Homework 6: Decision Trees and Boosting

Due: Friday, April 22nd, 2022 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better. **The optional problems should not take you too much time and help you navigate the material, consider taking a shot at them.**

---

1 Decision Tree Implementation

In this problem we'll implement decision trees for both classification and regression. The strategy will be to implement a generic class, called `Decision_Tree`, which we'll supply with the loss function we want to use to make node splitting decisions, as well as the estimator we'll use to come up with the prediction associated with each leaf node. For classification, this prediction could be a vector of probabilities, but for simplicity we'll just consider hard classifications here. We'll work with the classification and regression data sets from previous assignments.

1. Complete the `compute_entropy` and `compute_gini` functions.

2. Complete the class `Decision_Tree`, given in the skeleton code. The intended implementation is as follows: Each object of type `Decision_Tree` represents a single node of the tree. The depth of that node is represented by the variable self.depth, with the root node having depth 0. The main job of the fit function is to decide, given the data provided, how to split the node or whether it should remain a leaf node. If the node will split, then the splitting feature and splitting value are recorded, and the left and right subtrees are fit on the relevant portions of the data. Thus tree-building is a recursive procedure. We should have as many `Decision_Tree` objects as there are nodes in the tree. We will not implement pruning here. Some additional details are given in the skeleton code.

3. Run the code provided that builds trees for the two-dimensional classification data. Include the results. For debugging, you may want to compare results with sklearn's decision tree (code provided in the skeleton code). For visualization, you'll need to install `graphviz`.

4. Complete the function `mean_absolute_deviation_around_median` (MAE). Use the code provided to fit the `Regression_Tree` to the krr dataset using both the MAE loss and median predictions. Include the plots for the 6 fits.

2 Ensembling

Recall the general gradient boosting algorithm , for a given loss function ℓ and a hypothesis space $\mathcal{F}$ of regression functions (i.e. functions mapping from the input space to $\mathbb{R}$):

```
0: Initialize $f_0(x) = 0$.
1: For $m = 1$ to $M$:
    (a) Compute:
        $$\mathbf{g}_m = \left( \frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i)\right) \right)_{j=1}^{n}$$
    (b) Fit regression model to $-\mathbf{g}_m$:
        $$h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left( (-\mathbf{g}_m)_i - h(x_i) \right)^2 .$$
    (c) Choose fixed step size $\nu_m = \nu \in (0, 1]$, or take
        $$\nu_m = \arg\min_{\nu > 0} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i) + \nu h_m(x_i)\right) .$$
    (d) Take the step:
        $$f_m(x) = f_{m-1}(x) + \nu_m h_m(x)$$
3: Return $f_M$.
```

This method goes by many names, including gradient boosting machines (GBM), generalized boosting models (GBM), AnyBoost, and gradient boosted regression trees (GBRT), among others. One of the nice aspects of gradient boosting is that it can be applied to any problem with a subdifferentiable loss function.

Gradient Boosting Regression Implementation

First we'll keep things simple and consider the standard regression setting with square loss. In this case the we have $\mathcal{Y} = \mathbb{R}$, our loss function is given by $\ell(\hat{y}, y) = 1/2 \left(\hat{y} - y\right)^2$, and at the m'th round of gradient boosting, we have

$$h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left[(y_i - f_{m-1}(x_i)) - h(x_i) \right]^2 .$$

5. Complete the `gradient_boosting` class. As the base regression algorithm to compute the argmin, you should use sklearn's regression tree. You should use the square loss for the tree splitting rule (`criterion` keyword argument) and use the default sklearn leaf prediction rule from the `predict` method [1]. We will also use a constant step size ν.

6. Run the code provided to build gradient boosting models on the regression data sets `krr-train.txt`, and include the plots generated. For debugging you can use the sklearn implementation of `GradientBoostingRegressor`[2].

[1] Examples of usage are given in the skeleton code to debug previous problems, and you can check the docs `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html`

[2] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html`

Classification of images with Gradient Boosting

In this problem we will consider the classification of MNIST, the dataset of handwritten digits images, with ensembles of trees. For simplicity, we only retain the '0' and '1' examples and perform binary classification.

First we'll derive a special case of the general gradient boosting framework: BinomialBoost. Let's consider the classification framework, where $\mathcal{Y} = \{-1, 1\}$. In lecture, we noted that AdaBoost corresponds to forward stagewise additive modeling with the exponential loss, and that the exponential loss is not very robust to outliers (i.e. outliers can have a large effect on the final prediction function). Instead, let's consider the logistic loss

$$\ell(m) = \ln\left(1 + e^{-m}\right),$$

where $m = yf(x)$ is the margin.

7. Give the expression of the negative gradient step direction, or pseudo residual, $-\mathbf{g}_m$ for the logistic loss as a function of the prediction function f_{m-1} at the previous iteration and the dataset points $\{(x_i, y_i)\}_{i=1}^n$. What is the dimension of g_m?

