Boosting a decision stump

In this homework you will implement your own boosting module.

Brace yourselves! This is going to be a fun and challenging assignment.

- Use SFrames to do some feature engineering.
- Train a boosted ensemble of decision-trees (gradient boosted trees) on the lending club dataset.
- Predict whether a loan will default along with prediction probabilities (on a validation set).
- Evaluate the trained model and compare it with a baseline.
- Find the most positive and negative loans using the learned model.
- Explore how the number of trees influences classification performance.

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-8-boosting-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-8-boosting-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

Getting the data ready

We will be using a dataset from the LendingClub.

1. Load the dataset into a data frame named **loans**.

Extracting the target and the feature columns

2. We will now repeat some of the feature processing steps that we saw in the previous assignment:

First, we re-assign the target to have +1 as a safe (good) loan, and -1 as a risky (bad) loan.

Next, we select four categorical features:

- grade of the loan
- the length of the loan term
- the home ownership status: own, mortgage, rent
- number of years of employment.

Your code should be analogous to the following:

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

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-8-assignment-2-train-idx.json, module-8-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. Alternatively, see #7 for implementation hints.
- 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 "Weighted decision trees", skipping three sections below.

Subsample dataset to make sure classes are balanced

3. Just as we did in the previous assignment, we will undersample the larger class (safe loans) in order to balance out our dataset. This means we are throwing away many data points. We use seed=1 so everyone gets the same results. Your code should be analogous to the following:

```
safe_loans_raw = loans[loans[target] == 1]
risky_loans_raw = loans[loans[target] == -1]
# Undersample the safe loans.
percentage = len(risky_loans_raw)/float(len(safe_loans_raw))
risky_loans = risky_loans_raw
safe_loans = safe_loans_raw.sample(percentage, seed=1)
loans_data = risky_loans_raw.append(safe_loans)
```

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

4. 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-test split

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

Call the training and test sets **train_data** and **test_data**, respectively.

Weighted decision trees

7. Let's modify our decision tree code from Module 5 to support weighting of individual data points.

Weighted error definition

8. Consider a model with N data points with:

- Predictions ŷ_1, ..., ŷ_n
- Target y_1, ..., y_n
- Data point weights α_1, ..., α_n

Then the **weighted error** is defined by:

$$\mathrm{E}(\alpha, \hat{\mathbf{y}}) = \frac{\sum_{i=1}^{n} \alpha_i \times 1[y_i \neq \hat{y}_i]}{\sum_{i=1}^{n} \alpha_i}$$

where 1[y_i ≠ ŷ_i] is an indicator function that is set to 1 if y_i ≠ ŷ_i.

Write a function to compute weight of mistakes

9. Write a function that calculates the weight of mistakes for making the "weighted-majority" predictions for a dataset. The function accepts two inputs:

- **labels_in_node**: y_1, ..., y_n
- **data_weights**: Data point weights α_1, ..., α_n

We are interested in computing the (total) weight of mistakes, i.e.

$$\mathrm{WM}(\alpha, \hat{\mathbf{y}}) = \sum_{i=1}^{n} \alpha_i \times 1[y_i \neq \hat{y}_i].$$

This quantity is analogous to the number of mistakes, except that each mistake now carries different weight. It is related to the weighted error in the following way:

$$\mathrm{E}(\alpha, \hat{\mathbf{y}}) = \frac{\mathrm{WM}(\alpha, \hat{\mathbf{y}})}{\sum_{i=1}^{n} \alpha_i}$$

The function **intermediate_node_weighted_mistakes** should first compute two weights:

- WM(−1): weight of mistakes when all predictions are ŷ_i = −1 i.e. WM(α,**−1**)
- WM(+1): weight of mistakes when all predictions are ŷ_i = +1 i.e. WM(α,**+1**)

where **−1** and **+1** are vectors where all values are -1 and +1 respectively.

