac: score of discriminant analysis of the amino acid content of outer membrane and periplasmic proteins.
6. alm1: score of the ALOM membrane spanning region prediction program.
7. alm2: score of ALOM program after excluding putative cleavable signal regions from the sequence.

Of the eight different classes, we selected three to be used for the testing of our method: OM (outer membrane), pp (periplasm), and imU (inner membrane, uncleavable signal sequence). Of the 336 total data members 20 belong to OM, 52 belong to pp, and 35 belong to imU.

6.2.3 Yeast Protein Localization Sites Dataset

The Yeast Protein Localization Sites Dataset (henceforth referred to as Yeast) was also created to predict the cellular localization sites of proteins using a probabilistic classification system, but this time in the yeast organism [25]. Each data member consists of eight measurements associated with a protein with an assigned label indicating one of 10 different cellular locations. These measurements/attributes are the following:

1. mcg: McGeoch's method for signal sequence recognition.
2. gvh: von Heijne's method for signal sequence recognition.
3. alm: Score of the ALOM membrane spanning region prediction program.
4. mit: Score of discriminant analysis of the amino acid content of the N-terminal region (20 residues long) of mitochondrial and non-mitochondrial proteins.
5. erl: Presence of "HDEL" substring (thought to act as a signal for retention in the endoplasmic reticulum lumen). Binary attribute.
6. pox: Peroxisomal targeting signal in the C-terminus.
7. vac: Score of discriminant analysis of the amino acid content of vacuolar and extracellular proteins.
8. nuc: Score of discriminant analysis of nuclear localization signals of nuclear and non-nuclear proteins.

Of the ten different classes, we selected three to be used for the testing of our method: CYT (cytosolic or cytoskeletal), EXC (extracellular), and ME2 (membrane protein, uncleaved signal). Of the 1484 data members 463 belong to CYT, 51 belong to ME2, and 37 belong to EXC.

6.2.4 Abalone Age Prediction Dataset

The Abalone Age Prediction Dataset (henceforth referred to as Abalone) was generated by the Sea Fisheries Division of the Department of Primary Industry and Fisheries of Tasmania [38] and was used to test a cascade-correlation methodology [55]. The data was used to determine if the age of an abalone can be determined based on physical measurements. These physical measurements are the following:

1. Length: Longest shell measurement
2. Diameter: perpendicular to length
3. Height with meat in shell
4. Whole weight: whole abalone
5. Shucked weight: weight of meat
6. Viscera weight: gut weight (after bleeding)

7. Shell weight: after being dried

Of the 28 different ages, we selected seven different ones (3 years - 9 years). Of the 4177 data members, 15 are age 3, 57 are age 4, 115 are age 5, 259 are age 6, 391 are age 7, 568 are age 8, 689 are age 9.

6.3 Classification Algorithms

In order to show the effectiveness of our methodology, we chose "basic" classification algorithms that perform "well" on the data but leave room for improvement. This was done in order that the improvement in subsequent classification is clearly due to our methods and not due to aspects of the classification algorithms. As such commonly used classification methods for rare-class datasets such as SVMs and random forests were not used. All classification methods were used using only their default parameters (as set by weka [57]). This was also done to ensure that it is our method that is improving these classification algorithms and not tweaks done to the classification algorithms themselves.

- **NaiveBayes**
  A naive Bayes classifier is a probabilistic classifier that makes its classifications based on the Bayes theorem and assumes strong independence between attributes. The implementation we use is based on an implementation by John and Langley [29]. It uses estimator classes to determine numeric estimator precision values based on analysis of the training data.

- **JRip**
  JRip is an implementation of the RIPPER algorithm (Repeated Incremental Pruning to Produce Error Reduction) and is a rule-based learner. It was developed by William Cohen [10] and is an optimized version of IREP.

- **PART**
  The PART classifier is a combination of a rule-based learner and a decision tree learner (in this case a C4.5 decision tree) [21]. Using a separate and conquer approach, it creates a partial decision tree with each iteration and makes the "best" leaf into a rule.

- **J4.8**
  J4.8 is a Java-based (hence the "J") implementation of creating a pruned or unpruned C4.5 decision tree [42]. The default is to use pruning with a confidence threshold of 0.25.

6.4 Splitting up training and test data

As mentioned in our methods, we use training data to train the classification algorithms through MetaCost and then we use test data to evaluate the performance of the generated classification models. This however biases our results towards the test data and any subsequent performance analyses on the performance of the classification model on this test data do not provide adequate proof of our methods effectiveness. So for performance analyses we need to test the final generated classification model on a set of test data that was not used in any part of the learning process, including the search for a best cost matrix.

An initial solution to this problem was to use the training data as both training data for the classification algorithm and evaluation data for the generated classification model. This however led to overfitting, as the classification model was essentially being trained twice on the same set of data. While moderate increases in performance were observed, they were not as good as the subsequent solution which consisted of splitting up the dataset into three sets. For each dataset, we split the data into training data (1/2 of the dataset), evaluation test data (1/4 of the dataset), and final test data (1/4 of the dataset).

In order to show that our method performs better than an unaltered classifier, we also needed to train and test classification models without the use of cost matrices. Since both the training data and the evaluation test data are being used in the learning of the classification model, these two datasets were then put together for the unaltered classifier, effectively using 3/4 of the data for training and 1/4 of the data for testing.

Due to the fact that performance can be greatly improved/decreased due to which data members are in the training or test data, we created 10 different samples of training/test data for each of the listed datasets. Since many of the datasets are small and in addition have an even smaller amount of positive (minority) class members, these samples were not created randomly. Random partitioning can lead to some sets of training and/or test data having no positive class members, which will not be useful for learning. Therefore each dataset was placed into positive and negative class members and then split up to preserve the ratio. So 1/2 of the positive data members were randomly selected to be used in the training data and the remaining 1/2 was then split into the two test sets. The negative class was split in the same fashion. This way the ratio is preserved, each sample contains all data members, and we have 10 samples to provide sufficient evidence of improvement.

6.5 Unaltered Classification vs. Search Methods

This section presents the results of our methods compared to unaltered classification models. For all datasets, the previously described split of training/evaluation test/final test data was used and 10 different samples of data were created. The evaluation metric used for these results is the F-Measure. For each dataset we provide a comparison of average/median F-

Measures, a comparison of standard deviation, minimum and maximum values, a comparison of average accuracy/true positives, and statistical significance of improvement. In all tables, the best values are in bold font.

6.5.1 Pima Results

Table 6.3: Pima Average/Median F-Measures

	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.633**	**0.634**	0.623	0.618	0.628	0.620	0.628	0.620
JRip	0.592	0.585	0.663	**0.673**	**0.665**	0.656	0.655	0.656
PART	0.578	0.579	0.661	0.658	0.675	0.677	**0.676**	**0.683**
J4.8	0.608	0.602	0.647	0.651	0.644	0.648	**0.650**	**0.661**

Table 6.4: Pima Standard Deviation/Minimum/Maximum F-Measures

	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	0.037	0.563	0.677	0.038	0.571	0.681	**0.034**	0.586	0.681	**0.034**	**0.580**	**0.686**
JRip	0.069	0.464	0.699	**0.045**	0.594	**0.731**	**0.045**	**0.606**	**0.731**	0.053	0.563	**0.731**
PART	0.082	0.420	0.707	0.040	0.616	**0.737**	**0.029**	**0.623**	0.711	0.030	**0.623**	0.711
J4.8	0.051	0.530	0.672	0.030	**0.599**	0.686	**0.029**	**0.599**	0.679	0.033	**0.599**	**0.683**

Table 6.5: Pima Average Accuracy and Average True Positives

	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.757**	40.2	0.675	51.6	0.636	**54.7**	0.639	53.8
JRip	**0.741**	36.7	0.727	**51.8**	0.739	42.8	0.724	42.5
PART	0.697	41.5	0.739	48.8	0.713	53.3	**0.742**	**49.6**
J4.8	**0.733**	40	0.720	49.2	0.711	49.1	0.709	**52.6**

In Table 6.3, we can see the average and median F-Measures across the four classifiers for both unaltered learning and our three approaches. Immediately evident in this dataset is that NaiveBayes is not improved upon in terms of increasing the F-Measure. In fact the F-Measure is decreased, even though it is only by a small amount (approximately 0.01 for the greedy approach and 0.005 for both simulated annealing and the genetic algorithm). The other three classifiers however do show an increase in their F-Measures. JRip has the best average F-Measure generated by the simulated annealing approach (an increase of 0.073). The best median is however generated by the greedy approach, improving by 0.088 over the unaltered median. PART shows the largest amount of improvements, which range between 0.083 and 0.097 for the improvement of the average F-Measure and 0.08 and 0.104 for the median F-Measure. PART also generates the highest overall average and median F-Measures, both generated by the genetic algorithm. J4.8 also shows F-Measure improvement, with the genetic algorithm again generating the best improvements.

Table 6.6: Pima Wilcoxon Signed Rank Test p-values of Significance of Improvement over Unaltered Classifiers

Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.0050	0.0027	0.0037
PART	0.0067	0.0037	0.0067
J4.8	0.0037	0.0249	0.0067

In Table 6.4, we can see the standard deviation of the F-Measures as well as the minimum and maximum generated F-Measures. As we showed in Table 6.3, NaiveBayes showed no improvement in terms of average F-Measures. However, here we can see that both simulated annealing and the genetic algorithm reduced the standard deviation while both the minimum and the maximum F-Measures are higher than those of the unaltered classifier. While this is not an overwhelming improvement, it does show that these methods are *capable* of improving upon the unaltered NaiveBayes classifier. Other interesting trends include the fact that simulated annealing consistently has the lowest standard deviations, across all four classifiers. For most classifiers, the genetic algorithm generates the highest overall F-Measure, except for PART. The highest F-Measure for PART, and the highest overall F-Measure, was generated by the greedy method.

In Table 6.5, we can see the average accuracy and average amount of true positives for the final generated classification models. As stated previously, accuracy is a poor metric for rare-class data and in many situations by improving upon the F-Measure, the accuracy is lowered. This is why we also include the amount of true positives, to show that while the classification might be "less accurate" the final generated classification models do correctly identify more of the desired class. Unsurprisingly, NaiveBayes has the best accuracy when left unaltered. The same can be said for JRip and J4.8 although their drops in accuracy are not as large. PART however, shows an improvement in accuracy in all methods with the best accuracy achieved by the genetic algorithm. And while the accuracy was usually worse for all classifiers, all methods improved upon the average accuracy with the combination of NaiveBayes and simulated annealing producing the best overall average true positives.

In order to determine if the improvement in F-Measures was statistically significant, we performed a Wilcoxon Signed Rank Test [56], comparing the F-Measures of the unaltered classification models to the F-Measures generated by the final classification models of the three methods across the 10 samplings of the dataset. The resulting p-values can be seen in Table 6.6. Due to the lack of improvement, NaiveBayes is not present in the table. The other three classifiers show a definite, statistically significant improvement across all methods (assuming that a p-value < 0.05 is statistically significant).

6.5.2 E. coli Results

In Table 6.7, we can see the average and median F-Measures across the three classes (OM, pp, imU) of the E. coli dataset. Unlike with the Pima dataset, we do not have an overwhelming improvement in the majority of classifiers. This is most likely due to the fact that the E. coli dataset is quite small (336 data members to Pima's 768). Nevertheless, improvements can be seen in some situations. Class OM achieves its highest F-Measures with unaltered NaiveBayes and any subsequent use of our methods only lowers both the average and median F-Measures. However, the other three classifiers are all improved upon, especially when using the simulated annealing method. The highest averages and medians are all achieved by it. Interesting is what occurs with the J4.8 classifier: both the greedy approach and the genetic algorithm produce worse average F-Measures, while the simulated annealing method improves upon the unaltered. Overall the greedy approach performs the poorest, improving only upon the PART classifier.

In Class pp the best F-Measures are again achieved through NaiveBayes. However, only the best average is achieved through the unaltered classifier, better median values are achieved by all three cost matrix search methods. For both JRip and PART the best F-Measures are achieved by unaltered classifiers. However, J4.8 sees improvement across all three methods, with the best results achieved by the genetic algorithm.

In Class imU we again see no improvement in the average and median F-Measures for NaiveBayes and JRip. However both PART and J4.8 see improvement with both simulated annealing and the genetic algorithm. While the highest median F-Measure is acheived through NaiveBayes, the highest overall average F-Measure is achieved through the combination of J4.8 and simulated annealing.

