

# PATTERN RECOGNITION AND MACHINE LEARNING

Slide Set 4: Machine Learning—Linear Models

October 2019

Heikki Huttunen  
[heikki.huttunen@tuni.fi](mailto:heikki.huttunen@tuni.fi)  
Signal Processing  
Tampere University

# Classification

- Many machine learning problems can be posed as **classification** tasks.
- Most classification tasks can be posed as a problem of partitioning a vector space into disjoint regions.
- These problems consist of following components:

Samples:  $\mathbf{x}[0], \mathbf{x}[1], \dots, \mathbf{x}[N - 1] \in \mathbb{R}^P$

Class labels:  $y[0], y[1], \dots, y[N - 1] \in \{1, 2, \dots, C\}$

Classifier:  $F(\mathbf{x}) : \mathbb{R}^P \mapsto \{1, 2, \dots, C\}$

- Now, the task is to find the function  $F$  that maps the samples most accurately to their corresponding labels.
- For example: Find the function  $F$  that minimizes the number of erroneous predictions, *i.e.*, the cases  $F(\mathbf{x}[k]) \neq y[k]$ .

# Regression

- The second large class of machine learning problems consists of **regression** tasks.
- For regression, the output is real-valued instead of categorical.
- These problems consist of following components:

Inputs:  $\mathbf{x}[0], \mathbf{x}[1], \dots, \mathbf{x}[N - 1] \in \mathbb{R}^P$

Targets:  $y[0], y[1], \dots, y[N - 1] \in \mathbb{R}$

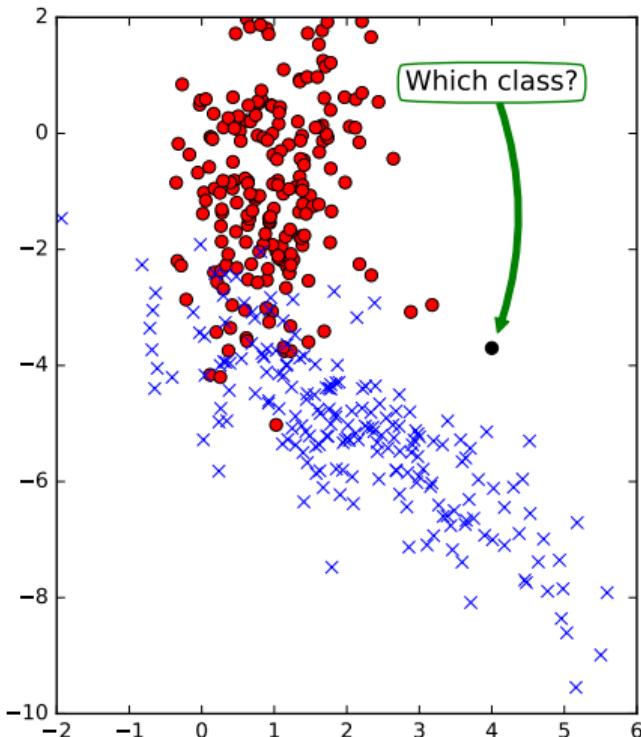
Predictor:  $F(\mathbf{x}) : \mathbb{R}^P \mapsto \mathbb{R}$

- This time the task is to find the function  $F$  that maps the samples most accurately to their corresponding targets.
- For example: Find the function  $F$  that minimizes the squared sum of distances between predictions and targets:

$$\mathcal{E} = \sum_{k=0}^{N-1} (y[k] - F(\mathbf{x}[k]))^2.$$

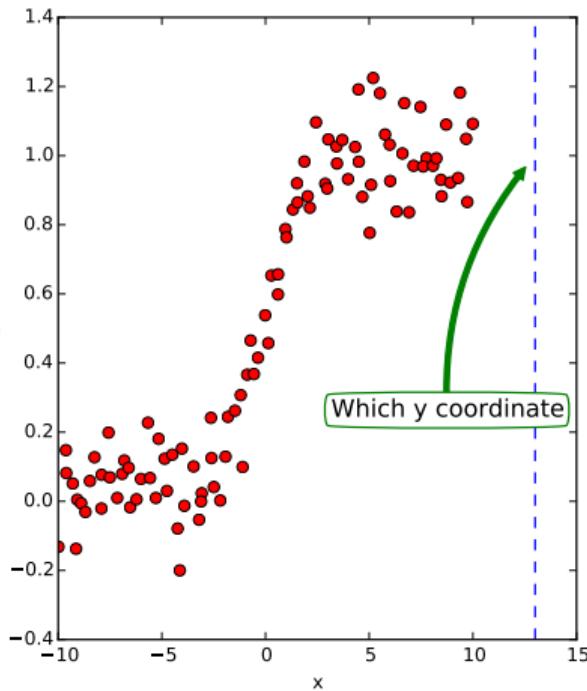
# Classification Example

- For example, consider the 2-dimensional dataset on the right.
- The data consists of 200 blue crosses and 200 red circles.
- Based on these data, what would be a good partition of the 2D space into "red" and "blue" regions?
- What kind of boundaries are allowed between the regions?
  - Straight lines?
  - Continuous curved boundaries?
  - Boundary without any restriction?
- Note: In 2-D this can be solved manually, but not in 4000-D.



# Regression Example

- For example, consider the 1-dimensional data on the right.
- The data consists of 100 data points, where y coordinate is a function of x.
- Based on these data, what would be a good prediction of the target value at  $x = 1.3$  (the dashed line)?
- An obvious solution is to fit a curve into the data points. What kind of forms may the curve have?
  - Straight lines?
  - Continuous curves?

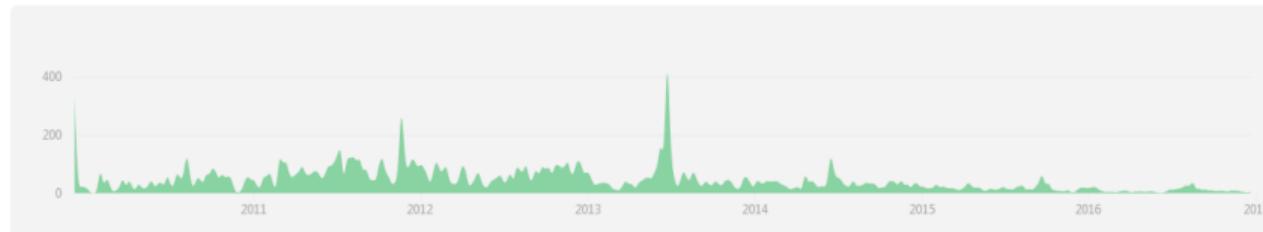


# Different Classifiers

- We will study the following widely used classifiers.
  - Nearest Neighbor classifier
  - Linear classifiers (with linear boundary)
  - The support vector machine (with nonlinear boundary)
  - Ensemble classifiers: Random Forest
  - Black boxes: Neural networks and deep neural networks
- For the first four, we will refer to the *Scikit-Learn* module:  
<http://scikit-learn.org/>.
- The neural network part will be studied with the *Keras* package: <http://keras.io/>.

# Scikit-Learn

- Scikit-Learn started as a Google Summer of Code project.
- The project became a success: There was a clear need for free elegant platform bringing together widely used methods.
- Instead of each contributor providing their own package, scikit-learn has a unified API for each method.
- For details: Pedregosa, *et al.* "Scikit-learn: Machine learning in Python," *The Journal of Machine Learning Research*, 2011.



# Scikit-Learn Methods

Sklearn API is straightforward:

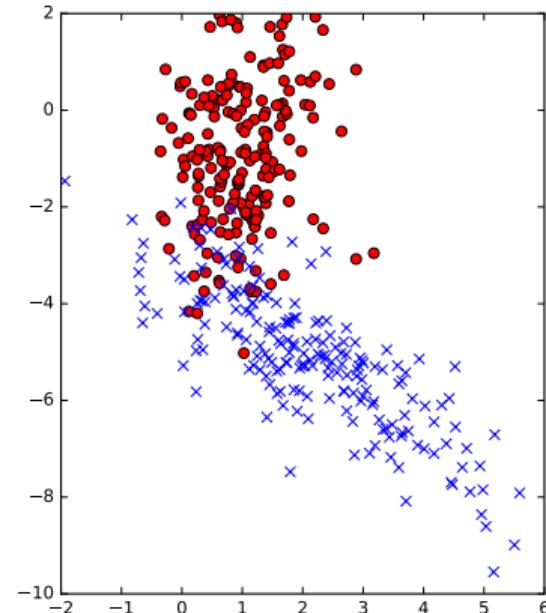
- **Initialization:** Each model has its own constructor, e.g.,  
`model = LogisticRegression(penalty = "l1", C = 0.1).`
- **Training:** Every model implements a `.fit()` method that trains the model; e.g.,  
`model.fit(X_train, y_train)`
- **Prediction:** Every model implements a `.predict()` method that predicts the target output for new inputs; e.g.,  
`y_hat = model.predict(X_test)`
- **Probabilities:** Many models also implement a `.predict_proba()` method that predicts the class probabilities for new inputs; e.g.,  
`p = model.predict_proba(X_test)`

# Sample Scikit-learn Session

- In the example code, X consists of 400 two-dimensional samples from two classes.

