Ensemble Methods

UW ECE 657A - Core Topic

Mark Crowley

Lecture Outline

Lecture Outline

The "Perfect" Algorithm

- No Free Lunch Theorem
  - see wikipedia on this, it's a fascinating area, not completely applicable here but related to the idea that there is never going to be a perfect model for prediction/classification because you've never trained on ALL the data.
  - IOW: There is always some data which could wreck your approach.
- So, there is no perfect algorithm.
- The goal needs to build a model that has regular behaviour on all data, that doesn't overfit to the training data so much that it does badly on the test data.

Regularization

avoid to overfitting

"Any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error." – Murphy 2012

- Example: Minimizing Global Cost Criterion
- Weight Decay: adding L2 or L1 penalties to parameter estimation cost
- Structural Regularization: in deep learning the *Dropout* method is a regularizer
- Today we'll talk about another: ensemble methods

Ensemble Methods for Improving Classification

- Basic Idea: Rather than training just one model on to predict the data with low error, try learning *multiple* models, on subsets of of the data, and combine them to get overall lower error than possible from a single model.
- This acts as a regularizer because it forces the model to learn a more general solution.
- Note: ensemble methods are more general than decision trees, they can be used for *any classifier*, but they go particalarly well with decision trees

Ensemble Methods for Improving Classification

General Essemble Methods (implemented with Decision Tree learners):

- Bagging
  - Random Forests
- Boosting
  - Adaboost (also called Adaptive Reweighting and Combining, Arcing)
- Randomized Subsampling
  - Extremely Randomized Trees
  - Totally Randomized Trees
  - Isolation Forests
- For Streaming Data
  - Hoeffding Trees
  - Mondrian Forests

Lecture Outline

Bootstrap Revisited

- A bootstrap data set is created by randomly selecting n points from the training set X of size n *with replacement*.
- There will usually be some duplicates.
- This is repeated B times with independent draws each time.

Bagging

also called : bootstrap aggregation

- Draw multiple subsamples of the data of size $n' < n$
- Learn a *component classifier* on using each bagged sample
- Final classification decision based on vote from each component classifier
- Usually use same classifier for each component, (neural network, SVM, decision tree) but they could be different (more on multi-classifer systems later)
- Advantage: reduces instability (ie. sensitivity of classifier performance to small changes in data)

Ensemble Methods for Improving Classification

General Essemble Methods:

- Bagging
- Boosting
- Adaboost
- Also called Adaptive Reweighting and Combining (**Arcing**)

Decision Tree Ensembles:

- Random Forests
- Extremely Randomized Trees
- Hoeffding Trees, Mondrian Forests, Streaming Random Forests
- Isolation Forests

Random Forest

- Very good way to reduce the variance of decision trees, make them more stable to different inputs.
- **Bagging:** Train M trees on randomly chosen subsets of the data (with replacement)

$$f(\mathbf{x}) = \sum_{m=1}^{M} \frac{1}{M} f_m(\mathbf{x})$$

- Bagging leads to highly correlated trees, so it helps but could be better.

Random Forests

From (Breiman, 2001)

- Builds on the bagging algorithm
- Will learn T trees.
- For each tree, obtain a bootstrap sample of the datapoints *and* a sample of K features to use for training that tree.
- Use the standard CART tree learning algorithm for each tree,
- But at each split step, find the optimal split by searching the K features and their values.
- The next tree will use a different sample of featuers and datapoints.

Random Forests

- Decorrelates the individual trees (base learners) by also randomly choosing the feature set each can choose from.
- *Result:* Each tree focusses on a subset of the features and learns them better.
- Outperforms many other methods in terms of accuracy.
- Boosting can take this even further

PROS and CONS of Random Forests

PROs:

- Robust to initial data disctribution, sampling process
- Easily tunable, more treats improves accuracy and reduces variances
- Relatively fast to train for such a flexible model.

