LAB GUIDELINES

FRAUD ANALYTICS LAB

Shay Amram, M.Sc., CISSP
Senior Data Scientist, RSA Labs
AMRAM.SHAY@GMAIL.COM
WWW.LINKEDIN.COM/IN/SHAY-AMRAM-69924051

RSA SECURITY™

DELL EMC

28-Nov-2017

Contents

Introduction

Financial fraud or bank fraud is the use of illegal or illicit means to obtain money, assets, or other property owned or held by a financial institution, or to obtain money from depositors. There is a wide variety of financial and bank fraud where predictive analytic is widely used such as on-line banking, internet credit card payment, and rogue trading.
In this lab we deal with credit loan. From a bank's perspective, loans are an asset which is, to some extent, exposed to risk, depending whether or not the lender returns the loan (the banking term for that is that the lender "defaults on the loan"). This lab applies a variety of supervised learning techniques to the "German Credit" data, and attempts to classify the sample data into **default** and **non-default**.

Objective of the Analysis

A loan application form seeks a lot of information about the applicant. Such data can be used to build a classifier to make predictions about which customers are most likely to default on their loan. The customers who have been predicted to have a high probability to default, are then declined the loan. The goal of this lab is to use historical data to build a model that could help the bank officer to decide whether or not to approve a loan request, based on the information submitted by the applicant, such as credit history, loan purpose, account status, employment, account status, the loan amount, purpose of the loan, gender and marital status of the requester, etc.
From the data science point of view, the aim is to evaluate various modeling techniques, calibrate their parameters as well as understand the trade-off between the different approaches such as data and model assumptions implied by each case. In order to be able to compare the different models, it is necessary to establish a figure of merit that is relevant to the business challenge.

This lab is organized as follows

1. Data gathering
2. Exploratory data analysis
3. Preparing the data (training and testing datasets)
4. Model training
5. Model evaluation

Data Gathering

The German credit data set is widely used for benchmarking machine learning algorithms. This data has been made available by Prof. Hofmann and it is available on the web at the following link:
https://archive.ics.uci.edu/ml/datasets/Statlog+%28German+Credit+Data%29
It is dataset of 1,000 real customers who had borrowed loan from a real German bank in which 70% repaid the loan and therefore are rated good credit, while the remaining 30% defaulted, and were rated bad credit by the bank. Along with these outcomes, the data includes customer credit history, employment, account-status, the purpose of the loan, gender, and marital status. Each loan form is summarized into 20 input fields.

LAB:

1. Copy the German.Credit.csv data from the network-folder to a new folder in your computer:
   C:\bgucourse\dataset
   If the directory doesn't exist, create it
2. Open your Jupyter-notebook environment by opening Run--> cmd
   In the command-prompt window type:

```
activate bgucourse
jupyter notebook --notebook-dir=C:\bgucourse
```

3. Open the GermanCreditLab.ipynb file (iPython Notebook) and load the German Credit data:

```
# load the dataset
fname_germancredit = r'dataset/German.Credit.csv'
data_raw = pd.read_csv(fname_germancredit)
```

set up the working directory, folder that contains the data file, called `German.Credit.csv` and then read and assign it to a dataframe.

Exploratory Data Analysis

Before getting into any sophisticated analysis, the first step is to do an exploratory data analysis in order to analyze the data sets and to summarize its main characteristics. Since both categorical and continuous variables are included in the data set, appropriate tables and summary statistics should be made.

LAB: to get an initial overview of the statistical properties of the data, use the function describe():

```
data_raw.describe()
```

	duration	credit_amount	installment_commitment	residence_since	age	existing_credits	num_dependents
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000
mean	20.903000	3271.258000	2.973000	2.845000	35.546000	1.407000	1.155000
std	12.058814	2822.736876	1.118715	1.103718	11.375469	0.577654	0.362086
min	4.000000	250.000000	1.000000	1.000000	19.000000	1.000000	1.000000
25%	12.000000	1365.500000	2.000000	2.000000	27.000000	1.000000	1.000000
50%	18.000000	2319.500000	3.000000	3.000000	33.000000	1.000000	1.000000
75%	24.000000	3972.250000	4.000000	4.000000	42.000000	2.000000	1.000000
max	72.000000	18424.000000	4.000000	4.000000	75.000000	4.000000	2.000000

NOTE: describe() will automatically list numerical variables only

List the numerical and categorical columns:

```
col_target = 'class'
cols_numeric = list(data_raw.describe().columns.values)
cols_categoric = list(set(data_raw.columns.values) - set(cols_numeric) - set([col_target]))
```

```
cols_categoric
```

```
['job',
 'employment',
 'personal_status',
 'property_magnitude',
 'credit_history',
 'purpose',
 'account_balance',
 'savings_status',
 'own_telephone',
 'housing',
 'other_parties',
 'foreign',
 'other_payment_plans']
```

Contingency Tables

A **contingency table** is a type of table in a matrix format that displays the (multivariate) frequency distribution of the variables. They are heavily used in survey research, business intelligence, engineering and scientific research. They provide a basic picture of the interrelation between two variables and can help find interactions between them. This is also an intuitive tool to explore categorical variables.