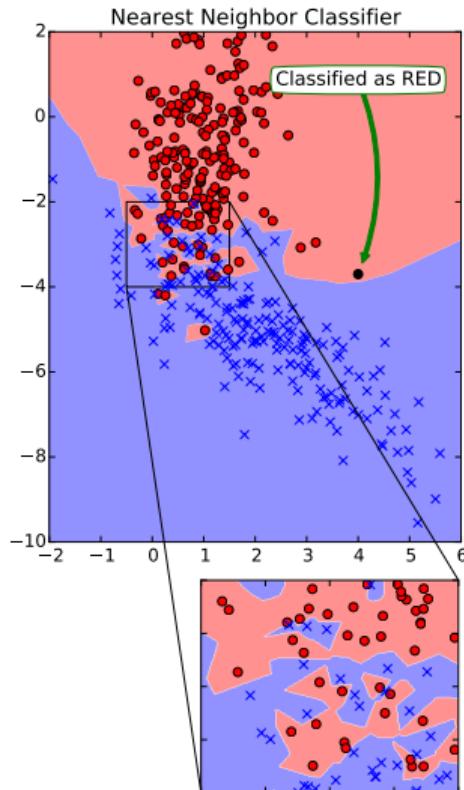
```
# Training code:  
from sklearn.linear_model import LogisticRegression  
  
model = LogisticRegression()  
model.fit(X, y)
```

```
# Testing code:  
>>> model.predict([1,2])  
array([ 1.])  
  
>>> model.predict([-10,-2])  
array([ 0.])  
  
>>> model.predict_proba([[1,2], [-10,-2]])  
Out[35]:  
array([[ 0.0044881 ,  0.9955119 ],  
       [ 0.56097861,  0.43902139]])  
  
# [1,2] is class 1 with 99.5 % confidence.  
# [-10,-2] is class 0 with 56.1 % confidence.
```



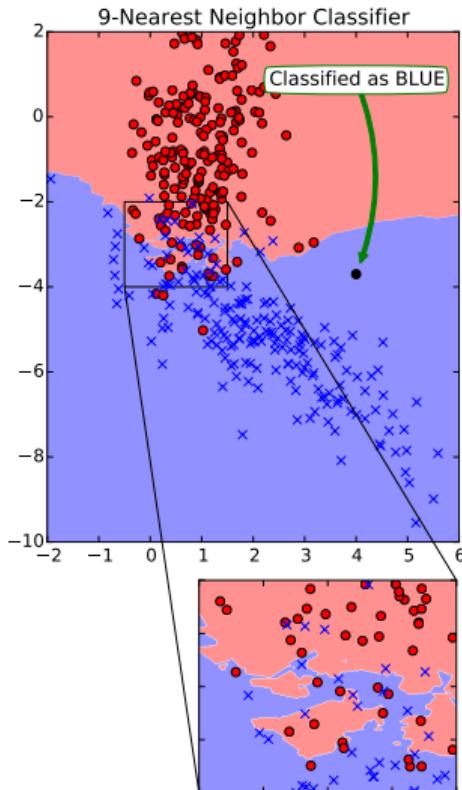
# Nearest Neighbor Classifier

- Probably the most natural approach for deciding the class is simply to see what kind of samples are nearby.
- This is the idea behind the **Nearest neighbor** classifier: Just copy the class label of the most similar training sample to the unknown test sample.



# K-Nearest Neighbor Classifier

- The problem with Nearest Neighbor is its fragility to changes in the training data.
- The classification boundary may change a lot by moving only one training sample.
- The robustness can be increased by taking a majority vote of more nearby samples.
- The **K-Nearest neighbor** classifier selects the most frequent class label among the  $K$  nearest training samples.



# Nearest Neighbor in Scikit-learn

```
# Training code:  
from sklearn.neighbors import KNeighborsClassifier  
  
model = KNeighborsClassifier(n_neighbors = 5, metric = "euclidean")  
model.fit(X, y)
```

```
# Testing code:  
>>> model.predict([-10,-2])  
array([ 0.])  
  
>>> model.predict([1,2])  
array([ 1.])  
  
>>> model.predict_proba([0,-3])  
array([[ 0.6,  0.4]])  
  
# Ask what are the 5 nearest neighbors.  
# Returns the distances and indices (rows) in X  
>>> distances, indices = model.kneighbors([0,-3])  
[0.144,  0.254,  0.362,  0.414,  0.422]  
[379,  11, 215, 370, 198]  
  
# What are the classes for these five:  
>>> y[indices]  
array([[ 0.,  1.,  0.,  0.,  1.]])
```

- Parameters of constructor include:
  - `n_neighbors`: Number of neighbors  $K$
  - `metric`: How distance is calculated
  - `algorithm`: Which algorithm finds the nearest samples; e.g., methods like *K-D Tree* or *brute force*
- Probability prediction counts how many of the nearest  $K$  samples belong to each group.
- Thus, the probabilities are very quantized and often either 0 or 1.

# Benefits of Nearest Neighbor

- Training time is minimal: Either just store the data as is; or reorganize it into a tree structure for efficient search.
- Accuracy is often relatively good.
- Allows complicated definitions of distance, *e.g.*,
  - Find the 5 nearest samples holding attributes A and B but not C.
- New data can be added/deleted without retraining.

# Problems with the Nearest Neighbor

- Nearest neighbor is prone to overlearning: Especially the 1-NN forms local regions around every isolated data point.
- It is highly unlikely that these represent the general trend of the whole population.
- Moreover, the training step extremely fast while the classification step becomes extremely slow (consider training data with billion high-dimensional samples).
- Therefore, more compact representations are preferred: Training time is usually not critical while execution time is.

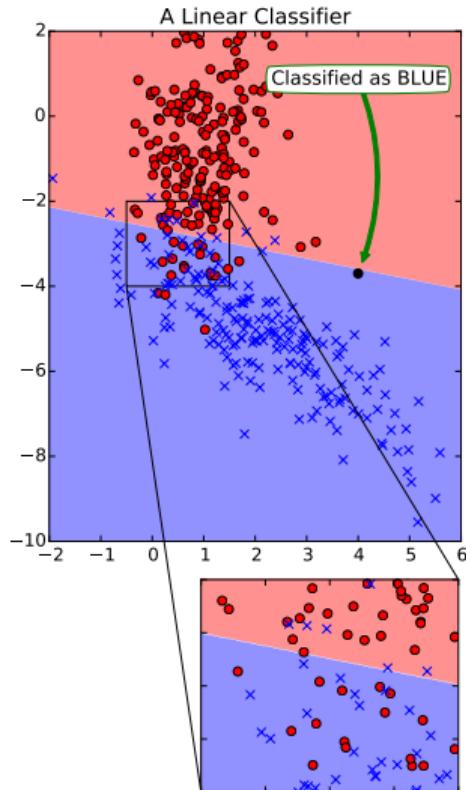
# Linear Classifiers

- A linear classifier learns a linear decision boundary between classes.
- In mathematical terms, the classification rule can be written as

$$F(\mathbf{x}) = \begin{cases} \text{Class 1,} & \text{if } \mathbf{w}^T \mathbf{x} < b \\ \text{Class 2,} & \text{if } \mathbf{w}^T \mathbf{x} \geq b \end{cases}$$

where the weights  $\mathbf{w}$  are learned from the data.

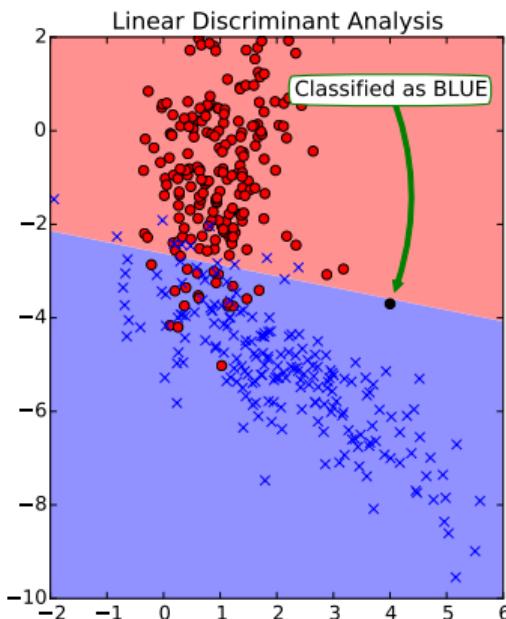
- The expression  $\mathbf{w}^T \mathbf{x} = \sum_k w_k x_k$  essentially transforms the multidimensional data  $\mathbf{x}$  to a real number, which is then compared to a threshold  $b$ .



