Survey of Cost-Sensitive Learning

Yiming Chen[*1], Thomas Cwalina[†1], Devansh Gupta[‡1], Rahul Pokharna[§1], Neil Steiner[¶1], and Stan Tian[‖1]

[1]Department of Electrical Engineering and Computer Science, Case Western Reserve University

December 8, 2018

Abstract

Many classification problems are considered to be example-dependent and cost-sensitive in nature, especially when real world data is utilized. Nevertheless, standard classification algorithms often fail to take the misclassification cost or the features cost into the learning process. Cost-sensitive learning algorithms have proved to be a good candidate when facing classification problems involving real world data, and the goal of this learning algorithm family is to minimize the total cost while maintaining a high generalization ability. This paper surveyed and implemented 15 classification cost-sensitive learning algorithms that optimize on either misclassification cost or feature cost or both. Apart from comparing the model accuracy and average cost, we also performed extensive Student's t-tests within the 15 models surveyed. Additionally, this paper also tried to account for the performance issue of some models and some related applications.

[*]yxc1103@case.edu
[†]tbc21@case.edu
[‡]dxg397@case.edu
[§]rkp43@case.edu
[¶]nfs15@case.edu
[‖]stan@case.edu

1 Introduction

1.1 Overview

Cost-sensitive learning describes a learning-algorithm modification that makes the algorithm better-suited to solving a learning problem. This type of modification involves introducing a hyperparameter into a targeted aspect of a learning algorithm such that the learning algorithm can better represent the real-life tradeoffs for a particular learning problem. The way in which a modification is integrated into a learning algorithm depends on the goal a designer is trying to achieve. This survey will focus specifically on modifications to learning algorithms that achieve either of the following two goals: to control the magnitude of type 1 and type 2 misclassifications; and to weight the attributes used in a dataset.

Most learning algorithms used in machine learning treat the costs of type 1 and type 2 errors as equal [1] . However, in many datasets, the real-life consequences of a type 1 error can be more severe than making a type 2 error, while in other datasets the reverse is true. For example, this is a very common problem in medical datasets that try to determine if a patient tests positive for a disease [1]. In these cases, classifying a patient as not having a disease when they do (type 2) has greater repercussions for a patient than mistakenly determining the patient has the disease (type 1) [1]. By altering the inductive bias to contain hypotheses that are geared towards one type of misclassification, a learning algorithm can create classifications that are better suited to the real-life consequences of its decisions.

With the rising use of machine-learning algorithms in industry, the computation time an algorithm requires to train and test on examples is an important factor [2]. As a result, resource constraints are placed on learning algorithms used in industry so they return classifications in a reasonable amount of time [2]. For example, in 2013, Google reported that its search engine executed approximately 5.9 billion searches each day [2]. Cost-sensitive learning can be used to improve the runtime of learning algorithms, through modifications that associate a cost parameter with each of the features in a dataset, to yield a feature- extraction cost [2]. These cost parameters give the learning algorithm flexibility in selecting features it uses to classify a particular test example, such that the algorithm meets the established resource constraint [2]. This flexibility, in turn, allows a learning algorithm to be implemented in scenarios that meet the demands of consumers [2].

The practical utility of cost-sensitive learning can be seen in real-life applications, such as the medical field. When a patient goes to the hospital to receive tests for a diagnosis, not all tests have the same cost. For example, a blood test to determine the patient's insulin levels may not be the same price as a blood test to determine their resistin levels. We can use these cost parameters to create a classification method that limits the total cost to diagnose the patient, benefitting both the hospital and the patient. In summary, cost-sensitive learning can be employed to tune models to their specific situations in which factors such as misclassification costs and data costs are paramount.

1.2 Cost Matrix

In misclassification cost problems, the cost matrix C in general misclassification cost problem is defined as:

$$C = \begin{array}{cccc|l} \text{True class 1} & \text{True class 2} & ... & \text{True class n} & \\ C_{11} & C_{12} & ... & C_{1n} & \text{Predict class 1} \\ C_{21} & C_{22} & ... & C_{2n} & \text{Predict class 2} \\ ... & ... & ... & ... & ... \\ C_{m1} & C_{m2} & ... & C_{mn} & \text{Predict class m} \end{array} \tag{1}$$

where C_{ij} stores the cost corresponding to classify an example of class j to be class i.

In binary classification problem, the matrix is defined as Equation below:

$$C = \begin{array}{cc|l} \text{True Negative} & \text{True Positive} & \\ C_{00} & C_{01} & \text{Predict Negative} \\ C_{10} & C_{11} & \text{Predict Negative} \end{array} \tag{2}$$

2 Related Works

2.1 General Overview of Research Papers

The research papers surveyed for this experiment address cost-sensitive learning with respect to either misclassification costs, feature costs, or both. Of the ten papers surveyed, six of them present algorithms that implement misclassification cost into cost-sensitive learning, three can be categorized as introducing algorithms involving cost-sensitive learning with respect to feature costs, and one proposes parameters that modify algorithms to accomplish both.

The following papers address misclassification cost within cost-sensitive learning. In *Cost-Sensitive Boosting Neural Networks for Software Defect Prediction*, three types of cost-sensitive boosted neural networks are introduced for solving the software defect problem [3]. Algorithms are made cost-sensitive by introducing a cost matrix into an aspect of the AdaBoost algorithm [3]. *On the Effectiveness of Cost-Sensitive Neural Networks for Software Defect Prediction* compares the three cost-sensitive boosted neural networks introduced in Zhengs paper, CSBNN-TM, CSBNN-WU1, and CSBNN-WU2, to other traditional learning algorithms using the NECM metric and Nemenyi diagrams [4]. *Cost-Weighted Boosting with Jittering and Over/Under Sampling* presents the JOUS-Boost algorithm as an extension of the AdaBoost algorithm, by taking into account misclassification costs, and constraining a learning algorithm from overfitting [10]. *Cost-Sensitive Classification with k-Nearest Neighbors* demonstrates modification of k-Nearest Neighbors (kNN) to include misclassification costs [7]. The two algorithms described are Direct-CS-kNN and Distance-CS-kNN [7]. *Condensed Filter Tree for Cost-Sensitive Multi-Label Classification* offers an algorithm known as condensed filter trees (CFT) for optimizing cost-sensitive multi-label classification (CSMLC) [14]. *Multiclass Classification with Filter Trees* explains the filter tree algorithm for cost-sensitive learning, which reduces a multi-labeled classification problem to binary classification [13]. *Ensemble of Example-Dependent Cost-Sensitive Decision Trees* introduced a cost-sensitive stacking, a way of combining cost-senstive base learners to generate an upper level training feature for another cost-sensitive linear classifier to be trained on [16].

Feature costs are a focus of cost-sensitive learning in the following. *A framework for cost-based feature selection* introduces selection methods that take into account the cost of each attribute: Correlation-based Feature Selection (CFS) and Minimal-Redundancy-Maximal-Relevance (mRMR) [6]. *Cost-*

Sensitive Decision Tree Learning for Feature Selection proposes a cost-sensitive decision tree by enhancing the ID3 learning algorithm to include attribute costs [5]. Performance of this algorithm is evaluated on forensic datasets [5]. *Randomly Selected Decision Tree for Cost-Sensitive Learning* is a model for reducing the total test cost of attributes in a dataset, while maintaining a high classification accuracy [7]. One paper addresses both misclassification and feature costs: *Test-Cost Sensitive Nave Bayes Classification* performs assessments to minimize the sum of misclassification and test costs [9].

2.2 Cost-Sensitive Boosted Neural Networks

Software defect predictors are learning algorithms that are used to determine whether a piece of code will likely contain any flaws. In this problem, the misclassification that arises when a defect predictor indicates that an example is defect-prone, when it is not, is more severe than the classifier indicating that an example is not defect prone, when it is [3]. This makes the software defect problem a good candidate for cost-sensitive learning with respect to misclassification costs [3]. Misclassification costs are represented using a cost matrix, where the correct classifications are given a value of one [3]. The misclassification costs are hyperparameters and can therefore be optimized for a particular dataset.

The learning algorithm that is modified to include cost-sensitive learning is a boosted neural network. Zheng uses a feed-forward neural network that has an input layer, one hidden layer, and an output layer. Each node in the input layer corresponds to one of the features in the given dataset [3]. For this particular study, the number of neurons in the hidden layer is set to eleven [3]. A single neuron in the output layer is used for binary classification, where -1 represents a defect in the software and 1 represents no defects [3]. This network also implements the backpropagation algorithm, which uses ordered derivatives to adjust the weights in the network based on the misclassification error of a given example. AdaBoost is used as the boosting algorithm, as it places greater emphasis on examples that are misclassified by adding a weight to each example [3].

Three different modifications are presented to the boosted neural-network architecture in order to make it cost-sensitive. The first modification, known as threshold moving, involves implementing the cost matrix into the final classifier of the AdaBoost algorithm [3]. In this approach, the weights do not have to be retrained if the values in the cost matrix change [3]. The other two modifications change the weight-updating step of the AdaBoost algorithm to include the cost matrix [3]. The first of these, Weight-Updating 1 removes the alpha component from the weight-updating equation [3]. However, the alpha values are still used when outputting the final classifier [3]. Weight-Updating 2, on the other hand, retains the alpha-value in the weight-updating equation [3].

