Boosting

Sean Sall

June 3rd, 2016

Morning Lecture Notes

Morning Objectives

- Understand the general idea of boosting, and how it compares to other ensemble methods (e.g. Random Forests)
- Understand the general form of the gradient boosting algorithm
- Explain what each of the tunable hyper-parameters are when decision trees are used as the weak learner in boosting, and the general strategy for fitting boosted trees
- Implement a grid search over the tunable hyper-parameters when fitting boosted trees, and interpret the results
- Explain the advantages and disadvantages of boosting

Morning Agenda

- Boosting motivation and high level overview
- Gradient boosting motivation and high level overview
- Gradient boosting algorithm discussion
- Boosting decision trees
  - Gradient boosted trees
  - Tunable hyper-parameters and grid searching

Why does this morning matter?

- Boosting algorithms perform extremely well in prediction settings, and XGBoost in particular (a variant of boosting) is one of the top performing algorithms on Kaggle.
- If you need to squeeze out that last ounce of performance, boosting is a good way to go.

Boosting Motivation - Toy examples

Boosting - Example 1

- It's exam time!

Figure 1:Exam time!

- Normally, this would stink. . .

Figure 2:Exams stink - am I right?

Boosting - Example 1

- But, today you get to take the exam as a class (kind of)! So, now you're all in it together...









Boosting - Example 1

- The way it's going to work is as follows:
  1. The first person will take the test, from start to finish.
  2. We'll then see how that person did, calculating what he/she got right and wrong.
  3. After this, we'll pass the test on to the next person, who will take it from start to finish. **But**, this person will also get told what the last person got right and wrong (so in essence they're taking a slightly different test).
  4. We'll iterate over steps 2 and 3 until everybody has had a chance to take a **version** of the test, and use the final error rate (how many questions were incorrect) as the grade for each person in the class.

Boosting - Example 1

- Sounds pretty good, right?

Figure 3:Crushed it!

Boosting - Example 2

- Let's predict the stock market (Step `n`... profit $$$$), specifically how much the market will change for a handful of stocks

Figure 4:How much will it change?

Boosting - Example 2

- For predicting, though, we'll have each person predict based off a **slighty different** version of the data:
  1. The first person will look over the original data, and then make predictions on how each stock's price will change for the next day.
  2. We'll look at tomorrow's prices (which we have) and figure out how much the first person missed by for each stock (maybe it's $1 for Apple, and $0 for Microsoft)
  3. Next, we'll give the next person a shot at predicting how the stock's price will change for the next day. **But**, this person is going to know how much the person before missed by. This means he/she will be able to focus more on where we're predicting poorly.
  4. Iterate over steps 2 and 3 until everybody has had one pass over a **version** of the data, and aggregate each of the predictions to get our final result.

Boosting - Example 2

- Time to retire?

Figure 5:Data sciencing never felt so good.

Boosting - High Level Overview

Boosting - High Level Overview I

- The motivation for boosting was to build an algorithm that combined the outputs of many **weak learners** to produce a powerful committee
  - A **weak learner** is defined as an algorithm (model) whose error rate is only slightly better than we would achieve through intelligent random guessing (think **high bias, low variance**)

Figure 6:Random Guessing - because why not?

Boosting - High Level Overview II

- So, we end up fitting a **sequence of weak learners**, where each **weak learner** builds on the previous one:

$$G_m(X) = G_{m-1}(X) + \alpha_m \phi(X, \gamma_m)$$

Note: ϕ is notation to denote our generic, *untrained* weak learner (whereas $G_m(X)$ is one of our *trained* weak learners)

Note: α_m is a weight that we apply to each of our weak learners, and γ_m denotes the hyper-parameters of whatever weak learner we are using (for a decision tree, this is our max depth, minimum samples per leaf, etc.)

- How do we build up this committee? Specifically, how do we do it with trees?

Ensemble fitting - Bagging and Random Forests review

- With **bagged trees**, we fit many deep trees (**low bias, high variance**) and then average them in the hopes that we could decrease the variance
  - Outperforms an individual decision tree, but doesn't give us as good a decrease in variance as we would like
- With **random forests**, we still fit many deep trees, but take a random subset of the input variables at each split in order to **decorrelate** the trees
  - Outperforms individual decision trees and bagged trees, as we're able to more readily leverage a decrease in variance when averaging the trees
- In both cases, each of the trees are fit independently of each other (e.g. the fitting of one tree doesn't affect the fitting of another tree, and there is no **sequential** nature to it)

Ensemble fitting - Boosting I

- With **boosting**, we fit the trees in a **sequential** manner, where each tree is dependent upon the last
- Each subsequent tree is able to kind of focus on where the previous tree didn't do so well
  - . . . . . . but how?

