Implementing binary decision trees

The goal of this notebook is to implement your own binary decision tree classifier. You will:

- Use SFrames to do some feature engineering.
- Transform categorical variables into binary variables.
- Write a function to compute the number of misclassified examples in an intermediate node.
- Write a function to find the best feature to split on.
- Build a binary decision tree from scratch.
- Make predictions using the decision tree.
- Evaluate the accuracy of the decision tree.
- Visualize the decision at the root node.

Important Note: In this assignment, we will focus on building decision trees where the data contain **only binary (0 or 1) features**. This allows us to avoid dealing with:

- Multiple intermediate nodes in a split
- The thresholding issues of real-valued features.

This assignment **may be challenging**, so brace yourself :)

Fire up GraphLab Create

Make sure you have the latest version of GraphLab Create.

In [1]:

```
import graphlab
```

Load the lending club dataset

We will be using the same LendingClub (https://www.lendingclub.com/) dataset as in the previous assignment.

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_1532764147.log.0
```

Like the previous assignment, we reassign the labels to have +1 for a safe loan, and -1 for a risky (bad) loan.

In [3]:

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

Unlike the previous assignment where we used several features, in this assignment, we will just be using 4 categorical features:

1. grade of the loan
2. the length of the loan term
3. the home ownership status: own, mortgage, rent
4. number of years of employment.

Since we are building a binary decision tree, we will have to convert these categorical features to a binary representation in a subsequent section using 1-hot encoding.

In [4]:

```
features = ['grade',              # grade of the loan
            'term',               # the term of the loan
            'home_ownership',     # home_ownership status: own, mortgage or rent
            'emp_length',         # number of years of employment
           ]
target = 'safe_loans'
loans = loans[features + [target]]
```

Let's explore what the dataset looks like.

In [5]:

```
loans
```

Out[5]:

grade	term	home_ownership	emp_length	safe_loans
B	36 months	RENT	10+ years	1
C	60 months	RENT	< 1 year	-1
C	36 months	RENT	10+ years	1
C	36 months	RENT	10+ years	1
A	36 months	RENT	3 years	1
E	36 months	RENT	9 years	1
F	60 months	OWN	4 years	-1
B	60 months	RENT	< 1 year	-1
C	60 months	OWN	5 years	1
B	36 months	OWN	10+ years	1

[122607 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

Subsample dataset to make sure classes are balanced

Just as we did in the previous assignment, 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 use `seed=1` so everyone gets the same results.

In [6]:

```
safe_loans_raw = loans[loans[target] == 1]
risky_loans_raw = loans[loans[target] == -1]

# Since there are less 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))
safe_loans = safe_loans_raw.sample(percentage, seed = 1)
risky_loans = risky_loans_raw
loans_data = risky_loans.append(safe_loans)

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 algorithm. These approaches are beyond the scope of this course, but some of them are reviewed in "Learning from Imbalanced Data (http://www.ele.uri.edu/faculty/he/PDFfiles/ImbalancedLearning.pdf)" by Haibo He and Edwardo A. Garcia, *IEEE Transactions on Knowledge and Data Engineering* **21**(9) (June 26, 2009), p. 1263–1284. For this assignment, we use the simplest possible approach, where we subsample the overly represented class to get a more balanced dataset. In general, and especially when the data is highly imbalanced, we recommend using more advanced methods.

Transform categorical data into binary features

In this assignment, we will implement **binary decision trees** (decision trees for binary features, a specific case of categorical variables taking on two values, e.g., true/false). Since all of our features are currently categorical features, we want to turn them into binary features.

For instance, the **home_ownership** feature represents the home ownership status of the loanee, which is either `own`, `mortgage` or `rent`. For example, if a data point has the feature

```
{'home_ownership': 'RENT'}
```

we want to turn this into three features:

```
{
  'home_ownership = OWN'      : 0,
  'home_ownership = MORTGAGE' : 0,
  'home_ownership = RENT'     : 1
}
```

Since this code requires a few Python and GraphLab tricks, feel free to use this block of code as is. Refer to the API documentation for a deeper understanding.

