Meta Learning II

CS771: Introduction to Machine Learning

Purushottam Kar

Announcements

Students who have been granted proration for any quiz/exam should have received an email (at their CC email addresses) today.

If you missed a quiz/exam for reasons that make you eligible for proration, but have not received an email from the instructor, please contact the instructor immediately to clear the discrepancy.

https://web.cse.iitk.ac.in/users/purushot/courses/ml/2019-20-a/policies.php (see Absentee, Make-up, and Proration Policy)

Recap of Last Lecture

Distinction between model and model parameters

Notion of models having hyperparameters and model classes

Model selection techniques: held out/k-fold validation

Bias-variance tradeoffs and generalization error

Overfitting and underfitting

Tweaking model complexity, feats, train set size, ML algo as possible solutions

Ensemble ML Algorithms

Voting Ensemble: flexible but cannot ensure success as "experts" may sync

Need to ensure experts are diverse so that they can correct each other's errors

Bagging and boosting take direct steps to ensure this

Bagging – Bootstrap AGGregatING

Usually used to reduce variance of a model

Given train set S with n points, first sample n points ***with replacement*** *from S (call this S_1). Repeat K times to get "bagged" datasets $S_1, S_2, \dots, S_K$*

Learn a model $f_i: \mathcal{X} \to \{-1,1\}$ using dataset S_i (maybe using the same algo)

Benefit: learning is parallelizable – e.g. the K models can be learnt on separate cores

Predict a new point $x \in \mathcal{X}$ using $\hat{f}_{\text{MAJ}}(x) = \text{sign}\left(\sum_{k=1}^{K} f_i(x)\right)$

Can show that only ~63% of S lands up in any S_i i.e. S_i are diverse

Even if we have a high variance method that overfits, it will overfit to diverse sets and when using majority voting at the end, errors may get cancelled out

Does not reduce bias – bagging is usually applied where variance is a problem i.e. model is too powerful so bias is not a problem to begin with

Can be seen as performing (implicit) regularization – no change to ML algo

Dropout, random forests are instances of bagging idea being applied

Random Forests

A collection of decision trees is called a decision forest

Suppose we have train data $S = \{(\mathbf{x}^i, y^i)\}_{i=1,\ldots,n}$ *with* $x^i \in \mathbb{R}^d, y^i \in \{-1,1\}$

Random forests learn K *decision trees. First, bagging done to get* $S_1, \ldots, S_K$

Next, feature bagging done – sample $F_1, F_2, \ldots F_K \subset [d]$*, each of size* $d' \sim \sqrt{d}$

Features are sampled for each F_i **without replacement** – no repeated feature in any F_i

Learn the k*-th DT on dataset* S_k *using only the features in* F_k

Intuition behind feature bagging: force trees to learn without certain features being available otherwise if a feature is really convenient, every tree will use it and then all trees will behave similarly

Similar to the idea used in dropout

Sometimes random forests are used without feature bagging too

Random forests are very successful (state of the art) in recommendation

Dropout

Dropout for NNs can be seen as an extreme form of feature bagging

- *If the NN has N nodes then dropout tries to train all 2^N possible subnetworks*
- *However, whereas in random forests, different trees usually have very different parameter values, dropout wants all these 2^N subnetworks to share the parameters (edge weights)*
  - Two subnetworks that contain the same edge must have the same weight on that edge
  - Good idea to conserve model size and also prevent overfitting
  - Intractable to execute explicitly which is why dropout does this approximately.
- *At every time step t a random subnetwork trained using a mini-batch of data*
- *Similarly, at test time, an approximation used instead of asking all 2^N subNNs*
  - Simply downweigh the output of a neuron using the probability of it not going missing
  - Random forests on the other hand, do explicitly ask every tree for their verdict

Boosting

Voting ensembles, bagging *hope* that various models in the ensemble will correct each other's errors – no explicit effort to ensure this

- *Models are trained independently in bagging for sake of speed*

Boosting rectifies this: learns ensembles that explicitly correct errors

- *k^{th} model is asked to correct the mistakes of the previous $k-1$ models*
- *Adaboost (Adaptive Boosting) [Freund-Schapire] – won the Gödel prize*
- *Popularly used to perform bias reduction (bagging does variance reduction)*

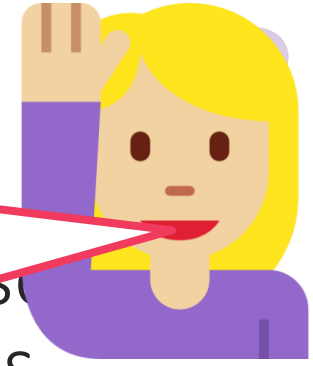
Adaboost requires a *weak learner* or a *stump* as its base algorithm