After computing WM(**−1**) and WM(**+1**), the function **intermediate_node_weighted_mistakes** should return the lower of the two weights of mistakes, along with the class associated with that weight. The function should be analogous to the following Python function:

```
def intermediate_node_weighted_mistakes(labels_in_node, data_weights):
    # Sum the weights of all entries with label +1
    total_weight_positive = sum(data_weights[labels_in_node == +1])

    # Weight of mistakes for predicting all -1's is equal to the sum above
    ### YOUR CODE HERE
    weighted_mistakes_all_negative = ...

    # Sum the weights of all entries with label -1
    ### YOUR CODE HERE
    total_weight_negative = ...

    # Weight of mistakes for predicting all +1's is equal to the sum above
    ### YOUR CODE HERE
    weighted_mistakes_all_positive = ...

    # Return the tuple (weight, class_label) representing the lower of the two weights
    #    class_label should be an integer of value +1 or -1.
    # If the two weights are identical, return (weighted_mistakes_all_positive,+1)
    ### YOUR CODE HERE
    ...
```

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

$$\text{classification error} = \frac{\#\text{ mistakes}}{\#\text{ all data points}}$$

Quiz Question: If we set the weights α=1 for all data points, how is the weight of mistakes WM(α,ŷ) related to the classification error?

Function to pick best feature to split on

11. We continue modifying our decision tree code from the earlier assignment to incorporate weighting of individual data points. The next step is to pick the best feature to split on.

The **best_splitting_feature** function is similar to the one from the earlier assignment with two minor modifications:

- The function **best_splitting_feature** should now accept an extra parameter data_weights to take account of weights of data points.
- Instead of computing the number of mistakes in the left and right side of the split, we compute the weight of mistakes for both sides, add up the two weights, and divide it by the total weight of the data.

Your function should be analogous to the following Python function:

```
# If the data is identical in each feature, this function should return None

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

    # These variables will keep track of the best feature and the corresponding error
    best_feature = None
    best_error = float('+inf')
    num_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
        # The right split will have all data points where the feature value is 1
        left_split = data[data[feature] == 0]
        right_split = data[data[feature] == 1]
```

```
        # Apply the same filtering to data_weights to create left_data_weights, right_data_we
ights
        ## YOUR CODE HERE
        left_data_weights = ...
        right_data_weights = ...

        # DIFFERENT HERE
        # Calculate the weight of mistakes for left and right sides
        ## YOUR CODE HERE
        left_weighted_mistakes, left_class = ...
        right_weighted_mistakes, right_class = ...

        # DIFFERENT HERE
        # Compute weighted error by computing
        #  ( [weight of mistakes (left)] + [weight of mistakes (right)] ) / [total weight of
all data points]
        ## YOUR CODE HERE
        error = ...

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

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

Very Optional. Relationship between weighted error and weight of mistakes

By definition, the weighted error is the weight of mistakes divided by the weight of all data points, so

$$\mathrm{E}(\alpha, \hat{\mathbf{y}}) = \frac{\sum_{i=1}^{n} \alpha_i \times 1[y_i \neq \hat{y}_i]}{\sum_{i=1}^{n} \alpha_i} = \frac{\mathrm{WM}(\alpha, \hat{\mathbf{y}})}{\sum_{i=1}^{n} \alpha_i}$$

In the code above, we obtain E(α, ŷ) from the two weights of mistakes from both sides, WM(α_left, ŷ_left) and WM(α_right, ŷ_right). First, notice that the overall weight of mistakes WM(α, ŷ) can be broken into two weights of mistakes over either side of the split:

$$\begin{aligned}\mathrm{WM}(\alpha, \hat{\mathbf{y}}) = \sum_{i=1}^{n} \alpha_i \times 1[y_i \neq \hat{y}_i] &= \sum_{\text{left}} \alpha_i \times 1[y_i \neq \hat{y}_i] + \sum_{\text{right}} \alpha_i \times 1[y_i \neq \hat{y}_i] \\ &= \mathrm{WM}(\alpha_{\text{left}}, \hat{\mathbf{y}}_{\text{left}}) + \mathrm{WM}(\alpha_{\text{right}}, \hat{\mathbf{y}}_{\text{right}})\end{aligned}$$

We then divide through by the total weight of all data points to obtain E(α, ŷ):

$$\mathrm{E}(\alpha, \hat{\mathbf{y}}) = \frac{\mathrm{WM}(\alpha_{\text{left}}, \hat{\mathbf{y}}_{\text{left}}) + \mathrm{WM}(\alpha_{\text{right}}, \hat{\mathbf{y}}_{\text{right}})}{\sum_{i=1}^{n} \alpha_i}$$

Building the tree

12. With the above functions implemented correctly, we are now ready to build our decision tree. Recall from the previous assignments that each node in the decision tree is represented as a dictionary which contains the following keys:

```
{
   'is_leaf'            : True/False.
   'prediction'         : Prediction at the leaf node.
   'left'               : (dictionary corresponding to the left tree).
   'right'              : (dictionary corresponding to the right tree).
   'features_remaining' : List of features that are posible splits.
}
```

Let us start with a function that creates a leaf node given a set of target values. The **create_leaf** function should be analogous to the following cell:

```
def create_leaf(target_values, data_weights):

    # Create a leaf node
    leaf = {'splitting_feature' : None,
            'is_leaf': True}

    # Computed weight of mistakes.
    # Store the predicted class (1 or -1) in leaf['prediction']
    weighted_error, best_class = intermediate_node_weighted_mistakes(target_values, data_weig
hts)
    leaf['prediction'] = ... ## YOUR CODE HERE

    return leaf
```

13. Now write a function that learns a weighted decision tree recursively and implements 3 stopping conditions:

- All data points in a node are from the same class.
- No more features to split on.
- Stop growing the tree when the tree depth reaches **max_depth**.

Since there are many steps involved, we provide you with a Python skeleton, along with explanatory comments.

```
def weighted_decision_tree_create(data, features, target, data_weights, current_depth = 1, ma
x_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. Error is 0.
    if intermediate_node_weighted_mistakes(target_values, data_weights)[0] <= 1e-15:
        print "Stopping condition 1 reached."
        return create_leaf(target_values, data_weights)

    # Stopping condition 2. No more features.
    if remaining_features == []:
        print "Stopping condition 2 reached."
        return create_leaf(target_values, data_weights)

    # Additional stopping condition (limit tree depth)
    if current_depth > max_depth:
        print "Reached maximum depth. Stopping for now."
        return create_leaf(target_values, data_weights)

    # If all the datapoints are the same, splitting_feature will be None. Create a leaf
    splitting_feature = best_splitting_feature(data, features, target, data_weights)
    remaining_features.remove(splitting_feature)

    left_split = data[data[splitting_feature] == 0]
    right_split = data[data[splitting_feature] == 1]

    left_data_weights = data_weights[data[splitting_feature] == 0]
    right_data_weights = data_weights[data[splitting_feature] == 1]
```

```
    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], data_weights)
    if len(right_split) == len(data):
        print "Creating leaf node."
        return create_leaf(right_split[target], data_weights)

    # Repeat (recurse) on left and right subtrees
    left_tree = weighted_decision_tree_create(
        left_split, remaining_features, target, left_data_weights, current_depth + 1, max_dep
th)
    right_tree = weighted_decision_tree_create(
        right_split, remaining_features, target, right_data_weights, current_depth + 1, max_d
epth)

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

14. Finally, write a recursive function to count the nodes in your tree. The function should be analogous to

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

Making predictions with a weighted decision tree

15. To make a single prediction, we must start at the root and traverse down the decision tree in recursive fashion. Write a function **classify** that makes a single prediction. It should be analogous to the following:

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

Evaluating the tree

16. Create a function called **evaluate_classification_error**. It takes in as input:

- **tree** (as described above)
- **data** (an data frame)

The function does not change because of adding data point weights. It is analogous to this Python function:

```
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
    return (prediction != data[target]).sum() / float(len(data))
```

Example: Training a weighted decision tree

17. To build intuition on how weighted data points affect the tree being built, consider the following:

Suppose we only care about making good predictions for the **first 10 and last 10 items** in train_data, we assign weights:

- 1 to the last 10 items
- 1 to the first 10 items
- and 0 to the rest.

