University of Texas at Austin, Cockrell School of Engineering
Data Mining – EE 380 L

Final Project
Amazon Employee Access Challenge

May 10, 2016

Gabrielson Eapen (EID: EAPENGP)
Mudra Gandhi (EID: MG54386)

Table of Contents

Introduction

This is a Final Project Report for the Option III Data Mining Class at UT Austin. For this project, we were to tackle the Amazon Employee Access Challenge which is an expired Kaggle competition from 2013. The challenge is to predict if an Amazon employee will be given access to a requested resource. We are provided with an anonymized training and testing dataset. Since this is an expired competition we have access to both the winner's code and their winning scores. In addition, we have the option of applying our models on the testing dataset and generating a submission file that can be uploaded to Kaggle for evaluation and feedback is provided in terms of a private score. More on that in sections to follow. Finally, our professor has requested that we do not make more than one hundred submissions and we have honored that request.

What is the Amazon Employee Access Challenge?

This challenge is basically about predicting an employee's access to any resource given his or her job role. The data consists of real historical data collected from 2010 & 2011. Employees were manually allowed or denied access to resources over time. Our task is to develop an algorithm or model capable of learning from this historical data and then predict approval/denial for an unseen set of employees. Sounds pretty straightforward but is it? We hope to find out in the course of this project.

Since we are to predict an ACTION value that is either 0 or 1 (binary outcomes), this type of classification problem is simply a binary classification problem. According to Wikipedia, "Binary or binomial classification is the task of classifying the elements of a given set into two groups on the basis of a classification rule." Wikipedia also mentions that "an important point is that in many practical binary classification problems, the two groups are not symmetric – rather than overall accuracy, the relative proportion of different types of errors is of interest." This is something we need to keep an eye out for.

What data do we have?

We have been provided with two data sets. A training dataset (train.csv) that contains eight features plus the identifier (RESOURCE ID) for the target resource the access was requested for. The training dataset includes an ACTION column (Ground Truth) that reflects whether access was granted or not. Table A1 in Appendix A describes each feature in greater detail. Figure A2 in Appendix A summarizes the contents of the training dataset. In total we have 32,769 samples and each supplied categorical feature has a numeric value with no NULL entries. Figure A3 in Appendix A provides summary statistics like mean, standard deviation, and the quartile spread (distribution) of the data. One thing that quickly stands out is that we may have only less than 25% of access denials (ACTION = 0) as the 25^{th} percentile value for ACTION is 1. This could mean that we have an imbalanced training dataset in our hands and that necessitates more tweaking of the models we use.

Figures A4 and A5 similarly describe and summarize the testing dataset (test.csv) that was provided. We have 58,921 samples present in this dataset. Obviously there is no ACTION column. But in its place we have an "id" column that stands for the Employee's ID. This is also the id value to be used in the submission file we generate along with the predicted ACTION value. It is interesting that the computed standard deviation of each features in the training dataset are very similar to the standard deviation of the corresponding feature in the testing dataset. Another quick observation on both datasets is that feature column has a high cardinality.

How do we Test our Model?

Kaggle permits us to upload a CSV submission file that contains all 58,921 employee IDs found in the testing dataset along with its associated "predicted" ACTION value that was determined by our model. Kaggle then provides feedback on the quality of the predictions in terms of a "Private Leader board Score" for expired competitions and a Public Leader board Score for competitions that are actively running. The Kaggle Member FAQ[1] explains this in greater detail but the essential difference is that the Public score posted during competition is what is determined from a part (between a quarter and a third) of your dataset and this done to guard against model overfitting to the test dataset.

What metric does Kaggle use for providing feedback?

We have an interesting story to share first. Before researching the specific metric Kaggle uses, we decided to put it through what we dubbed a "laymen" test. We prepared three submission files as outlined in Table 1. Very surprisingly the submission file with all 1's yielded an even Private Score of 0.5 and the other two variants (half 0's and half 1's) yielded in one case a higher score of about 0.023 and in the other case a higher score of about 0.0049. Clearly we were not able to game the system and that piqued our curiosity further. How could this diverse submission data not yield more vastly different scores? If the samples were taken as just some guess attempts, how are they equally performing poorly and yielding a score of between 0.5 and 0.52. How is the Kaggle metric protecting against random guesses? These are all valid questions that we will explore further.

Table 1: Private Score Results for Laymen test file submissions to Kaggle.

Submission File Name	File Content Description	Kaggle Private Score
LaymanFile1.csv	First 50% (29,460 samples) ACTION set to 0 and remaining rows ACTION set to 1.	**0.52329**
LaymanFile2.csv	All (58,921 samples) ACTION set to 1	**0.50000**
LaymanFile3.csv	First 50% (29,460 samples) ACTION set to 1 and remaining rows ACTION set to 0.	**0.50492**

The Kaggle Wiki [1] states that the Area under Curve (AUC) is the main evaluation metric for binary classification problems. This is because AUC measures the ability of a binary machine learning (ML) model to predict a higher score for positive examples as compared to negative examples. In addition, one characteristic of the AUC is that it is independent of the fraction of the test population that belongs to either outcome class (which in our case is class 0 or class 1). In other words, this makes the AUC the perfect metric to use for evaluating the performance of classifiers on unbalanced data sets.

What curve does AUC use and how does the AUC Work?

In the previous section we established that there is a matric called AUC that is very effective in evaluating the quality of binary classifiers especially when the training dataset is unbalanced. The curve that is associated with AUC is actually the Receiver Operating Characteristic (ROC) [2] curve that is

[1] The Kaggle Member FAQ (https://www.kaggle.com/wiki/KaggleMemberFAQ) explains the difference between Public and Private Leader board scores in greater detail.

created by plotting the True Positive Rate (TPR) or Sensitivity [2, 3] against the False Positive Rate (FPR) or fall-out [2, 3] at various threshold settings. Each of these terms along with the associated Confusion Matrix is illustrated in Figures 1 and 2 below.

Figure 1: A Confusion Matrix (or Error Matrix) for a Binary Classifier from Wikipedia

		Predicted condition	
	Total population	Predicted Condition positive	Predicted Condition negative
True condition	condition positive	**True positive**	**False Negative** (Type II error)
	condition negative	**False Positive** (Type I error)	**True negative**

Figure 2: Same Confusion Matrix from Figure 2 but with associated mathematical derivations

		Predicted condition			
	Total population	Predicted Condition positive	Predicted Condition negative	Prevalence $= \frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$	
True condition	condition positive	**True positive**	**False Negative** (Type II error)	True positive rate (TPR), Sensitivity, Recall $= \frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	False negative rate (FNR), Miss rate $= \frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$
	condition negative	**False Positive** (Type I error)	**True negative**	False positive rate (FPR), Fall-out $= \frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$	True negative rate (TNR), Specificity (SPC) $= \frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$
	Accuracy (ACC) $= \frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$	Positive predictive value (PPV), Precision $= \frac{\Sigma \text{ True positive}}{\Sigma \text{ Test outcome positive}}$	False omission rate (FOR) $= \frac{\Sigma \text{ False negative}}{\Sigma \text{ Test outcome negative}}$	Positive likelihood ratio (LR+) $= \frac{TPR}{FPR}$	Diagnostic odds ratio (DOR) $= \frac{LR+}{LR-}$
		False discovery rate (FDR) $= \frac{\Sigma \text{ False positive}}{\Sigma \text{ Test outcome positive}}$	Negative predictive value (NPV) $= \frac{\Sigma \text{ True negative}}{\Sigma \text{ Test outcome negative}}$	Negative likelihood ratio (LR−) $= \frac{FNR}{TNR}$	

In the context of Figure 2, let us now define the basic terms in terms of the Amazon Challenge:

- True positive(s) (TP): We predicted yes (ACCESS = 1), and employee actually has access to the RESOURCE.
- True negative(s) (TN): We predicted no (ACCESS = 0), and employee actually has NO access to the RESOURCE.
- False positive(s) (FP): We predicted yes, but employee actually did not have access. (Also known as a "Type I error.")
- False negative(s) (FN): We predicted no, but employee actually did have access. (Also known as a "Type II error.")

