More on Machine Learning Models

Battista Biggio

Outline

- Recap:
  - Loss functions and training with gradient descent
  - Regularization to prevent overfitting
  - Overview of linear models
  - Support Vector Machines (with kernels)
  - Logistic Classifier

- Today:
  - Gaussian Classifier
  - Nearest Mean Centroid (NMC) Classifier
  - k-Nearest Neighbor (kNN) Classifier
  - Decision Trees and Random Forests

Generative Models and MAP Criterion

- We have discussed linear models using a discriminative (non-parametric) approach
  - They directly estimate (**w**,b) without making *any* assumption on the underlying data distribution

- Generative (parametric) models aim to estimate the data distribution to find the decision function and take optimal decisions by maximizing the so-called "*a-posteriori*" probability
  - The a-posteriori probability of sample x belonging to class y is given by the Bayes' Theorem:

$$y_k^\star = \arg\max_{y_k} p(y_k|x) = \frac{p(x|y_k)p_k}{p(x)}$$

  - p_k is the prior probability of each class (fraction of samples belonging to class y)
  - $p(x|y)$ is the "likelihood" or class-conditional distribution
  - $p(x)$ is the "evidence", i.e., the probability of sampling/observing x, regardless of the class label
    - This is just a normalization factor, obtained via *marginalization*: $p(x) = \sum_k p(x|y_k)p_k$

The Base-Rate Fallacy in the Drunk-Driver Problem

- Problem setup:
  - Y={sober, drunk}; X={positive, negative}
  - p(X=positive | Y=drunk) = 100% (the test never fails to detect drunk drivers)
  - P(X=positive | Y=sober) = 5% (the test fails with 5% probability when the driver is sober)
  - p(Y=drunk) = 1/1000 and p(Y=sober) = 999/1000 (classes are very imbalanced, “drunk” is a rare event)

- We’d like to compute the probability that the driver is drunk, given that the test was positive
  - First, we compute the *probability of observing a positive test outcome* (evidence of x=positive)
    - p(X=positive) = p(positive | drunk) p(drunk) + p(positive | sober) p(sober) ~= 0.05
  - Then, using Bayes’ Theorem, we can compute the requested probability
    - p(Y= drunk | X= positive) = p(positive | drunk) p(drunk) / p(positive) ~= 0.02

- This means that in only 2% of the cases when the test is positive, the driver is drunk (~1/50)

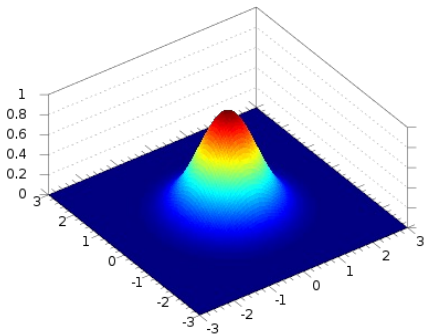
Gaussian Classifier - Prediction

- Bayesian classifier that uses MAP for predictions:

- Bayesian classifier that uses MAP for predictions:
- The likelihood is computed using a multivariate Gaussian distribution
  - This means that each class is modeled as Gaussian, with parameters μ_k, Σ_k
- Recall also that:
  - p_k is the prior probability of class y_k;
  - the evidence is obtained, as usual, by marginalization over the classes, i.e.,

- The likelihood is computed using a multivariate Gaussian distribution
  - This means that each class is modeled as Gaussian, with parameters μ_k, Σ_k

$$p(x|y_k) = \mathrm{g}(x; \mu_k, \Sigma_k) = \frac{1}{\sqrt{(2\pi)^d \det \Sigma_k}} \exp\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)\right)$$

