Machine Learning (CSE 446): Concepts & the "i.i.d." Supervised Learning Paradigm

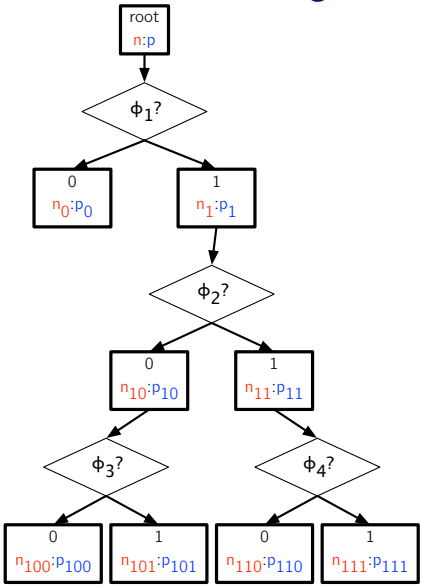
Sham M Kakade

© 2018

University of Washington
cse446-staff@cs.washington.edu

Review

Decision Tree: Making a Prediction



Data: decision tree t, input example x
Result: predicted class
if t *has the form* LEAF(y) **then**
    return y;
else
    # $t.\phi$ is the feature associated with t;
    # t.child(v) is the subtree for value v;
    return DTREETEST$(t.\text{child}(t.\phi(x)), x)$;
end

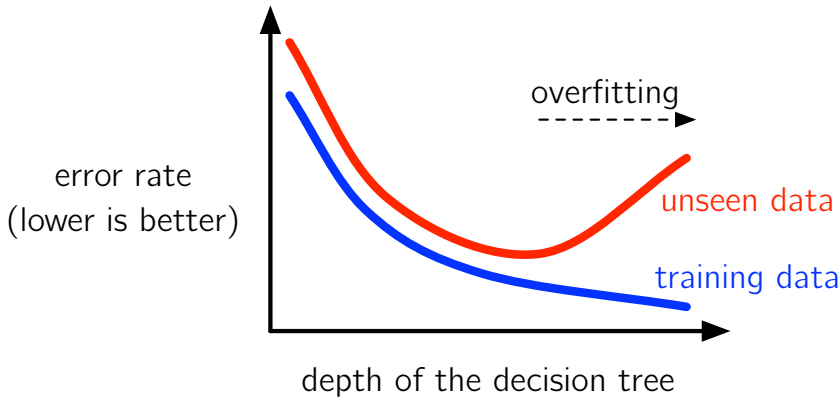
Algorithm 1: DTREETEST

(review) Greedily Building a Decision Tree (Binary Features)

Data: data D, feature set Φ
Result: decision tree
if *all examples in D have the same label y, or Φ is empty and y is the best guess* **then**
    return $\textsc{Leaf}(y)$;
else
    for *each feature ϕ in Φ* **do**
        partition D into D_0 and D_1 based on ϕ-values;
        let mistakes(ϕ) = (non-majority answers in D_0) + (non-majority answers in D_1);
    end
    let ϕ^* be the feature with the smallest number of mistakes;
    return $\textsc{Node}(\phi^*, \{0 \to \textsc{DTreeTrain}(D_0, \Phi \setminus \{\phi^*\}), 1 \to \textsc{DTreeTrain}(D_1, \Phi \setminus \{\phi^*\})\})$;
end

Algorithm 2: $\textsc{DTreeTrain}$

Danger: Overfitting



Today's Lecture

The "i.i.d." Supervised Learning Setup

- Let ℓ be a **loss function**; $\ell(y, \hat{y})$ is our loss by predicting $\hat{y}$ when y is the correct output.
- Let $\mathcal{D}(x, y)$ define the (unknown) underlying probability of input/output pair (x, y), in "nature." **We never "know" this distribution.**
- The training data $D = \langle (x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N) \rangle$ are assumed to be **identical, independently, distributed** (i.i.d.) samples from $\mathcal{D}$.
- We care about our **expected error (i.e. the expected loss, the "true" loss, ...)** with regards to the underlying distribution $\mathcal{D}$.
- Goal: find a **hypothesis** which as has "low" expected error, using the training set.