Figures B1-B4 in Appendix B illustrates the behavior of the ROC curve. The blue curve in each figure shows the distribution of negatives and the red curve in each figure shows the distribution of positives. This distribution is obtained from the result of a classifier which estimates the probability of a sample (test) point being positive.

So an ROC curve is the most commonly used way to visualize the performance of a binary classifier, and the AUC is (arguably) the best way to summarize its performance in a single number. It took us some time to gain a deep understanding of ROC curves and AUC metric. It also better explained why we were seeing the Private Scores listed in Table 1 for each "laymen" sample submission file.

With the preceding discussion we see that the AUC is probably better described as the Area under the ROC Curve (AUROC). We will interchangeably use AUC and AUROC to mean the same thing. One thing we were able to find out was that there is a built-in roc_auc_score()[2] function in the metrics package of scikit-learn (sklearn). This was a useful insight to have even though it came much later in our investigation as it opened up new avenues of thought in our critical thinking process.

Baby Steps – Simple Logistic Regression

Going back to our initial effort, one of the initial objectives our professor set for us was to run a Simple Logistic Regression on the Test dataset as we had done in our homework assignment and submit the results to Kaggle. We did just that using the code in Appendix C and surprise our AUC score was only 0.52329. This is almost as bad as the "Layman" submission dataset we created and the result signifies that the classifier is doing no better than almost random guessing as reflected by the AUROC score. We experimented further with this classifier by dropping columns we suspected were statistically insignificant but there was no significant change in the AUROC score.

Our results were consistent with what other students were reporting on Pizza. We got a better match than 0.5 as we were asking for a soft classification using predict_proba()[3] which returns probability estimates of being either 0 or 1 rather than the explicit labels 0 or 1 as returned by predict()[4]. But why is Logistic regression performing so poorly? We think we have an idea and we will explore the veracity of that idea in a subsequent section.

Next Steps – Try one hot encoding

On another post in Piazza, our instructor suggested trying a form of feature engineering with Logistic Regression via "one-hot encoding"[5]. The basic idea here is converting categorical features into a vector of zeros and one 1. This has an effect of creating more features and although it is an automated conversion via one-hot encoding, it does not use any intuition or information from the data. One might wonder why we would need to do this on the provided features which were already Int64 codes representing the various categories and the reason to do so is precise the previous statement. We don't want unintended information or intuition from the code (which is otherwise meaning) leaking into the classifier model.

---

[2] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html
[3] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression.predict_proba
[4] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression.predict
[5] http://fastml.com/converting-categorical-data-into-numbers-with-pandas-and-scikit-learn/

Appendix D shows our attempt at one hot encoding.   There was some improvement in the kaggle score but by about 0.14 points only. We are still missing something.

A closer look at the Data

We decided to take a more scientific look at the data in terms of the following areas:

- What is the ratio of true positives to true negatives in our training dataset?
- What are the number of unique values for each feature in the training and testing dataset?
- Are new categorical values introduced in the testing dataset and if yes, what is the percentage increase?
- Were there any categorical values that were outliers?
- What was the statistical spread?
- Are there any obvious correlation (if any) between any of the features in the dataset?

We had too many questions and it was time to get some real answers. In Appendix E we took a closer look at the distribution of the binary outcomes in the ACTION field. From Figure A3 in Appendix A, we had a suspicion we may have an imbalanced data set since everything above the first quartile was a 1. In Figure E1, we get an explicit count. Of the 32,769 samples, 30,872 samples had a true positive value of 1 which amounts to 94.21%.

I think this statistic gave us the first clue why Logistic regression classifier was performing so poorly. We definitely have imbalance![6] In our training dataset, the ACTION = 1 class was present with over a 16:1 ration compared to ACTION = 0 class. In fact, looking back at the simple Logistic Regression results, we now realized that we experienced the accuracy paradox[7] since the regression model was reporting an accuracy of 94.25% (see Ln [7] in Appendix C). In other words it was actually reporting on the extent of the imbalance we had. Little did we know!

The next thing that stood out from the data examination was that MGR_ID has the most unique features (4242) in the training dataset and introduced 16% of unseen values in the testing dataset. The surprise feature was ROLE_FAMILY_DESC (2358 unique values) which introduced 25% more unseen values in the testing dataset. Should these features be dropped outright? We are not sure yet.

The next thing that stood out was that we had the exact same number of unique values for ROLE_TITLE and ROLE_CODE. 343 unique in training and 351 unique in testing. Are these two correlated in any way? We explore that further in Appendix F. Although the relationship is not linear we establish that there is a 1-to-1 mapping in values between the two features. Now the question is can we discard one and if yes which one. Our line of thinking is that the standard deviation is more on ROLE_TITLE and therefore if one feature can be discarded, that is the one we are targeting. It was very interesting to learn that while Kendall and Spearman coefficient establish the strong correlation (strong monotonic trend), it was the low value for the Pearson coefficient that confirmed the relationship is not linear.

The final thing we look at in Appendix G is the distribution and count of each unique value that a feature can have. ROLE_ROLLUP_1 had a very disproportionate number of one value namely 117961. There

[6] When you have imbalanced data in binary classification, it means that the outcome classes are not represented equally

[7] Accuracy Paradox is defined in more detail at https://en.wikipedia.org/wiki/Accuracy_paradox

were 21407 samples with that value in the training data set and 37658 samples of that value for ROLL_ROLLUP_1 in the testing dataset. We are unsure how to address for that anomaly in our models.

Could we make the Logistic Regression classifier do much better?

Armed with the new knowledge that we had a very imbalanced dataset with a 16:1 split between 1's and 0's, we knew that it was providing misleading classification accuracy by only average AUROC scores as reported by Kaggle. The question we had was could this knowledge be useful in somehow applying some bias offset to compensate. We were thinking along the lines of what we learnt in class about regularization. We embarked on some research and what we discovered is that a number of documented techniques do exist although not very well documented. We will summarize them as follows and we tried to explore each viable option to see how it might affect the AUROC score.

These are the options we discovered that might exist:

1. Try to collect more data. Not an option for us as the training dataset was all that was available.
2. Try changing the accuracy metric to use. From Kaggle we knew AUROC was the metric to use
3. Try resampling the data to build a better ratio between the classes. We explored this idea in Appendix L. We took the provided training dataset and created a 1000 samples dataset with sampling by selecting 250 random entries with ACTION = 0 and 750 random samples with ACTION = 1. In other words, we over sampled by picking more 0's. The pandas dataframe makes this task very trivial. We then trained on this new
4. Try generating synthetic samples. We read about "SMOTE: Synthetic Minority Over-sampling Technique"[8]. We even looked at the UnBalancedDataset[9] module.
5. Try different algorithms

Back to the Real world, are there classifiers that can do better?

Talk about Random Forest Classifier here with no hyper-parameter tuning

What strategies did the competition winners employ?

We did not want to look at any of the top winners solution or it would blind us and make us ignore promising possibilities. Running theme about Miroslav's starter code

Future Exploration

[8] SMOTE is described in more detail at http://www.jair.org/media/953/live-953-2037-jair.pdf
[9] Python UnBalancedDataset module is at https://github.com/fmfn/UnbalancedDataset

Conclusion

What a tumultuous adventure of data exploration that last three and half weeks have been. Personally speaking we had mixed feelings about proceeding with the project and taking the no final route as we initially struggled with obtaining good AUC scores from Kaggle. But the thought of a final after the level setting midterm probably scared the entire class even more.

However, at this point in time as we are about to turn in the report, we feel like the project was the best path forward. It allowed us to work on real data, take the baby gloves off, and really do some serious research when things didn't go as expected. Both of us learnt a lot in the last three week more so than we thought was ever possible in such a short timeframe. We think we are better (junior) data scientists for it and we feel empowered that we can take on any dataset alongside the very best data scientists in the world.

This project made us get over the hump so to speak. Look at what we accomplished! We cracked the 0.9 AUC barrier we had set ourselves, we think that given enough time we could make the logistic regression classifier work much better than it has to date on an imbalanced binary classification problem like the Amazon Employee Access Challenge.

Perhaps, down the line, this could lead to a Master's Report for either one of us. Maybe we could try to build that elusive logistf() function (that is available in R) but is missing from all the available Python data libraries.