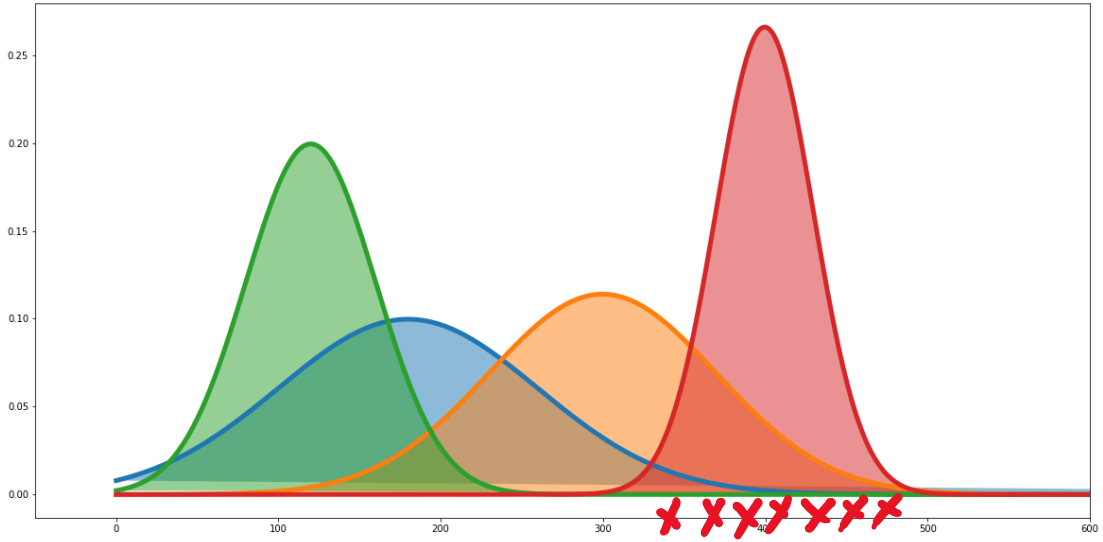


- Recall also that:
  - p_k is the prior probability of class y_k;
  - the evidence is obtained, as usual, by marginalization over the classes, i.e.,

$$p(x) = \sum_k p(x|y_k)p_k$$

Gaussian Classifier - Training

- During training, we aim to fit one Gaussian distribution per class.
  - But how do we find the best parameters μ_k, Σ_k for each Gaussian?
  - What is the meaning of *best parameters* here?



Which of these Gaussian distributions is a better fit to the 'x' data points? Why?

Gaussian Classifier - Training

- **Maximum Likelihood Estimation (MLE).** MLE defines the best parameters of our model $\theta^\star$ as those maximizing the likelihood L that the data $x_1, \dots, x_n$ is generated by the model:

$$\theta^\star = \underset{\theta}{\operatorname{argmax}}\, L(x_1, \dots, x_n \,|\theta)$$

- In our case, we aim to fit one Gaussian distribution per class. Thus, for each class,
  - we first extract the samples $x_1, \dots, x_n$ for that class, and then
  - estimate the parameters $\theta = (\mu_k, \Sigma_\mathrm{k})$ for that class via MLE.

- **Good news:** MLE for Gaussian fitting can be solved in closed form!
  - The optimal $\mu_k^\star, \Sigma_\mathrm{k}^\star$ values are obtained as the **sample mean** and the **sample covariance**

$$\mu_k^\star = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \Sigma_k^\star = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_k^\star)(x_i - \mu_k^\star)^\mathrm{T}$$

  - ... and these equations are already implemented in many statistical tools and libraries!

Exercise 1

- Implement a Gaussian classifier that
  1. fits a Gaussian distribution to each training class and
  2. predicts test samples by assigning them to the class with the max. a-posteriori probability (MAP)

- Test it on some Gaussian dataset
  - Measure test error
  - Visualize the decision regions

Solution (small excerpt from the notebook code)

```
class CClassifierGaussian:
    """
    Class implementing a Gaussian classifier
    """

    def fit(self, x, y):
        """Estimate priors, centroids and covariances with
        maximum likelihood estimates from the training data x,y.
        """
        n_classes = np.unique(y).size
        n_features = x.shape[1]

        self._priors = np.zeros(shape=(n_classes,))
        self._centroids = np.zeros(shape=(n_classes, n_features))
        self._covariances = np.zeros(shape=(n_classes, n_features, n_features))

        for k in range(n_classes):
            self._centroids[k, :] = x[y == k, :].mean(axis=0)
            self._priors[k] = (y == k).mean()
            self._covariances[k, :, :] = np.cov(x[y == k, :].T)

        self._priors /= self._priors.sum()  # ensure priors sum up to 1
        return self
```