Ensemble fitting - Boosting II

- Each subsequent tree is going to be fit on the gradient of the loss function from the previous tree:

$$r_{im} = -\frac{\partial L(y_i, G_{m-1}(x_i))}{\partial G_{m-1}(x_i)}$$

Gradient Boosting

Gradient Descent - Review

- Recall that with **gradient descent**, we are trying to intelligently update our parameters such that we decrease our loss with each update:

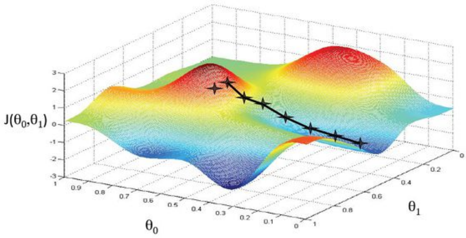


Figure 7:Gradient Descent :)

Gradient Boosting - General Description

- Fitting sequentially on the gradient of the loss function is in fact the **gradient boosting algorithm** - we fit a weak learner (here a tree) in a **sequential** manner, where each one (except the first) is fit on the gradient of the loss function from the previous one
- We each model sequentially:

$$G_m(X) = G_{m-1}(X) + \alpha_m \phi(X, \gamma_m)$$

Gradient Boosting - Algorithm

Gradient boosting algorithm:

1. Initialize $G_0(x)$ (the first tree) $= argmin_\gamma \sum_{i=1}^{N} L(y_i, \phi(x_i; \gamma))$
2. For m = 1 to M, **do**:
   1. Compute the gradient (for all obs.): $r_{im} = -\frac{\partial L(y_i, G_{m-1}(x_i))}{\partial G_{m-1}(x_i)}$
   2. Use the weak leaner (here a tree) to compute γ_m which minimizes:

      $$\sum_{i=1}^{N} L(r_{im}, \phi(x_i; \gamma))$$

   3. Update $G_m(X) = G_{m-1}(X) + \alpha\phi(X; \gamma_m)$
3. Return $G(X) = G_M(X)$

Note: See the Appendix for the walk-through given in class.

Gradient Boosting - Algorithm

Gradient boosting algorithm:

ee) $= argmin_{\gamma} \sum_{i=1}^{N} L(y_i, \phi(x_i;\gamma))$

2. For m = 1 to M, **do**:

for all ob : $r_{im} = -\dfrac{\partial L(y_i, G_{m-1}(x_i))}{\partial G_{m-1}(x_i)}$

2. Use the we eaner (here a tree

$$\sum^{N} L(r_{im}, \phi(x_i;\gamma))$$

3. Update $G_m(X) = G_{m-1}(X) + \alpha\phi(X;\gamma_m)$

Note: See the Appendix for the walk-through given in class.

Gradient Boosting - Revisiting Examples

- In our testing scenario earlier, the "gradient" would be the knowledge of what questions the previous tester got right or wrong

Figure 8:Testing Again!?

- In our stock market example, the "gradient" would be the knowledge of how far off the previous person's predictions were for each stock

Figure 9:$$$$$$

Gradient Boosted Trees - Squared Error Loss

- If we use squared error loss, then the gradient is actually just the residual:

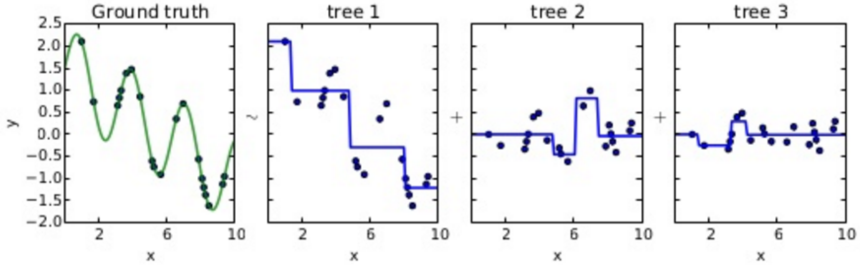


Figure 10:Boosting squared error loss

Gradient Boosting Trees - Hyper-parameters I

- With gradient boosted trees, we have all of the same hyper-parameters to tune that we had with an individual decision tree:
  - max depth, minimum samples per leaf, etc.
- We also have additional parameters that come with boosted trees:
  - learning rate (shrinkage parameter, α), number of trees
  - max features, whether to fit trees on a random subset of the data (**stochastic gradient boosting**)
- The question, though, is how we make a tree a **weak learner**? We know that we want trees that are **high bias, low variance**... how do we do that?

Gradient Boosted Trees - Hyper-parameters II

- Typically, boosted trees are grown fairly shallow, with a max-depth of 4-8 (although there has been reported success with stumps)
  - This is what allows us to get trees that are **weak learners**, because it controls the degree of interaction in each tree
- What about the other hyper-parameters, though?
  - ... To find those (in addition to tuning the max depth), we can use **cross-validation** and **grid-searching**

Gradient Boosted Trees - Grid Search I

- **Grid-searching** allows us to search over combinations of different hyper-parameters:

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.grid_search import GridSearchCV
param_grid = {'learning_rate': [0.001, 0.01, 0.1],
              'max_depth': [4, 6],
              'min_samples_leaf': [4, 8, 16],
              'max_features': [0.75, 0.9, 1]}
model = GradientBoostingRegressor(n_estimators=3000)
gs_cv = GridSearchCV(model, param_grid).fit(X, y)

gs_cv.best_params_, gs_cv.best_score_, gs_cv.best_estimator_
```

Gradient Boosted Trees - Grid Search II

