Identifying safe loans with decision trees

The LendingClub (https://www.lendingclub.com/) is a peer-to-peer leading company that directly connects borrowers and potential lenders/investors. In this notebook, you will build a classification model to predict whether or not a loan provided by LendingClub is likely to default (https://en.wikipedia.org/wiki/Default_%28finance%29).

In this notebook you will use data from the LendingClub to predict whether a loan will be paid off in full or the loan will be charged off (https://en.wikipedia.org/wiki/Charge-off) and possibly go into default. In this assignment you will:

- Use SFrames to do some feature engineering.
- Train a decision-tree on the LendingClub dataset.
- Visualize the tree.
- Predict whether a loan will default along with prediction probabilities (on a validation set).
- Train a complex tree model and compare it to simple tree model.

Let's get started!

Fire up GraphLab Create

Make sure you have the latest version of GraphLab Create. If you don't find the decision tree module, then you would need to upgrade GraphLab Create using

```
pip install graphlab-create --upgrade
```

In [1]:

```
import graphlab
graphlab.canvas.set_target('ipynb')
```

Load LendingClub dataset

We will be using a dataset from the LendingClub (https://www.lendingclub.com/). A parsed and cleaned form of the dataset is availiable here (https://github.com/learnml/machine-learning-specialization-private). Make sure you **download the dataset** before running the following command.

In [2]:

```
loans = graphlab.SFrame('lending-club-data.gl/')
```

```
This non-commercial license of GraphLab Create for academic use is assigned
to amitha353@gmail.com and will expire on May 07, 2019.
```

```
[INFO] graphlab.cython.cy_server: GraphLab Create v2.1 started. Logging:
C:\Users\Amitha\AppData\Local\Temp\graphlab_server_1532762274.log.0
```

Exploring some features

Let's quickly explore what the dataset looks like. First, let's print out the column names to see what features we have in this dataset.

In [3]:

```
loans.column_names()
```

Out[3]:

```
['id',
 'member_id',
 'loan_amnt',
 'funded_amnt',
 'funded_amnt_inv',
 'term',
 'int_rate',
 'installment',
 'grade',
 'sub_grade',
 'emp_title',
 'emp_length',
 'home_ownership',
 'annual_inc',
 'is_inc_v',
 'issue_d',
 'loan_status',
 'pymnt_plan',
 'url',
 'desc',
 'purpose',
 'title',
 'zip_code',
 'addr_state',
 'dti',
 'delinq_2yrs',
 'earliest_cr_line',
 'inq_last_6mths',
 'mths_since_last_delinq',
 'mths_since_last_record',
 'open_acc',
 'pub_rec',
 'revol_bal',
 'revol_util',
 'total_acc',
 'initial_list_status',
 'out_prncp',
 'out_prncp_inv',
 'total_pymnt',
 'total_pymnt_inv',
 'total_rec_prncp',
 'total_rec_int',
 'total_rec_late_fee',
 'recoveries',
 'collection_recovery_fee',
 'last_pymnt_d',
 'last_pymnt_amnt',
 'next_pymnt_d',
 'last_credit_pull_d',
 'collections_12_mths_ex_med',
 'mths_since_last_major_derog',
 'policy_code',
 'not_compliant',
 'status',
 'inactive_loans',
 'bad_loans',
 'emp_length_num',
 'grade_num',
 'sub_grade_num',
 'delinq_2yrs_zero',
 'pub_rec_zero',
 'collections_12_mths_zero',
 'short_emp',
 'payment_inc_ratio',
 'final_d',
 'last_delinq_none',
 'last_record_none',
 'last_major_derog_none']
```

Here, we see that we have some feature columns that have to do with grade of the loan, annual income, home ownership status, etc. Let's take a look at the distribution of loan grades in the dataset.

