Gradient Boosting and Ada boosting, a comparative study of machine learning Techniques

A progress report submitted for Info7017 Postgraduate Project B in partial fulfilment of the requirements for the degree of Master of Data Science

Supervisor: Dr Laurence Park

School of Computer, Data and Mathematical Sciences

Western Sydney University

April 2025

Introduction

Over the decades as technology advances, traditional programming has shown that they cannot handle modern datasets whether it be due to sheer variety, volume or complexity. This led to the development of machine learning by necessity, since then machine learning has become invaluable in various fields such as cloud computing and e-commerce, with every modern business and organizations participating in its use or development in some manner with continued technological advancements in computational power meaning it will stay relevant for decades to come.

Initially machine learning started off as a method to train artificial intelligence before splitting off into its own field and gained prominence with the rise of the internet, allowing it both to gain access to digital data and to spread its use across the world.

Unlike rigid rule-based systems there is room in machine learning models to continuously improve itself as they interact with more data, allowing them to become more accurate and effective over time, allowing them to take on problems that are too difficult to program explicitly.

Of course, machine learning itself is just a broad term to describe several self-learning methods that attempt to categorize, study and predict data such as decision trees, support vector machines and neural networks etc. While effective for a time, they were just the precursors to future techniques known as ensemble methods which aim to combine the strength of multiple simpler models to improve performance and accuracy. These developments soon led to ensemble machine learning models decisively outperforming traditional methods in classification and regression tasks. This is achieved by building models sequentially, where each new model attempts to correct errors made by previous models, leading to a powerful final model.

Through our project we wish to explore just how far ensemble methods have come and whether more traditional methods can still hope to compete, for now we will briefly cover a few of these methods.

The first model covered in our report, AdaBoost can be traced back to the early 1990s, AdaBoost was one of the first algorithms to demonstrate that weak learners (models slightly better than random guessing) could be combined into a powerful ensemble through a principled iterative procedure by building models sequentially, where each new model attempts to correct errors made by previous models, leading to a powerful final model (This method of combining multiple weak learners to create a single accurate model became known as boosting).

Adaboost had demonstrated a remarkable ability to reduce bias and variance while being relatively resistant to overfitting. Its relative simplicity and elegant formula garnered it great attention. A key feature of AdaBoost is how sample weights are updated to focus on hard-to-classify instances, showcasing a novel way of dealing with the limitations of weak classifiers.

Gradient Boosting, developed in the late 90s and early 2000s, extended the idea of boosting into the realm of numerical optimization. Instead of simply reweighting samples, Gradient Boosting interprets boosting as a gradient descent procedure in function space, aimed at minimizing a chosen loss function. This change extended its use into the realms of regression rather than just binary classification. Gradient Boost's flexibility to choose different loss functions, paired with the strength of decision trees as base learners, made it a significant leap forward in the world of data science.

Objective

The primary objectives of this project are:

1. To explain the theoretical foundations and differences between Boost and Gradient Boosting algorithms.
2. To implement algorithms using Python and apply them to multiple tasks in classification and regression. For this report we will just be using datasets sourced from the UCI Machine Learning Repository
3. Study the effects of different Loss functions in gradient boosting methods as well as other variables such as learning rate, iteration number etc.
4. To compare performance between gradient and non-gradient algorithms and see at what point non-gradient methods catch up to gradient methods. We will test their accuracy, computational complexity, and ability to handle real-world data.

Methodology

For this project to succeed, we must first gain a deeper understanding of algorithms Adaboost and Gradient boost relies on. As such we will present an in-depth examination of the two methods' theoretical foundations. As mentioned, we will heavily rely on Chapter 10 of *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (Second Edition) by Hastie, Tibshirani, and Friedman. These two algorithms will remain a focal part of our project for the foreseeable future.

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

This algorithm can be found in *The Elements of Statistical Learning: Data Mining, Inference, and Prediction,* it captures a faithful summary of how Adaboost functions.

- We initialize each weak classifier with weight 1/N as a starting point

- We train a weak classifier with the goal to minimize weighted error, samples that hard harder to classify are assigned higher weights which cause them to influence the training more

- We compute alpha, if there is no error, alpha approaches infinity (perfect classifier), if error is 0.5, alpha is 0 (random, flip of a coin basically) and higher erro means worse than random in the case of binary classification.