```
gs_cv = GridSearchCV(model, param_grid).fit(X, y)
```

- When this is called, the GridSearchCV is going to iterate over every possible combination of parameters that could be created given the parameters in our `param_grid`, create folds for the data, and for each fold:
    1. call the `fit` step on the `model`
    2. call the `predict` step on the `model`
    3. record the score (which you'll be able to access later)

Gradient Boosted Trees - Hyper-parameter Tuning I

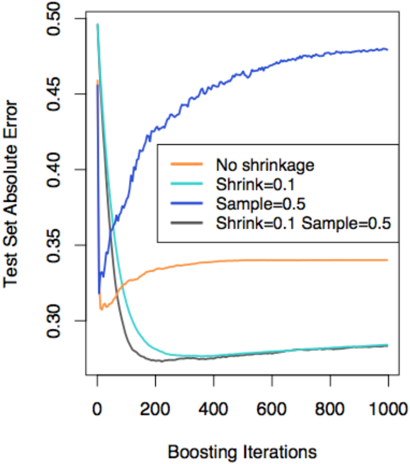
- Note that in the last slides we didn't tune `n_estimators`...
- This is because `n_estimators` and the `learning_rate` work opposite each other
  - As the `learning_rate` decreases, all else being equal, `n_estimators` has to increase (a smaller `learning_rate` means you're descending more slowly along the error curve)
  - Typically, it's a good idea to fix one or the other and only tune one (it's common to try to optimize the `learning_rate` to be low and simply use lots of trees)
    - Be careful to try to avoid overfitting - too large of a `learning_rate` can lead to overfitting with even a small number of trees

Gradient Boosted Trees - Stochastic Gradient Boosting

- **Stochastic Gradient Boosting** may involve tuning the `max_features` as well as `sub_sample`
  - **`max_features`** - random subsample of features considered for a split (like in Random Forest)
  - **`sub_sample`** - randomly subset the training data for each tree
- Has been shown to work well in practice (see next slide)

Gradient Boosted Trees - Hyper-parameter Tuning Visual

- **Takeaway**: Trying out combinations of hyper-parameters is critical in terms of finding a good set



Gradient Boosted Trees - Staged Predict

- Note that you could (and will) create the graph using python's `staged_predict`:

```
# Get train score from each boost -
# estimator is a gradient boosted model
for i, y_pred in enumerate(estimator.staged_predict(x)):
        scores[i] = mean_squared_error(y_true, y_pred)
```

- This will iterate over all of the individual trees that the gradient boosted model fit, and call the predict step on them with `x` as input
- You can do this with your training or your test set (in the previous slide we used our test set to obtain the values for that curve)

- Gradient boosted trees often outperform most (if not all) other models

Figure 12:Happy Dance

Gradient Boosted Trees - Disadvantages

- Typically, to get the level of performance that gradient boosted trees offer, fairly extensive tuning is required
- They also aren't parallelizeable, since they are fit **sequentially**

Figure 13:So much less time...

Morning Break/Individual Exercise Time

Afternoon Lecture Notes

Afternoon Objectives

- Solidify an understanding of boosting as a general algorithm
- Understand and implement AdaBoost
- Understand the advantages of using XGBoost
- Explain how to construct and interpret a partial dependency plot

Afternoon Agenda

- Boosting overview
  - General weak learners
  - General Loss function
- AdaBoost (Adaptive Boosting)
- XGboost (Extreme Gradient Boosting)
- Partial dependency plots

Why does this afternoon matter?

- Boosting algorithms perform extremely well in prediction settings, and XGBoost in particular (a variant of boosting) is one of the top performing algorithms on Kaggle
  - If you need to squeeze out that last ounce of performance, boosting is a good way to go.
- Constructing **partial dependency plots** are a useful technique/tool for interpreting the results of any "black box" learning algorithm
  - They're going to get us as close to β's as we can get with our non-parametric models

Boosting Overview

Gradient Boosting - Weak Learners

- We have at this point been mainly talking about trees as our **weak learner**, but note that we could use any model as a weak learner, subject to the degree with which we can train that model to have an error rate only **slightly better** than intelligent random guessing