In Table 6.8, we can see the standard deviations, minimums, and maximums of the F-Measures for the classifiers. In Class OM only the PART classifier exhibits any improvement upon its standard deviation. It is also the only classifier that sees an improvement in its minimum F-Measure, as the other classifiers display minimums far worse (some as low as 0). Both simulated annealing and the genetic algorithm achieve the highest maximum F-Measures.

In Class pp the greedy approach performs well, achieving the highest F-Measures in all classifiers except NaiveBayes. Standard deviation largely does not improve greatly, with the only large improvement being with the combination of J4.8 and the genetic algorithm. In Class imU the only improvement in standard deviation can be seen in the greedy approach with the NaiveBayes classifier. All of the highest minimum F-Measures belong to the unaltered classifiers but aside from the NaiveBayes classifier, the highest F-Measures are achieved through simulated annealing and the genetic algorithm. The overall highest F-Measure belongs to the J4.8 classification algorithm.

In Table 6.9, we can see the average accuracy values and average true positives across all three classes. Interestingly in class OM, the genetic algorithm achieves the highest accuracy

and highest number of true positives across all classifiers. This is despite not having the highest average F-Measures for any classifier. In class pp the only improvement in accuracy can be seen with J4.8 using simulated annealing. Both PART and J4.8 show an improvement in the number of true positives using simulated annealing and the genetic algorithm.

Overall, our approach did not perform uniformly well on the E. coli dataset. In fact, the only classifier that showed a statistically significant increase in its F-Measure was simulated annealing. This was only in the OM class and only with the PART classifier. The Wilcoxon signed rank test generated a p-value of 0.0279. The poor, inconsistent performance could easily be attributed to the lack of data. This is made even more evident when comparing the results in the yeast dataset, which are of a similar type and for which there are more data members.

Table 6.7: E. coli Average/Median F-Measures

Class OM								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.834**	**0.899**	0.697	0.750	0.779	0.889	0.772	0.844
JRip	0.623	0.641	0.592	0.641	**0.644**	**0.667**	0.632	0.641
PART	0.554	0.619	0.632	**0.727**	**0.742**	**0.727**	0.717	**0.727**
J4.8	0.677	0.667	0.625	0.667	**0.702**	**0.727**	0.637	**0.727**
Class pp								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.858**	0.884	0.837	**0.917**	0.841	**0.917**	0.839	**0.917**
JRip	**0.839**	**0.846**	0.816	0.801	0.793	0.796	0.788	0.788
PART	**0.802**	**0.800**	0.799	0.784	0.781	0.778	0.787	0.784
J4.8	0.790	0.807	0.797	**0.823**	0.815	0.806	**0.834**	**0.823**
Class imU								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.446**	**0.429**	0.392	0.361	0.400	0.361	0.408	0.361
JRip	**0.373**	**0.348**	0.350	0.319	0.359	0.325	0.353	0.329
PART	0.400	0.390	0.421	0.390	**0.449**	**0.391**	**0.449**	**0.391**
J4.8	0.414	0.408	0.309	0.254	**0.457**	**0.416**	0.455	**0.416**

6.5.3 Yeast Results

In Table 6.10 we can see the average and median F-Measures for the classifiers across all 3 tested classes. In class CYT we see that our methods performed extremely well. All four classifiers (even NaiveBayes) are improved in terms of both average and median F-Measure. In fact all of the unaltered classifiers have F-Measures below 0.5, while all F-Measures generated through our methods are above 0.5. Interesting also is the better performance of simulated annealing as it produces the best F-Measures across all classifiers. The best clas-

Table 6.8: E. coli Standard Deviation/Minimum/Maximum F-Measures

Class OM												
	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.225**	**0.333**	**1.000**	0.232	**0.333**	**1.000**	0.245	0.278	**1.000**	0.255	0.270	**1.000**
JRip	**0.183**	**0.222**	0.833	0.249	0.000	**0.909**	0.252	0.000	**0.909**	0.254	0.000	**0.909**
PART	0.207	0.000	0.714	**0.166**	0.333	0.833	0.181	**0.400**	**1.000**	0.199	**0.400**	**1.000**
J4.8	**0.106**	**0.545**	0.833	0.165	0.333	0.833	0.185	0.370	**0.909**	0.197	0.333	**0.909**
Class pp												
	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.074**	**0.688**	0.917	0.114	0.595	0.917	0.114	0.595	**0.923**	0.111	0.611	**0.923**
JRip	**0.065**	0.720	**0.923**	0.068	**0.727**	**0.923**	0.084	0.643	**0.923**	0.086	0.643	**0.923**
PART	0.069	**0.696**	0.880	0.076	0.667	**0.923**	0.069	0.647	0.917	**0.068**	0.647	0.917
J4.8	0.073	0.625	0.870	0.141	0.435	**0.917**	0.080	0.647	**0.917**	**0.054**	**0.759**	**0.917**
Class imU												
	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	0.180	**0.225**	**0.800**	**0.162**	0.213	0.696	0.175	0.213	0.700	0.187	0.213	0.762
JRip	**0.102**	**0.267**	0.545	0.142	0.203	**0.600**	0.136	0.203	**0.600**	0.129	0.203	**0.600**
PART	**0.111**	**0.250**	0.588	0.188	0.200	0.696	0.228	0.200	**0.875**	0.228	0.200	**0.875**
J4.8	**0.137**	**0.250**	0.667	0.152	0.200	0.696	0.228	0.200	**0.941**	0.230	0.200	**0.941**

Table 6.9: E. coli Average Accuracy and Average True Positives

Class OM								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	0.979	4.4	0.957	3.4	0.991	4.9	**0.992**	**5**
JRip	0.951	3.6	**0.954**	3.2	0.934	3.6	**0.954**	**4.9**
PART	0.950	3	0.946	3.7	0.937	4.3	**0.975**	**4.9**
J4.8	0.958	3.7	0.960	3.2	0.915	4.6	**0.962**	**4.7**
Class pp								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.955**	**11**	0.945	**11**	0.943	10.5	0.943	10.5
JRip	**0.949**	**11.2**	0.938	**11.2**	0.913	10.7	0.911	10.7
PART	**0.939**	10.2	0.937	10.2	0.924	**11.4**	0.928	**11.4**
J4.8	0.933	10.1	0.923	10.1	**0.939**	11.3	0.887	**11.4**
Class imU								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.704**	7.2	0.652	7.2	0.697	6.8	0.692	**7.5**
JRip	0.788	4.7	0.675	**5.6**	**0.807**	4.8	0.804	4.8
PART	0.799	4.8	0.705	6.4	0.792	**7.3**	**0.811**	7
J4.8	0.800	4.7	0.670	5.6	**0.825**	**7.4**	0.821	7.3

Table 6.10: Yeast Average/Median F-Measures

Class CYT								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	0.479	0.463	0.526	0.518	**0.528**	**0.522**	0.526	0.521
JRip	0.437	0.452	0.523	0.517	**0.550**	**0.548**	0.531	0.526
PART	0.362	0.345	0.528	0.532	**0.557**	**0.556**	0.553	0.553
J4.8	0.462	0.490	0.537	0.547	**0.549**	**0.549**	0.548	**0.549**
Class EXC								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.459**	**0.459**	0.368	0.367	0.385	0.382	0.395	0.390
JRip	0.442	0.462	0.416	0.429	**0.522**	**0.521**	**0.522**	**0.521**
PART	0.320	0.333	0.433	0.418	0.460	0.431	**0.480**	**0.462**
J4.8	0.304	0.276	0.461	**0.481**	0.476	**0.481**	**0.479**	0.462
Class ME2								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	0.318	**0.326**	0.321	0.311	**0.339**	0.322	0.337	0.322
JRip	0.398	0.372	0.441	0.400	0.453	**0.449**	**0.479**	**0.449**
PART	0.268	0.238	**0.441**	**0.428**	0.425	0.392	0.437	**0.428**
J4.8	0.378	0.388	0.416	0.411	**0.464**	**0.441**	0.439	0.421

Table 6.11: Yeast Standard Deviation/Minimum/Maximum F-Measures

Class CYT												
	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.036**	0.439	0.546	0.045	0.447	**0.586**	0.042	**0.451**	**0.586**	0.041	0.447	**0.586**
JRip	0.074	0.265	0.536	0.048	0.448	0.605	0.050	**0.461**	**0.613**	**0.043**	0.448	0.602
PART	0.147	0.135	0.576	0.044	0.448	**0.593**	**0.025**	0.512	**0.593**	**0.025**	**0.516**	**0.593**
J4.8	0.066	0.327	0.521	0.028	0.487	0.565	**0.016**	**0.524**	0.574	**0.016**	**0.524**	**0.575**
Class EXC												
	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.079**	**0.353**	**0.625**	0.096	0.211	0.500	0.088	0.211	0.500	0.095	0.211	0.500
JRip	0.169	0.200	0.667	0.184	0.000	0.667	**0.124**	**0.308**	**0.714**	**0.124**	**0.308**	**0.714**
PART	**0.108**	0.167	0.471	0.118	0.286	0.632	0.135	**0.300**	**0.750**	0.143	0.273	**0.750**
J4.8	0.136	0.154	0.471	**0.108**	0.316	0.615	0.143	0.261	**0.667**	0.132	**0.333**	**0.667**
Class ME2												
	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.035**	0.250	0.375	0.085	0.186	**0.483**	0.090	0.186	**0.483**	0.088	0.186	**0.483**
JRip	0.114	0.267	0.600	0.117	0.333	**0.667**	0.110	0.250	**0.667**	**0.103**	**0.357**	**0.667**
PART	**0.097**	0.133	0.476	0.100	**0.276**	0.600	0.126	**0.276**	**0.667**	0.108	**0.276**	0.600
J4.8	**0.048**	0.273	0.444	0.130	0.154	0.600	0.112	**0.286**	**0.667**	0.096	**0.286**	**0.667**

Table 6.12: Yeast Average Accuracy and Average True Positives

Class CYT								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.646**	80.8	0.571	89.4	0.538	**108.6**	0.532	108.5
JRip	**0.677**	47.3	0.573	85.2	0.615	91.5	0.603	**93.9**
PART	**0.661**	41.2	0.661	85.3	0.602	**100.8**	0.585	100.1
J4.8	**0.672**	52.9	**0.672**	81.5	0.588	**96.5**	0.589	96
Class EXC								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.965**	7.4	0.963	4	0.960	5.4	0.959	5.3
JRip	0.978	3.3	**0.981**	2.8	0.976	3.3	0.976	**3.5**
PART	**0.977**	2.1	**0.977**	**3.3**	0.970	3	0.969	3
J4.8	0.974	2.2	**0.979**	**3.3**	0.965	3	0.966	3
Class ME2								
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.936**	7.4	0.924	6.5	0.917	**8.1**	0.915	**8.1**
JRip	0.966	4.2	**0.970**	4.4	0.964	4.6	0.960	**4.9**
PART	0.961	2.7	**0.965**	4.8	0.950	5.3	0.950	**5.4**
J4.8	0.965	3.9	**0.967**	4.5	0.956	**5.8**	0.956	**5.8**

Table 6.13: Yeast Wilcoxon Signed Rank Test p-values of Significance of Improvement over Unaltered Classifiers

Class CYT			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
NaiveBayes	0.031428	0.031428	0.031428
JRip	0.006722	0.002738	0.002738
PART	0.008897	0.002738	0.002738
J4.8	0.006722	0.002738	0.002738
Class EXC			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.389621	0.017213	0.017213
PART	0.019505	0.007743	0.003729
J4.8	0.015156	0.031428	0.024873
Class ME2			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.262043	0.200197	0.048825
PART	0.002738	0.002738	0.002738
J4.8	0.200197	0.006722	0.019505

sification in terms of average/median F-Measures is achieved through PART and simulated annealing.

In class EXC we see that our methods were unable to improve upon NaiveBayes, producing lower F-Measures. The other three classification algorithms were improved upon, especially by the genetic algorithm which achieves the highest average F-Measures for all three. The best F-Measures are achieved with JRip and both simulated annealing and the genetic algorithm (although the greedy approach was unable to improve upon the unaltered classifier).

