Nearest Neighbor Classification

Seed classification by area and compactness

▶ What should we predict for unlabeled test points (stars)?
▶ Nearest neighbor classification: predict label of nearest training example
▶ \( k \)-nearest neighbor: predict consensus of \( k \) nearest training examples

\( k \)-Nearest Neighbor Classification

▶ Training: store the training data (trivial!)
\[ D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\} \]

▶ Prediction: for a new instance \( x \), predict label that is most frequent among \( k \) training examples closest to \( x \)

▶ KNN can work with any distance function and any value of \( k \). We need to choose these.

Distance and Similarity

▶ KNN can use any distance function to determine \( k \) nearest neighbors. A distance function \( d(x, x') \) takes two data points and returns a distance. It should satisfy
  - \( d(x, x') \geq 0 \) (non-negativity)
  - \( d(x, x') = 0 \) (distance from a point to itself is zero)

▶ Or you can use a similarity function
  - \( s(x, x') \geq 0 \)
  - \( s(x, x) \geq s(x, x') \) for all other \( x' \) (\( x \) is more similar to itself than any other point)

Euclidean Distance

▶ We’ve already seen one distance function, the Euclidean distance:
\[ d(x, x') = \|x - x'\| \]
  - Length of straight line between \( x \) and \( x' \) (= vector norm of \( x - x' \))

Minkowski Distance

▶ A more general class of distance functions come from Minkowski Distance
\[ d_p(x, x') := \|x - x'\|_p \]
\[ \|r\|_p := \left( \sum_{i=1}^{n} |r_i|^p \right)^{1/p} \]
  - \( p = 2 \) is Euclidean distance (verify on own)
  - \( p = 1 \) is called the “Manhattan distance”

Examples

▶ Jupyter Demo 1: different distance functions

KNN Implementation

▶ The “brute force” version of KNN is very straightforward:
  - Given test point \( x \), compute distances \( d^{(i)} := d(x, x^{(i)}) \) to each training example
  - Sort training examples by distance
  - \( k \)-nearest neighbors — find \( k \) examples in this sorted list.
  - Now, making the prediction is straightforward.
  - Running time: \( O(m \log m) \) for one prediction

▶ In practice, clever data structures (e.g., KD-trees) can be constructed to find \( k \) nearest neighbors and make predictions more quickly.
**KNN Trade-Offs**

- **Strengths**
  - Simple
  - Converges to the correct decision surface as data goes to infinity

- **Weaknesses**
  - Lots of variability in the decision surface when amount of data is low
  - Curse of dimensionality: everything is far from everything else in high dimensions
  - Running time and memory usage: store all training data and perform neighbor search for every prediction → use a lot of memory / time

- Jupyter Demo 2: KNN in action
  - Effect of \( k \)
  - KNN convergence as data goes to infinity

---

**Decision Trees**

- Classical model for making a decision or classification using “splitting rules” organized into tree data structure
- Data instance \( x \) is routed from the root to leaf
  - Nodes = “splitting rules”
    - Continuous variables: test if \((x_j < c)\) or \((x_j \geq c)\) (2 branches)
    - Discrete variables: test \((x_j = 1), (x_j = 2), \ldots\) for \( k \) possible values of \( x_j \) (\( k \) branches)
  - \( x \) goes down branch corresponding to result of test
  - Leaf nodes are assigned labels → prediction for \( x \)

---

**Decision Tree Intuition**

- Board work
  - Geometric illustration of decision tree: recursive axis-aligned partitioning
  - Intuition for how to partition to fit a dataset (= learning a decision tree)

---

**Decision Tree Learning**

- How do we fit a decision tree to training data? We won’t give details here, just some intuition...
  - Idea: recursive splitting of training set

- Start with all training examples at root of tree
- Find “best” splitting rule at root
- Recurse on each branch

---

**Decision Tree Trade-Offs**

- **Strengths**
  - Interpretability: the learned model is easy to understand
  - Running time for predictions: shallow trees can be extremely fast classifiers

- **Weaknesses**
  - Running time for learning: finding the optimal trees is computationally intractable (NP-complete), so we need to design greedy heuristics.
  - Representation: we may need very large trees to accurately model geometry of our problem with axis-aligned splits

- General advice: decision trees are very competitive “out-of-the-box” machine learning models for lots of problems!