AMATH 582: Final Project

Lee Burke
Taylor Cunnington

March 16th, 2017

Abstract

Sec. I Introduction and Overview

Classifying bank transactions as fraudulent is a difficult task: (1) data is highly sensitive and so difficult to come by, (2) relevant features are not always clear, and (3) classification algorithms have trouble with unbalanced data where fraud is much less common than honest purchases.

To answer the first problem, we analyze real credit card data from a challenge on link?. However, to preserve confidentiality, the features have been obscured by a singular value decomposition (SVD): we are given $U\Sigma$ if the original data $A = U\Sigma V^*$ in the usual SVD. That is, we have no say in the second problem, feature extraction. Indeed, the PCA modes have no discernable structure, and decay slowly. Instead, we focus on classifying skewed data.

That popular algorithms have trouble with skew is, in some sense, expected: minimizing the error rate inherently favors the major class. Guessing the larger class every time is already above 50% accuracy. In addition, overall error is not the metric we care most about: false positives (mistakenly flagging a transaction as fraud) have much lower cost to the bank and consumer than false negatives (someone gets away with fraud). Our dataset is a great example of imbalanced data. Only 0.17% of the data points are classified positive. It is difficult to improve on 99.83% accuracy!

We apply several methods learned in lecture, including linear discriminant analysis (LDA), a binary classification tree (BCT), and a support vector machine (SVM). A new method is used to reduce the skew in the class sizes in combination with LDA. We focus especially on how to compare the performance of these classifiers, because common performance metrics may not be as descriptive with high-cost, highly-skewed data.

Sec. II Theoretical Background

A brief theoretical exploration of each classification scheme and its evaluation with respect to imbalanced data are given below.

Sec. II.1 Classification

Sec. II.1.1 Linear Discriminant Analysis

The LDA algorithm, `fitcdiscr` in MATLAB, builds normal probability distributions for each class with mean and covariance parameters $(\vec{\mu}_0, \sigma_0)\,(\vec{\mu}_0, \sigma_0)\,and\,(\vec{\mu}_1, \sigma_1)\,(\vec{\mu}_1, \sigma_1)$ These are referred to as the prior distributions. Classifying subsequent observations is then potentailly just a matter of comparing their probability of being in class 0 or 1 and applying a threshold. MATLAB actually attempts to minimize the cost function:

$$\hat{y} = \arg \min_{y=1,\dots,K} \sum_{k=1}^{K} \hat{P}(k|x) C(y|k) \tag{1}$$

where $\hat{y}$ is the predicted probability. K is the number of classes. $\hat{P}$ is the posterior probability of class k for a given observation x, and $C(y|k)$ is the cost of misclassifying k as y. $\hat{P}$ is determined from Bayes

rule:

$$\hat{P}(k|x) = \frac{P(x|k)P(k)}{P(x)} \tag{2}$$

Here the posterior probability that x is in class k is what LDA estmiates from the training data. $P(k)$ and $P(x)$ are readily calcuated the from the data. Note that for the case of an extremely small minority class such as ours $P(n)$ will be extremely small (where $k = n$); this means that $\hat{P}(n|x)$ will be small as well. This is a source of bias for LDA toward the majority class.

Sec. II.1.2 Binary Classification Trees

Binary decision trees are a common and effective choice for classification tasks. The MATLAB command to build a binary classification decision tree model, `fitctree`, uses an algorithm called CART, for Classification and Regression Tree. You classify an unknown observation by starting at the root node and at each node in the tree you make binary decision (meaning there are two branches) based on the features of the data used for training. You work your way up the tree until you hit a terminal leaf node. This node determines your prediction.

You can think about this process as splitting up m-dimensional space into "blocks" (if you imagine the 3-dimensional version), where m is the number of features and each block represents a class prediction. It starts with the full space and splits it into two, then one of those subspaces in two, and then one of those three subspaces into two, and onwards until it finishes. The algorithm chooses where to put the boundaries based on minimizing the Gigi index:

$$G = \sum_k p_k(1 - p_k) \tag{3}$$

Here p_k is the proportion of points in a subsection of class k, summed over all classes. Note that the minimum you can acheive is 0 for a pure class and 0.50 for half of each class. It calculates the Gini index for all the potenatial split points and chooses the lowest one in a greedy fashion–it does not consider global error. The criterion for no longer continuing to split the subspaces is often that you have too few training instances at a node.

