k-nn and decision trees

CS 446 / ECE 449

2021-03-09 17:47:17 -0600 (f7e1f60)
Today we’ll cover two standard machine learning methods.

Nearest neighbors (“$k$-nn”).

Decision trees.
**pytorch meta-algorithm.**

1. Clean/augment data (lecture 10?).
2. Pick model/architecture (anything from lectures 2-13).
3. Pick a loss function measuring model fit to data.
4. Run a gradient descent variant to fit model to data.
5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

$k$-nn and decision trees will **not** use GD!
1-nearest-neighbor (1-nn)

1. Pick a distance function \( \rho(\cdot, \cdot) \).

2. Memorize training set \( ((x_i, y_i))_{i=1}^n \).

3. Given \( x \), output label \( y_i \) of closest \( x_i \):

\[
\rho(x, x_i) = \min_j \rho(x, x_j).
\]

(Break ties arbitrarily but consistently.)

In this way, 1-nn uses the training set to form a Voronoi partition of the input space.
$k$-nearest-neighbor ($k$-nn)

1. Pick a distance function $\rho(\cdot, \cdot)$ and integer $k \geq 1$.
2. Memorize training set $((x_i, y_i))_{i=1}^n$.
3. Given $x$,
   - (classification case) output plurality label $y$ amongst $k$ closest training examples ("$k$" nearest neighbors).
     (In binary case, "plurality" = majority.)
   - (regression case) output average label $y$ amongst $k$ closest training examples ("$k$" nearest neighbors).
$k$-nearest-neighbor ($k$-nn)

1. Pick a distance function $\rho(\cdot, \cdot)$ and integer $k \geq 1$.
2. Memorize training set $((x_i, y_i))_{i=1}^n$.
3. Given $x$,
   ▶ (classification case) output plurality label $y$ amongst $k$ closest training examples ("$k$" nearest neighbors).
   (In binary case, "plurality" = majority.)
   ▶ (regression case) output average label $y$ amongst $k$ closest training examples ("$k$" nearest neighbors).

Remarks.
   ▶ If $(x_i)_{i=1}^n$ are distinct, 1-nn gets 0 training error.
   ▶ $k$-nn may fail to get 0 training error. (What is an example?)
   ▶ Why use $k$-nn?
pytorch meta-algorithm.

6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

- Here, $k$ and the distance function are the model hyper-parameters.
- 1-nn can have bad testing error.
- For carefully chosen $k$, e.g., $O(\ln n)$, $k$-nn is guaranteed to achieve optimal test error.
- Higher $k$ smooths the predictor, and gives a “less complex” model in an interesting way.
Task: classify handwritten digits into \( \{0, \ldots, 9\} \).

Digits from standard MNIST dataset (Lecun, Cortes, Burges).
Test error of $k$-nn with $\ell_2$ distance:

<table>
<thead>
<tr>
<th>OCR digits classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
</tr>
<tr>
<td>Test error rate</td>
</tr>
</tbody>
</table>

Test error of 1-nn with different distances:

<table>
<thead>
<tr>
<th>Distance</th>
<th>$\ell_2$</th>
<th>$\ell_3$</th>
<th>Tangent</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test error rate</td>
<td>3.09%</td>
<td>2.83%</td>
<td>1.10%</td>
<td>0.63%</td>
</tr>
</tbody>
</table>
\textbf{k-nn and features}

\textbf{Caution:} nearest neighbor classifier can be broken by bad/noisy features!

\textbf{Curse of dimension.} Given \( \text{poly}(d) \) random unit norm points in \( \mathbb{R}^d \), with probability \( > 99\% \), each is squared distance \( 2 \pm O \left( \frac{1}{\sqrt{d}} \right) \) from all others.

\textbf{Popular approach:} train a deep network \( f : \mathbb{R}^d \rightarrow \mathbb{R}^p \), and run \( k \)-nn on its outputs!
Naïve method for computing NN predictions: $O(n)$ distance computations.
Naïve method for computing NN predictions: $O(n)$ distance computations.