In [4]:

```
loans['grade'].show()
```

Most frequent items from *<SArray>*

Value	Count	Percent	
B	37,172	30.318%	
C	29,950	24.428%	
A	22,314	18.2%	
D	19,175	15.639%	
E	8,990	7.332%	
F	3,932	3.207%	
G	1,074	0.876%	

We can see that over half of the loan grades are assigned values `B` or `C`. Each loan is assigned one of these grades, along with a more finely discretized feature called `sub_grade` (feel free to explore that feature column as well!). These values depend on the loan application and credit report, and determine the interest rate of the loan. More information can be found here (https://www.lendingclub.com/public/rates-and-fees.action).

Now, let's look at a different feature.

In [5]:

```
loans['home_ownership'].show()
```

Most frequent items from <*SArray*>

Value	Count	Percent
MORTGAGE	59,240	48.317%
RENT	53,245	43.427%
OWN	9,943	8.11%
OTHER	179	0.146%

This feature describes whether the loanee is mortaging, renting, or owns a home. We can see that a small percentage of the loanees own a home.

Exploring the target column

The target column (label column) of the dataset that we are interested in is called `bad_loans`. In this column **1** means a risky (bad) loan **0** means a safe loan.

In order to make this more intuitive and consistent with the lectures, we reassign the target to be:

- **+1** as a safe loan,
- **-1** as a risky (bad) loan.

We put this in a new column called `safe_loans`.

In [6]:

```
# safe_loans =  1 => safe
# safe_loans = -1 => risky
loans['safe_loans'] = loans['bad_loans'].apply(lambda x : +1 if x==0 else -1)
loans = loans.remove_column('bad_loans')
```

Now, let us explore the distribution of the column `safe_loans`. This gives us a sense of how many safe and risky loans are present in the dataset.

In [7]:

```
loans['safe_loans'].show(view = 'Categorical')
```

Most frequent items from *<SArray>*

Value	Count	Percent
1	99,457	81.119%
-1	23,150	18.881%

You should have:

- Around 81% safe loans
- Around 19% risky loans

It looks like most of these loans are safe loans (thankfully). But this does make our problem of identifying risky loans challenging.

Features for the classification algorithm

In this assignment, we will be using a subset of features (categorical and numeric). The features we will be using are **described in the code comments** below. If you are a finance geek, the LendingClub (https://www.lendingclub.com/) website has a lot more details about these features.

In [8]:

```
features = ['grade',                     # grade of the loan
            'sub_grade',                 # sub-grade of the loan
            'short_emp',                 # one year or less of employment
            'emp_length_num',            # number of years of employment
            'home_ownership',            # home_ownership status: own, mortgage or rent
            'dti',                       # debt to income ratio
            'purpose',                   # the purpose of the loan
            'term',                      # the term of the loan
            'last_delinq_none',          # has borrower had a delinquincy
            'last_major_derog_none',     # has borrower had 90 day or worse rating
            'revol_util',                # percent of available credit being used
            'total_rec_late_fee',        # total late fees received to day
           ]

target = 'safe_loans'                   # prediction target (y) (+1 means safe, -1 is risky

# Extract the feature columns and target column
loans = loans[features + [target]]
```

What remains now is a **subset of features** and the **target** that we will use for the rest of this notebook.

Sample data to balance classes

As we explored above, our data is disproportionally full of safe loans. Let's create two datasets: one with just the safe loans (`safe_loans_raw`) and one with just the risky loans (`risky_loans_raw`).

In [9]:

```
safe_loans_raw = loans[loans[target] == +1]
risky_loans_raw = loans[loans[target] == -1]
print "Number of safe loans  : %s" % len(safe_loans_raw)
print "Number of risky loans : %s" % len(risky_loans_raw)
```

```
Number of safe loans  : 99457
Number of risky loans : 23150
```

Now, write some code to compute below the percentage of safe and risky loans in the dataset and validate these numbers against what was given using `.show` earlier in the assignment:

In [10]:

```
print "Percentage of safe loans  :",(len(safe_loans_raw)/float(len(safe_loans_raw) + len(ri
print "Percentage of risky loans :",(len(risky_loans_raw)/float(len(safe_loans_raw) + len(r
```

```
Percentage of safe loans  : 0.811185331996
Percentage of risky loans : 0.188814668004
```

One way to combat class imbalance is to undersample the larger class until the class distribution is approximately half and half. Here, we will undersample the larger class (safe loans) in order to balance out our dataset. This means we are throwing away many data points. We used `seed=1` so everyone gets the same results.