In [7]:

```
loans_data = risky_loans.append(safe_loans)
for feature in features:
    loans_data_one_hot_encoded = loans_data[feature].apply(lambda x: {x: 1})
    loans_data_unpacked = loans_data_one_hot_encoded.unpack(column_name_prefix=feature)

    # Change None's to 0's
    for column in loans_data_unpacked.column_names():
        loans_data_unpacked[column] = loans_data_unpacked[column].fillna(0)

    loans_data.remove_column(feature)
    loans_data.add_columns(loans_data_unpacked)
```

Let's see what the feature columns look like now:

In [8]:

```
features = loans_data.column_names()
features.remove('safe_loans')  # Remove the response variable
features
```

Out[8]:

```
['grade.A',
 'grade.B',
 'grade.C',
 'grade.D',
 'grade.E',
 'grade.F',
 'grade.G',
 'term. 36 months',
 'term. 60 months',
 'home_ownership.MORTGAGE',
 'home_ownership.OTHER',
 'home_ownership.OWN',
 'home_ownership.RENT',
 'emp_length.1 year',
 'emp_length.10+ years',
 'emp_length.2 years',
 'emp_length.3 years',
 'emp_length.4 years',
 'emp_length.5 years',
 'emp_length.6 years',
 'emp_length.7 years',
 'emp_length.8 years',
 'emp_length.9 years',
 'emp_length.< 1 year',
 'emp_length.n/a']
```

In [9]:

```
print "Number of features (after binarizing categorical variables) = %s" % len(features)
```

```
Number of features (after binarizing categorical variables) = 25
```

Let's explore what one of these columns looks like:

In [10]:

```
loans_data['grade.A']
```

Out[10]:

```
dtype: int
Rows: 46508
[0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 1L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L,
0L, 0L, 0L, 0L, 1L, 0L, 0L, 0L, 0L, 1L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L,
0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 1L, 0L, 0L, 1L, 0L, 0L, 0L, 0L, 0L, 0L, 0L,
0L, 0L, 1L, 0L, 0L, 1L, 0L, 1L, 0L, 0L, 1L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L,
0L, 1L, 0L, 1L, 0L, 0L, 0L, 0L, 0L, 0L, 1L, 0L, 0L, 0L, 0L, 0L, 0L, 1L, 1L,
0L, 0L, 0L, 1L, 0L, ... ]
```

This column is set to 1 if the loan grade is A and 0 otherwise.

Checkpoint: Make sure the following answers match up.

In [11]:

```
print "Total number of grade.A loans : %s" % loans_data['grade.A'].sum()
print "Expexted answer               : 6422"
```

```
Total number of grade.A loans : 6422
Expexted answer               : 6422
```

Train-test split

We split the data into a train test split with 80% of the data in the training set and 20% of the data in the test set. We use `seed=1` so that everyone gets the same result.

In [12]:

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

Decision tree implementation

In this section, we will implement binary decision trees from scratch. There are several steps involved in building a decision tree. For that reason, we have split the entire assignment into several sections.

Function to count number of mistakes while predicting majority class

Recall from the lecture that prediction at an intermediate node works by predicting the **majority class** for all data points that belong to this node.

Now, we will write a function that calculates the number of **missclassified examples** when predicting the **majority class**. This will be used to help determine which feature is the best to split on at a given node of the tree.

Note: Keep in mind that in order to compute the number of mistakes for a majority classifier, we only need the label (y values) of the data points in the node.

Steps to follow :

- **Step 1:** Calculate the number of safe loans and risky loans.
- **Step 2:** Since we are assuming majority class prediction, all the data points that are **not** in the majority class are considered **mistakes**.
- **Step 3:** Return the number of **mistakes**.

