APC 524 : Final Report: Parallelized Boosting

Mehmet Basbug (mbasbug@cs.princeton.edu)
Burcin Cakir (bcakir@princeton.edu)
Ali JavadiAbhari (ajavadia@princeton.edu)

Date: 18 Jan 2013
1. Background & Overview

The underlying idea behind boosting is to harness the wisdom from a council of fools. In real life, it seems highly unlikely that a committee of blockheads can somehow arrive together at a highly reasoned decision; but this strategy turns out to be very useful in machine learning.

Let's say we have the following data and labels (whether the patient has diabetes) and our goal is to come up with a classifier which will predict if any given patient has the disease.

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Height (cm)</th>
<th>Weight (kg)</th>
<th>Day 1 BGL-m</th>
<th>Day 1 BGL-a</th>
<th>Day 1 BGL-e</th>
<th>Day 2 BGL-m</th>
<th>Day 2 BGL-a</th>
<th>Day 2 BGL-e</th>
<th>Has Diabet es</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>45</td>
<td>155</td>
<td>100</td>
<td>4.1</td>
<td>1.1</td>
<td>4.2</td>
<td>4.3</td>
<td>1.5</td>
<td>4.6</td>
<td>yes</td>
</tr>
<tr>
<td>Bob</td>
<td>35</td>
<td>178</td>
<td>82</td>
<td>6.2</td>
<td>3.5</td>
<td>8.1</td>
<td>5.4</td>
<td>3.3</td>
<td>7.2</td>
<td>yes</td>
</tr>
<tr>
<td>Cynthia</td>
<td>21</td>
<td>161</td>
<td>55</td>
<td>5.7</td>
<td>2.1</td>
<td>3.2</td>
<td>3.6</td>
<td>2.2</td>
<td>4.6</td>
<td>no</td>
</tr>
<tr>
<td>Drew</td>
<td>67</td>
<td>182</td>
<td>78</td>
<td>4.5</td>
<td>3.1</td>
<td>3.7</td>
<td>5.8</td>
<td>3.3</td>
<td>5.6</td>
<td>no</td>
</tr>
<tr>
<td>Elizabet h</td>
<td>53</td>
<td>181</td>
<td>77</td>
<td>9.0</td>
<td>6.7</td>
<td>10.9</td>
<td>7.7</td>
<td>6.7</td>
<td>8.8</td>
<td>yes</td>
</tr>
<tr>
<td>Fabian</td>
<td>28</td>
<td>198</td>
<td>115</td>
<td>5.5</td>
<td>2.1</td>
<td>8.2</td>
<td>6.1</td>
<td>2.4</td>
<td>6.8</td>
<td>no</td>
</tr>
<tr>
<td>George</td>
<td>25</td>
<td>172</td>
<td>105</td>
<td>7.2</td>
<td>3.1</td>
<td>3.6</td>
<td>7.8</td>
<td>4.5</td>
<td>4.1</td>
<td>yes</td>
</tr>
<tr>
<td>Helena</td>
<td>48</td>
<td>155</td>
<td>71</td>
<td>5.5</td>
<td>5.6</td>
<td>11.1</td>
<td>5.1</td>
<td>4.8</td>
<td>8.9</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 1 Raw data (BGL,m,a,e refers to Blood Glucose Levels in the morning, afternoon and evening, respectively)

Using this raw data, we might form simple hypotheses such as

- h1: If weight is more than 76, patient has diabetes
- h2: If Day1 BGL-m is more than 6.0, patient has diabetes
- h3: If patient is older than 34, patient has diabetes

If we look at the performance of these hypotheses, we'll see that

- h1 is correct on 5 out of 8 examples, which means that err(h1) = 0.375
- h2 is correct on 6 out of 8 examples, which means that err(h2) = 0.250
- h3 is correct on 5 out of 8 examples, which means that err(h3) = 0.375

If we apply boosting for 3 rounds using these hypotheses, we will end up with the following classifier:

\[ H(x) = \text{sign}(0.5493*h2(x) + 0.1682*h1(x) + 0.7878*h3(x)) \]

which has 0.125 error.