In [11]:

```
# Since there are fewer risky loans than safe loans, find the ratio of the sizes
# and use that percentage to undersample the safe loans.
percentage = len(risky_loans_raw)/float(len(safe_loans_raw))

risky_loans = risky_loans_raw
safe_loans = safe_loans_raw.sample(percentage, seed=1)

# Append the risky_loans with the downsampled version of safe_loans
loans_data = risky_loans.append(safe_loans)
```

Now, let's verify that the resulting percentage of safe and risky loans are each nearly 50%.

In [12]:

```
print "Percentage of safe loans                 :", len(safe_loans) / float(len(loans_data)
print "Percentage of risky loans                :", len(risky_loans) / float(len(loans_data
print "Total number of loans in our new dataset :", len(loans_data)
```

```
Percentage of safe loans                 : 0.502236174422
Percentage of risky loans                : 0.497763825578
Total number of loans in our new dataset : 46508
```

Note: There are many approaches for dealing with imbalanced data, including some where we modify the learning approaches are beyond the scope of this course, but some of them are reviewed in this paper (http://ieeexplore.iee tp=&arnumber=5128907&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F69%2F5173046%2F05128907.pdf For this assignment, we use the simplest possible approach, where we subsample the overly represented class to dataset. In general, and especially when the data is highly imbalanced, we recommend using more advanced meth

Split data into training and validation sets

We split the data into training and validation sets using an 80/20 split and specifying `seed=1` so everyone gets the same results.

Note: In previous assignments, we have called this a **train-test split**. However, the portion of data that we don't train on will be used to help **select model parameters** (this is known as model selection). Thus, this portion of data should be called a **validation set**. Recall that examining performance of various potential models (i.e. models with different parameters) should be on validation set, while evaluation of the final selected model should always be on test data. Typically, we would also save a portion of the data (a real test set) to test our final model on or use cross-validation on the training set to select our final model. But for the learning purposes of this assignment, we won't do that.

In [13]:

```
train_data, validation_data = loans_data.random_split(.8, seed=1)
```

Use decision tree to build a classifier

Now, let's use the built-in GraphLab Create decision tree learner to create a loan prediction model on the training data. (In the next assignment, you will implement your own decision tree learning algorithm.) Our feature columns and target column have already been decided above. Use `validation_set=None` to get the same results as everyone else.

In [14]:

```
decision_tree_model = graphlab.decision_tree_classifier.create(train_data, validation_set=N
                                target = target, features = features)
```

```
Decision tree classifier:

--------------------------------------------------------

Number of examples          : 37224

Number of classes           : 2

Number of feature columns   : 12

Number of unpacked features : 12

+-----------+--------------+-------------------+-------------------+

| Iteration | Elapsed Time | Training-accuracy | Training-log_loss |

+-----------+--------------+-------------------+-------------------+

| 1         | 0.085014     | 0.640581          | 0.663259          |

+-----------+--------------+-------------------+-------------------+
```

Visualizing a learned model

As noted in the documentation (https://dato.com/products/create/docs/generated/graphlab.boosted_trees_classifier.create.html#graphlab.boosted typically the max depth of the tree is capped at 6. However, such a tree can be hard to visualize graphically. Here, smaller model with **max depth of 2** to gain some intuition by visualizing the learned tree.