Now, let us write the function `intermediate_node_num_mistakes` which computes the number of misclassified examples of an intermediate node given the set of labels (y values) of the data points contained in the node. Fill in the places where you find `## YOUR CODE HERE`. There are **three** places in this function for you to fill in.

In [13]:

```
def intermediate_node_num_mistakes(labels_in_node):
    # Corner case: If labels_in_node is empty, return 0
    if len(labels_in_node) == 0:
        return 0

    # Count the number of 1's (safe loans)
    ## YOUR CODE HERE
    num_of_positive = (labels_in_node == +1).sum()

    # Count the number of -1's (risky loans)
    ## YOUR CODE HERE
    num_of_negative = (labels_in_node == -1).sum()

    # Return the number of mistakes that the majority classifier makes.
    ## YOUR CODE HERE
    return num_of_negative if num_of_positive > num_of_negative else num_of_positive
```

Because there are several steps in this assignment, we have introduced some stopping points where you can check your code and make sure it is correct before proceeding. To test your `intermediate_node_num_mistakes` function, run the following code until you get a **Test passed!**, then you should proceed. Otherwise, you should spend some time figuring out where things went wrong.

In [14]:

```
# Test case 1
example_labels = graphlab.SArray([-1, -1, 1, 1, 1])
if intermediate_node_num_mistakes(example_labels) == 2:
    print 'Test passed!'
else:
    print 'Test 1 failed... try again!'

# Test case 2
example_labels = graphlab.SArray([-1, -1, 1, 1, 1, 1, 1])
if intermediate_node_num_mistakes(example_labels) == 2:
    print 'Test passed!'
else:
    print 'Test 2 failed... try again!'

# Test case 3
example_labels = graphlab.SArray([-1, -1, -1, -1, -1, 1, 1])
if intermediate_node_num_mistakes(example_labels) == 2:
    print 'Test passed!'
else:
    print 'Test 3 failed... try again!'
```

```
Test passed!
Test passed!
Test passed!
```

Function to pick best feature to split on

The function **best_splitting_feature** takes 3 arguments:

1. The data (SFrame of data which includes all of the feature columns and label column)
2. The features to consider for splits (a list of strings of column names to consider for splits)
3. The name of the target/label column (string)

The function will loop through the list of possible features, and consider splitting on each of them. It will calculate the classification error of each split and return the feature that had the smallest classification error when split on.

Recall that the **classification error** is defined as follows:

$$\text{classification error} = \frac{\text{\# mistakes}}{\text{\# total examples}}$$

Follow these steps:

- **Step 1:** Loop over each feature in the feature list
- **Step 2:** Within the loop, split the data into two groups: one group where all of the data has feature value 0 or False (we will call this the **left** split), and one group where all of the data has feature value 1 or True (we will call this the **right** split). Make sure the **left** split corresponds with 0 and the **right** split corresponds with 1 to ensure your implementation fits with our implementation of the tree building process.
- **Step 3:** Calculate the number of misclassified examples in both groups of data and use the above formula to compute the **classification error**.
- **Step 4:** If the computed error is smaller than the best error found so far, store this **feature and its error**.

This may seem like a lot, but we have provided pseudocode in the comments in order to help you implement the function correctly.

Note: Remember that since we are only dealing with binary features, we do not have to consider thresholds for real-valued features. This makes the implementation of this function much easier.

Fill in the places where you find `## YOUR CODE HERE`. There are **five** places in this function for you to fill in.