Class ME2 has an interesting result in terms of the NaiveBayes classifier. The average F-Measure is higher in all of our methods, but the median is the highest for the unaltered classifier. However the remaining three classifiers are all improved by our methods. Interesting here is that for each classifier, a different method produces the best result. JRip achieves its best F-Measure through the genetic algorithm, PART achieves its best F-Measure through the greedy approach, and J4.8 achieves its best result through simulated annealing. The overall best F-Measures are achieved through JRip and the genetic algorithm.

In Table 6.11 we can see standard deviation, minimum, and maximum of the F-Measures. In class CYT, aside from with NaiveBayes, the genetic algorithm achieves the lowest standard deviations across all classifiers. The genetic algorithm also achieves the highest F-Measures across all the classifiers with the exception of JRip, which achieves its highest results with simulated annealing.

As expected based on Table 6.10, NaiveBayes has its best values in its unaltered state. Other than that, the highest F-Measures are achieved by both simulated annealing and the genetic algorithm. The greedy method does have the largest improvement on standard deviation with the J4.8 classification algorithm. In class ME2, only JRip sees an improvement in its standard deviation, the best result being achieved by the genetic algorithm. Simulated annealing achieves the highest F-Measures for all classifiers.

In Table 6.12, we can see the average accuracy and average true positives for the classifiers. While we saw universal improvement in terms of F-Measures in the CYT class, we see the opposite in terms of accuracy as all the unaltered classifiers have the best accuracy. However, in terms of average true positives, we see drastic increases, especially with JRip, PART, and J4.8. The largest amount of true positives are achieved through the combination of NaiveBayes and simulated annealing.

In class EXC, the largest amount of true positives are achieved by the unaltered NaiveBayes classifier and our methods do not improve upon its accuracy. However, the greedy approach does achieve better or comparable accuracy values for all other classifiers. It also increases the amount of average true positives for both PART and J4.8. The greedy approach has a similar effect on the accuracy in class ME2, improving upon all accuracy values except those of the unaltered NaiveBayes. The genetic algorithm consistently generates the highest number of average true positives, with the highest number being generated through it and NaiveBayes. Simulated annealing also achieves this level with NaiveBayes.

In Table 6.13, we can see the p-values generated by the Wilcoxon Signed Rank Test on the generated F-Measures. The NaiveBayes results were left out from class EXC and ME2 due to a lack of any significance. In class CYT, we see that all combinations of classification algorithm and method generated significant results. In class EXC, we see that both simulated annealing and the genetic algorithm generated significant results for all three classifiers, but the greedy approach could not with JRip. In class ME2 we see that the genetic algorithm generated significant results for all classifiers, but simulated annealing fell short with JRip and the greedy approach only achieved significant results with PART.

6.5.4 Abalone Results

In Table 6.14, we can see the average and median F-Measure values across the various methods and the different classes. In class 3, we see that NaiveBayes shows no improvement. However, the other three classifiers show much improvement, especially PART and J4.8, which improve from very poor performance (PART makes no correct classifications in its unaltered state). The overall best average belongs to the combination of JRip and simulated annealing.

NaiveBayes's poor improvement continues in class 4 and 5. However, the other three classifiers are improved upon in class 3 with the greedy approach achieving the best results with PART and J4.8 and simulated annealing with JRip. The best average was achieved with J4.8 and the greedy approach. In Class 5, the best average for each classifier is achieved by a different approach: JRip is best with simulated annealing, PART is best with the greedy approach, and J4.8 is best with the genetic algorithm. PART sees the largest amount of improvement, going from the worst performing classifier to the best.

In class 6, we actually improve upon NaiveBayes with both simulated annealing and the genetic algorithm. While it is not a large improvement, it is still an overall improvement that is not observed in any of the other abalone data classes. The other three classifiers perform best with simulated annealing, achieving the best results when combined with J4.8. Again, we see a large improvement with PART.

In class 7, we again fail to improve upon NaiveBayes. However, the other three classifiers are improved, with PART again going from worst performing to best performing classifier. A further interesting trend can be observed in J4.8, where all approaches achieve the same average F-Measure. However, simulated annealing seems to perform best based on the fact that it achieves a higher F-Measure.

Finally, in classes 8 and 9, we see consistent results across all approaches for JRip. Meanwhile, PART and J4.8 achieve their best results when combined with the genetic algorithm approach in class 8. PART also achieves its best results when combined with the genetic algorithm in class 9, however J4.8 achieves better results when combined with simulated annealing.

In Table 6.15, we can see the standard deviation, minimums, and maximums across all

Table 6.14: Abalone Average/Median F-Measures

					Class 3			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.246**	**0.231**	0.233	0.211	0.235	0.223	0.240	0.226
JRip	0.250	**0.400**	0.307	0.325	**0.312**	0.348	0.295	0.333
PART	0.000	0.000	0.205	0.216	0.244	0.297	**0.304**	**0.312**
J4.8	0.040	0.000	0.182	0.118	0.273	**0.301**	**0.295**	**0.301**
					Class 4			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.371**	**0.366**	0.330	0.324	0.331	0.324	0.336	0.326
JRip	0.471	0.490	0.468	0.468	**0.502**	**0.567**	0.497	0.524
PART	0.364	0.397	**0.482**	**0.501**	0.472	0.441	0.457	0.483
J4.8	0.448	0.440	**0.507**	0.489	0.489	0.466	0.499	**0.507**
					Class 5			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.318**	**0.316**	0.302	0.303	0.308	0.306	0.314	0.312
JRip	0.300	0.289	0.400	0.405	**0.410**	0.421	0.404	**0.431**
PART	0.089	0.061	**0.419**	0.419	0.415	0.422	0.417	**0.440**
J4.8	0.194	0.222	0.390	0.413	0.393	0.403	**0.408**	**0.419**
					Class 6			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	0.364	0.361	0.355	0.353	0.368	0.365	**0.373**	**0.372**
JRip	0.202	0.211	0.375	0.380	0.411	**0.427**	**0.412**	0.420
PART	0.050	0.030	0.430	0.430	**0.441**	**0.442**	0.438	0.436
J4.8	0.264	0.273	0.440	0.430	**0.444**	**0.436**	0.423	0.422
					Class 7			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.368**	**0.359**	0.357	0.355	0.361	0.359	0.360	0.355
JRip	0.165	0.180	0.376	0.376	**0.389**	0.381	0.384	**0.382**
PART	0.060	0.019	0.411	**0.424**	**0.418**	0.421	0.413	0.413
J4.8	0.230	0.216	**0.407**	0.405	**0.407**	**0.413**	**0.407**	0.407
					Class 8			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.359**	0.355	0.344	0.342	0.356	**0.360**	0.355	0.359
JRip	0.124	0.129	**0.335**	**0.335**	**0.335**	**0.335**	0.334	**0.335**
PART	0.156	0.168	0.360	0.351	0.367	0.371	**0.375**	**0.381**
J4.8	0.246	0.241	0.350	0.356	0.363	0.358	**0.366**	**0.369**
					Class 9			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median	Average	Median	Average	Median
NaiveBayes	**0.358**	**0.361**	0.341	0.342	0.344	0.344	0.341	0.342
JRip	0.059	0.048	**0.292**	**0.283**	**0.292**	**0.283**	**0.292**	**0.283**
PART	0.051	0.050	0.351	0.358	0.357	0.367	**0.363**	**0.369**
J4.8	0.221	0.238	0.346	0.351	**0.354**	**0.352**	0.339	0.345

Table 6.15: Abalone Standard Deviation/Minimum/Maximum F-Measures

Class 3

	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.046**	**0.207**	0.333	0.074	0.154	**0.375**	0.068	0.162	**0.375**	0.069	0.171	**0.375**
JRip	0.217	0.000	0.500	**0.121**	**0.143**	**0.545**	0.166	0.000	**0.545**	0.153	0.000	**0.545**
PART	0.000	0.000	0.000	0.200	0.000	0.462	**0.183**	0.000	0.462	0.192	0.000	**0.600**
J4.8	**0.126**	0.000	0.400	0.207	0.000	**0.545**	0.193	0.000	**0.545**	0.164	0.000	**0.545**

Class 4

	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.051**	**0.306**	**0.441**	0.063	0.255	0.406	0.063	0.260	0.400	0.062	0.260	0.412
JRip	0.104	0.222	**0.615**	0.124	0.267	0.606	0.107	**0.286**	0.593	**0.098**	**0.286**	0.593
PART	0.170	0.000	0.583	**0.104**	**0.348**	0.600	0.109	**0.348**	**0.625**	0.143	0.111	0.600
J4.8	0.177	0.200	**0.828**	0.098	0.348	0.645	**0.089**	0.348	0.645	0.100	**0.351**	0.649

Class 5

	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	0.028	**0.284**	**0.379**	**0.023**	0.270	0.341	**0.023**	**0.284**	0.355	0.027	0.275	0.352
JRip	**0.062**	0.200	0.400	0.113	**0.250**	0.551	0.108	**0.250**	**0.564**	0.097	**0.250**	0.535
PART	0.102	0.000	0.271	0.080	**0.318**	**0.538**	**0.069**	0.310	0.533	0.081	0.283	0.533
J4.8	0.094	0.000	0.300	0.090	0.217	0.485	0.098	0.217	**0.537**	**0.077**	**0.289**	0.511

Class 6

	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	0.020	0.330	0.397	**0.018**	**0.331**	0.390	0.023	0.320	**0.408**	0.024	0.320	0.406
JRip	0.056	0.132	0.295	0.066	0.274	0.477	0.059	0.274	**0.479**	**0.047**	**0.323**	0.478
PART	0.048	0.000	0.123	0.031	0.380	0.484	0.026	0.393	0.481	0.031	0.376	0.481
J4.8	0.068	0.108	0.365	0.037	0.379	0.512	0.038	0.379	0.512	0.053	0.300	0.485

Class 7

	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	0.023	**0.344**	**0.410**	**0.018**	0.329	0.394	0.021	0.337	0.401	0.021	0.337	0.400
JRip	0.064	0.037	0.252	0.048	**0.376**	0.376	**0.039**	0.333	**0.477**	**0.039**	0.333	**0.477**
PART	0.113	0.000	0.372	0.037	0.342	0.452	0.033	0.366	**0.473**	**0.027**	**0.376**	0.458
J4.8	0.058	0.130	0.315	0.035	**0.356**	**0.462**	**0.031**	**0.356**	0.449	0.039	0.353	**0.462**

Class 8

	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.019**	**0.329**	**0.390**	0.020	0.312	0.377	0.021	0.325	0.385	0.022	0.324	0.381
JRip	0.124	0.060	0.205	**0.045**	**0.268**	**0.415**	**0.045**	**0.268**	**0.415**	**0.045**	**0.268**	**0.415**
PART	0.114	0.014	0.343	0.035	0.303	0.413	0.037	0.303	**0.432**	**0.017**	**0.341**	0.394
J4.8	0.045	0.168	0.308	0.028	0.292	0.385	**0.018**	**0.330**	0.385	0.030	0.301	**0.402**

Class 9

	Unaltered			**Greedy**			**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max	Std Dev	Min	Max
NaiveBayes	**0.014**	**0.339**	**0.380**	0.019	0.311	0.374	0.018	0.312	0.374	0.017	0.312	0.362
JRip	0.063	0.000	0.183	**0.031**	**0.257**	**0.369**	**0.031**	**0.257**	**0.369**	**0.031**	**0.257**	**0.369**
PART	0.042	0.000	0.149	0.029	**0.284**	**0.385**	0.030	**0.284**	**0.383**	**0.023**	0.303	0.379
J4.8	0.041	0.162	0.288	0.019	0.301	**0.368**	**0.011**	**0.330**	**0.368**	0.021	0.283	0.355

