Team DAB: CS74/174 Final Project

Machine Learning and Statistical Data Analysis: Winter 2016

Daniel Chen, **A**ndrew Kim, **B**enjamin Packer

Contents

1 Introduction

This project worked on finding a solution to the BNP Paribas Cardif Claims Management Kaggle competition. The competition is to use machine learning algorithms to effectively classify claims with anonymized data into two classes with minimal error:

(a) claims for which approval could be accelerated leading to faster payments

(b) claims for which additional information is required before approval

The purpose of this classification is to allow BNP Paribas Cardif to accelerate its claims process and provide a better service to its customers.

Error on both Kaggle and as presented in this paper is measured through Log Loss:

$$\text{logloss} = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(p_i) + (1 - y_i)\log(1 - p_i))$$

N is the number of observations, log is the natural log, y_i is the binary target and p_i is the predicted probability that $y_i = 1$

In the end, our solution is a ensemble of logistic regression, neural networks, and boosted trees. Our best testing error is 0.45695, which ranks 490 on the leaderboard.

2 The Data

The data given is split into training and testing. The sets are the same, except we do not know the expected targets on the testing data (Kaggle retains this information to prevent cheating).

The training set consists of 114,321 examples while the test set contains 166,607. Each example consists of 131 features, 19 which are categorical with the remaining numerical. The features are all anonymized, so we don't know what the data means.

2.1 Analysis of the Data

2.2 One Hot Encoding

We originally converted our categorical features to integer values. This confused our classifiers into believing the features expressed a numerical relationship. By switching over to a one-hot-encoding of our categorical features, we were able to improve performance.

One-hot-encoding splits each feature into n features, where n is the number of unique values that this feature takes on in both the training and testing sets. Each of these resulting features either take on 0 or 1. For each data point, only a single feature (the one which corresponds to the value that the original categorical feature had) has the value 1. This increases the number of features, but allows for correct encoding of categorical data.

One issue with one-hot-encoding was that we had to remove one of the categorical features, v22. This is because this feature had over 1000 unique values, so it increased the number of features to a number which was computation cumbersome. We prefer the empirical increase in performance to the loss of this single feature.

Our original logistic regression without one hot encoding achieved an error of 0.49859. After switching to one-hot-encoding our error significantly to 0.48213.

2.3 Missing Values

There was lots of missing data in both the testing and training sets, which was a problem for both our Logistic Regression classifier and our Neural Networks, which XGBoost could handle missing values. Our first intuition was to impute each missing data value with the mean value for each feature, which seemed to work well.

However, we also experimented with setting the missing values to the median of the missing data. Using the mean imputation strategy and our other best logistic regression parameters, we achieved a 5-fold cross validation error of 0.483757535773. Using the median achieved an error of 0.483781376194, which is worse by a negligible 0.000024. The differences in the error were very small.

3 Logistic Regression

Logistic regression was implemented using SciKit Learn's Logistic Regression classifier, which can output a probability for a given value. It allows specification of the C parameter, which is equivalent to $\frac{1}{\alpha}$, where α is the regularization parameter.

3.1 Logistic Regression Parameter Tuning

Given a simple logistic regression model, the main parameter available to us for tuning was α, or the regularization parameter. To find the optimal value for α, we used 5-fold cross validation to as the objective function, and selected the α that best minimized it. The result was $\frac{1}{\alpha} = 0.9296875$, which is a slight regularization.

The 5-fold cross validation error with $\frac{1}{\alpha} = 0.9296875$ was 0.483757535773, versus the same error where $\frac{1}{\alpha} = 1$ which was 0.483758005391. This is quite an insignificant difference, which indicates that our logistic regression classifier was not overfitting to begin with.

3.2 Performance

The final performance on the testing data was 0.48213, slightly better than our best Neural Network performance but significantly worse than our best XGBoost performance. However, this does not mean the model has no utility, since it was useful as part of the final ensemble.

4 Boosted Trees

Following suggestions on the forum, we implemented boosted trees using the XGBoost library, which stands for eXtreme gradient boosting. The library is designed and optimized for boosted tree algorithms. We utilized the Python package for this library. The classifier uses an ensemble of trees.