The performance of these three cost-sensitive learning algorithms are compared using the following four performance measures: MR; ErrI; ErrII; and NECM. MR is defined as the total number of misclassifications over the total number of classifications [3]. ErrI and ErrII represent the type 1 and type 2 errors of a learning algorithm, respectively. NECM is a cost-sensitive performance measure that is used to measure a cost-sensitive learning algorithms misclassification cost, which is useful when trying to find a learning algorithm with the minimum misclassification cost [4]. The independent variable in each of these performance measures is the cost ratio, which is defined as the cost of the type 2 error, divided by the cost the type 1 error [3]. From these experiments, it was concluded that cost-sensitive boosted neural networks with threshold moving was the best of the three methods for software defect predictors [3].

2.3 Cost Sensitive Decision Trees

The structure of decision trees – a series of nodes which splits the data in order to separate the classes – lends itself very well to the case of cost sensitive learning. The general idea for the inclusion of cost sensitivity is to modify the function that chooses the best feature to split upon at each node to have some sort of penalty based upon the cost of the feature. There are many different functions used to decide which feature is the best to split on. In the case of information gain, the algorithm chooses the feature that most reduces the entropy or uncertainty of the class label [7]. This is with the goal of slowly lowering the entropy of the data in the tree until the data reaches zero entropy – in which all of the data contained has the same class label. An extension to this idea is gain ratio, which modifies information gain by diving by the split information of a feature [7]. The split information is a measure that is high when the feature includes many different values to split upon. By dividing gain ratio by this value, the function prevents the inclusion of features that are reducing entropy but not truly observing the underlying concept of the data but rather benefiting from the number of splits (which leads to memorization of the data).

Cost sensitivity pairs naturally with both information gain and gain ratio. To add a penalty of the cost of the feature, most methods either divide or subtract these values by the cost of a features [5]. With the addition of cost to these feature ranking methods, most include a parameter such as γ that can be used to control the effect that cost has on the selection of features. This parameter can be set so that only cost, only the information value of the feature, or any combination is between is taken into consideration when choosing a feature. An example of some of these functions that are used to select a feature is the information cost function by Nunez $\frac{2^{Gain(X,f)}-1}{(Cost(f)+1)^{\gamma}}$ [5].The should be optimized in order to find the best balance of cost reduction, while maintaining an acceptable accuracy for the specific problem. There are many ways to integrate costs into the creation of decision trees, some even including controlled random selection, such as the randomly selected decision tree algorithm [7]. In the end, the goal of every costs sensitive decision tree is to reduce overall tree cost without significantly decreasing the accuracy.

With there being many different ways to integrate cost sensitivity into decision tree creation, there are also many ways to frame cost to a real-life situation. One of these considerations is the manner that the features will be acquired in real life. Decision trees fit naturally with the idea of sequential testing, that is ordering a test after receiving the results of a previous one. In many situations, although, this sequential test ordering is not possible [5]. This can change the manner in which both the tree is constructed, and how the tree cost is evaluated. If the tree allows for sequential testing then at each node only direct ancestors are considered to have been purchased. But in the case of non-sequential testing in which all features are bought up front, a node can use a feature free of cost if it has been purchased in any level higher up in the tree. Furthermore, to calculate the cost of a tree in sequential testing, the average cost of each example as it goes through the tree is calculated to determine the cost of the model [5]. In the case of non-sequential testing, the cost of the tree is simply the sum of the costs all features included in the tree, since the features are bought for all examples at the beginning. There are many ways to frame cost sensitivity in decision trees and then many ways to integrate those costs into an algorithms node feature selection method.

2.4 Cost Sensitive Feature Selection

Cost sensitive learning algorithms attempt to isolate features to contain the most information while limiting cost. In order to assist these learning algorithms in their search, there are a set of feature selection methods that consider the cost of the features. As with normal feature selection, there are three main types of cost-sensitive feature selection methods filter methods, embedded methods, and wrapper methods [3]. There have not been many wrapper or embedded cost sensitive feature selection methods developed due to the computational power likely used by these algorithms. An example of one that was developed is a genetic feature selection method by Yang and Honovar in which the fitness function is a combination of the accuracy of the neural network classifier and the cost of performing the classification [3]. Rather, more focused has been placed on cost-sensitive feature selection methods which are much less computationally intensive.

Within cost sensitive filter methods, the two main metrics are correlation and mutual information [3]. Two examples of filter methods are Cost Based CFS (Correlation-based feature selection) and Cost-based mRMR (Minimal-Redundancy-Maximal-Relevance). Cost Based CFS for example evaluates a subset of features MCs by the following equation: $MC_S = \frac{k\bar{r}_{ci}}{\sqrt{k+k(k-1)\bar{r}_{ii}}} - \gamma\frac{\sum_{i=1}^{k} C_{test}(A_i)}{k}$. In which $\bar{r}_{ci}$ is the average correlation between the features of S and the class and $\bar{r}_{ii}$ is the average intercorrelation between the features in S. The second part of the equation controls for the weight of the features by subtracting the average feature cost in S multiplied by a constant γ. By using these features selection methods to only include features that are economical and contain much information, hopefully the learning algorithms can avoid the curse of dimensionality and produce an accurate classifier that minimizes cost at the same time [3].

2.5 Cost Sensitive Naive Bayes

Cost sensitive naive Bayes is similar to normal naive Bayes, but the differences appear during the classification process. The costs that are tracked include the test cost for finding the value of attributes that are hidden, as well as the cost for misclassification. The training data uses full set of known values for each attribute, but during testing, unknown values are introduced, hiding the value in the testing data. These unknown values are randomly assigned to be hidden, and the classifier has a choice on whether or not the value should be tested, adding the cost for finding that attribute and revealing the hidden value.

The discretization of the data for continuous attributes is done using a minimal entropy method, and helps increase accuracy [9]. It does so by trying to split the data into nominal values by keeping values where the classes are the same within a range, similar to the entropy method for generating splits in a decision tree.

With the missing values, an example is passed through to get classified. For each missing value, the utility is calculated, and if that utility is greater than 0, that means that it is worth running the test and obtaining the value for that attribute. This can be for values such as whether or not a politician voted to repeal Obamacare or not, from the voting dataset, which can be extremely useful in determining if the voter is Republican. The utility is calculated with a series of equations and methods for determining them, which are written below. The utility is equal to the gain of the example, given the set of known and the unknown attribute, minus the cost for the specific attribute. The gain is the reduction in the expected misclassification cost obtained from knowing the true value of the unknown attribute [9].

Once the utilities are calculated, then comes the part where the values must either be revealed and add to the test cost, or the values are left unknown. The paper described the method for doing the batch test as well as a sequential test, to reveal which attributes are tested and which are left. The sequential test calculates the utility, selects the single attribute with the highest utility that is above 0, and then repeats until there are no unknown values left or there are no values with a utility above 0. The batch test is different, as in the real world it often is not feasible to run the tests in order and wait for the results to know what to test next. Thus, the algorithm calculates the utility, and using a greedy algorithm, test each attribute that has a utility above 0.

Finally, after calculating these values and getting to this point, the class label is determined by comparing the risk. In the paper, the algorithm was compared to Lazy Naive Bayes, Exacting Naive Bayes, and Cost Sensitive Decision Trees. In all of these comparisons, the csNB algorithm performed the best. In addition, the Cost Sensitive Naive Bayes method itself can be modified to run similarly to the lazy and exacting versions of Naive Bayes. This can be done by setting the test costs sufficiently high that the tests are never run, resulting in the Lazy Naive Bayes. To do the same but for Exacting Naive Bayes, the test costs can be set to 0, such that every test is run.

Cost Sensitive Naive Bayes works well in all situations that Naive Bayes itself works well in. It is good at being a classifier, but not very well for density estimations. This means that the cost sensitive version behaves the same, and has very simplistic independence assumptions. These assumptions often work well, even though they are almost always incorrect. The cost sensitivity portion changes the way that it is classified though, and as such to make the decision boundary linear, it is much more complicated, unless the parameters are modified in a way that makes the cost sensitive portion disappear, such as setting the costs to all be a cost of 0. The weakness of Cost Sensitive Naive Bayes is that the evaluation of which attributes should be revealed can become slow and very intensive. The computations can take a long time, and oftentimes the values for the cost matrix and the test cost vector can greatly affect the performance of the algorithm. The algorithm tries to minimize the cost as much as possible, which also tries to increase the accuracy as a result.

2.6 Cost Sensitive K Nearest Neighbors

KNN classification is a type of instance based learning algorithm that stores the entire training dataset in computer memory and calculates relevant metric during prediction time. This algorithm has three key elements: the labeled training examples, the distance function that serves as metric, and a K value that specifies number of nearest neighbors to look for.