In the case of imbalanced data p_k will always be very small for the minority class in the general case where the classes are highly overlapping. Even if the algorithm has narrowed down to a box that just barely contains all the minority class observations and just a small proportion of the majoirty class observations, it is still very possible that the number of majority class points in that space will be of a similar number or greater than the minority class. The Gini index naturally gives high weight to the more represented class, which is a bias for the majority class.

Sec. II.1.3 Support Vector Machines

Sec. II.2 Handling Class Skew

There are some known methods for improving performance when classifying imbalanced data. The four common solution types are given here:

1. Under-sampling – Removing data from the larger class to balance the class sizes. May lead to a poorer choice of decision line due to losing data at the border of the classes.
2. Oversampling – Adding extra observations on top of existing minority class observations to balance out the class sizes. May lead to overfitting with some classification models.
3. Synthetic data generation – Generating artificial data from your existing data to balance classes. Generated data generally stays within the n-dimensional volume that minimally encloses the existing data.
4. Cost functions– Classification algorithms use cost functions (decision functions) to define their decision boundaries. With imbalanced data you would set the misclassification of the minority class to be much more costly, to encourage the algorithm to classify them correctly more often than the majority class.

The first three attempt to reduce the imbalance by reducing the number of majority data point or increasing the number of minorty data points. These methods deal with the data only, not the classifier. The final option does alter the classifier however, to try to make it classify the minority class more reliably.

Sec. II.3 Performance Evaluation

Sec. II.3.1 Receiver Operating Characteristics (ROC) Curve

Sec. II.3.2 Precision/Recall (PR) Curve

Sec. III Algorithm Implementation and Development

As stated above our data was prepared and features provided as U columns of the SVD decomposition, so we did not alter or clean the data. We created a "full" data matrix X from the 28 columns of U and the amount of the transaction. We used standard Z-score of the X matrix, Z, for all implementations of the LDA classifier.

Then we split the data for cross validation... or did we do k-fold validation??

Using LDA as our classification algorithm we compared the synthetic data generation and alternate cost function approaches to the default algorithm. For synthetic data generation we chose to use ADASYN (Adaptive Synthetic Sampling Approach for Imbalanced Learning) a generation algorithm similar to the well known SMOTE (Synthetic Minority Over-Sampling Technique) algorithm, which is ADASYN's precursor. The goal of both is to improve the class balance by increasing the number of minority class members. SMOTE places synthetic data between existing data points randomly (linear interpolation), with no preference shown to any specific points. ADASYN does the same thing but places more synthetic data points close to the boundary between classes because those are the original data points that are more difficult to learn. (Does this favor decision trees or SVMs or something? I imagine that all this would do for an LDA is to move the mean of the minority gaussian close to the boundary... SMOTE might be better here...)

We also tried changing the cost function of our LDA model to discourage FNs. Since LDA works by creating probability distributions the cost comes into play only when making predictions. Each observation that it is trying to predict has a calculated posterior probability. It tries to minimize the "classification cost"–this is a decision cost function. Unfortunately if an observation has an extremely small posterior probability then even with a vary large cost for miscalculating that observation it may not change the classification cost by much, meaning there will be little change to the resulting prediction.

Sec. IV Computational Results

GIVE BASELINE LDA RESULTS

Altering the prediction cost function does not make a difference in our case. Weighting false negatives as very costly makes very little difference, even when the cost of a false negative is 10^8 as costly as a false positive. This seems odd, since we only expect class p to be about 500 times as likely class n ($P(p) \approx 500 \times P(n)$). There must be something else going on... PERFORMANCE MATRICS

Synthetic data generation did improve recall. As expected this came at the cost of precision and accuracy. PERFORMANCE METRICS

Table 1:

	AUCs	AUCPR
LDA	0.976	0.124
LDA w/ADASYN	0.980	0.538
Binary Classification Tree	0.868	0.416
SVM	0.905	0.660