In [15]:

```
def best_splitting_feature(data, features, target):

    best_feature = None # Keep track of the best feature
    best_error = 10     # Keep track of the best error so far
    # Note: Since error is always <= 1, we should intialize it with something larger than 1

    # Convert to float to make sure error gets computed correctly.
    num_data_points = float(len(data))

    # Loop through each feature to consider splitting on that feature
    for feature in features:

        # The left split will have all data points where the feature value is 0
        left_split = data[data[feature] == 0]

        # The right split will have all data points where the feature value is 1
        ## YOUR CODE HERE
        right_split =  data[data[feature] == 1]

        # Calculate the number of misclassified examples in the left split.
        # Remember that we implemented a function for this! (It was called intermediate_noa
        # YOUR CODE HERE
        left_mistakes = intermediate_node_num_mistakes(left_split[target])

        # Calculate the number of misclassified examples in the right split.
        ## YOUR CODE HERE
        right_mistakes = intermediate_node_num_mistakes(right_split[target])

        # Compute the classification error of this split.
        # Error = (# of mistakes (left) + # of mistakes (right)) / (# of data points)
        ## YOUR CODE HERE
        error = (left_mistakes + right_mistakes) / num_data_points

        # If this is the best error we have found so far, store the feature as best_feature
        ## YOUR CODE HERE
        if error < best_error:
            best_feature = feature
            best_error = error

    return best_feature # Return the best feature we found
```

To test your `best_splitting_feature` function, run the following code:

In [16]:

```
if best_splitting_feature(train_data, features, 'safe_loans') == 'term. 36 months':
    print 'Test passed!'
else:
    print 'Test failed... try again!'
```

```
Test passed!
```

Building the tree

With the above functions implemented correctly, we are now ready to build our decision tree. Each node in the decision tree is represented as a dictionary which contains the following keys and possible values:

```
{
   'is_leaf'            : True/False.
   'prediction'         : Prediction at the leaf node.
   'left'               : (dictionary corresponding to the left tree).
   'right'              : (dictionary corresponding to the right tree).
   'splitting_feature'  : The feature that this node splits on.
}
```

First, we will write a function that creates a leaf node given a set of target values. Fill in the places where you find `## YOUR CODE HERE`. There are **three** places in this function for you to fill in.

In [19]:

```
def create_leaf(target_values):

    # Create a leaf node
    leaf = {'splitting_feature' : None,
            'left' : None,
            'right' : None,
            'is_leaf':  True   }   ## YOUR CODE HERE

    # Count the number of data points that are +1 and -1 in this node.
    num_ones = len(target_values[target_values == +1])
    num_minus_ones = len(target_values[target_values == -1])

    # For the leaf node, set the prediction to be the majority class.
    # Store the predicted class (1 or -1) in leaf['prediction']
    if num_ones > num_minus_ones:
        leaf['prediction'] = +1         ## YOUR CODE HERE
    else:
        leaf['prediction'] = -1         ## YOUR CODE HERE

    # Return the leaf node
    return leaf
```

We have provided a function that learns the decision tree recursively and implements 3 stopping conditions:

1. **Stopping condition 1:** All data points in a node are from the same class.
2. **Stopping condition 2:** No more features to split on.
3. **Additional stopping condition:** In addition to the above two stopping conditions covered in lecture, in this assignment we will also consider a stopping condition based on the **max_depth** of the tree. By not letting the tree grow too deep, we will save computational effort in the learning process.

Now, we will write down the skeleton of the learning algorithm. Fill in the places where you find `## YOUR CODE HERE`. There are **seven** places in this function for you to fill in.