LAB: use the following code to describe 'account_balance'

```
# Contingency table
pd.crosstab(data_raw['class'], data_raw['account_balance'],  margins=True)
```

account_balance	'0<=X<200'	'<0'	'>=200'	'no account'	All
class					
bad	105	135	14	46	300
good	164	139	49	348	700
All	269	274	63	394	1000

Change `'account_balance'` and replacing it with other categorical variables.

Doing the same, using ratios.

- In your opinion, which is more informative – ratios by rows or columns? Why?

```
# Contingency table, ratios. Rows add-up to 100%
pd.crosstab(data_raw['class'], data_raw['account_balance'],  margins=False, normalize='index')
```

account_balance	'0<=X<200'	'<0'	'>=200'	'no account'
class				
bad	0.350000	0.450000	0.046667	0.153333
good	0.234286	0.198571	0.070000	0.497143

```
# Contingency table, ratios. Columns add-up to 100%
pd.crosstab(data_raw['class'], data_raw['account_balance'],  margins=False, normalize='columns')
```

account_balance	'0<=X<200'	'<0'	'>=200'	'no account'
class				
bad	0.390335	0.492701	0.222222	0.116751
good	0.609665	0.507299	0.777778	0.883249

The tables shown counts, and ratios. First row-ratios, then column ratios. Looking at the first table there are 274 (27.4%) of 1000 applicants who have a negative balance. Another 26.9% have some budget (less than 200 DM) in their account while 39.4% do not have an account in the bank. Among those who have negative balance 135 are found to be non-creditable and 139 are found to be creditable. In the group with some budget in their account, 61% were found to be creditable whereas in the group that do not has an account in the bank only 11.7% are found to be Non-creditable.

LAB: use the p-value to determine the features that seems to be most relevant (use $p < 0.10$) for all numerical predictors. Use the following code to run over all categorical variables, in a loop:

```
for col in cols_categoric:
    contingency = pd.crosstab(data_raw['class'], data_raw[col])
    c, pval, dof, expected = chi2_contingency(contingency)
    print("p-value:",pval, '\t variable:', col)
```

- Which variables do you expect to be most relevant?

Descriptive Statistics of Numerical Predictors

Correlation between variables is important to understand the data, but it is also important from a modeling perspective: many algorithms have issues with dependent or correlated variables. In some cases correlated variables will "only" increase the contribution of a certain feature, which masks the contribution of the rest of the predictors (such is the case with Naïve Bayes, for example). In other cases, the algorithm itself might altogether crash. Common example: any algorithm that has to calculate inverse matrix of the predictors will probably have serious issues with dependent predictors (non-invertible matrix, and therefore, would crash).

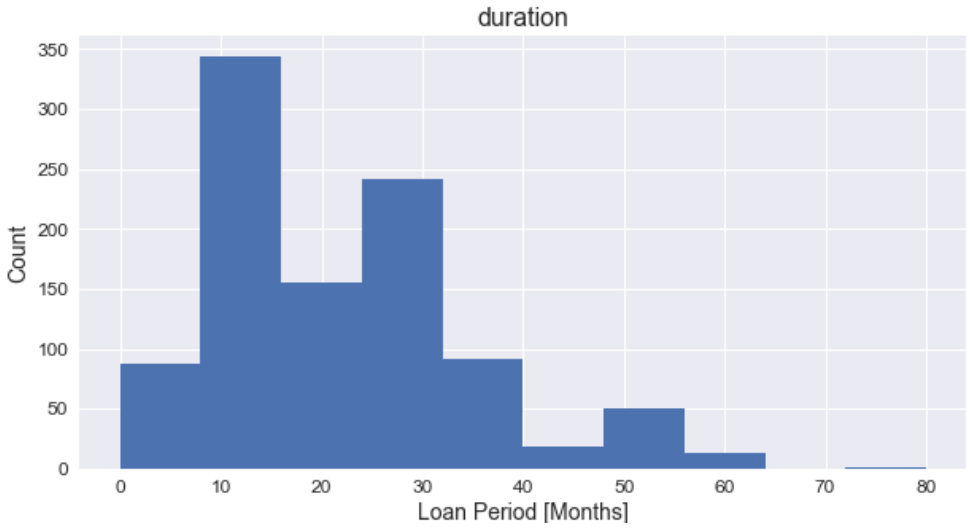
LAB:

```
# Correlation between numeric variables
data_numeric = data_raw[cols_numeric].copy(deep=True)
corr_mat = data_numeric.corr(method='pearson')
cbar_ticks =np.linspace(-1,1,11)
cmap = sns.diverging_palette(220, 10, as_cmap=True)
plt.figure(figsize=[8,8])
plt.xticks(fontsize=fontsz+2)
plt.yticks(fontsize=fontsz+2)
ax = sns.heatmap(corr_mat, cmap=cmap, vmin=-1, vmax=1, square=True, linewidths=.5, cbar_kws={"shrink": .5})
cbar = ax.collections[0].colorbar
cbar.set_ticks(cbar_ticks)
cbar.set_ticklabels(cbar_ticks)
plt.show()
```