In short, boosting provides a clever method of combining simple predictors in a way that the error on the data set is minimized. However, if we can incorporate human knowledge into the algorithm we might achieve further improvements. One way of doing that is to come up with better simple predictors. Continuing with the example, instead of considering blood glucose level on the first day’s morning we might want to make a prediction based on the difference between afternoon and morning levels, or better we might take the average of this difference for two days. Another hypothesis can be to consider body mass index rather than weight itself, this would eliminate the height bias. If we were to have BGL levels
for hundreds of days this approach would make more sense. In machine learning terminology, this is called dimension reduction. However this kind of transformation (from raw data to more useful simple predictors) does not have to reduce the number of hypothesis. For instance if we have ten measurements per day and we don’t have a clear idea of which pair difference is useful; we may want to consider every pair of measurement per day which would increase the number of hypothesis we have. In the most generic case, user has the ability to define any deterministic function over the columns of raw data (called “attributes”). There is also no condition that each of these hypotheses should have at least 0.5 error. The only condition we have is not to have a random prediction. In our implementation, we will also have another condition that each hypothesis should have a transitivity property such that we can have a threshold associated with this function when we are making our prediction. Predictions and labels do not have to be binary as well but the labels should be either binary or discrete. Predictions can be discrete, continuous or binary.

Returning to the boosting algorithm, specifically AdaBoost which is by far the most popular family of boosting algorithms, we start with a probability distribution over examples and in each round we are trying to find the best hypothesis given that distribution and using the predictions of that hypothesis we are updating the distribution such that incorrectly predicted examples are given more weight. In this way, for the next rounds the algorithm will focus on harder examples. We repeat this procedure for a certain number of rounds, and finally form a linear combination of best hypothesis found in each round. As you might guess, the most time consuming step is to search for the best possible hypothesis in the given hypothesis space. Fortunately this step is parallelizable if we can somehow partition the hypothesis space.

2. Project Description

For the purpose of this project, implementations of boosting algorithms is be realized in a parallelized manner. Given a dataset (training set) and its corresponding results (labels), a chosen boosting algorithm will be used to reach a predictor (strong learner). Each example is expected to have certain number of attributes.

The user of the program is expected to provide behavioral function classes, which inherit from an abstract class provided by us, to generate different hypothesis. These numerous hypotheses are applied over the attributes of each example in the given data. If users do not provide their own hypotheses, the main program will use its own default generator, where each single attribute in the dataset is used as a hypothesis. This stage is part of a pre-processing phase, explained in section 3.2.

The program then starts executing on a “master” core, while using the help of other cores as “slaves” each time there is a resource-intensive computation to be performed. The details of this interaction are explained in section 3.3. After the execution is over, the best classifier is generated, similar to what we have in the example above.

One important point to address here is what is called cross-validation. It is a common practice in machine learning to train algorithms on some part of the data and test its performance on the remaining part. One way of doing that is to partition the whole training data into k “folds,” holding out one fold at a time and train k different algorithms on the remaining folds to test on the held out part of the data. We also need to train the algorithm on the whole dataset. Therefore we need to perform this step k+1 times. It is noteworthy that each of these training steps are independent, therefore can be done in parallel if there
is enough resources. There are different versions of boosting algorithms with some variations in their method of weighing training examples and hypotheses. For our project AdaBoost is considered to be the default boosting algorithm.

Finally, for its output, our program is provide the user with a few useful information. It plots the training error versus the number of iterations, which shows an optimal number of rounds to iterate. It also plots ROC curve to have a better understanding of where the classifier might indicate a false positive, a false negative, etc. As a final feature, the user can provide the path to a test dataset in the configuration file, and our algorithm applies the trained predictor to that data and reports the test error. These results is generated in a post-processing phase, explained in section 3.4.

3. Structure

There are three main steps in the implementation of boosting.

In the first step, we start with forming all possible hypotheses specified by the user. Each of these hypotheses is applied over the training data. The results and the sorting indices of each example are stored for each classifier. In the second step, we train the boosting algorithm. In the third step, we collect the results, specifically labels assigned to each example. Using these results we report the performance of the algorithm and some other metrics to consider.

3.1 Utilities Module

There are a few small modules that are useful in our program’s execution, which we refer to as utilities. They are not part of the core of boosting but are necessary for the program to run successfully. One of these modules is the configuration parser, which takes one of the configuration numbers as an input (assuming the user would like the program run with multiple configurations), and produce a dictionary of items, including the number of rounds, the number of cores, etc. This parsed information would be passed around to all other modules during the execution of the program. The parser catches many errors in the configuration file and reports to the user, such as non-sensible RAM values (maybe the user forgot to report them in Gigabytes), or non-integer number of rounds or cross-validation indices. The parser’s job is to catch bugs at the configuration stage and report a clean dictionary of configuration items for later use.

