# Lecture 12 - Feb 19

## Nearest Neighbor Methods

## Reading

Week 7 Notes in GitHub

*Elements of Statistical Learning*

- 2.3.2 Nearest-Neighbor Methods
- Ch 13: Prototype Methods and Nearest-Neighbors
  - 13.3: *k*-Nearest-Neighbor Classifier
  - 13.4: Adaptive Nearest-Neighbor Methods

*Data Mining and Machine Learning*

- 18.3: K Nearest Neighbor Classifier

## Upcoming Deadlines

Project 1 Proposal (coming soon)

Least squares linear model makes huge assumptions about structure with stable but possibly inaccurate predictions

k-Nearest neighbors makes mild structural assumptions with unstable but possibly accurate predictions

These are two extremes in a sense.

## 2.3.2 Nearest-Neighbor Methods

NN methods use points in the training set $T$ nearest to $x$ to form $\hat{Y}$... k-NN uses

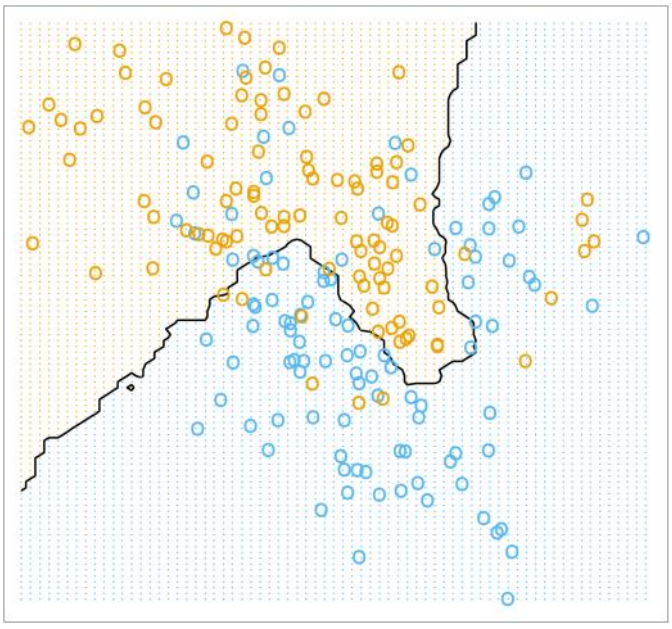
$$\hat{Y}(x) = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i$$

← average the outputs of the k nearest points in T

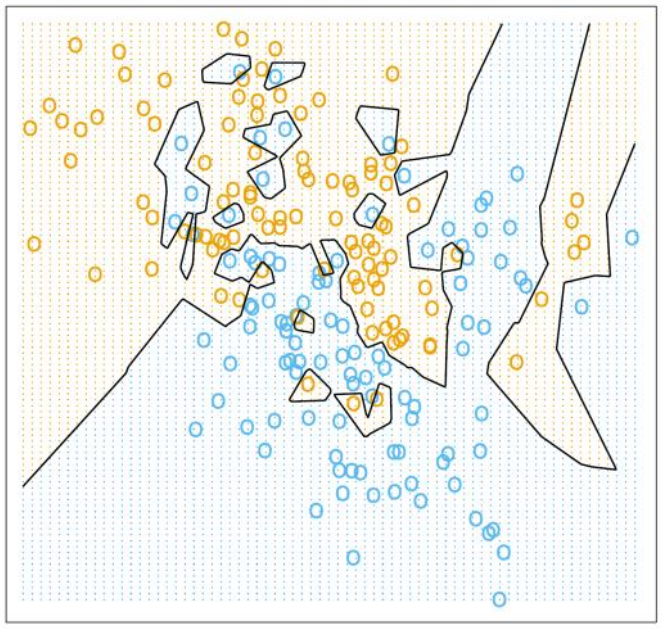
where $N_k(x)$ is the neighborhood of $x$ defined by the $k$ closest points $x_i$ in $T$.

↑ this implies a metric, use Euclidean distance here

15-Nearest Neighbor Classifier



1-Nearest Neighbor Classifier



high variance

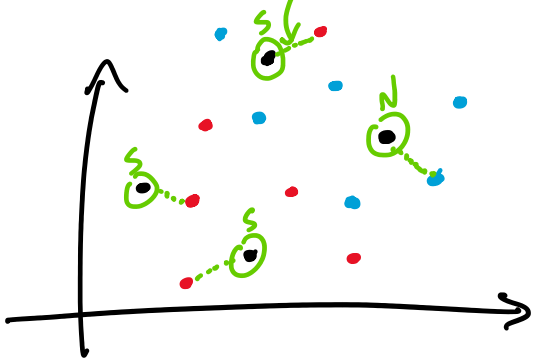
low bias

much more flexible decision boundaries, but very unstable to data perturbation.