Let us fit a weighted decision tree with max_depth = 2. Then compute the classification error on the **subset_20**, i.e. the subset of data points whose weight is 1 (namely the first and last 10 data points).

```
# Assign weights
example_data_weights = graphlab.SArray([1.] * 10 + [0.]*(len(train_data) - 20) + [1.] * 10)
# Train a weighted decision tree model.
small_data_decision_tree_subset_20 = weighted_decision_tree_create(train_data, features, targ
et,
                         example_data_weights, max_depth=2)
```

18. Now, we will compute the classification error on the **subset_20**, i.e. the subset of data points whose weight is 1 (namely the first and last 10 data points).

```
evaluate_classification_error(small_data_decision_tree_subset_20, train_data)
```

The model small_data_decision_tree_subset_20 performs **a lot** better on subset_20 than on train_data.

So, what does this mean?

- The points with higher weights are the ones that are more important during the training process of the weighted decision tree.
- The points with zero weights are basically ignored during training.

Quiz Question: Will you get the same model as **small_data_decision_tree_subset_20** if you trained a decision tree with only the 20 data points with non-zero weights from the set of points in **subset_20**?

Implementing your own Adaboost (on decision stumps)

19. Now that we have a weighted decision tree working, it takes only a bit of work to implement Adaboost. For the sake of simplicity, let us stick with decision tree stumps by training trees with max_depth=1.

Recall from the lecture the procedure for Adaboost:

* Start with unweighted data with α_j = 1

* For t=1,...,T:

- Learn f_t(x) with data weights α_j
- Compute coefficient ŵ_t:

$$\hat{w}_t = \frac{1}{2}\ln\left(\frac{1-\mathrm{E}(\alpha,\hat{\mathbf{y}})}{\mathrm{E}(\alpha,\hat{\mathbf{y}})}\right)$$

- Re-compute weights α_j

$$\alpha_j \leftarrow \begin{cases} \alpha_j \exp\left(-\hat{w}_t\right) & \text{if } f_t(x_j) = y_j \\ \alpha_j \exp\left(\hat{w}_t\right) & \text{if } f_t(x_j) \neq y_j \end{cases}$$

- Normalize weights α_j

$$\alpha_j \leftarrow \frac{\alpha_j}{\sum_{i=1}^{N} \alpha_i}$$

Now write your own Adaboost function. The function accepts 4 parameters:

- data: a data frame with binary features
- features: list of feature names
- target: name of target column
- num_tree_stumps: number of tree stumps to train for the ensemble

The function should return the list of tree stumps, along with the list of corresponding tree stump weights.

It should be analogous to the following code skeleton:

```
from math import log
from math import exp

def adaboost_with_tree_stumps(data, features, target, num_tree_stumps):
    # start with unweighted data
    alpha = graphlab.SArray([1.]*len(data))
    weights = []
    tree_stumps = []
    target_values = data[target]

    for t in xrange(num_tree_stumps):
        print '======================================================'
        print 'Adaboost Iteration %d' % t
        print '======================================================'
        # Learn a weighted decision tree stump. Use max_depth=1
        tree_stump = weighted_decision_tree_create(data, features, target, data_weights=alph
a, max_depth=1)
        tree_stumps.append(tree_stump)

        # Make predictions
        predictions = data.apply(lambda x: classify(tree_stump, x))

        # Produce a Boolean array indicating whether
        # each data point was correctly classified
        is_correct = predictions == target_values
        is_wrong   = predictions != target_values
```

```
        # Compute weighted error
        # YOUR CODE HERE
        weighted_error = ...

        # Compute model coefficient using weighted error
        # YOUR CODE HERE
        weight = ...
        weights.append(weight)

        # Adjust weights on data point
        adjustment = is_correct.apply(lambda is_correct : exp(-weight) if is_correct else exp
(weight))

        # Scale alpha by multiplying by adjustment
        # Then normalize data points weights
        ## YOUR CODE HERE
        ...

    return weights, tree_stumps
```

Reminders

- Stump weights ($\hat{w}$) and data point weights (α) are two different concepts.
- Stump weights ($\hat{w}$) tell you how important each stump is while making predictions with the entire boosted ensemble.
- Data point weights (α) tell you how important each data point is while training a decision stump.