In [15]:

```
small_model = graphlab.decision_tree_classifier.create(train_data, validation_set=None,
                   target = target, features = features, max_depth = 2)
```

```
Decision tree classifier:

--------------------------------------------------------

Number of examples          : 37224

Number of classes           : 2

Number of feature columns   : 12

Number of unpacked features : 12

+-----------+--------------+-------------------+-------------------+

| Iteration | Elapsed Time | Training-accuracy | Training-log_loss |

+-----------+--------------+-------------------+-------------------+

| 1         | 0.064308     | 0.613502          | 0.676098          |

+-----------+--------------+-------------------+-------------------+
```

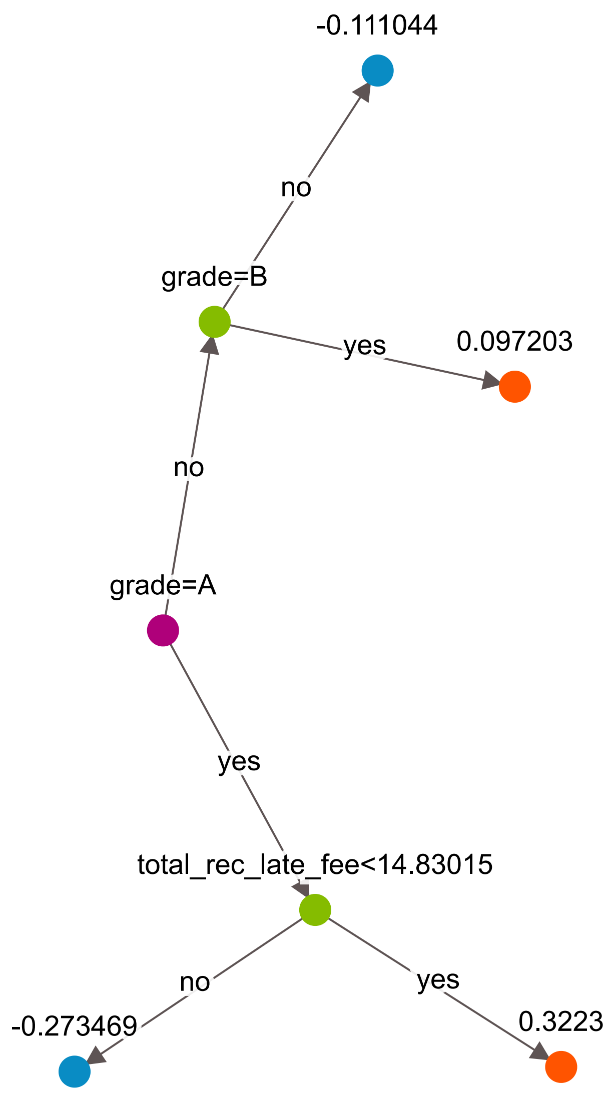
In the view that is provided by GraphLab Create, you can see each node, and each split at each node. This visualization is great for considering what happens when this model predicts the target of a new data point.

Note: To better understand this visual:

- The root node is represented using pink.
- Intermediate nodes are in green.
- Leaf nodes in blue and orange.

In [16]:

```
small_model.show(view="Tree")
```



Making predictions

Let's consider two positive and two negative examples **from the validation set** and see what the model predicts. We will do the following:

- Predict whether or not a loan is safe.
- Predict the probability that a loan is safe.

In [17]:

```
validation_safe_loans = validation_data[validation_data[target] == 1]
validation_risky_loans = validation_data[validation_data[target] == -1]

sample_validation_data_risky = validation_risky_loans[0:2]
sample_validation_data_safe = validation_safe_loans[0:2]

sample_validation_data = sample_validation_data_safe.append(sample_validation_data_risky)
sample_validation_data
```

Out[17]:

grade	sub_grade	short_emp	emp_length_num	home_ownership	d
B	B3	0	11	OWN	11.
D	D1	0	10	RENT	16.
D	D2	0	3	RENT	13.
A	A5	0	11	MORTGAGE	16.

last_major_derog_none	revol_util	total_rec_late_fee	safe_loans
1	82.4	0.0	1
1	96.4	0.0	1
1	59.5	0.0	-1
1	62.1	0.0	-1

[4 rows x 13 columns]

Explore label predictions

Now, we will use our model to predict whether or not a loan is likely to default. For each row in the **sample_validation_data**, use the **decision_tree_model** to predict whether or not the loan is classified as a **safe loan**.