Concepts and terminology

set of depth of decision trees

- The **learning algorithm** maps the training set D to a some hypothesis $\hat{f}$.
  - often have a "**hypothesis class**" $\mathcal{F}$, where our algorithm chooses $\hat{f} \in \mathcal{F}$.
- The **training error** of f is the loss of f on the training set.
- **overfitting!** (and **underfitting**)
  Also: The **generalization error** is often referred to as the difference between the training error of $\hat{f}$ and the expected error of $\hat{f}$.
- Ways to check/avoid overfitting:
  - use **test set**, i.i.d. data sampled $\mathcal{D}$, to estimate the the **expected error**.
  - use a "**Dev**elopment **set**", i.i.d. from $\mathcal{D}$, for hyperparameter turning (or **cross validation**)
- We really just get sampled data, and **we can break it up as we like**.

Loss functions

- $\ell(y, \hat{y})$ is our loss by outputting $\hat{y}$ when y is the correct output.
- Many loss functions:
  - For binary classification, where $y \in \{0, 1\}$:
  $$\ell(y, \hat{y}) = [\![y \neq \hat{y}]\!]$$
  - For multi-class classification, where y is one of k-outcomes:
  $$\ell(y, \hat{y}) = [\![y \neq \hat{y}]\!]$$
  - For regression, where $y \in \mathbb{R}$, we often use the square loss:
  $$\ell(y, \hat{y}) = (y - \hat{y})^2$$
- Classifier f's true **expected error (or loss)**:
$$\epsilon(f) = \sum_{(x,y)} \mathcal{D}(x, y) \cdot \ell(y, f(x)) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y, f(x))]$$

Sometimes, when clear from context, the loss or error refers to the expected loss.

Training error

- Goal: We want to find an f which has low $\epsilon(f)$.
  But we don't know $\epsilon(f)$?
- The **training error** of hypothesis f is f's average error on the **training data**:

$$\hat{\epsilon}(f) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(x_n))$$

- In contrast, classifier f's **true** expected loss:

$$\epsilon(f) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y, f(x))]$$

- Idea: Use the training error $\hat{\epsilon}(f)$ as an empirical approximation to $\epsilon(f)$.
  And hope that this approximation is good!

The training error and the LLN

- For a **fixed** f (which does not depend on the training set D), the training error is an unbiased estimate of the expected error.
  Proof: Taking an expectation over the dataset D

$$\mathbb{E}_{\mathrm{D}}[\hat{\epsilon}(f)] = \mathbb{E}[\frac{1}{N}\sum_n \ell(y_n, f(x_n))] = \frac{1}{N}\sum_n \mathbb{E}\ell(y_n, f(x_n)) = \frac{1}{N}\sum_n \epsilon(f) = \epsilon(f)$$

Law of large numbers

- LLN: for a fixed f (not a function of D) and for large N, $\hat{\epsilon}(f) \to \epsilon(f)$
  e.g. for any fixed classifier, you can get a good estimate of its mistake rate with a large dataset..
- This suggests: finding f which makes the training error small is a good approach?

What could go wrong?

- A learning algorithm which "memorizes" the data is easy to construct:

consider decision stump on continuous "weight"

While such algorithms have 0 training error, they often have true expected error no better than guessing.

- What went wrong?
  - for a given f, we just need a training set to estimate the bias of a coin (for binary classification). this is easy.
  - **BUT there is a ("very small") chance this approximation fails (for "large N")**
  - try enough hypothesis, and, by chance alone, one will look good.

