Recall

- Classify a point as the label of the “most similar” training point

- Euclidean distance: \( d(\vec{x}, \vec{x}') = ||\vec{x} - \vec{x}'|| = \sum_{j=1}^{D} (x_j - x'_j)^2 \)

- Given \( D = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_n, y_n)\} \) and a point \( \vec{x} \), let \( \vec{x}[i](\vec{x}) \) be the \( i^{th} \) closest point to \( \vec{x} \) in \( D \)
The Nearest Neighbor Hypothesis

\[ g(\vec{x}) = y_{[1]}(\vec{x}) \]
Claim: $E_{out}$ for the nearest neighbor hypothesis is not much worse than the best possible $E_{out}$!

Formally: under certain conditions, with high probability, $E_{out}(g) \leq 2E_{out}(g^*)$ as $n \to \infty$

Interpretation: half of the data’s predictive power is in the nearest neighbor!
$k$-Nearest Neighbors ($k$NN)

- Classify a point as the most common label among the labels of the $k$ nearest training points
- If we have a binary classification problem and $k$ is odd:
  \[
g(\vec{x}) = \text{sign} \left( \sum_{i=1}^{k} y_{[i]}(\vec{x}) \right)
\]
- $k$ controls the complexity of the hypothesis set $\implies k$ affects how well the learned hypothesis will generalize
  - $k = 3$
  - $k = \lfloor \sqrt{n} \rfloor$
  - Cross-validation
$k$NN Pros and Cons

Pros:
- Intuitive / explainable
- No training / retraining
- Self-regularizes
- Provably near-optimal in terms of $E_{out}$

Cons:
- Computationally expensive
  - Always needs to store all data: $O(nD)$
  - Computing $g(\tilde{x})$ requires computing $d(\tilde{x}, \tilde{x}') \forall \tilde{x}' \in \mathcal{D}$ and finding the $k$ closest points: $O(nD + n \log(k))$
- Suffers from the “curse of dimensionality”
The fundamental assumption of kNN is that “similar” points or points close to one another should have the same label.

The closer two points are, the more confident we can be that they will have the same label.

As the number of dimensions the input has grows, the less likely it is that two random points will be close.

As the number of dimensions the input has grows, it takes more points to “cover” the input space.
Suppose you independently draw two one-dimensional points between 0 and 1 uniformly at random:

\[ d(x, x') = \frac{1}{2} - \frac{2}{3}x + \frac{1}{3}x' \]

\[ \mathbb{E}[d(x, x')] = \mathbb{E}[(x - x')^2] = \mathbb{E}[x^2 - 2xx' + x'^2] \\
= \mathbb{E}[x^2] - 2\mathbb{E}[x]\mathbb{E}[x'] + \mathbb{E}[x'^2] \\
= 2\mathbb{E}[x^2] - 2\mathbb{E}[x]^2 = 2 \left( \frac{1}{3} \right) - 2 \left( \frac{1}{2} \right)^2 = \frac{1}{6} \]
• Suppose you independently draw two two-dimensional points in the unit square uniformly at random:

\[ d(x, x') = (x_1 - x'_1)^2 + (x_2 - x'_2)^2 \]

\[ \mathbb{E}[d(x, x')] = \mathbb{E}[(x_1 - x'_1)^2 + (x_2 - x'_2)^2] \]

\[ = 2 \mathbb{E}[(x_1 - x'_1)^2] \]

\[ = 2 \left( \frac{1}{6} \right) = \frac{1}{3} \]
Suppose you independently draw two three-dimensional points in the unit cube uniformly at random:

\[ \begin{align*}
\mathbb{E}[d(x, x')] &= \mathbb{E}[(x_1 - x'_1)^2 + (x_2 - x'_2)^2 + (x_3 - x'_3)^2] \\
&= 3 \mathbb{E}[(x_1 - x'_1)^2] \\
&= 3 \left( \frac{1}{6} \right) = \frac{1}{2}
\end{align*} \]
Curse of Dimensionality

- Assume all dimensions of the input are i.i.d.

$$\mathcal{P}(\vec{x}) = \prod_{j=1}^{D} p(x_j)$$

- Given $n$ inputs, $\{\vec{x}_1, ..., \vec{x}_n\}$, and a random query point, $\vec{x}^*$, all drawn i.i.d. from the distribution above, let

$$d_+ = \max_{\vec{x} \in \{\vec{x}_1, ..., \vec{x}_n\}} d(\vec{x}, \vec{x}^*) \quad \text{and} \quad d_- = \min_{\vec{x} \in \{\vec{x}_1, ..., \vec{x}_n\}} d(\vec{x}, \vec{x}^*)$$

- Then:

$$\lim_{D \to \infty} \mathbb{E} \left[ \frac{d_+ - d_-}{d_-} \right] \to 0$$
Curing the Curse of Dimensionality

- More data
- Fewer dimensions
- Blessing of non-uniformity: data from the real world is rarely uniformly distributed across the input space
Computational Cost of $k$NN

- No training required!
- Memory: $O(nD)$
- Computing $g(\tilde{x}): O(nD + n \log(k))$

- Idea: preprocess inputs in order to speed up predictions
  - Reduce the number of inputs held in memory by eliminating redundancies
  - Organize inputs in data structures that make searching for nearest neighbors more efficient
Data Condensing

- Reduce the number of inputs while maintaining the same decision boundary
- Let $g_D$ be the $k$NN hypothesis when trained on $\mathcal{D}$
- $S \subseteq \mathcal{D}$ is decision-boundary consistent if:
  \[ g_S(\vec{x}) = g_D(\vec{x}) \forall \vec{x} \in \mathcal{X} \]
- Decision-boundary consistent subsets are computationally expensive to find
Data Condensing

• Reduce the number of inputs while maintaining the same predictions on all inputs

• Let $g_D$ be the $k$NN hypothesis when trained on $D$

• $S \subseteq D$ is training-set consistent if:

$$g_S(x_i) = g_D(x_i) \forall x_i \in D$$

• Training-set consistent is a much weaker constraint than decision-boundary consistent
• Input: \( \mathcal{D} = \{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \}, k \)

• Compute \( g_D(x_i) \forall x_i \in \mathcal{D} \)

• Initialize \( S \) to \( k \) random points in \( \mathcal{D} \) and compute

\( g_S(x_i) \forall x_i \in \mathcal{D} \)

• While \( \exists x_j \in \mathcal{D} \) s.t. \( g_S(x_j) \neq g_D(x_j) \)

  • Randomly pick a point \( x_j \in \mathcal{D} \) s.t. \( g_S(x_j) \neq g_D(x_j) \)

  • Let \( \hat{x}^* \) be the point closest to \( x_j \) that is not already in \( S \) and has label \( y^* = g_D(x_j) \)

  • Add \( \hat{x}^* \) to \( S \) and recompute \( g_S(x_i) \forall x_i \in \mathcal{D} \)

• Output: \( S \), a training-set consistent subset of \( \mathcal{D} \)
CNN Example

$k = 3$

\[ g_S(\tilde{x}) = -1, \quad g_D(\tilde{x}) = +1 \]
CNN Example

\[
k = 3
\]

\[
g_S(\vec{x}) = -1, \ g_D(\vec{x}) = +1
\]
• Intuition: split the inputs into clusters, groups of points that are close to one another but far from other groups.

• If an input point is really close to one group of points and really far from the other groups ... 

• ... then we can skip searching through the other groups and just look for nearest neighbors in the close group!

• Questions:
  • What does it mean for a point to be close to a group?
  • How can we split the input into clusters?
Organizing the Inputs

Let $\mu_i$ ($\bar{x}_i$) be the closest point to $\hat{x}$ in the cluster $S_j$.
Organizing the Inputs

Let \( \tilde{x}_j^i(x) \) be the \( i^{th} \) closest point to \( x \) in the cluster \( S_j \).

- \( \| \tilde{x} - \tilde{\mu}_j \| - r_j \leq \| \tilde{x} - \tilde{x}_j^i(x) \| \)
- If \( \exists S_i \) s.t. \( \| \tilde{x} - \tilde{x}_j^i(x) \| \leq \| \tilde{x} - \tilde{\mu}_j \| - r_j \) ...
- ... then we don’t need to search \( S_j \) for the nearest neighbor
Organizing the Inputs

Let $\vec{x}_{[i]}^{j}(\vec{x})$ be the $i^{th}$ closest point to $\vec{x}$ in the cluster $S_j$

- $\|\vec{x} - \vec{\mu}_j\| - r_j \leq \|\vec{x} - \vec{x}_{[1]}^{j}(\vec{x})\| \leq \cdots \leq \|\vec{x} - \vec{x}_{[k]}^{j}(\vec{x})\|$.

- If $\exists S_i$ s.t. $\|\vec{x} - \vec{x}_{[k]}^{i}(\vec{x})\| \leq \|\vec{x} - \vec{\mu}_j\| - r_j$ ...

- ... then we don’t need to search $S_j$ for the $k$ nearest neighbors.
Clustering Inputs

- We want \( \| \hat{x} - \hat{x}_{[1]}(\hat{x}) \| \leq \| \hat{x} - \mu_j \| - r_j \)
- \( \| \hat{x} - \hat{x}_{[1]}(\hat{x}) \| \leq \| \hat{x} - \mu_i \| + \| \mu_i - \hat{x}_{[1]}(\hat{x}) \| \leq \| \hat{x} - \mu_i \| + r_i \)
- We want \( \| \hat{x} - \mu_i \| + r_i \leq \| \hat{x} - \mu_j \| - r_j \)
- Suppose \( \hat{x} \approx \mu_i \)
  - \( \| \hat{x} - \mu_i \| \approx 0 \) and \( \| \hat{x} - \mu_j \| \approx \| \mu_i - \mu_j \| \)
- We want \( r_i \leq \| \mu_i - \mu_j \| - r_j \Rightarrow r_i + r_j \leq \| \mu_i - \mu_j \| \)
- We want cluster centers to be far apart and cluster radii to be small
• Input: $D = \{\vec{x}_1, \vec{x}_2, ..., \vec{x}_n\}, C$

• Initialize $M$ to a random point $\vec{x} \in D$ and set $S_1 = \vec{x}$

• For $c = 2 \ldots C$
  • Find the point $\vec{x}^* \in D \setminus M$ that is farthest from $M$, add $\vec{x}^*$ to $M$ and set $S_c = \vec{x}^*$:
    $$\vec{x}^* = \arg\max_{\vec{x}^* \in D \setminus M} \left( \min_{\vec{x} \in M} ||\vec{x}^* - \vec{x}|| \right)$$

• For $i = 1 \ldots n$
  • Find the cluster center $\vec{x}_c \in M$ closest to $\vec{x}_i$ and assign $\vec{x}_i$ to $S_c$:
    $$\vec{x}_c = \arg\min_{\vec{x} \in M} (||\vec{x}_i - \vec{x}||)$$

• For $c = 1 \ldots C$
  • Update $S_c$’s cluster center and compute the radius:
    $$\vec{\mu}_c = \frac{1}{|S_c|} \sum_{\vec{x} \in S_c} \vec{x} \quad \text{and} \quad r_c = \max_{\vec{x} \in S_c} ||\vec{\mu}_c - \vec{x}||$$

• Output: cluster centers $\mu = \{\vec{\mu}_1, \vec{\mu}_2, ..., \vec{\mu}_C\}$, radii $r = \{r_1, r_2, ..., r_C\}$ and clusters $S = \{S_1, S_2, ..., S_C\}$
Making Predictions: Example

- Input: \( D = \{ (\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n) \}, C, k, \mathbf{x} \)
- Cluster the input into \( C \) clusters
- Find \( S_c \), the cluster whose center is closest to \( \mathbf{x} \)
- Find \( \mathbf{x}^c[k](\mathbf{x}) \), the \( k^{th} \) closest point to \( \mathbf{x} \) in \( S_c \)
- Find \( \{ \mathbf{x}_{[1]}(\mathbf{x}), \ldots, \mathbf{x}_{[k]}(\mathbf{x}) \} \) the \( k \) closest points to \( \mathbf{x} \), ignoring all clusters \( S_c' \) s.t. \[ \| \mathbf{x} - \mathbf{x}^c[k](\mathbf{x}) \| \leq \| \mathbf{x} - \mu_{c'} \| - r_{c'} \]
- Output: \( \{ \mathbf{x}_{[1]}(\mathbf{x}), \ldots, \mathbf{x}_{[k]}(\mathbf{x}) \} \)
Making Predictions: Example

• Input: \( D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}, C, k, \bar{x} \)

• Cluster the input into \( C \) clusters \( O(CnD) \)

• Find \( S_c \), the cluster whose center is closest to \( \bar{x} \)

• Find \( \bar{x}_c^c(\bar{x}) \), the \( k \)th closest point to \( \bar{x} \) in \( S_c \)

• Find \( \{\bar{x}_{[1]}(\bar{x}), \ldots, \bar{x}_{[k]}(\bar{x})\} \) the \( k \) closest points to \( \bar{x} \), ignoring all clusters \( S_{c'} \) s.t. \( ||\bar{x} - \bar{x}_{[k]}^c(\bar{x})|| \leq ||\bar{x} - \bar{\mu}_{c'}|| - r_{c'} \) (hopefully less than \( O(nD + n \log(k)) \))

• Output: \( \{\bar{x}_{[1]}(\bar{x}), \ldots, \bar{x}_{[k]}(\bar{x})\} \)
Parametric vs. Non-parametric

- Parametric learning models
  - Hypotheses have a parametrized form
    - Example: linear regression \( g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \) has \( D + 1 \) parameters, \( \{w_0, w_1, \ldots, w_D\} \)
  - Parameters learned from training data; can discard the training data afterwards
  - Cannot exactly model every target function

- Non-parametric learning models
  - Hypotheses cannot be expressed using a finite number of parameters
    - Example: kNN, decision trees
  - Training data generally needs to be stored in order to make predictions
  - Can recover any target function given enough data