In the learning phase of traditional KNN, the classifier would parse the input data, encode non-numerical attributes through one hot encoding or label encoding, and finally store the processed training data for prediction. In its prediction phase, the traditional KNN classifier first prepare the non-numerical attribute in the new instance, if any, in the same way as what it did to training data, so that the distance between different values can be quantified. Then, the classifier calculates the distance between the new example to all labeled training examples with the specified choice of distance function, and mark the k nearest neighbors. Finally, it returns the most likely label by either majority vote of the k neighbors:

$$y' = argnax_v \sum_{xi,yi \in D_Z} \delta(v, y_i)$$

Or alternatively by weighted vote (eq(x+1)), in which case closer training examples weighs more than further ones:

$$y' = argnax_v \sum_{xi,yi \in D_Z} w_i \cdot \delta(v, y_i)$$

Popular distance function that does not require additional parameter includes:

- Euclidean distance $= \sqrt{\sum(a-b)^2}$
- Manhattan distance $= \sum(|a-b|)$
- Chebyshev distance $= \max(|a-b|)$

Still, there are two major issues with KNN. One is its very sensitive to noise in the training data, and the other is majority vote can be not very effective if the k neighbors vary widely in the distance.

To solve the general issue of imbalanced data, direct-CS-kNN and distance-CS-kNN are proposed as two types of cost sensitive learning algorithm. In addition, several strategies (i.e., smoothing, minimum-cost k value selection) are used to improve the performance.

Direct cost sensitive approach is simple in its nature, and its developed based on the fact that classifiers that give conditional probability estimates for training examples also gives that for test examples, with which the optimal class label can be computed with corresponding cost matrix.

In the learning phase of direct cost sensitive KNN, everything is the same as traditional KNN. However, in prediction phase, the cost sensitive classifier calculates the class probability estimates $P(i|x) = \frac{K_i}{K}$ after selecting its k nearest neighbors, where K_i denotes the number of selected neighbors with true label i. Then, this conditional probability is plugged into a loss function defined as:

$$L(x,i) = \sum_{j=1}^{n} p(j|x)C(i,j) \tag{3}$$

with C denoting the cost matrix. Ultimately, rather than making prediction based on majority votes or weighted votes of the k neighbors, the classifier predicts the label of the new instance to be whichever class that minimizes the loss.

Still, this algorithm also has two issues. One is that if the value of k is too small, then the probability estimate would become unstable and easy to overfit, which would result in an increase in misclassification cost at test examples. The other issue is that the classifier this learning algorithm produces is still susceptible to noises.

Several improvements are made to boost the performance of direct cost sensitive KNN, including calibration methods such as M-smoothing, Laplace correction, feature selection methods, ensemble methods such as bagging and boosting, and stacking method [16].

Distance cost sensitive approach uses modified distance function to select the k nearest neighbors of a given example. For the traditional KNN classifiers, the objective is to minimize the error rate of the classification. In this case, the most commonly used distance functions mentioned above are describing the similarity between two examples, by calculating the difference between these two examples attributes.

For a cost sensitive KNN classifier, however, the goal is to minimize the potential cost of the classification instead of error rate. Considering this new goal, the selected k nearest neighbors which will vote for the classification decision, should be the examples that will vote for the decision with the lowest potential cost. In this case, for each training example, we say that there is a potential cost to select it to vote for the final classification. So the k examples with lowest potential cost should be selected to as the k nearest neighbors.

To calculate the potential cost of a training example, we can calculate the weighted sum of the costs of its classification under different situations of the true label. The weight should be proportional to the probability of the true label. To simplify the problem, we can consider a binary classification problem with cost of correct classifications equal to 0. For the probability of misclassification, it should be inversely proportional to the similarity between this training example and the example to be classified. So one way to calculate the potential cost is

$$C = \frac{C_L}{W}$$

where L is the label of the training example, which means C_L is false positive cost if L is positive, and false negative cost if L is negative. W is the distance-weight of the training example, calculated by $W = 1/d^2$.

After the k nearest neighbors are selected, a weighted vote is held for the probabilities of the true label, with the weights being the distance-weights. After this, plug in the probabilities to the loss function mentioned above, and find the class prediction that will minimize it. This prediction will be the final result of the distance cost-sensitive KNN classifier.

For additional enhancement, feature selection is a good way to improve the performance of KNN. In the given data of examples, there may be some features that are irrelevant to the classification problem we are concerned about. However, KNN will still take them into account and calculate the distance with regard to them. The distance calculated this way is not the real difference between examples in our classification problem, and it is very likely to lead to worse performance of the classifier.

To deal with this problem, we can enhance KNN with feature selection. A forward feature selection is run on the training examples as the training phase of a KNN classifier. Starting from an empty set of features, we perform an iterative feature selection, with each iteration adding one feature to the selected feature set. The selected feature at each iteration is the one that improves the expected performance of KNN classifier. Expected performance is measured with cross validation on the training examples.

The feature selection stops when adding a feature will no more improve the performance, or all features are already selected.

The stacking method is an ensemble learning algorithm that allows us to combine different base classifier by learning a second level classifier on top of them. In the paper [16], Bahnsen proposed that use several cost-sensitive classifiers with different cost matrix parameters to serve as low-level learners, and use the prediction they made in each fold during cross validation to train and produce a set of upper level features for the top-level cost-sensitive linear classifier, such as logistic regression, to be trained on. Due to the fact that nobody in the group is working on cost-sensitive linear classifier, I chose to use a regular logistic regression as the top-level learner and several cost-sensitive Direct-CS-kNN classifiers with different hyperparameters as base learners.

2.7 Condensed Filter Tree for CSMLC

In CSMLC, we feed the multi-label classification algorithm with a cost function that quantifies the difference between a predicted label-set and a desired one. A general CSMLC algorithm operates on the given cost function, with the goal being better performance on that cost function. Three main cost matrices are considered: Hamming loss, Rank loss and F1 score [14].

2.7.1 Tree Model for CSMLC

Tree models form a binary tree by weighted binary classifiers to conduct cost-sensitive classification. Each non-leaf node of the tree is a binary classifier for deciding which subtree to go to, and each leaf node represents a class. Without loss of generality, we assume that the leaf nodes are indexed orderly by 1, 2, ..., # classes. Making a prediction for each instance follows the decisions of binary classifiers, starting from the root to the leaf. That is, only $O(\log(\#ofclasses))$ decisions are required for making each prediction.

As this classification mainly deals with the multi-label classification, we needs a bijective functions

$$enc() : 0, 1^K \to 1, ..., 2^K$$

for encoding y to c and decoding the predicted class c' to the corresponding label vector y. The more general function is One-Hot Encoding.

As suggested in the paper, we have used the *K-Classifier Trick*. Even with proper ordering, there are $2^K - 1$ total internal nodes (classifiers) on the tree. One existing idea for feasible representations is called the 1-classifier trick , which lets all $2^K - 1$ internal nodes t share one classifier h(x, t). Nevertheless, using the 1-classifier trick often requires the classifier to be of sufficient power to capture different characteristics of different nodes.

2.7.2 Training of Tree Model

Two major classifiers are used to train the binary classifiers: Filter Tree (UFT) and Condensed Filter Tree (CFT).

Filter Tree, which trains the classifiers in a bottom- up manner starting from the last non-leaf layer, and each example (x_n, y_n) is used to train all nodes. The last non-leaf layer of classifiers is trained by forming weighted examples based on the better leaf of the two. After training, each node on the last layer decides the winning leaf of the two by predicting on xn. Then the winning labels form the new leaves of a smaller filter tree, and the classifiers on the upper layer are trained similarly (Alina Beygelzimer, 2007).

In Filter Tree, there are 2^K possible traversing paths from the root to the leaves for each instance; however, many of them are seldom needed if we have reasonably good classifiers, such as paths that result in high costs. Therefore, we can shift our focus to the important nodes on each layer instead of uniform sampling for each instance. Next, we revisit the regret bound of Filter Tree, and show that the bound can be revised to focus on a key path of the nodes on the tree.

In CSMLC, for a feature vector x and some distribution P—x for generating the label vector y, the regret rg of a classifier h on x is defined as:

$$rg(h, P) = E_{y \ P|x}[C(h(x), y)] \min_{g} E_{y \ P|x}[C(g(x), y)] \tag{4}$$

CFT can currently handle evaluation criteria defined by a desired label vector and a predicted label vector. We can view CFT as the first step towards tackling more complicated evaluation criteria, which shall be an important future research direction.

3 Overview of Learning Algorithms

Cost-Sensitive Boosted Neural Networks with Weight-Updating uses a neural network as the weak classifier for AdaBoost [3]. Modifications are presented in AdaBoosts weight-updating step to make the algorithm cost-sensitive with respect to the misclassification costs [3]. Cost-Sensitive Nave Bayes modifies evaluation of the testing data by factoring in the misclassification and feature costs when predicting an examples class label [9]. Cost-Sensitive Decision Trees uses both information-gain and attribute-costs to determine the feature with which a node should partition the data[5].Randomly Selected Decision Trees has two criteria for selecting the next feature used for splitting: the attribute with the most information gain; or the lowest cost[7]. Instability is limited by wrapping the algorithm using 30 iterations of bagging [7]. Cost-Sensitive Boosting implements the AdaBoost algorithm with Cost-Sensitive Decision Tree stumps as the weak classifier. Condensed Filter Trees creates binary trees, based on Adaptive Directed Acyclic Graphs, and classifies examples using Kernelized Logistic Regression [14]. Each weight in the binary classifier is updated by taking into account misclassification costs [14].Direct Cost-Sensitive k-Nearest Neighbors extends the k-Nearest Neighbors algorithm to choose K values that minimize the misclassification costs of the training data [7]. Distance Cost-Sensitive k-Nearest Neighbors implements cost-sensitive distance- weighted voting when calculating the real label that minimizes misclassification costs [7].