Table 6.16: Abalone Average Accuracy/True Positives

					Class 3			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.982**	3	0.979	3	0.976	3	0.979	3
JRip	**0.997**	0.6	0.992	1.9	0.990	**2.2**	0.991	2
PART	**0.996**	0	0.993	1.3	0.992	1.6	0.992	**1.9**
J4.8	**0.996**	0.1	0.894	1.3	0.991	1.9	0.941	**2.3**
					Class 4			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.955**	13.4	0.946	13.4	0.943	13.4	0.945	13.4
JRip	**0.986**	6.4	0.984	7.6	0.983	**8.6**	0.983	8.3
PART	**0.986**	4.6	0.984	8.1	0.982	8.4	0.974	**10.4**
J4.8	**0.987**	5.6	0.986	7.9	0.984	8	0.968	**10.8**
					Class 5			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.900**	24	0.891	24.5	0.889	**25.2**	0.892	24.5
JRip	**0.967**	7.5	0.965	12.2	0.958	**14.5**	0.959	14.3
PART	**0.968**	2.1	0.961	15.2	0.947	18.6	0.938	**20.4**
J4.8	**0.965**	4.7	0.958	14.4	0.949	16.2	0.940	**18.9**
					Class 6			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	0.803	**58.4**	0.796	58.3	0.802	57.6	**0.810**	57
JRip	**0.930**	9.4	0.911	28.8	0.894	**37.5**	0.900	36.1
PART	**0.935**	1.9	0.895	41.3	0.886	45.3	0.849	**53.9**
J4.8	**0.923**	14.6	0.887	46.1	0.873	49.5	0.838	**55.3**
					Class 7			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	0.738	**79.1**	**0.740**	75	0.735	77	0.739	75.2
JRip	**0.897**	11	0.861	44.2	0.854	**47.4**	0.856	45.9
PART	**0.898**	4.7	0.853	54.1	0.843	56.8	0.797	**70.2**
J4.8	**0.878**	19.7	0.851	53.8	0.848	54.2	0.822	**60.4**
					Class 8			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	0.689	**90.9**	0.676	88.2	**0.707**	84.2	0.697	86.2
JRip	**0.855**	11	0.810	47.4	0.806	50.5	0.807	50.5
PART	**0.834**	19.2	0.770	69	0.751	75.9	0.701	94.7
J4.8	**0.813**	32.1	0.731	77.6	0.714	84.9	0.706	88
					Class 9			
	Unaltered		**Greedy**		**Simulated Annealing**		**Genetic Algorithm**	
Classifier	Accuracy	TP	Accuracy	TP	Accuracy	TP	Accuracy	TP
NaiveBayes	**0.660**	98.7	0.589	112	0.566	**120.1**	0.566	118.2
JRip	**0.829**	6.2	0.403	**125.8**	0.403	**125.8**	0.403	**125.8**
PART	**0.821**	5.4	0.625	108.2	0.606	124.2	0.571	**133.2**
J4.8	**0.768**	35	0.591	114	0.580	121.4	0.482	**148.1**

classes. These results continue to reflect what we have seen in the previous table, that NaiveBayes performs best when left unaltered. While some classes have less standard deviation, overall the best F-Measures are achieved in the unaltered state. The other classes show great improvement in not only their maximums and minimums, but also in lower standard deviations, showing that their results are more consistent.

In Table 6.16, we see the average accuracies and average true positive results for each classifier across the classes and approaches. With little exception, the best results in accuracy are achieved by the unaltered classifiers. However, our methods generally improve upon the number of true positives classified.

Table 6.17: Abalone Wilcoxon Signed Rank Test p-values of Significance of Improvement over Unaltered Classifiers

Class 3			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.2789	0.2789	0.3511
PART	0.0067	0.0043	0.0032
J4.8	0.0043	0.0032	0.0032
Class 4			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.5708	0.0732	0.0732
PART	0.0600	0.1258	0.1059
J4.8	0.0884	0.1729	0.1481
Class 5			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.0117	0.0037	0.0037
PART	0.0027	0.0027	0.0027
J4.8	0.0027	0.0027	0.0027
Class 6			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.0027	0.0027	0.0027
PART	0.0027	0.0027	0.0027
J4.8	0.0027	0.0027	0.0027
Class 7			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.0027	0.0027	0.0027
PART	0.0027	0.0027	0.0027
J4.8	0.0027	0.0027	0.0027
Class 8			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.0027	0.0027	0.0027
PART	0.0037	0.0027	0.0027
J4.8	0.0027	0.0027	0.0027
Class 9			
Classifier	**Greedy**	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.0032	0.0032	0.0032
PART	0.0027	0.0027	0.0027
J4.8	0.0027	0.0027	0.0027

In Table 6.17 we see the p-value results from the Wilcoxon Signed Rank test. NaiveBayes values were excluded due to their general lack of improvement. In classes 5-9 we see an improvement across all metrics, with most classifier approach combinations achieving the lowest possible p-value of 0.0027. In class 3 however, we see no significant improvement over unaltered JRip classification. And in class 4, we see no significant improvements at all.

6.6 Comparison of Metrics

In the previous section, we detailed our results for the various datasets in terms of their improvement on the F-Measure. As mentioned in our definitions, the F-Measure is a commonly used metric for rare-class classification. Its focus is on recall and precision so a high F-Measure value contains a high amount of true positives without increasing the amount of false positives too much. However, as we also covered in the definition section, other rare-class metrics exist. These are the Area Under the ROC curve (or AUC) and the G-Mean. Both range in value from 0 to 1 (although the AUC usually never falls below 0.5) and can just as easily be used as an evaluation metric in all three approaches.

As an experiment on whether our approaches can be effective using other metrics, we ran the various cost matrix search approaches using AUC and G-Mean as the evaluation metric on the Pima dataset. The results (along with the original results using the F-Measure) can be seen in Table 6.18. For each evaluation metric, we show the average value of that search metric, the average accuracy, and the average amount of true positives.

The first thing that should be evident when looking at the results, is that regardless of the metric used, our approaches will improve upon the unaltered classifier. This is not that surprising as all of the cost matrix optimization approaches are designed to find the best classification model using the given metric. Unfortunately, none of the given metrics were able to improve upon the metrics for NaiveBayes. All other classifiers were however improved.

So ultimately it comes down to the results that the specific user wants. Judging by these results each of the classifiers seems to have its own strengths. The F-Measure seems to increase the amount of true positives the best, doing so better than AUC and G-Mean. The AUC however does better in increasing accuracy, doing so with three of the four classifiers (NaiveBayes being the long exception). The G-Mean however seems to fall in between these two metrics. It improves upon the accuracy of two of the four classifier but increases the number of true positives more than the AUC. It however does not increase the number of true positives more than the F-Measure.

In order to gain more insight into the different metrics and how they affect the search for a cost matrix, we decided to look at one of the 10 data samples to see what best cost matrices were generated by each of the different metrics. These results can be seen in Table6.19. For each search (F-Measure Search, AUC Search, and G-Mean Search) we show the best cost

Table 6.18: Pima Comparison of Metric Averages/Average Accuracy/Average True Positives

F-Measure

	Unaltered			**Greedy**		**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP
NaiveBayes	**0.633**	**0.757**	40.2	0.623	0.675	51.6	0.628	0.670	**55.7**	0.628	0.667	55.2
JRip	0.592	**0.741**	36.7	0.663	0.727	51.8	**0.665**	0.733	**52.5**	0.655	0.727	51.2
PART	0.578	0.697	41.5	0.661	0.739	48.8	0.675	0.742	**51.4**	**0.676**	**0.751**	50.1
J4.8	0.608	**0.733**	40.0	0.647	0.720	49.2	0.644	0.712	51.1	**0.650**	0.715	**52.6**

AUC

	Unaltered			**Greedy**		**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP
NaiveBayes	**0.817**	**0.757**	40.2	0.778	0.704	37.3	0.784	0.686	37.1	0.683	0.573	**61.6**
JRip	0.701	0.741	36.7	0.756	0.746	48.3	**0.770**	0.747	**52.7**	0.766	**0.749**	51.6
PART	0.725	0.697	41.5	0.766	0.732	45.2	**0.791**	0.751	47.7	0.789	**0.754**	**49.6**
J4.8	0.714	0.733	40.0	0.745	0.724	44.5	0.749	0.726	45.4	**0.763**	**0.744**	**50.5**

G-Mean

	Unaltered			**Greedy**		**Simulated Annealing**			**Genetic Algorithm**		
Classifier	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP	Average	Accuracy	Avg TP
NaiveBayes	**0.710**	**0.757**	40.2	0.702	0.694	50	0.702	0.694	50.2	0.703	0.695	**51.4**
JRip	0.676	0.741	36.7	0.734	0.740	48.8	**0.741**	**0.746**	50.2	0.739	0.743	**51**
PART	0.663	0.697	41.5	0.734	0.743	48	0.745	0.750	**51.5**	**0.751**	**0.763**	50.5
J4.8	0.692	**0.733**	40.0	**0.721**	0.728	47.3	0.719	0.723	48.8	**0.721**	0.724	**49**

matrix and what F-Measure (F), AUC, and G-Mean (G) values these cost matrices produce for each classification algorithm.

Table 6.19: Pima Comparison of Cost Matrices Across All Metrics

	F-Measure Search														
	Greedy					**Simulated Annealing**					**Genetic Algorithm**				
Classifier	CM		F	AUC	G	CM		F	AUC	G	CM		F	AUC	G
NaiveBayes	-2	1	0.675	0.793	0.740	-2	1	0.675	0.793	0.740	-21	16	0.675	0.793	0.740
	2	0				2	0				44	0			
JRip	-2	1	0.731	0.808	0.791	-10	5	0.731	0.808	0.791	-71	34	0.731	0.808	0.791
	2	0				10	0				64	0			
PART	-5	5	0.737	0.838	0.795	-2	10	0.711	0.831	0.776	-15	33	0.711	0.825	0.776
	7	0				27	0				80	0			
J4.8	-10	4	0.667	0.738	0.718	-5	4	0.667	0.738	0.718	-98	55	0.683	0.753	0.752
	13	0				18	0				81	0			
	AUC Search														
	Greedy					**Simulated Annealing**					**Genetic Algorithm**				
Classifier	CM		F	AUC	G	CM		F	AUC	G	CM		F	AUC	G
NaiveBayes	-8	9	0.658	0.805	0.733	0	30	0.317	0.844	0.437	0	0	0.517	0.500	0.000
	9	0				1	0				0	0			
JRip	-2	1	0.731	0.808	0.791	-16	8	0.731	0.808	0.791	-40	37	0.735	0.815	0.800
	2	0				16	0				90	0			
PART	-3	3	0.676	0.836	0.749	-6	16	0.681	0.833	0.753	-79	35	0.702	0.809	0.762
	4	0				19	0				64	0			
J4.8	-2	7	0.632	0.754	0.712	-8	13	0.632	0.754	0.712	-73	79	0.662	0.789	0.738
	7	0				9	0				56	0			
	G-Mean Search														
	Greedy					**Simulated Annealing**					**Genetic Algorithm**				
Classifier	CM		F	AUC	G	CM		F	AUC	G	CM		F	AUC	G
NaiveBayes	-2	1	0.675	0.793	0.740	-21	7	0.675	0.792	0.740	-87	40	0.675	0.792	0.740
	2	0				8	0				79	0			
JRip	-2	1	0.731	0.808	0.791	-26	11	0.731	0.808	0.791	-59	16	0.731	0.808	0.791
	2	0				18	0				5	0			
PART	-5	5	0.737	0.838	0.795	-20	17	0.711	0.825	0.776	-39	37	0.711	0.825	0.776
	7	0				28	0				66	0			
J4.8	-4	5	0.653	0.718	0.730	-21	17	0.653	0.718	0.730	-66	35	0.653	0.718	0.730
	5	0				18	0				13	0			

One thing that can be seen immediately is that the three metrics are not necessarily correlated. In other words, a good F-Measure does not guarantee a good AUC value and vice versa. This is abundantly clear when looking at the simulated annealing results for NaiveBayes. In the F-Measure search, we find an F-Measure value 0.675 and an AUC value of 0.793. In the AUC search, we see a greatly improved AUC value at 0.844 but a very poor F-Measure of 0.317.

Another interesting trend can be seen when comparing the F-Measure results and the G-Mean results. For NaiveBayes, JRip, and PART virtually identical results were achieved. In fact, the greedy search found the same cost matrices for all three. However, a difference can be seen in the J4.8 classifier. Here the greedy approach finds a different cost matrix and the F-Measures of the greedy and simulated annealing approach are lower, while the G-Means are higher. The lone exception occurs with the genetic algorithm, where a better cost matrix was found in the F-Measure search as all three metrics are higher than the genetic algorithm with G-Mean as the evaluation metric.

Interesting also are the situations in which the search for one metric achieves better results in the other metrics. The aforementioned F-Measure/genetic algorithm/J4.8 had this occur but the same thing can be seen in the AUC/genetic algorithm/JRip search. Here not only is the AUC value improved over the other two searches but the G-Mean and the F-Measure as well.