The seven parameters that we optimize are shown below:

max_depth The maximum depth of a tree in the ensemble

min_child_weight Defines the minimum sum of weights of all observations required in a child

gamma The minimum loss reduction required to make a split

colsample_bytree The fraction of columns to be randomly sampled for each tree

subsample The fraction of observations to be randomly samples for each tree

eta The learning rate

num_rounds The number of rounds

4.1 XGBoost Parameter Tuning

The parameters were tuned using two-fold cross validation. After playing around with tuning the parameters manually, we ran two large sets of parameters in an attempt to reduce error and understand the empirical relationships between the parameters.

4.1.1 Tuning 1

The first optimization iterated through all 120 combinations of these parameters:

```
max_depths = [2, 3, 4, 5, 6]
min_child_weights = [1]
gammas = [0, 1]
colsample_bytrees = [0.5, 1]
subsamples = [0.5, 1]
rounds_and_eta = [(20, 0.3), (50, 0.1), (100, 0.05)]
```

With this, our best results are shown below:

error	runtime	minchildweight	subsample	eta	colsamplebytree	max depth	gamma
0.468638593	347.661603	1	1	0.05	0.5	6	0
0.468680062	346.063396	1	1	0.05	0.5	6	1
0.468846706	177.2962441	1	1	0.1	0.5	6	1
0.468898392	178.4002779	1	1	0.1	0.5	6	0
0.469002588	661.6131201	1	1	0.05	1	6	1

The testing error found using the best parameters from the tuning was: 0.46853

4.1.2 Tuning 2

Our second optimization iterated through all 24 combinations of these parameters:

```
max_depths = [6, 8, 10]
min_child_weights = [1, 2]
gammas = [0]
colsample_bytrees = [0.5, 1]
subsamples = [1]
rounds_and_eta = [(200, 0.05), (300, 0.01)]
```

With this, our best results are shown below:

error	runtime	minchildweight	subsample	eta	colsamplebytree	max depth	gamma
0.464182985	1191.265073	1	1	0.05	0.5	10	0
0.464419596	1154.138998	2	1	0.05	0.5	10	0
0.46442771	959.313591	2	1	0.05	0.5	8	0
0.464540668	1020.961268	1	1	0.05	0.5	8	0
0.466287292	760.0540562	2	1	0.05	0.5	6	0

The testing error found using the best parameters from this tuning was: 0.47856

Since this testing error is greater than the testing error achieved from the first tuning, there is evidence that somewhere between the first tuning and the second tuning we began to overfit the training data.

4.1.3 Final Selection of Parameters to be Ensembled

Although we did not end up using the optimal results from our tuning, it provided valuable insights to how we can use the parameters to achieve better results. Ultimately the results from our tuning was leading us to areas where this kind of brute force technique towards parameter tuning would become too computationally expensive, as tuning 2 already took nearly 12 hours to run.

Mostly we spent time modifying eta, max_depth, and colsample_bytree.

Ultimately, the parameters that we came up with were:

```
max_depths = 16
min_child_weights = 1
gammas = 0
colsample_bytrees = 0.68
subsamples = 0.5
num_round = 1800
eta = 0.01
```

This gave us a testing error of 0.45695 and the training was 0.286874075363. The runtime was approximately 4 hours.

5 Neural Networks

The Kaggle forums indicated that there was not much success using neural networks for this dataset. However, we attempted to build neural network models to help build the ensemble and to gain experience building neural nets. We trained a variety of different neural network architectures using the Theano Python libraries. We also used Lasagne which is a module built on top of Theano. All of the neural network structures has 498 input units and 2 output units. The 2 output units are softmax units, which have output between 0 and 1. We used rectified linear units (ReLU) in the hidden layer because we trained deep neural networks, which works well with ReLU. [1]

The lowest testing error we got was 0.48274, which was worse than XGBoost's.