# K-Nearest Neighbor classification

$\sqrt{(x_{11}-x_{21})^2+(x_{21}-x_{22})^2}$

k = 1

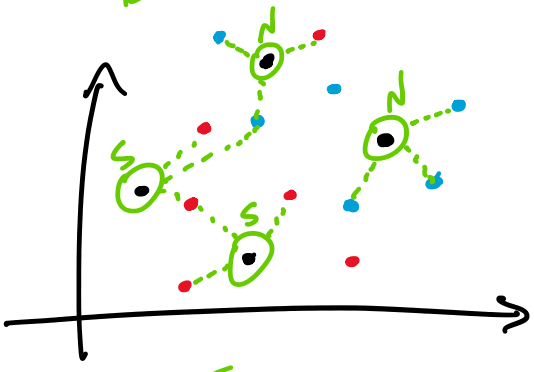


- second peak
- No second peak

} training set

- test set

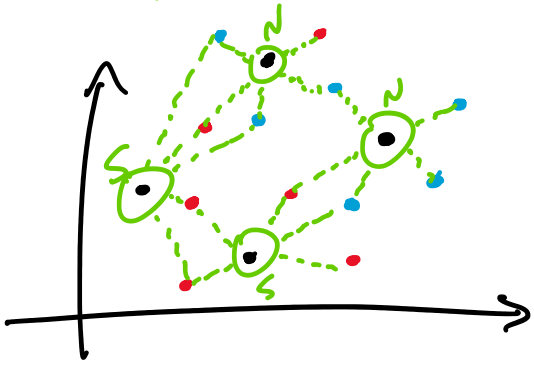
k = 3



for each •

1. Find distance to every other dot
2. Find the k "nearest neighbors"
3. Let them "vote"

k = 5



Small k → very specific
more volatile

Large k → less specific
less volatile

## The Procedure to Classify an Example $x$ with $k$-Nearest Neighbors

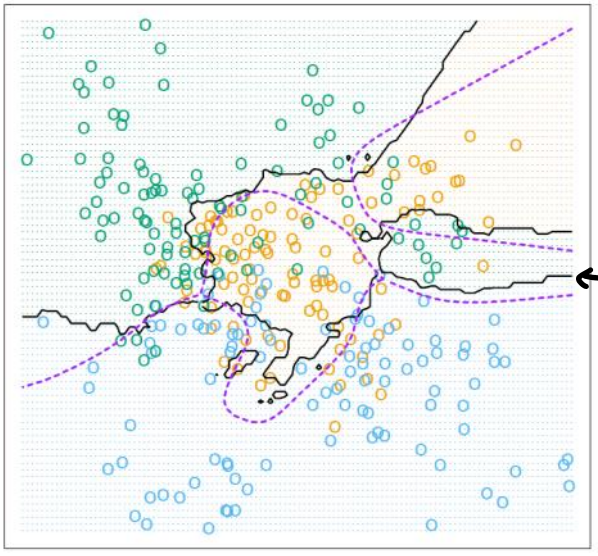
1. Choose a positive integer for $k$.
2. Find the distances $\|x - x_j\|$ for each $j = 1, \ldots, n$ with any norm you choose.
3. Find the points with the $k$ shortest distances from $x$ (the $k$ nearest "neighbors").
4. Assign the most frequent class among the $k$ nearest neighbors to $x$

Memory-based classifier, no model to fit. Given a point $x_0$, find the nearest $k$ training points $x_{(1)}, \ldots, x_{(k)}$ and classify using majority class among the $k$ neighbors

Typically, standardize each feature before using k-NN

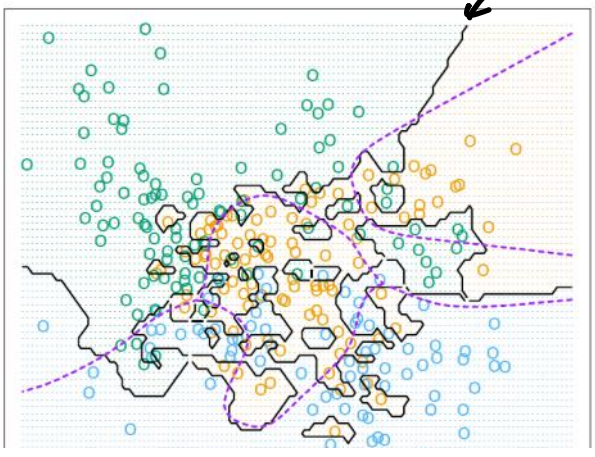
k-NN is good when a class has many prototypes + decision boundary is very irregular