Sec. V Summary and Conclusions

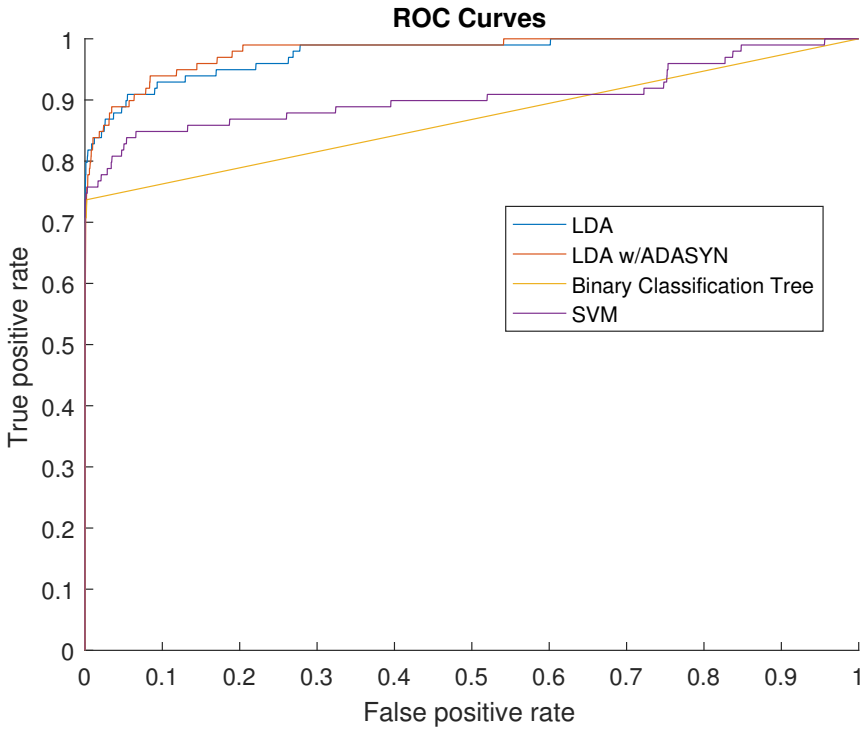

Figure 1:

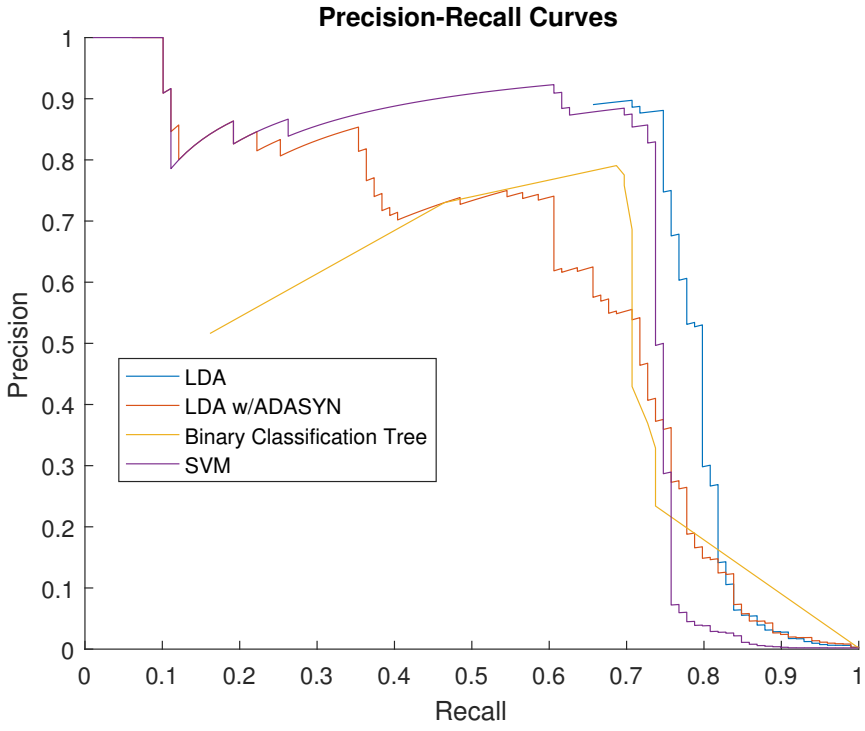

Figure 2:

References

[1] https://www3.nd.edu/ dial/publications/dalpozzolo2015calibrating.pdf

[2] wiki: ROC, AUC

[3] http://machinelearningmastery.com/classification-and-regression-trees-for-machine-learning/

Appendix A MATLAB functions used and brief implementation explanation

- fitctree
- fitcsvm
- fitcdiscr
- ADASYN
- predict
- perfcurve

Appendix B MATLAB codes

Listing 1: Main

```
clear,close all
tic
disp('Loading data and initializing')
%% Parameters
% Ratio of training to test subsets
ratio = 0.8;
% Desired ratio of class 0 to class 1 for ADASYN
classBalance = 0.1;
% Cost of false negative for cost models
costs = [10 100 10^5 10^10];

%% Initialize variables
% amount  | amount of transaction in cents
% time    | time of transaction in seconds
% class   | boolean fraudulent/legitimate-->1/0
% data    | features (US in svd USV')

if exist('data.mat', 'file') == 2
    load('data.mat')
else
    readData
end

data = [data amount]; % time does not seem to improve results

%% Test and train subsets
[trainData, testData, trainClasses, testClasses] = ...
    splitBinaryClassData(ratio, data, class);

%% Undersampling / Oversampling / ADASYN
disp(['Running ADASYN (',num2str(toc),')'])
% Scaling is required for ADASYN's internal KNN search
Ztrain = zscore(trainData);
Ztest = zscore(testData);
[synZ, synClassOut] = ADASYN(Ztrain, trainClasses, classBalance);
synFull = [Ztrain; synZ];
```

```
synClass = [trainClasses; synClassOut];

%% Classify
disp(['Building/ running classification models (',num2str(toc),')'])
% Binary classification decision tree
disp(['Binary classification decision tree (',num2str(toc),')']);
tree = evaluate(fitctree(Ztrain,trainClasses), Ztest, testClasses);

% Support vector machine
disp(['Support vector machine (',num2str(toc),')']);
svm = evaluate(fitcsvm(Ztrain, trainClasses), Ztest, testClasses);

% Linear discriminant analysis
disp(['Linear discriminant analyses (',num2str(toc),')']);
lda = evaluate(fitcdiscr(Ztrain, trainClasses), Ztest, testClasses);

% LDA with synthetic data -- train with syn data but test with original
ldaSyn = evaluate(fitcdiscr(synFull, synClass), Ztest, testClasses);

% LDA with custom cost function
costModel = fitcdiscr(Ztrain, trainClasses,'ClassNames',[0,1]);
for i=1:length(costs) % TODO is something wrong here?? all perf. values are ...
    exactly the same as vanilla LDA......
    costModel.Cost = [0 1; costs(i) 0];
    ldaCosts(i) = evaluate(costModel, testData, testClasses);
end

disp(['Done classifying (',num2str(toc),')'])


%% Plot
% lda
plotModels = {lda};
modelNames = {'LDA'};
AUCs = lda.AUC;
AUCPRs = lda.AUCPR;
% ldaSyn
plotModels = {plotModels{:},ldaSyn};
modelNames = {modelNames{:},'LDA w/ADASYN'};
AUCs = [AUCs;ldaSyn.AUC];
AUCPRs = [AUCPRs;ldaSyn.AUCPR];
% tree
plotModels = {plotModels{:},tree};
modelNames = {modelNames{:},'Tree'};
AUCs = [AUCs;tree.AUC];
AUCPRs = [AUCPRs;tree.AUCPR];
% svm
plotModels = {plotModels{:},svm};
modelNames = {modelNames{:},'SVM'};
AUCs = [AUCs;svm.AUC];
AUCPRs = [AUCPRs;svm.AUCPR];
% ldaCosts
for i=1:length(costs)
    plotModels = {plotModels{:},ldaCosts(i)};
    modelNames = {modelNames{:},['LDA w/Costs ',num2str(i)]};
    AUCs = [AUCs;ldaCosts(i).AUC];
    AUCPRs = [AUCPRs;ldaCosts(i).AUCPR];
end

T = table(AUCs,AUCPRs,'RowNames',modelNames)

% ROC Curve
figure
hold on
for i=1:length(plotModels)
```

```
    mdl = plotModels{i};
%     ...
    errorbar(mdl.rocX,mdl.rocY(:,1),mdl.rocY(:,1)-mdl.rocY(:,2),mdl.rocY(:,3)-mdl.rocY(:,1));
    plot(plotModels{i}.rocX,plotModels{i}.rocY);
end
hold off
legend(modelNames,'Location','Best');
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC Curves')

% PR Curve
figure
hold on
for i=1:length(plotModels)
    mdl = plotModels{i};
%     ...
    errorbar(mdl.prX,mdl.prY(:,1),mdl.prY(:,1)-mdl.prY(:,2),mdl.prY(:,3)-mdl.prY(:,1));
    plot(plotModels{i}.prX,plotModels{i}.prY);
end
hold off
legend(modelNames,'Location','Best')
xlabel('Recall')
ylabel('Precision')
title('Precision-Recall Curves')









%% Explore feature space
% Singular values and POD modes
% S = zeros(size(data,2),1);
% U = zeros(size(data));
% for i=1:size(data,2)
%     S(i)=norm(data(:,i));
%     U(:,i)=data(:,i)/S(i);
% end
% plot(S)
% plotFeatureSpace(U(class==1,:), U(class==0,:), [1 2 3]);
% plotFeatureSpace(posData, negData, [3 10 29]);
%% Run model nIter times
% nIter = 1;
% disp(['Starting trials (',num2str(toc),')'])
% for i=1:nIter
%     disp(['Running test ',num2str(i),' out of ',num2str(nIter)])
%     t = tic;
%     t = toc(t);
%     disp(['Test ',num2str(i),' took ',num2str(t),' seconds.'])
% end
% % NOTE: if we are going to do cross validation MATLAB suggests:
%     cvmodel = crossval(lda,'kfold',2);
%     cverror = kfoldLoss(cvmodel)
```