- We use alpha to assign weights to our classifiers so the worse the alpha for a classifier is, the more we focus on them

We emulate this algorithm in Python

```python
# AdaBoost
class AdaBoost:
    def __init__(self, n_clf=5):
        self.n_clf = n_clf
        self.clfs = []
        self.train_errors = []

    def fit(self, X, y):
        n_samples = X.shape[0]
        w = np.full(n_samples, (1 / n_samples))
        y_mod = 2 * y - 1  # Convert to {-1, 1}

        pred_sum = np.zeros(n_samples)

        for i in range(self.n_clf):
            clf = DecisionStump()
            clf.fit(X, y, w)
            predictions = clf.predict(X)
            pred_mod = 2 * predictions - 1

            # Update weights
            w *= np.exp(-clf.alpha * y_mod * pred_mod)
            w /= np.sum(w)
```

```
            self.clfs.append(clf)

            # Accumulate prediction and compute training error
            pred_sum += clf.alpha * pred_mod
            final_pred = np.sign(pred_sum)
            final_binary = (final_pred + 1) // 2
            error = 1 - accuracy_score(y, final_binary)
            self.train_errors.append(error)

    def predict(self, X):
        clf_preds = [clf.alpha * (2 * clf.predict(X) - 1) for clf in self.clfs]
        y_pred = np.sign(np.sum(clf_preds, axis=0))
        return (y_pred + 1) // 2
```

Important points in the code:

- In the Init function we state the number of weak classifers to use as well as create 2 lists, one to store trained classifers and the other to track training error (used for plotting).

- n_samples is the total number of trained examples, initial weight(w) is equal for all examples, y_mod is just to convert labels from {0,1} to {-1,1}

-pred_sum: running total of the weighted predictions over rounds of iterations

- for loop to loop over number of classifiers

-Decision stump is the classifier we are training

- we update weights with the exponential tern penalizing wrong samples by increasing the weight. The weight gets normalized so the sum is always 1

- each weak learner makes a prediction either –1,1. Weighted predictions are added up with the final prediction being based on the sign of the sum.

That same chapter also introduces us to the gradient boosting algorithm as well:

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

- 1 to M refers to number of boosting rounds
- r in 2.a refers to the negative gradients of the loss function
- A regression tree is fitted to the residuals, the tree partitions the input space in 'J' amount of regions, where the model makes the same correction in each region
- In 2.c we compute for each terminal region how much we should adjust the model's prediction
- In 2.d we add the new tree's contribution, generally this moves the prediction in whatever direction that reduces the loss the most
- After M rounds of boosting we release the final model which is the sum of all previous moves/corrections.

This is python code for Gradient boosting although it has been alterered for binary classification instead of regression.

```python
# Gradient Boosting Machine
class GBM:
    def __init__(self, n_estimators=10, learning_rate=0.1):
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.base_learners = []
        self.init_prediction = 0
        self.train_errors = []  # For plotting

    def fit(self, X, y):
        N = len(y)
        self.init_prediction = sum(y) / N
        current_preds = [self.init_prediction] * N

        for m in range(self.n_estimators):
            residuals = [y[i] - current_preds[i] for i in range(N)]
            stump = Stump()
            stump.fit(X, residuals)
            predictions = stump.predict(X)
```

```python
            current_preds = [
                current_preds[i] + self.learning_rate * predictions[i]
                for i in range(N)
            ]
            self.base_learners.append(stump)

            # Convert predictions to binary and record training error
            binary_preds = [1 if p >= 0.5 else 0 for p in current_preds]
            error = 1 - accuracy_score(y, binary_preds)
            self.train_errors.append(error)

    def predict(self, X):
        N = len(X)
        preds = [self.init_prediction] * N
        for stump in self.base_learners:
            stump_preds = stump.predict(X)
            preds = [preds[i] + self.learning_rate * stump_preds[i] for i in range(N)]
        return preds
```

Important points:

- Just like with ada we use init to initialise a few values, n_estimator refers to number of weak learners to train, base_learners which store trained stumps, init_prediction is the initial guess and train_errors to store training error for plotting purposes
- Current_preds stores running predictions over the training data

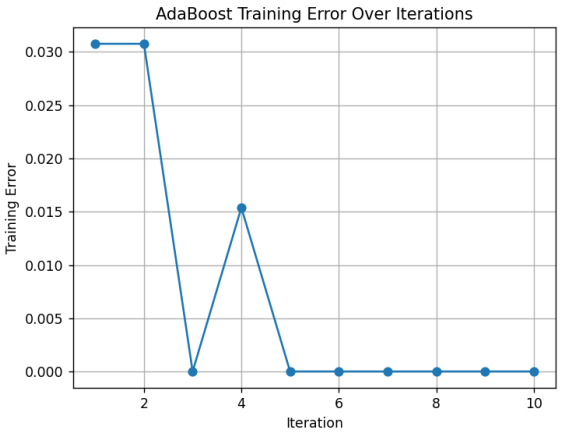
- Residuals tend to be the negative gradient for squared error loss
- Stump just refers to weak classifer, which being trained to predict residuals, model is then updated in current_pred
- Conversion of running pred to binary
- Prediction occurs next, by adding the contributions of all learned stumps

Sample Datasets

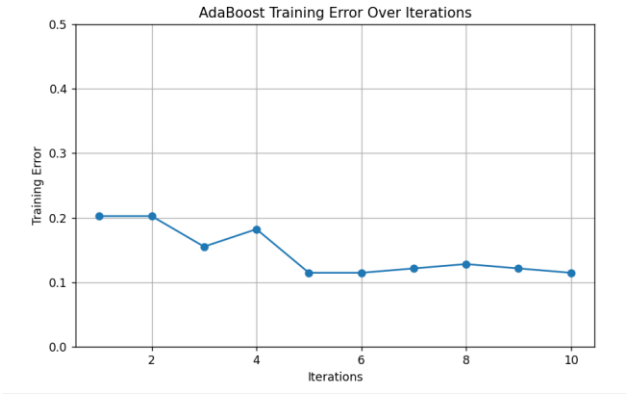
Naturally having gained a theoretical understanding of the two alogrithms , we decide to test the algorithms on various datasets. For this report a simple weak learner is chosen, namely the decision stump, in the future other weak learners such as support vector machines, and decision trees will also be used. In Adaboost, each stump is trained to minimize classification errors on the weighted distribution of the training data while in Gradient Boosting each stump is used to fit the residuals of a loss function

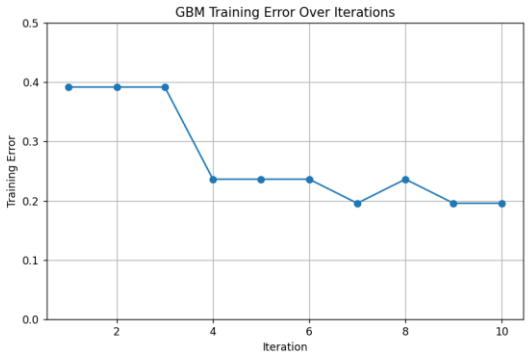
For the purpose of this report, two datasets are used, all of which can be found in the UCI machine learning repository and all three will be using the same two algorithms described. For now, we will focus on binary classification and stick with only 10 iterations.

The Wine dataset is a multivariate dataset that can be found in the UCI Machine Learning Repository and the sklearn dataset library. The target variable is to determine the variety of grapes used to create wine (there are three type but for now we will just focus on two of them) based on features. There are 178 samples and 13 features in the dataset such as alcohol content, malic acid, ash, alkalinity of ash etc. To simplify the problem to binary classification, only classes 0 and 1 were retained, excluding all samples belonging to class 2. Testing results indicate on average that Adaboost achieves an accuracy of approximately 95% while Gradient Boost follows with 92%. Now let us track and plot the training error after every decision stump is added to get an idea of how our models are learning.



Our 2nd dataset is a subset of the heart disease set from UCI, containing processed data from cleveland. There are 303 samples with 13 features such as age, sex, pain level, cholestrol etc. Our target Variable would be a simple categorical label 0 or 1 with 0 indicating no pain and 1 indicating some level of disease. The data in question does have a handful of missing values which were dropped. Adaboost consistently scores around 78% and Gradient boost scores 75%, the lower accuracy shows the limitations of decision stumps especially since the original dataset aimed at categorizing heart disease into multiple level of severity rather than a simple yes or no. While small the model does show gradual improvement in accuracy, although further testing and making changes to weak classifiers and loss functions is perhaps in order.





Future Milestones

The adoption of Machine Learning algorithms has accelerated in recent years with the availability of practical implementations and validation in competitions and real-world applications. In this report we have covered two of the most popular methods, our experiments showed that even with a simplistic set up both boosting algorithms were capable of achieving results, nevertheless there are numerous areas for improvement. In the upcoming weeks it is tended that we:

- Experiment with different learning rates to evaulate their impact on convergence
- Implement other weak learners and visualize training error trends to see which is more optimal
- Expand to other datasets with more complex features and move away from just binary classification
- Replace and test other loss functions
- Run competitive tests between gradient boosting and non-gradient boosting methods
- Submit a final report

Reference (APA)

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences, 55*(1), 119–139. https://doi.org/10.1006/jcss.1997.1504

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics, 29*(5), 1189–1232. https://doi.org/10.1214/aos/1013203451

Foote, K. D. (2021, December 3). A brief history of machine learning. *DATAVERSITY*. https://www.dataversity.net/a-brief-history-of-machine-learning/

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2nd ed.). Springer.

Janosi, A., Steinbrunn, W., Pfisterer, M., & Detrano, R. (1989). *Heart Disease* [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C52P4X

Aeberhard, S., & Forina, M. (1992). *Wine* [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C5PC7J