APPENDIX

APPENDIX A

Table A1: Amazon Employee Access Dataset Feature Description.

Feature Name	Feature Meaning
ACTION	"1": Approved or Access Granted to Resource; "0": Rejected or Access Denied to Resource
RESOURCE	Resource ID
MGR ID	Employee ID of the Employee's manager (can only have one manager)
ROLE ROLLUP 1	Company Role Grouping Category ID1 (e.g. US Engineering)
ROLE ROLLUP 2	Company Role Grouping Category ID2 (e.g. US Retail)
ROLE DEPTNAME	Company Role Department Description (e.g. Retail)
ROLE TITLE	Business Title Description (e.g. Senior Engineering Retail Manager)
ROLE FAMILY DESC	Role family extended description (e.g. Retail manager, Software Engineering)
ROLE FAMILY	Role family description (e.g. Retail Manager)
ROLE CODE	Unique ID for each company role (e.g. Manager)

Note: Each categorical feature is expressed as numeric code

Figure A2: Number of entries in the training dataset (train.csv)

```
In [3]: # Gather
        print train.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32769 entries, 0 to 32768
Data columns (total 10 columns):
ACTION              32769 non-null int64
RESOURCE            32769 non-null int64
MGR_ID              32769 non-null int64
ROLE_ROLLUP_1       32769 non-null int64
ROLE_ROLLUP_2       32769 non-null int64
ROLE_DEPTNAME       32769 non-null int64
ROLE_TITLE          32769 non-null int64
ROLE_FAMILY_DESC    32769 non-null int64
ROLE_FAMILY         32769 non-null int64
ROLE_CODE           32769 non-null int64
dtypes: int64(10)
memory usage: 2.5 MB
None
```

Figure A3: Summary Statistics of the Features with strong indications of an imbalanced dataset

```
In [4]: train.describe().transpose()
```

Out[4]:

	count	mean	std	min	25%	50%	75%	max
ACTION	32769.0	0.942110	0.233539	0.0	1.0	1.0	1.0	1.0
RESOURCE	32769.0	42923.916171	34173.892702	0.0	20299.0	35376.0	74189.0	312153.0
MGR_ID	32769.0	25988.957979	35928.031650	25.0	4566.0	13545.0	42034.0	311696.0
ROLE_ROLLUP_1	32769.0	116952.627788	10875.563591	4292.0	117961.0	117961.0	117961.0	311178.0
ROLE_ROLLUP_2	32769.0	118301.823156	4551.588572	23779.0	118102.0	118300.0	118386.0	286791.0
ROLE_DEPTNAME	32769.0	118912.779914	18961.322917	4674.0	118395.0	118921.0	120535.0	286792.0
ROLE_TITLE	32769.0	125916.152644	31036.465825	117879.0	118274.0	118568.0	120006.0	311867.0
ROLE_FAMILY_DESC	32769.0	170178.369648	69509.462130	4673.0	117906.0	128696.0	235280.0	311867.0
ROLE_FAMILY	32769.0	183703.408893	100488.407413	3130.0	118363.0	119006.0	290919.0	308574.0
ROLE_CODE	32769.0	119789.430132	5784.275516	117880.0	118232.0	118570.0	119348.0	270691.0

Figure A4: Number of entries in the training dataset (train.csv)

```
In [5]: print test.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 58921 entries, 0 to 58920
Data columns (total 10 columns):
id                  58921 non-null int64
RESOURCE            58921 non-null int64
MGR_ID              58921 non-null int64
ROLE_ROLLUP_1       58921 non-null int64
ROLE_ROLLUP_2       58921 non-null int64
ROLE_DEPTNAME       58921 non-null int64
ROLE_TITLE          58921 non-null int64
ROLE_FAMILY_DESC    58921 non-null int64
ROLE_FAMILY         58921 non-null int64
ROLE_CODE           58921 non-null int64
dtypes: int64(10)
memory usage: 4.5 MB
None
```

Figure A5: Summary Statistics of the Features of the testing dataset

```
In [6]: test.describe().transpose()
```

Out[6]:

	count	mean	std	min	25%	50%	75%	max
id	58921.0	29461.000000	17009.171942	1.0	14731.0	29461.0	44191.0	58921.0
RESOURCE	58921.0	39383.739482	33717.397122	0.0	18418.0	33248.0	45481.0	312136.0
MGR_ID	58921.0	26691.645050	35110.244281	25.0	4663.0	14789.0	46512.0	311779.0
ROLE_ROLLUP_1	58921.0	117028.638041	10805.446548	4292.0	117961.0	117961.0	117961.0	311178.0
ROLE_ROLLUP_2	58921.0	118316.334091	4284.678750	23779.0	118096.0	118300.0	118386.0	194897.0
ROLE_DEPTNAME	58921.0	118858.006721	17916.179109	4674.0	118378.0	118910.0	120410.0	277693.0
ROLE_TITLE	58921.0	126358.019993	32068.294507	117879.0	118259.0	118636.0	120006.0	311867.0
ROLE_FAMILY_DESC	58921.0	170455.861425	69684.692799	4673.0	117913.0	129282.0	234813.0	311867.0
ROLE_FAMILY	58921.0	179278.058960	99639.965300	3130.0	118331.0	118704.0	290919.0	308574.0
ROLE_CODE	58921.0	119707.754264	5326.979178	117880.0	118055.0	118570.0	119353.0	270691.0

APPENDIX B

This figures below illustrate the behavior of the ROC curve.

The key point to note is the area under curve (AUC) is the highest when the two curves are farthest apart with little overlap and our ML model (classifier) is most optimized.

Figure B1: Best AUC Score

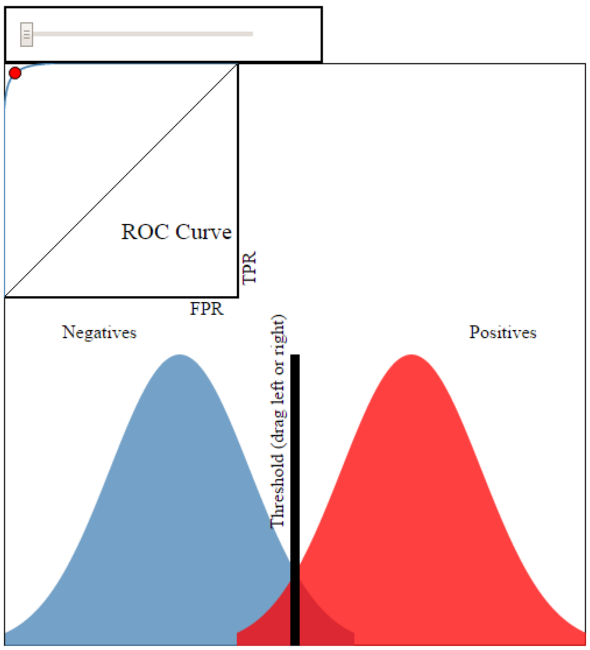


Figure B2: Better AUC Score

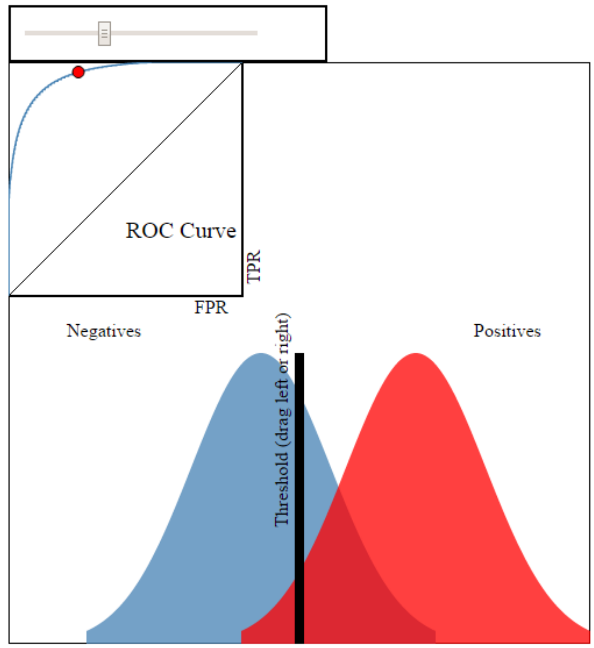


Figure B3: Moderate AUC Score

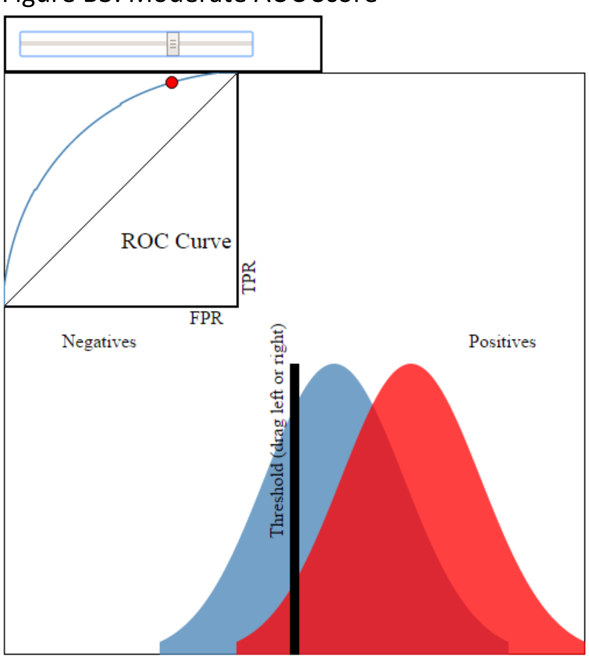
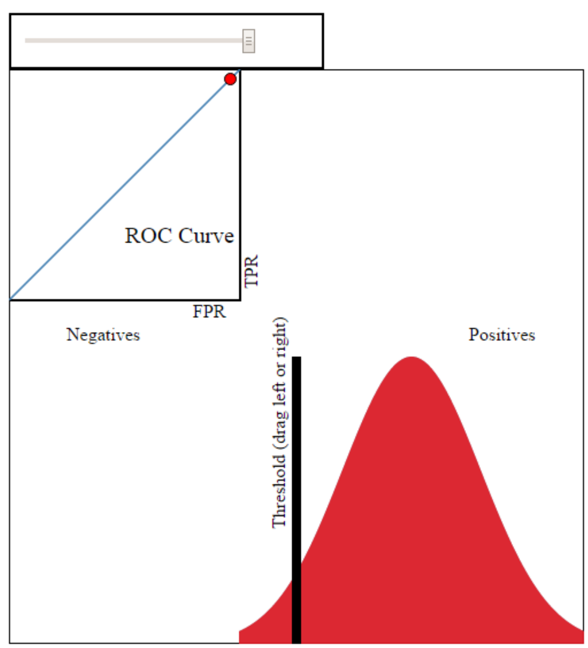


Figure B4: Lowest AUC Score



APPENDIX C

Code C1: Simple Logistic Regression and resulting Kaggle Score

```
In [1]: import pandas as pd
        from sklearn.cross_validation import KFold
        from sklearn import svm
        import numpy as np
        from sklearn.linear_model import LogisticRegression

In [2]: # load training dataset - run
        training=pd.read_csv("train.csv")

In [3]: # get X values except "ACTION" column -run
        df_x = training.drop(['ACTION'], axis=1)
        X= df_x.values
        print X.shape

        (32769L, 9L)

In [4]: # Extract "ACTION" columns as Y axis -run
        Y = training.as_matrix(["ACTION"])
        print Y.shape

        (32769L, 1L)

In [5]: # Using Kfold divide training dataset into two training and testing data
        kf = KFold(len(X), n_folds=2)
        print kf

        for train_index, test_index in kf:
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = Y[train_index], Y[test_index]
        print len(X_train),len(X_test)

        sklearn.cross_validation.KFold(n=32769, n_folds=2, shuffle=False, random_state=None)
        16385 16384

In [7]: # Use logistic regression to get training and testing score and get predicted value for training
        model2 = LogisticRegression()
        model2.fit(X_train,np.ravel(y_train))
        print model2.score(X_test,np.ravel(y_test))
        print "Coefficeint:", model2.coef_
        predictVal= model2.predict(X_train)

        0.942565917969
        Coefficeint: [[ -7.03918265e-08  -6.36847124e-07  -4.17189982e-06   1.01556683e-06
            2.25946795e-06  -1.45528212e-06   7.14101686e-07   2.49097386e-07
            2.44184289e-05]]

In [8]: #Load testing dataset -run
        testing=pd.read_csv("test.csv", index_col='id')
        #print testing

In [9]: # Use logistic regression to get best fit model for testing dataset and get the predicted value
        Y=np.ravel(Y)
        model = LogisticRegression()
        model.fit(X,Y)
        test_predictVal = pd.DataFrame(columns=['ACTION'], index=testing.index, data=model.predict_proba
        test_predictVal.to_csv("submission-simple-LR.csv")
```

Amazon.com - Employee Access Challenge, obtaining 0.52329 | 12 days ago

APPENDIX D

Our attempt at testing one hot encoding of the features and then applying the Linear Regression classifier.

```
In [1]: import pandas as pd
        import numpy as np
        from sklearn.linear_model import LogisticRegression
        from sklearn.ensemble import (RandomTreesEmbedding, RandomForestClassifier,
                                      GradientBoostingClassifier)
        from sklearn.preprocessing import OneHotEncoder
```

```
In [2]: # load training dataset - run
        training=pd.read_csv("train.csv")
```

```
In [3]: # get X values except "ACTION" column -run
        df_x = training.drop(['ACTION'], axis=1)
        X= df_x.values
        # Extract "ACTION" columns as Y axis -run
        Y = training.as_matrix(["ACTION"])
```

```
In [4]: #Load testing dataset -run
        testing=pd.read_csv("test.csv", index_col='id')
```

```
In [5]: #Trial 1
        Y=np.ravel(Y)
        rf = RandomForestClassifier(max_depth=3, n_estimators=10)
        rf_enc =OneHotEncoder()
        rf_lm = LogisticRegression()
        rf.fit(X, Y)
        rf_enc.fit(rf.apply(X))
        rf_lm.fit(rf_enc.transform(rf.apply(X)), Y)

        encode = pd.DataFrame(columns=['ACTION'], index=testing.index,
                              data=rf_lm.predict_proba(rf_enc.transform(rf.apply(testing)))[:, 1])
        encode.to_csv("simplesubmission_OE-1.csv")
```

Score marginally improves to 0.66941 from 0.52329

Amazon.com - Employee Access Challenge, obtaining 0.66941

APPENDIX E

Figure E1: Confirmation of a very imbalanced training dataset

```
In [34]: train.groupby(['ACTION'])['ACTION'].count()

Out[34]: ACTION
         0      1897
         1     30872
         Name: ACTION, dtype: int64

In [36]: train.groupby(['ACTION'])['ACTION'].hist(bins=1)

Out[36]: ACTION
         0      Axes(0.125,0.125;0.775x0.775)
         1      Axes(0.125,0.125;0.775x0.775)
         Name: ACTION, dtype: object
```

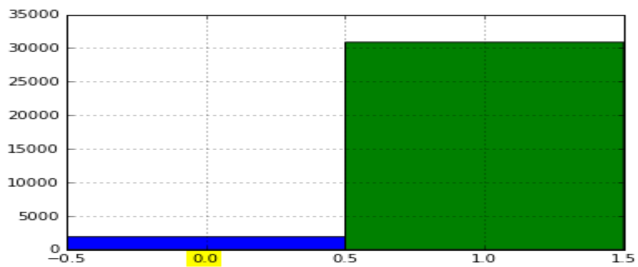


Figure E2: Confirmation of a very imbalanced training dataset

```
In [10]: # Compare values in single COLUMN for introduction of new data in test dataset
         Train_Resource_Index = train.groupby(['RESOURCE'])['RESOURCE'].count().keys()
         Test_Resource_Index = test.groupby(['RESOURCE'])['RESOURCE'].count().keys()
         print Test_Resource_Index.difference(Train_Resource_Index).size
         print Train_Resource_Index.size
         print Test_Resource_Index.size

         0
         7518
         4971
```

Table E3: Summary of new features in testing dataset that is not present in training dataset

Feature Name	Num of Unique Values in Training dataset	Num of Unique Values in Testing dataset	Num of new Values in Testing dataset	% of new values in testing dataset
RESOURCE	7518	4971	0	0%
MGR ID	**4242**	**4689**	**670**	**16%**
ROLE ROLLUP 1	128	126	2	2%
ROLE ROLLUP 2	177	177	6	3%
ROLE DEPTNAME	449	466	27	6%
ROLE TITLE	343	351	18	5%
ROLE FAMILY DESC	**2358**	**2749**	**593**	**25%**
ROLE FAMILY	67	68	1	1%
ROLE CODE	343	351	18	5%

In [10]:
```
# Compare values in single COLUMN for introduction of new data in test dataset
Train_Resource_Index = train.groupby(['RESOURCE'])['RESOURCE'].count().keys()
Test_Resource_Index = test.groupby(['RESOURCE'])['RESOURCE'].count().keys()
print Test_Resource_Index.difference(Train_Resource_Index).size
print "{0:.0f}%".format((Test_Resource_Index.difference(Train_Resource_Index).size / Train_Resou
print Train_Resource_Index.size
print Test_Resource_Index.size
```

```
0
0%
7518
4971
```

In [11]:
```
# Compare values in single COLUMN for introduction of new data in test dataset
Train_MGR_ID_Index = train.groupby(['MGR_ID'])['MGR_ID'].count().keys()
Test_MGR_ID_Index = test.groupby(['MGR_ID'])['MGR_ID'].count().keys()
print Test_MGR_ID_Index.difference(Train_MGR_ID_Index).size
print "{0:.0f}%".format((Test_MGR_ID_Index.difference(Train_MGR_ID_Index).size / Train_MGR_ID_In
print Train_MGR_ID_Index.size
print Test_MGR_ID_Index.size
```

```
670
16%
4243
4689
```

In [12]:
```
# Compare values in single COLUMN for introduction of new data in test dataset
Train_ROLE_ROLLUP_1_Index = train.groupby(['ROLE_ROLLUP_1'])['ROLE_ROLLUP_1'].count().keys()
Test_ROLE_ROLLUP_1_Index = test.groupby(['ROLE_ROLLUP_1'])['ROLE_ROLLUP_1'].count().keys()
print Test_ROLE_ROLLUP_1_Index.difference(Train_ROLE_ROLLUP_1_Index).size
print "{0:.0f}%".format((Test_ROLE_ROLLUP_1_Index.difference(Train_ROLE_ROLLUP_1_Index).size / T
print Train_ROLE_ROLLUP_1_Index.size
print Test_ROLE_ROLLUP_1_Index.size
```

```
2
2%
128
126
```

In [13]:
```
# Compare values in single COLUMN for introduction of new data in test dataset
Train_ROLE_ROLLUP_2_Index = train.groupby(['ROLE_ROLLUP_2'])['ROLE_ROLLUP_2'].count().keys()
Test_ROLE_ROLLUP_2_Index = test.groupby(['ROLE_ROLLUP_2'])['ROLE_ROLLUP_2'].count().keys()
print Test_ROLE_ROLLUP_2_Index.difference(Train_ROLE_ROLLUP_2_Index).size
print "{0:.0f}%".format((Test_ROLE_ROLLUP_2_Index.difference(Train_ROLE_ROLLUP_2_Index).size / T
print Train_ROLE_ROLLUP_2_Index.size
print Test_ROLE_ROLLUP_2_Index.size
```

```
6
3%
177
177
```

In [14]:
```
# Compare values in single COLUMN for introduction of new data in test dataset
Train_ROLE_DEPTNAME_Index = train.groupby(['ROLE_DEPTNAME'])['ROLE_DEPTNAME'].count().keys()
Test_ROLE_DEPTNAME_Index = test.groupby(['ROLE_DEPTNAME'])['ROLE_DEPTNAME'].count().keys()
print Test_ROLE_DEPTNAME_Index.difference(Train_ROLE_DEPTNAME_Index).size
print "{0:.0f}%".format((Test_ROLE_DEPTNAME_Index.difference(Train_ROLE_DEPTNAME_Index).size / T
print Train_ROLE_DEPTNAME_Index.size
print Test_ROLE_DEPTNAME_Index.size
```

```
27
6%
449
466
```

```
In [15]: # Compare values in single COLUMN for introduction of new data in test dataset
Train_ROLE_TITLE_Index = train.groupby(['ROLE_TITLE'])['ROLE_TITLE'].count().keys()
Test_ROLE_TITLE_Index = test.groupby(['ROLE_TITLE'])['ROLE_TITLE'].count().keys()
print Test_ROLE_TITLE_Index.difference(Train_ROLE_TITLE_Index).size
print "{0:.0f}%".format((Test_ROLE_TITLE_Index.difference(Train_ROLE_TITLE_Index).size / Train_R
print Train_ROLE_TITLE_Index.size
print Test_ROLE_TITLE_Index.size
```

```
18
5%
343
351
```

```
In [16]: # Compare values in single COLUMN for introduction of new data in test dataset
Train_ROLE_FAMILY_DESC_Index = train.groupby(['ROLE_FAMILY_DESC'])['ROLE_FAMILY_DESC'].count().k
Test_ROLE_FAMILY_DESC_Index = test.groupby(['ROLE_FAMILY_DESC'])['ROLE_FAMILY_DESC'].count().key
print Test_ROLE_FAMILY_DESC_Index.difference(Train_ROLE_FAMILY_DESC_Index).size
print "{0:.0f}%".format((Test_ROLE_FAMILY_DESC_Index.difference(Train_ROLE_FAMILY_DESC_Index).si
print Train_ROLE_FAMILY_DESC_Index.size
print Test_ROLE_FAMILY_DESC_Index.size
```

```
593
25%
2358
2749
```

```
In [17]: # Compare values in single COLUMN for introduction of new data in test dataset
Train_ROLE_FAMILY_Index = train.groupby(['ROLE_FAMILY'])['ROLE_FAMILY'].count().keys()
Test_ROLE_FAMILY_Index = test.groupby(['ROLE_FAMILY'])['ROLE_FAMILY'].count().keys()
print Test_ROLE_FAMILY_Index.difference(Train_ROLE_FAMILY_Index).size
print "{0:.0f}%".format((Test_ROLE_FAMILY_Index.difference(Train_ROLE_FAMILY_Index).size / Train_
print Train_ROLE_FAMILY_Index.size
print Test_ROLE_FAMILY_Index.size
```

```
1
1%
67
68
```

```
In [18]: # Compare values in single COLUMN for introduction of new data in test dataset
Train_ROLE_CODE_Index = train.groupby(['ROLE_CODE'])['ROLE_CODE'].count().keys()
Test_ROLE_CODE_Index = test.groupby(['ROLE_CODE'])['ROLE_CODE'].count().keys()
print Test_ROLE_CODE_Index.difference(Train_ROLE_CODE_Index).size
print "{0:.0f}%".format((Test_ROLE_CODE_Index.difference(Train_ROLE_CODE_Index).size / Train_ROL
print Train_ROLE_CODE_Index.size
print Test_ROLE_CODE_Index.size
```

```
18
5%
343
351
```

APPENDIX F

Figure F1: Exploring relationship between ROLE_CODE and ROLE_TITLE

```
In [54]: # Is there a correlation between ROLE_CODE and ROLE_TITLE
         RC_RT = train[['ROLE_CODE','ROLE_TITLE']]
         print RC_RT.corr(method='kendall')
         print RC_RT.corr(method='spearman')
         print RC_RT.corr(method='pearson')

                     ROLE_CODE  ROLE_TITLE
         ROLE_CODE    1.000000    0.905023
         ROLE_TITLE   0.905023    1.000000
                     ROLE_CODE  ROLE_TITLE
         ROLE_CODE    1.000000    0.916368
         ROLE_TITLE   0.916368    1.000000
                     ROLE_CODE  ROLE_TITLE
         ROLE_CODE     1.00000     0.15592
         ROLE_TITLE    0.15592     1.00000
```

Figure F2: Exploring ROLE_CODE and ROLE_TITLE values map 1:1

```
In [85]: # Both Kendall and Spearman coefficient indicate a very high monotonic
         # trend between ROLE_CODE and ROLE_TITLE
         # But a low Pearson coefficient indicates relationship is not LINEAR
         # But in the training and test sets, we have an identical number of values
         # for both features (343 and 351)
         # Could there be a 1:1 mapping of values?  Let us do a groupby and 343 will # confirm
         print train.groupby(['ROLE_CODE','ROLE_TITLE'])['ROLE_TITLE'].nunique().size
         print train.groupby(['ROLE_TITLE','ROLE_CODE'])['ROLE_CODE'].nunique().size
         RT_RC = train.groupby(['ROLE_TITLE','ROLE_CODE'])['ROLE_CODE'].count()
         RT_RC.sort_values(ascending=False).head(10)

         343
         343

Out[85]: ROLE_TITLE  ROLE_CODE
         118321      118322       4649
         117905      117908       3583
         118784      118786       1772
         117879      117880       1256
         118568      118570       1043
         117885      117888        806
         118054      118055        774
         118685      118687        597
         118777      118779        566
         118451      118454        521
         Name: ROLE_CODE, dtype: int64
```

Figure F3: Exploring ROLE_CODE and ROLE_TITLE values map 1:1 in Test dataset as well

```
In [86]: # Does 1:1 mapping hold true for Test DataSet?  IT DOES!
         print test.groupby(['ROLE_CODE','ROLE_TITLE'])['ROLE_TITLE'].nunique().size
         print test.groupby(['ROLE_TITLE','ROLE_CODE'])['ROLE_CODE'].nunique().size

         351
         351
```

APPENDIX G

Frequency distribution of each Categorical feature in the Amazon Training Dataset
{include from Python}

```
In [2]:  # Read In dataset
         train = pd.read_csv('train.csv')
```

```
In [6]:  train.groupby(['RESOURCE'])['RESOURCE'].count().sort_values(ascending=False).head(20)

Out[6]:  RESOURCE
         4675      839
         79092     484
         25993     409
         75078     409
         3853      404
         6977      299
         75834     299
         32270     295
         42085     247
         17308     239
         1020      236
         13878     220
         42093     204
         18418     192
         7543      186
         23921     167
         278393    163
         34924     161
         79121     157
         28149     137
         Name: RESOURCE, dtype: int64
```

```
In [7]:  train.groupby(['MGR_ID'])['MGR_ID'].count().sort_values(ascending=False).head(20)

Out[7]:  MGR_ID
         770      152
         2270      99
         2594      82
         1350      71
         2014      67
         16850     66
         7807      64
         3966      64
         3526      62
         5244      62
         5396      62
         7411      61
         4659      61
         54618     61
         18686     60
         7389      58
         7578      58
         3281      57
         70062     57
         18213     57
         Name: MGR_ID, dtype: int64
```

```
In [8]:  train.groupby(['ROLE_DEPTNAME'])['ROLE_DEPTNAME'].count().sort_values(ascending=False).head(20)

Out[8]:  ROLE_DEPTNAME
         117878    1135
         117941     763
         117945     659
         118514     601
         117920     597
         117884     546
         119598     543
         118403     532
         119181     525
         120722     501
         118320     435
         117895     431
         118746     415
         118783     366
         120663     335
         118910     325
         118437     317
         118352     305
         118631     304
         120551     304
         Name: ROLE_DEPTNAME, dtype: int64
```

In [9]:
```
train.groupby(['ROLE_TITLE'])['ROLE_TITLE'].count().sort_values(ascending=False).head(20)
```

Out[9]:
```
ROLE_TITLE
118321    4649
117905    3583
118784    1772
117879    1256
118568    1043
117885     806
118054     774
118685     597
118777     566
118451     521
120344     473
307024     467
280788     394
179731     384
118422     376
118890     347
118636     344
118396     342
119849     337
118834     335
Name: ROLE_TITLE, dtype: int64
```

In [10]:
```
train.groupby(['ROLE_FAMILY_DESC'])['ROLE_FAMILY_DESC'].count().sort_values(ascending=False).head(20)
```

Out[10]:
```
ROLE_FAMILY_DESC
117906    6896
240983    1244
117913     670
279443     665
117886     530
130134     419
117897     351
117879     333
168365     324
133686     321
118054     311
118448     282
118959     246
280788     244
118785     233
302830     225
300136     222
311622     219
269406     211
306399     205
Name: ROLE_FAMILY_DESC, dtype: int64
```

In [11]:
```
train.groupby(['ROLE_FAMILY'])['ROLE_FAMILY'].count().sort_values(ascending=False).head(20)
```

Out[11]:
```
ROLE_FAMILY
290919    10980
118424     2690
19721      2636
117887     2400
292795     1318
118398     1294
308574     1287
118453      941
118331      892
118638      783
118643      783
270488      689
118295      493
118960      465
118205      449
119095      412
4673        384
19793       362
120518      294
119184      293
Name: ROLE_FAMILY, dtype: int64
```

In [12]:
```python
train.groupby(['ROLE_CODE'])['ROLE_CODE'].count().sort_values(ascending=False).head(20)
```

Out[12]:
```
ROLE_CODE
118322    4649
117908    3583
118786    1772
117880    1256
118570    1043
117888     806
118055     774
118687     597
118779     566
118454     521
120346     473
118332     467
119082     394
117973     384
118425     376
118892     347
118639     344
118399     342
119851     337
118836     335
Name: ROLE_CODE, dtype: int64
```

In [13]:
```python
train.groupby(['ROLE_ROLLUP_1'])['ROLE_ROLLUP_1'].count().sort_values(ascending=False).head(20)
```

Out[13]:
```
ROLE_ROLLUP_1
117961    21407
117902      742
91261       721
118315      498
118212      400
118290      398
119062      375
118887      334
117916      295
118169      291
118752      282
117929      276
118256      275
117926      269
119596      239
117890      234
118079      221
118006      197
118573      190
5110        186
Name: ROLE_ROLLUP_1, dtype: int64
```

In [14]:
```python
train.groupby(['ROLE_ROLLUP_2'])['ROLE_ROLLUP_2'].count().sort_values(ascending=False).head(20)
```

Out[14]:
```
ROLE_ROLLUP_2
118300    4424
118343    3945
118327    2641
118225    2547
118386    1796
118052    1665
117962    1567
118413    1295
118446     971
118026     721
117903     489
117969     397
118291     396
118888     334
119091     321
118213     295
118170     291
118463     267
118257     257
118041     253
Name: ROLE_ROLLUP_2, dtype: int64
```

APPENDIX H

Installing XGBoost on Windows turned out to be a Herculean task. No clear step-by-step directions were available online. Our professor was gracious enough to share the following post in Piazza[10] from input of a student in another class on how to get XGBoost installed on Windows.

XGBoost on Windows

Based on a bunch of failure and this post by the XGBoost author: https://github.com/dmlc/xgboos
Windows. Every one of the how-to articles points to a windows code branch on github that no lon

the instructors' answer, *where instructors collectively construct a single answer*

Hi All,

As of a few weeks ago (prior to cmake), it was possible to install on Windows, though as I mentioned, somewhat difficult. Here are instructions that another student (from another class) provided to his classmates. Does this still work for anyone?

-constantine

I used instructions from two separate guides and managed to build xgboost on Windows 10 for a 64-bit machine:

http://dnc1994.com/2016/03/installing-xgboost-on-windows/
http://stackoverflow.com/questions/33749735/how-to-install-xgboost-package-in-python-windows-platform

Here are my steps for installing xgboost:
1. First, before you do anything else, make sure you install Git Bash for Windows:
https://git-scm.com/download/win

2. Then, install mingw-w64. Select the appropriate architecture for your computer: i686 for 32-bit machines, and x86_64 for 64-bit.
https://sourceforge.net/projects/mingw-w64/files/latest/download

But unfortunately those steps did not work for us either. We checked with a number of students in class and no one had successfully installed XGBoost on Windows.

We kept hacking at it and finally had a break through.