  - It turns out that in practice, boosting anything but trees isn't terribly worthwhile (e.g. it doesn't improve performance that much)
  - As such, we pretty much only boost trees

Gradient Boosting - Loss functions I

- Up to now we haven't specified a loss function. When we specify a loss function, we move from a general discussion of boosting to actually discussing **specific** boosting algorithms
- Popular loss functions include:

Name	Loss Function
Squared error	$\frac{1}{2}(y_i - \hat{y}_i)^2$
Absolute error	$\|y_i - \hat{y}_i)\|$
Exponential loss	$exp(-y_i * \hat{y}_i))$
Deviance	kth component: $I(y_i = G_k) - p_k(x_i)$

- $p_k(x_i)$ is the predicted probability that your observation x_i belongs to the $k^{th} class$

Gradient Boosting - Loss functions II

- The particular loss function you use depends on your problem
  - In regression problems, squared or absolute error is used
  - In classification problems, deviance is common
    - Using exponential loss leads to **AdaBoost**, which generally doesn't perform as well as using deviance (we'll cover **AdaBoost**)
- That being said, **AdaBoost** was actually one of the earliest boosting algorithms used, so let's dive into that...

Adaptive Boosting (AdaBoost)

AdaBoost - General Overview I

- **AdaBoost** (short for Adaptive Boosting) is gradient boosting when we plug in an **exponential loss** function
- It has a nice interpretation of assigning weights to individual observations, and then iteratively adjusting those weights based on how well we are predicting for each individual observation (sounds familiar, right?)

AdaBoost Algorithm

1. Initialize the observation weights $w_i = \frac{1}{N}$, for i = 1, 2, . . . , N
2. For m = 1 to M, **do**:
   1. Fit a classifier $G_m(x)$ to the training data using weights w_i.
   2. Compute: $err_m = \dfrac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$
   3. Compute $\alpha_m = log((1 - err_m)/err_m)$
   4. Set $w_i = w_i * exp[\alpha_m * I(y_i \neq G_m(x_i))]$, i = 1, 2, . . . , N.
3. Output $G(X) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$

AdaBoost Algorithm

1. Initialize the observation weights $w_i = \frac{1}{N}$, for i = 1, 2, . . . , N
2. For m = 1 to M, **do**:
    1. Fit a classifier $G_m(x)$ to the training data using weights w_i.
    2. Compute: $err_m = \dfrac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$
    3. Compute $\alpha_m = log((1 - err_m)/err_m)$
    4. Set $w_i = w_i * exp[\alpha_m * I(y_i \neq G_m(x_i))]$, i = 1, 2, . . . , N.
3. Output $G(X) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$

- See the `Appendix` to link AdaBoost to our general Gradient Boosting Algorithm formula we looked at earlier

Extreme Gradient Boosting (XGBoost)

XGBoost Algorithm - Part I

- XGBoost (short for extreme gradient boosting) is gradient boosting with some niceties built in that make it much faster and more efficient than standard gradient boosting
- The primary change from standard gradient boosting is that they bin observation's column values into percentiles when performing splitting
  1. For some j_{th}, column, we split all observations up into percentiles based off their values for that j_{th} column
  2. Then, only consider some average value (mean, median, etc.) across each percentile for splitting (this narrows the search space quite a bit)

XGboost Algorithm - Part II

- Other changes that XGBoost includes are:
  - Handles sparsity in the data (e.g. missing values)
  - Handles mixed data types (e.g. categoricals, continuous)
  - Allows for out-of-core computation
  - Builds in smart memory management

XGBoost Resources

- It can be installed using `pip`, and it does have an `sklearn` interface (e.g. `fit`, `predict` methods, etc.)