5.1 One-Layer Network

We first trained two-layer networks. We tested different numbers of units in the hidden layer. Since there are 498 input features, the first layer has 498 units and there are two output units. We read online that it is generally better to normalize the input data before feeding it into the network, so we normalized the data. We used 80% of the training data for training and cross-validated on 20% of the data. The picture below is a picture of the terminal output when training the neural network. The lasagne package prints out its own values for cross-validation and training error each epoch as it trains itself on the training dataset. You can see that we should have stopped training the neural network after epoch 14 since the cross-validation error consistently when up after that epoch, meaning that the network started to overfit. At the end of training, the python script prints out the log-loss error on the cross-validation set.

epoch	train loss	valid loss	train/val	valid acc	dur
1	0.55564	0.51338	1.08231	0.75863	0.81s
2	0.50559	0.50116	1.00885	0.76436	0.77s
3	0.49546	0.49656	0.99777	0.76693	0.87s
4	0.48978	0.49411	0.99124	0.76808	0.75s
5	0.48589	0.49253	0.98652	0.76846	0.74s
6	0.48284	0.49145	0.98249	0.76955	0.76s
7	0.48030	0.49065	0.97891	0.77000	0.75s
8	0.47808	0.49003	0.97561	0.76984	0.74s
9	0.47609	0.48958	0.97245	0.76968	0.74s
10	0.47426	0.48922	0.96942	0.76913	0.75s
11	0.47255	0.48897	0.96643	0.76935	0.74s
12	0.47095	0.48879	0.96351	0.76951	0.74s
13	0.46944	0.48869	0.96061	0.76974	0.75s
14	0.46799	0.48866	0.95771	0.76913	0.80s
15	0.46659	0.48871	0.95473	0.76880	0.76s
16	0.46522	0.48882	0.95172	0.76885	0.74s
17	0.46388	0.48898	0.94867	0.76891	0.75s
18	0.46255	0.48918	0.94556	0.76873	0.74s
19	0.46122	0.48944	0.94233	0.76884	0.75s
20	0.45989	0.48976	0.93899	0.76933	0.74s
21	0.45855	0.49013	0.93558	0.76863	0.74s
22	0.45722	0.49050	0.93215	0.76820	0.75s
23	0.45589	0.49092	0.92863	0.76754	0.74s
24	0.45454	0.49140	0.92500	0.76748	0.74s

^CTraining runtime: 0.0mins, 18.8697960377s.
Log loss: 0.4978

Below are the results for different numbers of hidden units. As you can see, all three networks are clearly overfitting since the cross-validation error is higher than the training error. Therefore, we took steps to try to address this issue.

Hidden Units	cv log-loss	training log-loss	epochs
150	0.4978	0.45454	25
100	0.493	0.46028	25
50	0.4923	0.46281	25

5.2 Dropout

In order to prevent overfitting, we implemented a neural network with dropout layers. Though dropout is a technique particularly useful for deep neural networks, we tried the dropout layers to see its effect. Dropout prevents neural networks from overfitting by randomly dropping units and their connections with a probability, p.[2] The results are below for the same networks as above with dropout implemented in the hidden layer with probability of dropout set to 0.5. Including dropout causes the training error to decrease more slowly, but the the cross-validation error decreased for all three networks.

Hidden Units	cv log-loss	training log-loss	epochs
150	0.4844	0.47525	100
100	0.4845	0.47909	100
50	0.4831	0.47761	100

5.3 Two-Layer Network

Rather than implementing PCA for feature selection and then using those features as input to the 1-hidden layer network, we tried implementing a two-layer network because deep neural networks select the relevant features itself. We used dropout in both layers because when testing with dropout in only one layer, overfitting occurred. The probability in the dropout layers was set to 0.5, the default value.

Layer 1 Hidden Units	Layer 2 Hidden Units	cv log-loss	training log-loss	epochs
200	70	0.4849	0.48191	100
200	50	0.4806	0.48322	100
200	70	0.4890	0.48430	100
150	100	0.4854	0.48089	100
150	50	0.4824	0.48540	100
150	20	0.4870	0.48982	100
100	50	0.4866	0.48389	100
100	20	0.4924	0.48718	100

As you can see in the results, we achieved the lowest cross-validation error with the neural network with 200 hidden units in the first layer and 50 units in the second layer. What was interesting was that the training log-loss error was higher than the cross-validation error for some networks. This may be because the validation set contains data that is ëasierẗo predict or excessive regularization due to dropout. In general, a higher number of parameters results in the neural network to overfit on the training data, but regularizing prevents overfitting.