15-Nearest Neighbors

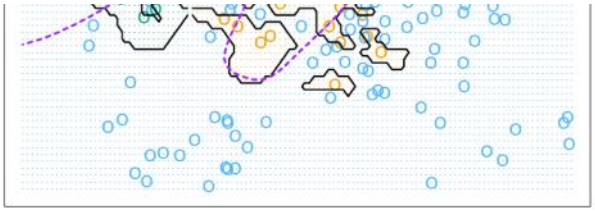


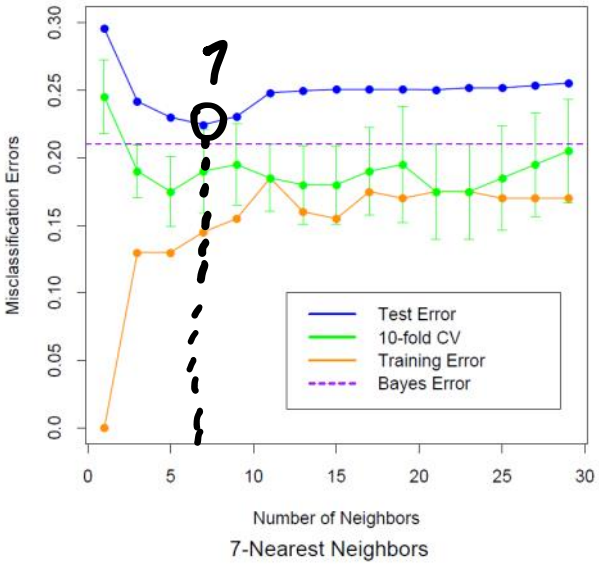
large k = more regular

1-Nearest Neighbor

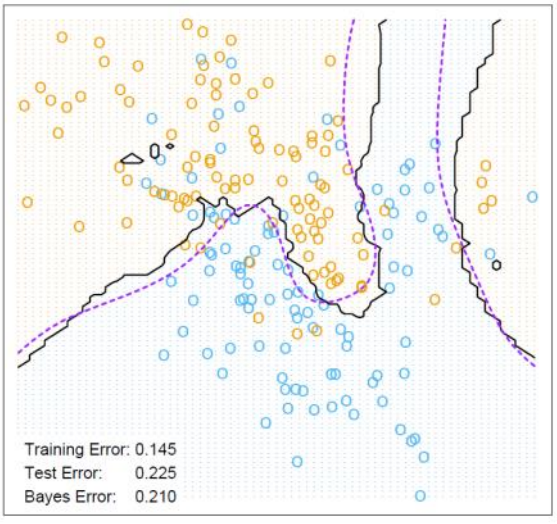


low k = less regular






7-Nearest Neighbors




7 is the best k here

# k-NN Code

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

## $k$-Nearest Neighbor Code

Let's write some code. I will use the style used by the popular machine learning library scikit-learn, where a classifier is written as a class in the programming sense (not to be confused with the "classes" of our classification problem) with hyperparameters as inputs. The classifier's class will have some functions that fit the model to the data and predict the class of input datapoints.

The 'fit' function for the $k$-nearest neighbors will do very little but record some data. The 'predict' function will carry out steps 1-4 above.

First, we import some libraries.

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sn

from scipy.stats import mode
from sklearn import datasets
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from tensorflow.keras.datasets import mnist
from tensorflow.keras.datasets import cifar10
```

```python
# Create a class for the k-nearest neighbor classifier
class kNearestNeighborClassifier:
    # constructor to save the hyperparameter k
    def __init__(self, k = 5):
        # initialize the number of neighbors to use
        self.neighbors = k

    # fit the model to the training data (for kNN, there's no actual fitting involved)
    def fit(self, X, y):

        # record the data and labels
        self.data = X
        self.labels = y

    # use the classifier to predict the classifications of the testing data
    def predict(self, X):
        # initialize the predicted classes
        yPredicted = np.empty([X.shape[0],1])

        # loop over the datapoints in X
        for row in range(X.shape[0]):
            datapoint = X[row,]

            # find the distances from the datapoint to each training point using the L2 norm
            distances = np.sqrt(np.sum(((self.data - datapoint)**2), axis = 1))

            # find the indices of the smallest k distances
            indices = np.argsort(distances)[:self.neighbors]

            # find the the class labels of the nearest neighbors
            nearestClasses = self.labels[indices]

            # determine the predicted class by finding the mode
            yPredicted[row] = int(mode(nearestClasses)[0][0])

        return yPredicted
```

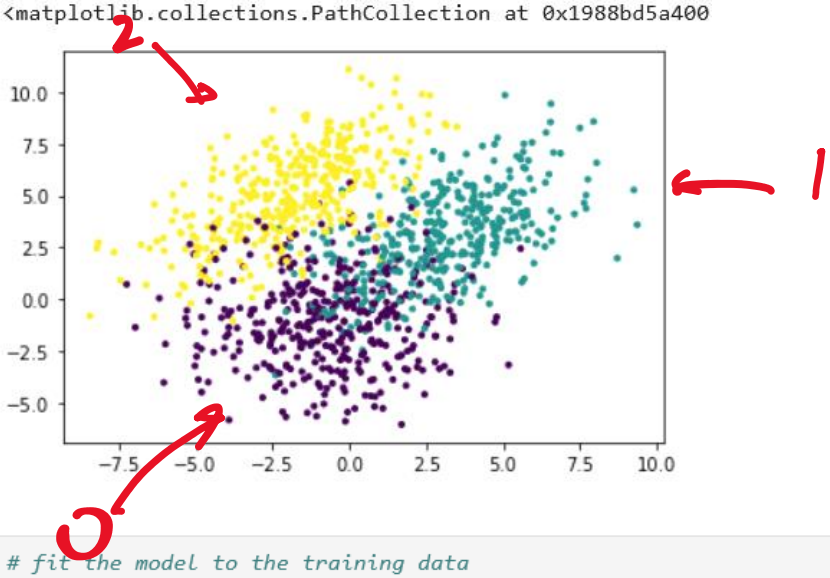
or np.linalg.norm

# Example with Randomly Generated Data

## Example: Randomly Generated Points