```
import xgboost as xgb
gbm = xgb.XGBClassifier(max_depth=3, n_estimators=300,
                        learning_rate=0.05)
                       .fit(train_X, train_y)
predictions = gbm.predict(test_X)
```

Partial Dependency Plots

It's time for an example

- Let's try to predict median housing value in California for neighborhoods (pictures stolen from Elements of Statistical Learning, Chapter 10)...

Figure 14:Real estate?

Feature Importance Review

- Remember that when using an ensemble of trees, we can get a look at which variables are most important by looking at **feature importance**

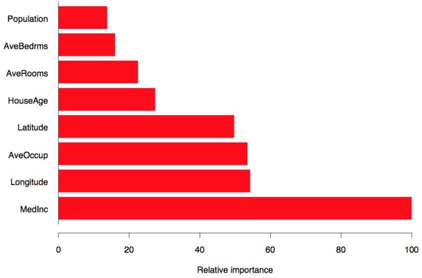


Figure 15:Feature Importances on California Housing

- How do we actually quantify the effects of one of these variables on the response, though?

Partial Dependency Plots - Overview I

- With **partial dependency plots**, we have a useful tool for teasing out and quantifying the effects of an individual variable on our response
- Effectively, **after fitting the model**, we'll cycle over some pre-determined values of the individual variable of interest, predicting on those values and observing how our responses changes
- How the response changes across different values of our variable of interest is the **partial dependency** of the response on that variable

Partial Dependency Plots Visual I

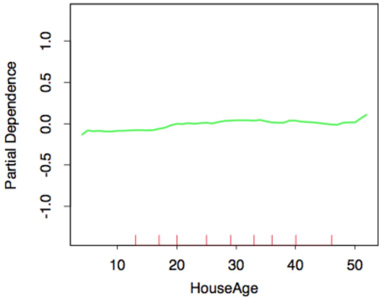


Figure 16:Partial dependence of median house value on median age of houses in the neighborhood

- Here, we can see that once we control for the average effects of all other variables, median house value has a small partial dependence on median age of the house

Partial Dependency Plots Visual II

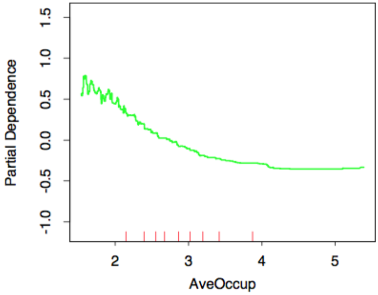


Figure 17:Partial dependence of median house value on average occupancy of houses in the neighborhood

- Here, we can see that once we control for the average effects of all other variables, median house value has a noticeable partial dependence on the average occupancy of houses in the neighborhood.

Partial Dependency Plots - Calculation

- To calculate the partial dependence by hand, a common way of doing it for a single variable is the following:
  1. Fit our machine learning model/algorithm - this should be able to basically tease out all of the average effects of each individual variable
  2. Pick a variable that you would like to calculate the partial dependency of
  3. Pick a range of values that you want to calculate the partial dependency for
  4. Loop over those values, one at a time doing the following for **each value**:
     1. Replace the entire column corresponding to the variable of interest with the current value that is being cycled over (we'll do this with our training set)
     2. Use the model to `predict` (again with the training data)
     3. Average all of the reponses, and calculate the difference of this average to the average calculated in the last iteration of the loop
     4. This (`value`, `difference`) becomes an (`x`, `y`) pair for your partial dependency plot

Partial Dependency Plots - Overview II

- We're first finding the average effects of each individual variable (that's the fitting step, `1`)
- Then, we're observing how the response changes as the values of **one** variable change, **holding the effects of the other variables fixed**

Note: See the `Appendix` section for the hand calculation example used in class.

Partial Dependency Plots Visual III

- We can even plot the partial dependency of two variables relative to the response (more than two gets tough):

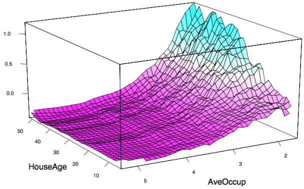


Figure 18:Partial dependence of median house value on median house age and average occupancy

- Here, we see that there is a strong interaction between `HouseAge` and `AveOccup`, which we weren't able to see in looking at either the feature importances or partial dependency plots of a single variable

Partial Dependency Plots Visual IV

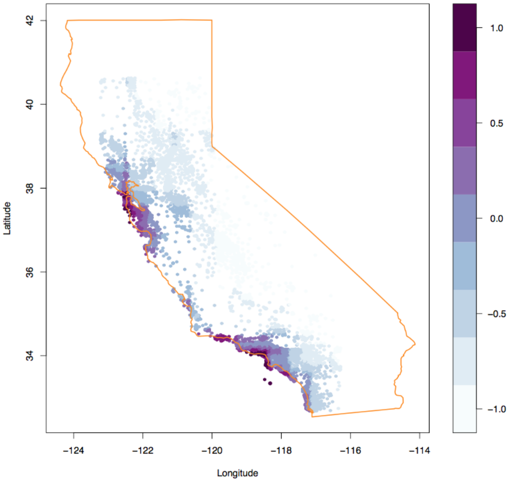


Figure 19:Partial dependence of median house value on latitude and longitude

Partial Dependency Plots - Code

- Let's walk through the code here (sklearn built-in for partial dependency):

Appendix - Gradient Boosting Algorithm

Gradient boosting algorithm:

Recall our general Gradient boosting algorithm

1. Initialize $G_0(x)$ (the first tree) $= argmin_\gamma \sum_{i=1}^{N} L(y_i, \phi(x_i; \gamma))$
2. For m = 1 to M, **do**:
    1. Compute the gradient (for all obs.): $r_{im} = -\frac{\partial L(y_i, G_{m-1}(x_i))}{\partial G_{m-1}(x_i)}$
    2. Use the weak leaner (here a tree) to compute γ_m which minimizes:

$$\sum_{i=1}^{N} L(r_{im}, \phi(x_i; \gamma))$$

    3. Update $G_m(X) = G_{m-1}(X) + \alpha\phi(X; \gamma_m)$
3. Return $G(X) = G_M(X)$

Let's break this down step by step...

Gradient Boosting Algorithm - Part 1