CONs:

- Harder to interpret than decision trees, can try to average rules across trees
- Still needs to be trained on batch data, not streaming (same as Decision Trees)

Lecture Outline

Boosting

- A meta-algorithm that allows us to create an ensemble of classifiers with arbitrarily high accuracy
- Train mutiple classifiers as in bagging but each bootstrap sample is chosen to maximize the information gain conditioned on the previously trained classifiers.
- This classifier could be a **weak learner**: only guaranteed to be better than chance.

Boosting Example (for regression)

1. Divide D into three sets $D_{1:3}$ of size $< n$
2. Run your favourite classification algorithm on D_1 to create C_1
3. Select subset D_2 by randomly selecting 50% of points that C_1 gets correct and 50% that C_1 gets incorrect.
4. Train C_2 on D_2
5. Select D_3 to be all points in D that C_1 *or* C_2 got wrong
6. Train C_3 on D_3
7. Answer returned by the model is a *weighted sum* of the answers of the weak learners.

Adaboost

- Most popular initial implementation of the boosting meta-algorithm
- We assume the sign of the weak learner $h_k(x)$ gives the class, and the magnitude of the value gives the confidence in the classification
- Compute a joint disciminant function $g(x) = \sum_{k=1}^{k_{max}} \alpha_k h_k(x)$
  - This essentially classifies the datapoints using a **weighted majority** vote of different classifiers, each trained to perform well where all the previous classifiers are doing badly.
  - *vote weight* = measure of the reduction of error this tree contributes
- Now, classify the datapoints using $\text{sign}(g(x))$
- **Further reading:** great chapter by Shapire about possible explanations of why AdaBoost is so effective: `http://rob.schapire.net/papers/explaining-adaboost.pdf`

AdaBoost

Algorithm 1: AdaBoost Algorithm($\mathbf{X}, \mathbf{Y}$)

Input: Datapoints (x_i, y_i) where $x_i \in \mathcal{X}$ and the label $y_i \in \{-1, +1\}$ for $i = 1, \ldots, n$

1

2 Assign initial weights $w_1(i) = 1/n$

3 **foreach** $k=1,\ldots,K$ **do**

4 Train a weak-learner classifier C_k using data sampled by $w_k(i)$

5 $\epsilon_k = Pr_{i \sim w_1}[C_k(x_i) \neq y_i]$ `/* error rate of` C_k `*/`

6

7 $\alpha_k = \frac{1}{2} \ln\left(\frac{1-\epsilon_k}{\epsilon_k}\right)$

8 **foreach** $i = 1, \ldots, n$ **do**

9 $w_{k+1}(i) = \frac{w_k(i) \exp(-\alpha_k y_i C_k(x_i))}{Z_k}$

10 **return** classification function $C(x) = \text{sign}\left(\sum_{k=1}^{K} \alpha_k C_k(x)\right)$