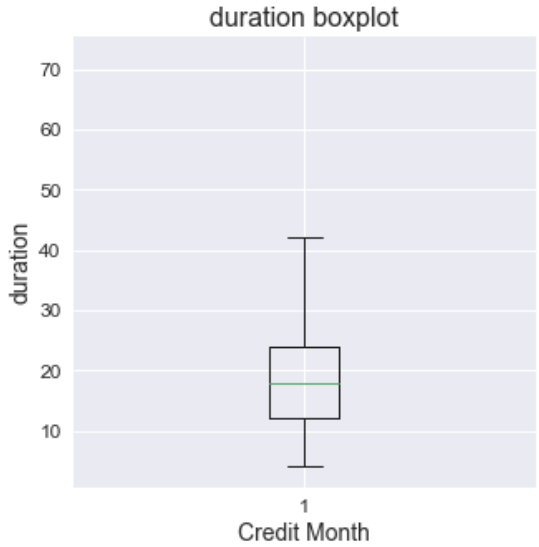
- Inspect the correlation matrix by typing
  `print (tabulate.tabulate(corr_mat))`
  Are there highly correlated variables (use |0.8| as a threshold)?
  Can you describe other variables (if any) that are somewhat correlated? Why?

LAB: **Histograms** and **boxplots** are also a useful way to explore the numerical predictors:

```
brksCredits = np.linspace(0,80,11) # Bins for a nice looking histogram
plt.figure(figsize=(10,5))
plt.hist(data_raw['duration'], bins=brksCredits)
plt.title('duration', fontsize=fontsz+4)
plt.xlabel('Loan Period [Months]', fontsize=fontsz+2)
plt.ylabel('Count', fontsize=fontsz+2)
plt.xticks(fontsize=fontsz)
plt.yticks(fontsize=fontsz)
plt.show()
```



```
plt.figure(figsize=(5,5))
plt.boxplot(data_raw['duration'])
plt.title('duration boxplot', fontsize=fontsz+4)
plt.xlabel('Credit Month', fontsize=fontsz+2)
plt.ylabel('duration', fontsize=fontsz+2)
plt.xticks(fontsize=fontsz)
plt.yticks(fontsize=fontsz)
plt.show()
```



The **box plot** (a.k.a. box and whisker diagram) is a standardized way of displaying the distribution of data based on the five number summary: minimum, first quartile, median, third quartile, and maximum.

Preprocessing the Data

Often, we'll need to make some preprocessing to the data, such as removing nulls, converting formats, or replacing some values. There is a common type of preprocessing required in Python: converting categorical variables to dummy variables.

A dummy variable is a numerical variable used in regression analysis to represent subgroups of the sample in your study. In research design, a dummy variable is often used to distinguish different treatment groups. For example, Logistic regression can't multiply the coefficient c_i by "personal_status" (values: 'female single ', 'male single', etc.). Instead we replace "personal_status" categorical predictor, with dummy variables, each one corresponding to one value in "personal_status".

A similar task, is converting the "class" variable. "class" is the label of each customer request. "Good" indicates that the loan was returned, while "bad" indicates the customer defaulted on the load (this is the term used to describe people who couldn't repay their debt. Try not to be those, if possible).

"bad" will be converted to the numerical value 0, and "good" to 1. When the algorithm prediction → 1, the bank is advised to grant the loan, while with 0, the bank will deny the loan. This is purely semantic, and roles can be reversed, with reversed meanings.

LAB: Replace categorical variables with dummy-variables

```
print ("Number of columns before dummy-variables:\t", len(data_raw.columns.values))
for i in cols_categoric:
    dummy_ranks = pd.get_dummies(data_raw[i], prefix=i)
    data_raw = data_raw.join(dummy_ranks)
    # dropping the original categoric column (not needed - it was replaced by dummy columns)
    data_raw = data_raw.drop(i, 1)

# all feature, numeric, and categoric (now dummified)
cols_features = list(set(data_raw.columns.values) - set([col_target]))
print ("Number of columns after dummy-variables:\t", len(data_raw.columns.values))
```

```
Number of columns before dummy-variables:       21
Number of columns after dummy-variables:        62
```

LAB: Replace 'bad' and 'good' class labels with 0 and 1, before continuing with the exercise:

```
data_raw['class'].replace('bad', 0, inplace=True)
data_raw['class'].replace('good', 1, inplace=True)
data_raw['class'] = pd.to_numeric(data_raw['class'])
```

Model Evaluation

When a bank receives a loan application, based on the applicant's profile the bank has to make a decision regarding whether to go ahead with the loan approval or not. Two types of risks are associated with the bank's decision:

- If the applicant got a good credit risk, i.e. is likely to repay the loan, then not approving the loan to the person results in a loss of business to the bank.
- If the applicant got a bad credit risk, i.e. is not likely to repay the loan, then approving the loan to the person results in a financial loss to the bank.

The goal is to minimize loss from the bank's perspective using a decision engine regarding who to give approval of the loan and who not to. An applicant's demographic and socio-economic profiles are considered by loan managers before a decision is taken regarding his/her loan application.