1. Initialize $G_0(x)$ (the first tree) $= argmin_\gamma \sum_{i=1}^{N} L(y_i, \phi(x_i; \gamma))$
   - γ here is simply the set of hyper-parameters that are tunable for the given weak learner
     - ★ For a decision tree, this is the max depth, minimum samples per leaf, number of variables to consier at each split, etc.
   - Here, we're basically just saying that we're going to fit our weak-learner like we normally would, by minimizing the loss function that we're using
     - ★ Note we are fitting on the **original** data (e.g. y_i above)

Gradient Boosting Algorithm - Part 2

2. For m = 1 to M, **do**:
   1. Compute the gradient: $r_{im} = -\frac{\partial L(y_i, G_{m-1}(x_i))}{\partial G_{m-1}(x_i)}$
   2. Use the weak leaner (here a tree) to compute γ_m which minimizes:
   $$\sum_{i=1}^{N} L(r_{im}, \phi(x_i; \gamma))$$
   3. Update $G_m(X) = G_{m-1}(X) + \alpha\phi(X; \gamma_m)$

Let's break this down step by step...

Gradient Boosting Algorithm - Part 2.1

2. 1. Compute the gradient: $r_{im} = -\dfrac{\partial L(y_i, G_{m-1}(x_i))}{\partial G_{m-1}(x_i)}$
   - We'll let the computers do this
   - Note, though, that if our loss is squared error, then this calculation simply yields the residuals themselves:

$$\textit{Loss: } \frac{1}{2}(y_i - f(x_i))^2$$
$$\textit{Derivative: } y_i - f(x_i)$$

- The $\dfrac{1}{2}$ just allows us to not have a coefficient floating around

Gradient Boosting Algorithm - Part 2.2

2. 2. Use the weak leaner (here a tree) to compute γ_m which minimizes:

$$\sum_{i=1}^{N} L(r_{im}, \phi(x_i; \gamma))$$

- Again, γ is simply the set of hyper-parameters that are tunable for the given weak learner
  - ★ For a decision tree, this is the max depth, minimum samples per leaf, number of variables to consier at each split, etc.
- Here, we're fitting our weak-learner like we normally would in terms of minimizing the loss function that we're using, **but** we're fitting on the **gradient** that we calculated in step 2.1 (e.g. r_{im} above)

Gradient Boosting Algorithm - Part 2.3

2. 3. Update $G_m(X) = G_{m-1}(X) + \alpha\phi(x_i; \gamma_m)$

- This is our kind of standard **gradient descent** step, where we update and take a step in (hopefully) the right direction on our error curve.
- The difference between this and the gradient descent algorithm we looked at before is that we're updating the output of a function ($G_{m-1}(X)$) instead of a set of parameters (like our θ_1 and θ_2 on gradient descent day)

Gradient Boosting Algorithm - Part 3

3. Return $G(X) = G_M(x)$
   - This is the aggregate of all of the functions that we learned along the way

Appendix - AdaBoost link to Gradient Boosting

AdaBoost Algorithm

Recall our AdaBoost Algorithm...

1. Initialize the observation weights $w_i = \frac{1}{N}$, for i = 1, 2, ..., N
2. For m = 1 to M, **do**:
    1. Fit a classifier $G_m(x)$ to the training data using weights w_i.
    2. Compute: $err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$
    3. Compute $\alpha_m = log((1 - err_m)/err_m)$
    4. Set $w_i = w_i * exp[\alpha_m * I(y_i \neq G_m(x_i))]$, i = 1, 2, ..., N.
3. Output $G(X) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$

Gradient Boosting Algorithm

Recall our general Gradient boosting algorithm. . .

1. Initialize $G_0(x)$ (the first tree) $= argmin_{\gamma} \sum_{i=1}^{N} L(y_i, \phi(x_i;\gamma))$
2. For m = 1 to M, **do**:
   1. Compute the gradient: $r_{im} = -\frac{\partial L(y_i, G_{m-1}(x_i))}{\partial G_{m-1}(x_i)}$
   2. Use the weak leaner (here a tree) to compute γ_m which minimizes:

      $$\sum_{i=1}^{N} L(r_{im}, \phi(x_i;\gamma))$$

   3. Update $G_m(X) = G_{m-1}(X) + v\phi(X;\gamma_m)$
3. Return $G(X) = G_M(x)$

Gradient Boosting - Step 2.2

- Use the weak leaner (here a tree) to compute γ_m which minimizes:

$$\sum_{i=1}^{N} L(r_{im}, \phi(x_i; \gamma))$$

- With **AdaBoost**, we use exponential loss

$$exp(-y_i * \hat{y}_i)$$

- At Step 2.2 (above), then, we find minimize the following:

$$argmin_{\gamma} \sum_{i=1}^{N} exp[-y_i * \hat{y}_i]$$

Gradient Boosting - Simplifications/Notes

- Recall that at stage m, $\hat{y}_i = f_m(x_i)$, and $f_m(x_i) = f_{m-1}(x_i) + \alpha_m \phi(x_i, \gamma_m)$
- So, for Step 2.2, we can rewrite our minimization as follows:

$$argmin_{\gamma,\alpha} \sum_{i=1}^{N} exp[-y_i * (f_{m-1}(x_i) + \alpha\phi(x_i, \gamma))]$$

- Some notes before diving in:
  - For this derivation, we are assuming **binary classification**, where we are fitting to $y_i \epsilon$-1, +1 (e.g. the negative cases are given by -1, and the positive cases a +1). This will simplify the math.
  - For the remainder of this derivation, we're going to drop γ within ϕ (let's just minimize our loss over α), and denote our loss as follows:

$$L_m(\phi) = \sum_{i=1}^{N} exp[-y_i * (f_{m-1}(x_i) + \alpha\phi(x_i))]$$

  - We're basically going to take α from this loss and work through to $\alpha_m = log((1 - err_m)/err_m)$ that we saw in the **AdaBoost** algorithm

