Northeastern University

K-Nearest Neighbors

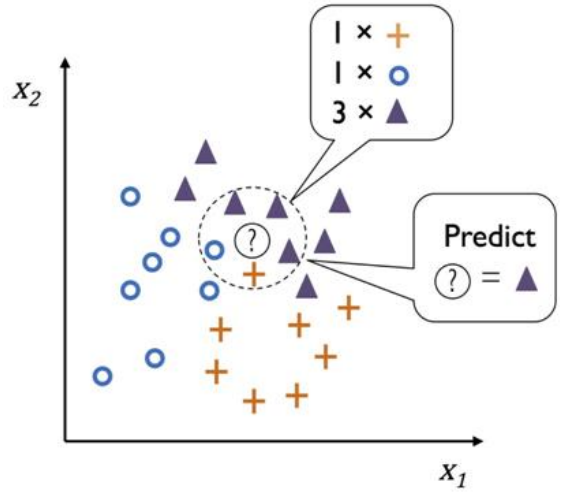
Roi Yehoshua

Agenda

- *k*-nearest neighbors
- Distance metrics
- KNN regression

k-Nearest Neighbors (KNN)

- KNN is a very simple, versatile and one of the topmost ML algorithms
- The label assigned to a sample is based on a majority vote of its *k* nearest neighbors
- KNN is a type of **instance-based learner** or **lazy learner**
  - Doesn't build an internal model from the data and defers all computation until prediction time
- KNN is also a **nonparametric method**
  - Doesn't assume anything about the form of the mapping from the features to the label



KNN Classification Algorithm

Algorithm The k-nearest neighbors classification algorithm

Input: A set of training examples D, number of nearest neighbors k, and a test example $\mathbf{x}'$

for each training example $(\mathbf{x}, y) \in D$ **do**
  Compute $d(\mathbf{x}', \mathbf{x})$, the distance between $\mathbf{x}'$ and $\mathbf{x}$
Select $D_z \subseteq D$, the set of k closest training examples to $\mathbf{x}'$
$y' = \text{argmax}_v \sum_{(\mathbf{x}_i, y_i) \in D_z} \mathbb{1}\{y_i = v\}$
return y'

- 1{·} is an indicator function that returns 1 if its argument is true, and 0 otherwise
  - e.g., 1{2 = 3} = 0 and 1{3 = 5 - 2} = 1

Example

- Given the following data set:

i	x_1	x_2	x_3	y
1	1	4	1	1
2	1	0	-2	0
3	0	0	1	0
4	-1	4	0	1
5	-1	-1	1	1
6	1	2	3	1
7	0	-4	0	0
8	1	0	-3	0

- Classify the vector (1, 0, 1) using KNN with $k = 3$ and using Euclidean distance

Example

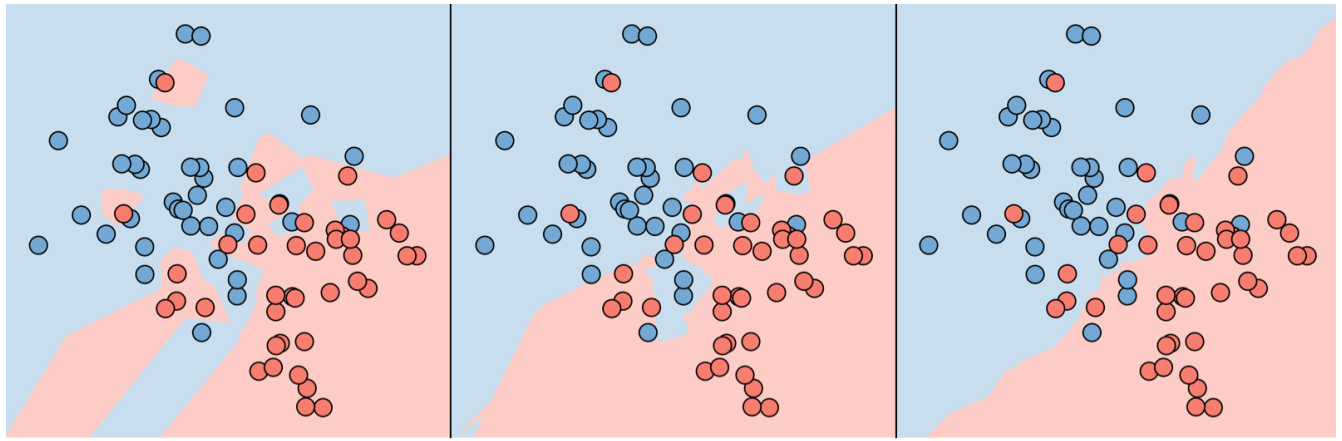
- The distances of (1,0,1) from the training samples are:

i	x_1	x_2	x_3	y	distance from (1, 0, 1)
1	1	4	1	1	$(1-1)^2 + (4-0)^2 + (1-1)^2 = 16$
2	1	0	-2	0	$(1-1)^2 + (0-0)^2 + (-2-1)^2 = 9$
3	0	0	1	0	$(0-1)^2 + (0-0)^2 + (1-1)^2 = 1$
4	-1	4	0	1	$(-1-1)^2 + (4-0)^2 + (0-1)^2 = 19$
5	-1	-1	1	1	$(-1-1)^2 + (-1-0)^2 + (1-1)^2 = 5$
6	1	2	3	1	$(1-1)^2 + (2-0)^2 + (3-1)^2 = 8$
7	0	-4	0	0	$(0-1)^2 + (-4-0)^2 + (0-1)^2 = 18$
8	1	0	-3	0	$(1-1)^2 + (0-0)^2 + (-3-1)^2 = 16$

- The majority label of the 3 closest neighbors is 1, therefore the predicted label is 1

Choosing the Number of Neighbors *k*

- The choice of *k* has a large impact on the performance of KNN
- If *k* is too small, the result can be sensitive to noise (mislabeled training examples)
- If *k* is too large, the neighborhood may include too many points from other classes
- Rule of thumb: choose $k = \sqrt{n}$, where *n* is the number of points in the training set
  - Usually works when the number of samples is high enough



Distance Measures

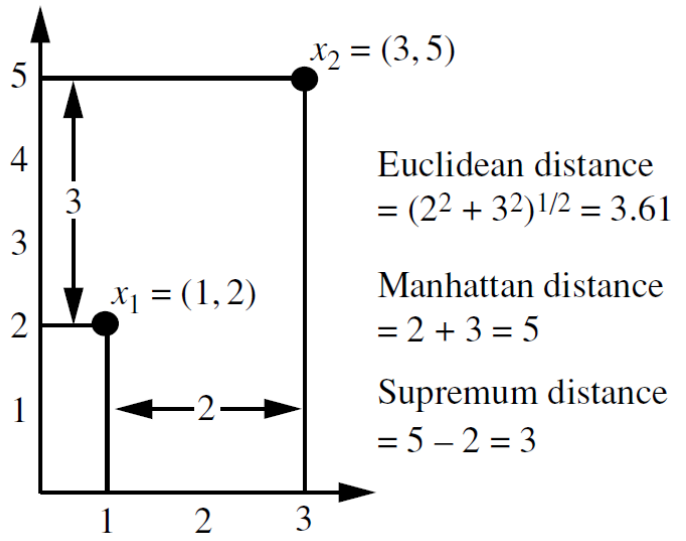
- The choice of the distance measure is another important consideration
- Depends on the type of the features (numerical, categorical, text, …)
- Features have to be scaled to prevent distance measures from being dominated by one of the features

Minkowski Distance

- Metric used for real-valued vectors
- Minkowski distance is a generalization of Euclidean distance

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{j=1}^{d} |x_j - y_j|^p \right)^{1/p}$$

- Common examples:
  - p = 1: Manhattan (city block, taxicab) distance
  - p = 2: Euclidean distance
  - $p \to \infty$: Supremum distance
    - The maximum difference between components of the vectors



Jaccard Similarity

- Metric used for **asymmetric binary vectors**
  - where the agreement of two 1s is considered more significant than of two 0s
  - e.g., when comparing users in a recommendation system where each user is represented by a binary vector of the items that they purchased
- Jaccard similarity is defined as

$$J(\mathbf{x}, \mathbf{y}) = \frac{\text{number of 11 matches}}{\text{number of non-both-zero attributes}} = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}$$