4 Specifics of Learning Algorithms

4.1 Cost-Sensitive Boosted Neural Networks with Weight Updating

The cost-sensitive neural networks with boosting used in this experiment, like Zhengs implementation, consisted of an input neuron for each of the attributes in a given dataset. This implementation, however, used 30 hidden neurons in the input layer rather than Zhengs eleven. Using a weak classifier that contains more hidden neurons gives the neural network more flexibility in adjusting weights to deduce an accurate decision. Each hidden neuron uses tanh as its activation function because this function maps information to values between -1 and 1, which are the values used for the set of real and predicted labels. Tanh also relaxes the programming implementation because this function does not produce any overflow or underflow errors; this eliminates a common error for activation functions, such as sigmoid, that produce very precise values. Weighted sums from the hidden neurons are converted to a final binary decision in the output neuron using a sign function, since this function converts any weighted sum that is inputted to either -1 or 1.

When an instance of the weak classifier is created in the boosting algorithm, all of the examples containing the inputted instance of the dataset are propagated through the network. The set of predicted labels and the matrix consisting of the values that each hidden neurons activation-function returns are then used in the backpropagation algorithm to perform batch gradient-descent on the weight matrices. Batch gradient-descent is used rather than stochastic gradient-descent so that the network returns consistent results for classifying a dataset. Stochastic gradient-descent chooses randomized examples to update the weight matrices. Since many cost-sensitive learning datasets contain the class imbalance problem, whereby there are more examples of one real label than another, stochastic gradient-descent may sample examples that only include the more common real label.

The learning rate is set to a value of 0.001 to prevent the weights from oscillating between a local minimum. Using a smaller learning rate also contains the influence each example has on the weight values, in case the datasets contain some kind of noise. Each weight matrix that is returned from the backpropagation algorithm is then propagated through the network once more so that predicted labels are updated using the new weight values. This procedure is repeated for every iteration in the AdaBoost algorithm, which is capped at ten iterations for these experiments. Setting the number of iterations for boosting to a higher value can potentially hurt the performance of the weak classifiers on the training set if the network overfits to the testing set.

A percent-defective parameter is included, which represents the percent of defective examples in each dataset. This parameter is used when calculating common performance metrics for cost-sensitive learning algorithms, such as ECM and NECM. There is no change in the performance of the learning algorithm when the percent defective parameter is changed.

Misclassification costs for type 1 and type 2 errors are hyperparameters that are used to adjust the performance of the cost-sensitive boosted neural networks with respect to the dataset of choice. Weight-Updating 1 and Weight-Updating 2 are the cost-sensitive learning modifications that are implemented to make the weak classifier cost-sensitive, such that misclassification costs are accounted for in the weight-updating step of the AdaBoost algorithm. A parameter selects whether the learning algorithm uses Weight-Updating 1 or Weight-Updating 2 as the cost-sensitive learning modification.

The training and testing data is split up using fivefold cross-validation. This is also the fold number used in the Zheng experiments. A smaller fold size for cross-validation, including five folds, works for partitioning smaller datasets. Cross-validation is useful for determining how robust the learning algorithm is to overfitting, and is crucial when recording validation accuracies for t-test experiments. Each t-test can be performed quickly as well, because only five validation accuracies need to be recorded in order to compare learning algorithms. As a result, there are still enough folds to conclude these results while maintaining a high confidence that they are accurate.

4.2 Cost Sensitive Naive Bayes

The cost sensitive naive Bayes being used in this experiment consists of a model that is generated with a training set with no missing values. The data was discretized using a minimal entropy method, as described by Ling. Once the schema has been updated, I generated a cost for each of the attributes, in order to use for retrieval of missing values in the validation set. Then, using cross validation, I split the data into folds, and then train the model accordingly. After training, the test data is generated having some values randomly assigned as missing for each attributes of each example, at a probability 0.2 of it being labeled as missing. The major part that sets cost sensitive naive Bayes apart from normal naive Bayes happens during classification. With the missing values for attributes, I then begin testing. The classification for this is done using functions to calculate the risk, utility, gain, and estimated values for attributes when an attribute is missing.

The discretization process creates splits for the nominal values, which are determined through the changes of the class and the value, and is then updated in the schema. The reason I chose this method was that it increased the accuracy, though it did decrease the speed slightly when compared to a traditional method of splitting, such as binning. The tradeoff for speed is worth it though, as the accuracy was significantly higher in comparison to the binning method. This might be due to not losing as much resolution from the data, or at least it loses less than other methods.

For the test and misclassification costs, I assigned them in two ways. For the test costs, I generated them uniform randomly from 0 to 1, whereas the misclassification costs were assigned according to the dataset, in order to allow for testing between different algorithms. These costs are only used during the classification of new examples, to determine whether or not it is worth the risk to run the test for the attribute or not. These are done using the utility of each attribute, which takes into account the costs, potential risks, and expected values for the attribute.

The utility is calculated by using the gain, which is the expected reduction in the misclassification cost. This is evaluated by taking the risk of misclassification of the data, with the current set of known attributes, and then subtracting the same risk, but with the additional estimated value of the unknown attribute. This first part is done easily using the normal method for calculating risk, but in order to evaluate the risk with the expected value for the unknown attribute, the process is more complicated. I created a function to determine the cost of the misclassification, where for each value of the attribute I calculate the risk, and them multiply that by the probability of that value occuring. This method is like what is proposed in the paper, and by using the naive Bayes assumption of conditional independence, I was able to evaluate the probability of the attribute having that value. By summing up the individual probabilities, I was able to then get the probability of the value itself occurring out of every other value for that attribute. Using that, I then calculated the risk, and took the lower risk out of the one with the label of true and the one with false. The risk times the probability of the value became the reduction in the misclassification cost, which was then used in the gain.

Once the utilities are calculated, then comes the part where the values must either be revealed and add to the test cost, or the values are left unknown. The paper described the method for doing the batch test as well as a sequential test, and I tested both. Ultimately, I decided on doing the batch test, as the evaluation is faster, and the performance was more reflective of real world situations where it might not be feasible to wait for an individual test to finish running before testing the other values. The paper describes this method as a greedy algorithm, where the utility for each unknown attribute is calculated, and the tests are run on all that have a utility greater than 0.

After revealing each value and adding the appropriate test cost, the example is classified by determining which label, true or false, has a lower risk of misclassification, used earlier in the code. The method which I used to determine the ROC graph was the risk, rather than the confidence. The confidence was not a metric that was explained in the paper, nor was the ROC graph, and so I implemented it using the risk as the threshold values and evaluated the area using the pooling method previously implemented, though it does not return results that are useful.

The results I obtained by running these tests gave very high accuracies, generally around 80%, and precisions extremely high, sometimes even as high as 1.0 with certain parameters and datasets, though the recall was generally low. The reason for the high precision is that sometimes it does not run any tests, which can skew the labeling towards false. This means that I rarely have false positives, but I may have false negatives. A method to change this would be by increasing the cost of misclassifying, specifically that of false negatives. Doing so would reduce the performance, but should increase the recall.

The parameters for my program are first the dataset that is to be used, the second is whether or not the data is cross validated. The last parameter is which type of costs to use, essentially whether or not to emulate Lazy Naive Bayes or Exacting Naive Bayes. This parameter takes 3 values. For using both misclassification costs and test costs which is the normal method, the parameter is 0 or below. For just misclassification cost without evaluating the tests, the parameter is 1. By setting all test costs to be very high, no tests will ever be run, and this is Lazy Naive Bayes. For evaluating every attribute value, the parameter value for that is 2, which is calculated by setting the test costs to 0. To add in the costs of the attributes found, which is Exacting Naive Bayes, the parameter for that is 3.

4.3 Cost Sensitive Tree Algorithm

The Cost Sensitive Tree Algorithm takes in a dataset with a set of costs for each attribute then produces a tree classifier that balances both accuracy and tree cost based upon a parameter . The feature selection criteria, the equation that is used to select the feature to split on a specific node is as followed:

$$CSG(X^i, f) = \frac{|X^i|}{X} Gain(X^i, f) - \lambda \cdot Cost(f)_{[f \notin F]} \tag{5}$$

The first term $\frac{|X^i|}{X}$ is a normalization constant which is a ratio of the number of original examples to the current examples at the node to encourage the algorithm to choose more expensive features higher up in the tree. This is because more examples pass through these higher nodes, meaning that it is more worth it to pay for these features. The second term $Gain(X^i, f)$is the information gain value of that feature which the reduction of class entropy by splitting on that feature. Lastly, the third term $Cost(f)_{[f \notin F]}$ is the attribute cost of that feature multiplied by a constant. It includes an indicator function so that the cost of the feature only factors in if that features has not been selected higher up in the tree. This method allows for features to be purchased only once and re-used any time after.