Better: organize training data in a data structure to improve look-up time.
Naïve method for computing NN predictions: \( O(n) \) distance computations.

Better: organize training data in a data structure to improve look-up time.

- Space: \( O(nd) \) for \( n \) points in \( \mathbb{R}^d \).
- Query time: \( O(2^d \log n) \) time in worst-case.
Naïve method for computing NN predictions: \( O(n) \) distance computations.

Better: organize training data in a data structure to improve look-up time.
- Space: \( O(nd) \) for \( n \) points in \( \mathbb{R}^d \).
- Query time: \( O(2^d \log n) \) time in worst-case.

Finding an “approximate” NN can be more efficient.
Naïve method for computing NN predictions: $O(n)$ distance computations.

Better: organize training data in a data structure to improve look-up time.
  - Space: $O(nd)$ for $n$ points in $\mathbb{R}^d$.
  - Query time: $O(2^d \log n)$ time in worst-case.

Finding an “approximate” NN can be more efficient.

E.g., how to quickly find a point among the top-1% closest points?
Naïve method for computing NN predictions: $O(n)$ distance computations.

Better: organize training data in a data structure to improve look-up time.

- Space: $O(nd)$ for $n$ points in $\mathbb{R}^d$.
- Query time: $O(2^d \log n)$ time in worst-case.

Finding an “approximate” NN can be more efficient.

E.g., how to quickly find a point among the top-1% closest points?

- Popular technique: Locality sensitive hashing
Decision trees

A decision tree is a binary tree which recursively partitions/refines the input space:

▶ Each tree node is associated with a splitting rule $g$:
  \[ X \to \{0, 1\} \]
  (interpreted as “recurse left” and “recurse right”).

▶ Each leaf node is associated with a label $\hat{y}$.

When $X = \mathbb{R}^d$, typically only consider splitting rules of the form

\[ g(x) = \{x_i > t\} \]

for some $i \in \{1, 2, ..., d\}$ and $t \in \mathbb{R}$.

Called axis-aligned or coordinate splits.

(Notation: $\{d\} := \{1, 2, ..., d\}$.)

$x_1 > 1.7$

$x_2 > 2.8$

$\hat{y} = 1$

$\hat{y} = 2$

$\hat{y} = 3$
A decision tree is a binary tree which recursively partitions/refines the input space:

- Each tree node is associated with a splitting rule $g: \mathcal{X} \to \{0, 1\}$ (interpreted as “recurse left” and “recurse right”).
- Each leaf node is associated with a label $\hat{y}$.

When $\mathcal{X} = \mathbb{R}^d$, typically only consider splitting rules of the form

$$g(\mathbf{x}) = \mathbb{1}\{x_i > t\}$$

for some $i \in [d]$ and $t \in \mathbb{R}$. Called axis-aligned or coordinate splits.

(Notation: $[d] := \{1, 2, \ldots, d\}$.)
Decision tree example

Classifying irises by sepal and petal measurements

- $\mathcal{X} = \mathbb{R}^2$, $\mathcal{Y} = \{1, 2, 3\}$
- $x_1 =$ ratio of sepal length to width
- $x_2 =$ ratio of petal length to width
Decision tree example

Classifying irises by sepal and petal measurements

- $\mathcal{X} = \mathbb{R}^2$, $\mathcal{Y} = \{1, 2, 3\}$
- $x_1 = \text{ratio of sepal length to width}$
- $x_2 = \text{ratio of petal length to width}$

$\hat{y} = 2$
Decision tree example

Classifying irises by sepal and petal measurements

- $\mathcal{X} = \mathbb{R}^2$, $\mathcal{Y} = \{1, 2, 3\}$
- $x_1 = \text{ratio of sepal length to width}$
- $x_2 = \text{ratio of petal length to width}$
Classifying irises by sepal and petal measurements