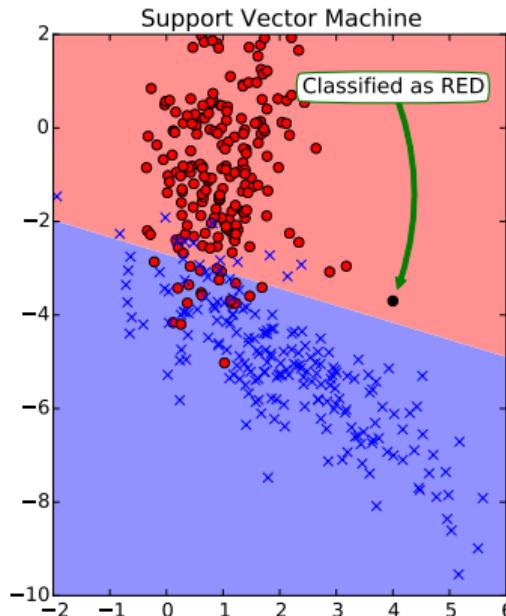
# Flavors of Linear Classifiers

There exists many algorithms for learning the weights  $\mathbf{w}$ , including:

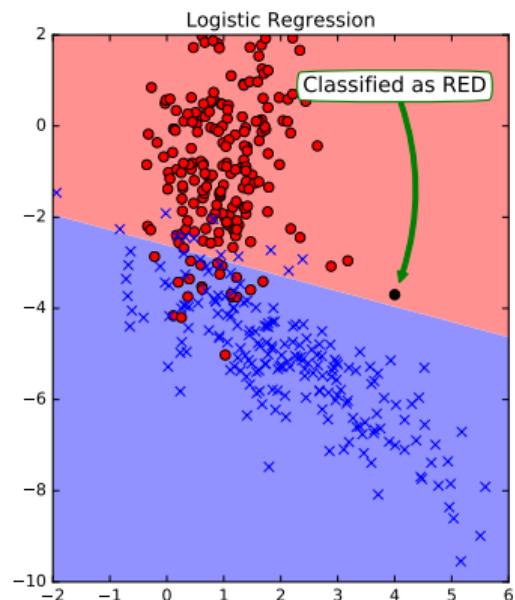
- *Linear Discriminant Analysis (LDA)*



- *Support Vector Machine (SVM)*



- *Logistic Regression (LR)*



# Flavors of Linear Classifiers

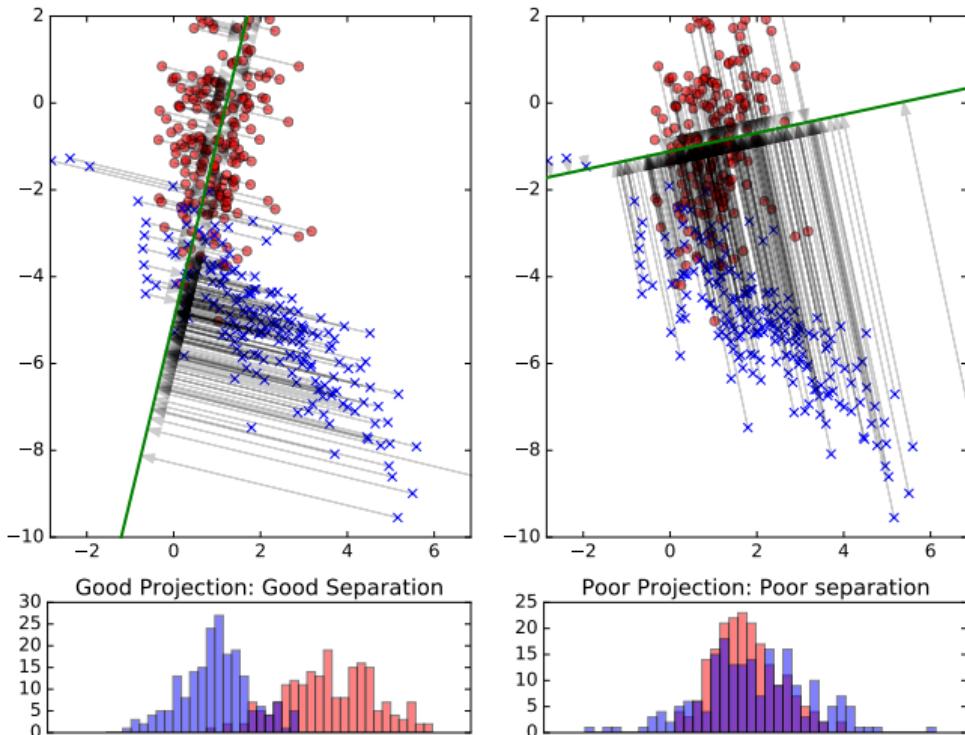
- The LDA:
  - The oldest of the three: Fisher, 1935.
  - "Find the projection that maximizes class separation", *i.e.*, pull the classes as far from each other as possible.
  - Closed form solution, fast to train.
- The SVM:
  - Vapnik and Chervonenkis, 1963.
  - "Maximize the margin between classes".
  - Slowest of the three<sup>1</sup>, performs well with sparse high-dimensional data.
- Logistic Regression (*a.k.a. Generalized Linear Model*):
  - History traces back to 1700's, but proper formulation and efficient training algorithm by Nelder and Wedderburn in 1972.
  - Statistical algorithm: "Maximize the likelihood of the data".
  - Outputs also class probability. Has been extended to automatic feature selection.

---

<sup>1</sup>Slowest in *training*; testing time is the same for all

# Linear Discriminant Analysis

- The LDA maps high dimensional data into a single scalar simultaneously pulling individual classes apart.
- Thus, the task is essentially finding a good linear projection  $\mathbb{R}^N \mapsto \mathbb{R}$ .



# Linear Discriminant Analysis

- A good measure of class separation could be, e.g., the distance between their projected means.
- However, this quantity depends on the scale, and increases simply by multiplying by a large number.
- Thus, we want to pull class means apart while keeping the variance of each class small.
- As a result, we look into the following separability score:

$$\begin{aligned} J(\mathbf{w}) &= \frac{\text{Distance of Class Means}}{\text{Variance of Classes}} \\ &= \frac{(\mathbf{w}^T \boldsymbol{\mu}_1 - \mathbf{w}^T \boldsymbol{\mu}_0)^2}{\mathbf{w}^T \boldsymbol{\Sigma}_1 \mathbf{w} + \mathbf{w}^T \boldsymbol{\Sigma}_0 \mathbf{w}}, \end{aligned}$$

where  $\boldsymbol{\mu}_1$  and  $\boldsymbol{\mu}_0$  are the class means and  $\boldsymbol{\Sigma}_1$  and  $\boldsymbol{\Sigma}_0$  covariance matrices of the two classes.

## Maximization of $J(\mathbf{w})$

- The Fisher separation score can be maximized analytically.
- First simplify  $J(\mathbf{w})$  a bit:

$$J(\mathbf{w}) = \frac{(\mathbf{w}^T \boldsymbol{\mu}_1 - \mathbf{w}^T \boldsymbol{\mu}_0)^2}{\mathbf{w}^T \boldsymbol{\Sigma}_1 \mathbf{w} + \mathbf{w}^T \boldsymbol{\Sigma}_0 \mathbf{w}} = \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

with  $\mathbf{S}_B = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T$  and  $\mathbf{S}_W = \boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_0$ .

- This form is known as *Generalized Rayleigh Quotient*, which appears in many contexts.
- Using Lagrange multipliers, one can show that the maximum must satisfy

$$\mathbf{S}_B \mathbf{w} = \lambda \mathbf{S}_W \mathbf{w}$$

with  $\lambda \in \mathbb{R}$  a constant.

## Maximization of $J(\mathbf{w})$

- The above condition is a *generalized eigenvalue problem*.
- In other words, there are many solutions: one for each eigenvalue.
- Thus, it is straightforward to conclude that the quantity

$$J(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

is maximized when

$$\mathbf{S}_B \mathbf{w} = \lambda \mathbf{S}_W \mathbf{w}$$

or equivalently

$$\mathbf{w}^T \mathbf{S}_B \mathbf{w} = \lambda \mathbf{w}^T \mathbf{S}_W \mathbf{w}$$

- Finally, the ratio  $\mathbf{w}^T \mathbf{S}_B \mathbf{w} / \mathbf{w}^T \mathbf{S}_W \mathbf{w}$  is maximized by choosing  $\lambda$  as the largest eigenvalue (and  $\mathbf{w}$  as the corresponding eigenvector).