An alternative by similar feature selection criteria uses gain ratio instead of simply information gain:

$$CSGR(X^i, f) = \frac{|X^i|}{X}\left(\prod_{j \in Path(i)} \frac{1}{H(X^i, F^j)}\right) \cdot Gain(X^i, f) - \lambda \cdot Cost(f)_{[f \notin F]} \tag{6}$$

To allow for indicator function that removes the cost of a feature if it has already been selected higher up in the tree, the order of tree creation was modified. The tree needs to be created in a level order manner, so a node in a lower level and re-use all of the features that were used in a level higher. This can be either done recursively or iteratively, both the CSG and CSGR algorithms were done iteratively by places children nodes into a FIFO queue. This was decided because the iterative tree level creation runs in $O(N)$ time while recursive runs in $O(N^2)$.

In summary, both the CSG and the CSGR algorithms go as followed:

```
Input: training data, $\gamma$
Output: a cost-sensitive decision tree
Create a root node and place it into the queue;
while queue is not empty do
    Remove the first node from the queue;
    Get the best feature based upon the given cost sensitive selection
      criteria;
    Split node into left and right children;
    if child has values to split on and not pure node then
        Add it to the queue;
    end
end
```

Algorithm 1: CSG and CSGR Algorithm

Upon completion of the algorithm, examples can be sorted through the tree from the reference of the root node. The tree has been selected to minimize tree cost and maximize accuracy. For the specific test the parameter value for γ was set to the highest value that does not decrease the accuracy more than 1% than the baseline when γ is set to 0. The tree was not a height limit on the tree. All tests were run with CSG algorithm as the performance was consistently higher with ig rather than gain ratio. Since this does not pertain to the comparison between algorithms, it should be noted that the cost-sensitive decision tree algorithm showed success at reducing the overall cost of the tree while not significantly impacting accuracy when the γ

was set to the appropriate level. For simplicity of results in the algorithm comparison, the CSG algorithm was chosen to be included as it consistently performed better than CSGR.

4.4 Randomly Selected Decision Tree

The randomly selected decision tree is an alternative algorithm to the cost sensitive decision tree algorithm CSG and CSGR above. It is based upon the idea that by randomly choosing between choosing the best attribute that with the highest information gain – and the proper attribute – that with the lowest feature cost will create a tree that correctly balances both tree cost and accuracy. The randomness of this selection is handled by a variable β. The algorithm chooses the best attribute when rand(0,1) $<\beta$ and the proper attribute otherwise. Since an individual randomly selected decision tree is inherently unstable, 30 rounds of bagging using 100% of the data are performed to create the final classifier. The cost of the tree is calculated as the average of the 30 trees. In summary, the algorithm is as followed

```
**Input:** training data, $\beta$
**Output:** a randomly selected decision tree
Use bagging to create 30 of the following RSDT classifiers ;
Create a root node and place it into the queue;
while queue is not empty do
    Remove the first node from the queue;
    $r \leftarrow$ randomly assign $\in [0, 1]$ ;
    if $r <$ then
        select "best" attribute;
    end
    else
        select "proper" attribute;
    end
    Split node into left and right children ;
    if child has values to split on and not pure node then
        Add it to the queue;
    end
end
```

Algorithm 2: Randomly Selected Decision Tree Algorithm

Upon completing, use the ensemble of classifiers to classify an individual example. The β value was set to .2 for all test per recommendation from the paper the algorithm was derived. There was not a height limit on the tree. Again, the Randomly Selected Decision Tree algorithm showed success at decreasing the overall tree costs without significantly impacting accuracy.

4.5 Cost Based Boosting Feature Selection

This feature selection method is based upon a method discussed in class. To select a subset of features, boosting is run on a dataset with a series of decision tree stumps (only containing one attribute). Upon termination of boosting, the attributes contained in the selected decision stumps are returned. The decision tree stumps are based upon CSG Cost Sensitive Decision Tree algorithm using weighted entropy, so therefore the features selected by the boosting algorithm consider both feature cost and ability to classify the weighted examples. The algorithm is as followed:

```
Input: training data
Output: a set of selected feature
Initialize equal weights for each example in the dataset;
while the weighted error of the last classifier $\in (0, .5)$ do
    Create a decision tree stump on the weighted examples;
    Update the weights, increasing those missed by the previous
      classifier and vice versa;
end
return a set of the features selected by the boosting algorithms tree
  stumps
```

Algorithm 3: Cost Based Boosting Feature Selection Algorithm

The result of the Cost Based Boosting Feature selection will return a list of features that are both cost conscious and do a good job at covering the features that are important for classifying all examples.

4.6 Filter Tree for Cost-Sensitive Multi-Label Classification

Filter tree is a "bottom-up" tree algorithm, essentially equivalent to a single elimination tournament on the set of labels. The underlying graph structure is a binary tree, constructed recursively from the root as in the tree reduction. Prediction problems associated with the nodes are different, however. Working from the leaves toward the root, each internal node predicts which of its two inputs is more likely, given the features.

In the first round, the labels are paired according to the graph, and a classifier is trained for each pair to predict which of the two labels is more likely. (The labels that dont́ have a pair in a given round, win that round for free.) The winning labels from the first round are in turn paired in the second round, and a classifier is trained to predict whether the winner of one pair is more likely than the winner of the other. This process repeats until an overall winner is declared at the root. The base classifier used in the below two algorithms is the Kernelized Logistic-Regression. The filter tree algorithm comes in two parts: training and testing.

```
Input: training data
Output: a filter tree
Cost-Sensitive training set, $S = (x_n, y_n)_{n=1}^{N}$ ;
Fix a binary tree T over the labels;
for each internal node n in the order from leaves to roots do
    Initialize training set for each node, S$_n$ =$\emptyset$;
    for each example $\in$ training data do
        Let a and b be the two classes input to n ;
        Find the best node ;
        Update label, l$_n$ = $argmin(C_a, C_b)$;
        Update weight, w$_n$ =| C$_a - C_b$| ;
        S$_n$$\leftarrow$ S$_n$$\cup$ (x$_n, l_n, w_n$);
        Learn the best node, predict$_n$ = $Learn(S_n)$;
    end
end
return best node predict$_n$.
```

Algorithm 4: Filter Tree Training Algorithm

```
Input: a new example
Output: a class label
return the label l such that every classifier on the path from leaf l to
  the root prefers label l.
```

Algorithm 5: Filter Tree Testing Algorithm

Each training example for node n is formed conditionally on the predictions of the classifiers on the path from n to the true label. The process of training classifiers to predict the best of a pair of winners from the previous round is repeated until the "root" classifier is trained. The testing algorithm is very simple. We just predict which input to the root node has the best label, then go to the node predicting that output, and repeat until a leaf is reached, determining the multiclass prediction.

In Condensed-Filter Tree the initial algorithm remains the same but instead of the misclassification costs the evaluation matrix is used as the cost matrix (cost function) to find the ideal path that results in low costs. The most common of them are the Hamming-Loss, Rank-Loss, F1-Score and the Accuracy of the classifier. As the Hamming-Loss is most closely related to the misclassification matrix, so the average misclassification cost for the Condensed-Filter Tree is calculated as the Hamming-Loss.

5 Experimental Evaluation

5.1 Experiment Setup

The main purpose of the experiment is to evaluate the performance of all implemented cost-sensitive learning algorithms by comparing their cost and other key performance measurements such as accuracy against each other. All algorithms are listed in Table below:

Index	**Algorithm**	**Abbr**	**Base Classifier**
1	RSDT	RSDT	ID3
2	CSTree	CST	ID3
3	RSDT w/ boostFeatureSelection	RSDT-BFS	ID3
4	CSTree w/ boostFeatureSelection	CST-BFS	ID3
5	CS Naive Bayes	CSNB	Naive Bayes
6	Distance CS KNN	DT-CS-KNN	KNN
7	Distance CS KNN w/ FeatureSelection	DT-CS-KNN-FS	KNN
8	Direct CS KNN	DR-CS-KNN	KNN
9	Direct CS kNN w/ M-Smoothing	DR-CS-KNN-SM	KNN
10	Direct CS kNN w/ Stacking	DR-CS-KNN-STK	KNN
11	CSBNN-WU1	WU1	Neural Network
12	CSBNN-WU2	WU2	Neural Network
13	Filter Tree for CS Classification	FT-CS	Filter Tree
14	Condensed Filter Tree for CS Classification	CFT-CS	Filter Tree

Table 1: All Implemented Algorithms

5.2 Dataset Information

We conducted experiments using above algorithms and three UCI data sets listed in Table below:

Datatset	**No. attributes**	**No. examples**	**Distribution (T/N)**
cm1	21	498	48/450
breastCancer	9	126	64/52
thoracicSurgery	16	470	70/400

Table 2: Summary of Dataset

Software-defect dataset analysis is used in numerous applications in order to predict whether a given piece of software will encounter any flaws during execution. This learning problem benefits significantly from utilizing cost-sensitive learning algorithms with respect to misclassification costs, because the magnitude of misclassifying a defect-prone example is much larger than misclassifying one not prone to defects [3]. CM1 is a relatively small software defect dataset that consists of slightly fewer than 500 examples. The attributes within this dataset are only continuous features, which add variability to the type of features encountered in the datasets used for this experiment.