Hint: Be sure to use the `.predict()` method.

In [18]:

```
decision_tree_model.predict(sample_validation_data)
```

Out[18]:

```
dtype: int
Rows: 4
[1L, -1L, -1L, 1L]
```

In [19]:

```
(sample_validation_data['safe_loans'] == decision_tree_model.predict(sample_validation_data
```

Out[19]:

```
0.5
```

Quiz Question: What percentage of the predictions on `sample_validation_data` did `decision_tree_model` get correct?

Explore probability predictions

For each row in the **sample_validation_data**, what is the probability (according **decision_tree_model**) of a loan being classified as **safe**?

Hint: Set `output_type='probability'` to make **probability** predictions using **decision_tree_model** on `sample_validation_data`:

In [20]:

```
decision_tree_model.predict(sample_validation_data, output_type='probability')
```

Out[20]:

```
dtype: float
Rows: 4
[0.5473502278327942, 0.4891221821308136, 0.4559234082698822, 0.5864479541778
564]
```

Quiz Question: Which loan has the highest probability of being classified as a **safe loan**?

Checkpoint: Can you verify that for all the predictions with `probability >= 0.5`, the model predicted the label **+1**?

Tricky predictions!

Now, we will explore something pretty interesting. For each row in the **sample_validation_data**, what is the probability (according to **small_model**) of a loan being classified as **safe**?

Hint: Set `output_type='probability'` to make **probability** predictions using **small_model** on `sample_validation_data`:

In [21]:

```
small_model.predict(sample_validation_data, output_type='probability')
```

Out[21]:

```
dtype: float
Rows: 4
[0.5242817997932434, 0.47226759791374207, 0.47226759791374207, 0.57988476753
23486]
```

Quiz Question: Notice that the probability preditions are the **exact same** for the 2nd and 3rd loans. Why would this happen?

Visualize the prediction on a tree

Note that you should be able to look at the small tree, traverse it yourself, and visualize the prediction being made. Consider the following point in the **sample_validation_data**