Gradient Boosting to AdaBoost

- Let's minimize our loss:

$$L_m(\phi) = \sum_{i=1}^{N} exp[-y_i * (f_{m-1}(x_i) + \alpha\phi(x_i))]$$

- Before we get there, let's somehow work in those weights (w_i) that we have in AdaBoost (this is going to be a long, messy derivation, but we'll put it all back together at some point)

Gradient Boosting to AdaBoost I

1. $L_m(\phi) = \sum_{i=1}^{N} exp[-y_i * (f_{m-1}(x_i) + \alpha\phi(x_i))]$
2. $L_m(\phi) = \sum_{i=1}^{N} exp[-y_i * f_{m-1}(x_1)] * exp[-\alpha y_i \phi(x_i)]$
3.
    1. Define $w_{i,m} = exp[-y_i * f_{m-1}(x_i)]$, and then plug that in:
    2. $L_m(\phi) = \sum_{i=1}^{N} w_{i,m} * exp[-\alpha y_i \phi(x_i)]$
4.
    1. If $y_i = \phi(x_i)$, then $y_i * \phi(x_i) = 1$, and $exp[-\alpha y_i \phi(x_i)] = exp[-\alpha]$
    2. If $y_i \neq \phi(x_i)$, then $y_i * \phi(x_i) = -1$, and $exp[-\alpha y_i \phi(x_i)] = exp[\alpha]$
    3. Use that result to obtain the following:
    $L_m(\phi) = \sum_{y_i = \phi(x_i)} w_{i,m} * e^{-\alpha} + \sum_{y_i \neq \phi(x_i)} w_{i,m} * e^{\alpha}$

Gradient Boosting to AdaBoost II

5. Since those α's aren't depending on i, we can move them outside the summation:
$L_m(\phi) = e^{-\alpha} \sum_{y_i=\phi(x_i)} w_{i,m} + e^{\alpha} \sum_{y_i \neq \phi(x_i)} w_{i,m}$
6. Instead of taking the sum over just those observations where $y_i = \phi(x_i)$ or $y_i \neq \phi(x_i)$, we can take it over all observations and move that condition inside the summation:

$$L_m(\phi) = e^{-\alpha} \sum_{i=1}^{N} w_{i,m} I(y_i = \phi(x_i)) + e^{\alpha} \sum_{i=1}^{N} w_{i,m} I(y_1 \neq \phi(x_i))$$

Note: The I is the indicator function.

Gradient Boosting to AdaBoost III

7. Next, we do a little math (see the change in the first term):

$$L_m(\phi) = e^{-\alpha} \sum_{i=1}^{N} w_{i,m}(1 - I(y_i \neq \phi(x_i))) + e^{\alpha} \sum_{i=1}^{N} w_{i,m} I(y_1 \neq \phi(x_i))$$

Gradient Boosting to AdaBoost IV

Some more manipulation...

8. $L_m(\phi) = e^{-\alpha} \sum_{i=1}^{N} w_{i,m} - e^{-\alpha} \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i)) + e^{\alpha} \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))$
9. $L_m(\phi) = e^{-\alpha} \sum_{i=1}^{N} w_{i,m} + (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))$
10. Okay, so now we have our loss in a format where we can take the derivative with respect to α, set it equal to 0, and solve.

Gradient Boosting to AdaBoost V

11. First let's take the derivative $\frac{\partial L_m(\phi)}{\partial \alpha}$, and set it equal to 0:

$$0 = -e^{-\alpha} \sum_{i=1}^{N} w_{i,m} + (e^{\alpha} + e^{-\alpha}) \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))$$

12. Multiply through on the right side:

$$0 = -e^{-\alpha} \sum_{i=1}^{N} w_{i,m} + e^{\alpha} \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i)) + e^{-\alpha} \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))$$

Gradient Boosting to AdaBoost VI

13. Move the $-e^{-\alpha}\sum_{i=1}^{N} w_{i,m}$ to the left side, and then divide each side by it: $1 = \dfrac{e^{\alpha}\sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i)) + e^{-\alpha}\sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))}{-e^{-\alpha}\sum_{i=1}^{N} w_{i,m}}$

Gradient Boosting to AdaBoost VII

14. Divide every term on the right side by $e^{-\alpha}$:

$$1 = \frac{e^{2\alpha} \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i)) + \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))}{\sum_{i=1}^{N} w_{i,m}}$$

15. Multiply through by the $\sum_{i=1}^{N} w_{i,m}$ on the bottom:

$$\sum_{i=1}^{N} w_{i,m} = e^{2\alpha} \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i)) + \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))$$

Gradient Boosting to AdaBoost VII

16. Subtract the $\sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))$ from the right, and then divide by it to isolate $e^{2\alpha}$:
$$e^{2\alpha} = \frac{\sum_{i=1}^{N} w_{i,m} - \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))}{\sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))}$$
17. Take the log of everything (with base e), and simplifiy:
$$\alpha = \frac{1}{2} \log \left(\frac{\sum_{i=1}^{N} w_{i,m} - \sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))}{\sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))} \right)$$

Gradient Boosting to AdaBoost VIII

18 Because math (I'm tired of this derivation and you should be, too):

$$\alpha = \frac{1}{2} \log \left(\frac{\sum_{i=1}^{N} w_{i,m}}{\sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))} - 1 \right)$$

19 Denote err_m as **AdaBoost** does (then see what we do in the next slide):

$$err_m = \frac{\sum_{i=1}^{N} w_{i,m} I(y_i \neq \phi(x_i))}{\sum_{i=1}^{N} w_{i,m}}$$

Gradient Boosting to AdaBoost IX

20. Using that definition of err_m in 19, we can re-write 18 as:

$$\alpha = \frac{1}{2}\log\left(\frac{1}{err_m} - \frac{err_m}{err_m}\right)$$