As a further comparison of the metrics, we decided to perform the search for a cost matrix on the E. coli dataset using the AUC as our evaluation metric. The goal here is to see if more significant results can be achieved, as only the use of simulated annealing and the PART classification algorithm on class OM achieved a p-value less than 0.05 (0.02799). While there were not any drastic differences in terms of improvement, we did find significant results for class OM when using the J4.8 classification algorithm combined with both simulated annealing and the genetic algorithm. They achieved p-values of 0.0102 and 0.0221, respectively. However, the PART classifer, no longer produced any significant results.

In conclusion, our results do not show any metric being necessarily better than the rest. Our method can improve on any of them and it is us up to the user what type of results they want. If the user wants a high amount of true positives, the F-Measure seems to be the best. A potential increase in accuracy? Use the AUC measure. A balance between the two? Use the G-Mean.

6.7 Comparison of Search Methods

While we have shown that all three methods can be used to improve upon an unaltered classifer, the question still remains: which one performs best? Since this can depend on a variety of factors we decided to analyze this issue in two different ways.

1. Using the default parameters, which method determines the best average F-Measure?
2. In a restricted search space, which method finds the best cost matrix first?

6.7.1 Default Comparison

In this comparison, we shall see which of the methods performs best in terms of best average F-Measure when using the default parameters. Due to poor performance, we left out the E. coli dataset and all classification with the NaiveBayes algorithm, which with few exceptions generally was not improved.

In Tables 6.20, 6.21, and 6.22, we can see the differences (improvement) between the F-Measure averages of simulated annealing and the genetic algorithm over the greedy approach. This allows us to not only to see how much improvement occurred (if any) but also which

approach produced the best average classification. In the Pima dataset, for the three different classification algorithms, 1 had their best result with simulated annealing and 2 with the genetic algorithm. In the Yeast dataset, there are 9 combinations of classification algorithms and datasets: 5 had their best results with simulated annealing, 4 with the genetic algorithm, and 1 with the greedy approach. In the Abalone dataset, there are 18 combinations of classification algorithms and datasets: 10 had their best results with simulated annealing, 10 with the genetic algorithm, and 6 with the greedy approach. In this dataset there were many instances where the best classification was achieved by more than one method.

Table 6.20: Pima Average F-Measure Improvements over Greedy Approach

Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	**0.002**	-0.008
PART	0.013	**0.014**
J4.8	-0.003	**0.003**

Table 6.21: Yeast Average F-Measure Improvements over Greedy Approach

	Class CYT	
Classifier	Simulated Annealing	Genetic Algorithm
JRip	**0.027**	0.008
PART	**0.028**	0.025
J4.8	**0.012**	0.011
	Class EXC	
Classifier	Simulated Annealing	Genetic Algorithm
JRip	**0.106**	**0.106**
PART	0.027	**0.047**
J4.8	0.015	**0.018**
	Class ME2	
Classifier	Simulated Annealing	Genetic Algorithm
JRip	0.011	**0.037**
PART	-0.015	-0.003
J4.8	**0.048**	0.023

So in total, of the 30 combinations of classification algorithms and datasets, simulated annealing generated the best results 16 times, the genetic algorithm 16 times, and the greedy approach only 7 times. Based on these results, it would seem that both simulated annealing and the genetic algorithm perform better than the greedy approach. To see if this difference is significant, we performed the Wilcoxon signed rank test comparing both simulated annealing F-Measure results and genetic algorithm F-Measure results to the greedy results. The generated p-values can be seen in Tables 6.23, 6.24, and 6.25.

In the Pima dataset only the combination of PART and simulated annealing seem to have any significant improvement. In the Yeast dataset, 4 of the 9 combinations exhibit significant

Table 6.22: Abalone F-Measure Improvements over Greedy Approach

	Class 3			
	Simulated Annealing		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median
JRip	**0.005**	**0.024**	-0.012	0.009
PART	0.038	0.080	**0.098**	**0.095**
J4.8	0.091	**0.183**	**0.113**	**0.183**
	Class 4			
	Simulated Annealing		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median
JRip	**0.034**	**0.099**	0.029	0.056
PART	-0.010	-0.060	-0.025	-0.019
J4.8	-0.018	-0.023	-0.008	0.018
	Class 5			
	Simulated Annealing		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median
JRip	**0.010**	0.016	0.004	**0.026**
PART	-0.005	0.003	-0.002	**0.021**
J4.8	0.003	-0.010	**0.018**	**0.006**
	Class 6			
	Simulated Annealing		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median
JRip	0.035	**0.047**	**0.037**	0.040
PART	**0.011**	**0.013**	0.008	0.007
J4.8	**0.003**	**0.006**	-0.017	-0.008
	Class 7			
	Simulated Annealing		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median
JRip	**0.013**	0.005	0.009	**0.006**
PART	**0.006**	-0.003	0.001	-0.011
J4.8	-0.001	**0.008**	0.000	0.002
	Class 8			
	Simulated Annealing		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median
JRip	0.000	0.000	0.000	0.000
PART	0.007	0.020	**0.016**	**0.030**
J4.8	0.013	0.002	**0.016**	**0.013**
	Class 9			
	Simulated Annealing		**Genetic Algorithm**	
Classifier	Average	Median	Average	Median
JRip	0.000	0.000	0.000	0.000
PART	0.006	0.009	**0.012**	**0.011**
J4.8	**0.008**	**0.001**	-0.007	-0.006

p-values with simulated annealing and 5 with the genetic algorithm. Finally, in the abalone dataset, of the 18 combinations, 9 exhibit significant improvement with simulated annealing and 5 with the genetic algorithm.

If we further take into consideration that using the default parameters, simulated annealing generates 178 classification models and the genetic algorithm generates 586, it is clear that using the default parameters, simulated annealing achieves the best results.

Table 6.23: Pima Wilcoxon Signed Rank Test p-values of Significance of Improvement over Greedy Approach

Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.29628	0.737957
PART	0.031428	0.17288
J4.8	0.570786	0.919472

Table 6.24: Yeast Wilcoxon Signed Rank Test p-values of Significance of Improvement over Greedy Approach

	Class CYT	
Classifier	**Simulated Annealing**	**Genetic Algorithm**
NaiveBayes	0.3141	0.3896
JRip	0.0172	0.2457
PART	0.0067	0.0133
J4.8	0.1059	0.0884
	Class EXC	
Classifier	**Simulated Annealing**	**Genetic Algorithm**
NaiveBayes	0.0221	0.0221
JRip	0.0439	0.0439
PART	0.3511	0.3324
J4.8	0.2002	0.1059
	Class ME2	
Classifier	**Simulated Annealing**	**Genetic Algorithm**
NaiveBayes	0.0221	0.0439
JRip	0.2457	0.0352
PART	0.5708	0.5507
J4.8	0.1481	0.2300

6.7.2 Restricted Search Space

As can be seen in the previous section, the greedy approach generally generates the worst average F-Measure, even if it does often improve upon an unaltered classifier. But as can be seen in the generated cost matrices in the metric comparison, the values for the found cost matrices are typically smaller than those of simulated annealing and the genetic algorithm. This is due to the fact that the search space is much larger for simulated annealing and the genetic algorithm. The greedy search starts with an initial cost matrix initialized to 0

Table 6.25: Abalone Wilcoxon Signed Rank Test p-values of Significance of Improvement over Greedy Approach

Class 3		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.2457	0.3702
PART	0.0067	0.0043
J4.8	0.0117	0.0117
Class 4		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.1258	0.2002
PART	0.4292	0.7380
J4.8	0.5907	0.5305
Class 5		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.1366	0.4093
PART	0.3511	0.4493
J4.8	0.2620	0.2963
Class 6		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.0043	0.0067
PART	0.0280	0.2300
J4.8	0.0884	0.7543
Class 7		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.0221	0.0221
PART	0.0969	0.3702
J4.8	0.3896	0.4898
Class 8		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.3141	0.5305
PART	0.0314	0.0884
J4.8	0.0314	0.0314
Class 9		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
JRip	0.3141	0.3141
PART	0.0488	0.1258
J4.8	0.0439	0.8398
Class 10		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
PART	0.0117	0.7543
J4.8	0.0488	0.2963
Class 11		
Classifier	**Simulated Annealing**	**Genetic Algorithm**
PART	0.0133	0.0393
J4.8	0.0600	0.0314

for all rewards/punishments and incrementally searches for a better cost matrix. Simulated annealing also starts with an initial cost matrix, but is capable of making much larger jumps. The genetic algorithm starts with an initial population of random cost matrices, with values between 0 and 100 (-100 for true positive rewards). So in order for a fair comparison to occur, we need to restrict the search space.

In essence our restricted search space handicaps simulated annealing and the genetic algorithm, bringing them down to the greedy approaches level. A maximum reward/punishment of 10 was set for the simulated annealing method and the neighbor function could only increment/decrement within 5. The genetic algorithm was also limited to values between 0 and 10 (-10 for true positives) with the additional restraint that the population was of size 7. This population is then equal to the amount of new cost matrices created with each iteration of the greedy approach. The greedy method was kept the same.

For data we used the Pima dataset as it is the only binary dataset (at least without any alterations). We then created one split of the data into training and test sets and did an exhaustive search for the best cost matrices for each classifier within the range of 0 to 10 for false negative/positive punishments and 0 to -10 for true positive rewards. A total of 1331 cost matrices were tried out for each classifier. Those cost matrices with the best F-Measure were then kept track of for each classifier: for NaiveBayes there were 13 (F-Measure: 0.625), for JRip there were 13 (F-Measure: 0.658), for PART there were 15 (F-Measure: 0.694), and for J4.8 there were 10 (F-Measure: 0.655).

Given these F-Measures, we can then adjust the search methods to stop once they have reached this F-Measure for the given classification algorithm. In order to determine which algorithm performs the fastest, we selected as our metric the number of models built. Since the training of a classification algorithm through MetaCost, takes far longer than any comparison operations or array swaps that the search algorithms use, it makes sense to use this as a metric.

One last issue that needed to be resolved, is the pseudo-random nature of both simulated annealing and the genetic algorithm. While the default parameters we selected were chosen because at this point the final classification models had highly similar results, this aspect is lost in this limited search space. As such, we ran both simulated annealing and the genetic algorithm 1000 times a piece.

Table 6.26: Pima Search Comparison

	Greedy	**Simulated Annealing**				**Genetic Algorithm**			
Classifier	Count	Avg	Median	Min	Max	Avg	Median	Min	Max
NaiveBayes	22	52.219	37	2	178	40.347	23	13	187
JRip	22	21.1	15.5	2	178	38.0475	22	13	182
PART	29	21.14	15	2	176	35.71	21	13	180
J48	50	128.202	178	3	178	90.385	72	13	189

The model counts of the different search methods can be seen in Table 6.26. The difficulty here lies with the fact that both simulated annealing and the genetic algorithm have highly

random results within this limited search space. We limited simulated annealing to the default temperature values and limited the genetic algorithm to 25 iterations. So in those instances where the maximum is reached, it is likely the algorithm did not even find the desired F-Measure. On the other hand, both are capable of finding a good result efficiently. Simulated annealing can find the desired F-Measure in the first neighbor jump and the genetic algorithm can find one in the first generation of offspring (hence it must create 13 models: 7 for the initial population and 6 for the offspring generation). In terms of averages, simulated annealing performs better with JRip and PART than the greedy approach and its median values are lower as well. The genetic algorithm has only median values in JRip and PART that perform as good or better than the greedy approach. It however produces better averages and medians than simulated annealing in NaiveBayes and J4.8.

Further breakdowns of these results can be seen in Figures 6.1, 6.2, 6.3, and 6.4. In these histograms, the x-axis represents the models created in the search and the y-axis gives the frequency of how many times the search algorithm created that many models to find the desired F-Measure. As can be seen in all figures, while there is a great amount of variance in terms of models created, both methods peak before they reach the number of models created by the greedy approach.

Overall, the greedy approach performs better than the averages and medians of both other approaches with NaiveBayes and J4.8. It also comes with the advantage that it will always take that amount of models to find this solution. Since no steps are random, it will always produce the same results.

6.8 Comparison to Oversampling/Undersampling

As detailed in our literature review, oversampling and undersampling are commonly used to deal with the rare-class problem. Put simply, in oversampling the minority data is increased and in undersampling the majority data is decreased. In their study of the class imbalance problem, Japkowicz et al. [28] had the best results with "oversampling with replacement." So in order to test the effectiveness of our approach, we implemented both oversampling and undersampling with replacement on the yeast dataset to see if better results could be achieved.