Training a boosted ensemble of 10 stumps

20. Let us train an ensemble of 10 decision tree stumps with Adaboost. We run the **adaboost_with_tree_stumps** function with the following parameters:

- train_data
- features
- target
- num_tree_stumps = 10

Making predictions

21. Recall from the lecture that in order to make predictions, we use the following formula:

$$\hat{y} = sign\left(\sum_{t=1}^{T} \hat{w}_t f_t(x)\right)$$

Do the following things in a new function **predict_adaboost**:

- Compute the predictions f_t(x) using the t-th decision tree
- Compute ŵ_t f_t(x) by multiplying the stump_weights with the predictions f_t(x) from the decision trees
- Sum the weighted predictions over each stump in the ensemble.

In the end, your **predict_adaboost** should be analogous to this Python function:

```
def predict_adaboost(stump_weights, tree_stumps, data):
    scores = graphlab.SArray([0.]*len(data))

    for i, tree_stump in enumerate(tree_stumps):
        predictions = data.apply(lambda x: classify(tree_stump, x))

        # Accumulate predictions on scaores array
        # YOUR CODE HERE
        ...

    return scores.apply(lambda score : +1 if score > 0 else -1)
```

Use this function to answer the following question:

Quiz Question: Are the weights monotonically decreasing, monotonically increasing, or neither?

Reminder: Stump weights (ŵ) tell you how important each stump is while making predictions with the entire boosted ensemble.

Performance plots

How does accuracy change with adding stumps to the ensemble?

22. We will now train an ensemble with:

- train_data
- features
- target
- num_tree_stumps = 30

Once we are done with this, we will then do the following:

- Compute the classification error at the end of each iteration.
- Plot a curve of classification error vs iteration.

First, let's train the model.

Computing training error at the end of each iteration

23. Let us compute the classification error on the **train_data** and see how it is reduced as trees are added.

For n = 1 to 30, do the following:

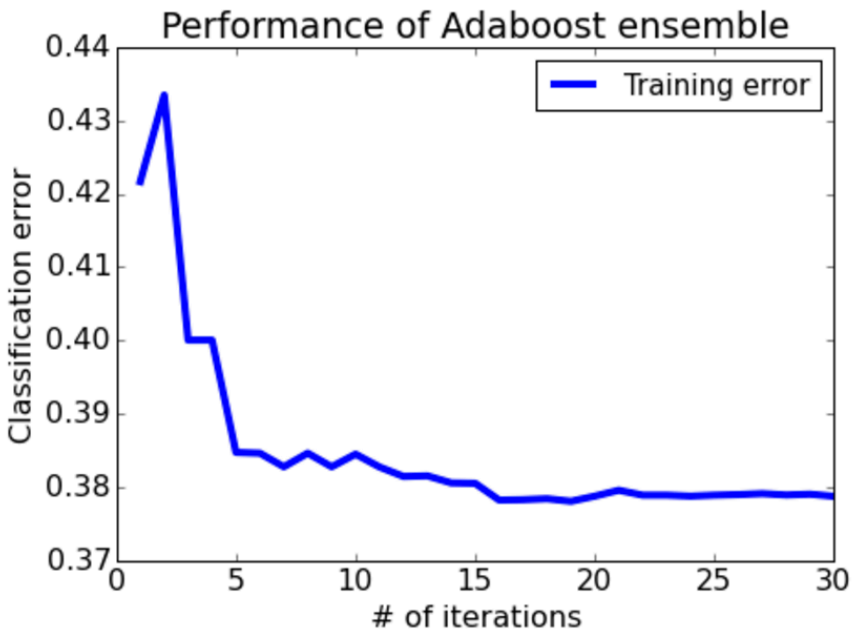
- Make predictions on **train_data** using tree stumps 0, ..., n-1.
- Compute classification error for the predictions
- Record the classification error for that n.

