CS 280 Programming Assignment 3

Backpropagation Algorithm

Isabelle Tingzon

2011-46060

November 29, 2015

1 Introduction

The Artificial Neural Network (ANN) is a computational model inspired by the biological system of processing information. ANNs are composed of artificial neurons that receive input (like synapses), process information, and produce output (like signals). ANN has a wide range of application in various fields such physics, mathematics, economics, medicine, and many more.

In this programming assignment, I construct an Artificial Neural Network (ANN) trained on the Backpropagation Algorithm. I then compare its performance with that of the C5.0 Cost-sensitive Decision Tree .

2 Methodology

2.1 ANN Implementation

The implementation of ANN trained on Backpropagation algorithm described in this paper is based on the attached code `cs280_neural_net.m`. The code was rewritten and built using Java in Netbeans IDE 8.0.2 running on Windows 8 OS.

2.2 Classifier Evaluation - Training Set

The quality of the classifier depends on several factors, including the quality of the data set on which it is trained as well as the fitting process used to optimize model parameters. In this section, I will discuss several methods used to reduce the effects of problems such as overfitting and unbalanced data sets.

2.2.1 K-fold Cross Validation

Cross validation involves the partitioning of the data set into two complementary subsets: the *training set*, on which the analysis is performed, and the *validation set* or test test, on which the analysis is validated. This method is generally used to get insight of the performance of the classifier on an unknown independent data set. However, with one-round cross validation, there is a danger of overfitting. Overfitting occurs when the model becomes tailored to fit the specific sample rather than reflective of the overall population. One solution to overcome the problem of overfitting is using *k-fold cross validation* wherein data is partitioned and trained on multiple rounds.

The algorithm for k-fold cross validation is as follows:

Listing 1: k-fold Cross Validation

```
Cross Validation (k, data set)
1. Shuffle data set using random permute
2. Divide data set into k partitions
3. For i = 3 ... k
   3.1 Train classifier on all k-1 partitions
   3.2 Test/validate the model on the kth partition
   3.3 Compute the error and accuracy
4. return the mean error and accuracy over k rounds
```

The results from the k-fold cross validation method can be used to select optimal values for the parameters in the model (which will be discussed in detail in section 2.2.2).

2.2.2 Selecting Optimal Parameters

To achieve the best results, it is necessary to find the optimal values for the the learning rate α and the number of nodes per hidden layer. Unfortunately, there is no definitive formula or method to attain these values; however, results from the k-fold cross validation method can provide insight on the behavior of the error and accuracy as the values for the parameters change.

First, to get the optimal values for the number of hidden nodes within each layer, let n be the number of nodes in a hidden layer. I select the candidate values for n as 5, 10, 20, 30, 40, ..., 100. Cross validation is then performed for each value of n (for simplicity, both hidden layers have the same number of nodes). The results of the prediction error and accuracy can be seen in Figure