Implemented as part of our classification suite, our versions of oversampling and undersampling randomly select what data members to reuse/remove. In addition, our implementation allows us to specify how much to oversample/undersample. This is done because as Japkowicz et al. [28] demonstrated, a 50/50 ratio between the positive and negative class is not necessary to see improved results. So our method allows us to specify the amount of data to add/subtract as a percentage of the difference between minority and majority class. So 0 would equal no change, and 1 would lead to a completely balanced dataset. To further explain this, assume we had 100 minority class members and 300 majority class members.

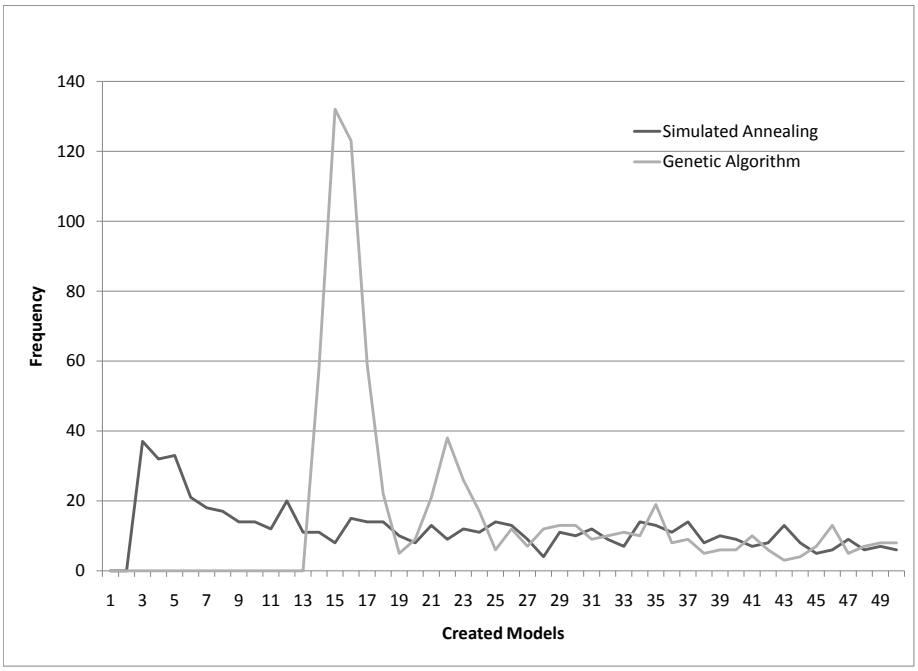

Figure 6.1: NaiveBayes Histogram Of Models Created in Cost Matrix Search

Table 6.27: Yeast Oversampling/Cost Matrix Search Average F-Measures

					Class CYT				
			Oversampling				**Cost Matrix Optimization**		
Classifier	0	0.2	0.4	0.6	0.8	1	Greedy	SA	GA
NaiveBayes	0.479	0.517	0.533	0.540	0.540	**0.545**	0.526	0.528	0.526
JRip	0.437	0.491	0.494	0.501	0.508	0.528	0.523	**0.550**	0.531
PART	0.362	0.511	0.525	0.508	0.514	0.526	0.528	**0.557**	0.553
J48	0.462	0.502	0.500	0.512	0.498	0.509	0.537	**0.549**	0.548
					Class EXC				
			Oversampling				**Cost Matrix Optimization**		
Classifier	0	0.2	0.4	0.6	0.8	1	Greedy	SA	GA
NaiveBayes	**0.459**	0.339	0.319	0.298	0.298	0.293	0.368	0.385	0.395
JRip	0.442	0.366	0.409	0.354	0.353	0.383	0.416	**0.522**	**0.522**
PART	0.320	0.400	0.330	0.341	0.322	0.358	0.433	0.460	**0.480**
J48	0.304	0.351	0.400	0.303	0.343	0.319	0.461	0.476	**0.479**
					Class ME2				
			Oversampling				**Cost Matrix Optimization**		
Classifier	0	0.2	0.4	0.6	0.8	1	Greedy	SA	GA
NaiveBayes	0.318	**0.359**	0.324	0.299	0.284	0.264	0.321	0.339	0.337
JRip	0.398	0.310	0.395	0.376	0.344	0.391	0.441	0.453	**0.479**
PART	0.268	0.247	0.350	0.332	0.329	0.353	**0.441**	0.425	0.437
J48	0.378	0.340	0.344	0.415	0.418	0.384	0.416	**0.464**	0.439

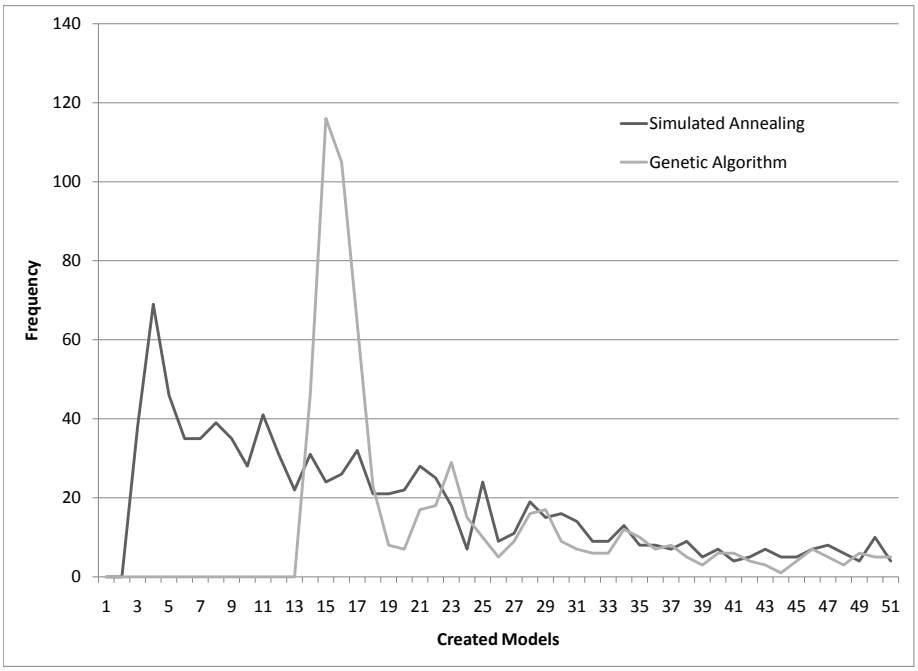

Figure 6.2: JRip Histogram of Models Created in Cost Matrix Search

Table 6.28: Yeast Undersampling/Cost Matrix Search Average F-Measures

				Class CYT					
			Undersampling				**Cost Matrix Optimization**		
Classifier	0	0.2	0.4	0.6	0.8	1	Greedy	SA	GA
NaiveBayes	0.479	0.506	0.520	**0.536**	0.540	0.546	0.526	0.528	0.526
JRip	0.437	0.427	0.473	0.487	0.505	0.513	0.523	**0.550**	0.531
PART	0.362	0.432	0.482	0.524	0.510	0.529	0.528	**0.557**	0.553
J48	0.462	0.510	0.514	0.504	**0.558**	0.532	0.537	0.549	0.548
				Class EXC					
			Undersampling				**Cost Matrix Optimization**		
Classifier	0	0.2	0.4	0.6	0.8	1	Greedy	SA	GA
NaiveBayes	**0.459**	0.412	0.366	0.367	0.333	0.276	0.368	0.385	0.395
JRip	0.442	0.477	0.476	0.456	0.345	0.263	0.416	**0.522**	**0.522**
PART	0.320	**0.494**	0.465	0.407	0.354	0.170	0.433	0.460	0.480
J48	0.304	0.478	**0.487**	0.404	0.385	0.180	0.461	0.476	0.479
				Class ME2					
			Undersampling				**Cost Matrix Optimization**		
Classifier	0	0.2	0.4	0.6	0.8	1	Greedy	SA	GA
NaiveBayes	0.318	0.318	0.326	0.344	**0.345**	0.241	0.321	0.339	0.337
JRip	0.398	0.476	0.441	0.424	0.348	0.268	0.441	0.453	**0.479**
PART	0.268	0.333	0.295	0.377	0.380	0.244	**0.441**	0.425	0.437
J48	0.378	0.432	0.412	0.355	0.322	0.233	0.416	**0.464**	0.439

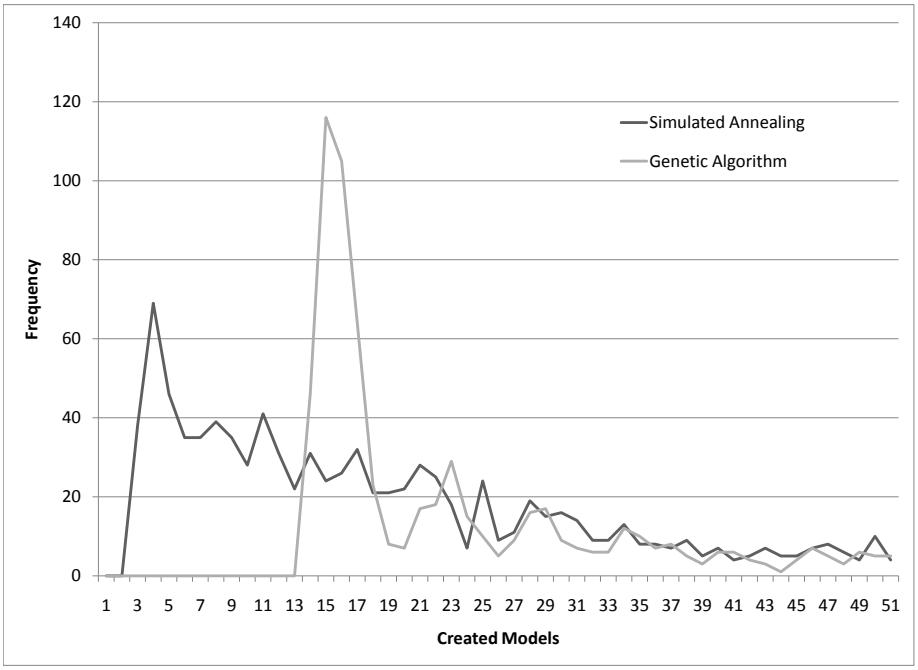

Figure 6.3: PART Histogram of Models Created in Cost Matrix Search

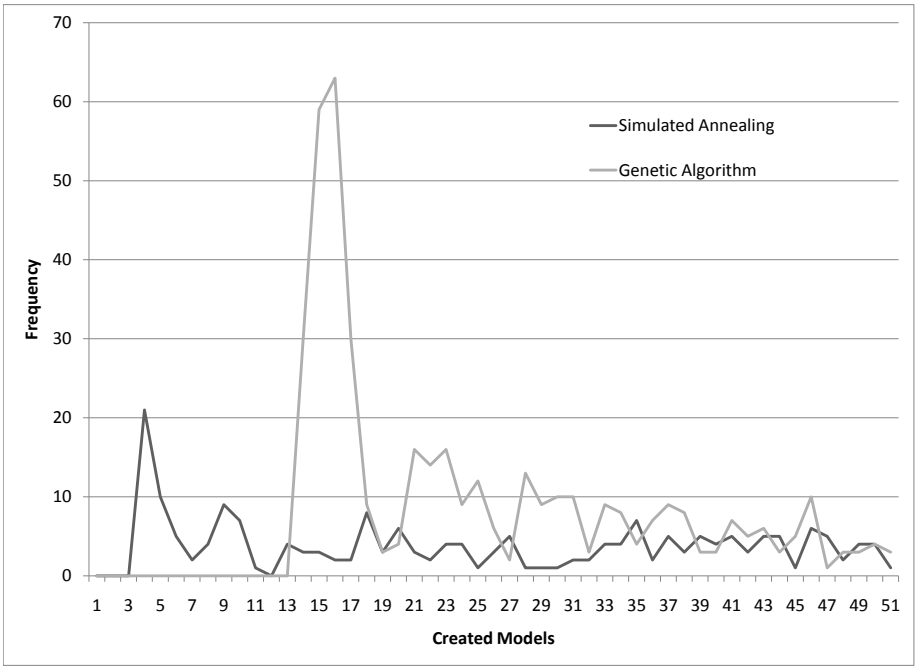

Figure 6.4: J4.8 Histogram of Models Created in Cost Matrix Search

If we oversample by 0.5, we would then have 200 minority class members. If instead we undersampled by 0.5, we would have 200 majority class members.