```
# number of points to generate
numberOfPoints = 500

# generate points from class 0
mean1 = np.array([-1, -1])
covariance1 = np.array([[5, 0], [0, 5]])
X1 = np.random.multivariate_normal(mean1, covariance1, numberOfPoints)

# generate points from class 1
mean2 = np.array([3, 3])
covariance2 = np.array([[5, 3], [3, 5]])
X2 = np.random.multivariate_normal(mean2, covariance2, numberOfPoints)

# generate points from class 2
mean3 = np.array([-2, 5])
covariance3 = np.array([[5, 3], [3, 5]])
X3 = np.random.multivariate_normal(mean3, covariance3, numberOfPoints)

# stack the points
X = np.vstack((X1, X2, X3))

# create a vector of the labels
Y = np.hstack((numberOfPoints * [0], numberOfPoints * [1], numberOfPoints * [2]))

# randomly choose 75% of the data to be the training set and 25% for the testing set
trainX, testX, trainY, testY = train_test_split(X, Y, test_size = 0.25, random_state = 1)

# plot the training set
plt.scatter(trainX[:,0], trainX[:,1], c = trainY, marker = '.')
```

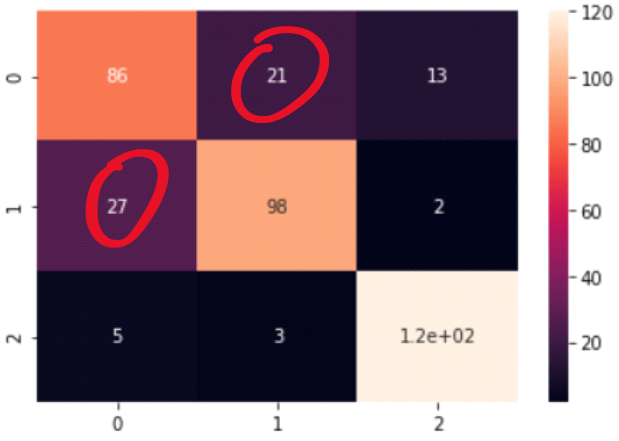
```
<matplotlib.collections.PathCollection at 0x1988bd5a400
```




```
# fit the model to the training data
model = kNearestNeighborClassifier(k = 3)
model.fit(trainX,trainY)

# predict the labels of the test set
predictedY = model.predict(testX)

plt.scatter(trainX[:,0], trainX[:,1], c = trainY, marker = '.')
```

```
# fit the model to the training data
model = kNearestNeighborClassifier(k = 3)
model.fit(trainX,trainY)

# predict the labels of the test set
predictedY = model.predict(testX)

plt.scatter(trainX[:,0], trainX[:,1], c = trainY, marker = '.')

# print quality metrics
print('\nClassification Report:\n\n', classification_report(testY, predictedY))
print('\nConfusion Matrix:\n')

sn.heatmap(confusion_matrix(testY, predictedY), annot = True)
```

```
Classification Report:

              precision    recall  f1-score   support

           0       0.73      0.72      0.72       120
           1       0.80      0.77      0.79       127
           2       0.89      0.94      0.91       128

    accuracy                           0.81       375
   macro avg       0.81      0.81      0.81       375
weighted avg       0.81      0.81      0.81       375

Confusion Matrix:

<AxesSubplot:>
```



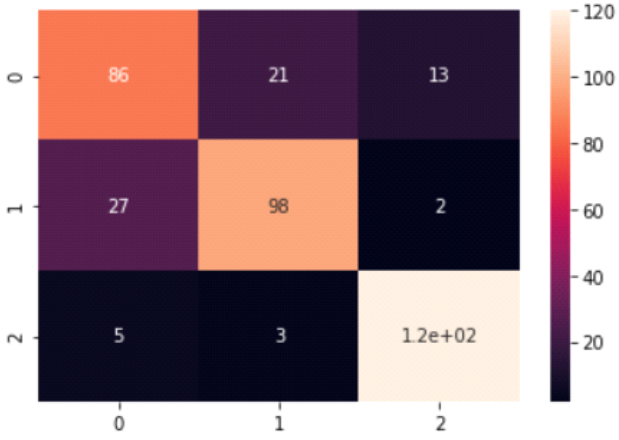

The code above computes the neighbors in a pretty brute-force way, which is pretty slow, so it is generally best to use an existing implementation like `KNeighborsClassifier` in `scikit-learn`.

