Decision Trees in Practice

In this assignment we will explore various techniques for preventing overfitting in decision trees. We will extend the implementation of the binary decision trees that we implemented in the previous assignment. You will have to use your solutions from this previous assignment and extend them.

In this assignment you will:

- Implement binary decision trees with different early stopping methods.
- Compare models with different stopping parameters.
- Visualize the concept of overfitting in decision trees.

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-6-decision-tree-practical-assignment-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-6-decision-tree-practical-assignment-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. As before, we 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. We will be using the same 4 categorical features as in the previous assignment:

```
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 and target column 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 validation sets: module-6-assignment-train-idx.json, module-6-assignment-validation-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 **validation_idx**.
- Perform train/validation split using **train_idx** and **validation_idx**. In Pandas, for instance:

```
train_data = loans.iloc[train_idx]
validation_data = loans.iloc[validation_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 **validation_data**. This is to perform sampling to achieve class balance.

Now proceed to the section "Early stopping methods for decision trees", 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

Just like the previous assignment, we will implement **binary decision trees**. Since all of our features are currently categorical features, we want to turn them into binary features. Here is a reminder of what one-hot encoding is.

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-validation split

6. Split the data into a train-validation split with 80% of the data in the training set and 20% of the data in the validation set. Call these datasets **train_data** and **validation_data**, respectively. Using SFrame, this would look like:

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

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

Early stopping methods for decision trees

In this section, we will extend the **binary tree implementation** from the previous assignment in order to handle some early stopping conditions. Recall the 3 early stopping methods that were discussed in lecture:

1. Reached a **maximum depth**. (set by parameter *max_depth*).

2. Reached a **minimum node size**. (set by parameter *min_node_size*).
3. Don't split if the **gain in error reduction** is too small. (set by parameter *min_error_reduction*).

For the rest of this assignment, we will refer to these three as **early stopping conditions 1, 2, and 3**.

Early stopping condition 1: Maximum depth

Recall that we already implemented the maximum depth stopping condition in the previous assignment. In this assignment, we will experiment with this condition a bit more and also write code to implement the 2nd and 3rd early stopping conditions.

We will be reusing code from the previous assignment and then building upon this. We will **alert you** when you reach a function that was part of the previous assignment so that you can simply copy and past your previous code.

Early stopping condition 2: Minimum node size

The function **reached_minimum_node_size** takes 2 arguments:

1. The data (from a node)
2. The minimum number of data points that a node is allowed to split on, *min_node_size*.

7. This function simply calculates whether the number of data points at a given node is less than or equal to the specified minimum node size. This function will be used to detect this early stopping condition in the**decision_tree_create** function. Your code should be analogous to

```
def reached_minimum_node_size(data, min_node_size):
    # Return True if the number of data points is less than or equal to the minimum node size.
    ## YOUR CODE HERE
```

Quiz question: Given an intermediate node with 6 safe loans and 3 risky loans, if the min_node_size parameter is 10, what should the tree learning algorithm do next?

Early stopping condition 3: Minimum gain in error reduction

The function **error_reduction** takes 2 arguments:

1. The error **before** a split, *error_before_split*.
2. The error **after** a split, *error_after_split*.

8. This function computes the gain in error reduction, i.e., the difference between the error before the split and that after the split. This function will be used to detect this early stopping condition in the **decision_tree_create**function. Your code should be analogous to

```
def error_reduction(error_before_split, error_after_split):
    # Return the error before the split minus the error after the split.
    ## YOUR CODE HERE
```

Quiz question: Assume an intermediate node has 6 safe loans and 3 risky loans. For each of 4 possible features to split on, the error reduction is 0.0, 0.05, 0.1, and 0.14, respectively. If the **minimum gain in error reduction**parameter is set to 0.2, what should the tree learning algorithm do next?

