Implementing binary decision trees

Note: This notebook was modified to support [pandas (https://pandas.pydata.org/)](https://pandas.pydata.org/) (as a replacement for graphlab) by [Faris Hijazi (https://github.com/FarisHijazi)](https://github.com/FarisHijazi).

Work distribution

Member	Contrubution (%)
Faris Hijazi	50
Abdullah AlSlamah	50
Rayan Gadhi	0

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 :)

In [1]:
```
import pandas as pd
```

Load the lending club dataset

We will be using the [LendingClub (https://www.lendingclub.com/)](https://www.lendingclub.com/) dataset for this assignment.

In [2]:
```
loans = pd.read_csv('lending-club-data.csv', error_bad_lines=True)
```

```
C:\Users\faris\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py:3
057: DtypeWarning: Columns (19,47) have mixed types. Specify dtype option on
import or set low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)
```

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.drop(['bad_loans'], axis=1)
```

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 re
nt
            '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.columns
```

Out[5]:
```
Index(['grade', 'term', 'home_ownership', 'emp_length', 'safe_loans'], dtype
='object')
```

In [6]:
```
loans.head()
```

Out[6]:

	grade	term	home_ownership	emp_length	safe_loans
0	B	36 months	RENT	10+ years	1
1	C	60 months	RENT	< 1 year	-1
2	C	36 months	RENT	10+ years	1
3	C	36 months	RENT	10+ years	1
4	A	36 months	RENT	3 years	1

Subsample dataset to make sure classes are balanced

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 [7]:
```
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 size
s
# 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(frac=percentage, random_state=1)
risky_loans = risky_loans_raw
loans_data = risky_loans.append(safe_loans)

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

```
Percentage of safe loans                 : 0.5
Percentage of risky loans                : 0.5
Total number of loans in our new dataset : 46300
```

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)](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
}
```

In [8]:
```
loans_data = risky_loans.append(safe_loans)
```

Using pandas dataframe, to one-hot-encode, here we are using get_dummies() (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html).

In [9]:
```
#one-hot-encoding

loans_data = pd.get_dummies(loans_data, columns=features, prefix_sep='.', dummy_na=False).fillna(0)
loans_data.head()
```

Out[9]:

	safe_loans	grade.A	grade.B	grade.C	grade.D	grade.E	grade.F	grade.G	term. 36 months	term. 60 months
1	-1	0	0	1	0	0	0	0	0	1
6	-1	0	0	0	0	0	1	0	0	1
7	-1	0	1	0	0	0	0	0	0	1
10	-1	0	0	1	0	0	0	0	1	0
12	-1	0	1	0	0	0	0	0	1	0

5 rows × 25 columns

As can be seen above, the feature columns are now one-hot-encoded

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

In [10]:
```
features = loans_data.columns
features = features.drop('safe_loans')  # Remove the response variable
features
```

Out[10]:
```
Index(['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.OW
N',
       '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'],
      dtype='object')
```

In [11]:
```
print("Number of features (after binarizing categorical variables) = ", len(fe
atures))
```

```
Number of features (after binarizing categorical variables) =  24
```

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

In [12]:
```
loans_data['grade.A'].head()
```

Out[12]:
```
1     0
6     0
7     0
10    0
12    0
Name: grade.A, dtype: uint8
```

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

Checkpoint: Make sure the following answers match up.

In [13]:
```
print("Total number of grade.A loans : %s" % loans_data['grade.A'].sum())
print("Expexted answer               : 6422")
```

```
Total number of grade.A loans : 6508
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 `np.random.seed(1)` so that everyone gets the same result.

Splitting from the help of this [stackoverflow answer (https://stackoverflow.com/a/24147363/7771202)](https://stackoverflow.com/a/24147363/7771202)

In [14]:
```
import numpy as np

np.random.seed(1)
msk = np.random.rand(len(loans_data)) < 0.8 # a mask with 80% True

train_data, test_data = loans_data[msk], loans_data[~msk]

print(train_data.shape, test_data.shape)
```

```
(37038, 25) (9262, 25)
```

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 [15]:
```python
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

#     print('labels_in_node', labels_in_node)

    # Count the number of 1's (safe loans)
    ## YOUR CODE HERE
    n_1s = (labels_in_node==1).sum()

    # Count the number of -1's (risky loans)
    ## YOUR CODE HERE
    n_neg1s = (labels_in_node==-1).sum()

    # Return the number of mistakes that the majority classifier makes.
    ## YOUR CODE HERE
    majority, minority = (n_1s,n_neg1s) if (n_1s>n_neg1s) else (n_neg1s,n_1s)

    return minority
```

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 [16]:
```python
# Test case 1
example_labels = pd.Series([-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 = pd.Series([-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 = pd.Series([-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 [17]:
```
def best_splitting_feature(data, features, target):

    best_feature = None # Keep track of the best feature
    best_error = np.Inf     # 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 int
ermediate_node_num_mistakes)
        # 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

#         print(f'{str(feature).ljust(30)}:  {error} = ({left_mistakes}+{right
_mistakes})/{len(data[feature])}')

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

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

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

In [18]:
```
# test_data[test_data[features[1]] == 1]
```

In [19]:
```python
if best_splitting_feature(train_data, features, 'safe_loans') == 'term. 36 mon
ths':
    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 [20]:
```python
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']

    leaf['prediction'] = 1 if num_ones>num_minus_ones else -1

    # 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:** Reached max depth.
   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 [21]:
```
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 th
is.)
    if intermediate_node_num_mistakes(target_values) == 0:  ## YOUR CODE HERE
        print("Stopping condition 1 reached. All data points in a node are fro
m the same class."      )
        # 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 len(remaining_features) == 0:   ## YOUR CODE HERE
        print("Stopping condition 2 reached. No more features to split on."
)
        # 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 nod
e

        return create_leaf(target_values)

    # Find the best splitting feature (recall the function best_splitting_feat
ure implemented above)
    ## 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 = remaining_features.drop(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.")
        return create_leaf(right_split[target])
```

```
        ## YOUR CODE HERE


    # Repeat (recurse) on left and right subtrees
    left_tree = decision_tree_create(left_split, remaining_features, target, c
urrent_depth+1, max_depth)
    ## YOUR CODE HERE
    right_tree = decision_tree_create(right_split, remaining_features, target,
current_depth+1, max_depth)


    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 [22]:
```
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 [23]:
```python
small_data_decision_tree = decision_tree_create(train_data, features, 'safe_loans', max_depth = 3)
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 (37038 data points).
Split on feature term. 36 months. (9383, 27655)
--------------------------------------------------------------------
Subtree, depth = 1 (9383 data points).
Split on feature grade.A. (9255, 128)
--------------------------------------------------------------------
Subtree, depth = 2 (9255 data points).
Split on feature grade.B. (8194, 1061)
--------------------------------------------------------------------
Subtree, depth = 3 (8194 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (1061 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (128 data points).
Split on feature grade.B. (128, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 1 (27655 data points).
Split on feature grade.D. (22962, 4693)
--------------------------------------------------------------------
Subtree, depth = 2 (22962 data points).
Split on feature grade.E. (21677, 1285)
--------------------------------------------------------------------
Subtree, depth = 3 (21677 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (1285 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (4693 data points).
Split on feature grade.A. (4693, 0)
Creating leaf node.
Test failed... try again!
Number of nodes found                : 11
Number of nodes that should be there : 13
```

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 [24]:

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

```
--------------------------------------------------------------------
Subtree, depth = 0 (37038 data points).
Split on feature term. 36 months. (9383, 27655)
--------------------------------------------------------------------
Subtree, depth = 1 (9383 data points).
Split on feature grade.A. (9255, 128)
--------------------------------------------------------------------
Subtree, depth = 2 (9255 data points).
Split on feature grade.B. (8194, 1061)
--------------------------------------------------------------------
Subtree, depth = 3 (8194 data points).
Split on feature grade.C. (5955, 2239)
--------------------------------------------------------------------
Subtree, depth = 4 (5955 data points).
Split on feature grade.D. (3857, 2098)
--------------------------------------------------------------------
Subtree, depth = 5 (3857 data points).
Split on feature home_ownership.OTHER. (3856, 1)
--------------------------------------------------------------------
Subtree, depth = 6 (3856 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (1 data points).
Stopping condition 1 reached. All data points in a node are from the same cla
ss.
--------------------------------------------------------------------
Subtree, depth = 5 (2098 data points).
Split on feature grade.E. (2098, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 4 (2239 data points).
Split on feature emp_length.4 years. (2106, 133)
--------------------------------------------------------------------
Subtree, depth = 5 (2106 data points).
Split on feature grade.D. (2106, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 5 (133 data points).
Split on feature home_ownership.MORTGAGE. (65, 68)
--------------------------------------------------------------------
Subtree, depth = 6 (65 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (68 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (1061 data points).
Split on feature emp_length.3 years. (974, 87)
--------------------------------------------------------------------
Subtree, depth = 4 (974 data points).
Split on feature home_ownership.OWN. (899, 75)
--------------------------------------------------------------------
Subtree, depth = 5 (899 data points).
Split on feature emp_length.< 1 year. (816, 83)
--------------------------------------------------------------------
Subtree, depth = 6 (816 data points).
Reached maximum depth. Stopping for now.
```

```
--------------------------------------------------------------------
Subtree, depth = 6 (83 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 5 (75 data points).
Split on feature emp_length.10+ years. (52, 23)
--------------------------------------------------------------------
Subtree, depth = 6 (52 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (23 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 4 (87 data points).
Split on feature home_ownership.OWN. (79, 8)
--------------------------------------------------------------------
Subtree, depth = 5 (79 data points).
Split on feature grade.C. (79, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 5 (8 data points).
Split on feature grade.C. (8, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 2 (128 data points).
Split on feature grade.B. (128, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 1 (27655 data points).
Split on feature grade.D. (22962, 4693)
--------------------------------------------------------------------
Subtree, depth = 2 (22962 data points).
Split on feature grade.E. (21677, 1285)
--------------------------------------------------------------------
Subtree, depth = 3 (21677 data points).
Split on feature grade.F. (21317, 360)
--------------------------------------------------------------------
Subtree, depth = 4 (21317 data points).
Split on feature grade.C. (14215, 7102)
--------------------------------------------------------------------
Subtree, depth = 5 (14215 data points).
Split on feature grade.G. (14109, 106)
--------------------------------------------------------------------
Subtree, depth = 6 (14109 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (106 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 5 (7102 data points).
Split on feature home_ownership.RENT. (3466, 3636)
--------------------------------------------------------------------
Subtree, depth = 6 (3466 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (3636 data points).
Reached maximum depth. Stopping for now.
```

```
--------------------------------------------------------------------
Subtree, depth = 4 (360 data points).
Split on feature emp_length.8 years. (347, 13)
--------------------------------------------------------------------
Subtree, depth = 5 (347 data points).
Split on feature grade.A. (347, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 5 (13 data points).
Split on feature home_ownership.OWN. (8, 5)
--------------------------------------------------------------------
Subtree, depth = 6 (8 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 6 (5 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (1285 data points).
Split on feature grade.A. (1285, 0)
Creating leaf node.
--------------------------------------------------------------------
Subtree, depth = 2 (4693 data points).
Split on feature grade.A. (4693, 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 [25]:
```python
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_featu
re_value))

        if split_feature_value == 0:
            return classify(tree['left'], x, annotate)
        else:
            return classify(tree['right'], x, annotate)
```

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

In [26]:
```
test_data.iloc[0]
```

Out[26]:
```
safe_loans                  -1
grade.A                      0
grade.B                      0
grade.C                      0
grade.D                      1
grade.E                      0
grade.F                      0
grade.G                      0
term. 36 months              0
term. 60 months              1
home_ownership.MORTGAGE      0
home_ownership.OTHER         0
home_ownership.OWN           0
home_ownership.RENT          1
emp_length.1 year            0
emp_length.10+ years         0
emp_length.2 years           0
emp_length.3 years           0
emp_length.4 years           0
emp_length.5 years           1
emp_length.6 years           0
emp_length.7 years           0
emp_length.8 years           0
emp_length.9 years           0
emp_length.< 1 year          0
Name: 58, dtype: int64
```

In [27]:
```
print('Predicted class: %s ' % classify(my_decision_tree, test_data.iloc[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 [28]:
```
classify(my_decision_tree, test_data.iloc[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[28]:
```
-1
```

Analysis Question 1: What was the feature that **my_decision_tree** first split on while making the prediction for test_data[0]?

Split on `term. 36 months`

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

`grade.D`

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

`grade.D`

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 [29]:
```
def evaluate_classification_error(tree, data, target):
    # Apply the classify(tree, x) to each row in your data
    predictions = data.apply(lambda x: classify(tree, x), axis=1)
    # Once you've made the predictions, calculate the classification error and
return it
    mistakes = (predictions!=data[target]).sum()
    return mistakes / len(data[target])
```

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

In [30]:
```python
test_err = evaluate_classification_error(my_decision_tree, test_data, target)
print(test_err)
```

```
0.3787518894407255
```

Analysis Question 4: Rounded to 2nd decimal point, what is the classification error of **my_decision_tree** on the **test_data**?

In [31]:
```python
print('rounded test_err={:.2f}'.format(test_err))
```

```
rounded test_err=0.38
```

Printing out a decision stump

We can print(out a single decision stump (printing out the entire tree is left as an exercise for the curious).)

In [32]:
```python
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.ljust(10)))
    print('         |                                |')
    print('         |                                |')
    print('         |                                |')
    print('    (%s)                         (%s)' % (('leaf, label: ' +
          str(tree['left']['prediction']) if tree['left']['is_leaf'] else 'subtree'),
           ('leaf, label: ' + str(tree['right']['prediction']) if tree['right']['is_leaf'] else 'subtree'))
         )
```

In [33]:
```
print_stump(my_decision_tree)
```

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

Analysis Question 5: What is the feature that is used for the split at the root node?

Ans: `term. 36 months`

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)                        (leaf, label: 1)
```

Exploring the left subtree of the left subtree

In [35]:
```python
print_stump(my_decision_tree['left']['left'], my_decision_tree['left']['splitting_feature'])
```

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

In [36]:
```python
print_stump(my_decision_tree['left']['left']['left'], my_decision_tree['left']['left']['splitting_feature'])
```

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

Analysis Question 6: What is the path of the **first 3 feature splits** considered along the **left-most** branch of **my_decision_tree**?

In [37]:
```python
#traverse left

current_node = my_decision_tree
while True:
    if not current_node or 'splitting_feature' not in current_node:
        break

    print(current_node['splitting_feature'])
    current_node = current_node['left']
```

```
term. 36 months
grade.A
grade.B
grade.C
grade.D
home_ownership.OTHER
None
```

Analysis Question 7: What is the path of the **first 3 feature splits** considered along the **right-most** branch of **my_decision_tree**?

In [38]:

```python
#traverse right

current_node = my_decision_tree
while True:
    if not current_node or 'splitting_feature' not in current_node:
        break

    print(current_node['splitting_feature'])
    current_node = current_node['right']
```

```
term. 36 months
grade.D
None
```