Step 8 of the posted instructions were as follows:

8. Run the remaining commands: Make sure you don't run make with the '-j4' flag -- this may lead to compilation failures due to missing dependencies.

```
cp make/mingw64.mk config.mk
make
```

The resolution was to replace "make" with "mingw32-make" when on an x64 bit machine. It was frustrating to have spent almost two days trying to get this package installed.

Additional tip is to disable real-time AV scanning prior to install.

[10] https://utexas.instructure.com/courses/1158730/external_tools/23798

APPENDIX I

```
In [1]: # ge_mg_RandomForest
        # Use RandomForestClassifier rather than Regressor.
        # Use regressor when outcome can have unseen values
        # Get very high AUC score
        from sklearn.ensemble import RandomForestClassifier
        import pandas as pd

        # Read In dataset
        train = pd.read_csv('train.csv')
        test = pd.read_csv('test.csv', index_col='id') # Test data has an id column, train does not.

        # Pull out dependent variable
        y = train.pop("ACTION")
        X = train
```

```
In [2]: # Define the model random forest's with few parametrs
        model = RandomForestClassifier(n_estimators=100, oob_score=True, random_state = 42)
        %time model.fit(X, y)

        # Predict the fitted model on the test data and output predictions to a csv
        submission = pd.DataFrame(columns=['ACTION'], index=test.index, data=model.predict_proba(test)[:

        #Create submission file
        submission.to_csv("submission-FC5.csv")

        Wall time: 4.99 s
```

Submitted an entry to Amazon.com - Employee Access Challenge, obtaining 0.85220

Change to n_estimators = 400. Takes roughly 4 times longer

```
In [4]: # Define the model random forest's with few parametrs
        model = RandomForestClassifier(n_estimators=400, oob_score=True, random_state = 31)
        %time model.fit(X, y)

        # Predict the fitted model on the test data and output predictions to a csv
        submission = pd.DataFrame(columns=['ACTION'], index=test.index, data=model.predict_proba(test)[:

        #Create submission file
        submission.to_csv("submission-FC5.csv")

        Wall time: 20.2 s
```

Score improvement of 0.012 (very promising no effort model)

Submitted an entry to Amazon.com - Employee Access Challenge, obtaining 0.86411

APPENDIX J

```
In [1]: # Adapted from Forum posted code (not WINNER's Code)
        from numpy import array, hstack
        from sklearn import metrics, cross_validation, linear_model
        from scipy import sparse
        from itertools import combinations
        import numpy as np
        import pandas as pd

        SEED = 31
```

```
In [2]: def group_data(data, degree=3, hash=hash):
            new_data = []
            m,n = data.shape
            for indicies in combinations(range(n), degree):
                new_data.append([hash(tuple(v)) for v in data[:,indicies]])
            return array(new_data).T
```

```
In [3]: def OneHotEncoder(data, keymap=None):

            if keymap is None:
                keymap = []
                for col in data.T:
                    uniques = set(list(col))
                    keymap.append(dict((key, i) for i, key in enumerate(uniques)))
            total_pts = data.shape[0]
            outdat = []
            for i, col in enumerate(data.T):
                km = keymap[i]
                num_labels = len(km)
                spmat = sparse.lil_matrix((total_pts, num_labels))
                for j, val in enumerate(col):
                    if val in km:
                        spmat[j, km[val]] = 1
                outdat.append(spmat)
            outdat = sparse.hstack(outdat).tocsr()
            return outdat, keymap
```

```
In [4]: # AUC comparison loop
        def cv_loop(X, y, model, N):
            mean_auc = 0.
            for i in range(N):
                X_train, X_cv, y_train, y_cv = cross_validation.train_test_split(
                                               X, y, test_size=.20,
                                               random_state = i*SEED)
                model.fit(X_train, y_train)
                preds = model.predict_proba(X_cv)[:,1]
                auc = metrics.roc_auc_score(y_cv, preds)
                print "AUC (fold %d/%d): %f" % (i + 1, N, auc)
                mean_auc += auc
            return mean_auc/N
```

```
In [6]: # Read In dataset
        train = pd.read_csv('train.csv')
        test = pd.read_csv('test.csv')
        all_data = np.vstack((train.ix[:,1:-1], test.ix[:,1:-1]))

        num_train = np.shape(train)[0]
        print num_train

        %time group_by_two = group_data(all_data, degree=2)
        %time group_by_three = group_data(all_data, degree=3)

        y = array(train.ACTION)

        32769
        Wall time: 4.48 s
        Wall time: 9.94 s
```

```
In [8]: %time X_test = all_data[num_train:]
        %time X_test_2 = group_by_two[num_train:]
        %time X_test_3 = group_by_three[num_train:]

        Wall time: 0 ns
        Wall time: 0 ns
        Wall time: 0 ns

In [9]: %time X_train_all = np.hstack((X, X_2, X_3))
        %time X_test_all = np.hstack((X_test, X_test_2, X_test_3))
        num_features = X_train_all.shape[1]
        print num_features

        Wall time: 12 ms
        Wall time: 19 ms
        92
```

```
In [10]: model = linear_model.LogisticRegression()
         %time Xts = [OneHotEncoder(X_train_all[:,[i]])[0] for i in range(num_features)]

         Wall time: 42.1 s
```

```
In [11]: print "Greedy feature selection (Two Folds)"
         score_hist = []
         N = 2
         good_features = set([])
         # Greedy feature selection loop
         while len(score_hist) < 2 or score_hist[-1][0] > score_hist[-2][0]:
             scores = []
             for f in range(len(Xts)):
                 if f not in good_features:
                     feats = list(good_features) + [f]
                     Xt = sparse.hstack([Xts[j] for j in feats]).tocsr()
                     score = cv_loop(Xt, y, model, N)
                     scores.append((score, f))
                     print "Feature: %i Mean AUC: %f" % (f, score)
             good_features.add(sorted(scores)[-1][1])
             score_hist.append(sorted(scores)[-1])
             print "Current features: %s" % sorted(list(good_features))
```

Start

```
Performing greedy feature selection...
AUC (fold 1/2): 0.659692
AUC (fold 2/2): 0.638387
```

And Finish

```
Feature: 87 Mean AUC: 0.896951
AUC (fold 1/2): 0.892836
AUC (fold 2/2): 0.901122
Feature: 88 Mean AUC: 0.896979
AUC (fold 1/2): 0.892282
AUC (fold 2/2): 0.899977
Feature: 89 Mean AUC: 0.896129
AUC (fold 1/2): 0.893138
AUC (fold 2/2): 0.900177
Feature: 90 Mean AUC: 0.896657
AUC (fold 1/2): 0.895245
AUC (fold 2/2): 0.900612
Feature: 91 Mean AUC: 0.897929
Current features: [0, 25, 37, 42, 47, 64, 68, 69, 79, 82]
```

This would keep running for days but we stopped after 15 minutes of number crunching. If we had time, we would have liked to keep this running until it stooped on its own.

```
In [12]: # Remove last added feature from good_features
         good_features.remove(score_hist[-1][1])
         good_features = sorted(list(good_features))
         print "Selected features %s" % good_features

         Selected features [0, 25, 42, 47, 64, 68, 69, 79, 82]
```

The order of the selected features after we aborted the run

```
In [13]: print "Performing hyperparameter selection..."
         # Hyperparameter selection loop
         score_hist = []
         Xt = sparse.hstack([Xts[j] for j in good_features]).tocsr()
         Cvals = np.logspace(-4, 4, 15, base=2)
         for C in Cvals:
             model.C = C
             score = cv_loop(Xt, y, model, N)
             score_hist.append((score,C))
             print "C: %f Mean AUC: %f" %(C, score)
         bestC = sorted(score_hist)[-1][1]
         print "Best C value: %f" % (bestC)

         Performing hyperparameter selection...
         AUC (fold 1/2): 0.857650
         AUC (fold 2/2): 0.854405
         C: 0.062500 Mean AUC: 0.856027
```

- .
- .
- .

```
         C: 4.876055 Mean AUC: 0.894907
         AUC (fold 1/2): 0.887926
         AUC (fold 2/2): 0.899145
         C: 7.245789 Mean AUC: 0.893535
         AUC (fold 1/2): 0.886211
         AUC (fold 2/2): 0.898240
         C: 10.767202 Mean AUC: 0.892226
         AUC (fold 1/2): 0.884750
         AUC (fold 2/2): 0.897133
         C: 16.000000 Mean AUC: 0.890941
         Best C value: 1.000000
```