Lecture Outline

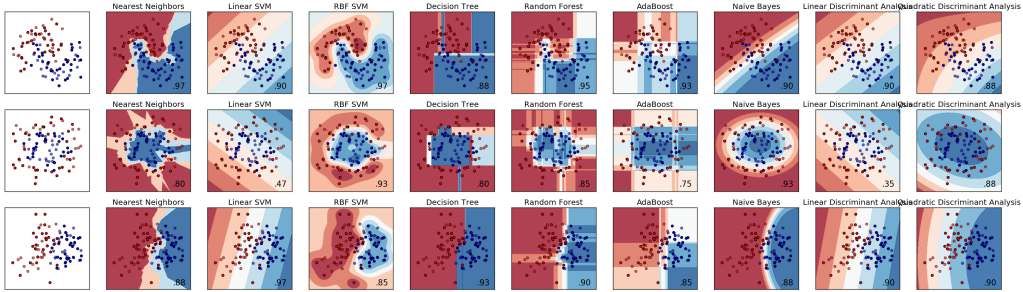
Performance Comparisons

MODEL	1ST	2ND	3RD	4TH	5TH	6TH	7TH	8TH	9TH	10TH
BST-DT	0.580	0.228	0.160	0.023	0.009	0.000	0.000	0.000	0.000	0.000
RF	0.390	0.525	0.084	0.001	0.000	0.000	0.000	0.000	0.000	0.000
BAG-DT	0.030	0.232	0.571	0.150	0.017	0.000	0.000	0.000	0.000	0.000
SVM	0.000	0.008	0.148	0.574	0.240	0.029	0.001	0.000	0.000	0.000
ANN	0.000	0.007	0.035	0.230	0.606	0.122	0.000	0.000	0.000	0.000
KNN	0.000	0.000	0.000	0.009	0.114	0.592	0.245	0.038	0.002	0.000
BST-STMP	0.000	0.000	0.002	0.013	0.014	0.257	0.710	0.004	0.000	0.000
DT	0.000	0.000	0.000	0.000	0.000	0.000	0.004	0.616	0.291	0.089
LOGREG	0.000	0.000	0.000	0.000	0.000	0.000	0.040	0.312	0.423	0.225
NB	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.030	0.284	0.686

Table 16.3 Fraction of time each method achieved a specified rank, when sorting by mean performance across 11 datasets and 8 metrics. Based on Table 4 of (Caruana and Niculescu-Mizil 2006). Used with kind permission of Alexandru Niculescu-Mizil.

From (Murphy, 2012, p.585)

Classification Performance Comparisons



From http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html

i

Lecture Outline

Boosting the Power of our Ensemble

Now that we have know about

- DecisionTrees
- Boosting and
- RandomForests

we are ready to learn about a powerful combination of all these concepts with gradient search, **Gradient Tree Boosting**.

Boosting Your Success

People realized that the very successful `Boosting` method was in essence **a very general meta-algorithm** for *optimization of the mapping function* from input variables to output target variables.
This algorithm *combines mutiple weak functions* together just as the ensemble of decision trees do for Random Forests

What's the Gradient of a Tree?

Does this function have a gradient?

Gradient in terms of parameters θ of $f(X|\theta)$

An the infinistesimal changeincrease in each of the function parameters that would strengthen the current response.

What's the Gradient of a Tree?

Does this function have a gradient?

Gradient in terms of parameters θ of $f(X|\theta)$

An the infinistesimal changeincrease in each of the function parameters that would strengthen the current response.

Question: What is the gradient of an *ensemble of decision trees*?

What's the Gradient of a Tree?

Does this function have a gradient?

Gradient in terms of parameters θ of $f(X|\theta)$

An the infinistesimal changeincrease in each of the function parameters that would strengthen the current response.

Question: What is the gradient of an *ensemble of decision trees*?

Answer:

- Why do we need to refer to *parameters* at all?

Does Some Model Know This Already?

- Random Forests does not use a gradient, just random sampling
- We could say that *AdaBoost uses a gradient implicitely* in a simple way:
  - A **gradient** is a direction to increase the output value of a function.
  - Each new decision tree (weak learner) is *optimized to make a small improvement in the overall function (the strong learner)*
- This is kind of like adjusting f relative to the negative of the "gradient".
  - Even though in AdaBoost there is no gradient explicitely calculated.
  - It simply does this by trying to do *well* on what the existing model does *badly* on

Weak Learners for the Win!

Now we generalize this idea:

- For each new **weak learner** m
  - we can explicitely treat it as an update $h_m(\mathbf{X})$ that attempts to step *away* along the gradient of the current combined function
  - then we add the resulting model to the **strong model** $F(\mathbf{X})$ being built up iteratively

At any time we have the current strong model which is already composed of m weak learners which we can refer to on...

- on the entire dataset $\mathcal{D}(\mathbf{X}, \mathbf{y}) : F_m(\mathbf{X}) = \hat{\mathbf{y}}$
- or, on an individual training point $(x_i, y) \in \mathcal{D} : F_m(x_i) = \hat{y}_i$

The Best New Weak Learner is the Residual

Thus the ideal update to F_m for the next weak learner h_m would be...

$$h_m(x_i) = y_i - F_m(x_i)$$

This is because then the new combined model would be

$$F_{(m+1)}(x_i) = F_m(x_i) + h_m(x_i) = y_i$$

which perfectly matches the data.