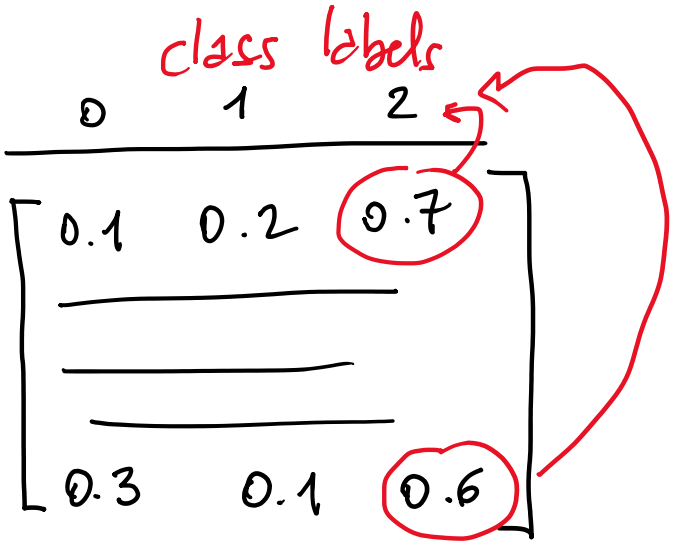
These are the maximum-likelihood estimates for the parameters of each class

Solution (small excerpt from the notebook code)

```python
def decision_function(self, x):
    """Return posterior estimates for each class"""»

    n_samples = x.shape[0]
    n_classes = self._centroids.shape[0]
    scores = np.zeros(shape=(n_samples, n_classes))
```

The matrix *scores* will contain, for each row (sample), the posterior probability predicted for each class.

Each sample will then be assigned to the class exhibiting the maximum a-posteriori probability estimate



Solution (small excerpt from the notebook code)

```python
def decision_function(self, x):
    """Return posterior or joint probability estimates for each class,
    depending on whether posterior=True or False."""
    n_samples = x.shape[0]
    n_classes = self._centroids.shape[0]
    scores = np.zeros(shape=(n_samples, n_classes))
    for k in range(n_classes):
        likelihood_k = mvn.pdf(
            x, mean=self._centroids[k, :], cov=self._covariances[k, :, :])
        scores[:, k] = self._priors[k] * likelihood_k  # joint probability
    if self.posterior:
        # if posterior probs are required, divide joint probs by evidence
        evidence = scores.sum(axis=1)
        for k in range(n_classes):
            # normalize per row to estimate posterior
            scores[:, k] /= evidence
    return scores

def predict(self, x):
    """Return predicted labels."""
    scores = self.decision_function(x)
    y_pred = np.argmax(scores, axis=1)
    return y_pred
```

Compute Gaussian PDF at x. This is our likelihood:

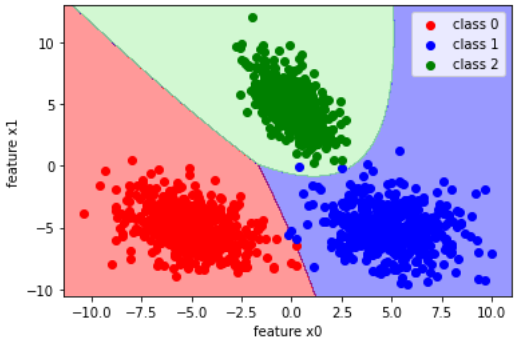
$$\frac{1}{\sqrt{(2\pi)^d \det \Sigma_k}} \exp\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1} (x-\mu_k)\right)$$

Then multiply it by the prior to obtain the joint probability, and divide by the evidence p(x) to obtain the posterior