In [20]:

```
def decision_tree_create(data, features, target, current_depth = 0, max_depth = 10):
    remaining_features = features[:] # Make a copy of the features.

    target_values = data[target]
    print "--------------------------------------------------------------------"
    print "Subtree, depth = %s (%s data points)." % (current_depth, len(target_values))


    # Stopping condition 1
    # (Check if there are mistakes at current node.
    # Recall you wrote a function intermediate_node_num_mistakes to compute this.)
    if intermediate_node_num_mistakes(target_values)  == 0:  ## YOUR CODE HERE
        print "Stopping condition 1 reached."
        # If not mistakes at current node, make current node a leaf node
        return create_leaf(target_values)

    # Stopping condition 2 (check if there are remaining features to consider splitting on)
    if remaining_features == []:   ## YOUR CODE HERE
        print "Stopping condition 2 reached."
        # If there are no remaining features to consider, make current node a leaf node
        return create_leaf(target_values)

    # Additional stopping condition (limit tree depth)
    if current_depth >= max_depth:  ## YOUR CODE HERE
        print "Reached maximum depth. Stopping for now."
        # If the max tree depth has been reached, make current node a leaf node
        return create_leaf(target_values)

    # Find the best splitting feature (recall the function best_splitting_feature implement
    ## YOUR CODE HERE
    splitting_feature = best_splitting_feature(data, features, target)

    # Split on the best feature that we found.
    left_split = data[data[splitting_feature] == 0]
    right_split = data[data[splitting_feature] == 1]    ## YOUR CODE HERE
    remaining_features.remove(splitting_feature)
    print "Split on feature %s. (%s, %s)" % (\
                      splitting_feature, len(left_split), len(right_split))

    # Create a leaf node if the split is "perfect"
    if len(left_split) == len(data):
        print "Creating leaf node."
        return create_leaf(left_split[target])
    if len(right_split) == len(data):
        print "Creating leaf node."
        ## YOUR CODE HERE
        return create_leaf(right_split[target])

    # Repeat (recurse) on left and right subtrees
    left_tree = decision_tree_create(left_split, remaining_features, target, current_depth
    ## YOUR CODE HERE
    right_tree = decision_tree_create(right_split, remaining_features, target, current_dept

    return {'is_leaf'          : False,
            'prediction'       : None,
            'splitting_feature': splitting_feature,
            'left'             : left_tree,
            'right'            : right_tree}
```

Here is a recursive function to count the nodes in your tree:

In [21]:

```
def count_nodes(tree):
    if tree['is_leaf']:
        return 1
    return 1 + count_nodes(tree['left']) + count_nodes(tree['right'])
```

Run the following test code to check your implementation. Make sure you get **'Test passed'** before proceeding.

In [22]:

```
small_data_decision_tree = decision_tree_create(train_data, features, 'safe_loans', max_dep
if count_nodes(small_data_decision_tree) == 13:
    print 'Test passed!'
else:
    print 'Test failed... try again!'
    print 'Number of nodes found                :', count_nodes(small_data_decision_tree)
    print 'Number of nodes that should be there : 13'
```

```
--------------------------------------------------------------------
Subtree, depth = 0 (37224 data points).
Split on feature term. 36 months. (9223, 28001)
--------------------------------------------------------------------
Subtree, depth = 1 (9223 data points).
Split on feature grade.A. (9122, 101)
--------------------------------------------------------------------
Subtree, depth = 2 (9122 data points).
Split on feature grade.B. (8074, 1048)
--------------------------------------------------------------------
Subtree, depth = 3 (8074 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (1048 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (101 data points).
Split on feature emp_length.n/a. (96, 5)
--------------------------------------------------------------------
Subtree, depth = 3 (96 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (5 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 1 (28001 data points).
Split on feature grade.D. (23300, 4701)
--------------------------------------------------------------------
Subtree, depth = 2 (23300 data points).
Split on feature grade.E. (22024, 1276)
--------------------------------------------------------------------
Subtree, depth = 3 (22024 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (1276 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (4701 data points).
Split on feature grade.A. (4701, 0)
Creating leaf node.
Test passed!
```

Build the tree!

Now that all the tests are passing, we will train a tree model on the **train_data**. Limit the depth to 6 (**max_depth = 6**) to make sure the algorithm doesn't run for too long. Call this tree **my_decision_tree**.

Warning: This code block may take 1-2 minutes to learn.