For the medical datasets, the different test have different costs in real life, so it is important to be able to create an accurate classifier while limiting the feature costs. Also the cost of a FP and a FN may not be the same for the datasets. For example with breast cancer, it is more costly to tell a patient that they dont have breast cancer when they actually do instead of telling them the do when they actually do not.

The Thoracic Surgery dataset was chosen due to the medical context of the data. It includes features such as the patients pain before surgery and their medical diagnosis. These features are used to predict the patient's prognosis – whether they will live one year past the thoracic surgery. It is important to get an idea of this so patients and be split into low risk and high risk and treated accordingly.

The Breast Cancer dataset was chosen due to the medical context of the data. It includes features such as the patients and the amount of glucose and resisting in the patients blood. These features are used to determine whether the patient has breast cancer.

5.3 Methods and Results

The cost-sensitive learning algorithms surveyed in this experiment were divided into two categories: algorithms that implemented cost-sensitive learning using misclassification costs and those that used feature costs. For each dataset, values for the misclassification and feature-cost hyperparameters were set. The values for the type 1 and type 2 errors in the cost matrix were tuned based on optimizing the average validation accuracies that they produced for each learning algorithm. Feature costs were randomly generated for each dataset. These hyperparameters are summarized as follows:

$$C_{\text{cm1}} = \begin{bmatrix} 0 & 0.25 \\ 0.5 & 0 \end{bmatrix}, C_{\text{breastCancer}} = \begin{bmatrix} 0 & 0.25 \\ 0.9 & 0 \end{bmatrix}, C_{\text{thoracicSurgery}} = \begin{bmatrix} 0 & 0.25 \\ 0.6 & 0 \end{bmatrix}$$

After the hyperparameters were treated as controlled variables, t-tests were performed on each combination of the learning algorithms in a given category for each dataset. Each result will contain either the abbreviation of the learning algorithm that significantly outperformed, or nothing if the t-test failed to reject the null hypothesis. T-tests were performed on the validation accuracies of five-fold cross validation, with a 95 percent confidence interval. The script was used to calculate all of the t-tests. Results of the t-tests for both the feature and misclassification cost-sensitive learning algorithms are summarized as follows:

5.3.1 Results for Testing Feature Cost-Sensitive Classifiers

Family	**Algo**	**cm1**	**breastCancer**	**thoracicSurgery**
ID3	CST	0.821 ± 0.031	0.661 ± 0.101	0.763 ± 0.029
	RSDT	0.879 ± 0.035	0.714 ± 0.044	0.810 ± 0.030
ID3 with FS	CST-BFS	0.883 ± 0.028	0.713 ± 0.074	0.844 ± 0.009
	RSDT-BFS	0.885 ± 0.013	0.646 ± 0.075	0.844 ± 0.009
Naive Bayes	CSNB	0.706 ± 0.017	0.677 ± 0.011	0.353 ± 0.031

Table 3: The mean feature cost of feature cost classifiers across 5 folds

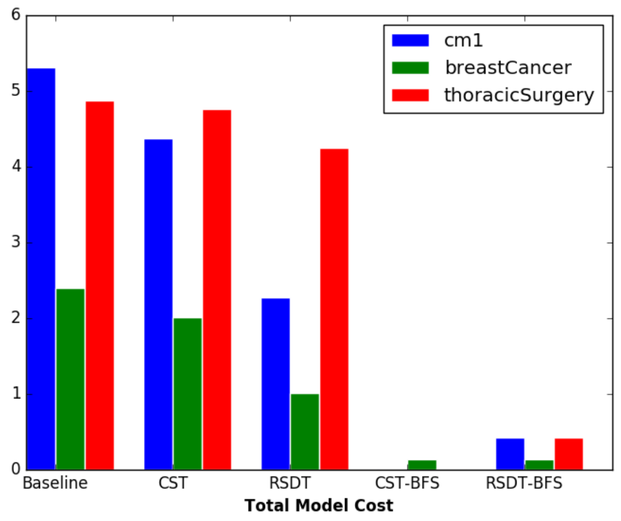

Figure 1: The feature costs performance bar graph of different classifiers across three data sets

Student's T-Test for **feature cost algorithms**:

cm1	**RSDT**	**CST**	**RSDT-BFS**	**RSDT-BFS**	**CSNB**
RSDT		RSDT			RSDT
CST			RSDT-BFS	CST-BFS	CST
RSDT-BFS					RSDT-BFS
CST-BFS					CST-BFS
CSNB					

breastCancer	**RSDT**	**CST**	**RSDT-BFS**	**RSDT-BFS**	**CSNB**
RSDT					RSDT
CST					CST
RSDT-BFS					RSDT-BFS
CST-BFS					CST-BFS
CSNB					

thoracicSurgery	**RSDT**	**CST**	**RSDT-BFS**	**RSDT-BFS**	**CSNB**
RSDT			RSDT-BFS		RSDT
CST			RSDT-BFS	CST-BFS	CST
RSDT-BFS					RSDT-BFS
CST-BFS					CST-BFS
CSNB					

Table 4: T-Test of Feature Cost-Sensitive Algorithms on cm1, bc(breastCancer), ts(thoracicSurgery) datasets

5.3.2 Results for Testing Misclassification Cost-Sensitive Classifiers

Family	**Algo**	**cm1**	**breastCancer**	**thoracicSurgery**
Neural Net	WU1	0.901 ± 0.008	0.548 ± 0.137	0.821 ± 0.044
	WU2	0.899 ± 0.006	0.054 ± 0.056	0.670 ± 0.298
Naive Bayes	CSNB	0.705 ± 0.0195	0.814 ± 0.0183	0.918 ± 0.024
KNN	DT-CS-KNN	0.852 ± 0.0346	0.560 ± 0.0968	0.174 ± 0.0479
	DT-CS-KNN-FS	0.865 ± 0.0387	0.578 ± 0.119	0.647 ± 0.0679
	DR-CS-KNN	0.904 ± 0.022	0.561 ± 0.123	0.851 ± 0.036
	DR-CS-KNN-SM	0.904 ± 0.022	0.448 ± 0.084	0.851 ± 0.036
	DR-CS-KNN-STK	0.903 ± 0.023	0.552 ± 0.103	0.851 ± 0.036
Filter Tree	FT-CS	0.90 ± 0.024	0.431 ± 0.0895	0.850 ± 0.0421
	CFT-CS	0.90 ± 0.024	0.561 ± 0.112	0.852 ± 0.0395

Table 5: Accuracy results of misclassification cost-sensitive classifiers across 5 fold

Family	**Algo**	**cm1**	**breastCancer**	**thoracicSurgery**
Naive Bayes	CSNB	0.227 ± 0.005	0.254 ± 0.005	0.588 ± 0.007
KNN	DT-CS-KNN	0.054 ± 0.012	0.121 ± 0.032	0.207 ± 0.009
	DT-CS-KNN-FS	0.056 ± 0.007	0.106 ± 0.027	0.126 ± 0.013
	DR-CS-KNN	0.098 ± 0.030	0.161 ± 0.023	0.002 ± 0.002
	DR-CS-KNN-SM	0.124 ± 0.001	0.127 ± 0.002	0.119 ± 0.000

Table 6: Mean misclassification costs of misclassification cost-sensitive classifiers across 5 fold

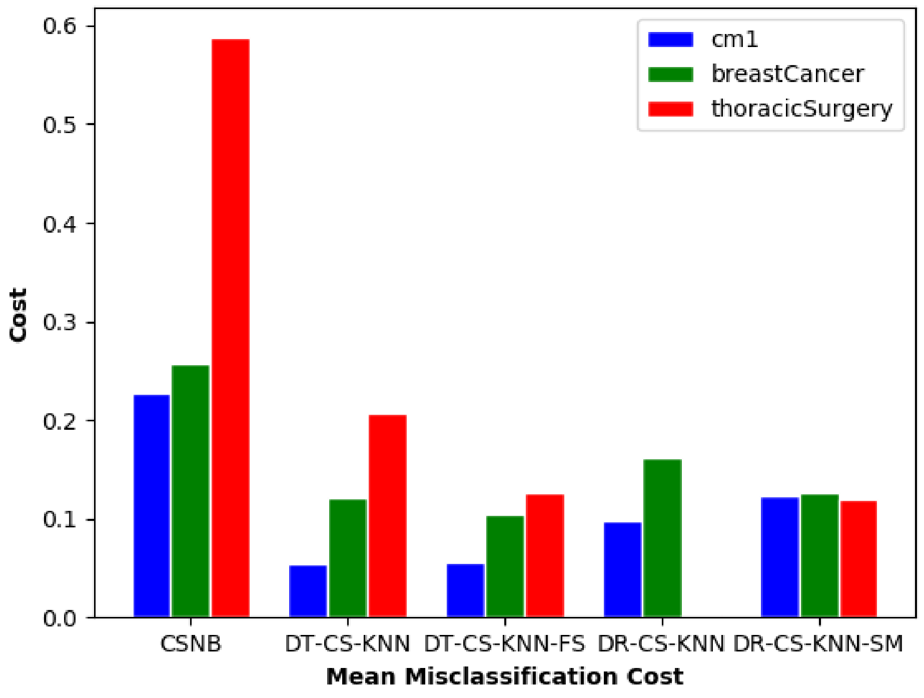

Figure 2: The mean misclassification costs of 5 different classifiers across 3 training examples

Student's T Test for **misclassification cost algorithms**:

cm1	**WU1**	**WU2**	**DT-CS-KNN**	**DT-CS-KNN-FS**	**CSNB**	**DR-CS-KNN**	**DR-CS-KNN-SM**	**DR-CS-KNN-STK**	**FT-CS**	**CFT-CS**
WU1			WU1		WU1					
WU2			WU2		WU2					
DT-CS-KNN						DR-CS-KNN	DR-CS-KNN-SM	DR-CS-KNN-STK	FT-CS	CFT-CS
DT-CS-KNN-FS					DT-CS-KNN-FS					
CSNB						DR-CS-KNN	DR-CS-KNN-SM	DR-CS-KNN-STK	FT-CS	CFT-CS
DR-CS-KNN										
DR-CS-KNN-SM										
DR-CS-KNN-STK										
FT-CS										
CFT-CS										

breastCancer	**WU1**	**WU2**	**DT-CS-KNN**	**DT-CS-KNN-FS**	**CSNB**	**DR-CS-KNN**	**DR-CS-KNN-SM**	**DR-CS-KNN-STK**	**FT-CS**	**CFT-CS**
WU1					CSNB					
WU2					CSNB				WU2	
DT-CS-KNN					CSNB					
DT-CS-KNN-FS					CSNB					
CSNB						CSNB	CSNB	CSNB	CSNB	CSNB
DR-CS-KNN										
DR-CS-KNN-SM								DR-CS-KNN-STK		
DR-CS-KNN-STK										
FT-CS										
CFT-CS										

thoracicSurgery	**WU1**	**WU2**	**DT-CS-KNN**	**DT-CS-KNN-FS**	**CSNB**	**DR-CS-KNN**	**DR-CS-KNN-SM**	**DR-CS-KNN-STK**	**FT-CS**	**CFT-CS**
WU1			WU1	WU1						
WU2			WU2							
DT-CS-KNN				DT-CS-KNN-FS	CSNB	DR-CS-KNN	DR-CS-KNN-SM	DR-CS-KNN-STK	FT-CS	CFT-CS
DT-CS-KNN-FS					CSNB	DR-CS-KNN	DR-CS-KNN-SM	DR-CS-KNN-STK	FT-CS	CFT-CS
CSNB							CSNB			
DR-CS-KNN							DR-CS-KNN-STK			
DR-CS-KNN-SM							DR-CS-KNN-STK			
DR-CS-KNN-STK										
FT-CS										
CFT-CS										

Table 7: Student's t-test of misclassification cost-sensitive classifiers on cm1, breastCancer and thoracicSurgery datasets

5.4 Discussion of Results

5.4.1 Feature Costs Discussion

The algorithms that were included feature costs in their implementation of cost-sensitive learning are cost-sensitive decision tree (CST), randomly selected decision tree (RSDT), cost-sensitive decision tree with boosted cost-sensitive feature selection (CST-BFS), randomly selected decision tree with boosted cost-sensitive feature selection (RSDT-BFS), and cost-sensitive naive bayes (CSNB). With keeping the feature costs the same for each dataset, there are trends that can be drawn across the datasets.

First, the cost-sensitive naive bayes algorithm performs significantly worst out of all the algorithms in regards to accuracy as shown by the t-tests. This is believed to be so because when the misclassification cost was set to 0, the algorithm did not run any tests, as to not increase the costs at all. The algorithm only reveals values of attributes if the risk of misclassification is higher than the risk involved after revealing the attribute. This caused the algorithm to not properly evaluate and determine the label behind the examples due to using only the known values.

Next, there is a significant difference between the performance of the CST and RSDT algorithms with and without the cost-sensitive boosting feature selection. In general, the CST and RSDT with the feature selected tend to outperform their non feature selected counterparts and the CSNB algorithm. This is believed to be because of the overfitting that occurs to the CST and RSDT algorithm without feature selection. When the CST and RSDT algorithms were run without cross validation they were generally able to achieve a perfect fit on the data without a depth restriction on the tree. Therefore, when cross validation is employed the models are too focused on the specific examples that they trained on and are not able to generalize. Employing the cost-sensitive boosting feature selection is an efficient way to fix this issue. It is able to select those features which are both affordable and are important in classifying the examples. For these datasets there are small subsets of data that are most important for classifying the examples and the other features that are included can add unnecessary costs and increase generalization error.

The common features selected by the cost sensitive boosting for the cm1 dataset was LOC, for breastCancer were age and resistin, and for thoracic-Surgery was DGN. These selected features make sense because your cancer risk goes up with age and resistin has been shown to be an important prognosis marker in breast cancer [8]. Also, according to the info for the thoracic surgery DGN is the specific diagnosis the doctors gave the patient, which makes sense to have an impact on the patients prognosis.

In addition to the accuracy of the different classifiers in the test, it is also important to look at the cost that it took to create the models. CSNB was not included in the model cost comparisons as the costs are generated after the model is created. The costs only are found during classification, as the model is still trained in the same method as Naive Bayes, and the costs represent the cost of misclassifying an example and obtaining a missing value. The baseline is the cost of a non-cost-sensitive tree algorithm. For the other algorithms again there is a clear difference in cost. Each algorithm is able to decrease the total cost compared to the non cost sensitive baseline without significantly affecting the accuracy. Comparing the algorithms, the cost for those that have feature selected are much less than those without. That is because the cost-sensitive boosting feature selected limits the amount of features the algorithms can choose from. This greatly limits the overall cost of the algorithm.

When comparing the RSDT and CST algorithms for accuracy and costs there does not appear to be a significant difference between the two with or without feature selection. Both of them are effective at reducing total classifier cost without significantly affecting the accuracy. In summary, the cost-sensitive boosting feature selection used in conjunction with the RSDT and CST algorithms appear to be the best option as they are able to produce a classifier that is much less costly than baseline while maintaining a high accuracy and generalizability.

5.4.2 Misclassification Costs Discussion

There was no single learning algorithm that significantly outperformed all of the others for all datasets used in the experiments. One factor that impacted these results is the limited number of examples in the datasets used. Each dataset contained no more than 500 examples, which made it very difficult to draw general insights from and make accurate classifications about these learning problems. From the perspective of cost-sensitive learning, these datasets also encountered the class-imbalance problem because there was a significant discrepancy in the number of examples for each class label. This caused the margins between the decision boundaries generated by each training example to be vague, and testing examples were easily misclassified as a result.

Some of the learning algorithms evaluated produced inconsistent performance, with respect to validation accuracies, across each fold in cross validation. For example, when CSBNN-WU2 was trained on the thoracicSurgery dataset using fivefold cross-validation, three of the folds had an accuracy over 80%, while the remaining two folds had accuracies of 69% and 14%. These results suggest that the learning algorithm was overfitting its classifiers to the training examples, and that each classifier limited variance to make generalizations that would allow for accurate classifications on the testing data. T-tests cannot detect that one learning algorithm performs significantly better than another for a certain dataset, when the learning algorithms themselves are not returning consistent results on each fold.

A notable mistake in the execution of our research design occurred in the cross-validation methods used for each learning algorithm. Three researchers inadvertently used one version of the cross-validation algorithm, while the other three used a different one. This dissimilarity could lead to inconsistency in the examples used in the training and testing sets for each cross-validation fold. While, in practice, learning algorithms should be robust to overfitting and produce consistent results, having the same examples in each cross-validation fold for all of the learning algorithms would increase the chances of rejecting the null-hypothesis in each t-test.

5.4.3 Naive Bayes on Breast Cancer

For the tests on misclassification and the misclassification and test costs, the accuracies were extremely high, and with the t-test the algorithm proved that it was ahead of the other algorithms with a confidence of 95%. This is most likely due to a few causes, including Naive Bayes being very good at inferencing from the data as well as the method for discretizing the data. Naive Bayes in itself is a very good classifier, as it is strong at working with inferences due to working directly with probabilities. This makes the analysis and extrapolation an easier process. The discretization of data used an entropy method, using it to split the attributes into labels that were more representative of the data, rather than splitting them up using a binning method. By splitting along points where the value and the label changed, this would help create more accurate discretized features to better represent any relations or indicators in the data. This split might have helped in grouping together attribute values for different ranges, as some specific ranges could be indicators for breast cancer, such as a very low or a very high value for an attribute like adiponectin might be a stronger indicator of breast cancer in a patient as an example. In combination, these are what might have helped the Cost Sensitive Naive Bayes algorithm perform so well on this dataset.

6 Applications of Algorithms

If an algorithm is needed to balance the costs that it takes to obtain individual features, it is recommended to used some sort of feature selection method, such as the cost-sensitive boosting feature selection, as it is able to greatly reduce the total cost be getting rid of expensive and irrelevant features. Both the randomly selected decision tree and the cost-sensitive decision tree are good potential algorithms to use in conjunction with the feature selection, but it is still recommended to explore many methods for your specific learning problem and compare.

Cost-sensitive learning problems that have a well-defined inductive bias are suitable candidates for cost-sensitive boosted neural-networks. Conversely, vague problems contain a hypothesis space with a high variance; these are difficult problems for neural networks to solve because neural networks are very prone to overfitting and cannot easily generalize for new examples. When there is a significant amount of training data for a learning problem, neural networks can have their hyperparameters fine-tuned to both produce better accuracy and meet the real-life constraints of the problem. Also, the base classifiers in cost-sensitive boosted neural-networks are flexible in terms of how their architectures are defined. The underlying concepts that a neural network learns are often not obvious or intuitive. Therefore, training a network on a dataset that contains features predictive of the class label facilitates fine-tuning of the hyperparameters and architecture.

7 Conclusion

Cost-sensitive learning implementations modify learning algorithms to make them better suited to the real-life requirements of a learning problem, aside from the issue of accuracy. Misclassification costs can be addressed through a type of cost-sensitive refinement that allows learning algorithms to better manage the class-imbalance problem [1]. In this approach, a cost matrix weighs type 1 and 2 errors such that a learning algorithm can classify examples in a format that establishes a bias towards one type of misclassification error [1]. Feature costs introduce resource values on each attribute in a dataset, allowing a learning algorithm to produce results that more accurately reflect the real-life constraints of a learning problem [2]. These approaches to cost-sensitive learning introduce hyperparameters that allow a learning algorithm to be fine-tuned such that it can approach a learning problem in a format that more accurately reflects the problems actual restrictions.

The current study addressed the issue of cost-sensitive learning by modifying novel learning algorithms to include either misclassification costs, feature costs, or both. T-tests were implemented on each subsection of cost-sensitive learning algorithms in order to determine if there were any approaches that significantly outperformed others for a particular dataset. From these experiments, we determined that cost-sensitive naive bayes was able to better classify the breastCancer dataset compared to the other learning algorithms that implemented misclassification costs. The success of the experiment was limited by inconsistent application of cross validation on each learning algorithm, which likely had the effect of treating testing and training sets as independent variables. The relatively small sizes of the data sets also appeared to restrict the ability of each algorithm to generalize from the training data. However, even with these limitations, the study reinforced the concept that use of cost-sensitive learning is an important, situation-specific contribution to learning algorithms, that will yield real-life benefit.

Acknowledgment

The workload of each author is listed below in alphabetical order of surname.

- Yiming Chen
  - Wrote section about distance cost sensitive method of KNN
  - Implemented Distance-CS-KNN algorithm
  - Added feature selection to improve Distance-CS-KNN algorithm
- Thomas Cwalina
  - Created and laid out paper structure.
  - Obtained and set up .data, .names, .info with breastCancer and thoracicSugery dataset.
  - Wrote about medical situation in intro cost-sensitive learning.
  - Wrote section about cost-sensitive decision trees.
  - Wrote section about cost-sensitive feature selection.
  - Wrote about the medical datasets selected.
  - Implemented Cost Sensitive Decision Tree Algorithm.
  - Implemented Randomly Selected Decision Tree Algorithm. Cost Sensitive Boosting Feature Selected (Used in conjunction with two other algorithms).
  - Created accuracy table for feature costs.
  - Wrote feature cost result conclusion section.
  - Wrote application of algorithms for feature costs section.
- Devansh Gupta
  - Wrote section about Condensed Filter Tree for Cost-sensitive multi-label classification.
  - Implemented Condensed Filter Tree for Cost-sensitive multi-label classification.
  - Implemented General Filter Tree for Cost-sensitive multi-label classification.
  - Updated the References section according to IEEE citation style.
  - Helped with Formatting of raw texts into latex.

- Rahul Pokharna
  - Wrote up section about Cost Sensitive Naive Bayes for misclassification and feature cost.
  - Wrote about results for Cost Sensitive Naive Bayes on the Breast Cancer dataset.
  - Wrote parts of discussion of results and conclusions.
  - Implemented Cost Sensitive Naive Bayes algorithm.
- Neil Steiner
  - Wrote (First three paragraphs of) Overview, General Overview of Research Papers, Cost-Sensitive Boosted Neural Networks, General Overview of Algorithms, Cost-Sensitive Boosted Neural Networks with Weight Updating, Misclassification Costs Discussion, (Second paragraph of) Applications and Algorithms, and Conclusion.
  - Coded t-test script and performed the t-tests for each combination of learning algorithms. Summarized this methodology Methods and Results section. Figures and tables were created by Stan.
  - Parsed CM1 dataset for experiments. Wrote about software defect datasets in Dataset Information.
  - Implemented CSBNN-WU1 and CSBNN-WU2 algorithms.
- Stan Tian
  - Wrote the Abstract.
  - Wrote Section 1.2 misclassification cost and cost matrix.
  - Wrote Section 2.6 explanation for cost-sensitive KNN algorithm and cost-sensitive stacking.
  - Implemented 2 choices of DirectCS-KNN algorithm (base, m-smoothing).
  - Implemented stacking algorithm with regular logistic regression.
  - Wrote Section 5.1 Experiment setup and 5.3 testing results
  - Produced accuracy result table and misclassification result table.
  - Formatted most raw texts into latex to get better pseudocode, math equations, tables, figures, and references.

References

[1] Zhi-Hua Zhou and Xu-Ying Liu, "Training cost-sensitive neural networks with methods addressing the class imbalance problem" in IEEE Transactions on Knowledge and Data Engineering, vol. 18, no. 1, pp. 63-77, Jan. 2006.

[2] M. J. Kusner, W. Chen, Q. Zhou, Z. Xu, K. Q. Weinberger, and Y. Chen,"Feature-Cost Sensitive Learning with Submodular Trees of Classifiers", AAAI, pp. 1939 1945, 2014.

[3] J. Zheng,"Cost-sensitive boosting neural networks for software defect prediction", Expert Systems with Applications, vol. 37, no. 6, pp. 4537 4543, 2010.

[4] K. Muthukumaran, A. Dasgupta, S. Abhidnya, and L. B. M. Neti,"On the Effectiveness of Cost Sensitive Neural Networks for Software Defect Prediction", International Conference on Soft Computing and Pattern Recognition, pp. 557 570, 2016.

[5] J. V. Davis, J. Ha, C. J. Rossbach, H. E. Ramadan, and E. Witchel, "Cost-sensitive decision tree learning for forensic classification" in Machine Learning: ECML 2006 (J. Fu rnkranz, T. Scheffer, and M. Spiliopoulou, eds.), (Berlin, Heidelberg), pp. 622629, Springer Berlin Heidelberg, 2006.

[6] V. Boln-Canedo, I. Porto-Daz, N. Snchez-Maroo, and A. Alonso-Betanzos, "A framework for cost-based feature selection" Pattern Recognition, vol. 47, no. 7, pp. 2481 2489, 2014.

[7] C. Qiu, L. Jiang, and C. Li, "Randomly selected decision tree for test-cost sensitive learning", Applied Soft Computing, vol. 53, pp. 27 33, 2017.

[8] Y.-C. Lee, Y.-J. Chen, C.-C. Wu, S. Lo, M.-F. Hou, and S.-S. F. Yuan, "Resistin expression in breast cancer tissue as a marker of prognosis and hormone therapy stratification", Gynecologic Oncology, vol. 125, no. 3, pp. 742 750, 2012.

[9] X. Chai, L. Deng, Q. Yang, and C. Ling, "Test-Cost Sensitive Naive Bayes Classification", Fourth IEEE International Conference on Data Mining (ICDM04), 2004.

[10] D. Mease, A. Wyner, A. Buja, “Cost-weighted boosting with jittering and over/under-sampling: Jous-boost”, J. Machine Learning Research 8 (2007) 409439

[11] S. Lomax, S. Vadera “A Survey of Cost-Sensitive Decision Tree Induction Algorithms”, ACM Computing Surveys, 2011

[12] Z. Qin, A.T. Wang, C. Zhang, and S. Zhang “Cost-Sensitive Classification with k-Nearest Neighbors”, Knowledge Science, Knowledge Science, Engineering and Management 6th International Conference, pp112-131, KSEM 2013, Dalian, China, August 10-12, 2013.

[13] Beygelzimer, A., Langford, J., Ravikumar, P. ,“Multiclass Classification with Filter Trees”, June 2007.

[14] Li, Chun-Liang and Lin, Hsuan-Tien., “Condensed filter tree for cost-sensitive multi-label classification”, In Proceedings of the 31st International Conference on Machine Learning, pp. 423431, January 2014.

[15] W. Bi J. Kwok, “Bayes-optimal hierarchical multilabel classification”, IEEE Trans. Knowl. Data Eng., vol. 27, no. 11, pp. 2907-2918, Nov. 2015.

[16] A. C. Bahsen, D. Aouada, and B. Ottersen, “Ensemble of Example-Dependent Cost-Sensitive Decision Trees”, arXiv preprint arXiv:1505.04637, 2015.