In Tables 6.27 and 6.28, we can see the results achieved by using these techniques in comparison to our own methods. In addition, we also see the unaltered classifiers (oversampling/undersampling of 0 percent).

Looking at oversampling, we see that good results are achieved with the NaiveBayes classifier, but not with any other classification algorithm. For both class CYT and class ME2, the best average F-Measure is achieved by oversampling. However, it is at different percentage points. While NaiveBayes is greatly improved in CYT when using 100 percent oversampling, it is actually worse when oversampling by 100 percent in class ME2. Our methods are consistently good at increasing the F-Measure.

Looking at undersampling in Table 6.28, we see that undersampling performs better than oversampling in terms of finding better F-Measures. In class CYT, we find better results with NaiveBayes and J4.8 In class EXC, the only classification algorithm that performs better with our methods is JRip. Finally, in class ME2, we see only improvement in NaiveBayes. However, despite achieving the best results with many classification algorithms, we again have an issue with inconsistency. While undersampling is capable of increasing average F-Measures, it is also capable of lowering them significantly. In addition, percentages that work for one classification algorithm may not work for others as well. 0.8 percent undersampling may work well for NaiveBayes in the ME2 class, but it creates worse F-Measures in both JRip and J4.8.

6.9 The Nature of Cost Matrices

Looking at the cost matrices we showed in Table 6.19, we see that these potentially optimal cost matrices are very different from one another. Moreover, many cost matrices seem to produce the same performance (at least in terms of metrics) but have different values for punishments/rewards. During our initial using of cost matrix optimization to classify gene conversions using the greedy approach [34], we saw certain trends among the generated cost matrices that included false negative punishments being higher than all other values. This does not seem to be the case in the cost matrices shown in Table 6.19, as cost matrices can have higher values for either false negatives or false positives, and even true positives if we look at the absolute values. They can even seem to perform the same despite having different higher values. Case in point the G-Mean search for the best cost matrix with J4.8. Simulated annealing produces a cost matrix in which the false negative punishment is higher than the false positive ($18 > 17$) while the genetic algorithm produces one that has a higher false positive value ($35 > 13$). So how does one make sense of such contradictory behavior? In order to do so, we must look at the MetaCost algorithm and how it uses a cost matrix.

The majority of the MetaCost algorithm is devoted to the determining of probabilities that a

data member belongs to each class as determined by the committee of classification models that were generated through bagging. The cost matrix finally enters into play when the conditional risk function is addressed to determine what class label should be given to the current data member x:

$$\text{Let } x\text{'s class} = argmin_i \sum_j P(j|x)C_{i,j} \tag{6.1}$$

Since we only have two classes to chose from (due to the binary nature of our datasets), this can be broken down into two functions:

$$P(0|x)C_{0,0} + P(1|x)C_{0,1} \tag{6.2}$$

$$P(0|x)C_{1,0} + P(1|x)C_{1,1} \tag{6.3}$$

Equation 6.2 shows the conditional risk value of setting the class of x to be 0, and Equation 6.3 shows the conditional risk of setting the class to be 1. We can further simplify the conditional risk of Equation 6.3 by eliminating the $P(1|x)C_{1,1}$ value since $C_{1,1}$ is always 0. So we are then left with two situations that determine the class label:

$$P(0|x)C_{0,0} + P(1|x)C_{0,1} < P(0|x)C_{1,0} \longrightarrow x\text{'s class} = 0 \tag{6.4}$$

$$P(0|x)C_{0,0} + P(1|x)C_{0,1} > P(0|x)C_{1,0} \longrightarrow x\text{'s class} = 1 \tag{6.5}$$

Further simplification leads to

$$C_{0,0} + \frac{P(1|x)}{P(0/x)}C_{0,1} < or > C_{1,0} \tag{6.6}$$

One thing we can immediately see through breaking down these equations, multiplying a cost matrix by a scalar value will not change the outcomes of these functions. This is why in Table 6.19, the following cost matrices achieve the same results with JRip:

$$C1 = \begin{bmatrix} -2 & 1 \\ 2 & 0 \end{bmatrix} \tag{6.7}$$

$$C2 = \begin{bmatrix} -10 & 5 \\ 10 & 0 \end{bmatrix} \tag{6.8}$$

$$C3 = \begin{bmatrix} -16 & 8 \\ 16 & 0 \end{bmatrix} \tag{6.9}$$

$C2$ is 5 * $C1$ and $C3$ is 8 * $C1$. And as we can see from the conditional risk equations, there is no reason the generated classification models should be any different either.

But how then do the following two cost matrices produce highly similar results?

$$C4 = \begin{bmatrix} -21 & 17 \\ 18 & 0 \end{bmatrix} \tag{6.10}$$

$$C5 = \begin{bmatrix} -66 & 35 \\ 13 & 0 \end{bmatrix} \tag{6.11}$$

This can be better understood in equation 6.6. Let r denote the ratio of relative class probabilities, $\frac{P(1|x)}{P(0|x)}$, and w denote the ratio of relative risks, where

$$w = \frac{rC_{0,1} + |C_{0,0}|}{C_{1,0}} \tag{6.12}$$

Because the only thing that is changed is the cost matrix, r behaves essentially as a constant, it is the ratio of relative risks w that determines the performance of the classification based on the current cost matrix. Therefore, various cost matrices with the same w will have exactly the same performance (e.g. yield the same F-measures), regardless of our search methods, and the classifiers thus produced are the same.

The cost matrices $C1$, $C2$, and $C3$ all produce the same ratio value of 4. If we plug in the values for cost matrices $C4$ and $C5$ we get the values 2.026 and 2.059. While these values are not identical, they are quite similar and similar enough to achieve the same performance metrics. It is however quite possible that given more data the resulting classification algorithms of these two cost matrices might perform differently.

Ultimately, what is important for the determining of a cost matrix is the ratio in which the three values of the cost matrix are. And our approaches effectively find good ratios for classifying rare-class datasets.

Chapter 7

Conclusions

7.1 Insights into the rare-class problem solution

An effective solution for any choice of metric. In this paper, we have detailed our solution to the rare-class problem, namely to find a cost matrix that is optimal for a given classification algorithm and a given set of data. We go about this search three different ways (greedy, simulated annealing, and genetic algorithm), and all have been shown to be quite effective, although to varying degrees. All three methods can be used with different metrics and improve upon all, leaving it up to the user what type of metric fits the results they are looking for. For our experiments, we chose the F-Measure because it generated an increase in true positives, without increasing the number of false positives too much.

The poor performance of NaiveBayes. As detailed in our results, NaiveBayes performed quite poorly. In very few situations did it achieve better performance in terms of F-Measures and it mostly resulted in decreased F-Measures. The likely reason for this is the fact that MetaCost uses bagging for its predictions. When he introduces the concept of bagging, Breiman [3] states that a stable classifier will perform quite poorly using this method. As it is important that altering the learning set achieve differing results with the classification algorithm, NaiveBayes performs too consistently in order to be improved upon. The other classifiers, however, perform quite well on all the other datasets, with the one possible exception of E. coli.

The E. coli dataset. Overall, our methods perform quite inconsistently in improving on the unaltered classifiers in the E. coli dataset. Some classification algorithms achieve worse results in terms of average F-Measures and only one classification algorithm achieves significant results for one dataset. The explanation for this is that the E. coli dataset is quite small, and, as such, representative data is in too limited a quantity. The yeast dataset

consists of highly similar data (many of the attributes are the same and both are determining protein localization sites) and we are able to improve classification due in no small part that instead of 336 data members, we have 1484.

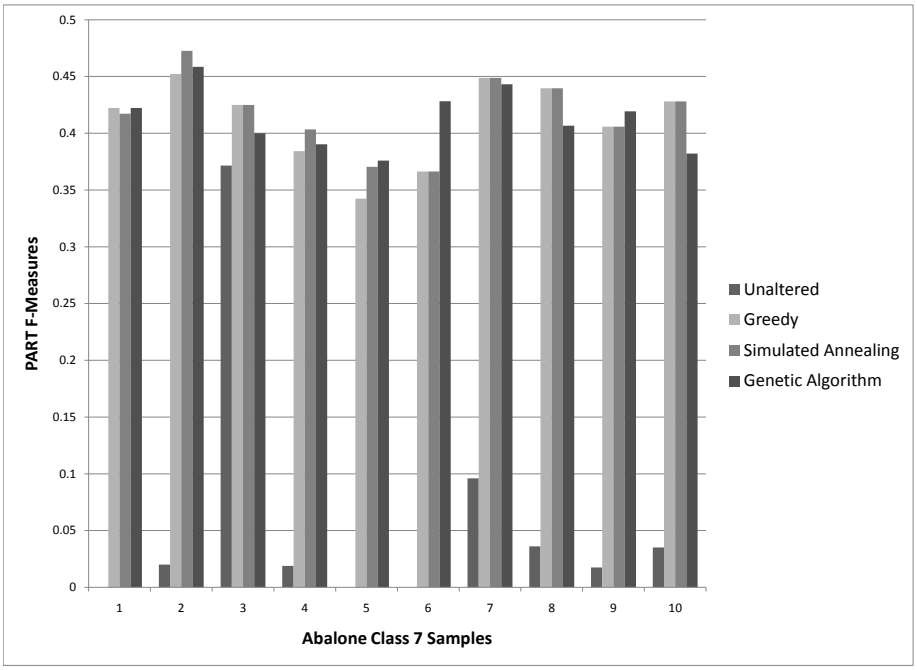

Figure 7.1: Bar Plot of PART F-Measures Generated by the Cost Matrix Search Methods Across all Samples for Abalone Class 7

Great improvements with the PART classification algorithm. The other three classifiers are generally improved upon and in most cases achieve significant improvement based on the Wilcoxon Ranked Sign Test. Overall it would seem that the approach improves upon PART the best. It is consistently improved, achieves in many situations the highest F-Measures, and shows the most increase in terms of F-Measures. For instance, in Abalone Class 7, it improves from an average F-Measure of 0.060 in the unaltered state to an average of 0.418 with simulated annealing (and above 0.4 in both other methods as well). This improvement can be seen in Figure 7.1.

Best Search Method: Simulated Annealing. As we have shown in the results section, simulated annealing seems to be the best of the three search methods. While the genetic algorithm can on occasion produce better results, simulated annealing produces its results

more efficiently. And while the greedy approach has been shown to achieve good results more efficiently and consistently, simulated annealing has also shown to generate more significant improvement in terms of F-Measures. So when given a rare-class dataset, a user should start with the simulated annealing approach as this will give him/her the highest likelihood of success. However, the other search methods should also be experimented with as we have shown that in some situations better results can be achieved with either one.

A continuation of the Black Box approach. In his description of the MetaCost algorithm, Domingos [13] stated that it was of importance that the algorithm have a "black box approach" so that the user can easily adjust the learning of the training data through a cost matrix without being bogged down with the details. With our search methods, we are able to continue this black box approach, even taking it one step further due to the fact that the user does not even need to generate the cost matrices themselves. As can be seen in our pseudocode and the implementations, minimal adjustments need to be made by the user in order to generate an optimized classification model. We do however allow for the user to make adjustments to the search and in theory the user can even start with a different initial cost matrix (except with the genetic algorithm).

A good solution for any dataset. In addition to this black box approach, there is nothing inherent in any of the methods that makes it only suited for biological classification. As rare-class problems are common within biological data and biological problems such as identifying gene conversions are commonplace for the author, the focus was on classifying biological datasets. However there is no reason the methods should not work on other classification problems, given a sufficient amount of data and numerical data attributes.

Attempts at improving the greedy search. As mentioned in the results section, part of the reason the greedy approach does not perform as well as the others is that it has a limited search range. When we created a similar limit in the other two methods, the greedy approach performed in some ways better than the others. However, we were unable to reverse this situation by expanding the search range of the greedy approach. One idea is based on our analysis of the nature of cost matrices. Since a cost matrix multiplied by a scalar produces the same results, it was conceivable that we could multiple the overall best cost matrix and achieve a "scalar jump." This could be done as soon as improvement has stopped and would allow for the search to continue in a new search space. Unfortunately, preliminary results revealed no improvement, and difficulties such as the size of the scalar still need to be resolved.