In [23]:

```
# Make sure to cap the depth at 6 by using max_depth = 6
my_decision_tree = decision_tree_create(train_data, features, 'safe_loans', max_depth = 6)
```

```
--------------------------------------------------------------------
Subtree, depth = 0 (37224 data points).
Split on feature term. 36 months. (9223, 28001)
--------------------------------------------------------------------
Subtree, depth = 1 (9223 data points).
Split on feature grade.A. (9122, 101)
--------------------------------------------------------------------
Subtree, depth = 2 (9122 data points).
Split on feature grade.B. (8074, 1048)
--------------------------------------------------------------------
Subtree, depth = 3 (8074 data points).
Split on feature grade.C. (5884, 2190)
--------------------------------------------------------------------
Subtree, depth = 4 (5884 data points).
Split on feature grade.D. (3826, 2058)
--------------------------------------------------------------------
Subtree, depth = 5 (3826 data points).
Split on feature grade.E. (1693, 2133)
--------------------------------------------------------------------
Subtree, depth = 6 (1693 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (2133 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 5 (2058 data points).
Split on feature grade.E. (2058, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 4 (2190 data points).
Split on feature grade.D. (2190, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 3 (1048 data points).
Split on feature emp_length.5 years. (969, 79)
--------------------------------------------------------------------
Subtree, depth = 4 (969 data points).
Split on feature grade.C. (969, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 4 (79 data points).
Split on feature home_ownership.MORTGAGE. (34, 45)
--------------------------------------------------------------------
Subtree, depth = 5 (34 data points).
Split on feature grade.C. (34, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 5 (45 data points).
Split on feature grade.C. (45, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 2 (101 data points).
Split on feature emp_length.n/a. (96, 5)
--------------------------------------------------------------------
Subtree, depth = 3 (96 data points).
Split on feature emp_length.< 1 year. (85, 11)
```

```
--------------------------------------------------------------------
Subtree, depth = 4 (85 data points).
Split on feature grade.B. (85, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 4 (11 data points).
Split on feature grade.B. (11, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 3 (5 data points).
Split on feature grade.B. (5, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 1 (28001 data points).
Split on feature grade.D. (23300, 4701)
--------------------------------------------------------------------
Subtree, depth = 2 (23300 data points).
Split on feature grade.E. (22024, 1276)
--------------------------------------------------------------------
Subtree, depth = 3 (22024 data points).
Split on feature grade.F. (21666, 358)
--------------------------------------------------------------------
Subtree, depth = 4 (21666 data points).
Split on feature emp_length.n/a. (20734, 932)
--------------------------------------------------------------------
Subtree, depth = 5 (20734 data points).
Split on feature grade.G. (20638, 96)
--------------------------------------------------------------------
Subtree, depth = 6 (20638 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (96 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 5 (932 data points).
Split on feature grade.A. (702, 230)
--------------------------------------------------------------------
Subtree, depth = 6 (702 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (230 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 4 (358 data points).
Split on feature emp_length.8 years. (347, 11)
--------------------------------------------------------------------
Subtree, depth = 5 (347 data points).
Split on feature grade.A. (347, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 5 (11 data points).
Split on feature home_ownership.OWN. (9, 2)
--------------------------------------------------------------------
Subtree, depth = 6 (9 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (2 data points).
Stopping condition 1 reached.
--------------------------------------------------------------------
Subtree, depth = 3 (1276 data points).
Split on feature grade.A. (1276, 0)
```

```
Creating leaf node.
---------------------------------------------------------------------
Subtree, depth = 2 (4701 data points).
Split on feature grade.A. (4701, 0)
Creating leaf node.
```

Making predictions with a decision tree

As discussed in the lecture, we can make predictions from the decision tree with a simple recursive function. Below, we call this function `classify`, which takes in a learned `tree` and a test point `x` to classify. We include an option `annotate` that describes the prediction path when set to `True`.

Fill in the places where you find `## YOUR CODE HERE`. There is **one** place in this function for you to fill in.

In [24]:

```
def classify(tree, x, annotate = False):
    # if the node is a leaf node.
    if tree['is_leaf']:
        if annotate:
            print "At leaf, predicting %s" % tree['prediction']
        return tree['prediction']
    else:
        # split on feature.
        split_feature_value = x[tree['splitting_feature']]
        if annotate:
            print "Split on %s = %s" % (tree['splitting_feature'], split_feature_value)
        if split_feature_value == 0:
            return classify(tree['left'], x, annotate)
        else:
            return classify(tree['right'], x, annotate)   ### YOUR CODE HERE
```

Now, let's consider the first example of the test set and see what `my_decision_tree` model predicts for this data point.