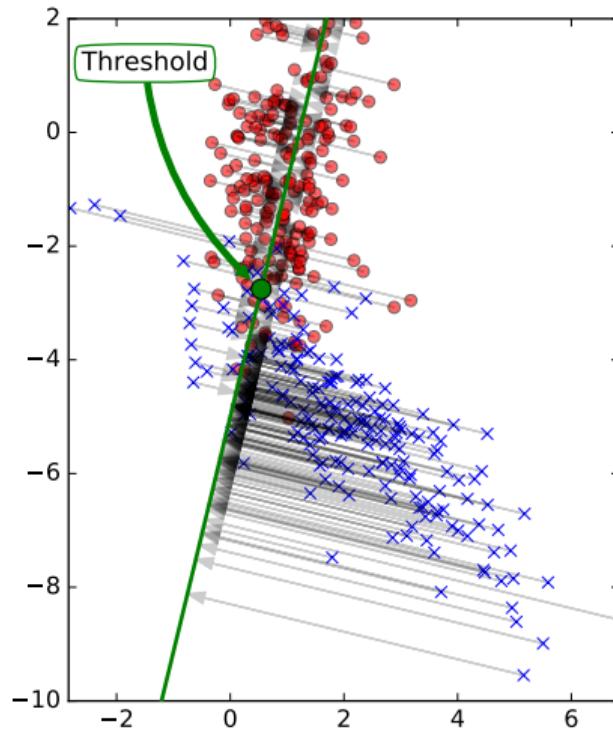
# Solution 1 using Eigenvalues

```
# Training code:  
import numpy as np  
from scipy.linalg import eig  
  
# X0 and X1 contain data of the 2 classes  
m0 = np.mean(X0, axis = 0)  
m1 = np.mean(X1, axis = 0)  
C0 = np.cov(X0 - m0, rowvar = False)  
C1 = np.cov(X1 - m1, rowvar = False)  
  
SB = np.multiply.outer(m1 - m0, m1 - m0)  
SW = C0 + C1  
  
D, V = eig(SB, SW)  
w = V[:, -1]  
  
T = np.mean([np.dot(m1, w), np.dot(m2, w)])
```

```
# Testing code:  
>>> w  
array([ 0.23535937,  0.97190842])  
  
>>> np.dot(w, [0,-2]) - T  
0.606  
# Positive -> in class "red circles"  
  
>>> np.dot(w, [0,-3]) - T  
-0.3657  
# Negative -> in class "blue crosses"
```

- We need to solve the generalized eigenvalue problem  $\mathbf{S}_B \mathbf{w} = \lambda \mathbf{S}_W \mathbf{w}$ .
- scipy has a solver for that.
- `scipy.linalg.eig` returns
  - Array of eigenvalues D in increasing order.
  - Matrix of eigenvectors V (in columns).
- Thus, we want the rightmost ("−1<sup>th</sup>") column V[ :, -1] (corresponding to largest eigenvalue).
- The decision is done by comparing the projected vector to threshold T; the center of projected class means.

# LDA Projection with Threshold



## Solution 2 without Eigenvalues

- It turns out that solving the generalized eigenvalue problem  $\mathbf{S}_B \mathbf{w} = \lambda \mathbf{S}_W \mathbf{w}$  is not necessary.
- The maximum of  $J(\mathbf{w})$  has to satisfy  $\mathbf{S}_B \mathbf{w} = \lambda \mathbf{S}_W \mathbf{w}$ , or equivalently,  $\mathbf{S}_W^{-1} \mathbf{S}_B \mathbf{w} = \lambda \mathbf{w}$ .
- Insert the definition of  $\mathbf{S}_B$  to get:

$$\mathbf{S}_W^{-1}(\mu_1 - \mu_0) \underbrace{(\mu_1 - \mu_0)^T \mathbf{w}}_{\text{scalar } C} = \lambda \mathbf{w}$$

or,

$$\mathbf{S}_W^{-1}(\mu_1 - \mu_0) \frac{C}{\lambda} = \mathbf{w}$$

- Thus, the maximum of  $J(\mathbf{w})$  has to satisfy this condition as well (regardless of  $\lambda$ ).

## Solution 2 without Eigenvalues

- We are looking for a direction vector  $\mathbf{w}$ , so the multiplier  $C/\lambda$  is not relevant either ( $\frac{C}{\lambda}\mathbf{w}$  has the same direction as  $\mathbf{w}$ ).
- Thus,  $\mathbf{w}$  is given by

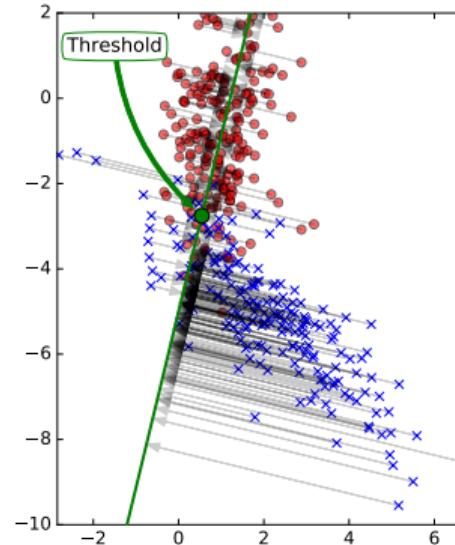
$$\mathbf{w} = \mathbf{S}_W^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) = (\boldsymbol{\Sigma}_0 + \boldsymbol{\Sigma}_1)^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0).$$

# Solution 2 without Eigenvalues

```
# Training code:  
import numpy as np  
from scipy.linalg import eig  
  
# X0 and X1 contain data of the 2 classes  
m0 = np.mean(X0.T, axis = 0)  
m1 = np.mean(X1.T, axis = 0)  
  
C0 = np.cov(X0 - m0, rowvar = False)  
C1 = np.cov(X1 - m1, rowvar = False)  
  
w = np.dot(np.linalg.inv(C0 + C1), (m1 - m0))  
  
T = np.mean([np.dot(m1, w), np.dot(m2, w)])
```

```
# Testing code:  
>>> w  
array([ 0.25237576,  1.04217702])  
  
>>> np.dot(w, [0,-2]) - T  
0.65004  
# Positive -> in class "red circles"  
  
>>> np.dot(w, [0,-3]) - T  
-0.3921  
# Negative -> in class "blue crosses"
```

- The Python code is very similar to the eigenvalue based solution.
- Note that the resulting  $w$  is not exactly the same: The lengths are different, but directions are the same.



# Solution 3: Scikit-Learn

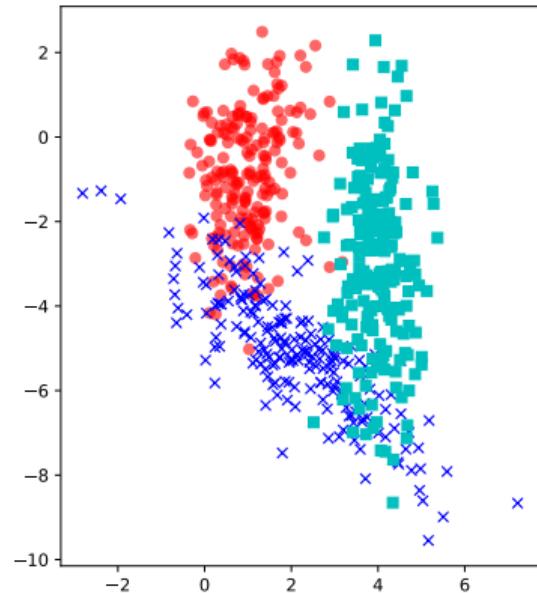
```
# Training code:  
from sklearn.discriminant_analysis import  
    LinearDiscriminantAnalysis  
  
clf = LinearDiscriminantAnalysis()  
  
# X contains all samples, and y their class  
# labels: y = [0,1,1,0,...]  
clf.fit(X, y)
```

```
# Testing code:  
>>> clf.coef_ # This is our w vector  
array([[ 0.50475151,  2.08435403]])  
  
>>> clf.intercept_ # This is the threshold  
array([ 5.46879769])  
  
>>> clf.predict([0, -2])  
array([ 1.])  
  
>>> clf.predict([0, -3])  
array([ 0.])  
  
>>> clf.predict_proba([0, -3])  
array([[ 0.68659846,  0.31340154]])
```

- Scikit-Learn implements LDA in a straightforward manner.
- First construct your classifier using `LinearDiscriminantAnalysis()`
- Then train the model using `fit()`
- Then predict classes using `predict()`
- Or class probabilities using `predict_proba()`

# Multiclass LDA

- The LDA can be generalized to handle more than 2 classes.
- *Multiclass LDA* can be understood in two contexts:
  - As *dimensionality reduction*: We seek for a set of projections, that lower the dimensionality of the data while retaining its original separability.
  - As *classification*: We seek for a set of discriminant functions that give a probability score for each class.
- We are more interested in the latter interpretation.



# Multiclass LDA

- The multiclass LDA is defined by a set of  $K$  linear functions—one per class.

$$g_1(\mathbf{x}) = \mathbf{w}_1^T \mathbf{x} + w_1$$

⋮

$$g_K(\mathbf{x}) = \mathbf{w}_K^T \mathbf{x} + w_K$$

- Then the vector  $\mathbf{x}$  is assigned class  $k = 1, 2, \dots, K$  if

$$g_k(\mathbf{x}) > g_j(\mathbf{x})$$

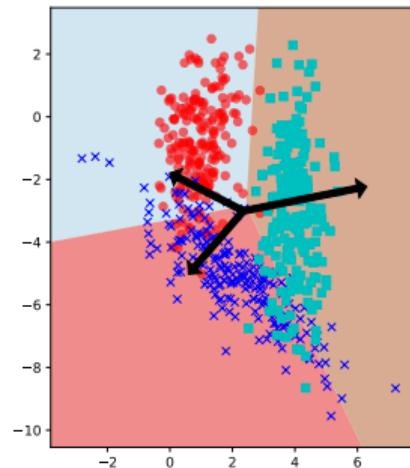
for all  $j \neq k$ .

# Multiclass LDA

```
# Training code:  
import numpy as np  
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis  
  
clf = LinearDiscriminantAnalysis()  
  
# X contains all samples, and y their class  
# labels: y = [1,3,1,2,...]  
clf.fit(X, y)
```

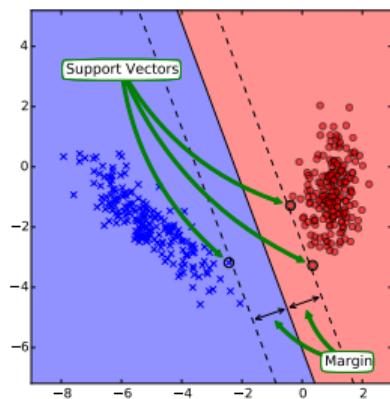
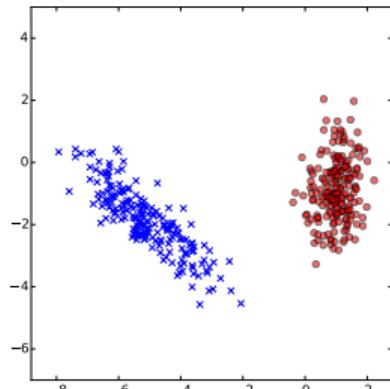
```
# Testing code:  
  
# Show the three discriminants (rows):  
>>> clf.coef_  
array([[-1.00629245,  0.51513354],  
       [-0.74957209, -0.8613368 ],  
       [ 1.75586454,  0.34620326]])  
  
# Compute discriminants for [0,-2]  
>>> scores = np.dot(clf.coef_, [0,-2]) + clf.intercept_  
array([ 0.56090889, -1.20225066, -6.31434368])  
# First class (red) has largest score  
  
# Predict class more easily using predict()  
>>> clf.predict([[0, -2], [0, -4]])  
array([ 1.,  2.])
```

- `LinearDiscriminantAnalysis()` model applies to multiclass case as well.
- The `fit()` method finds the three discriminants also shown in the plot below.



# The Support Vector Machine

- The Support Vector Machine (SVM) is characterized by its *maximum margin property*.
- In other words, it attempts to maximize the margin between classes.
- In the attached example, the binary data is perfectly separable, and the SVM sets the boundary in the middle of the two classes.
- Thus, the boundary location is defined by three samples only; called *support vectors*.



# The Support Vector Machine

- Maximizing the margin  $M$  for the SVM can be characterized as an optimization problem

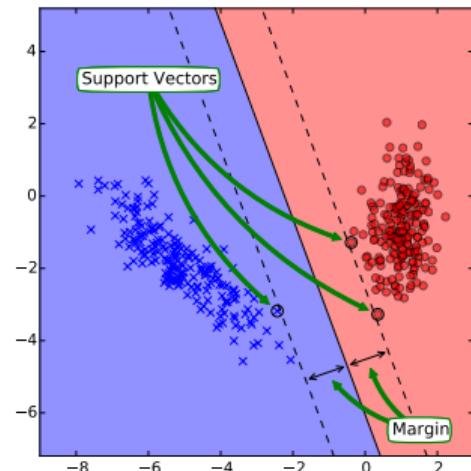
$$\underset{\mathbf{w}, b, \|\mathbf{w}\|=1}{\text{maximize}} \quad M,$$

subject to

$$y_i(\mathbf{x}_i^T \mathbf{w} + b) \geq M, \text{ for } i = 1, 2, \dots, N$$

- Here,  $y_i$  encodes the class as  $y_i \in \{-1, 1\}$ , so the last condition can also be written as

$$\begin{cases} \mathbf{x}_i^T \mathbf{w} + b \geq M, & \text{if } y_i = 1 \\ \mathbf{x}_i^T \mathbf{w} + b \leq -M, & \text{if } y_i = -1 \end{cases}$$



# The Support Vector Machine

- Alternatively, the SVM criterion can be simplified into an equivalent form

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \|\mathbf{w}\|,$$

*subject to*

$$y_i(\mathbf{x}_i^T \mathbf{w} + b) \geq 1, \text{ for } i = 1, 2, \dots, N$$

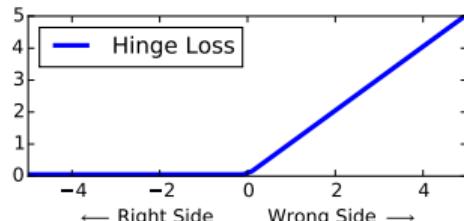
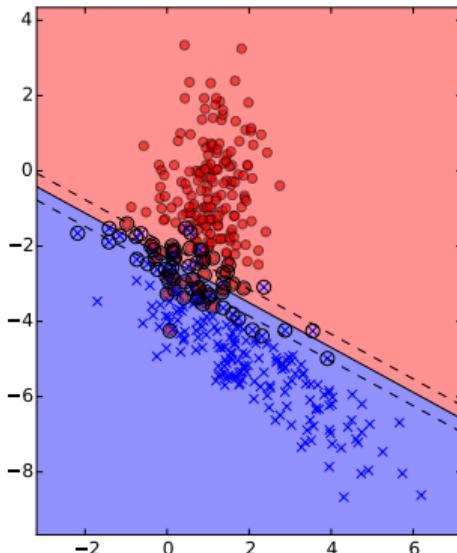
- This is a convex optimization problem with constraints. Efficient tailored algorithms exist.
- The implementation of Scikit-learn is based on the widely used LibSVM library.

# The Support Vector Machine

- The earlier definition of SVM assumes that the classes are separate (and there exists a margin between classes).
- In reality we have to allow some samples to reside on the wrong side of the margin.
- The solution is to penalize for samples on the wrong side.
- The resulting function to be minimized is:

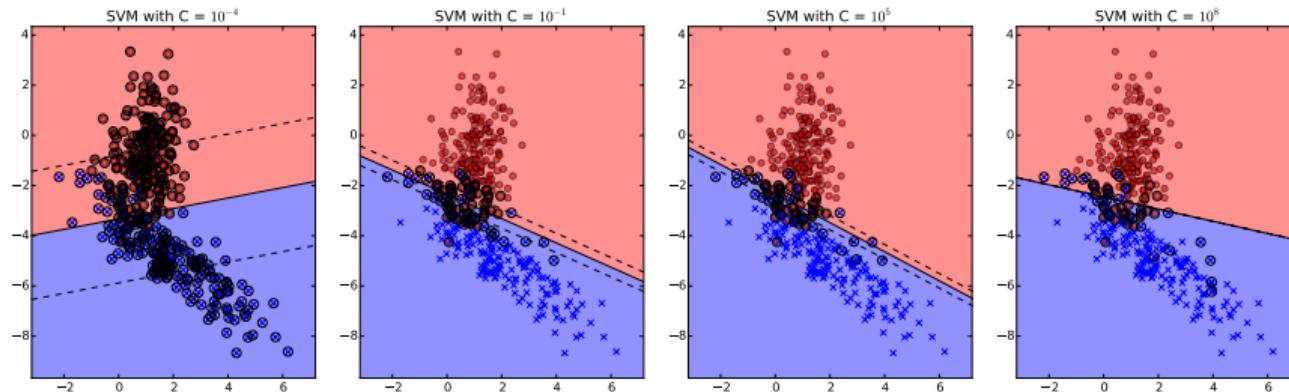
$$\underset{\mathbf{w}, b}{\text{minimize}} \sum_{i=1}^N [\max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b))] + C \|\mathbf{w}\|^2.$$

- For each sample on the wrong side, we add a penalty (called *hinge loss*) defined as  $\max(0, x)$
- Note, that this increases the number of SV's.



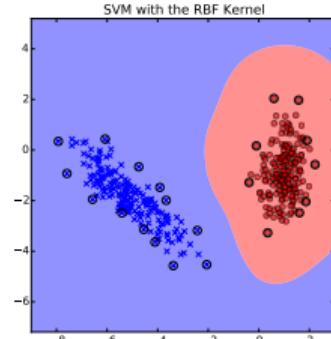
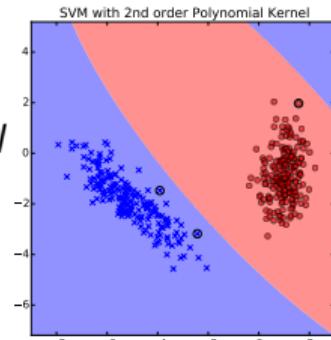
# SVM in Scikit-Learn

- The parameter  $C$  determines the balance between margin width and penalty for being on the wrong side.
- Left to right: Linear SVM's with decreasing penalty.



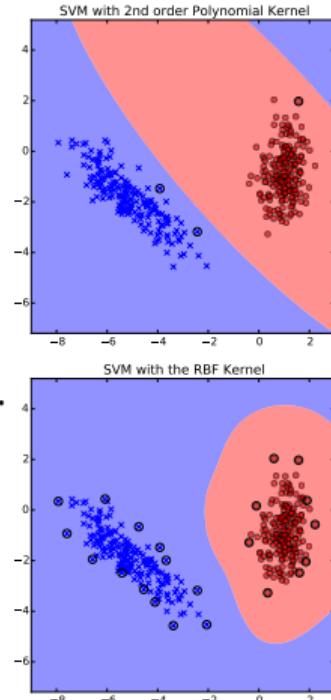
# Kernel Trick

- The SVM can be extended to nonlinear boundaries using the *kernel trick*.
- The kernel trick essentially maps the data into a higher dimension and designs the linear SVM there.
- For example, the top plot can be generated as follows:
  - Map the 2D samples into 3D:  $(x, y) \mapsto (x^2, y^2, \sqrt{2}xy)$
  - Train the SVM with the new 3D samples.
  - The decision boundary is linear in 3D but nonlinear in 2D.
- However, this explicit transformation is slow; the kernel trick does the same thing implicitly.



# Kernel Trick

- It can be shown that substitution of the dot product by some nonlinear function is equivalent to the explicit mapping.
- More specifically, an algorithm can be kernelized by inserting a kernel function  $\kappa(\mathbf{x}, \mathbf{y})$  in place of all dot products  $\mathbf{x} \cdot \mathbf{y}$ .
- It can be shown that under certain conditions on the kernel (e.g., positive semidefiniteness), this is equivalent to an explicit mapping.
- A lot of research has been done on the relation of kernel and the corresponding mapping.
- However, we take a more practical approach and only consider an example.



# Kernel Trick

- As an example, consider what happens when the inner product of 2D vectors  $\mathbf{x} = (x_1, x_2)$  and  $\mathbf{y} = (y_1, y_2)$  is substituted by its second power:

$$\kappa(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^2$$

- We can expand the kernel as follows

$$\begin{aligned}\kappa(\mathbf{x}, \mathbf{y}) &= (\mathbf{x} \cdot \mathbf{y})^2 \\ &= (x_1 y_1 + x_2 y_2)^2 \\ &= (x_1 y_1)^2 + (x_2 y_2)^2 + 2x_1 y_1 x_2 y_2\end{aligned}$$

# Kernel Trick

- The result can be cleverly rearranged to make it look like a dot product in 3D:

$$\begin{aligned}\kappa(\mathbf{x}, \mathbf{y}) &= (x_1 y_1)^2 + (x_2 y_2)^2 + 2x_1 y_1 x_2 y_2 \\&= (x_1 y_1)^2 + (x_2 y_2)^2 + (\sqrt{2}x_1 x_2)(\sqrt{2}y_1 y_2) \\&= (x_1^2, x_2^2, \sqrt{2}x_1 x_2) \cdot (y_1^2, y_2^2, \sqrt{2}y_1 y_2)\end{aligned}$$

# Kernel Trick

- Thus, the following two things are equivalent:
  - **Explicit Mapping:** Transform 2D data into 3D explicitly and fit the SVM with transformed data:

$$\begin{pmatrix} u \\ v \end{pmatrix} \mapsto \begin{pmatrix} u^2 \\ v^2 \\ \sqrt{2}uv \end{pmatrix}$$

- **Implicit Mapping:** Substitute each dot product in the SVM algorithm by the kernel  $\kappa(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^2$  and fit with original 2D data.

# Popular Kernels

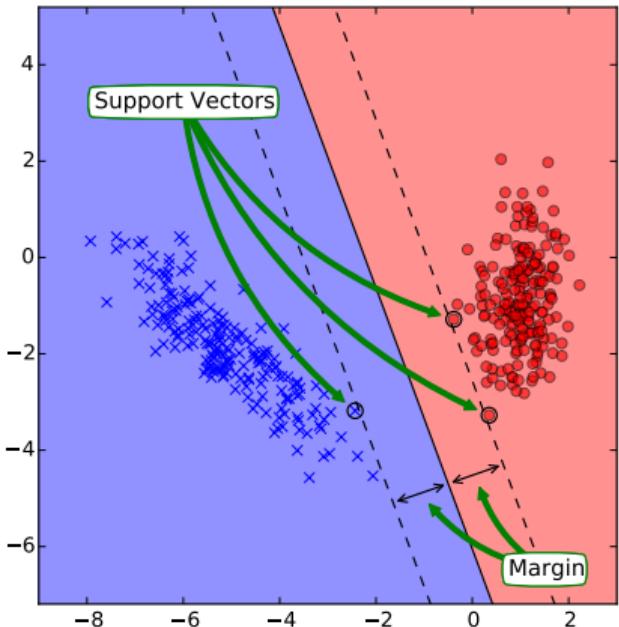
- As mentioned, there is lot of literature on kernels. Popular ones include
  - Linear Kernel:**  $\kappa(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$ . This is the basic SVM with no mapping.
  - Polynomial Kernel:**  $\kappa(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^d$ . Raises dot product to  $d$ 'th power.
  - Inhomogeneous Polynomial Kernel:**  $\kappa(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^d$ . Similar to polynomial kernel, but produces a bit more dimensions.
  - Sigmoid Kernel:**  $\kappa(\mathbf{x}, \mathbf{y}) = \tanh(a\mathbf{x} \cdot \mathbf{y} + b)$ , with  $a > 0$  and  $b < 0$ .
  - Gaussian Kernel:**  $\kappa(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}\right)$ . Probably the most widely used kernel. Also known as **Radial Basis Function (RBF)** kernel.
- The RBF kernel is special in the sense that it corresponds to a mapping into an **infinite dimensional** vector space.
- In addition to the mathematical excitement, the RBF kernel is often the best performing one, as well.

# SVM in Scikit-Learn

```
# Training code:  
from sklearn.svm import SVC  
# Alternatively: from sklearn.svm import LinearSVC  
  
# Kernels include: "linear", "rbf", "poly", "sigmoid"  
# C is the penalty term (default = 1)  
clf = SVC(kernel = 'linear', C = 1)  
  
# X contains all samples, and y their class  
# labels: y = [0,1,1,0,...]  
clf.fit(X, y)
```

```
# Testing code:  
>>> clf.coef_ # This is our w vector (linear case)  
array([[ 0.72817808,  0.26866005]])  
  
>>> clf.intercept_ # This is the threshold  
array([ 3.31965658])  
  
>>> clf.predict([-4,-2], [-2,0])  
array([ 0.,  1.])  
  
>>> clf.support_vectors_  
array([[-2.43263229, -3.18119051],  
       [ 0.34905559, -3.27417889],  
       [-0.38544996, -1.28336992]])
```

- Scikit-Learn wraps the LibSVM and LibLinear libraries (latter is linear kernel optimized).



# Multiclass SVM

- SVM is inherently two-class.
- Generalization to many classes is done by comparing pairs of classes.
- For each class, we train a SVM that classifies this class vs. all others.
- Scikit-Learn provides a few *meta-classifiers* for this purpose.
  - `multiclass.OneVsRestClassifier` compares each class vs. all others (requires  $K$  classifiers)
  - `multiclass.OneVsOneClassifier` compares all class pairs (requires  $K(K - 1)/2$  classifiers)

# Multiclass SVM

- A further extension is *multilabel* classification, where several classes can be present simultaneously.
- Targets are presented as binary indicators:

```
>>> from sklearn.preprocessing import MultiLabelBinarizer  
>>> y = [[2, 3, 4], [2], [0, 1, 3], [0, 1, 2, 3, 4], [0, 1, 2]]  
>>> MultiLabelBinarizer().fit_transform(y)  
array([[0, 0, 1, 1, 1], # Classes 2,3,4 are "on"  
       [0, 0, 1, 0, 0], # Class 3 is "on"  
       [1, 1, 0, 1, 0], # ...  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 0, 0]])
```

- Occurs often in, e.g., image recognition problems: "What is shown in this picture."
- For example, the following attributes are "on" in the attached picture: {"fly", "flower", "summer"}

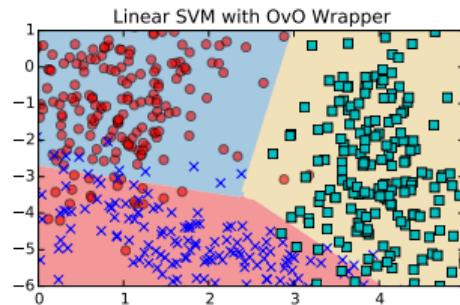
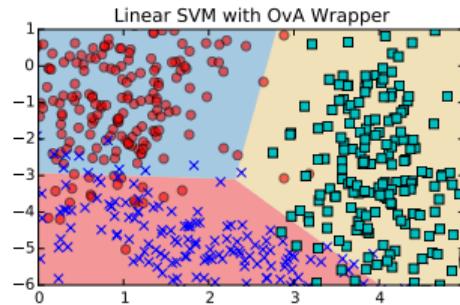


# Multiclass SVM

- SVC() in fact implements OvO heuristic inherently.
- LinearSVC() does not, so let's use that as our example.

```
# Training code:  
from sklearn.svm import LinearSVC  
from sklearn.multiclass import OneVsRestClassifier  
from sklearn.multiclass import OneVsOneClassifier  
  
clf_ova = OneVsRestClassifier(LinearSVC())  
clf_ova.fit(X, y)  
  
clf_ovo = OneVsOneClassifier(LinearSVC())  
clf_ovo.fit(X, y)
```

```
# Testing code:  
>>> clf_ova.predict(np.array([[2,-3.2], [4,-2]]))  
array([ 2.,  3.])  
  
>>> clf_ovo.predict(np.array([[2,-3.2], [4,-2]]))  
array([ 1.,  3.])  
  
>>> len(clf_ova.estimators_)  
3
```

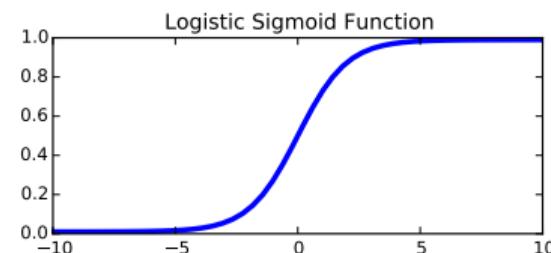


# Logistic Regression

- The third member of linear classifier family is *Logistic Regression* (LR).
- Unlike the other two, LR is *probabilistic*, i.e., it models the class probabilities instead of plain class memberships.
- For two-class case ( $c \in \{0, 1\}$ ), the model is:

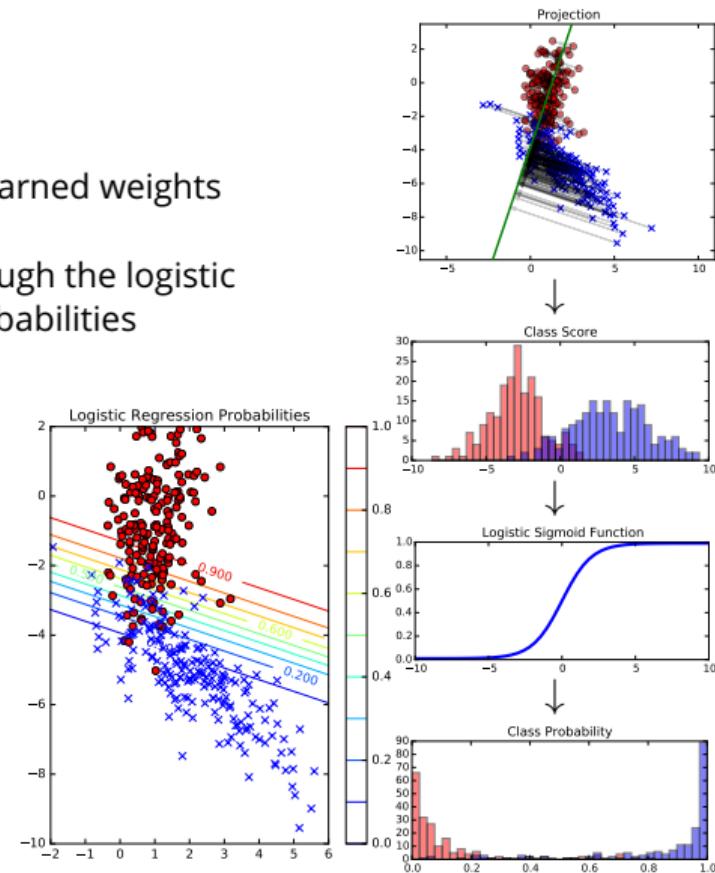
$$p(c = 1 \mid \mathbf{x}) = \frac{1}{1 + \exp[-(\mathbf{w}^T \mathbf{x} + b)]}.$$

- Also:  $p(c = 0 \mid \mathbf{x}) = 1 - p(c = 1 \mid \mathbf{x})$ .
- In essence, the model maps the projection  $\mathbf{w}^T \mathbf{x} + b$  through the sigmoid function (thus limiting the range to  $[0, 1]$ ).



# Logistic Regression

- The model is illustrated in the flowgraph:
  - The samples are projected to 1D using the learned weights  $\mathbf{w}$  and  $b$ .
  - The resulting "class scores" are mapped through the logistic function, which transforms the scores to probabilities  $p(c = 1 | \mathbf{x}) \in [0, 1]$ .
- The probability estimates can also be mapped back to 2D, as shown in the figure on the right.



# Training Logistic Regression Models

- Logistic regression is trained by *maximum likelihood*.
- In other words, we maximize the likelihood of observing these data points with respect to model parameters.
- The likelihood of samples  $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{N-1}]$  with class labels  $y_0, y_1, \dots, y_{N-1} \in \{1, 2, \dots, K\}$  to have occurred from the model with parameters  $\theta$  is

$$p(\mathbf{X} | \theta) = \prod_{n=0}^{N-1} p(\mathbf{x}_n; \theta, y_n).$$

- As usual, we consider the log-likelihood instead:

$$\ln p(\mathbf{X} | \theta) = \sum_{n=0}^{N-1} \ln p(\mathbf{x}_n; \theta, y_n).$$

# Training Logistic Regression Models

- Due to simplicity, we will continue with the two-class case only.
- Let's label the classes as  $y_k \in \{-1, 1\}$ , because this will simplify the notation later.
- Also: let's hide the constant term  $b$  by catenating it at the end of  $\mathbf{w}$ :

$$\mathbf{w} \leftarrow [\mathbf{w}, b] \quad \text{and} \quad \mathbf{x} \leftarrow [\mathbf{x}, 1]$$

# Training Logistic Regression Models

- The likelihood for classes  $-1$  and  $1$  are now

$$\begin{aligned} p(\mathbf{x}_n \mid y_n = 1) &= \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_n)} \\ p(\mathbf{x}_n \mid y_n = -1) &= 1 - \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_n)} \\ &= \frac{(1 + \exp(-\mathbf{w}^T \mathbf{x}_n)) - 1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_n)} \\ &= \frac{\exp(-\mathbf{w}^T \mathbf{x}_n)}{1 + \exp(-\mathbf{w}^T \mathbf{x}_n)} \\ &= \frac{1}{\exp(\mathbf{w}^T \mathbf{x}_n) + 1} = \frac{1}{1 + \exp(\mathbf{w}^T \mathbf{x}_n)} \end{aligned}$$