- *Typically a very fast ML algo that need only assure modest accuracy*
  - For example, assure at least 51% classification accuracy in a binary classification task
  - Only catch is that Adaboost requires algorithm to work with weights on data points
- *Theoretical results show that Adaboost can construct extremely accurate classifiers (e.g. 99% accuracy) out of these weak classifiers*

Boosting

Given a dataset $S = \{(x^i, y^i, w^i)\}_{i=1,\dots,n}$ with $x^i \in \mathcal{X}, y^i \in \{-1,1\}, w^i \in [0,1]$,

Adaboost requires a weak learning algo/stump $\mathcal{A}$ so that $\mathcal{A}(S) = f: \mathcal{X} \to \{-1,1\}$ s.t. $\sum_{i=1}^{n} w^i \cdot \mathbb{I}\{f(x^i) \neq y^i\} \leq 0.49$ or so.



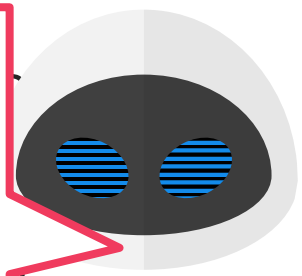
Models are trained independently in bagging for sake of speed

Boosting rectifies this: lea

k^{th} model is asked to corre

Adaboost (Adaptive Boost

Sometimes may need to iterate through the above experiences (experience high bias, reduce it only to increase variance, then decrease variance etc) before reaching a sweet spot



Popularly used to perform bias reduction (bagging does variance reduction)

Adaboost requ

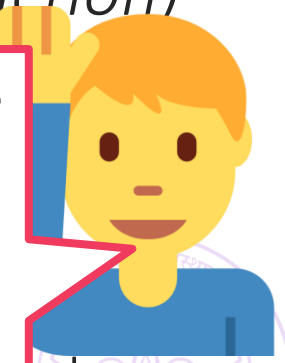
Typically a ve

For example,

Only catch is that Adaboost requires algorithm to work with weights on data points

Adaboost not used with strong learners e.g. 18 layer DCNN since

1. Powerful models usually have tiny bias (can memorize data) – the problem with powerful models is variance and not bias
2. Training usually NP-hard, time consuming for such models



Theoretical results show that Adaboost can construct extremely accurate classifiers (e.g. 99% accuracy) out of these weak classifiers

AdaBoost

ADABOOST

1. Data $S = \{(\mathbf{x}^i, y^i)\}_{i=1,\dots,n}$, stump learner
2. Assign initial weights $w^i = \frac{1}{n}$ for $i \in [n]$
3. For $t = 1,2, \dots$
   1. Use $(S, \{w^i\})$ to learn $f_t : \mathcal{X} \to \{-1,1\}$
   2. Let $\epsilon_t \leftarrow \sum_{i=1}^{n} w^i \cdot \mathbb{I}\{f_t(\mathbf{x}^i) \neq y^i\}$
   3. Let $\alpha_t \leftarrow \frac{1}{2} \cdot \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
   4. For each data point $i = 1,2, \dots, n$
      1. Let $u^i \leftarrow w^i \cdot \exp\left(-\alpha_t y^i f_t(\mathbf{x}^i)\right)$
   5. Let $w^i \leftarrow u^i / \sum_j u^j$
4. Repeat until convergence

Final classifier for Adaboost

$\mathbf{sign}(\sum_{t=1}^{T} \alpha_t f_t(\mathbf{x}))$

α_t tells us how useful f_t is

If $\epsilon_t \to 0$ then $\alpha_t \to \infty$

More accurate f_t trusted more

If $\epsilon_t = 0.5$ then $\alpha_t = 0$

A random classifier has no use!