In [22]:

```
sample_validation_data[1]
```

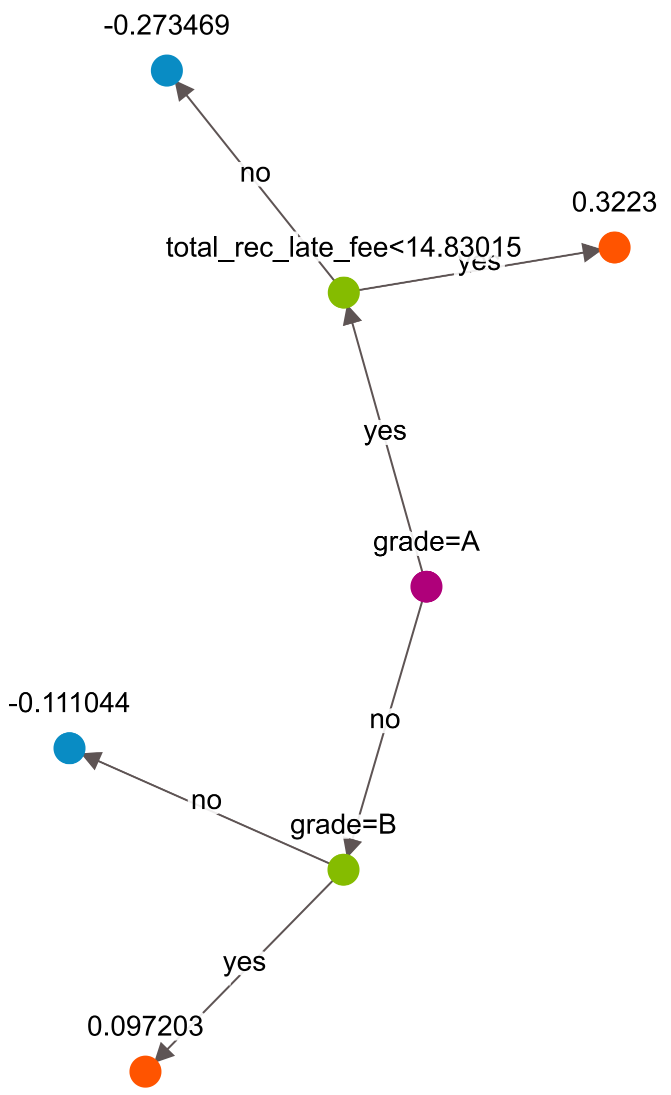
Out[22]:

```
{'dti': 16.85,
 'emp_length_num': 10L,
 'grade': 'D',
 'home_ownership': 'RENT',
 'last_delinq_none': 1L,
 'last_major_derog_none': 1L,
 'purpose': 'debt_consolidation',
 'revol_util': 96.4,
 'safe_loans': 1L,
 'short_emp': 0L,
 'sub_grade': 'D1',
 'term': ' 36 months',
 'total_rec_late_fee': 0.0}
```

Let's visualize the small tree here to do the traversing for this data point.

In [23]:

```
small_model.show(view="Tree")
```



Note: In the tree visualization above, the values at the leaf nodes are not class predictions but scores (a slightly advanced concept that is out of the scope of this course). You can read more about this here (https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf). If the score is $\geq$ 0, the class +1 is predicted. Otherwise, if the score < 0, we predict class -1.

Quiz Question: Based on the visualized tree, what prediction would you make for this data point?

Now, let's verify your prediction by examining the prediction made using GraphLab Create. Use the `.predict` function on `small_model`.

In [24]:

```
small_model.predict(sample_validation_data[1])
```

Out[24]:

```
dtype: int
Rows: 1
[-1L]
```

Evaluating accuracy of the decision tree model

Recall that the accuracy is defined as follows:

$$\text{accuracy} = \frac{\text{\# correctly classified examples}}{\text{\# total examples}}$$

Let us start by evaluating the accuracy of the `small_model` and `decision_tree_model` on the training data

In [25]:

```
print small_model.evaluate(train_data)['accuracy']
print decision_tree_model.evaluate(train_data)['accuracy']
```

```
0.613502041694
0.640581345369
```

Checkpoint: You should see that the **small_model** performs worse than the **decision_tree_model** on the training data.

Now, let us evaluate the accuracy of the **small_model** and **decision_tree_model** on the entire **validation_data**, not just the subsample considered above.

In [26]:

```
print small_model.evaluate(validation_data)['accuracy']
print decision_tree_model.evaluate(validation_data)['accuracy']
```

```
0.619345109866
0.636794485136
```

Quiz Question: What is the accuracy of `decision_tree_model` on the validation set, rounded to the nearest .01?

Evaluating accuracy of a complex decision tree model

Here, we will train a large decision tree with `max_depth=10`. This will allow the learned tree to become very deep, and result in a very complex model. Recall that in lecture, we prefer simpler models with similar predictive power. This will be an example of a more complicated model which has similar predictive power, i.e. something we don't want.