Another form of utility are the abstract class for behavioral functions (hypotheses), and their default implementation in case this information is not given from the user’s side. The default implementation is that we assume each column of the data set (every attribute) to form a naive hypotheses. This is the most raw form of hypotheses forming. Our abstract class also provides a method, insert, which the user can just call in every class implementation, to automatically insert different behaviors into the behavior table. Section 4 contains more details about these configuration files and the abstract behavioral function classes.

3.2 Preprocessing Module

For the preprocessing part, the user is expected to give a training data and abstract function definitions to generate the hypotheses that will be used by the weak learner. The program reads a configuration
file provided by the user to get the path of the required data and definitions. This file also specifies the necessary information about number of nodes, the number of cores per node and maximum available memory for the program to parallelize the code and the cross-validation number. A sample configuration file is illustrated in the Section 4.

The user provides python classes that define different behaviors. By specifying different arguments for a particular behavior, the program obtains a set of hypotheses. To store these hypotheses and make it available to different cores, a table is created in the shared disk (see Table 2.). Each hypothesis is represented as one of the entries of that table. Behavioral class that each particular hypothesis belongs to, and the arguments of these hypotheses are stored as the columns of this table along with an id for each hypothesis. One efficient way of storing generic arguments is to use json format. Dictionary of arguments is converted to a json object and then a string such that it would be easier to store them in a table. To access a hypothesis, the program reads its behavioral class and the string representation of its arguments. (See Section 4 for more details).

<table>
<thead>
<tr>
<th>Hypothesis id</th>
<th>Behavioral Class</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;my_behavioral.exponential&quot;</td>
<td>&quot;{attr_1:3,'power':6}&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;my_behavioral.polynomial&quot;</td>
<td>&quot;{attr:1,'coeff1':3,'coeff2':6}&quot;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N</td>
<td>&quot;my_behavioral.cosine&quot;</td>
<td>&quot;{attr:5,'amplitude':3,'frequency':100}&quot;</td>
</tr>
</tbody>
</table>

Table 2. Function Definitions Table

Once, the hypotheses are obtained, we apply them over the attributes of the examples to create a hypothesis result table as seen in Table 4. The shape of a simple training set which is read during calculation process is illustrated in Table 3, which is a general form of the example data presented Table 1.

<table>
<thead>
<tr>
<th>Attribute 1</th>
<th>Attribute 2</th>
<th>...</th>
<th>Attribute N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Example data format
Table 4. Function Output Table

<table>
<thead>
<tr>
<th>Function 1</th>
<th>Function 2</th>
<th>Function 3</th>
<th>Function N</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>99</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>M-1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>M</td>
<td>165</td>
<td>17</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5. Sorting Index Table

Applying each function to each example is a parallelizable task, and actually, requires the use of parallelization for large data sets. Therefore, another important step that needs to be implemented in the pre-processing part is to read the machine information from the configuration file provided by the user and initialize the parallelization of the code. The partition is done over the functions. Each core applies the functions assigned to them over all examples, and saves the results in a function table. Then, these values are used to create a new table where the sorting indices of each example are saved (see Table 5). These two tables are stored in the disk with hdf5 file system. Since Python has an interface to hdf5, for coding part of the project, it is thought to be easy to interact with files and datasets using traditional Python and NumPy calls.

As the last step, if the cross-validation is enabled, the xval number is read from the configuration file and a new column is added to data file indicating the fold numbers.

### 3.3 Boosting Module

The main part of our program is the boosting module. As described in the sections above, our final goal is to discover a linear combination of the given simple hypotheses that would result in a more complex and well-performing predictor for future test cases. This process involves training a boosting algorithm in an iterative fashion.

The inputs to this module are pboost object containing global variables, some outputs from the preprocessing module, and the ranks of master and slaves.

A part of this process involves what is called a “weak learner”, the discovery of the best performing hypothesis (function of attributes) among the many hypotheses at hand. The input matrices are usually huge, and this task easily ends up being the most resource-intensive and time-consuming of the whole program, and is thus the motivation for our parallelization effort. This parallelization can be accomplished
by dividing the set of independent hypotheses among different computation nodes, where each node will then be responsible for performing the weak learner on its assigned hypotheses, returning what it deems as the best one. At the end, the master node chooses the very best hypotheses among all of them, and that would be the nominated hypotheses for that round.