Attempts at improving the MetaCost algorithm. Another part where we did not see improvement is in an adjustment made to the MetaCost algorithm itself. Since boosting is considered a better approach than bagging [41], it would seem that better overall results could

be achieved by replacing the determining of probabilities in MetaCost through boosting. A paper posited this very idea, claiming better results can be obtained [54]. In our implementation, we created a modified version of MetaCost using the AdaBoost algorithm [22] to determine the probabilities. However, no improvement over the regular MetaCost algorithm was noticed. The MetaCost with AdaBoost algorithm however still remains as part of our implementation.

7.2 Future Work

Rare-class classification suite. As stated, our approach was implemented as a suite of java classes. It is our goal to be able to distribute this and thereby to provide other researchers with this solution. Additional design goals then include creating an actual user interface and determining if our solution can be somehow added to the weka open source project. Furthermore, since our solutions were implemented as java classes, we can create new classes that improve upon the individual components.

Improving simulated annealing. The simulated annealing approach we use is basic and is not all that different from the original proposed method [32, 4]. However, many advances have been made to the algorithm itself and to the way it implements the cooling rule. Improved results can potentially be achieved by combining simulated annealing with a tabu search [43] or using a chaotic distribution to create new solutions, also referred to as chaos simulated annealing [26]. In addition, using an adaptive cooling schedule [1] can also improve upon results. Overall, it will be interesting to see if better performance or more efficient performance can be achieved.

Improving the genetic algorithm. The genetic algorithm can also be improved upon to see if we can achieve better results. Minor changes such as altering mutation rates at different loci [20], or by utilizing roulette wheel selection [2] or tournament selection [46]. In addition, JGAP has a variety of different selection methods and variation operators that can be experimented with.

Improving the classification algorithms. As with the search methods, we used basic classification algorithms with their default parameters to show that our methods were generating the improvement and not the classification algorithms themselves. As such, we can see if better results can be achieved by adjusting the parameters for these classification algorithms, tweaking them to achieve higher performance in their unaltered states. We can then see if our methods still achieve improvement when the classification algorithm has been essentially tailored for the current dataset.

In addition, we can see if better results can be achieved if we use a classification algorithm that is less likely to be affected by rare-class data. Random forests have been found to be effective in identifying DNA binding sites [59], while Support Vector Machines (SVMs) have been demonstrated to be effective in promoter region recognition of DNA sequences [11] and transcription factor binding site predictions [51]. All of these are rare-class problems and it would be interesting to see if the combination of random forests or SVMs with our methods achieve better results.

However, there does exist the possibility that these classification algorithms will exhibit similar behavior to NaiveBayes, which generally performed best when unaltered but rarely improved when using our methods. These other methods could be "too stable" to effectively utilize boosting in the MetaCost algorithm. If this is the case, they will at least make for good counter-examples for dealing with rare-class data. Right now we have shown our approaches to be more consistent in improving classification than simple oversampling and undersampling. So it will be interesting to see how our method does against these classification algorithms or other methods such as SMOTE [7].

Bibliography

[1] Nader Azizi and Saeed Zolfaghari. Adaptive temperature control for simulated annealing: A comparative study. *Comput. Oper. Res.*, 31(14):2439–2451, 2004.

[2] J.E. Baker. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 14–21, 1987.

[3] Leo Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

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

[5] Nitesh V. Chawla, David A. Ciestak, Lawrence O. Hall, and Ajay Joshi. Automatically countering imbalance and its empirical relationship to cost. *Data Mining and Knowledge Discovery*, 2008.

[6] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kolcz. Editorial: Special issue on learning from imbalanced data sets. *SIGKDD Explorations*, 6(1):1–6, 2004.

[7] NV Chawla, KW Bowyer, LO Hall, and WP Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[8] NV Chawla, A Lazarevic, LO Hall, and KW Bowyer. SMOTEBoost: Improving prediction of the minority class in boosting. *Knowledge Discovery In Databases: Pkdd 2003, Proceedings*, 2838:107–119, 2003.

[9] Jian-Min Chen, David N. Cooper, Nadia Chuzhanova, Claude Ferec, and George P. Patrinos. Gene conversion: Mechanisms, evolution and human disease. *Nature Reviews Genetics*, 8:762–775, 2007.

[10] William W. Cohen. Fast effective rule induction. *Machine Learning: Proceedings of the Twelfth International Conference (ML95)*, 1995.

[11] R. Damasevicius. Analysis of binary feature mapping rules for promoter recognition in imbalanced DNA sequence datasets using Support Vector Machines. *Intelligent Systems, 2008. IS '08. 4th International IEEE Conference*, pages 11–20–11–25, 2008.

[12] Jesse Davis and Mark Goadrich. The Relationship Between Precison-Recall and ROC Curves. *Proceedings of the 23rd International Conference on Machine Learning*, 2006.

[13] Pedro Domingos. MetaCost: A General Method for Making Classifiers Cost-Sensitive. *Advances in Neural Networks, International Journal of Pattern Recognition and Artificial Intelligence*, pages 155–164, 1999.

[14] Chris Drummond and Robert C. Holte. Cost curves: An improved method for visualizing classifier performance. *Machine Learning*, 65:95–130, 2006.

[15] Charles Elkan. The foundations of cost-sensitive learning. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, 2001.

[16] Klaus Meffert et al. JGAP - Java Genetic Algorithms and Genetic Programming Package. url: http://jgap.sf.net.

[17] Wei Fan, Salvatore J. Stolfo, Junxin Zhang, and Philip K. Chan. AdaCost: Misclassification Cost-sensitive Boosting. *Proceedings of the 16th International Conference on Machine Learning*, 1999.

[18] T Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27:861–874, 2006.

[19] T Fawcett and F Provost. Adaptive fraud detection. *Data Mining and Knowledge Discovery*, 1:291–316, 1997.

[20] T.C. Fogarty. Varying the probability of mutation in the genetic algorithm. *Proceedings of the 3rd International Conference in Genetic Algorithms*, pages 104–109, 1989.

[21] Eibe Frank and Ian H. Witten. Generating accurate rules sets without global optimization. *Fifteenth International Conference on Machine Learning*, 1998.

[22] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55:119–139, 1997.

[23] H Han, WY Wang, and BH Mao. Borderline-SMOTE: A new over-sampling method in imbalanced data sets learning. *Advances In Intelligent Computing, Pt 1, Proceedings*, 3644:878–887, 2005.

[24] J.H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[25] Paul Horton and Kenta Nakai. A probabilistic classification system for predicting the cellular localization sites of proteins. In *Proceeding of the Fourth International Conference on Intelligent Systems for Molecular Biology*, pages 109–115, 1996.

[26] Mingjun J and Huanwen T. Application of chaos in simulated annealing. *Chaos, Solitons & Fractals*, 21:933–944, 2004.

[27] I. B. Jakobsen, S. R. Wilson, and S. Easteal. The partition matrix: Exploring variable phylogenetic signals along nucleotide sequence alignments. *Molecular Biology and Evolution*, 14(5):474–484, 1997.

[28] N Japkowicz and Shaju Stephen. The class imbalance problem: A systematic study. *Intelligent Data Analysis*, 6(5):429–450, 2002.

[29] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.

[30] Mahesh V. Joshi, Vipin Kumar, and Ramesh C. Agarwal. Evaluating boosting algorithms to classify rare classes: Comparison and improvements. *Proceedings of the First IEEE International Conference on Data Mining (ICDM'01)*, 2001.

[31] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc Natl Acad Sci U S A*, 87(6):2264–8, 1990.

[32] S Kirkpatrick, CD Gelatt, and MP Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[33] M. Kubat, R. Holte, and S. Matwin. Machine learning for the detection of oil spills in satellite radar images. *Machine Learning*, 30:195–215, 1998.

[34] Mark Lawson, Lenwood Heath, Naren Ramakrishnan, and Liqing Zhang. Using cost-sensitive learning to determine gene conversions. *Advanced Intelligent Computing Theories and Applications. With Aspects of Artificial Intelligence*, 5227:1030–1038, 2008.

[35] Mark J. Lawson and Liqing Zhang. Sexy gene conversions: Locating gene conversions on the X-chromosome. *Nucl. Acids Res.*, page gkp421, 2009.

[36] G. Marais. Biased gene conversion: Implications for genome and sex evolution. *Trends Genetics*, 19(6):330–8, 2003.

[37] PM Murphy and MJ Pazzani. Exporing the decision forest: An empirical investigation of Occam's razor in decision tree induction. *Journal of Artificial Intelligence Research*, 1994.

[38] Warwick J. Nash and Tasmania. *The Population biology of abalone (Haliotis species) in Tasmania. 1, Blacklip abalone (H. rubra) from the north coast and the islands of Bass Strait / Warwick J. Nash ... [et al.].* Sea Fisheries Division, Dept. of Primary Industry and Fisheries, Tasmania, Hobart, 1994.

[39] D. Newman, S. Hettich, C. Blake, and C. Merz. UC Irvine Machine Learning Repository. *http://archive.ics.uci.edu/ml/*, 1998.

[40] D.T Pham and D. Karaboga. *Intelligent Optimisation Techniques*. Springer, 2000.

[41] R. Polikar. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, 2006.

[42] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[43] Swarnkar R. and Tiwari MK. Modeling machine loading problem of FMSs and its solution methodology using a hybrid tabu search and simulated annealing-based heuristic approach. *Robot Computer-Integrated Manufacturing*, 20:199–209, 2004.

[44] Colin R. Reeves and Jonathan E. Rowe. *Genetic Algorithms - Principles and Perspectives*. Kluwer Academic Publishers, 2003.

[45] P Riddle, R Segal, and O Etzioni. Representation design and brute-force induction in a Boeing manufacturing domain. *Applied Artificial Intelligence*, 8:125–147, 1994.

[46] E. Saliby. Descriptive Sampling: a better approach to Monte Carlo simulation. *Journal of Operational Research Society*, 41:1133–1142, 1990.

[47] S. Sawyer. Statistical tests for detecting gene conversion. *Molecular Biology and Evolution*, 6(5):526–538, 1989.

[48] Everhart J.E. Dickson W.C. Knowler W.C. & Johannes R.S Smith, J.W. Using the ADAP learning algorithm to forecast the onset of diabetes mellitus. *Proceedings of the Symposium on Computer Applications and Medical Care. IEEE Computer Society Press.*, pages 261–265, 1988.

[49] B. Suman and P. Kumar. A survey of simulated annealing as a tool for single and multiobjective optimization. *Journal of the operational research society*, 57(10):1143–1160, 2006.

[50] Yamnin Sun, Mohamed S. Kamel, Andrew K. C. Wong, and Yang Wang. Cost-sensitive boosting for classification of imbalanced data. *Pattern Recognition*, 40:3358–3378, 2007.

[51] Yi Sun, Mark Robinson, Rod Adams, Paul Kaye, Alistair G. Rust, and Neil Davey. Uusing real-valued metaclassifiers to integrate binding site predictions. In *Proceedings of International Joint Conference on Neural Networks*, 2005.

[52] P.D. Surry and N.J Radcliffe. Inoculation to initialise evolutionary search. *Evolutionary Computing: AISB Workshop*, pages 269–285, 1996.

[53] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction To Data Mining*. Addison-Wesley, 2006.

[54] KM Ting. An empirical study of MetaCost using boosting algorithms. *Machine Learning: Ecml 2000*, 1810:413–425, 2000.

[55] Sam Waugh. *Extending and Benchmarking Cascade-Correlation*. PhD thesis, University of Tasmania, 1995.

[56] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1:80–83, 1945.

[57] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, second edition, 2005.

[58] K. Woods, C. Doss, J. Solka, C. Priebe, and P. Kegelmeyer. Comparative evaluation of pattern recognition techniques for detection of microcalcifications in mammography. *Internations Journal of Pattern Recognition and Artificial Intelligence*, 7(6):1417–1436, 1993.

[59] Hongde; Duan Xueye; Ding Yan; Wu Hongtao; Bai Yunfei; Sun Xiao Wu, Jiansheng; Liu. Prediction of DNA-binding residues in proteins from amino acid sequences using a random forest model with a hybrid feature. *Bioinformatics*, 25:30–35, 2009.

[60] Kihoon Yoon and Stephen Kwek. A data reduction approach for resolving the imbalanced data issue in functional genomics. *Neural Computing & Applications*, 16:295–306, 2007.

[61] Xing-Ming Zhao, Xin Li, Luonan Chen, and Kazuyuki Aihara. Protein classification with imbalanced data. *Proteins-Structure Function and Bioinformatics*, 70:1125–1132, 2008.