  - M_{11} = the number of attributes where **x** is 1 and **y** is 1
  - M_{01} = the number of attributes where **x** is 0 and **y** is 1
  - M_{10} = the number of attributes where **x** is 1 and **y** is 0
- Jaccard distance is defined as

$$d_J(\mathbf{x}, \mathbf{y}) = 1 - J(\mathbf{x}, \mathbf{y})$$

Jaccard Similarity: Example

- Given the vectors:

$$\mathbf{x} = (1, 0, 0, 1, 0, 0, 0, 1, 0, 0)^T$$
$$\mathbf{y} = (0, 0, 0, 1, 0, 0, 1, 1, 0, 1)^T$$

- The Jaccard similarity between them is:

$$J(\mathbf{x}, \mathbf{y}) = \frac{M_{11}}{M_{01} + M_{10} + M_{11}} = \frac{2}{2 + 1 + 2} = \frac{2}{5} = 0.4$$

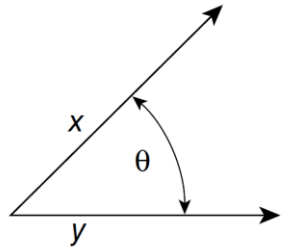
- And the Jaccard distance is:

$$d_J(\mathbf{x}, \mathbf{y}) = 1 - 0.4 = 0.6$$

Cosine Similarity

- Documents are often represented as word count vectors, where each feature represents the frequency of a word in the document
- This is a very sparse representation, since it has very few non-zero attributes
- We need a similarity measure that ignores 0-0 matches (like Jaccard similarity) but handles non-binary vectors
- Cosine similarity is the cosine of the angle between the two document vectors:

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||||\mathbf{y}||}$$



  - The smaller the angle, the higher the similarity
- Cosine similarity doesn't depend on the magnitude of the two data objects
  - which is desirable for comparing documents of different lengths

Cosine Similarity: Example

- Given the vectors:

$$\mathbf{x} = (3, 2, 0, 5, 0, 0, 0, 2, 0, 0)$$
$$\mathbf{y} = (1, 0, 0, 0, 0, 0, 0, 1, 0, 2)$$

- The cosine similarity between them is:

$$\mathbf{x} \cdot \mathbf{y} = 3 \cdot 1 + 2 \cdot 0 + 0 \cdot 0 + 5 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 2 \cdot 1 + 0 \cdot 0 + 0 \cdot 2 = 5$$
$$||\mathbf{x}|| = \sqrt{3 \cdot 3 + 2 \cdot 2 + 0 \cdot 0 + 5 \cdot 5 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 2 \cdot 2 + 0 \cdot 0 + 0 \cdot 0} = \sqrt{42} = 6.481$$
$$||\mathbf{y}|| = \sqrt{1 \cdot 1 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 + 2 \cdot 2} = \sqrt{6} = 2.449$$

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||||\mathbf{y}||} = \frac{5}{6.481 \cdot 2.449} = 0.315$$

Distance-Weighted Voting

- Often the closer neighbors more reliably indicate the class of the instance
- Thus, we can weight each neighbor's vote, so that nearer neighbors contribute more to the average than the more distant ones
- In this case, the class label is determined as follows:

$$y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} w_i \cdot \mathbb{1}\{y_i = v\}$$

- A common weighting scheme is to give each neighbor a weight of 1/*d*, where *d* is its distance from the test point $w_i = \frac{1}{d(\mathbf{x}', \mathbf{x}_i)}$
- This method is less sensitive to the choice of *k*

Example

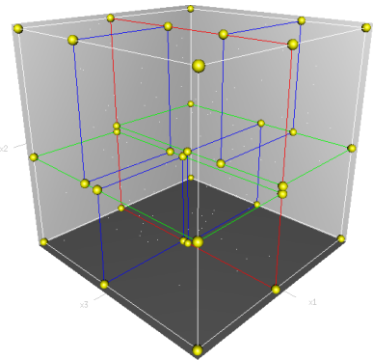
- The distances of (1,0,1) from the training samples are:

i	x_1	x_2	x_3	y	distance from (1, 0, 1)	weight
1	1	4	1	1	$(1-1)^2 + (4-0)^2 + (1-1)^2 = 16$	
2	1	0	-2	0	$(1-1)^2 + (0-0)^2 + (-2-1)^2 = 9$	
3	0	0	1	0	$(0-1)^2 + (0-0)^2 + (1-1)^2 = 1$	1/1 = 1
4	-1	4	0	1	$(-1-1)^2 + (4-0)^2 + (0-1)^2 = 19$	
5	-1	-1	1	1	$(-1-1)^2 + (-1-0)^2 + (1-1)^2 = 5$	1/5 = 0.2
6	1	2	3	1	$(1-1)^2 + (2-0)^2 + (3-1)^2 = 8$	1/8 = 0.125
7	0	-4	0	0	$(0-1)^2 + (-4-0)^2 + (0-1)^2 = 18$	
8	1	0	-3	0	$(1-1)^2 + (0-0)^2 + (-3-1)^2 = 16$	