Take the index of the maximum posterior to predict the class label

Solution

```
n_samples = [500, 500, 500]
centroids = [[-5, -5],
             [+5, -5],
             [0, +5]]
cov=[[[3, -1],
      [-1, 3]],
     [[3, -0.5],
      [-0.5, 3]],
     [[1, -1],
      [-1, 3]]]

# generate data
x_tr, y_tr = make_gaussian_dataset(n_samples, centroids, cov=cov)
x_ts, y_ts = make_gaussian_dataset(n_samples, centroids, cov=cov)

clf = CClassifierGaussian()
clf.fit(x_tr, y_tr)
plot_decision_regions(x_tr, y_tr, classifier=clf)
plot_dataset(x_tr, y_tr)
plt.show()

scores = clf.decision_function(x_ts)
y_pred = clf.predict(x_ts)

print('Estimated priors: ', clf.priors)
print('Estimated centroids (with MLE): ', clf.centroids)
print('Estimated covariances (with MLE): ', clf.covariances)

print('Test error (%): ', (y_pred != y_ts).mean()*100)
```




```
Estimated priors:
[0.33333333 0.33333333 0.33333333]

Estimated centroids (with MLE):
[[-5.02818576 -5.07365035]
 [ 4.95564731 -5.04938789]
 [-0.03591899 4.87763586]]

Estimated covariances (with MLE):
[[[ 2.89255506 -1.05314784]
  [-1.05314784 2.77306075]]

 [[ 2.88868113 -0.46682182]
  [-0.46682182 3.04256309]]

 [[ 1.04041627 -1.09369177]
  [-1.09369177 3.26337424]]]

Test error (%): 0.26666666666666666
```

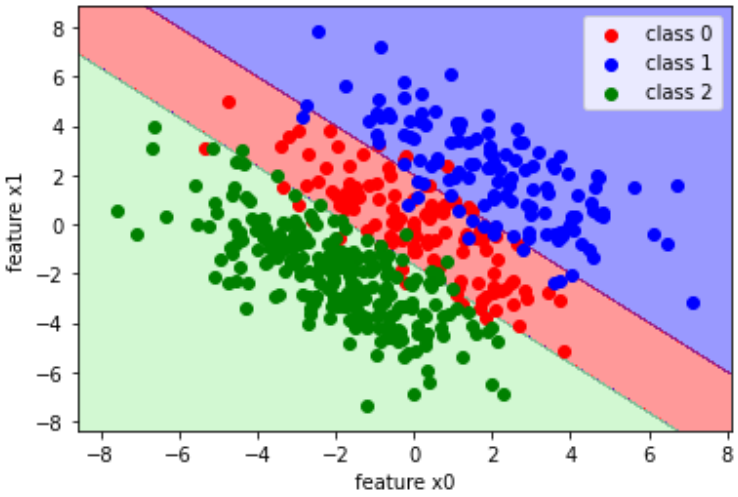
Exercise 2

- Visualize the optimal decision regions for a 3-class Gaussian Classifier with parameters:
  - $p_0 = p_1 = \frac{1}{4}, p_2 = \frac{1}{2}$
  - $\mu_0 = [0, 0]^T,\ \mu_1 = [2, 2]^T, \mu_2 = [-2, -2]^T$
  - $\Sigma_0 = \Sigma_1 = \Sigma_2 = \begin{bmatrix} 4 & -3 \\ -3 & 4 \end{bmatrix}$