   The dimension of g_m is $Dim(g_m) = n$, (aka: $g_m \in \mathbb{R}^n$) since there are n preditions corresponding to data points. Setting up our gradient as:

   $$-g_{m_i} = \ell(y_i, f_{m-1}(x_i))$$

   Where m is the variable which we differentiate with respect to, and i is the i^{th} entry of $-g_m$. The i^{th} entry in the negative gradient step direction is given by the following:

   $$-g_{m_i} = \frac{\partial \ell(m)}{\partial m} = \frac{y_i \times -e^{-y_i f_{m-1}(x_i)}}{1 + e^{-y_i f_{m-1}(x_i)}} = \frac{1}{1 + e^{y_i f_{m-1}(x_i)}}$$

   We showed what the negative gradient is for the i^{th} entry, here's the full thing:

   $$-g_m = \left(\frac{1}{1 + e^{y_i f_{m-1}(x_i)}}, \ldots, \frac{1}{1 + e^{y_n f_{m-1}(x_n)}}\right)$$

8. Write an expression for h_m as an argmin over functions h in $\mathcal{F}$.

   We need to minimize the following expression:

   $$h_m = \arg\min_{h \in H} \sum_{i=1}^{n} \left[-g_i - h_i(x_i)\right]^2$$

   Substitutign our identity we found in problem 7 for $-g_i$, we can write the expression for the argmin of h_m as follows:

   $$h_m = \arg\min_{h \in H} \sum_{i=1}^{n} \left[\frac{y_i}{1 + e^{-y_i f_{m-1}(x_i)}} - h_i(x_i)\right]^2$$

9. Load the MNIST dataset using the helper preprocessing function in the skeleton code.Using the scikit learn implementation of `GradientBoostingClassifier`, with the logistic loss (`loss='deviance'`) and trees of maximum depth 3, fit the data with 2, 5, 10, 100 and 200 iterations (estimators). Plot the train and test accurary as a function of the number of estimators.

Classification of images with Random Forests (Optional)

10. Another type of ensembling method we discussed in class are random forests. Explain in your own words the construction principle of random forests.

    Random forests is an ensemble method in which many trees are built and then used to create a final decision tree.

    To build random forests we take advantage of ensemble methods and bootstrapping.

    Ensemble methods are ML methods which combine many weak models into one powerful model. In our case, this means combining many trees (low bias, high variance) into a single tree that hopefully achieves lower variance, and higher accuracy on test set performance.

    Bootstrapping is a method in which we re-sample our data with replacement to simulate taking additional independent samples of a true underlying distribution from which we have attained our training data. Independence here is a strong word, as the bootstrapped samples are independent of the training data, but not the underlying distribution, $P_{\mathcal{X}\times\mathcal{Y}}$.

    The benefit of this, is that when we achieve some sample statistic θ, it is an unbiased estimator of the θ of the true underlying distribution. However, it also has some variance, σ^2, so to try to decrease the variance of our estimator, θ, we use bootstrapping which decreases the variance to $\frac{\sigma^2}{n}$, while not changing the expectation of our sample statistic.

    Our procedure to build a random forest is as follows:

    - Simulate data by using resampling methods like Bootstrapping.
    - Train many decision trees on separate portions of the simulated data (this can be done in parallel!). We restrict our choice of splitting variable to a randomly chosen subset of features of size m. This hopefully avoids the situation in which smaller trees are dominated by highly correlated features that explain most of the variance. A good size for m is $m = \sqrt{p}$, where p is the number of features.
    - Combine the output of our random forest model (the many trees we just created), by averaging the decision criteria or taking majority vote (the method you will use will depend on what prediction task you are trying to accomplish).

11. Using the scikit learn implementation of `RandomForestClassifier`[3], with the entropy loss (`criterion='entropy'`) and trees of maximum depth 3, fit the preprocessed binary MNIST dataset with 2, 5, 10, 50, 100 and 200 estimators.

[3] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier`

12. What general remark can you make on overfitting for Random Forests and Gradient Boosted Trees? Which method achieves the best train accuracy overall? Is this result expected? Can you think of a practical disadvantage of the best performing method? How do the algorithms compare in term of test accuracy?

Model Results:

Both models are prone to overfitting, with Gradient Boosted Trees overfittitng slightly more than the Random Forest model.

In the experiments we ran as part of the homework, both methods achieved very high test accuracy at 99.99%. However, Gradient Boosting Methods achieved 100% training accuracy, meaning it was more prone to overfitting, than Random Forest, which approach 99.98% (though that margin is razor thin).

This result is expected, as Gradient Boost continues to find functions that explain the variance of the residuals, and Random Forest can continually improve its performance as more estimators are added. That being said, for the dataset we used in the homework, the Random Forest model approached its highest test accuracy much quicker than Gradient Boosted Trees.

Disadvantages:

The fact that the Random Forest task can be computed in parallel is incredibly valuable. Gradient Boosted Trees have additional hyper-parameters like learning rate, hypothesis space, that must be tuned to work optimally, which can be a disadvantage. They also cannot be computed in parallel, and require a lot of CPU power.

For Random forest, you must choose how many features to include in each tree, which is another hyper-parameter you must tune (though $m = \sqrt{p}$ is used in practice mostly.

Both methods can suffer in interpretability, and the significance of each variable can be hard to gauge. Both models are also prone to overfitting.