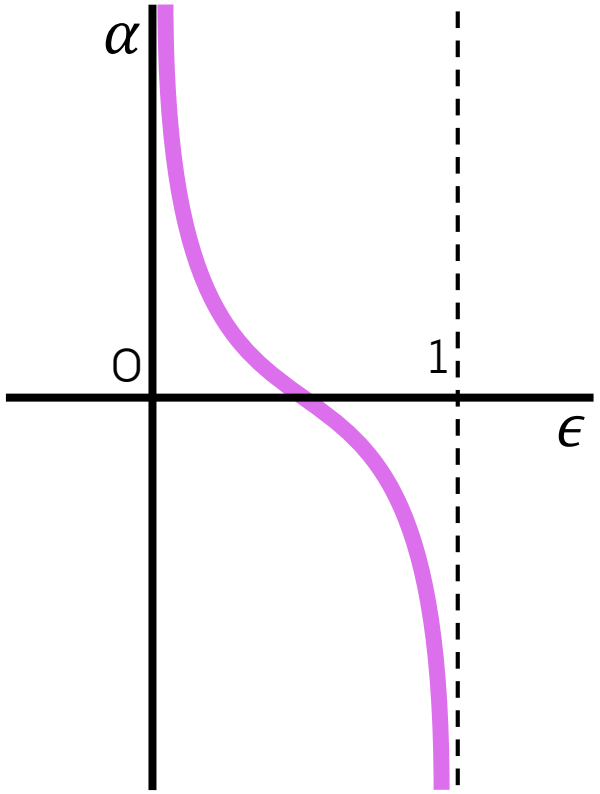
If $\epsilon_t > 0.5$ then $\alpha_t < 0$

The classifier is wrong more often than it is correct. Better to invert its decision ☺

If $\epsilon_t \to 1$ then $\alpha_t \to -\infty$

Inverted classifier is very good

Step 5 normalizes weights



ADABOOST

$\{(\mathbf{x}^i, y^i)\}_{i=1,\ldots,n}$, stump learner

Weights $w^i = \frac{1}{n}$ for $i \in [n]$

For $t = 1, \ldots, T$ (T stumps) to learn $f_t: \mathcal{X} \to \{-1, 1\}$

Let $\epsilon_t \leftarrow \sum_{i=1}^{n} w^i \cdot \mathbb{I}\{f_t(\mathbf{x}^i) \neq y^i\}$

3. Let $\alpha_t \leftarrow \frac{1}{2} \cdot \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
4. For each data point $i = 1, 2, \ldots, n$
   1. Let $u^i \leftarrow w^i \cdot \exp\left(-\alpha_t y^i f_t(\mathbf{x}^i)\right)$
5. Let $w^i \leftarrow u^i / \sum_j u^j$

4. Repeat until convergence

Final classifier for Adaboost

$\mathbf{sign}\left(\sum_{t=1}^{T} \alpha_t f_t(\mathbf{x})\right)$

α_t *tells us how useful* f_t *is*

If $\epsilon_t \to 0$ *then* $\alpha_t \to \infty$

More accurate f_t trusted more

If $\epsilon_t = 0.5$ *then* $\alpha_t = 0$

A random classifier has no use!

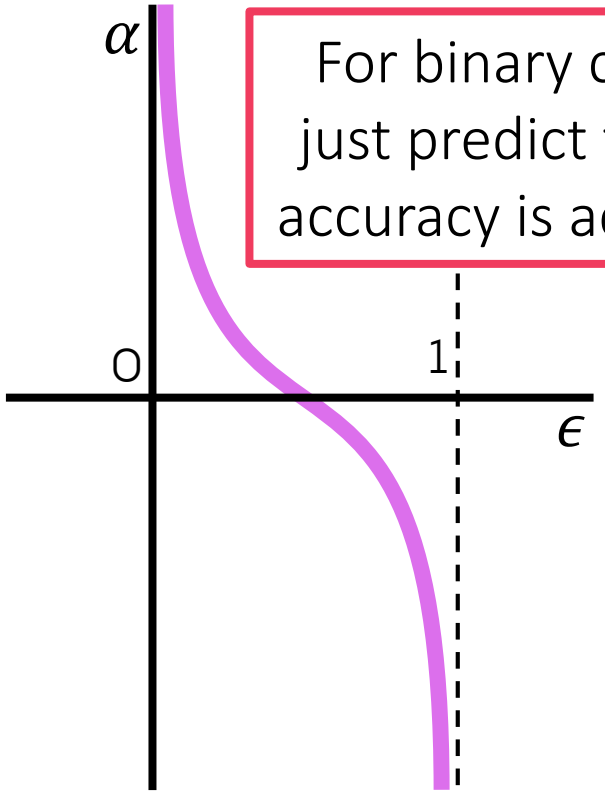
If $\epsilon_t > 0.5$ *then* $\alpha_t < 0$

The classifier is wrong more often than it is correct. Better to invert its decision ☺

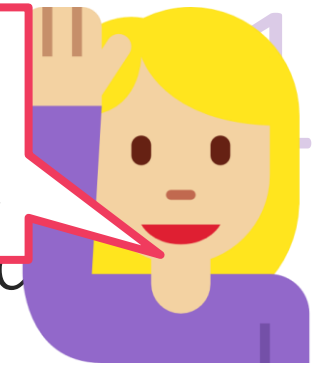
If $\epsilon_t \to 1$ *then* $\alpha_t \to -\infty$