Solution

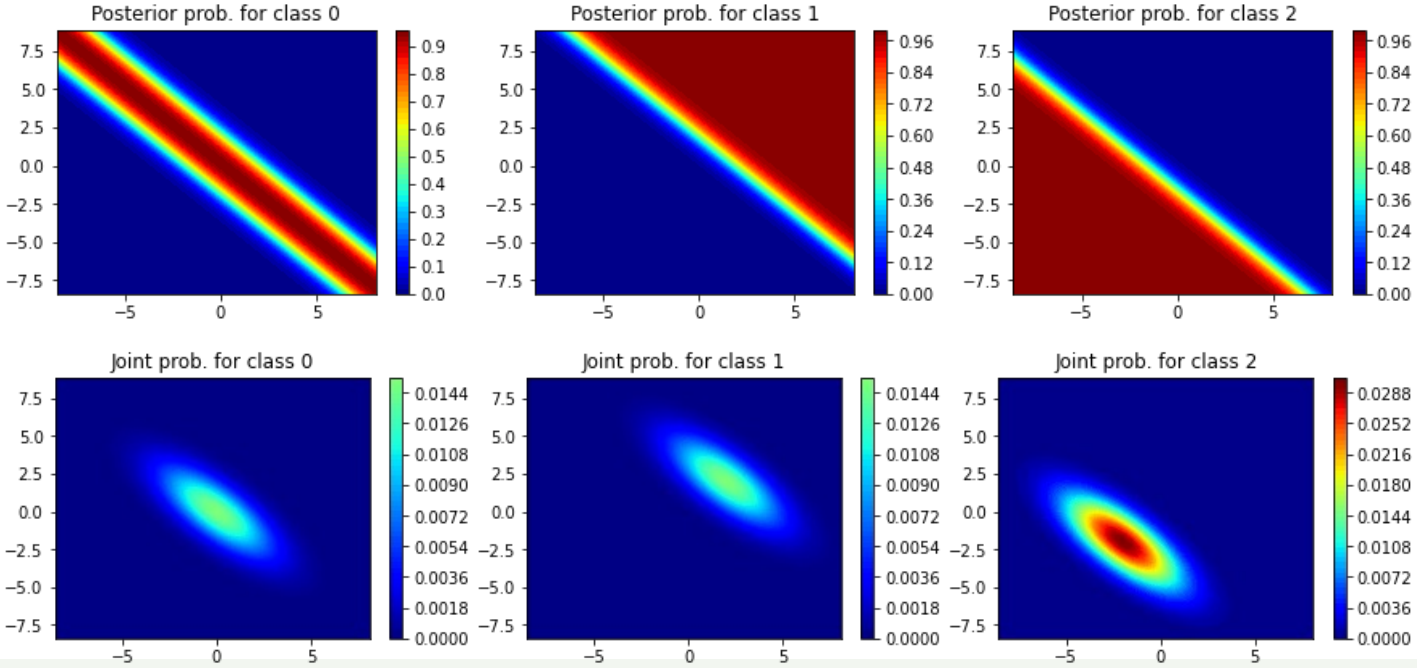
- In this case, we set the parameters directly to the Gaussian Classifier (instead of estimating them via *fit*), and then visualize the decision regions and boundaries

```
priors = np.array([1/4, 1/4, 1/2])
centroids = [[0, 0],
             [2, 2],
             [-2, -2]]
cov=[[[4, -3], [-3, 4]],
     [[4, -3], [-3, 4]],
     [[4, -3], [-3, 4]]]

clf = CClassifierGaussian()
# we do not estimate the parameters here,
# but use the true ones
# clf.fit(x_tr, y_tr)
clf._priors = np.array(priors)
clf._centroids = np.array(centroids)
clf._covariances = np.array(cov)
plot_decision_regions(x_tr, y_tr, classifier=clf)
plot_dataset(x_tr, y_tr)
plt.show()
```



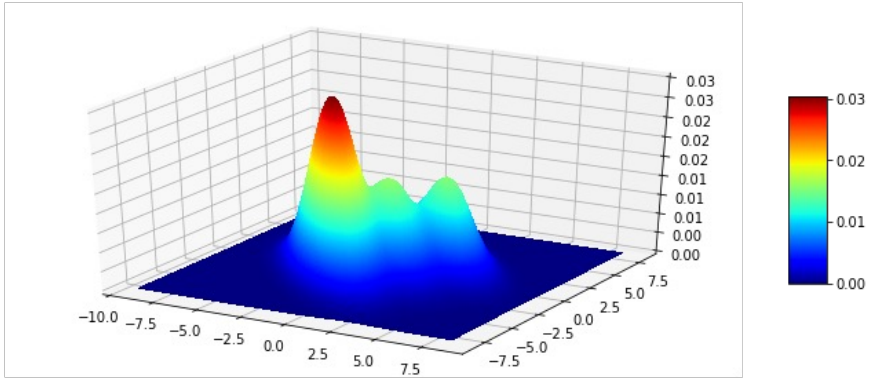
Solution

- We can also visualize the posterior and joint probabilities for each class



Solution

- Finally, we also plot the evidence *p(x)* along the z-axis in a 3D plot



- Note here the overlap among the three Gaussian distributions, one per class, and that the Gaussian associated to the highest prior (1/2) has a higher peak