```python
# fit the model to the training data
model = KNeighborsClassifier(3)
model.fit(trainX,trainY)

# predict the labels of the test set
predictedY = model.predict(testX)

plt.scatter(trainX[:,0], trainX[:,1], c = trainY, marker = '.')

# print quality metrics
print('\nClassification Report:\n\n', classification_report(testY, predictedY))
print('\nConfusion Matrix:\n')

sn.heatmap(confusion_matrix(testY, predictedY), annot = True)
```

```
Classification Report:

              precision    recall  f1-score   support

           0       0.73      0.72      0.72       120
           1       0.80      0.77      0.79       127
           2       0.89      0.94      0.91       128

    accuracy                           0.81       375
   macro avg       0.81      0.81      0.81       375
weighted avg       0.81      0.81      0.81       375
```

← Same results

```
Confusion Matrix:

<AxesSubplot:>
```



# MNIST with k-NN and Tuning

```
(trainX, trainY), (testX, testY) = mnist.load_data()

trainX = trainX.reshape(trainX.shape[0], trainX.shape[1] * trainX.shape[2]).astype('float')/255.0
testX = testX.reshape(testX.shape[0], testX.shape[1] * testX.shape[2]).astype('float')/255.0

devX, testX, devY, testY = train_test_split(testX, testY, test_size = 0.5)

trainX = trainX[:1000]
devX = devX[:1000]
testX = testX[:1000]
trainY = trainY[:1000]
devY = devY[:1000]
testY = testY[:1000]

bestAccuracy = [0, 0]

for k in range(1, 11):
    # build the knn classifier
    model = KNeighborsClassifier(k)

    # fit the knn classifier to the training data
    model.fit(trainX, trainY)

    # predict the labels of the test set
    predictedY = model.predict(devX)

    # compute the accuracy
    acc = accuracy_score(devY, predictedY)

    # print quality metrics
    print('Dev classification accuracy', k, 'neighbors is', acc)

    # save the hyperparameter k if better than found before
    if acc > bestAccuracy[0]:
        bestAccuracy = [acc, k]
```

```
print('\nThe best dev accuracy', bestAccuracy[0], 'occured with', bestAccuracy[1], 'neghbors')

# build the knn classifier
model = KNeighborsClassifier(bestAccuracy[1])

# fit the knn classifier to the training data
model.fit(trainX, trainY)

# predict the labels of the test set
predictedY = model.predict(testX)

# print quality metrics
print('\nTest Classification Report for', bestAccuracy[0], 'neighbors:\n\n', classification_report(testY, predictedY))
print('\nTest Confusion Matrix:\n')

sn.heatmap(confusion_matrix(testY, predictedY))
```

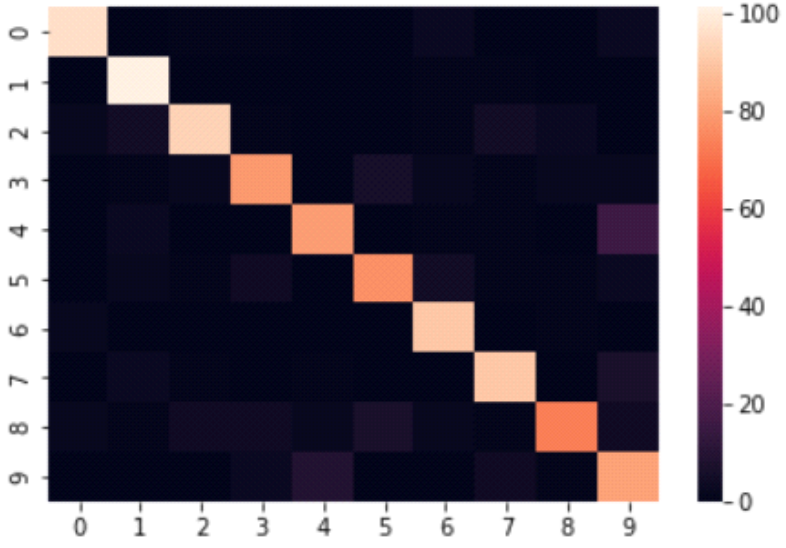
```
Dev classification accuracy 1 neighbors is 0.89
Dev classification accuracy 2 neighbors is 0.854
Dev classification accuracy 3 neighbors is 0.876
Dev classification accuracy 4 neighbors is 0.871
Dev classification accuracy 5 neighbors is 0.858
Dev classification accuracy 6 neighbors is 0.851
Dev classification accuracy 7 neighbors is 0.857
Dev classification accuracy 8 neighbors is 0.851
Dev classification accuracy 9 neighbors is 0.852
Dev classification accuracy 10 neighbors is 0.844