Grabbing binary decision tree helper functions from past assignment

9. Recall from the previous assignment that we wrote a function **intermediate_node_num_mistakes** that calculates the number of **misclassified examples** when predicting the **majority class**. This is used to help determine which feature is best to split on at a given node of the tree. **Use your code from the previous assignment. It 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
```

10. We then wrote a function **best_splitting_feature** that finds the best feature to split on given the data and a list of features to consider. **Use your code from the previous assignment. It 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
```

11. Finally, recall the function **create_leaf** from the previous assignment, which creates a leaf node given a set of target values. **Use your code from the previous assignment. It 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
```

Incorporating new early stopping conditions in binary decision tree implementation

12. Now, you will implement a function that builds a decision tree handling the three early stopping conditions described in this assignment. In particular, you will write code to detect early stopping conditions 2 and 3. You implemented above the functions needed to detect these conditions. The 1st early stopping condition, **max_depth**, was implemented in the previous assignment and you will not need to reimplement this. In addition to these early stopping conditions, the typical stopping conditions of having no mistakes or no more features to split on (which we denote by "stopping conditions" 1 and 2) are also included as in the previous assignment.

Implementing early stopping condition 2: minimum node size:

- **Step 1:** Use the function **reached_minimum_node_size** that you implemented earlier to write an if condition to detect whether we have hit the base case, i.e., the node does not have enough data points and should be turned into a leaf. Don't forget to use the *min_node_size* argument.
- **Step 2:** Return a leaf. This line of code should be the same as the other (pre-implemented) stopping conditions.

Implementing early stopping condition 3: minimum error reduction:

Note: This has to come after finding the best splitting feature so we can calculate the error after splitting in order to calculate the error reduction. Recall that classification error is defined as:

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

- Step 1: Calculate the **classification error before splitting**.
- Step 2: Calculate the **classification error after splitting**. This requires calculating the number of mistakes in the left and right splits, and then dividing by the total number of examples.
- Step 3: Use the function **error_reduction** to that you implemented earlier to write an if condition to detect whether the reduction in error is less than the constant provided (*min_error_reduction*). Don't forget to use that argument.

- Step 4: Return a leaf. This line of code should be the same as the other (pre-implemented) stopping conditions.

  Your code should be analogous to

```
def decision_tree_create(data, features, target, current_depth = 0,
                         max_depth = 10, min_node_size=1,
                         min_error_reduction=0.0):

    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: All nodes are of the same type.
    if intermediate_node_num_mistakes(target_values) == 0:
        print "Stopping condition 1 reached. All data points have the same target value."
        return create_leaf(target_values)

    # Stopping condition 2: No more features to split on.
    if remaining_features == []:
        print "Stopping condition 2 reached. No remaining features."
        return create_leaf(target_values)

    # Early stopping condition 1: Reached max depth limit.
    if current_depth >= max_depth:
        print "Early stopping condition 1 reached. Reached maximum depth."
        return create_leaf(target_values)

    # Early stopping condition 2: Reached the minimum node size.
    # If the number of data points is less than or equal to the minimum size, return a leaf.
    if           ## YOUR CODE HERE
        print "Early stopping condition 2 reached. Reached minimum node size."
```

```
    return   ## YOUR CODE HERE

# Find the best splitting feature
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]

# Early stopping condition 3: Minimum error reduction
# Calculate the error before splitting (number of misclassified examples
# divided by the total number of examples)
error_before_split = intermediate_node_num_mistakes(target_values) / float(len(data))

# Calculate the error after splitting (number of misclassified examples
# in both groups divided by the total number of examples)
left_mistakes =    ## YOUR CODE HERE
right_mistakes =   ## YOUR CODE HERE
error_after_split = (left_mistakes + right_mistakes) / float(len(data))

# If the error reduction is LESS THAN OR EQUAL TO min_error_reduction, return a leaf.
if         ## YOUR CODE HERE
    print "Early stopping condition 3 reached. Minimum error reduction."
    return   ## YOUR CODE HERE



remaining_features.remove(splitting_feature)
print "Split on feature %s. (%s, %s)" % (\
                splitting_feature, len(left_split), len(right_split))



# Repeat (recurse) on left and right subtrees
left_tree = decision_tree_create(left_split, remaining_features, target,
                                 current_depth + 1, max_depth, min_node_size, min_error_reduction)
```

```
    ## YOUR CODE HERE
    right_tree =


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

Build a tree!