Nearest Mean Centroid (NMC) Classifier

Ex. 3: NMC – Learning & Classification

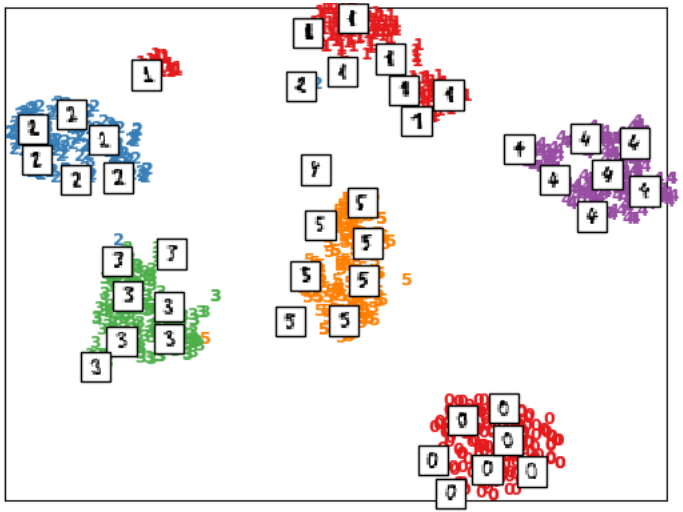
- During training, NMC is given a training set consisting of pairs (**x**, y) of samples along with their labels. For each class y
  - NMC estimates the mean of the samples in class y
  - stores the mean vector (centroid)

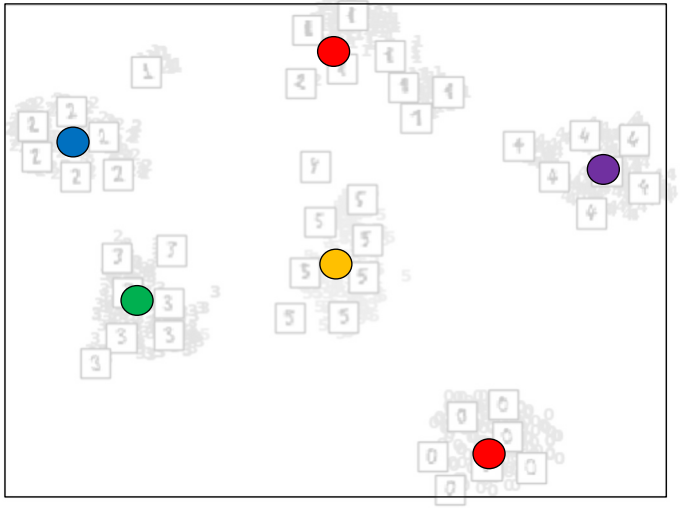
- During classification, NMC assigns the current test sample **x** to the class whose mean vector (centroid) is the closest one to **x**

- Implement the functions
  - `centroids = fit(x,y)`, corresponding to the learning phase, and
  - `y_pred = predict(x)`, corresponding to the classification phase, where `y_pred` is the label of the predicted class

NMC Classifier: «*fit*»

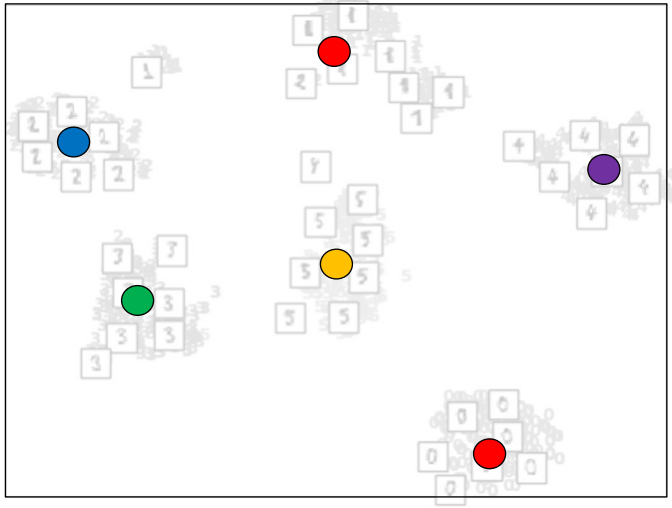
- Each sample is represented as a point in the feature space (e.g., for images, each dimension may correspond to the value of each pixel)
  - we can plot different classes with different colors / markers
  - *fit* estimates the average (centroid) for each class