5.4 Batch Normalization

To experiment, we built a three-layer neural network using dropout and batch normalization. Since the number of parameters is higher in a three-layer network, we used batch normalization to address the internal covariate shift problem. [3] The network had 200 hidden units in the first layer with batch normalization, 80 hidden units in the second layer with dropout, and 20 hidden units in the third layer with dropout. The cross-validation error was 0.4836 and the training error was 0.46089 after 75 epochs. We wanted to explore this approach further and tune parameters, but the long training times made it difficult to do so.

6 Ensemble

6.1 Motivation

Given that we had some reason to believe our best-performing model, XGBoost, was overfitting the data, we decided to attenuate that overfitting by including it in an ensemble with the Neural Networks and Logistic Regression. Using a simple weighted average where the output is the weighted average of the outputs of the models in the ensemble, we were able to experiment with the different weights.

We decided against using a boosting ensemble method, since such a method was likely to only overfit the data more. Furthermore, we were initially able to dramatically decrease our training error by optimizing the weights in the weighted average to produce the best training error using SciPy's minimization library, but then realized such a process was drastically overfitting the results of the training data when it produced a large increase in our testing result. Thus, we found that since our ensemble was chosen to correct XGBoost's overfitting, subtle and small manual manipulation of the weights from equal voting was the best way to do it.

6.2 Results

However, we were unable to improve our results beyond what XGBoost achieved. Using weights $[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$ gave a testing error of 0.46692 (worse than our best by 0.0099), and using weights $[.6, .2, .2]$

gave a testing error of 0.46017 (worse than our best by 0.0032). Thus, it seems that as we weight the XGBoost model more in the ensemble, it achieves a better testing result, which is simply telling us that XGBoost is the best model and that the other two models do not correct whatever overfitting remains.

7 Result

This project worked on finding a solution to the BNP Paribas Cardif Claims Management Kaggle competition. The competition is to use machine learning algorithms to effectively classify claims with anonymized data into two classes with minimal error:

(a) claims for which approval could be accelerated leading to faster payments

(b) claims for which additional information is required before approval

We tried using XGBoost (extreme gradient boost), logistic regression, and neural networks, making use of Python machine learning libraries including xgboost, scikit-learn, scipy.optimize, lasagne, and theano.

We achieved our best result log-loss error (0.45695, ranking 490 on the Kaggle leaderboard) using the XGBoost package and algorithm with final parameters:

```
max_depths = 16
min_child_weights = 1
gammas = 0
colsample_bytrees = 0.68
subsamples = 0.5
num_round = 1800
eta = 0.01
```

We preprocessed the data using OneHotEncoding for categorical variables so that they could be treated numerically, and impute missing data using the mean strategy for our logistic regression and neural network models. Our best Logistic Regression results were achieved with a regularization parameter of 0.9296875. Our best neural network results were achieved with 2 layers, 200 units in the first hidden layer, and 50 unites in the second layer.

We suspected our extreme gradient boost model of overfitting because its training error was significantly lower than its testing error, so we employed a bagged ensemble approach averaging all 3 models. However, our other 2 models were sufficiently worse than XGBoost that including them only yielded an increase in error.

8 Responsibilities

8.1 Daniel Chen

Daniel implemented the One Hot Encoding, which improved the performance of our classifiers by encoding the categorical data correctly. In addition, he implemented the boosted trees using XGBoost, and also ran the parameter tuning for that classifier. He assisted with the implementation of cross validation, especially K-Fold. For all of these contributions, he wrote up the corresponding sections in this report. In addition to those sections, he wrote up the introduction and data sections.

8.2 Andrew Kim

Andrew implemented the neural networks using a variety of different network architectures and tuning techniques. He researched papers on how to improve neural network training. He helped implement the cross-validation fold and wrote the code that normalized the dataset. He wrote the sections on these areas in this report.

8.3 Benjamin Packer

Ben implemented the missing values imputer, the process of producing an ensemble given several output csv's, some additional code to optimize the weights the ensemble assignmed each classifier, the logistic regression classifier including the parameter tuning and optimization, and the initial helper functions for reading and writing data from and to csv's. He has written the corresponding sections for the ensemble model, the logistic regression classifier, and the section on missing values.

References

[1] Hinton, Geoffrey E. "Rectified Linear Units Improve Restricted Boltzmann Machines." (2010).

[2] Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." Journal of Machine Learning (2014): 1929-958. Web.

[3] Ioffe, Sergey, and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training B Y Reducing Internal Covariate Shift." (2015).