Approaching But Never Reaching

- In practice we need to be satisfied with merely *approaching* this ideal update.
- In this case the approximation is simply the sum of the wrong answers (i.e. the residuals) from each weak learner decision tree
- We'll use a functional gradient descent approach on an *approximation of the true residual*, which we'll call the **loss function**, at each step.
- We'll aim for minimizing the **mean squared error** of this loss:

$$\mathcal{L}_{MSE}(y_i, F_m(x_i)) = \frac{1}{n}\left(\sum_i y_i - F_m(X)\right)^2$$

How Gradient Tree Boosting Works 1

- Gradient Tree Boosting explicitely uses the negative gradient of the loss with respect to $F_m(x_i)$:

$$\begin{aligned}\nabla \mathcal{L}_{MSE}(y_i, F_m(x_i)) &= \frac{1}{n}\left(\sum_i y_i - F_m(X)\right)^2 \\ &= \frac{2}{n}(y_i - F_m(x_i)) \\ &= \frac{2}{n} h_m(x_i)\end{aligned}$$

So the gradient is proportional to our new weak learner! *how convenient...*

How Gradient Tree Boosting Works 2

- *Recall:* The gradient tells us how to change a function to *increase* its value. But our function is a loss function, inf the form of MSE, so we want to *decrease* it.
- Thus, we follow the **negative gradient**

$$h_m(x_i) = F_m(x_i) - \sum_{i}^{n} \frac{2}{n} h_m(x_i)$$

- There is also further optimization of a weighting function parameter γ for each trees.
- The popular algorithm **XGBoost** implements this algorithm.

Gradient Boosting Algorithm

Input: training set $\{(x_i, y_i)\}_{i=1}^n$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^n$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

Lecture Outline

Extremely Randomized Trees

From (Geurts, Ernst and Wehenkel, 2006)

- Randomly choose a subset of features
- Randomly choose a cut point in each of those features (not by searching)
- Choose the best performing of those features as the split (maximize a score, could be impurity)
  - **Totally Randomized Trees:** just choose one at random, no maximization (faster, more robust, but bad for classification)
- Stop when minimize number of datapoints for a leaf K is reached

Extremely Randomized Trees

Two main differences with other tree-based ensemble methods:

- Splits nodes by choosing cut-points fully at random
- Uses the whole learning sample (rather than a bootstrap replica) to grow the trees

Lecture Outline

Tree Ensemble Methods for *Streaming Data*

- Hoeffding Trees (Domingos, 2000): streaming data for single trees for regression, confidence bounds. Nodes only split when enough data arrives for confidence level.
- Mondrian Forests (Lakshminarayanan, 2014): streaming data for forests, confidence bounds

Hoeffding Trees

- A **Hoeffding Tree** learns a regression tree model but stores no data points along the way.
- New points help build confidence in the split at each node, and once the confidence is high enough, the node is split.
- Later arriving points are used to start learning the new leaves which could in turn become new internal nodes.
- While Hoeffding trees perform online learning and scale well for regression and classification, they use a fairly complex incremental approach to tree building.

propriate attributes there, and so on recursively.[1] We solve the difficult problem of deciding exactly how many examples are necessary at each node by using a statistical result known as the *Hoeffding bound* (or additive Chernoff bound)

[7, 9]. Consider a real-valued random variable r whose range is R (e.g., for a probability the range is one, and for an information gain the range is $\log c$, where c is the number of classes). Suppose we have made n independent observations of this variable, and computed their mean $\overline{r}$. The Hoeffding bound states that, with probability $1-\delta$, the true mean of the variable is at least $\overline{r}-\epsilon$, where

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{1}$$

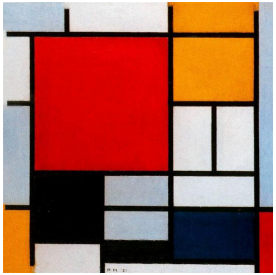
Algorithm 2 Hoeffding tree induction algorithm.