Training and testing datasets

In order to be able to evaluate the prediction power of a method it is desirable to spit the data set into two datasets:

- **Training dataset**: the models are built based on the observations included in this set. This set of data mimics the "historical" data used to tune the parameter of each one of the classifiers.
- **Testing dataset**: the model is applied on this set to evaluate its performance. It can be considered as new applicants for credit to be evaluated on these 20 predictor variables. The credit scoring that these customers got are used to evaluate to what extend the classification is correct.

LAB:

- Use the command `random.seed()` to ensure the results are reproducible. During this class use `seed=1013`
- Split the data using the function, `train_test_split()`

```
# Split the data using the function, train_test_split()
frac_train = 0.8 # 80% of the data is used for training
X_train, X_test, y_train, y_test = \
    train_test_split(data_raw[cols_features], data_raw[col_target], test_size=(1-frac_train), random_state=seed)

train_b = sum(y_train == 0)
train_g = sum(y_train == 1)
test_b = sum(y_test == 0)
test_g = sum(y_test == 1)
print ("Class ratios between each set:")
print ("Trainset")
print ("\t\tNormal class (good):", 100*train_g/len(y_train), "%\t", "Target class (bad):", 100*train_b/len(y_train),"%")
print ("Testset")
print ("\t\tNormal class (good):", 100*test_g/len(y_test), "%\t", "Target class (bad):", 100*test_b/len(y_test),"%")
```

  - `X_train : training data`
  - `y_train : labels for training data`
  - `X_test  : testing data`
  - `y_test  : labels for testing data`

  The `train_test_split()` function splits the data randomly, though it is usually a good idea to split the data so that there's a similar distribution of labels ('bad', 'good', etc.)

Performance Measurements

Different figures of merit have been introduced in the literature for the evaluation of classification performance, each of them with different origins and areas of application. These metrics include **accuracy**, the **area under curve** (**AUC**), the **ROC** convex hull, and many others (to name just a few, link). It is important to choose performance metrics that correspond well to the problem at hand. In this lab we use two metrics:

- Area under the ROC curve (AUC)
- LOSS

Probably the most widely used summary index is the area under the **ROC** curve, commonly denoted **AUC**. One interpretation of the AUC is that it is the average true positive rate, taken uniformly over all possible false positive rates in the range (0, 1). A less obvious, but frequently used interpretation is the probability that the classifier will allocate a higher score to a randomly chosen sample from "positive"-population *P* than it will to a randomly and independently chosen individual from the "negative"-population *N*. The AUC is by far the most popular index used, however it does not take into account the misclassification cost.

There are multiple Loss (sometimes referred to as cost) functions that are in use. The course material includes optional reading material for other commonly used loss-functions (Link). For the purpose of this lab, use a simplified version that penalizes for misclassifications – false-negative and false-positive:

$$Misclassification\ Loss = C_{FN} \cdot FN + C_{FP} \cdot FP$$

Where:

- C_{FN} is the cost of a False-Negative
- *FN* is the number of False-Negatives in the set
- C_{FP} is the cost of a False-Positive
- *FP* is the number of False-Positives in the set

The classifications are typically summarized in a **Confusion Matrix**:

		Predicted Class	
		0	1
Actual Class	0	True Negative (TN)	False Positive (FP)
	1	False Negative (FN)	True Positive (TP)

LAB: Use the cost parameters:

```
# Set Misclassification loss weights
c_tn = 0 # cost of true-negative
c_tp = 0 # cost of true-positive
c_fn = 1 # cost of false negative
c_fp = 5 # cost of false positive
```

Optimal Threshold

Use "Optimal Threshold" to compare the various models. The optimal threshold is the threshold where the confusion matrix gives optimal results, with respect to some misclassification-loss function.

The decision threshold is prediction:

x < prediction-probability: classify as negative class (0)

x >= prediction-probability: classify as positive class (1)

For more reading, please refer to the document "`One_ROC_Curve_and_Cutoff_Analysis.pdf`" in your course reading material. We use p. #7, but the other pages are interesting too! ☺

Modeling

The models selected in this lab can be considered as the reference models each one optimal in a different scenarios. They are commonly used as benchmark for new machine learning algorithms

1. Logistic regression
2. Naive Bayes
3. Classification tree (Decision Tree)
4. Random forest

For each one of them the following sequence is preformed:

1. Train the model
2. Display the obtained model along with most relevant statistics
3. Test the model
4. Draw the ROC and Calculate the AUC
5. Calculate the total misclassification loss and the optimal threshold (**using the training data**).
   Can you explain why?
6. Build the confusion matrix for the tests data for both the **default** and **optimal threshold**
7. Display the confusion matrices

Logistic regression

LAB:

1. Train the model

```
model = linear_model.LogisticRegression()
model.fit(X_train, y_train)
```

2. Display the obtained model along with most relevant statistics

```
model_coefficients = model.coef_[0]
df_lgm_coeffs = pd.DataFrame(data=[list(cols_features), list(model_coefficients)]).transpose()
df_lgm_coeffs.columns = ['feature', 'LGM_coeff']
# sort by coefficients absolute value
df_lgm_coeffs = df_lgm_coeffs.reindex(df_lgm_coeffs['LGM_coeff'].abs().sort_values(inplace=False, ascending=False).index)
display(df_lgm_coeffs)
```

- Based on the coefficients alone, would you give out a load for someone who wants to buy a car?

3. Test the model

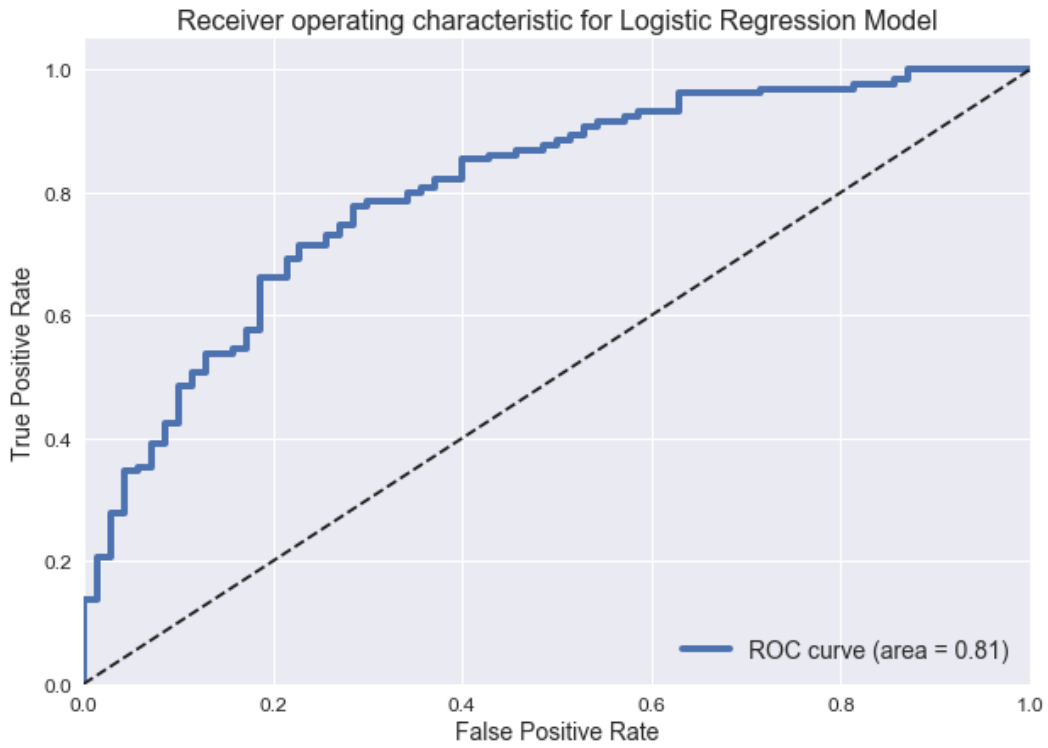
```
predicted = model.predict(X_test)
predicted_prob = model.predict_proba(X_test)[:,1]
```

The difference between `predict()` and `predict_proba()` is that the first assigns labels, and the second calculated probability of the label.

4. Draw ROC Curve and calculate AUC

```
fpr, tpr, _ = metrics.roc_curve(np.array(y_test), predicted_prob)
auc = metrics.auc(fpr,tpr)
print ("Area-Under-Curve:", round(auc,4))
# plot_ROC() is defined in helper_functions.py
plot_ROC(fpr,tpr, fontsz, 'Receiver operating characteristic for Logistic Regression Model')
```

```
Area-Under-Curve: 0.8055
```



5. Calculate the total misclassification loss

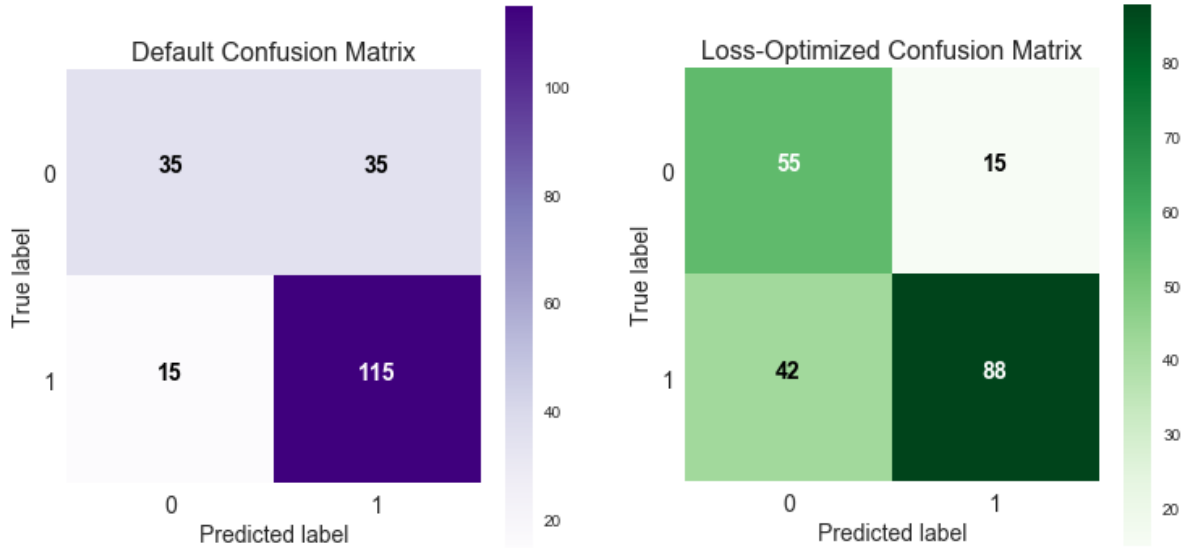
```
# finding the optimal values using the TRAIN-SET
train_predicted_prob = model.predict_proba(X_train)[:,1]
loss_matrix = calculate_loss(train_predicted_prob, y_train, c_fn, c_fp, c_tp, c_tn)
# finding optimal threshold:
opt_thr = list(loss_matrix[loss_matrix['loss'] == loss_matrix['loss'].min()]['prediction'])[0]
print("Optimal threshold at:\t",round(opt_thr,5))
print("Model Loss:", loss_matrix['loss'].min())
loss = loss_matrix['loss'].min()
predicted_prob_opt = copy.deepcopy(predicted_prob)
predicted_prob_opt[predicted_prob_opt >  opt_thr] = 1
predicted_prob_opt[predicted_prob_opt <= opt_thr] = 0
```