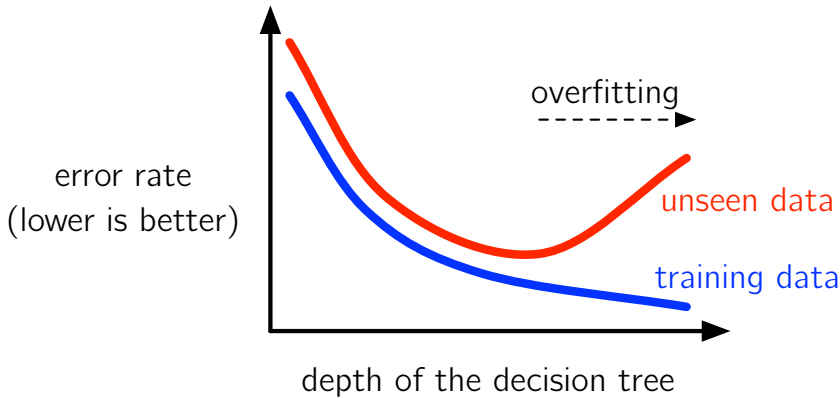
Overfitting, More Formally

- Let $\hat{f}$ be the output of training algorithm.
  - It is never true (in almost all cases) that $\hat{\epsilon}(\hat{f})$, the training error of $\hat{f}$, is an unbiased estimate $\epsilon(\hat{f})$, of the expected loss of $\hat{f}$.
  - It is usually a **gross underestimate**.
- The **generalization error** of our algorithm is:

$$\hat{\epsilon}(\hat{f}) - \epsilon(\hat{f})$$

  Large generalization error means we have overfit.
- We would like **both**:
  - our training error, $\hat{\epsilon}(\hat{f})$, to be small
  - our generalization error to be small
- If both occur, then we have low expected error :)
  - It is usually easy to get one of these two to be small.
  - **Overfitting: this is the fundamental problem of ML.**

Danger: Overfitting



Test sets and Dev. Sets

- Checking for overfitting:
  - use **test set**, i.i.d. data sampled $\mathcal{D}$, to estimate the the **expected error**.
  - We get an unbiased estimate of the true error (and an accurate one for "reasonable" N).
  - we should **never** use the test set during training, as this violates the approximation quality.
- **Hyperparameters** "def": params of our algorithm/pseudo-code
  1. usually they monotonically make training error lower
     e.g. decision tree maximal width and maximal depth.
  2. sometimes not we just don't know how to set them (e.g. learning rates)
- How do we set hyperparams? For case 1,
  - use a **dev set**, i.i.d. from $\mathcal{D}$, for hyperparameter turning
    (or **cross validation**)
  - learn with training set (using different hyperparams); then check on your dev set.

Back to decision trees . . .

Avoiding Overfitting by Stopping Early

- Set a maximum tree depth d_{max}.
  (also need to set a maximum width w)
- Only consider splits that decrease error by at least some Δ.
- Only consider splitting a node with more than N_{min} examples.

In each case, we have a **hyperparameter** $(d_{max}, w, \Delta, N_{min})$, which you should **tune** on **development data**.

Avoiding Overfitting by Pruning

- Build a big tree (i.e., let it overfit), call it t_0.
- For $i \in \{1, \ldots, |t_0|\}$: greedily choose a set of sibling-leaves in t_{i-1} to collapse that increases error the least; collapse to produce t_i.

  (Alternately, collapse the split whose contingency table is least surprising under chance assumptions.)
- Choose the t_i that performs best on development data.

More Things to Know

- Instead of using the number of mistakes, we often use information-theoretic quantities to choose the next feature.
- For continuous-valued features, we use thresholds, e.g., $\phi(x) \leq \tau$.
  In this case, you must choose τ.
  If the sorted values of ϕ are $\langle v_1, v_2, \ldots, v_N \rangle$, you only need to consider $\tau \in \left\{ \frac{v_n + v_{n+1}}{2} \right\}_{n=1}^{N-1}$ (midpoints between consecutive feature values).
- For continuous-valued **outputs**, what value makes sense as the prediction at a leaf? What loss should we use instead of $[\![y \neq \hat{y}]\!]$?