The best dev accuracy 0.89 occured with 1 neghbors
```

Test Classification Report for 0.89 neighbors:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.94 | 0.92 | 0.93 | 104 |
| 1 | 0.87 | 0.98 | 0.92 | 103 |
| 2 | 0.91 | 0.85 | 0.88 | 110 |
| 3 | 0.86 | 0.83 | 0.84 | 95 |
| 4 | 0.87 | 0.79 | 0.83 | 101 |
| 5 | 0.86 | 0.82 | 0.84 | 94 |
| 6 | 0.86 | 0.97 | 0.91 | 93 |
| 7 | 0.87 | 0.88 | 0.87 | 102 |
| 8 | 0.90 | 0.73 | 0.81 | 100 |
| 9 | 0.70 | 0.83 | 0.76 | 98 |
| accuracy | | | 0.86 | 1000 |
| macro avg | 0.86 | 0.86 | 0.86 | 1000 |
| weighted avg | 0.86 | 0.86 | 0.86 | 1000 |

Test Confusion Matrix:

<AxesSubplot:>



```
(trainX, trainY), (testX, testY) = mnist.load_data()

trainX = trainX.reshape(trainX.shape[0], trainX.shape[1] * trainX.shape[2]).astype('float')/255.0
testX = testX.reshape(testX.shape[0], testX.shape[1] * testX.shape[2]).astype('float')/255.0

trainX = trainX[:10000]
testX = testX[:10000]
trainY = trainY[:10000]
testY = testY[:10000]

bestAccuracy = [0, 0]

for k in range(1, 11):
    # build the knn classifier
    model = KNeighborsClassifier(k)

    # fit the knn classifier to the training data
    model.fit(trainX, trainY)

    mean_cv_scores = np.mean(cross_val_score(model, trainX, trainY, cv = 5))

    # print quality metrics
    print('Mean CV accuracy for', k, 'neighbors is', mean_cv_scores)

    # save the hyperparameter k if better than found before
    if mean_cv_scores > bestAccuracy[0]:
        bestAccuracy = [mean_cv_scores, k]

print('\nThe best cross-validated accuracy', bestAccuracy[0], 'occured with', bestAccuracy[1], 'neghbors')
```

```
# build the knn classifier
model = KNeighborsClassifier(bestAccuracy[1])

# fit the knn classifier to the training data
model.fit(trainX, trainY)

# predict the labels of the test set
predictedY = model.predict(testX)

# print quality metrics
print('\nTest Classification Report for', bestAccuracy[0], 'neighbors:\n\n', classification_report(testY, predictedY))
print('\nTest Confusion Matrix:\n')

sn.heatmap(confusion_matrix(testY, predictedY))
```

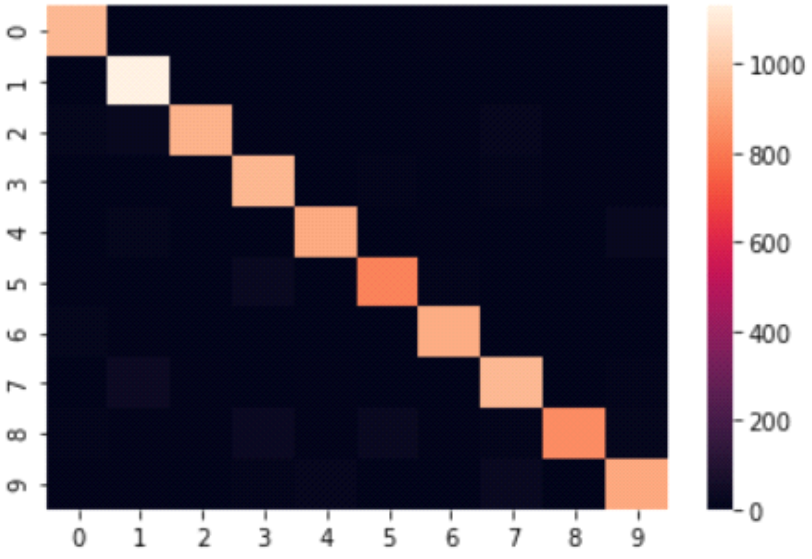
```
Mean CV accuracy for 1 neighbors is 0.9393
Mean CV accuracy for 2 neighbors is 0.9307000000000001
Mean CV accuracy for 3 neighbors is 0.9395000000000001
Mean CV accuracy for 4 neighbors is 0.9405000000000001
Mean CV accuracy for 5 neighbors is 0.9377000000000001
Mean CV accuracy for 6 neighbors is 0.9362999999999999
Mean CV accuracy for 7 neighbors is 0.9359999999999999
Mean CV accuracy for 8 neighbors is 0.9343999999999999
Mean CV accuracy for 9 neighbors is 0.9345000000000001
Mean CV accuracy for 10 neighbors is 0.9341000000000002