Let HT be a tree with a single leaf (the root)
for all training examples **do**
  Sort example into leaf l using HT
  Update sufficient statistics in l
  Increment n_l, the number of examples seen at l
  if $n_l \ mod \ n_{min} = 0$ **and** examples seen at l not all of same class **then**
    Compute $\overline{G}_l(X_i)$ for each attribute
    Let X_a be attribute with highest $\overline{G}_l$
    Let X_b be attribute with second-highest $\overline{G}_l$
    Compute Hoeffding bound $\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}}$
    if $X_a \neq X_\emptyset$ **and** $(\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ **or** $\epsilon < \tau)$ **then**
      Replace l with an internal node that splits on X_a
      for all branches of the split **do**
        Add a new leaf with initialized sufficient statistics
      end for
    end if
  end if
end for

Mondrian Processes and Mondrian Trees

- *Mondrian processes* are families of random, hierarchical, binary partitions and probability distributions over tree data structures (Roy, Daniel and Teh, 2008).
- In a *Mondrian Tree* every node r has a *split time* τ_r
- τ_r increases with the *depth* of the node
  but the increase is sampled *stochastically*

  root=0 $\rightarrow$ leaves=∞
- A *Mondrian forest* (Lakshminarayanan, Roy and Teh, 2014) is an ensemble of *Mondrian trees*



"Composition with Large Red Plane, Yellow, Black, Gray and Blue"

Piet Mondrian, 1921

Visualization of Mondrian Forests

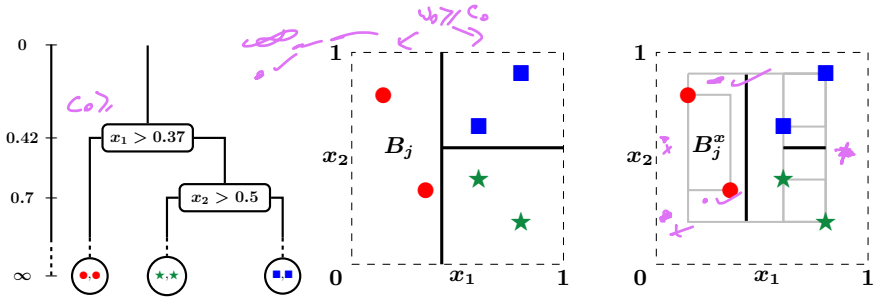


Figure 1: (left) A Mondrian tree over six *data points* in $[0,1]^2$. Every node in the tree represents a split and is embedded in time (vertical axis). **(middle)** An ordinary decision tree partitions the whole space. **(right)** In a Mondrian tree, splits are committed only within the range of the data in each block (denoted by gray rectangles). Let $j = \mathsf{left}(\epsilon)$ be the left child of the root: then $B_j = (0, 0.37] \times (0, 1]$ is the block containing the red circles and $B_j^x \subseteq B_j$ is the smallest rectangle enclosing the two data points. (Adapted from [13], with permission.)

Mondrian Trees for Tasks

- Mondrian trees can be updated with new data making them suitable for online streaming domains.
- Mondrian Forests can be used for *regression* or *classification*
- While Mondrian processes are *infinite structures*, Mondrian trees are restrictions of Mondrian processes on a *finite set of points*.

Lecture Outline

Streaming Updates

- As with other tree-based algorithms, each new data moves down the tree and is *checked for presence in current block*
  - **If it is present:** continue down tree just as in decision trees
  - **If it isn't:** compute the lower and upper errors for the block

Algorithm

- Every node r in the Mondrian tree has a *split time* τ_r which increases with the depth of the node. The split time is zero at the root and infinite at the leaves of the tree.
- When a new data point arrives, it is checked whether it belongs to an existing block or not.
- If not, the lower and upper errors (deviations) from the block are calculated.
- Again, a random variable is sampled from an exponential distribution with a rate which is a function of the lower and upper errors.
- As before, the split time of the new node is calculated and depending on it, its location in the tree is determined.
- In this way, new nodes can be added in the middle of a tree and not just grown at the end of tree like in regular online random forests.

Uses

- Note that the Mondrian forest is designed for *classification* and *regression* so its authors propose an analysis for smooth posterior updates in the blocks.
- However, the Mondrian forest can be used for *unsupervised* purposes where the posterior analysis and the pausing process can be bypassed.
- This is useful **anomaly detection** which is primarily an unsupervised task.