Best C value was 1.0

```
In [14]: print "Performing One Hot Encoding on entire dataset..."
         Xt = np.vstack((X_train_all[:,good_features], X_test_all[:,good_features]))
         Xt, keymap = OneHotEncoder(Xt)
         X_train = Xt[:num_train]
         X_test = Xt[num_train:]

         print "Training full model..."
         %time model.fit(X_train, y)

         Performing One Hot Encoding on entire dataset...
         Training full model...
         Wall time: 852 ms

Out[14]: LogisticRegression(C=16.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

```
In [16]: np.savetxt("submission-LR_Cust7.csv", submission, delimiter=",")

         output = ['id,ACTION']
         for i,pred in enumerate(submission):
             output.append('%i,%f' %(i+1,pred))

In [17]: output = ['id,ACTION']
         for i,pred in enumerate(submission):
             output.append('%i,%f' %(i+1,pred))

In [18]: f = open("submission-LR_Cust7.csv", 'w')
         f.write('\n'.join(output))
         f.close()
```

Submit to Kaggle and we had our barrier breaking score of 0.90331. Had we let the model run until completion, we feel we would have had an even higher score. But we were satisfied as we met the goals set for ourselves.

Submitted an entry to Amazon.com - Employee Access Challenge, obtaining 0.90331

APPENDIX K

```
In [1]: import pandas as pd
        import xgboost as xgb
        from sklearn.preprocessing import LabelEncoder
        import numpy as np
```

```
In [2]: training=pd.read_csv("train.csv")
        df_x = training.drop(['ACTION'], axis=1)
        X= np.matrix(df_x)
        Y = training.as_matrix(["ACTION"])
        Y=np.matrix(Y)
```

```
In [3]: testing=pd.read_csv("test.csv")
        testing.index=testing["id"]
        df_testing_x = testing.drop(['id'], axis=1)
        X_test= np.matrix(df_testing_x.values)
```

```
In [ ]: Y=np.ravel(Y)
        gbm = xgb.XGBClassifier(max_depth=10, n_estimators=1500,learning_rate=0.5).fit(X, Y)
        predictions = gbm.predict(X_test)
```

```
In [ ]: test_predictVal = pd.DataFrame(columns=['ACTION'], index=testing.index, data=gbm.predict(X_test))
        test_predictVal.to_csv("submission_XGB1.csv")
        np.unique(test_predictVal.ACTION)
```

Score:

Amazon.com - Employee Access Challenge, obtaining 0.68365

APPENDIX L

Sample biased Logistic regression to change odds or each class outcome by sampling and shooting for a desired split. In the example below we went from 5:95 ratio of the provided training set to a "controlled" sample of 1000 randomly chosen but with a 25:75 ratio between the class outcomes.

We could then apply Logistic regression on it and use the ROAUC from scikit-learn to evaluate how well the model performed in terms of the Error reduction discussed in Figure 2 on Page 5 rather than fit. We would have liked more time to explore this further to develop the most optimal simple regression model

In [1]:
```
# Project
# Gabe Eapen
# Mudra Gandhi
# Purpose : Create a biased LR Sample and note observations
from __future__ import division
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
from sklearn import datasets
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
```

In [3]:
```
train = pd.read_csv('train.csv')
```

In [8]:
```
train_Y = train[train.ACTION == 1]
train_N = train[train.ACTION == 0]
```

In [10]:
```
train_N_250= train_N.sample(n=250)
```

In [11]:
```
train_Y_750 = train_Y.sample(n=750)
```

In [12]:
```
train_bias_1000 = pd.concat([train_N_250,train_Y_750])
```

In [13]:
```
train_bias_1000.head()
```

Out[13]:

	ACTION	RESOURCE	MGR_ID	ROLE_ROLLUP_1	ROLE_ROLLUP_2	ROLE_DEPTNAME	ROLE_TITLE	ROLE
17091	0	26981	54618	117961	118052	118992	118321	11790
24699	0	6977	2163	117935	117936	120694	118636	13021
28782	0	20226	23871	118752	119070	117945	119899	12736
18824	0	82376	51761	117961	118413	120370	118321	11790
20597	0	32270	21658	117980	118076	118810	120033	27561

In [14]:
```
# get X values except "ACTION" column -run
df_x = train_bias_1000.drop(['ACTION'], axis=1)
X= df_x.values
print X.shape
```

```
(1000L, 9L)
```

In [15]:
```
# Extract "ACTION" columns as Y axis -run
Y = train_bias_1000.as_matrix(["ACTION"])
print Y.shape
```

```
(1000L, 1L)
```

In [16]:

```
# fit a logistic regression model to the data
model = LogisticRegression(C=16.0)
model.fit(X,np.ravel(Y))
print(model)
```

```
LogisticRegression(C=16.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

In [18]:

```
# make predictions
expected = np.ravel(Y)
predicted = model.predict(X)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

```
             precision    recall  f1-score   support

          0       0.00      0.00      0.00       250
          1       0.75      1.00      0.86       750

avg / total       0.56      0.75      0.64      1000

[[  0 250]
 [  1 749]]
```

APPENDIX M

In [1]:
```
import numpy as np
import pandas as pd
# Gaussian Naive Bayes
from sklearn import datasets
from sklearn import metrics
from sklearn.naive_bayes import GaussianNB
```

In [2]:
```
# load training dataset - run
training=pd.read_csv("train.csv")

#Get data which has action value = 0
action0=training['ACTION'] == 0
df_action0=training[action0]

#Get data which has action value = 1
action1=training['ACTION'] == 1
df_action1=training[action1]

#Extract 1898 rows from the dataset which has rows with Action = 1
df_x=df_action1[:1898]

#Merge data which has Action=0 with extracted 1898 records which has Action=1
merge = [df_action0, df_x]
result = pd.concat(merge)
```

In [3]:
```
#Get Y value from the new dataset
Y = result.as_matrix(["ACTION"])

#Get X values from new datset
df_x = result.drop(['ACTION'], axis=1)
X= df_x.values
```

In [4]:
```
# fit a Naive Bayes model to the data
Y=np.ravel(Y)
model = GaussianNB()
model.fit(X, Y)
print(model)
```

```
GaussianNB()
```

In [5]:
```
#Load testing dataset -run
testing=pd.read_csv("test.csv", index_col='id')
```

In [6]:
```
# make predictions
test_predictVal = pd.DataFrame(columns=['ACTION'], index=testing.index, data=model.predict(testi
test_predictVal.to_csv("simplesubmission-NV.csv")
```

Score: Nothing to write home about

Submitted an entry to Amazon.com - Employee Access Challenge, obtaining 0.51433

References

[1] Kaggle.com Wiki, 'Area Under Curve', 2016. [Online]. Available: https://www.kaggle.com/wiki/AreaUnderCurve. [Accessed: 25-Apr-2016].
[2] Wikipedia.com, 'Receiver operating characteristic', 2016. [Online]. Available: https://en.wikipedia.org/wiki/Receiver_operating_characteristic. [Accessed: 23-Apr-2016].
[3] DataSchool.io, 'Simple Guide to Confusion Matrix Terminology', 2016. [Online]. Available: http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology. [Accessed: 29-Apr-2016].
[4] Art B. Owen, 'Infinitely imbalanced Logistic Regression', 2007. [Online] Available: http://www.jmlr.org/papers/volume8/owen07a/owen07a.pdf. [Accessed: 01-May-2016]
[5] Scikit-learn.org, 'Choosing the right estimator', 2016. [Online]. Available: http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html. [Accessed: 16-Apr-2016].
[6] Wikipedia.com, 'Kendall's rank correlation coefficient ($\mathbf{T}$)', 2016. [Online]. Available: https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient. [Accessed: 24-Apr-2016].
[7] Wikipedia.com, 'Spearman's rank correlation coefficient ($\boldsymbol{\rho}$)', 2016. [Online]. Available: https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient. [Accessed: 24-Apr-2016].