Listing 2: Split data into training and test sets

```
function [train, test, trainClass, testClass] = splitBinaryClassData(ratio, ...
    data, class)
    n0 = sum(class==0);
    n1 = sum(class==1);

    data0 = data(class == 0,:);
    data1 = data(class == 1,:);

    [iTrain0, iTest0] = splitIndices(n0,ratio);
    [iTrain1, iTest1] = splitIndices(n1,ratio);

    train = [data1(iTrain1,:); data0(iTrain0,:)];
    test = [data1(iTest1,:); data0(iTest0,:)];

    trainClass = [ones(length(iTrain1),1); zeros(length(iTrain0),1)];
    testClass = [ones(length(iTest1),1); zeros(length(iTest0),1)];
end
```

Listing 3: Random permutation for indices

```
function [i_train, i_test, numTrain, numTest] = splitIndices(n, ratio)
    numTrain = floor(ratio*n);
    numTest = n - numTrain;

    % Mix it up
    ri = randperm(n);
    i_train = ri(1:numTrain);
    i_test  = ri(numTrain+1:end);
end
```

Listing 4: Performance characteristics of a classification model

```
function s = evaluate(model, testData, testClasses)
    s = struct;
    s.model = model;
    [s.labels,s.scores,s.costs] = predict(model, testData);
    s.confusion = confusionmat(testClasses,s.labels);
    [s.rocX,s.rocY,s.rocT,s.AUC] = perfcurve(testClasses,s.scores(:,2),'1');
    [s.prX,s.prY,s.prT,s.AUCPR] = ...
        perfcurve(testClasses,s.scores(:,2),'1','XCrit','reca','YCrit','prec');
%     disp('Bootstrapping ROC')
%     [s.rocX,s.rocY,s.rocT,s.AUC] = ...
    perfcurve(testClasses,s.scores(:,2),'1','NBoot',1,'TVals',0:0.05:1);
%     disp('Bootstrapping PR')
%     [s.prX,s.prY,s.prT,s.AUCPR] = ...
    perfcurve(testClasses,s.scores(:,2),'1','XCrit','reca','YCrit','prec','NBoot',1,'TVals',0:0.05:1);
end
```