$k$-nn and decision trees
Plan for today

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 $x_i$, meaning $\rho(x, x_i) = \min_j \rho(x, x_j)$.

(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 (most frequent) label $y$ amongst $k$ closest training examples ("$k$" nearest neighbors).
   - (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?
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 (most frequent) label $y$ amongst $k$ closest training examples ("$k$" nearest neighbors).
   - (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?
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.
  ("Optimal" means "bayes error rate", the best population risk over all possible predictors.)
- Higher $k$ smooths the predictor, and gives a "less complex" model in an interesting way.
Example: OCR (“optical character recognition”) for digits

**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>( k )</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test error rate</td>
<td>0.0309</td>
<td>0.0295</td>
<td>0.0312</td>
<td>0.0306</td>
<td>0.0341</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>
Caution: nearest neighbor classifier can be broken by bad/noisy features!
Caution: nearest neighbor classifier can be broken by bad/noisy features!

Curse of dimension. Given 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.
**Caution**: nearest neighbor classifier can be broken by bad/noisy features!

**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.

**Popular approach**: train a deep network \( f : \mathbb{R}^d \rightarrow \mathbb{R}^p \), and run \( k\)-nn on its outputs!
Computation

Naive $k$-nn takes $O(n)$ time for each prediction (and needs $O(n)$ storage at all times).

There are many ways to speed this up; see for instance algorithm parameter of `sklearn.neighbors.KNeighborsClassifier`. There is also locality sensitive hashing (LSH). (These are all beyond the scope of this course.)
A decision tree is a binary tree which recursively partitions/refines the input space:

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

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

To make a prediction: given $x$, recurse down the tree until a leaf is reached, and output its label.

When $X = \mathbb{R}^d$, typically only consider splitting rules of the form $g(x) = 1$ for some $i \in [d]$ and $t \in \mathbb{R}$.

Called axis-aligned or coordinate splits.

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

$x_1 > 1.7$

$x_2 > 2.8$

$\hat{y} = 1$

$\hat{y} = 2$

$\hat{y} = 3$
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 \( g : \mathcal{X} \rightarrow \{0, 1\} \) (interpreted as “recurse left” and “recurse right”).
- Each leaf node is associated with a label \( \hat{y} \).

**To make a prediction:**

given \( x \), recurse down the tree until a leaf is reached, and output its label.
A **decision tree** is a binary tree which recursively partitions/Refines the input space:

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

**To make a prediction:**
given \( x \), recurse down the tree until a leaf is reached, and output its label.

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

\[
g(x) = \mathbbm{1}\{x_i > t\}
\]

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
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

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

$x_1 > 1.7$
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

$\hat{y} = 1$  $\hat{y} = 3$
Decision tree example

Classifying irises by sepal and petal measurements

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

$\hat{y} = 1$ if $x_1 > 1.7$
$\hat{y} = 2$ if $x_2 > 2.8$
Decision tree example

Classifying irises by sepal and petal measurements

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

**Basic decision tree algorithm (further details soon):**
recursively partition data, minimizing *uncertainty*.
Notions of uncertainty for binary classification

Basic decision tree algorithm (further details soon): recursively partition data, minimizing uncertainty.

Example uncertainty measures for binary classification, given example set $S$, suppose $p|S|$ 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 examples set $S$, of which $p_y |S|$ 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., all labels appear equally).

Each is minimized when $p_y = 1$ for a single label $y \in \mathcal{Y}$ (i.e., $S$ is pure).
Basic decision tree learning algorithm

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

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

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

- Place all data in single root tree node.
Basic decision tree learning algorithm

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

- Pick an per-node uncertainty measure $u$; the uncertainty of a tree $T$ is
  \[ u(T) := \frac{1}{n} \sum_{\text{leaf } S \in T} |S| \cdot u(S). \]

- Place all data in single root tree node.
- 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.
Basic decision tree learning algorithm

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

1. Pick an per-node uncertainty measure $u$; the uncertainty of a tree $T$ is
   \[ u(T) := \frac{1}{n} \sum_{\text{leaf } S \in T} |S| \cdot u(S). \]

2. Place all data in single root tree node.
3. 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 (as before): traverse tree to corresponding leaf, output the plurality (or average) label of its training data.
Basic decision tree learning algorithm

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

- Pick an per-node uncertainty measure \( u \); the uncertainty of a tree \( T \) is
  \[
  u(T) := \frac{1}{n} \sum_{\text{leaf } S \in T} |S| \cdot u(S).
  \]

- Place all data in single root tree node.

- 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 (as before): traverse tree to corresponding leaf, output the plurality (or average) label of its training data.
Failure of greedy uncertainty reduction

Suppose $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = \{\text{red, blue}\}$, 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).
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).

**Remark:** if we do a random nonempty split, the subsequent step can make progress.
Many alternatives; two common choices are:
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).
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.
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.
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.
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**.
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.)

The stopping rule is related to reducing model complexity (step 6 in the pytorch meta-algorithm).
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%.

\[
\begin{array}{c|cc}
  & \hat{y} = \text{not spam} & \hat{y} = \text{spam} \\
  y = \text{not spam} & 57.3\% & 4.0\% \\
  y = \text{spam} & 5.3\% & 33.4\%
\end{array}
\]
FIGURE 9.5. The pruned tree for the spam example. The split variables are shown in blue on the branches, and the classification is shown in every node. The numbers under the terminal nodes indicate misclassification rates on the test data.

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.
Summary for today

**Nearest neighbors.**
**Training/fitting:** memorize data.
**Testing/predicting:** find $k$ closest memorized points, return plurality/average 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 naturally handle binary classification, multi-class classification, and regression.