21. We've made it!! (Rework 20. . . ..):

$$\alpha = \frac{1}{2}\log\left(\frac{1 - err_m}{err_m}\right)$$

Note: Since the $\frac{1}{2}$ is a constant, it's not important in this case.

Appendix - Partial Dependency Plots

Data

- Let's say we're trying to estimate the median housing value in California neighborhoods, and we have the following **original data** (yes, for now we only have 5 obs. and three columns):

med_value	avg_occup	med_age
3.5	2.3	4.1
4.5	3.1	1.2
3.7	4.9	4.7
2.1	1.6	3.3
1.3	2.8	5.8

med_value: Median housing value in a neighborhood (our response)
avg_occup: Average occupancy of houses in the neighborhood
med_age: Median age of houses in the neighborhood

Partial Dependence Process - A refresher

- To calculate the partial dependence by hand, a common way of doing it for a single variable is the following:
    1. Fit our machine learning model/algorithm - this should be able to basically tease out all of the average effects of each individual variable
    2. Pick a variable that you would like to calculate the partial dependency of
    3. Pick a range of values that you want to calculate the partial dependency for
    4. Loop over those values, one at a time doing the following for **each value**:
        1. Replace the entire column corresponding to the variable of interest with the current value that is being cycled over (we'll do this with our training set)
        2. Use the model to `predict` (again with the training data)
        3. Average all of the reponses, and calculate the difference of this average to the average calculated in the last iteration of the loop
        4. This (`value`, `difference`) becomes an (`x`, `y`) pair for your partial dependency plot

Partial Dependence Process - Steps I, II, and III

1. Fit our machine learning model/algorithm - this should be able to basically tease out all of the average effects of each individual variable
   - Let's say we fit gradient boosted trees
2. Pick a variable that you would like to calculate the partial dependency of
   - Let's go with `avg_occup`
3. Pick a range of values that you want to calculate the partial dependency for
   - Since our observations values range from 1.6 to 4.9, let's go from 1.5 to 5.0, by 0.1

Partial Dependence Process - Step IV

4. Loop over those values, one at a time doing the following for **each value**:
    1. Replace the entire column corresponding to the variable of interest with the current value that is being cycled over (we'll do this with our training set)
    2. Use the model to `predict` (again with the training data)
    3. Average all of the reponses, and calculate the difference of this average to the average calculated in the last iteration of the loop
    4. This (`value`, `difference`) becomes an (`x`, `y`) pair for your partial dependency plot

Partial Dependence Process - Step IV I

- We'll start by replacing the `avg_occup` with our first value (1.5), predicting, and the taking the mean response.

med_value	avg_occup	med_age	preds
3.5	1.5	4.1	3.4
4.5	1.5	1.2	4.6
3.7	1.5	4.7	3.5
2.1	1.5	3.3	2.0
1.3	1.5	5.8	1.4

- The mean prediction is $\frac{(3.4+4.6+3.5+2.0+1.4)}{5} = 2.98$
- Note there is no difference to calculate here becuase this is our first value

Partial Dependence Process - Step IV II

- We'll then by replace the avg_occup with our second value (1.6), predicting, and the taking the mean response.

med_value	avg_occup	med_age	preds
3.5	1.6	4.1	3.9
4.5	1.6	1.2	4.8
3.7	1.6	4.7	4.2
2.1	1.6	3.3	2.5
1.3	1.6	5.8	1.9

- The mean prediction is $\frac{(3.9+4.8+4.2+2.5+1.9)}{5} = 3.46$
- The difference between this and the mean prediction when avg_occup was 1.5 is 0.48, so we plot the point (1.6, 0.48)

Partial Dependence Process - Step IV III

- We'll then by replace the avg_occup with our third value (1.7), predicting, and the taking the mean response.

med_value	avg_occup	med_age	preds
3.5	1.7	4.1	3.7
4.5	1.7	1.2	4.6
3.7	1.7	4.7	4.1
2.1	1.7	3.3	2.6
1.3	1.7	5.8	1.5

- The mean prediction is $\frac{(3.7+4.6+4.1+2.6+1.5)}{5} = 3.3$
- The difference between this and the mean prediction when avg_occup was 1.6 is -0.16, so we plot the point (1.6, -0.16)

Partial Dependence Process - Iterating...

- We continue in this manner all the way up to the last value in the range of values we want to cycle over (we chose 5.0)...
- ....
- ....

Partial Dependence Process - Iterating...

- We'll then finish by replacing the `avg_occup` with our last value (5.0), predicting, and the taking the mean response.

med_value	avg_occup	med_age	preds
3.5	5.0	4.1	2.1
4.5	5.0	1.2	3.2
3.7	5.0	4.7	3.3
2.1	5.0	3.3	1.8
1.3	5.0	5.8	0.9

- The mean prediction is $\frac{(2.1+3.2+3.3+1.8+0.9)}{5} = 2.26$
- Pretending the last mean prediction (when we used 4.9) was 2.51, the difference between that and our current mean would be -0.25. So, we plot the point (5.0, -0.25)