The best dev accuracy 0.9405000000000001 occured with 4 neghbors
```

Test Classification Report for 0.9405000000000001 neighbors:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.94 | 0.99 | 0.96 | 980 |
| 1 | 0.91 | 0.99 | 0.95 | 1135 |
| 2 | 0.97 | 0.92 | 0.95 | 1032 |
| 3 | 0.92 | 0.95 | 0.94 | 1010 |
| 4 | 0.96 | 0.95 | 0.95 | 982 |
| 5 | 0.95 | 0.93 | 0.94 | 892 |
| 6 | 0.96 | 0.97 | 0.97 | 958 |
| 7 | 0.93 | 0.94 | 0.93 | 1028 |
| 8 | 0.98 | 0.87 | 0.92 | 974 |
| 9 | 0.94 | 0.92 | 0.93 | 1009 |
| accuracy | | | 0.94 | 10000 |
| macro avg | 0.95 | 0.94 | 0.94 | 10000 |
| weighted avg | 0.95 | 0.94 | 0.94 | 10000 |

Test Confusion Matrix:

<AxesSubplot:>



# Geometry of k-NN

Sunday, February 21, 2021 5:35 PM

In the preceding sections we considered a parametric approach for estimating the likelihood $P(\mathbf{x}|c_i)$. In this section, we consider a non-parametric approach, which does not make any assumptions about the underlying joint probability density function. Instead, it directly uses the data sample to estimate the density, for example, using the density estimation methods from Chapter 15. We illustrate the non-parametric approach using nearest neighbors density estimation from Section 15.2.3, which leads to the *K nearest neighbors* (KNN) classifier.

Let $\mathbf{D}$ be a training dataset comprising $n$ points $\mathbf{x}_i \in \mathbb{R}^d$, and let $\mathbf{D}_i$ denote the subset of points in $\mathbf{D}$ that are labeled with class $c_i$, with $n_i = |\mathbf{D}_i|$. Given a test point $\mathbf{x} \in \mathbb{R}^d$, and $K$, the number of neighbors to consider, let $r$ denote the distance from $\mathbf{x}$ to its $K$th nearest neighbor in $\mathbf{D}$.

Consider the $d$-dimensional hyperball of radius $r$ around the test point $\mathbf{x}$, defined as

$$B_d(\mathbf{x}, r) = \{\mathbf{x}_i \in \mathbf{D} \mid \|\mathbf{x} - \mathbf{x}_i\| \leq r\}$$

Here $\|\mathbf{x} - \mathbf{x}_i\|$ is the Euclidean distance between $\mathbf{x}$ and $\mathbf{x}_i$. However, other distance metrics can also be used. We assume that $|B_d(\mathbf{x}, r)| = K$.

Let $K_i$ denote the number of points among the $K$ nearest neighbors of $\mathbf{x}$ that are labeled with class $c_i$, that is

$$K_i = \{\mathbf{x}_j \in B_d(\mathbf{x}, r) \mid y_j = c_i\}$$

The class conditional probability density at $\mathbf{x}$ can be estimated as the fraction of points from class $c_i$ that lie within the hyperball divided by its volume, that is

$$\hat{f}(\mathbf{x}|c_i) = \frac{K_i/n_i}{V} = \frac{K_i}{n_i V} \tag{18.10}$$

where $V = \text{vol}(B_d(\mathbf{x}, r))$ is the volume of the $d$-dimensional hyperball [Eq. (6.10)].

Using Eq. (18.6), the posterior probability $P(c_i|\mathbf{x})$ can be estimated as

$$P(c_i|\mathbf{x}) = \frac{\hat{f}(\mathbf{x}|c_i)\hat{P}(c_i)}{\sum_{j=1}^{k} \hat{f}(\mathbf{x}|c_j)\hat{P}(c_j)}$$

However, because $\hat{P}(c_i) = \frac{n_i}{n}$, we have

$$\hat{f}(\mathbf{x}|c_i)\hat{P}(c_i) = \frac{K_i}{n_i V} \cdot \frac{n_i}{n} = \frac{K_i}{nV}$$

Thus the posterior probability is given as

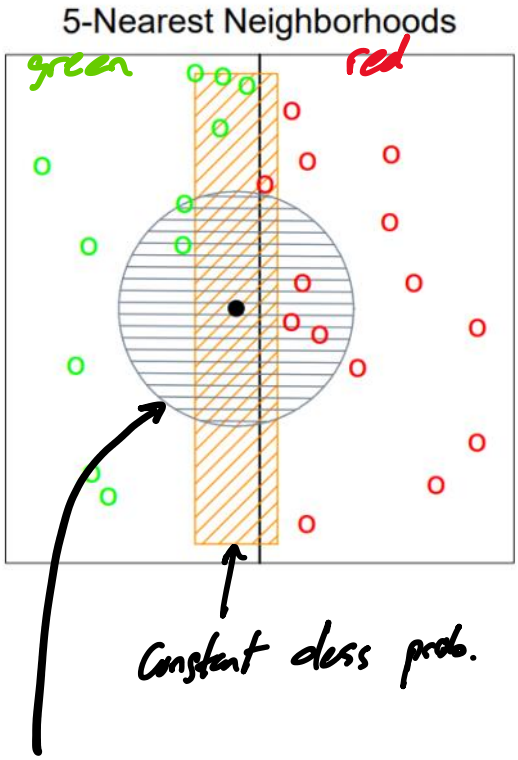
$$P(c_i|\mathbf{x}) = \frac{\frac{K_i}{nV}}{\sum_{j=1}^{k} \frac{K_j}{nV}} = \frac{K_i}{K}$$