1 and Figure 2, respectively. The prediction error is generally increasing in

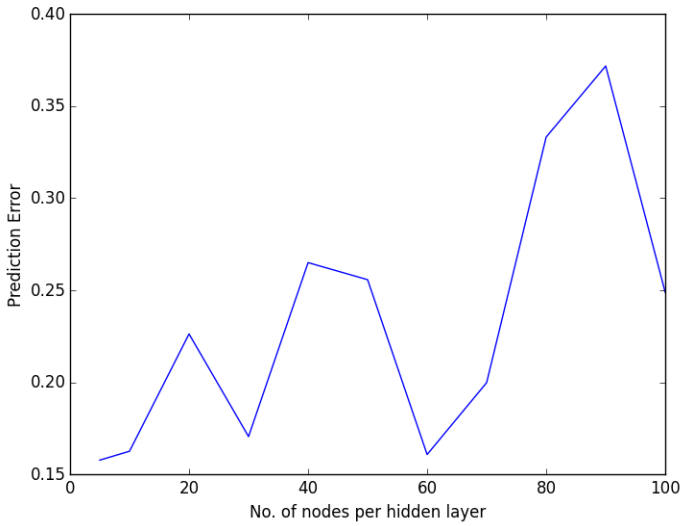

Figure 1: Prediction error vs. Number of nodes per hidden layer

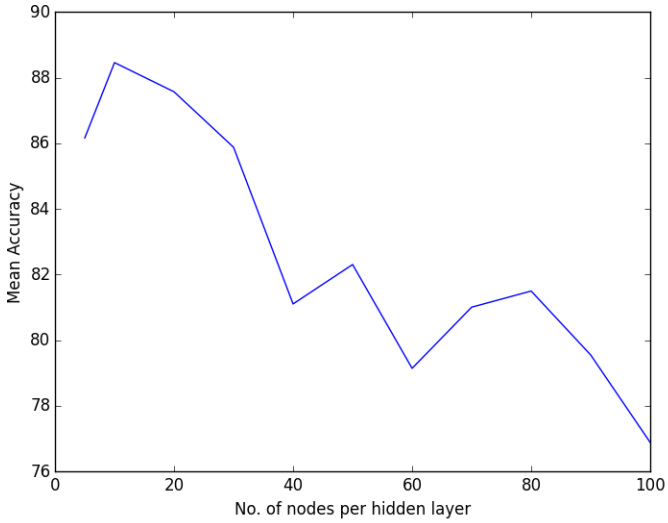

Figure 2: Accuracy vs. Number of nodes per hidden layer

n. The value for n with the smallest error is 5 (error = 0.157776). Since error

Table 1: Sample Proportions for each class

Class	1	2	3	4	5	6	7	8
Proportion (%)	45.82	6.84	0.903	13.68	8.09	9.04	1.38	14.24

diminishes as accuracy grows, the assumption that performance worsens as the number of nodes increases is further bolstered by the results in Figure 2. Here, we see that accuracy decreases as the number of nodes n increases. The value for n with the greatest accuracy is 10 (accuracy = 88.46%).

Since 5 and 10 have the lowest error and highest accuracy, respectively, I let `n_h1=5` and `n_h2=10` where `n_h1` is the number of nodes in the first layer and `n_h2` the number of nodes in the second layer. I then get a good resulting error of 0.009913 and an accuracy of 86.12%.

Similarly, to get the optimal value for the learning rate α, I select candidate values 0.05, 0.1, 0.5 and 1.0. Performing cross validation I find that the best value for α is 0.5 with a prediction error of 0.01138 and accuracy of 88.53%.

2.2.3 Handling Imbalanced data Set

To handle the problem of unbalanced data set, I employ a resampling techniques with the goal of having a proportional number of samples in all classes. Ideally, for data of size s number of classes c, the number of sample per class should be $n = s/c$. This is so that each class should compose $r = n/s$ (target ratio) of the whole data set to be equally represented. However, from Table 1, it is evident that some classes have more samples in the data set than others. In effect, the minority classes are more likely to be misclassified due to insufficient representation.

It is important to note that resampling is done only after splitting the

original data set into the training set and the validation set (i.e. oversampling is performed on the training set only). This is to prevent multiple copies of the same point to appear in both the training set and the validation set which would allow the classifier to 'cheat' on the validation set. In other words, this ensures that the validation set is independent of the training set.

- **Hybrid Oversampling and Undersampling**

  To achieve balanced data, I initially employed the methods of random oversampling and undersampling. *Oversampling* replicates existing samples belonging to the minority class; its counterpart, *undersampling*, drops samples belonging to the majority class [2].

Listing 2: Hybrid Random Oversampling and Undersampling

```
Hybrid Random Oversampling and Undersampling(dataset)
n = no. of samples in the data set
c = no of classes
1. Calculate the target number of samples, t = n/c
2. Construct class list for each class : iterate over the entire data set
     and push each sample into its corresponding class list
3. Construct a new data set S and do the following :
   for each class c in class list :
      Let k = no. of samples in class c
      if (k > t):
         select t random samples without replacement from class c and
              append to S (undersampling)
      else
         select t random samples with replacement from class c and append
              to S (oversampling)
```

4. return S

While such a method achieves a balanced data set, it also introduces its own set of problematic consequences that can potentially hinder learning. In undersampling, removing samples from the majority set can cause the classifier to miss important concepts essential to classifying examples belonging to the majority class. On the other hand, since oversampling appends replicated samples to the data set, multiple instances of the same sample becomes tied to the data set leading to overfitting. [2]

- **Random Oversampling**

  To avoid the the negative consequences of undersampling, I also implemented pure random oversampling such that if the number of example of any class is smaller than the maximum number of examples in the class, oversampling is done until the number of examples in each class are equal.

Listing 3: Random Oversampling

Random Oversampling(dataset)

1. Calculate the number of samples per class and set max = maximum no. of samples among all classes
2. Construct class list for each class : iterate over the entire data set and push each sample into its corresponding class list
3. Construct a new data set S and do the following :

   for each class c in class list :

   Append each sample in class c to S

   Let n = the number of samples in c, and set iter =n

   while (iter < max)

```
        select a random sample with replacement from class c and append
            to S
        iter = iter + 1
4. return S
```

2.2.4 Training and Validation Errors

Generally, we expect the validation errors to be higher than the training error due to the fact that parameters were optimized based on the training set. We also expect the training set to be a decreasing convex function. This is true for the Figure 3 which shows the error vs. epoch plot for the the data partition $k = 3$ (2/3 training set, 1/3 validation set). Here, both validation error and training error are decreasing in time with the validation error being greater than the training error.