The loop should be analogous to the following:

```
error_all = []
for n in xrange(1, 31):
    predictions = predict_adaboost(stump_weights[:n], tree_stumps[:n], train_data)
    error = 1.0 - graphlab.evaluation.accuracy(train_data[target], predictions)
    error_all.append(error)
    print "Iteration %s, training error = %s" % (n, error_all[n-1])
```

Visualizing training error vs number of iterations

24. Let us generate the plot of classification error as a function of the number of iterations. Use the classification error values recorded in #23.



For inspiration, we provide you with matplotlib plotting code.

```
plt.rcParams['figure.figsize'] = 7, 5
plt.plot(range(1,31), error_all, '-', linewidth=4.0, label='Training error')
plt.title('Performance of Adaboost ensemble')
plt.xlabel('# of iterations')
plt.ylabel('Classification error')
plt.legend(loc='best', prop={'size':15})

plt.rcParams.update({'font.size': 16})
```

Quiz Question: Which of the following best describes a **general trend in accuracy** as we add more and more components? Answer based on the 30 components learned so far.

- Training error goes down monotonically, i.e. the training error reduces with each iteration but never increases.
- Training error goes down in general, with some ups and downs in the middle.
- Training error goes up in general, with some ups and downs in the middle.
- Training error goes down in the beginning, achieves the best error, and then goes up sharply.
- None of the above

Evaluation on the test data

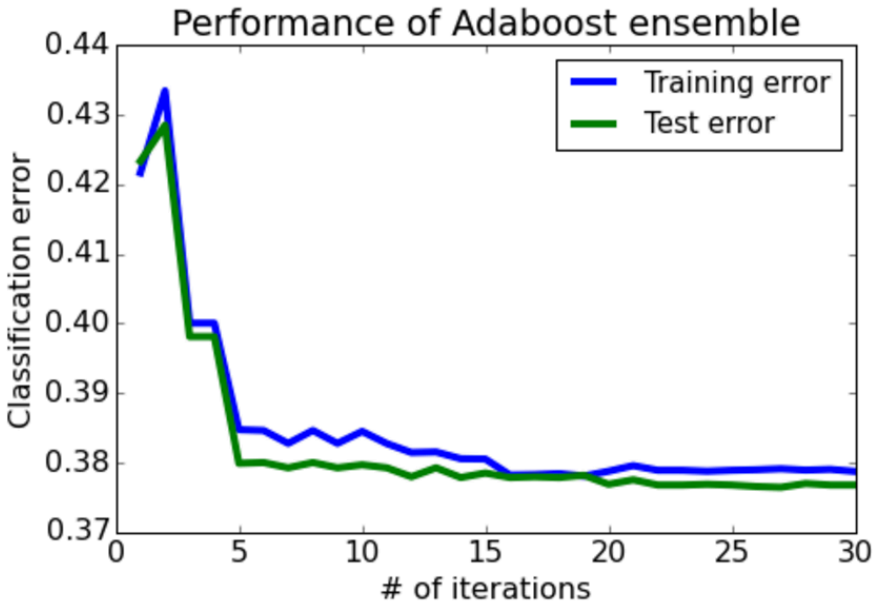
25. Performing well on the training data is cheating, so lets make sure it works on the test_data as well. Here, we will compute the classification error on the test_data at the end of each iteration.

For n = 1 to 30, do the following:

- Make predictions on **test_data** using tree stumps 0, ..., n-1.
- Compute classification error for the predictions
- Record the classification error for that n.

Visualize both the training and test errors

26. Let us plot the training & test error with the number of iterations.



Again, for inspiration, we provide you with matplotlib code.

```
plt.rcParams['figure.figsize'] = 7, 5
plt.plot(range(1,31), error_all, '-', linewidth=4.0, label='Training error')
plt.plot(range(1,31), test_error_all, '-', linewidth=4.0, label='Test error')

plt.title('Performance of Adaboost ensemble')
plt.xlabel('# of iterations')
plt.ylabel('Classification error')
plt.rcParams.update({'font.size': 16})
plt.legend(loc='best', prop={'size':15})
plt.tight_layout()
```

Quiz Question: From this plot (with 30 trees), is there massive overfitting as the # of iterations increases?