In [25]:

```
test_data[0]
```

Out[25]:

```
{'emp_length.1 year': 0L,
 'emp_length.10+ years': 0L,
 'emp_length.2 years': 1L,
 'emp_length.3 years': 0L,
 'emp_length.4 years': 0L,
 'emp_length.5 years': 0L,
 'emp_length.6 years': 0L,
 'emp_length.7 years': 0L,
 'emp_length.8 years': 0L,
 'emp_length.9 years': 0L,
 'emp_length.< 1 year': 0L,
 'emp_length.n/a': 0L,
 'grade.A': 0L,
 'grade.B': 0L,
 'grade.C': 0L,
 'grade.D': 1L,
 'grade.E': 0L,
 'grade.F': 0L,
 'grade.G': 0L,
 'home_ownership.MORTGAGE': 0L,
 'home_ownership.OTHER': 0L,
 'home_ownership.OWN': 0L,
 'home_ownership.RENT': 1L,
 'safe_loans': -1L,
 'term. 36 months': 0L,
 'term. 60 months': 1L}
```

In [26]:

```
print 'Predicted class: %s ' % classify(my_decision_tree, test_data[0])
```

```
Predicted class: -1
```

Let's add some annotations to our prediction to see what the prediction path was that lead to this predicted class:

In [27]:

```
classify(my_decision_tree, test_data[0], annotate=True)
```

```
Split on term. 36 months = 0
Split on grade.A = 0
Split on grade.B = 0
Split on grade.C = 0
Split on grade.D = 1
At leaf, predicting -1
```

Out[27]:

```
-1
```

Quiz Question: What was the feature that **my_decision_tree** first split on while making the prediction for test_data[0]?

Quiz Question: What was the first feature that lead to a right split of test_data[0]?

Quiz Question: What was the last feature split on before reaching a leaf node for test_data[0]?

Evaluating your decision tree

Now, we will write a function to evaluate a decision tree by computing the classification error of the tree on the given dataset.

Again, recall that the **classification error** is defined as follows:

$$\text{classification error} = \frac{\text{\# mistakes}}{\text{\# total examples}}$$

Now, write a function called `evaluate_classification_error` that takes in as input:

1. `tree` (as described above)
2. `data` (an SFrame)
3. `target` (a string - the name of the target/label column)

This function should calculate a prediction (class label) for each row in `data` using the decision `tree` and return the classification error computed using the above formula. Fill in the places where you find `## YOUR CODE HERE`. There is **one** place in this function for you to fill in.

In [28]:

```
def evaluate_classification_error(tree, data, target):
    # Apply the classify(tree, x) to each row in your data
    prediction = data.apply(lambda x: classify(tree, x))

    # Once you've made the predictions, calculate the classification error and return it
    ## YOUR CODE HERE
    num_of_mistakes = (prediction != data[target]).sum()/float(len(data))
    return num_of_mistakes
```

Now, let's use this function to evaluate the classification error on the test set.

In [29]:

```
evaluate_classification_error(my_decision_tree, test_data, target)
```

Out[29]:

```
0.3837785437311504
```

Quiz Question: Rounded to 2nd decimal point, what is the classification error of **my_decision_tree** on the **test_data**?

Printing out a decision stump

As discussed in the lecture, we can print out a single decision stump (printing out the entire tree is left as an exercise to the curious reader).