13. Now that your code is working, train a tree model on the **train_data** with

- *max_depth* = 6
- *min_node_size* = 100,
- *min_error_reduction* = 0.0

Warning: This code block may take a minute to learn. Call this model **my_decision_tree_new**. Your code should be analogous to

```
my_decision_tree_new = decision_tree_create(train_data, features, 'safe_loans', max_depth = 6,
                                            min_node_size = 100, min_error_reduction=0.0)
```

14. Let's now train a tree model **ignoring early stopping conditions 2 and 3** so that we get the **same tree** as in the previous assignment. To ignore these conditions, we set *min_node_size*=0 and *min_error_reduction*=-1 (a negative value). Call this model **my_decision_tree_old**. Your code should be analogous to

```
my_decision_tree_old = decision_tree_create(train_data, features, 'safe_loans', max_depth = 6,
min_node_size = 0, min_error_reduction=-1)
```

Making predictions

15. Recall that in the previous assignment you implemented a function classify to classify a new point x using a given tree. We will need that function here. **Use your code from the previous assignment. It 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
```

16. Now, let's consider the first example of the validation set and see what the **my_decision_tree_new** model predicts for this data point. Your code should be analogous to

```
print validation_set[0]
print 'Predicted class: %s ' % classify(my_decision_tree_new, validation_set[0])
```

17. 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_new, validation_set[0], annotate = True)
```

18. Let's now recall the prediction path for the decision tree learned in the previous assignment, which we recreated here as my_decision_tree_old.

```
classify(my_decision_tree_old, validation_set[0], annotate = True)
```

Quiz question: For **my_decision_tree_new** trained with *max_depth* = 6, *min_node_size* = 100,*min_error_reduction*=0.0, is the prediction path for validation_set[0] shorter, longer, or the same as for**my_decision_tree_old** that ignored the early stopping conditions 2 and 3?

Quiz question: For **my_decision_tree_new** trained with *max_depth* = 6, *min_node_size* = 100,*min_error_reduction*=0.0, is the prediction path for any point always shorter, always longer, always the same, shorter or the same, or longer or the same as for **my_decision_tree_old** that ignored the early stopping conditions 2 and 3?

Quiz question: For a tree trained on **any** dataset using *max_depth* = 6, *min_node_size* = 100,*min_error_reduction*=0.0, what is the maximum number of splits encountered while making a single prediction?

Evaluating the model

19. Now let us evaluate the model that we have trained. You implemented this evaluation in the function**evaluate_classification_error** from the previous assignment. **Use your code from the previous assignment. It should be analogous to**

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

    # Once you've made the prediction, calculate the classification error
    ## YOUR CODE HERE
```

20. Now, let's use this function to evaluate the classification error of **my_decision_tree_new** on the **validation_set**. Your code should be analogous to

```
evaluate_classification_error(my_decision_tree_new, validation_set)
```

21. Now, evaluate the validation error using **my_decision_tree_old**.

```
evaluate_classification_error(my_decision_tree_old, validation_set)
```

Quiz question: Is the validation error of the new decision tree (using early stopping conditions 2 and 3) lower than, higher than, or the same as that of the old decision tree from the previous assignment?

Exploring the effect of *max_depth*

We will compare three models trained with different values of the stopping criterion. We intentionally picked models at the extreme ends (**too small**, **just right**, and **too large**).

22. Train three models with these parameters:

1. **model_1**: *max_depth* = 2 (too small)
2. **model_2**: *max_depth* = 6 (just right)
3. **model_3**: *max_depth* = 14 (may be too large)

For each of these three, set *min_node_size* = 0 and *min_error_reduction* = -1. Make sure to call the models**model_1, model_2,** and **model_3**.

Note: Each tree can take up to a few minutes to train. In particular, **model_3** will probably take the longest to train.

23. Let us evaluate the models on the **train** and **validation** data. Let us start by evaluating the classification error on the training data. Your code should be analogous to:

```
print "Training data, classification error (model 1):", evaluate_classification_error(model_1, train_data)
print "Training data, classification error (model 2):", evaluate_classification_error(model_2, train_data)
print "Training data, classification error (model 3):", evaluate_classification_error(model_3, train_data)
```

24. Now evaluate the classification error on the validation data.

Quiz Question: Which tree has the smallest error on the validation data?

Quiz Question: Does the tree with the smallest error in the training data also have the smallest error in the validation data?

Quiz Question: Is it always true that the tree with the lowest classification error on the **training** set will result in the lowest classification error in the **validation** set?

Measuring the complexity of the tree

Recall in the lecture that we talked about deeper trees being more complex. We will measure the complexity of the tree as

```
complexity(T) = number of leaves in the tree T
```

25. Here, we provide a function count_leaves that counts the number of leaves in a tree. Using this implementation, compute the number of nodes in model_1, model_2, and model_3. This code is in Python. If you are using another language, make sure your code is analogous to

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

26. Using the function **count_leaves**, compute the number of nodes in **model_1**, **model_2**, and **model_3**.

Quiz question: Which tree has the largest complexity?

Quiz question: Is it always true that the most complex tree will result in the lowest classification error in the**validation_set**?

Exploring the effect of *min_error*

We will compare three models trained with different values of the stopping criterion. We intentionally picked models at the extreme ends (**negative**, **just right**, and **too positive**).

27. Train three models with these parameters:

- **model_4**: *min_error_reduction* = -1 (ignoring this early stopping condition)
- **model_5**: *min_error_reduction* = 0 (just right)
- **model_6**: *min_error_reduction* = 5 (too positive)

For each of these three, we set *max_depth* = 6, and *min_node_size* = 0. Make sure to call the models **model_4, model_5,** and **model_6**.

Note: Each tree can take up to 30 seconds to train.

28. Calculate the accuracy of each model (**model_4**, **model_5**, or **model_6**) on the validation set. Your code should be analogous to

```
print "Validation data, classification error (model 4):", evaluate_classification_error(model_4, validation_set)
print "Validation data, classification error (model 5):", evaluate_classification_error(model_5, validation_set)
print "Validation data, classification error (model 6):", evaluate_classification_error(model_6, validation_set)
```

29. Using the **count_leaves** function, compute the number of leaves in each of each models in (model_4, model_5, and model_6).

Quiz Question: Using the complexity definition above, which model (**model_4**, **model_5**, or **model_6**) has the largest complexity? Did this match your expectation?

Quiz Question: model_4 and **model_5** have similar classification error on the validation set but **model_5** has lower complexity? Should you pick **model_5** over **model_4**?

Exploring the effect of *min_node_size*

We will compare three models trained with different values of the stopping criterion. Again, intentionally picked models at the extreme ends (**too small**, **just right**, and **just right**).

30. Train three models with these parameters:

- **model_7**: *min_node_size* = 0 (too small)
- **model_8**: *min_node_size* = 2000 (just right)
- **model_9**: *min_node_size* = 50000 (too large)

For each of these three, we set *max_depth* = 6, and *min_error_reduction* = -1. Make sure to call these models**model_7, model_8,** and **model_9.**

Note: Each tree can take up to 30 seconds to train.

31. Calculate the accuracy of each model (**model_7**, **model_8**, or **model_9**) on the validation set.

32. Using the count_leaves function, compute the number of leaves in each of each models (**model_7**, **model_8**, and**model_9**).

Quiz Question: Using the results obtained in this section, which model (**model_7**, **model_8**, or **model_9**) would you choose to use?