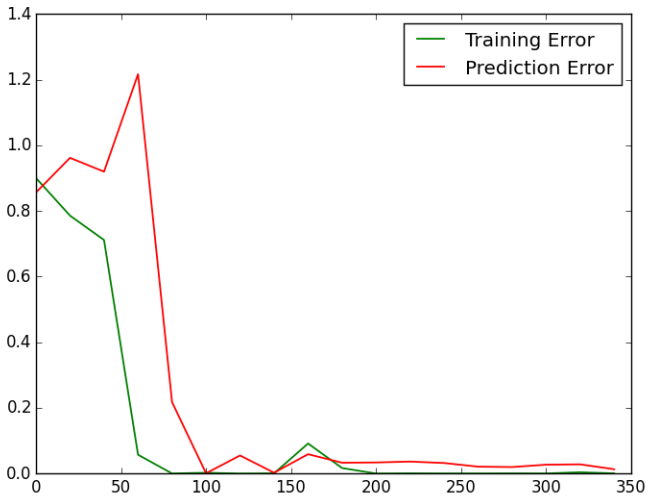

Figure 3: Training Error and Prediction Error vs. Epoch for k = 3

On the other hand, I have also observed results in which overfitting has

occurred. Figure 4 shows a plot for k = 7 (6/8 training set, 1/7 validation set) wherein the validation error is high and fluctuating wildly. While it was able to classify the training set properly, it did not perform well for the independent validation set suggesting that it has learned-by-heart the behavior of the training set, a characteristic of overfitting. The occurrence of overfitting can be attributed to the oversampling method used to handle the imbalanced data set. Oversampling duplicates samples in the minority set which can potentially cause the classifier to be tied down to fit these specific replicate samples.

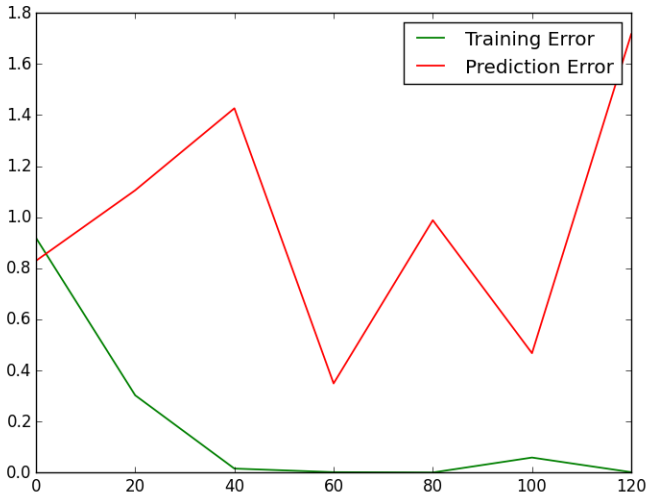

Figure 4: Overfitting: Training Error and Prediction Error vs. Epoch for k = 7

2.3 Cost-Sensitive Decision Trees

Cost-sensitive decision trees was chosen as our second classifier . Cost-sensitive classifiers have the capacity to state the cost of misclassification of the different classes, making them suitable for classifying highly unbalanced datasets.

2.3.1 Cost Matrix

Learning with cost-sensitive algorithms is important in handling imbalanced data sets because it considers the cost of misclassification wherein the the goal to minimize the total cost [4]. Constructing a cost matrix (or error matrix) allows one to assign error weights to misclassification and is particularly useful when specific classification errors are more severe than others. For an unbalanced data set, misclassification of a rare event (or minority class) should be assigned higher costs in able to reduce errors.

The cost matrix for the unbalanced data set used in this PA used can be seen in Figure 5. The cost of misclassification is greater for classes belonging to the minority set (i.e. classes 3, 7, 2, 5, and 6) and the the value of their weights depend on how scarce the samples appear in the data set (see Table 1 for class proportions).

		Predicted Classes							
		Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8
Actual Classes	Class 1	0	0	0	0	0	0	0	0
	Class 2	50	0	50	50	50	50	50	50
	Class 3	100	100	0	100	100	100	100	100
	Class 4	0	0	0	0	0	0	0	0
	Class 5	20	20	20	20	0	20	20	20
	Class 6	10	10	10	10	10	0	10	10
	Class 7	90	90	90	90	90	90	0	90
	Class 8	0	0	0	0	0	0	0	0

Figure 5: Cost Matrix

2.3.2 C5.0 Algorithm

Ross Quinlan was known for developing tree-based models (e.g. ID3 and C4.5). Quinlan continually worked on classification tree and rule-based models, and in the 1980s created C5.0, an extension of C4.5. In this PA, I use the C5.0

package developed by Kuhn, Weston, and Coulter in R to build predictive decision trees [3].