The actual procedure of parallelization is as follows: We assign one core as the master, and the other additional cores as slaves. The master uses the slaves to perform the job, but holds the ultimate responsibility for collecting the data, updating the information on each round, etc. It starts with a uniform probability distribution over the training examples. It then broadcasts this distribution to all of its slaves, waiting for them to return with their best hypothesis, its error, labeling, and the threshold used for labeling. In the first iteration of each slave, a new sorting index matrix is created for two reasons: to slice the original matrix column-wise to operate only on a certain subset of hypotheses and eliminate the cells that refers to test examples for that particular cross validation process. To find the best hypotheses, they then run the weak learner algorithm. After best hypothesis, threshold, its error and labeling have been communicated to master by every slave, the master picks the best hypotheses by looking at the measured errors, and assigns a coefficient to the chosen hypotheses. This coefficient is used in the final linear combination. Using this new information, the master then re-calculates the probability distribution of examples to give higher emphasis to poor-performing examples, which improves in their prediction at future stages. This whole procedure is repeated for the given number of rounds. Since one of the goals of our program is to plot the training error versus the number of rounds, the master calculates and stores the example labelings at each round too.

The output of this module is the final classifier (predictor), as a pair of hypotheses and linear coefficients. We also report our predictions for validation/training/test examples. In the end, we emphasize that this whole procedure is repeated for different cross-validation folds, which we compare at the post-processing phase.

Our implementation of boosting is faster than traditional implementations. Especially when the size of the hypotheses space is large, pboost comes really handy. There are couple of tricks we used:

1. Single Pass Through Data : We are passing through data only once and storing relevant information about each example in the disk.
2. Storing Indexes Not Results : Instead of using the result of each function applied to each example, we are using the sorting indexes for each function which represents all the necessary information and occupy a smaller memory space. This is especially important when the number of hypotheses is large. By storing indices which usually takes 16 bit instead of function results (64 bit), we are saving a lot of memory space. Since our limitation for the number of hypotheses a single core can process comes from its memory restrictions, this can decrease the required number of cores by 4 times, or possibly speed up the total runtime by 4 times.
3. Memory Based Computation : The program is optimized to use memory based computations whenever possible. Each slave is responsible only a certain part of the hypotheses space. The sorting index matrix, which is necessary to find the best hypothesis, is read from disk only once and kept in slave’s memory for subsequent calls. In traditional implementation of boosting, boosting is strictly separated from weak learner to provide the flexibility of specifying any machine learning algorithm as a weak learner. Our approach is slightly different. We are trying to optimize the compact implementation of boosting with a specific type of weak learner called decision stumps. This is especially useful for very large hypotheses spaces or time-series data. However, due to our flexible implementation of decision stumps; it is possible to implement arbitrary functions which allow user to implement many of the geometric machine learning algorithms as weak learner. With our extended decision stump, it is possible to implement k-level decision trees, SVMs, Naive Bayes Classifiers etc as weak learner.
4. LAPACK & numexpr: We tried to avoid every ‘for’ loop possible in computationally intensive part of the code and converted all these computations with linear algebra operations which can be performed with LAPACK and its wrapper numexpr package in python. Specifically we used that package in our computation of error matrices.

3.4 Post-processing Module

This module is responsible for getting the gathered data produced by the boosting module, and create meaningful results to report to the user. The results, consisting of a selected hypotheses and its respective coefficient, are stored in the master. Among what is created are:

1) Confusion Matrix:
The predictions that the classifier produces is at first a continuous number, and we only attribute binary (0/1) labels to them based on a certain chosen threshold. This threshold is up to us, and choosing different ones would result in different label assignments by the classifier. The most ordinary form of threshold is 0, which would correspond to label 0 being assigned to negative predictions and label 1 to positive predictions. The confusion matrix is a 2x2 matrix that shows the ratio of true positives, false positives, true negatives and false negatives in our classifier, for this threshold of 0. This can be thought of a particular case of what follows, as the ROC curve.

2) ROC Curve of Training and Testing Error:
ROC (Receiver Operating Characteristic) curve is a visualization of the reliability in classifiers’ performance. It sweeps the range of thresholds that we can choose for attributing binary labels to continuous prediction ranges, and shows the True Positive Rate (“sensitivity”) versus the False Positive Rate (1 - “specificity”). This is a form of trade-off in classifiers. Based on the application, we may choose to have more true positives at the expense of more false positives, or vice versa. For example, for a military radar classifier, it might be acceptable to tolerate some false positives, but a very-low false negative rate is desirable. Similar decisions may be made for other classifiers, and the ROC curve helps visualize this trade-off for that particular classifier.
“pyroc” was an open-source package that was used for this purpose:
https://github.com/marcelcaraciolo/PyROC