In [30]:

```
def print_stump(tree, name = 'root'):
    split_name = tree['splitting_feature'] # split_name is something like 'term. 36 months'
    if split_name is None:
        print "(leaf, label: %s)" % tree['prediction']
        return None
    split_feature, split_value = split_name.split('.')
    print '                       %s' % name
    print '         |---------------|----------------|'
    print '         |                                |'
    print '         |                                |'
    print '         |                                |'
    print '  [{0} == 0]               [{0} == 1]    '.format(split_name)
    print '         |                                |'
    print '         |                                |'
    print '         |                                |'
    print '    (%s)                         (%s)' \
        % (('leaf, label: ' + str(tree['left']['prediction']) if tree['left']['is_leaf'] el
           ('leaf, label: ' + str(tree['right']['prediction']) if tree['right']['is_leaf']
```

In [31]:

```
print_stump(my_decision_tree)
```

```
                      root
         |---------------|----------------|
         |                                |
         |                                |
         |                                |
  [term. 36 months == 0]               [term. 36 months == 1]
         |                                |
         |                                |
         |                                |
    (subtree)                        (subtree)
```

Quiz Question: What is the feature that is used for the split at the root node?

Exploring the intermediate left subtree

The tree is a recursive dictionary, so we do have access to all the nodes! We can use

- `my_decision_tree['left']` to go left
- `my_decision_tree['right']` to go right

In [34]:

```
print_stump(my_decision_tree['left'], my_decision_tree['splitting_feature'])
```

```
                      term. 36 months
         |---------------|----------------|
         |                                |
         |                                |
         |                                |
  [grade.A == 0]                 [grade.A == 1]
         |                                |
         |                                |
         |                                |
    (subtree)                        (subtree)
```

Exploring the left subtree of the left subtree

In [35]:

```
print_stump(my_decision_tree['left']['left'], my_decision_tree['left']['splitting_feature'])
```

```
                       grade.A
         |---------------|----------------|
         |                                |
         |                                |
         |                                |
  [grade.B == 0]                  [grade.B == 1]
         |                                |
         |                                |
         |                                |
    (subtree)                        (subtree)
```

Quiz Question: What is the path of the **first 3 feature splits** considered along the **left-most** branch of **my_decision_tree**?

Quiz Question: What is the path of the **first 3 feature splits** considered along the **right-most** branch of **my_decision_tree**?

In [36]:

```
print_stump(my_decision_tree['right'], my_decision_tree['splitting_feature'])
```

```
                       term. 36 months
         |---------------|----------------|
         |                                |
         |                                |
         |                                |
  [grade.D == 0]                  [grade.D == 1]
         |                                |
         |                                |
         |                                |
    (subtree)                        (leaf, label: -1)
```

Quiz

1. What was the feature that my_decision_tree first split on while making the prediction for test_data[0]?

- ( ) emp_length.4 years
- ( ) grade.A
- (•) term. 36 months
- ( ) home_ownership.MORTGAGE

---

2. What was the first feature that lead to a right split of test_data[0]?

- ( ) emp_length.< 1 year
- ( ) emp_length.10+ years
- ( ) grade.B
- (•) grade.D

---

3. What was the last feature split on before reaching a leaf node for test_data[0]?

- (•) grade.D
- ( ) grade.B
- ( ) term. 36 months
- ( ) grade.A

---

4. Rounded to 2nd decimal point (e.g. 0.76), what is the classification error of my_decision_tree on the test_data?

0.38

5. What is the feature that is used for the split at the root node?

- ○ grade.A
- ◉ term. 36 months
- ○ term. 60 months
- ○ home_ownership.OWN

6. What is the path of the first 3 feature splits considered along the left-most branch of my_decision_tree?

- ◉ term. 36 months, grade.A, grade.B
- ○ term. 36 months, grade.A, emp_length.4 years
- ○ term. 36 months, grade.A, no third feature because second split resulted in leaf

7. What is the path of the first 3 feature splits considered along the right-most branch of my_decision_tree?

- ○ term. 36 months, grade.D, grade.B
- ○ term. 36 months, grade.D, home_ownership.OWN
- ◉ term. 36 months, grade.D, no third feature because second split resulted in leaf

In [ ]: