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 :)

If you are doing the assignment with IPython Notebook

An IPython Notebook has been provided below to you for this assignment. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

Make sure that you are using GraphLab Create 1.8.3. See this post for installing the correct version of GraphLab Create.

What you need to download

If you are using GraphLab Create:

- Download the Lending club data In SFrame format: lending-club-data.gl.zip
- Download the companion IPython Notebook: module-5-decision-tree-assignment-2-blank.ipynb
- Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.
- Follow the instructions contained in the IPython notebook.

If you are not using GraphLab Create

- If you are using SFrame, download the LendingClub dataset in SFrame format: lending-club-data.gl.zip
- If you are using a different package, download the LendingClub dataset in CSV format: lending-club-data.csv.zip

If you are using GraphLab Create and the companion IPython Notebook

Open the companion IPython notebook and follow the instructions in the notebook.

If you are using other tools

This section is designed for people using tools other than GraphLab Create. **You will not need any machine learning packages** since we will be implementing decision trees from scratch. **We highly suggest you use SFrame since it is open source.** In this part of the assignment, we describe general instructions, however we will tailor the instructions for SFrame.

- If you choose to use SFrame, you should be able to follow the instructions in the next section and complete the assessment. **All code samples given here will be applicable to SFrame**.
- You are free to experiment with any tool of your choice, but **some many not produce correct numbers for the quiz questions.**

If you are using SFrame

Make sure to download the companion IPython notebook: module-5-decision-tree-assignment-2-blank.ipynb. You will be able to follow along exactly **if you replace the first two lines of code with these two lines:**

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

After running this, **you can follow the rest of the IPython notebook and disregard the rest of this reading.**

Note: To install SFrame (without installing GraphLab Create), run

```
pip install sframe
```

If you are NOT using SFrame

1. Load in the LendingClub dataset with the software of your choice.

2. Like the previous assignment, reassign the labels to have +1 for a safe loan, and -1 for a risky (bad) loan. You should have code analogous to

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

3. Unlike the previous assignment, we will only be considering these four features:

```
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'
```

Extract these feature columns from the dataset, and discard the rest of the feature columns.

Notes to people using other tools

If you are using SFrame, proceed to the section "Subsample dataset to make sure classes are balanced".

If you are NOT using SFrame, download the list of indices for the training and test sets: module-5-assignment-2-train-idx.json, module-5-assignment-2-test-idx.json. Then follow the following steps:

- Apply one-hot encoding to **loans**. Your tool may have a function for one-hot encoding.
- Load the JSON files into the lists **train_idx** and **test_idx**.
- Perform train/validation split using **train_idx** and **test_idx**. In Pandas, for instance:

```
train_data = loans.iloc[train_idx]
test_data = loans.iloc[test_idx]
```

IMPORTANT: If you are using a programming language with 1-based indexing (e.g. R, Matlab), make sure to increment all indices by 1.

Note. Some elements in loans are included neither in **train_data** nor **test_data**. This is to perform sampling to achieve class balance.

Now proceed to the section "Decision tree implementation", skipping three sections below.

Subsample dataset to make sure classes are balanced

4. 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. You should have code analogous to

```
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)
```

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 this paper. 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
}
```

5. This technique of turning categorical variables into binary variables is called one-hot encoding. Using the software of your choice, perform one-hot encoding on the four features described above. **You should now have 25 binary features.**

Train-test split

6. 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. Call these datasets **train_data** and **test_data**, respectively. Using SFrame, this would look like:

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

(with **seed=1** to ensure people get the same results.)

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 **misclassified 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.

7. 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. Your code should be analogous to

```
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
    # Count the number of -1's (risky loans)
    ## YOUR CODE HERE
```

```
    # Return the number of mistakes that the majority classifier makes.
    ## YOUR CODE HERE
```

8. 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. Again, remember that this code is specific to SFrame, but using your software of choice, you can construct similar tests.

```
# 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 3 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!'
```

Function to pick best feature to split on

The function *best_splitting_feature* takes 3 arguments:

1. The data
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:

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

9. Follow these steps to implement *best_splitting_feature*:

- **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**.

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.

Your code should be analogous to

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


    target_values = data[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 =

        # Calculate the number of misclassified examples in the left split.
        # Remember that we implemented a function for this! (It was called intermediate_node_num_mistak
es)
        # YOUR CODE HERE
        left_mistakes =

        # Calculate the number of misclassified examples in the right split.
        ## YOUR CODE HERE
        right_mistakes =

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

        # 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:
```

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

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.}
```

10. First, we will write a function that creates a leaf node given a set of target values. Your code should be analogous to

```
def create_leaf(target_values):
    # Create a leaf node
    leaf = {'splitting_feature' : None,
            'left' : None,
            'right' : None,
            'is_leaf':     }   ## 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'] =          ## YOUR CODE HERE
    else:
        leaf['prediction'] =          ## 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.

11. Now, we will provide a Python skeleton of the learning algorithm. Note that this code is not complete; it needs to be completed by you if you are using Python. Otherwise, your code should be analogous to

```
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  == 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 >= :  ## 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 implemented above)
## YOUR CODE HERE



# Split on the best feature that we found.
left_split = data[data[splitting_feature] == 0]
right_split =       ## 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



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

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

Build the tree!

12. 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**: The tree may take 1-2 minutes to learn.

Making predictions with a decision tree

13. As discussed in the lecture, we can make predictions from the decision tree with a simple recursive function. Write a function called *classify*, which takes in a learned tree and a test point x to classify. Include an option annotate that describes the prediction path when set to True. Your code should be analogous to

```
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:
```

```
    ### YOUR CODE HERE
```

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

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

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

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

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

16. Now, we will write a function to evaluate a decision tree by computing the classification error of the tree on the given dataset. Write a function called *evaluate_classification_error* that takes in as input:

1. tree (as described above)
2. data (a data frame of data points)

This function should return a prediction (class label) for each row in data using the decision tree. Your code should be analogous to

```
def evaluate_classification_error(tree, data):
    # 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
```

```

```

17. Now, use this function to evaluate the classification error on the test set.

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

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

18. 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). Here we provide Python code to visualize a decision stump. If you are using different software, make sure your code is analogous to:

```
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'] else 'subtree'),
           ('leaf, label: ' + str(tree['right']['prediction']) if tree['right']['is_leaf'] else 'subtree'))
```

19. Using this function, we can print out the root of our decision tree:

```
print_stump(my_decision_tree)
```

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

20. We can print out the left subtree by running the code

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

We can similarly print out the left subtree of the left subtree of the root by running the code

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

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?