In [27]:

```
big_model = graphlab.decision_tree_classifier.create(train_data, validation_set=None,
                   target = target, features = features, max_depth = 10)
```

```
Decision tree classifier:

--------------------------------------------------------

Number of examples          : 37224

Number of classes           : 2

Number of feature columns   : 12

Number of unpacked features : 12

+-----------+--------------+-------------------+-------------------+

| Iteration | Elapsed Time | Training-accuracy | Training-log_loss |

+-----------+--------------+-------------------+-------------------+

| 1         | 0.137505     | 0.665538          | 0.652730          |

+-----------+--------------+-------------------+-------------------+
```

Now, let us evaluate **big_model** on the training set and validation set.

In [28]:

```
print big_model.evaluate(train_data)['accuracy']
print big_model.evaluate(validation_data)['accuracy']
```

```
0.665538362347
0.627423524343
```

Checkpoint: We should see that **big_model** has even better performance on the training set than **decision_tree_model** did on the training set.

Quiz Question: How does the performance of **big_model** on the validation set compare to **decision_tree_model** on the validation set? Is this a sign of overfitting?

Quantifying the cost of mistakes

Every mistake the model makes costs money. In this section, we will try and quantify the cost of each mistake made by the model.

Assume the following:

- **False negatives**: Loans that were actually safe but were predicted to be risky. This results in an oppurtunity cost of losing a loan that would have otherwise been accepted.
- **False positives**: Loans that were actually risky but were predicted to be safe. These are much more expensive because it results in a risky loan being given.
- **Correct predictions**: All correct predictions don't typically incur any cost.

Let's write code that can compute the cost of mistakes made by the model. Complete the following 4 steps:

1. First, let us compute the predictions made by the model.
2. Second, compute the number of false positives.
3. Third, compute the number of false negatives.
4. Finally, compute the cost of mistakes made by the model by adding up the costs of true positives and false positives.

First, let us make predictions on `validation_data` using the `decision_tree_model`:

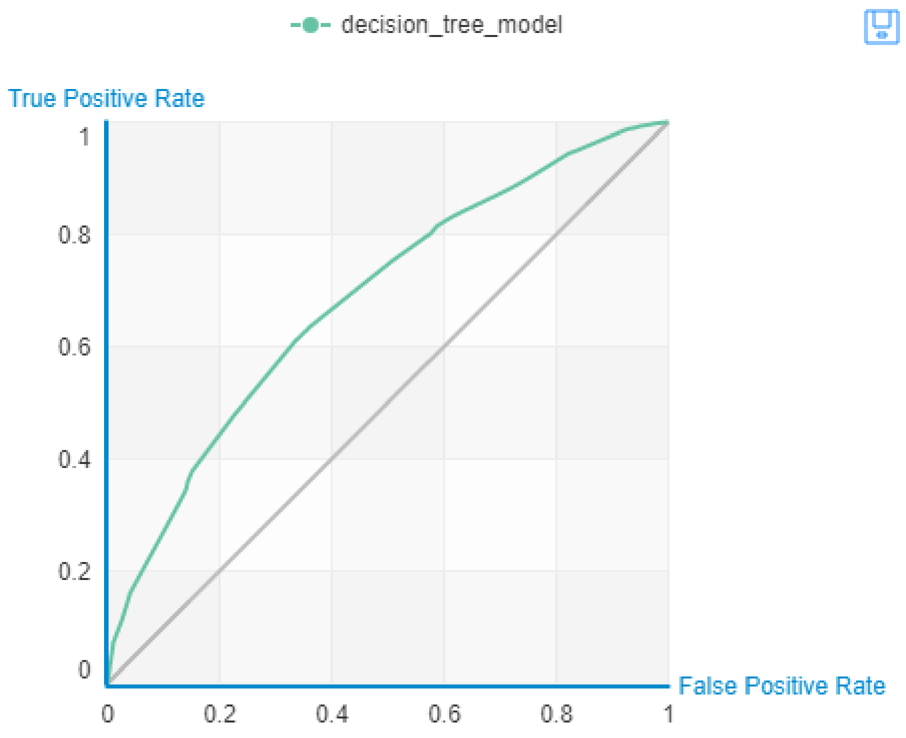
In [29]:

```
predictions = decision_tree_model.predict(validation_data)
```