Finally, the predicted class for $\mathbf{x}$ is

$$\hat{y} = \arg\max_{c_i} \{P(c_i|\mathbf{x})\} = \arg\max_{c_i} \left\{\frac{K_i}{K}\right\} = \arg\max_{c_i} \{K_i\} \tag{18.11}$$

Because $K$ is fixed, the KNN classifier predicts the class of $\mathbf{x}$ as the majority class among its $K$ nearest neighbors.

In high dims, neighbors can be very far away, causing bias and degrading performance...




Two features

black query point...

NN assumes class probabilities are roughly constant in a neighborhood, but here, class probs only differ in the horizontal direction

⇒ tall rect. Nhd makes more sense

Generally, adapting the metric (& thus, Neighborhood shape) so the neighborhoods stretch out in directions where class probabilities do not change too much

Idea: Take a Nhood of, say, 50 points of each query point & use it to decide how [illegible] the Nh (i.e. change the metric)

each query [ ]
to deform the Nh (i.e. change the metric)
(diff. metric at each query point)

Idea: stretch nh orthogonal to line connecting class centroids... this coincides with the linear discriminant boundary → the direction where class probs. change least...

This local discriminant model requires only within-class & between-class covariances of the local points to determine optimal neighborhood shapes

The discriminant-adapted NN (DANN) metric at query point $x_0$ is

$$D(x, x_0) = (x - x_0)^T \Sigma (x - x_0)$$

where $$\Sigma = W^{-1/2}\left[W^{-1/2} B W^{-1/2} + \varepsilon I\right] W^{-1/2}$$

$$= W^{-1/2}\left[B^* + \varepsilon I\right] W^{-1/2}$$

$\varepsilon = 1$ is typical... rounds the nhood

where $W$ is the pooled within-class covariance matrix

$$W = \sum_{i=1}^{k} \pi_i W_i$$

and B is the between-class covariance matrix

W + B computed from 50 NNs

e.vectors e.values

$W = Q \Lambda Q^{-1}$

$W^{-1/2} = Q \Lambda^{-1/2} Q^{T}$ ($\frac{-1}{2}$ power...)

This complicated formula is actually quite simple in its operation. It first spheres the data with respect to $\mathbf{W}$, and then stretches the neighborhood in the zero-eigenvalue directions of $\mathbf{B}^*$ (the between-matrix for the sphered data ). This makes sense, since locally the observed class means do not differ in these directions. The $\epsilon$ parameter rounds the neighborhood, from an infinite strip to an ellipsoid, to avoid using points far away from the query point. The value of $\epsilon = 1$ seems to work well in general. Figure 13.14 shows the resulting neighborhoods for a problem where the classes form two concentric circles. Notice how the neighborhoods stretch out orthogonally to the decision boundaries when both classes are present in the neighborhood. In the pure regions with only one class, the neighborhoods remain circular;

in these cases the between matrix $\mathbf{B} = 0$, and the $\mathbf{\Sigma}$ in (13.8) is the identity matrix.

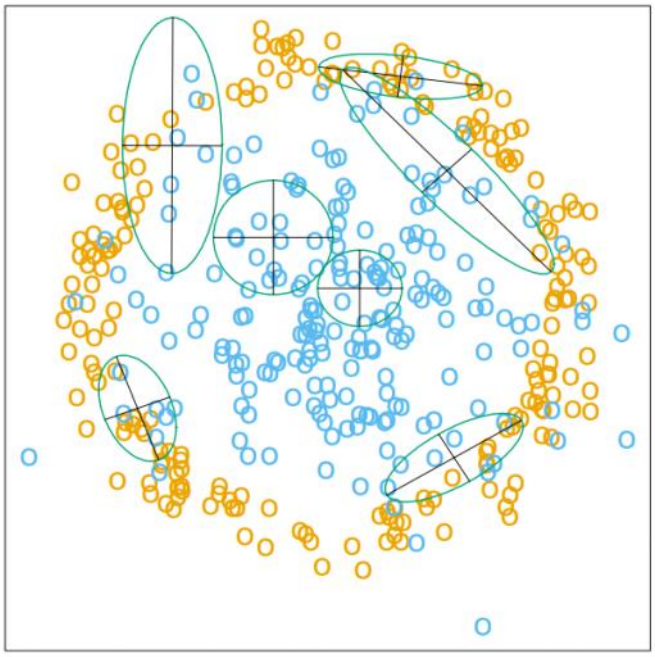


**FIGURE 13.14.** *Neighborhoods found by the DANN procedure, at various query points (centers of the crosses). There are two classes in the data, with one class surrounding the other.* 50 *nearest-neighbors were used to estimate the local metrics. Shown are the resulting metrics used to form* 15*-nearest-neighborhoods.*