3) Plots for Training, Testing and Validation Errors vs. Number of Rounds:
Error is the ratio of incorrect predictions when applying our classifier to some dataset. “Classifier” means the predictions that would be made using the combination of hypotheses found up to that particular round. Specifically, training error finds the ratio of discrepancies between the labels and predictions on the training dataset, using the whole dataset as visible to the boosting algorithm. Conversely, validation prediction finds the same ratio, but using the cross-validation indices to hide a part of the data-set and train on the rest of it. Finally, test prediction apply the former classifier, but to a separate test dataset. Plotting these errors against the number of rounds shows how the classifier’s performance changes as we train the algorithm more and more. Usually, it saturates after some number of rounds. Threshold 0 is used for computing labels in these cases.

*Important note about the plots:
The plots are presented using the matplotlib package, which offers an interactive presentation of the figures where the user can read the axis values of each point using the cursor. This method was chosen over the non-interactive form of PDF output, because it is specially important for cases of ROC curve to see the exact trade-off. The downside to this is that the program does not terminate until the user
manually closes the figures to return control to the command-line, because the python interpreter itself is used to show the graphs. For places where it is important to terminate the program, the user can set the "show_plots" option to "n" in the configuration file. This will cause the program to only save the data needed for plotting on disk, and terminate naturally. This data can later be used to reproduce the graphs, as explained in the “Dump File” section below.

4) Final Classifier Script:
We create a directory of **final_classifier**, which contains, among other things, a portable script that can be used on other computers to apply the trained classifier to various different datasets. This is useful when training of a certain boosting classifier takes a long time, and we would like to be able to apply the final classifier to a set of other datasets as well without going through the problem of training the classifier again. Having the **parallel_boosting** package installed on the computer, the user can simply use the following command to apply the classifier to a new example:

$ python final_classifier.py example_data.npy

5) Dump file of the information for reproduction of results:
Finally, the post-processor creates a dump file which contains information about configuration, predictions, etc. that can be used to reproduce the graphs on another computer. This is specially useful when the training is run on a server with no screen. In this case, the user can just take this dump file, and call the “plot_data” function of the post-processor on it. This stand-alone function would create the graphs.

$ python
$ from parallel_boosting.post_process.process import plot_data
$ plot_data(filename = 'dump.npz')

4. User Interface
The driver for our **parallel_boosting** package, **pboost**, operates from the command line. Per the design document, we did not plan on having any form of GUI, except for the plots at the end. The user will specify a configuration in a .cfg text file. The configuration will include things like workspace directory, path to data, path to behavioral function file, cross validation, number of rounds, results to report etc. An example configuration looks like:

```
# configurations.cfg
[Configuration 1]
data_file = diabetes.dat
test_file
fn_behaviors = behaviors_diabetes.py
algorithm = conf-rated
rounds = 1000
xval_no = 10
working_dir = ../example/
max_memory = 2
show_plots = y
```
An example behavior class implementation in the behaviors file looks like this:

```python
<behaviors_diabetes.py>

class BGL_Time_Period_Av(Behavioral):
    def behavior(self,bgl_list):
        s = zeros(self.data.shape[0])
        for bgl in bgl_list:
            s = s + self.data[:,bgl]
        return s/len(bgl_list)

    def fn_generator(self):
        bgl_m = list()
        bgl_n = list()
        bgl_e = list()
        for k in range(1,(self.data.shape[1]-4)/3+1):
            bgl_m.append(3*k)
            bgl_n.append(3*k+1)
            bgl_e.append(3*k+2)
        self.insert(bgl_m)  # average of all morning BGL levels
        self.insert(bgl_n)  # average of all noon BGL levels
        self.insert(bgl_e)  # average of all evening BGL levels
```

Users can define several Behavioral classes in this way. In this example, the behavior of these set of hypotheses is to take the average of morning, noon and evening glucose levels for all days and insert them into the table of hypotheses. Behavioral.insert is a method that we provide to the user. In general users can define several classes each of them inheriting from the Behavioral abstract class in the same file.

5. Module Interfaces

This section describes how different modules of the parallel_boosting package, as well as those defined by the user in the working_directory, interface with each other.
In Figure 1, the structure of our project is presented. Pre_processing, boosting and post_processing are the three subpackages which together with the main module (pboost) construct the project package (*parallel_boosting*). Pboost calls them by order to present the boosting output according to the data given by the user.