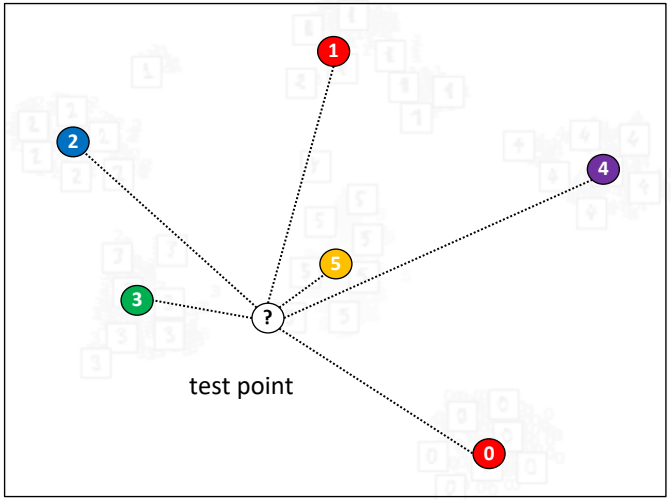




NMC Classifier: «*predict*»

- *Predict* computes the Euclidean distance of a given test point against all centroids, and assigns it to the class of the closest centroid
  - The test point ('?') below is classified as a '5', as it is closer to the centroid of class '5'
  - The length of each dashed line is the distance between the test point and the given centroid





Exercise 3: Solution

```
import numpy as np


def fit(x, y):
    n_classes = np.unique(y).size
    n_features = x.shape[1]

    centroids = np.zeros(shape=(n_classes, n_features))
    for k in xrange(n_classes):
        centroids[k] = x[y == k, :].mean(axis=0)
    return centroids

def predict(x, centroids):
    n_samples = x.shape[0]
    n_classes = centroids.shape[0]
    distances = np.zeros(shape=(n_samples, n_classes))

    for k in xrange(n_classes):
        distances[:,k] = np.linalg.norm(x-centroids[k, :], axis=1)
    y_pred = np.argmin(distances, axis=1)

    return y_pred, distances
```

Let's create a *class*

```
class CNearestMeanClassifier(object):

    def __init__(self):
        self._centroids = None

    def fit(self, x, y):
        n_classes = np.unique(y).size
        n_features = x.shape[1]
        centroids = np.zeros(shape=(n_classes, n_features))
        for k in xrange(n_classes):
            centroids[k] = x[y == k, :].mean(axis=0)
        self._centroids = centroids
        return

    def predict(self, x):
        n_samples = x.shape[0]
        n_classes = self._centroids.shape[0]
        distances = np.zeros(shape=(n_samples, n_classes))
        for k in xrange(n_classes):
            distances[:, k] = np.linalg.norm(x - self._centroids[k, :], axis=1)
        y_pred = np.argmin(distances, axis=1)
        return y_pred, distances
```