- $\mathcal{X} = \mathbb{R}^2$, $\mathcal{Y} = \{1, 2, 3\}$
- $x_1 = \text{ratio of sepal length to width}$
- $x_2 = \text{ratio of petal length to width}$

Decision tree example

$\hat{y} = 1$

$\hat{y} = 3$
Decision tree example

Classifying irises by sepal and petal measurements

- $\mathcal{X} = \mathbb{R}^2$, $\mathcal{Y} = \{1, 2, 3\}$
- $x_1 = \text{ratio of sepal length to width}$
- $x_2 = \text{ratio of petal length to width}$

$y = 1$ if $x_1 > 1.7$ and $x_2 > 2.8$
Decision tree example

Classifying irises by sepal and petal measurements

- $\mathcal{X} = \mathbb{R}^2$, $\mathcal{Y} = \{1, 2, 3\}$
- $x_1 = \text{ratio of sepal length to width}$
- $x_2 = \text{ratio of petal length to width}$

The decision tree is shown below:

- $x_1 > 1.7$
  - $\hat{y} = 1$
  - $x_2 > 2.8$
    - $\hat{y} = 2$
    - $\hat{y} = 3$
Basic decision tree learning algorithm

### Basic “top-down” greedy (training) algorithm.

- Pick an uncertainty measure $u$; the uncertainty of a tree $T$ is

$$u(T) := \frac{1}{n} \sum_{\text{leaf } S \in T} |S| \cdot u(S).$$

(Different $u(S)$ definitions are on the next slide.)

- Place all data in single root tree node. Initially, tree is a single leaf node containing all (training) data.

- Loop (until some stopping criterion is satisfied):
  - Pick the leaf $\ell$ and splitting rule $h$ that maximally reduces uncertainty of the current tree.
  - Split data in $\ell$ using $h$, and grow tree accordingly.

To predict on new data: traverse tree to corresponding leaf, output the plurality (or average) label of its training data.
Consider $S$ examples, $p|S|$ of which are labeled 1.

1. **Classification error**: 

   $u(S) := \min\{p, 1-p\}$

2. **Gini index**:

   $u(S) := 2p(1-p)$

3. **Entropy**:

   $u(S) := p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p}$

Gini index and entropy (after some rescaling) are concave upper-bounds on classification error.
Consider $S$ examples, $p_y |S|$ of which have label $y$.

1. **Classification error:**
   \[
   u(S) := 1 - \max_{y \in \mathcal{Y}} p_y
   \]

2. **Gini index:**
   \[
   u(S) := 1 - \sum_{y \in \mathcal{Y}} p_y^2
   \]

3. **Entropy:**
   \[
   u(S) := \sum_{y \in \mathcal{Y}} p_y \log \frac{1}{p_y}
   \]
Consider $S$ examples, $p_y|S|$ of which have label $y$.

1. **Classification error:**
   
   $$ u(S) := 1 - \max_{y \in \mathcal{Y}} p_y $$

2. **Gini index:**
   
   $$ u(S) := 1 - \sum_{y \in \mathcal{Y}} p_y^2 $$

3. **Entropy:**
   
   $$ u(S) := \sum_{y \in \mathcal{Y}} p_y \log \frac{1}{p_y} $$

Each is maximized when $p_y = 1/K$ for all $y \in \mathcal{Y}$ (i.e., equal numbers of each label in $S$).

Each is minimized when $p_y = 1$ for a single label $y \in \mathcal{Y}$ (so $S$ is pure in label).
Suppose $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = \{\text{red, blue}\}$, and the data is as follows:

Every split of the form $\mathbb{1}\{x_i > t\}$ provides no reduction in uncertainty (whether based on classification error, Gini index, or entropy).
Suppose $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = \{\text{red}, \text{blue}\}$, and the data is as follows:

Every split of the form $1\{x_i > t\}$ provides no reduction in uncertainty (whether based on classification error, Gini index, or entropy).

Remark: if we do a random nonempty split, the next step can make progress.
When to stop?

Many alternatives; two common choices are:

1. Stop when the tree reaches a pre-specified size. Involved setting additional “tuning parameters” (similar to \( k \) in \( k \)-NN).
2. Stop when every leaf is pure. (More common.) Serious danger of overfitting: spurious structure due to sampling.
Many alternatives; two common choices are:

1. Stop when the tree reaches a pre-specified size.
When to stop?

Many alternatives; two common choices are:

1. Stop when the **tree reaches a pre-specified size**.
   
   Involves setting additional “tuning parameters” (similar to $k$ in $k$-NN).
Many alternatives; two common choices are:

1. Stop when the **tree reaches a pre-specified size**.

   Involves setting additional “tuning parameters” (similar to \( k \) in \( k \)-NN).

2. Stop when **every leaf is pure**. (More common.)
When to stop?

Many alternatives; two common choices are:

1. Stop when the tree reaches a pre-specified size.
   
   Involves setting additional “tuning parameters” (similar to $k$ in $k$-NN).

2. Stop when every leaf is pure. (More common.)
   
   Serious danger of overfitting spurious structure due to sampling.
Many alternatives; two common choices are:

1. **Stop when the tree reaches a pre-specified size.**
   
   Involves setting additional “tuning parameters” (similar to $k$ in $k$-NN).

2. **Stop when every leaf is pure.** (More common.)
   
   Serious danger of **overfitting** spurious structure due to sampling.
When to stop?

Many alternatives; two common choices are:

1. **Stop when the tree reaches a pre-specified size.**
   
   Involves setting additional “tuning parameters” (similar to $k$ in $k$-NN).

2. **Stop when every leaf is pure.** (More common.)
   
   Serious danger of overfitting spurious structure due to sampling.
When to stop?

Many alternatives; two common choices are:

1. Stop when the tree reaches a pre-specified size.
   Involves setting additional “tuning parameters” (similar to $k$ in $k$-NN).

2. Stop when every leaf is pure. (More common.)
   Serious danger of overfitting spurious structure due to sampling.
Overfitting

- Training error **goes to zero** as the number of nodes in the tree increases.
- True error decreases initially, but eventually **increases due to overfitting**. (Fix this by stopping early, or by pruning tree afterwards.)
Example: Spam filtering

Data

- 4601 e-mail messages, 39.4% are spam.
- $\mathcal{Y} = \{\text{spam, not spam}\}$
- E-mails represented by 57 features:
  - 48: percentage of e-mail words that is specific word (e.g., “free”, “business”)
  - 6: percentage of e-mail characters that is specific character (e.g., “!”).
  - 3: other features (e.g., average length of ALL-CAPS words).

Results

Using variant of greedy algorithm to grow tree; prune tree using validation set.

Chosen tree has just 17 leaves. Test error is 9.3%.

<table>
<thead>
<tr>
<th>$y = \text{not spam}$</th>
<th>$\hat{y} = \text{not spam}$</th>
<th>57.3%</th>
<th>4.0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = \text{spam}$</td>
<td></td>
<td>5.3%</td>
<td>33.4%</td>
</tr>
</tbody>
</table>
Note this is somewhat interpretable. Interpretability is a popular and active subject these days, partially since deep networks are used extensively but hard to interpret.
Nearest neighbors.
- **Training/fitting**: memorize data.
- **Testing/predicting**: find $k$ closest memorized points, return plurality label.
- **Overfitting?** Vary $k$.

Decision trees.
- **Training/fitting**: greedily partition space, reducing “uncertainty”.
- **Testing/predicting**: traverse tree, output leaf label.
- **Overfitting?** Limit or prune tree.

Note: both methods can output real numbers (regression, not classification); return median/mean of \{ neighbors, points reaching leaf \}. 