Inverted classifier is very good

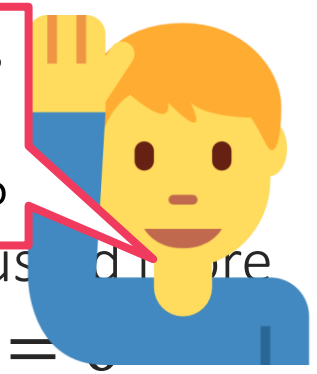
Step 5 normalizes weights



For binary classification, we can trivially get more than 50% accuracy by just predict the more common of the two labels. A classifier that gets 1% accuracy is actually very good since its inverted form will get 99% accuracy



Do the above statements also hold for multiclass problems? How would the above statements have to change to make them hold for C classes?



$\{(\mathbf{x}^i, y^i)\}_{i=1,\dots,n}$, stump learner

ights w^i

) to learn $f_t: \mathcal{X} \to \{-1,1\}$

$\sum_{i=1} w^i \cdot \mathbb{I}\{f_t(\mathbf{x}^i) \neq y^i\}$

3. Let $\alpha_t \leftarrow \frac{1}{2} \cdot \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
4. For each data point $i = 1,2,\dots,n$
   1. Let $u^i \leftarrow w^i \cdot \exp\left(-\alpha_t y^i f_t(\mathbf{x}^i)\right)$
5. Let $w^i \leftarrow u^i / \sum_j u^j$

4. Repeat until convergence

Final classifier for Ad

$\text{sign}(\sum_{t=1}^{T} \alpha_t f_t(\mathbf{x}))$

More accurate f_t trus d more

If $\epsilon_t = 0.5$ *then* $\alpha_t =$

A random classifier has no use!

If $\epsilon_t > 0.5$ *then* $\alpha_t < 0$

The classifier is wrong more often than it is correct. Better to invert its decision ☺

If $\epsilon_t \to 1$ *then* $\alpha_t \to -\infty$

Inverted classifier is very good

Step 5 normalizes weights

Adaboost

Adaboost prioritizes points that are misclassified

If $y^i \neq f_t(\mathbf{x}^i)$ *then* $u^i \leftarrow w^i \cdot \exp(\alpha_t)$ *i.e. weight increased if* $\alpha_t > 0$

If $y^i = f_t(\mathbf{x}^i)$ *then* $u^i \leftarrow w^i \cdot \exp(-\alpha_t)$ *i.e. weight decreased if* $\alpha_t > 0$

Note that final classifier is $\mathbf{sign}(\sum_{t=1}^{T} \alpha_t f_t(\mathbf{x}))$ where f_t gives actual ± 1 labels and not just scores

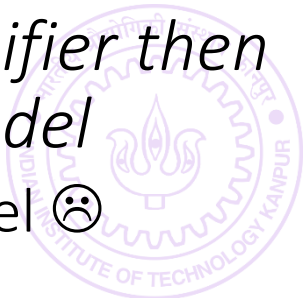
If we use linear classifiers as stumps, final classifier would look like $f(\mathbf{x}) = \mathbf{sign}(\sum_{t=1}^{T} \alpha_t \cdot \mathbf{sign}(\langle \mathbf{w}^t, \mathbf{x} \rangle))$

Note that Adaboost classifier with linear stumps looks almost like a 2 layer NN with sign function as the activation function in the hidden and output layers ☺

Caution*: if we instead use* $g(\mathbf{x}) = \mathbf{sign}(\sum_{t=1}^{T} \alpha_t \langle \mathbf{w}^t, \mathbf{x} \rangle)$ *as final classifier then* $g(\mathbf{x}) = \mathbf{sign}(\langle \tilde{\mathbf{w}}, \mathbf{x} \rangle)$ *where* $\tilde{\mathbf{w}} = \sum_{t=1}^{T} \alpha_t \cdot \mathbf{w}^t$ *i.e. we get a linear model*

No use of running Adaboost – SVMs could anyway give us the best linear model ☹

Thus, the non-linearity given by $\mathbf{sign}(\quad)$ is crucial to the success of Adaboost



Gradient Boosting

Suppose we learn a (nonlinear) regression model $f(\mathbf{x})$ to predict y

If $f(\mathbf{x}^i) \approx y^i \; \forall i$ then done else let $r(\mathbf{x}^i) = y^i - f(\mathbf{x}^i)$ denote residual error