- The weight of label 0 is higher than the total weight of 1 ($1 > 0.2 + 0.125$)
- Therefore, the predicted label is 0

KD-Tree

- Allows a more efficient retrieval of the closest neighbors
- A KD-tree is a binary tree structure that recursively splits the space into half-spaces
- Every node in the tree represents a *d*-dimensional point in space
- Every non-leaf node is associated with one of the *d* dimensions
  - Points that have a smaller value in that dimension will be in the left subtree of that node
  - Points with a larger value will be in its right subtree
- Points that are close to each other are usually stored at nearby nodes
- For small *d* ($d \leq 20$), the query time of KD-tree is approximately $O(d \log n)$
- For larger *d*, the cost increases nearly to $O(dn)$



The Curse of Dimensionality

- KNN may not work as well when there is a large number of features (dimensions)
- When dimensionality increases, data becomes increasingly sparse in space
  - Even the closest neighbors can be too far away to give a good estimate
- Solution: use dimensionality reduction

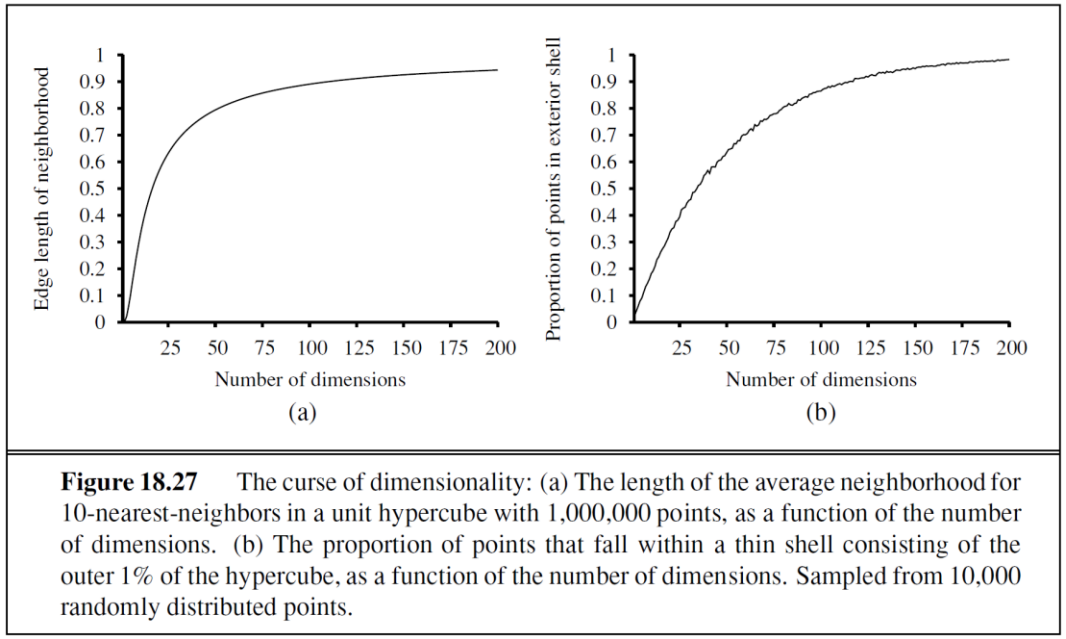


Figure 18.27 The curse of dimensionality: (a) The length of the average neighborhood for 10-nearest-neighbors in a unit hypercube with 1,000,000 points, as a function of the number of dimensions. (b) The proportion of points that fall within a thin shell consisting of the outer 1% of the hypercube, as a function of the number of dimensions. Sampled from 10,000 randomly distributed points.