In [30]:

```
decision_tree_model.show(view='Evaluation')
```

Most recent model evaluation with dataset *validation_data*



True Positive	False Negative	Accuracy	Precision
2895	**1715**	**0.637**	**0.636**
False Positive	True Negative	Recall	F1 Score
1655	**3019**	**0.628**	**0.632**

Threshold
0.501

General Metrics	
f1_score	0.632
auc	0.684
recall	0.628
precision	0.636
log_loss	0.665
accuracy	0.637

False positives are predictions where the model predicts +1 but the true label is -1. Complete the following code block for the number of false positives:

In [31]:

```
false_positives = (validation_data[validation_data['safe_loans'] != predictions]['safe_loar
print false_positives
```

```
1656
```

False negatives are predictions where the model predicts -1 but the true label is +1. Complete the following code block for the number of false negatives:

In [32]:

```
false_negatives = (validation_data[validation_data['safe_loans'] != predictions]['safe_loar
print false_negatives
```

```
1716
```

Quiz Question: Let us assume that each mistake costs money:

- Assume a cost of \$10,000 per false negative.
- Assume a cost of \$20,000 per false positive.

What is the total cost of mistakes made by `decision_tree_model` on `validation_data`?

In [33]:

```
cost_of_mistakes = (false_negatives * 10000) + (false_positives * 20000)
print cost_of_mistakes
```

```
50280000
```

Quiz

1. What percentage of the predictions on sample_validation_data did decision_tree_model get correct?

   - 25%
   - [✓ 50%]
   - 75%
   - 100%

2. Which loan has the highest probability of being classified as a safe loan?

   - First
   - Second
   - Third
   - [✓ Fourth]

3. Notice that the probability preditions are the exact same for the 2nd and 3rd loans. Why would this happen?

   - [✓ During tree traversal both examples fall into the same leaf node.]
   - This can only happen with sheer coincidence.

4. Based on the visualized tree, what prediction would you make for this data point?

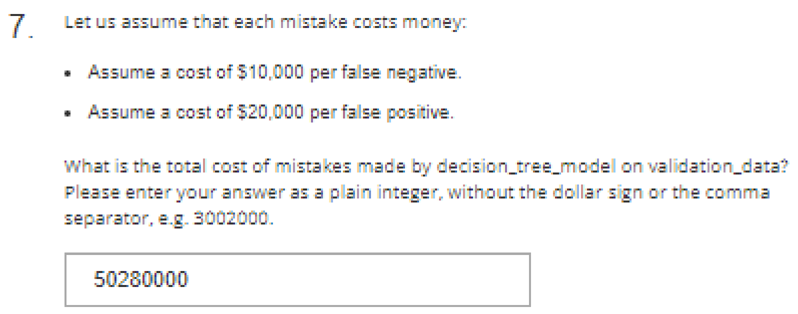
   - +1
   - [✓ -1]

5. What is the accuracy of decision_tree_model on the validation set, rounded to the nearest .01 (e.g. 0.76)?

   0.64

6. How does the performance of big_model on the validation set compare to decision_tree_model on the validation set? Is this a sign of overfitting?

   - big_model has higher accuracy on the validation set than decision_tree_model. This is overfitting.
   - big_model has higher accuracy on the validation set than decision_tree_model. This is not overfitting.
   - [✓ big_model has lower accuracy on the validation set than decision_tree_model. This is overfitting.]
   - big_model has lower accuracy on the validation set than decision_tree_model. This is not overfitting.



7. Let us assume that each mistake costs money:

- Assume a cost of $10,000 per false negative.
- Assume a cost of $20,000 per false positive.

What is the total cost of mistakes made by decision_tree_model on validation_data? Please enter your answer as a plain integer, without the dollar sign or the comma separator, e.g. 3002000.

50280000