2.3.3 C5.0 Implementation in R

In this section, I will discuss the R code implementation of predictive decision tree-based models using the C5.0 algorithm. To install C5.0, I run the R system and load the C5.0 package using the command:

Listing 4: Installing C5.0 Package in R

```
1 # Install  C5.0 Package
2 install . packages("C50")
3 #Load C5.0 Library
4 library (c50)
```

To load the datasets, I simply use read.csv() as follows:

Listing 5: Load Data Set

```
1 data <- read.csv("data.csv", header=FALSE, sep=",")
2 label <- read.csv("data_labels .csv", header=FALSE, sep=",")
3 testingData <- read.csv("test_set .csv", header=FALSE, sep=",")
```

I then split the data into two sets: trainingSet and validationSet. I split it at a 70:30 ratio (70% of the data is the trainingSet while the remaining 20% is the validationSet).

Listing 6: Data Partition

```
#Append trainingLabel to trainingData
data["label"] <- label
```

```
#Split data into training set and testing set 70:30
validationSet <- data[2441:3486,]
trainingSet <- data[1:2440,]
```

I also define the cost matrix a given in Figure 5 as follows

Listing 7: Define Cost Matrix

```
cost_matrix <- matrix(
  c(0, 0, 0, 0, 0, 0, 0, 0,
    50, 0, 50, 50, 50, 50, 50, 50,
    100, 100, 0, 100, 100, 100, 100, 100,
    0, 0, 0, 0, 0, 0, 0, 0,
    20, 20, 20, 20, 0, 20, 20, 20,
    10, 10, 10, 10, 10, 0, 10, 10,
    90, 90, 90, 90, 90, 90, 0, 90,
    0, 0, 0, 0, 0, 0, 0, 0),
  nrow=8, ncol = 8)
```

I now process the data by calling C5.0.default or C5.0() function. According to the documentation of the C5.0 Package, C5.0() takes primarily 2 arguments: a data frame or matrix of predictors, x and a factor vector with 2 or more levels, y.

Listing 8: Build decision tree model

```
treeModel <- C5.0(label ~ ., data = trainingSet, cost=cost_matrix)
```

To increase accuracy of data, I employ a technique called boosting. Boosting aids to increase accuracy of the tree model by adding weak learners such that new learners pick up the slack of old learners.

Listing 9: Build boosted decision tree model

```
boostTreeModel <- C5.0(label ~ .,  data = trainingSet,  trials   = 5,
    cost=cost_matrix)
```

To predict the labels for test data using the tree model with boosting, I invoke

Listing 10: Testing

```
pred <- predict(boostTreeModel, validationSet)
```

The accuracy can be calculated as follows

Listing 11: Calculate accuracy of the model

```
accuracy = sum(pred == validationSet$label) / length(pred)
```

3 Results and Analysis

The cost-sensitive C5.0 decision tree performs better than the ANN with a 90.73% and a running time of merely 1 minute and 20 seconds. This suggests that using a cost matrix to minimize costly misclassifications works better on highly imbalanced data sets than the method of using random resampling. The hybrid oversampling and undersampling performs the worst with an accuracy of 87% and a running time of 3 minutes and 19 seconds. Thish can be at-

	Method for handling unbalanced data set	Accuracy	Running Time
ANN	Hybrid Oversampling and Undersampling	87.04%	3 min 19 s
ANN	Random Oversampling	88.74 %	2 min 41 s
C5.0 Decision Tree	Cost Matrix	90.82 %	1 min 20 s

tributed to the combined negative consequences associated with oversampling and undersampling.

4 Conclusion

In this programming assignment,ANN using Backpropagation algorithm was implemented and used to classify a highly imbalanced data set. Techniques used to handle the imbalanced data set include hybrid oversampling and undersampling, random oversampling, and cost-sensitive learning. For cost sensitive learning, a C5.0 decision trees with an associated cost matrix was used as a second classifier to classify the data set.

Results show that the cost-sensitive learning using decision trees performs better than random resampling with ANN with an accuracy of 90.73%. Future work include using SMOTE (Synthetic Minority Over-sampling Technique) [1] to generate artificial data points which could be used to improve the results of ANN and may possible fix the problem of overfitting due to duplicate samples in the data set.

References

[1] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, pages 321–357, 2002.

[2] Haibo He, Edwardo Garcia, et al. Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on*, 21(9):1263–1284, 2009.

[3] Max Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software*, 28(5):1–26, 2008.

[4] Charles X Ling and Victor S Sheng. Cost-sensitive learning. In *Encyclopedia of Machine Learning*, pages 231–235. Springer, 2010.