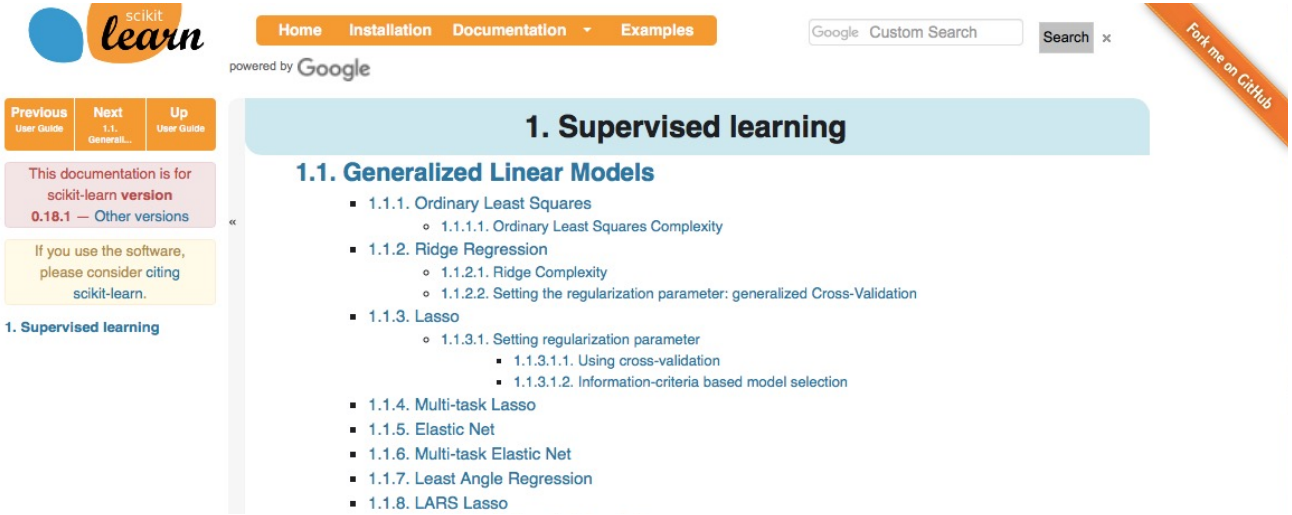
Class Properties

- Python decorator to access class private members
  - See also 'setters'

```
class CNearestMeanClassifier(object):
    """Class implementing a nearest mean classifier"""

    def __init__(self):
        self._centroids = None
        return

    @property
    def centroids(self):
        return self._centroids

    [...]
```

Scikit-learn Classifiers

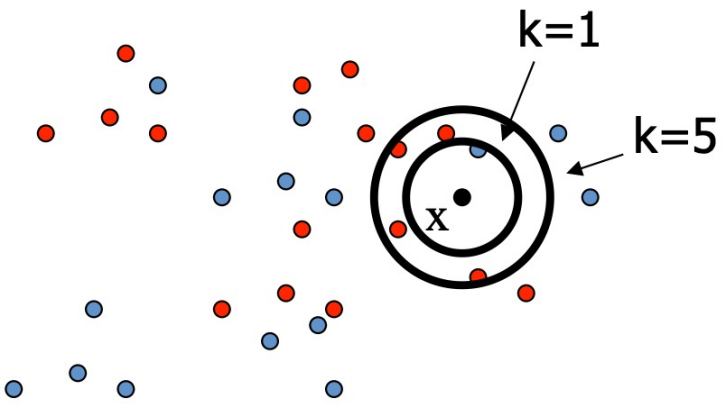
- Check http://scikit-learn.org/stable/supervised_learning.html
- NearestCentroid implements our CNearestMeanClassifier
  - http://scikit-learn.org/stable/modules/neighbors.html#nearest-centroid-classifier



k-Nearest Neighbor (kNN) Classifier

k-Nearest Neighbor (kNN) Classifier

- Training amounts to storing the full dataset
- At test time
  - we compute distances between the input **x** and the training samples
  - we select the closest k neighbors
  - the label is assigned by majority voting
- Different distance metrics can be selected
- Similar decision function to SVM-RBF models, when distance is Euclidean

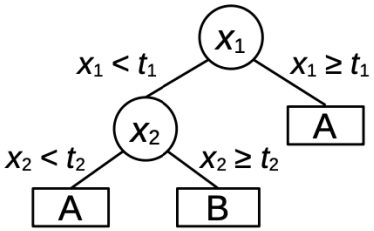


Decision Trees and Random Forests

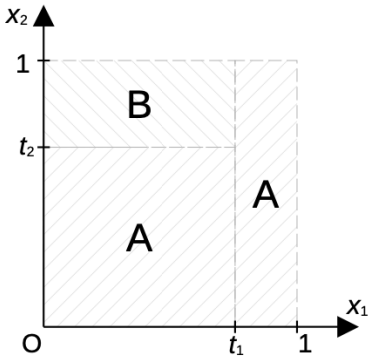
Decision Trees

- A decision tree is a tree graph that represents a mutually exclusive set of classification rules of the form:

  IF *condition* THEN *class*



- where
  - *condition* is a Boolean expression consisting of the conjunction (AND) of one or more conditions that involve the feature values
  - *class* is the class label predicted by the DT for any instance which fulfils the condition



- Fast and works well with qualitative/categorical features too

- Complexity of the decision function depends on the tree depth

Random Forests

- Classifier ensemble combining decision trees via majority voting

- Individual trees are built by using *bootstrap aggregation* (bagging) and the random subspace method
  - Individual training sets are obtained by sampling the training data with replacement (bagging)
  - and randomly selecting a subset of the input features