Learn a model $g(\mathbf{x}^i)$ to predict $r(\mathbf{x}^i)$. If $g(\mathbf{x}^i) \approx r(\mathbf{x}^i)$ for all data points then done

Else let $\tilde{r}(\mathbf{x}^i) = r(\mathbf{x}^i) - g(\mathbf{x}^i)$ and learn another model $h(\mathbf{x}^i)$ to approximate $\tilde{r}(\mathbf{x}^i)$

Repeat till residuals get acceptably small. Final model is $f(\mathbf{x}^i) + g(\mathbf{x}^i) + h(\mathbf{x}^i) + \cdots$

Such "residual" learning techniques are useful in deep learning as well

Gradient Boosting generalizes this to general (non-regression) settings

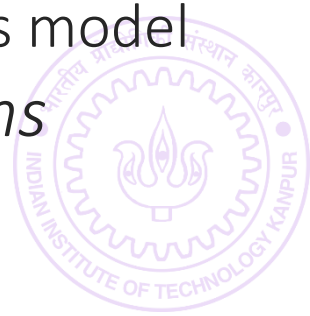
***Note**: if $\ell(f(\mathbf{x}^i), y^i) = (f(\mathbf{x}^i) - y^i)^2$ then $y^i - f(\mathbf{x}^i) = -\frac{1}{2} \cdot \frac{d\ell}{df(\mathbf{x}^i)}$*

Train the next model (stump) to learn the gradients of the loss on the previous model

Gradient-boosted Decision Trees GBDT are successful implementations

XGBoost (eXtreme Gradient Boosting) is currently a popular package

Provides very efficient parallelized implementations



Gradient Boosting

GB secretly tries to do GD ☺

GRADIENT BOOSTING

1. Data $S = \{(\mathbf{x}^i, y^i)\}_{i=1,\ldots,n}$, stump learner
2. Learn a const model $f_0 = \arg\min_{\alpha} \ell(\alpha, S)$
3. For $t = 1,2, \ldots$
   1. Get pseudo residuals $r^i = -\frac{d\ell(f_{t-1}(\mathbf{x}^i), y^i)}{df_{t-1}(\mathbf{x}^i)}$
   2. Learn h_t to fit data $R = \{(\mathbf{x}^i, r^i)\}_{i \in [n]}$
   3. Find the importance of the new stump $\alpha_t = \arg\min_{\alpha} \ell(f_{t-1} + \alpha \cdot h_t, S)$
   4. Expand the ensemble $f_t = f_{t-1} + \alpha_t \cdot h_t$

If possible, it would've done

$$f_t = f_{t-1} - \eta_t \cdot \frac{d\ell}{df}\Big]_{f_{t-1}}$$

Since taking gradients w.r.t functions is intractable, GB settles for an approximation

Approx. $-d\ell/df]_{f_{t-1}}$ *using* h_t

Search for best step length η_t

Step 3 does this implicitly

The final ensemble is actually the result of approximate GD in the function space!

Grad

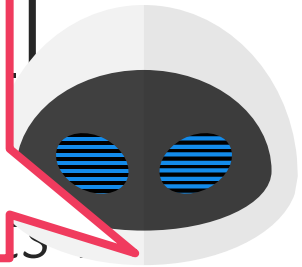
Common for first stump in Gradient Boosting (GB) to act as bias i.e. be a constant predictor i.e. $f(\mathbf{x}) = c$ for all $\mathbf{x}$. This can be thought of as giving importance c to the all ones predictor that predicts $f(\mathbf{x}) = 1$ on all $\mathbf{x}$

GB secretly tries to d

If possible, it would've done

GRADIENT BOOSTING

Gradient of the loss function would be large on data points which are experiencing large loss so GB would pay more attention to them while learning the next stump – similar intuition as Adaboost had



1. Data $S =$
2. Learn a
3. For $t = 1,2, \ldots$
   1. Get pseudo residuals $r^i = -\frac{d\ell(f_{t-1}(\mathbf{x}^i), y^i)}{df_{t-1}(\mathbf{x}^i)}$
   2. Learn h_t to fit data $R = \{(\mathbf{x}^i, r^i)\}_{i \in [n]}$
   3. Find the importance of the new stump $\alpha_t = \arg\min_{\alpha} \ell(f_{t-1} + \alpha \cdot h_t, S)$
   4. Expand the ensemble $f_t = f_{t-1} + \alpha_t \cdot h_t$

functions is intractable, GB settles for an approximation

Approx. $-d\ell/df]_{f_{t-1}}$ *using* h_t

Search for best step length η_t

Step 3 does this implicitly

The final ensemble is actually the result of approximate GD in the function space!