The following is a brief description of these dependencies:

1) *Pboost* is the module which includes the main routine of the project. The parallelization environment is set up in this module. Therefore, to initialize each submodule, a pboost object is passed to each subpackage.

2) *Pre_processing* module reads the necessary information from the provided data and prepares a meta dictionary which will be used by pboost module. This meta dictionary includes number of examples in training and test data, total hypothesis number, labeling information, assigned indices for cross-validation and the partition of the hypothesis space for the parallelization.

3) *Pboost* passes this meta data to *boosting* and *post_processing* module by creating corresponding objects and starts the boosting algorithm.
4) **Post_processing** reads the output of boosting module (train predictions, test predictions, hypotheses and the validation predictions) from the ‘disk’ and outputs the resulting plots.

5) **Utility** is the common module which includes the helper methods.
   a. All user-defined behaviors inherit from a *Behavioral* class, which acts as a base abstract class. This is defined in the utilities, in the file “behavior.py”. This contains the data and its path as attributes, which can be read in the implementations by the user.
   b. **Parser** module is called from pre_processing module. It reads the configuration file specified by the user and returns a dictionary.
   c. **Test_helpers** includes the necessary classes for the subpackage tests and called by these files to generate test data.

We also provide a setup.py file to enable users install our project as a regular python library in their machines. By running this script, the package is installed in the site-packages directory of the active python version and the directory of the package will be appended to the pythonpath. Therefore, the user will not have any problem with importing any module. In order the create the distribution package (*parallel_boosting*), the following command should be run.

```bash
$ python setup.py sdist
```

Once the tarball is obtained (in a folder called ‘dist’ in the same directory with the setup script), one can extract it and complete the installation with:

```bash
$ python setup.py install
```

The version information that will also be useful for the future developments is included in the __init__.py file of the *parallel_boosting* package.

6. **Experimentation Results**

When running our code on a test dataset of diabetes patients (synthetic), we get the following results:

- **Info**: Confusion matrix for combined validation error with zero threshold:
  ```
  {'FP': 39, 'TN': 61, 'FN': 15, 'TP': 85}
  ```

- **Info**: Confusion matrix for testing error with zero threshold:
  ```
  {'FP': 17, 'TN': 83, 'FN': 16, 'TP': 84}
  ```

This show how many false positives, true positives, etc. we get for a threshold of 0, in both our training (with validation) and testing datasets.

The following figure shows ROC curves. We see a clear trade-off between having more true positives and less false positives (both desirable). We see that for example, in the validation dataset, in order to get 90% true positives, we must tolerate around 49% of false positives.
Errors vs. Number of rounds (500 total) is shown in the following figure. We see that the training error goes to zero as we train more and more. This is expected as we are fitting our model better to the dataset at hand. Test error on the other hand is saturating somewhere around 16%. 10% of this is inherent in the model that we synthesized, and this shows our classifier, when fully trained, does pretty well to have only 16% error rate. Validation error is an expected value for testing error, with usually a bias. It is useful when we do not have a test dataset at hand and want to predict its error. Our approximation ends up being 28% compared to 16%, which is understandably worse.
7. Profiling and Tuning

For profiling purposes and identifying the bottlenecks of our algorithm, we create a dataset of 1000 examples and 400 attributes, which is quite large (183MB).

Below is a sample of the output from our profiler, which shows the function calls sorted by the total time they take.
We can see that based on cumulative time, `pboost.py:1` is essentially equal to the runtime of the whole program. The module that takes the most time is evidently the boosting module. It takes 2479 out of 2038 seconds, or 81% of the runtime. The difference between these two numbers is what is spent on preprocessing and postprocessing all together. From this, most of the time (90%) is spent executing the confidence-rated module. The rest is spent on mpi communication.

As the number of attributes gets very large, we end up with a lot (maybe around a million) of hypotheses that need to be inserted into the hypotheses table. We discovered that this would become a bottleneck for large numbers of attributes, and redesigned the insertion method so it does not increment table size each
time, but doubles it every time it fills.

8. Conclusion and Future Directions

In this project, we implemented a fairly large package with many features to perform boosting in parallel, on many number of cores, and for large datasets. In this way we were able to deal with very large datasets (millions of entries), which otherwise would not have been possible. The advantage of our method is in its runtime, because we used a memory-based method and did not store intermediary results on disk.

We implemented two boosting algorithms: adaboost and confidence-rated. For the future, there is the opportunity to expand these and try more boosting algorithms. The weak-learner discovery method can also be expanded to include SVM (support vector machine), etc.