KNN in Scikit-Learn

class `sklearn.neighbors.` **`KNeighborsClassifier`**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs*) [source]

Argument	Description
n_neighbors	Number of neighbors to use (default = 5)
weights	weight function used in prediction. Possible values: 'uniform': uniform weights. All points in each neighborhood are weighted equally (default). 'distance': weight points by the inverse of their distance. [callable]: a user-defined function
algorithm	Algorithm used to compute the nearest neighbors: 'ball_tree', 'kd_tree', 'brute', or 'auto'. 'auto' will choose the most appropriate algorithm based on the data set.
leaf_size	Leaf size passed to BallTree or KDTree (default = 30)
p	Power parameter for the Minkowski metric.
metric	The distance metric to use. A list of available metrics can be found here.

KNN in Scikit-Learn

- Let's train KNN classifiers with $k = 5$ and $k = 20$ on the Iris data set using only the first two features (sepal length and sepal width)

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data[:, :2]  # We only take the first two features
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier(n_neighbors=5)
clf.fit(X_train, y_train)

clf2 = KNeighborsClassifier(n_neighbors=20)
clf2.fit(X_train, y_train)
```

```
KNeighborsClassifier(n_neighbors=20)
```

KNN in Scikit-Learn

- The accuracy of the classifiers on the training and test sets:

```
print(f'Training set accuracy: {clf.score(X_train, y_train):.4f}')
print(f'Test set accuracy: {clf.score(X_test, y_test):.4f}')
```

```
Training set accuracy: 0.8393
Test set accuracy: 0.7632
```

```
print(f'Training set accuracy: {clf2.score(X_train, y_train):.4f}')
print(f'Test set accuracy: {clf2.score(X_test, y_test):.4f}')
```

```
Training set accuracy: 0.8214
Test set accuracy: 0.8158
```

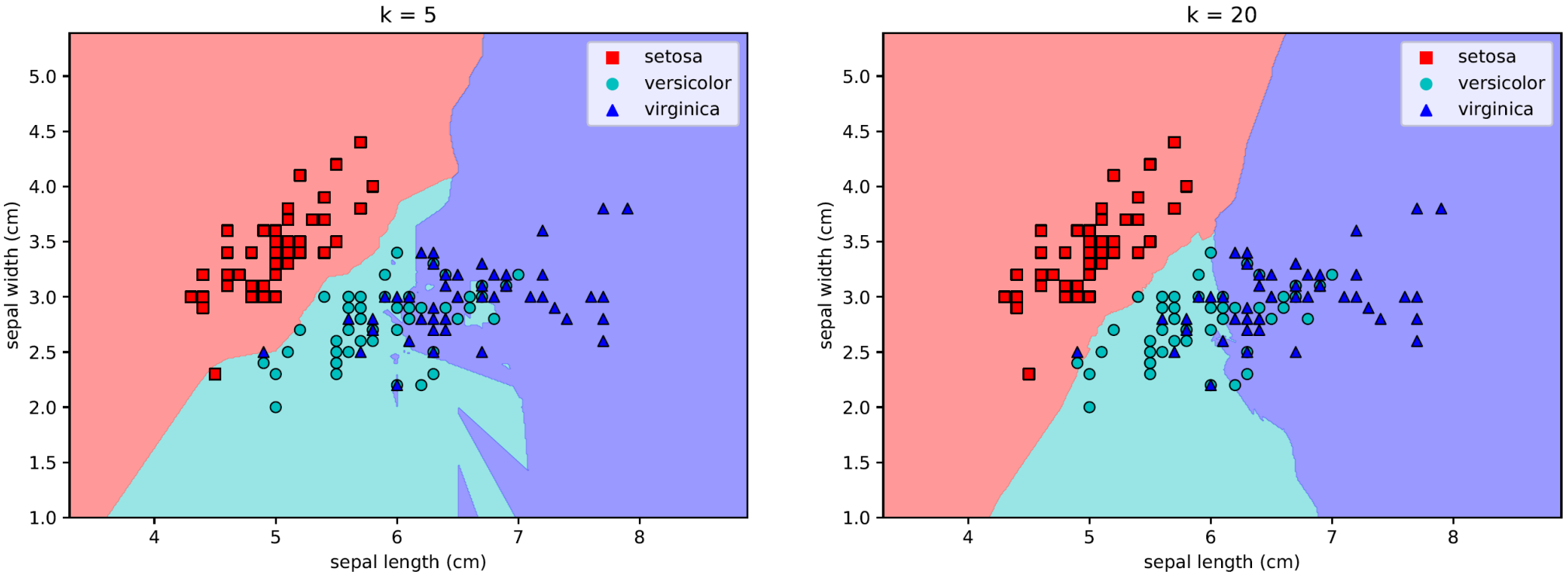
KNN in Scikit-Learn

- Let’s plot the decision boundaries of the classifiers:

```
from matplotlib.colors import ListedColormap

def plot_decision_boundaries(clf, X, y, feature_names, class_names, h=0.01):
    colors = ['r', 'c', 'b']
    cmap = ListedColormap(colors)

    # Assign a color to each point in the mesh [x_min, x_max]x[y_min, y_max]
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.4, cmap=cmap)

    # Plot also the sample points
    sns.scatterplot(X[:, 0], X[:, 1], hue=class_names[y], style=class_names[y],
                    palette=colors, markers=('s', 'o', '^'), edgecolor='black')
    plt.xlabel(feature_names[0])
    plt.ylabel(feature_names[1])
    plt.legend()
```

KNN in Scikit-Learn

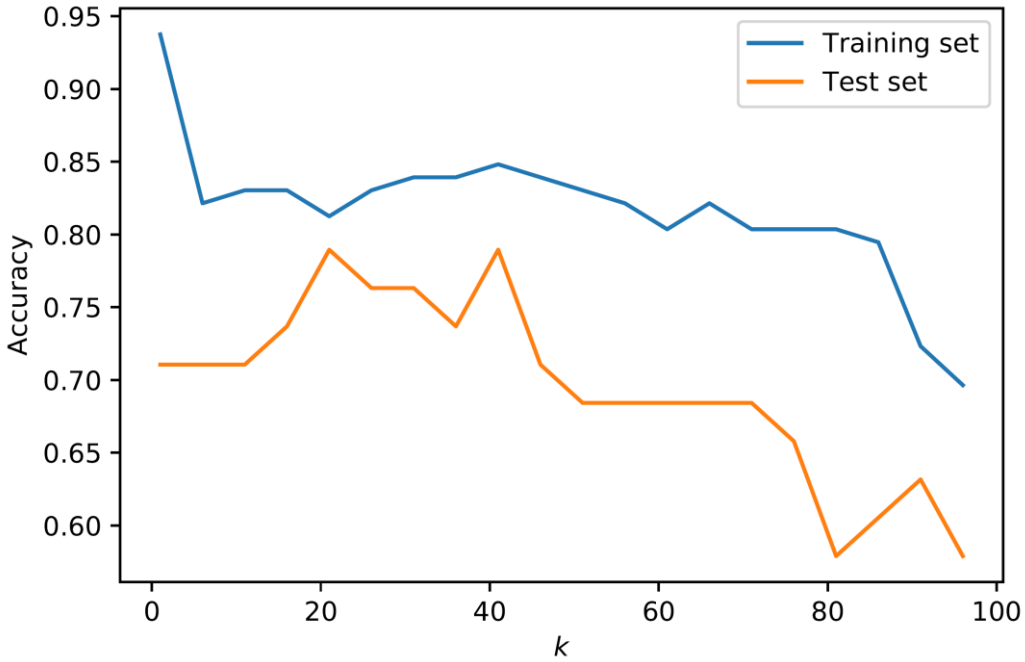
- Let’s plot the decision boundaries of the classifiers:



KNN in Scikit-Learn

- Let's examine the effect of changing the number of neighbors on the accuracy:

```
n_neighbors, train_scores, test_scores = [], [], []

for n in range(1, 100, 5):
    n_neighbors.append(n)
    clf = KNeighborsClassifier(n_neighbors=n)
    clf.fit(X_train, y_train)
    train_scores.append(clf.score(X_train, y_train))
    test_scores.append(clf.score(X_test, y_test))

plt.plot(n_neighbors, train_scores, label='Training set')
plt.plot(n_neighbors, test_scores, label='Test set')
plt.xlabel('$k$')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('figures/knn_results.pdf')
```



Nearest Neighbors Regression

- The KNN algorithm can also be used for regression
- The label assigned to a new data point is computed based on the mean of the labels of its nearest neighbors
- This algorithm is implemented in the class KNeighborsRegressor

class `sklearn.neighbors.` **`KNeighborsRegressor`**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs*) [source]

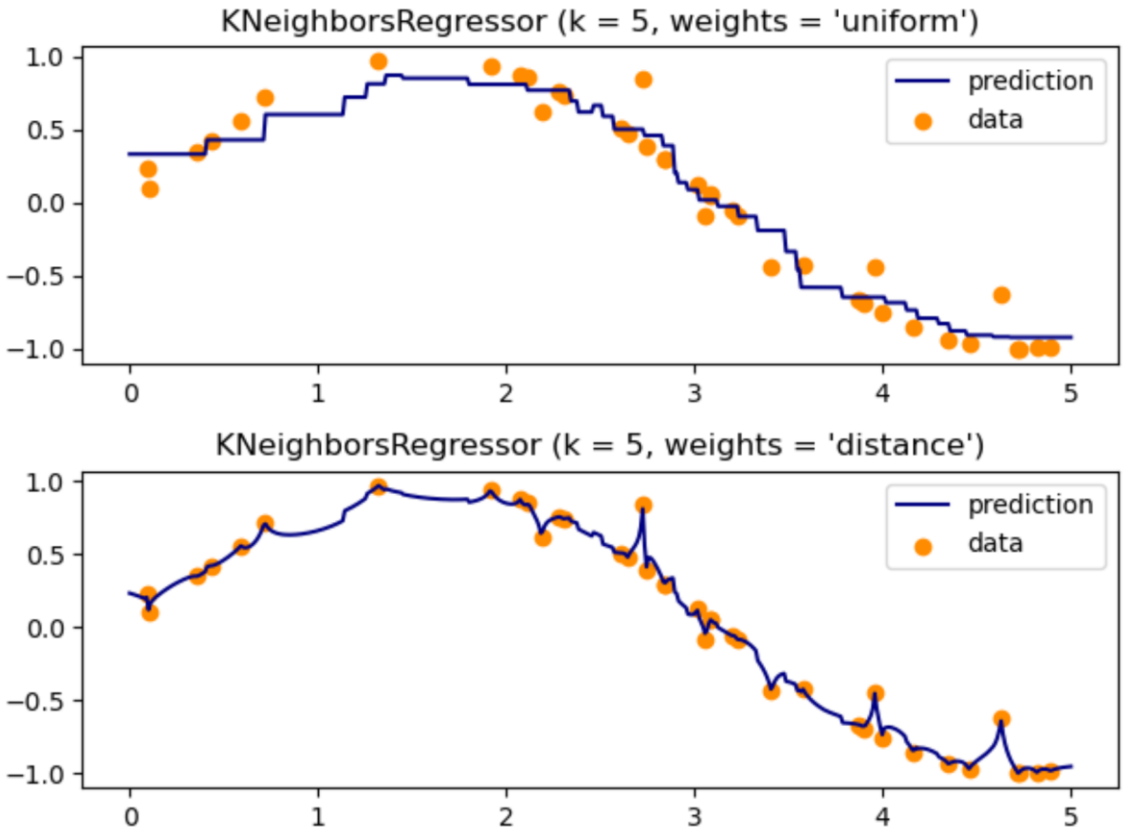
```
X = np.arange(5).reshape(-1, 1)
y = [2, 7, 4, 6, 8]
```

```
from sklearn.neighbors import KNeighborsRegressor

reg = KNeighborsRegressor(n_neighbors=2)
reg.fit(X, y)
reg.predict([[1.5]])
```

```
array([5.5])
```

Nearest Neighbors Regression

- Regression of the function $y = \sin(x)$ (+noise) using 40 random points in [0, 5]:



KNN Summary

Pros

- Doesn’t require to build a model
  - No information loss
  - Training is very fast
- Can produce decision boundaries of arbitrary shape
- Easy to implement
- Performs well in many problems
- Immediately adapts to new data
- Can be used for both classification and regression
- Small number of hyperparameters

Cons

- Slow at prediction time
- Need to store all the training examples
- Susceptible to noise/outliers (for small *k*)
- Sensitive to *k* and the distance measure
- Curse of dimensionality
- Doesn’t perform well on imbalanced data sets
- Limited interpretability
- Has difficulty in handling missing values
- Requires scaling of the data