6. Build the confusion matrix for the tests data for both the default and optimal thresholds

```
def_cfm = metrics.confusion_matrix(y_test, predicted) # default confusion matrix, default threshold = 0.5
opt_cfm = metrics.confusion_matrix(y_test, predicted_prob_opt) # optimal threshold
```

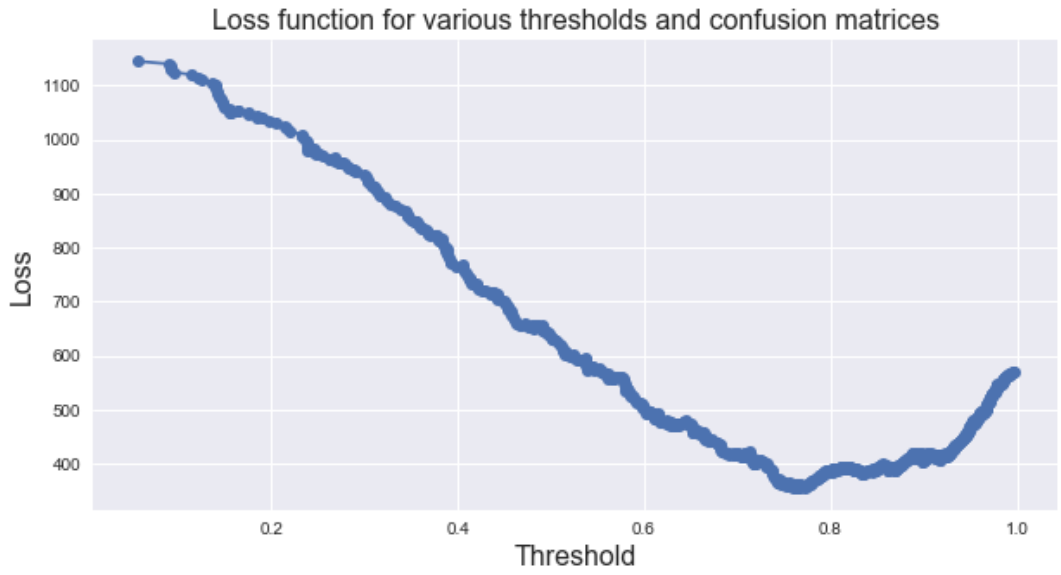
7. Plot the confusion matrices

```
plot_confusion_matrix(def_cfm,['bad', 'good'], "Default Confusion Matrix", 0)
plot_confusion_matrix(opt_cfm,['bad', 'good'], "Loss-Optimized Confusion Matrix", 1)
plt.show()
```



[**Optional**]: Plot the misclassification-loss vs threshold

```
plt.figure(figsize=(10,5), facecolor='white')
plt.plot(loss_matrix['prediction'], loss_matrix['loss'], 'o-')
plt.title('Loss function for various thresholds and confusion matrices', fontsize=fontsz+4)
plt.xlabel('Threshold', fontsize=fontsz+4)
plt.ylabel('Loss', fontsize=fontsz+4)
plt.show()
```



NOTE: To speed things along in the following models some of the steps will be skipped, or be done simultaneously. Feel free to copy & paste code

Naïve Bayes

LAB:

1. Train the model and apply predictions

```
gnb = BernoulliNB()
model = gnb.fit(X_train, y_train)
predicted = model.predict(X_test)
predicted_prob = model.predict_proba(X_test)[:, 1]
```

2. Display the obtained model along with most relevant statistics
   In Naïve Bayes, the coefficients are related the log-probability of the prediction.
   NOTE: `sort_values()` is **ASCENDING**

```
model_coefficients = model.coef_[0]
df_coeffs = pd.DataFrame(data=[list(cols_features), list(model_coefficients)]).transpose()
df_coeffs.columns = ['feature', 'coeff']
# sort by coefficients absolute value (log probability of the positive class)
df_coeffs = df_coeffs.reindex(df_coeffs['coeff'].abs().sort_values(inplace=False, ascending=True).index)
display(df_coeffs)
```

	feature	coeff
59	residence_since	-0.00174978
57	num_dependents	-0.00174978
40	existing_credits	-0.00174978
9	duration	-0.00174978

- Look-up the car-related coefficients from Logistic Regression. Do the models agree on the importance of the car-related predictors?

3. Draw ROC Curve and calculate AUC

```
fpr, tpr, _ = metrics.roc_curve(np.array(y_test), predicted_prob)
auc = metrics.auc(fpr,tpr)
print ("Area-Under-Curve:", round(auc,4))
plot_ROC(fpr,tpr, fontsz, 'Receiver operating characteristic for Naive Bayes Model')
```

4. Calculate the total misclassification loss

```
# finding the optimal values using the TRAIN-SET
train_predicted_prob = model.predict_proba(X_train)[:,1]
loss_matrix = calculate_loss(train_predicted_prob, y_train, c_fn, c_fp, c_tp, c_tn)
# finding optimal threshold:
opt_thr = list(loss_matrix[loss_matrix['loss'] == loss_matrix['loss'].min()]['prediction'])[0]
print("Optimal threshold at:\t",round(opt_thr,5))
print("Model Loss:", loss_matrix['loss'].min())
loss = loss_matrix['loss'].min()
predicted_prob_opt = copy.deepcopy(predicted_prob)
predicted_prob_opt[predicted_prob_opt >  opt_thr] = 1
predicted_prob_opt[predicted_prob_opt <= opt_thr] = 0
```

5. Build the confusion matrix for the tests data for both the default and optimal thresholds

```
def_cfm = metrics.confusion_matrix(y_test, predicted) # default confusion matrix, default threshold = 0.5
opt_cfm = metrics.confusion_matrix(y_test, predicted_prob_opt) # optimal threshold
```

6. Display the confusion matrices

```
plot_confusion_matrix(def_cfm,['bad', 'good'], "Default Confusion Matrix", 0)
plot_confusion_matrix(opt_cfm,['bad', 'good'], "Loss-Optimized Confusion Matrix", 1)
plt.show()
```

Recursive Partitioning Tree (Decision Tree)

LAB:

1. Train the model and apply predictions

```
md = 18                     # maximum tree depth
mf = len(cols_features)     # maximum number of features to consider
min_leaf = 10
criterion = 'entropy'
model = tree.DecisionTreeClassifier(max_depth=md, max_features=mf, criterion=criterion,
                                    min_samples_leaf=min_leaf, random_state=seed)

clf = model.fit(X_train, y_train)
predicted = model.predict(X_test)
predicted_prob = model.predict_proba(X_test)[:, 1]
```

2. Display the obtained model along with most relevant statistics
   Decision Trees can display the relative importance of each feature

```
importance = model.feature_importances_
df_importance = pd.DataFrame(data=[list(cols_features), list(importance)]).transpose()
df_importance.columns = ['feature', 'importance']
df_importance = df_importance[df_importance['importance'] != 0]
# sort by feature importance
df_importance = df_importance.reindex(df_importance['importance'].abs().sort_values(inplace=False, ascending=False).index)
display(df_importance)
```

	feature	importance
14	account_balance_'no account'	0.269599
39	credit_amount	0.146782

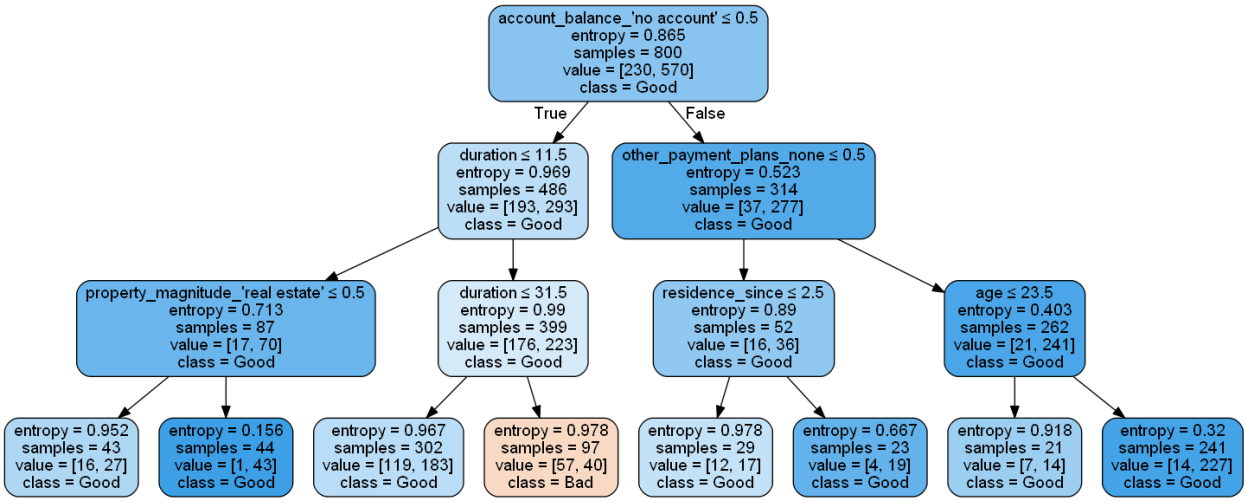


Figure 1: Example visualization of the decision tree output rules and predictions. Nifty.

3. Draw ROC Curve and calculate AUC

```
fpr, tpr, _ = metrics.roc_curve(np.array(y_test), predicted_prob)
auc = metrics.auc(fpr,tpr)
print ("Area-Under-Curve:", round(auc,4))
plot_ROC(fpr,tpr, fontsz, "Receiver operating characteristic for Decision Tree Model")
```

4. Calculate the total misclassification loss

```
# finding the optimal values using the TRAIN-SET
train_predicted_prob = model.predict_proba(X_train)[:,1]
loss_matrix = calculate_loss(train_predicted_prob, y_train, c_fn, c_fp, c_tp, c_tn)
# finding optimal threshold:
opt_thr = list(loss_matrix[loss_matrix['loss'] == loss_matrix['loss'].min()]['prediction'])[0]
print("Optimal threshold at:\t",round(opt_thr,5))
print("Model Loss:", loss_matrix['loss'].min())
loss = loss_matrix['loss'].min()
predicted_prob_opt = copy.deepcopy(predicted_prob)
predicted_prob_opt[predicted_prob_opt >  opt_thr] = 1
predicted_prob_opt[predicted_prob_opt <= opt_thr] = 0
```

5. Build the confusion matrix for the tests data for both the default and optimal thresholds

```
def_cfm = metrics.confusion_matrix(y_test, predicted) # default confusion matrix, default threshold = 0.5
opt_cfm = metrics.confusion_matrix(y_test, predicted_prob_opt) # optimal threshold
```

6. Display the confusion matrices

```
plot_confusion_matrix(def_cfm,['bad', 'good'], "Default Confusion Matrix", 0)
plot_confusion_matrix(opt_cfm,['bad', 'good'], "Loss-Optimized Confusion Matrix", 1)
plt.show()
```

Random Forest

NOTE: Random Forest will use the same parameters as the decision tree, for comparative purposes

LAB:

1. Train the model and apply predictions

```
model = RandomForestClassifier(max_depth=md, max_features=mf,
                               criterion=criterion, min_samples_leaf = min_leaf, random_state=seed)
clf = model.fit(X_train, y_train)
```

2. Display the obtained model along with most relevant statistics
   Random Forest displays the relative importance of each feature

```
importance = model.feature_importances_
df_importance = pd.DataFrame(data=[list(cols_features), list(importance)]).transpose()
df_importance.columns = ['feature', 'importance']
df_importance = df_importance[df_importance['importance'] != 0]
# sort by feature importance
df_importance = df_importance.reindex(df_importance['importance'].abs().sort_values(inplace=False, ascending=False).index)
display(df_importance)
```

3. Draw ROC Curve and calculate AUC

```
predicted = clf.predict(X_test)
predicted_prob = clf.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = metrics.roc_curve(np.array(y_test), predicted_prob)
auc = metrics.auc(fpr,tpr)
print ("Area-Under-Curve:", round(auc,4))
plot_ROC(fpr,tpr, fontsz, "Receiver operating characteristic for Random Forest Model")
```

4. Calculate the total misclassification loss

```
# finding the optimal values using the TRAIN-SET
train_predicted_prob = model.predict_proba(X_train)[:,1]
loss_matrix = calculate_loss(train_predicted_prob, y_train, c_fn, c_fp, c_tp, c_tn)
# finding optimal threshold:
opt_thr = list(loss_matrix[loss_matrix['loss'] == loss_matrix['loss'].min()]['prediction'])[0]
print("Optimal threshold at:\t",round(opt_thr,5))
print("Model Loss:", loss_matrix['loss'].min())
loss = loss_matrix['loss'].min()
predicted_prob_opt = copy.deepcopy(predicted_prob)
predicted_prob_opt[predicted_prob_opt >  opt_thr] = 1
predicted_prob_opt[predicted_prob_opt <= opt_thr] = 0
```

5. Build the confusion matrix for the tests data for both the default and optimal thresholds

```
def_cfm = metrics.confusion_matrix(y_test, predicted) # default confusion matrix, default threshold = 0.5
opt_cfm = metrics.confusion_matrix(y_test, predicted_prob_opt) # optimal threshold
```

6. Display the confusion matrices

```
plot_confusion_matrix(def_cfm,['bad', 'good'], "Default Confusion Matrix", 0)
plot_confusion_matrix(opt_cfm,['bad', 'good'], "Loss-Optimized Confusion Matrix", 1)
plt.show()
```

- Summarize AUCs and Classification-Loss per model
- Rank the models per each (AUC, Loss)
- Which model is the best?

Homework

- Perform 6-fold cross-validation, and note the results (AUC and Loss).
  (That is, an additional 5 experiments, not counting the one we did in class, with seed 1017).
  You'll achieve that by rerunning the code, and using the following seeds. Change the seed parameter for each iteration:
  [18671107, 18870812, 19010929, 19040422, 18790314]

- Summarize your results for each experiment:

Experiment	Logistic Regression	…
Exp1		
Exp2		
…		

- Calculate the mean and STD of each classifier's performance (AUC and Loss).
  Present your results in a graph.

- Rank the models again, this time, based on the mean and STD results.

- What is the classifier of your choice? Why?