# Training Logistic Regression Models

- Now  $p(\mathbf{x}_n | y_n = 1)$  and  $p(\mathbf{x}_n | y_n = -1)$  are almost the same form and can be conveniently combined into one formula:

$$p(\mathbf{x}_n | y_n) = \frac{1}{1 + \exp(-y_n \mathbf{w}^T \mathbf{x}_n)}$$

- The likelihood for all samples  $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{N-1}]$  is now

$$p(\mathbf{X} | \mathbf{w}, \mathbf{y}) = \prod_{n=0}^{N-1} \frac{1}{1 + \exp(-y_n \mathbf{w}^T \mathbf{x}_n)}$$

# Training Logistic Regression Models

- The log-likelihood becomes

$$\begin{aligned}\ln p(\mathbf{X} \mid \mathbf{w}, \mathbf{y}) &= \sum_{n=0}^{N-1} \ln 1 - \ln(1 + \exp(-y_n \mathbf{w}^T \mathbf{x}_n)) \\ &= - \sum_{n=0}^{N-1} \ln(1 + \exp(-y_n \mathbf{w}^T \mathbf{x}_n)).\end{aligned}$$

# Training Logistic Regression Models

- In order to maximize the likelihood, we can equivalently minimize the *logistic loss function*:

$$\ell(\mathbf{w}) = \sum_{n=0}^{N-1} \ln(1 + \exp(-y_n \mathbf{w}^T \mathbf{x}_n)).$$

- There exists several algorithms for minimization of logistic loss, e.g.,
  - Iterative Reweighted Least Squares (**IRLS**) is an algorithm specifically tailored for this kind of problems. Used by `statsmodels.api.Logit` and `statsmodels.api.MNLogit`.
  - Optimization theory based approaches. Since  $\ln p(\mathbf{X} | \boldsymbol{\theta})$  is convex, in principle any optimization approach can be used. `scikit-learn` uses something called *Trust Region Newton Algorithm*.

# Training Logistic Regression Models

- The optimization theory approach has become dominant, for two reasons:
  - ① The SVM training can also be posed in the same framework: minimize the *hinge loss*:

$$\text{hinge-loss} = \max(0, 1 - y_n \mathbf{w}^T \mathbf{x}_n).$$

- ② A general purpose optimizer can minimize modified losses, as well. Most importantly, one can add a penalty term into the loss function; *e.g.*,

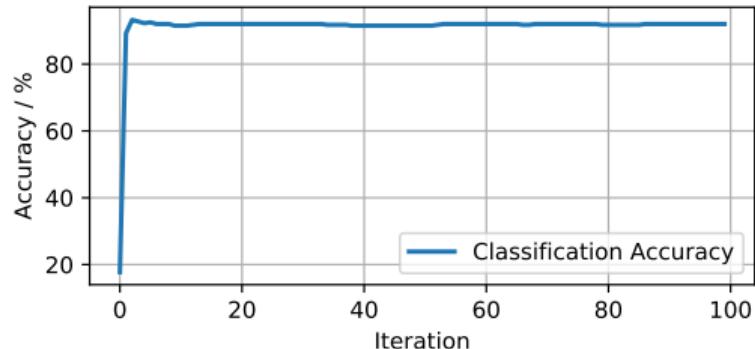
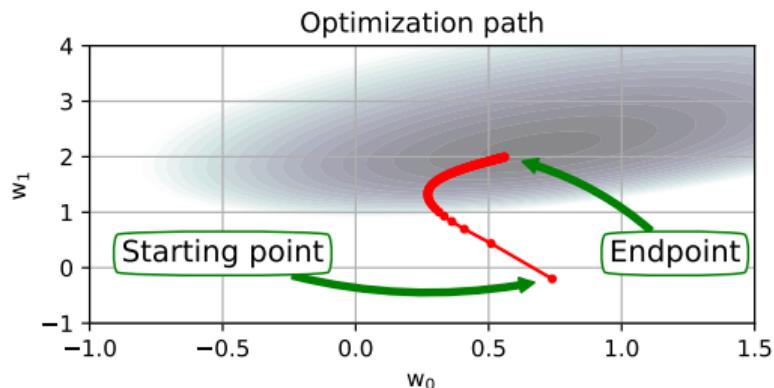
$$\text{penalized log-loss} = \sum_{n=0}^{N-1} \ln(1 + \exp(-y_n \mathbf{w}^T \mathbf{x}_n)) + C \mathbf{w}^T \mathbf{w}$$

where the latter term favors small coefficients in  $\mathbf{w}$ . We will return to this technique called *regularization* later.

# Training Logistic Regression Models

- In the exercises, we will implement log loss minimization for two dimensional data.
  - 1 Initialize  $\mathbf{w}$  at random.
  - 2 Adjust  $\mathbf{w}$  towards the negative gradient:
- The result is shown on the right.

$$\mathbf{w} = \mathbf{w} - \epsilon \cdot \ell'(\mathbf{w})$$



# Example: Effect of Regularization Parameter C

```
from sklearn.cross_validation import train_test_split

# Split data to training and testing
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size = 0.2)

# Test C values -4, -3, ..., 2
C_range = 10.0 ** np.arange(-4, 3)
clf = LogisticRegression()

for C in C_range:

    clf.C = C
    clf.fit(X_train, y_train)
    y_hat = clf.predict(X_test)
    accuracy = 100.0 * np.mean(y_hat == y_test)

    print ("Accuracy for C = %.2e is %.1f %% (||w|| = %.4f)" % \
        (C, accuracy, np.linalg.norm(clf.coef_)))
```

# Code output for the 3-class example data:

```
Accuracy for C = 1.00e-04 is 50.8 % (||w|| = 0.0660)
Accuracy for C = 1.00e-03 is 59.2 % (||w|| = 0.3332)
Accuracy for C = 1.00e-02 is 73.3 % (||w|| = 1.0372)
Accuracy for C = 1.00e-01 is 88.3 % (||w|| = 2.0715)
Accuracy for C = 1.00e+00 is 90.8 % (||w|| = 3.2153)
Accuracy for C = 1.00e+01 is 91.7 % (||w|| = 3.7732)
Accuracy for C = 1.00e+02 is 91.7 % (||w|| = 3.8704)
```

- In this example, we train the LR classifier with a range of values for parameter C.
- Each C value can be set inside a for loop as  
`clf.C = <new value>`
- We test the accuracy on a separate test data extracted from the complete data set.

# Multiclass Logistic Regression

There are two ways to extend LR to multiclass.

- ① A straightforward way would just normalize the exponential terms to sum up to 1:

$$p(c = k \mid \mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x} + b_k)}{\sum_{j=1}^K \exp(\mathbf{w}_j^T \mathbf{x} + b_j)}$$

However, one of the terms is unnecessary, because the probabilities add up to 1.

- ② Thus, we get an alternative model with fewer parameters:

$$p(c = k \mid \mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x} + b_k)}{1 + \sum_{j=1}^{K-1} \exp(\mathbf{w}_j^T \mathbf{x} + b_j)}, \quad k = 1, 2, \dots, K-1$$

$$p(c = K \mid \mathbf{x}) = \frac{1}{1 + \sum_{j=1}^{K-1} \exp(\mathbf{w}_j^T \mathbf{x} + b_j)}.$$

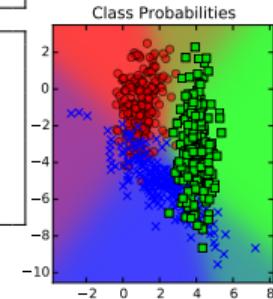
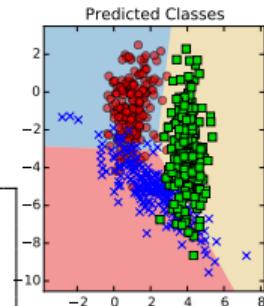
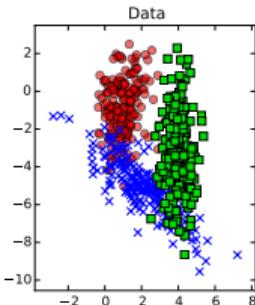
# Multiclass Version ①

- The first approach designs one probability model  $p(c = k | \mathbf{x})$  for each class.
- However, the model is ill-posed: there are infinitely many solutions (by scaling).
- Thus, the model needs to be regularized to create a unique solution.
- This is the model given by scikit-learn.

```
# Training code:  
from sklearn.linear_model import LogisticRegression  
  
clf = LogisticRegression()  
clf.fit(X, y)
```

```
# Test code:  
>>> clf.predict([[1, -1], [4, -6]])  
array([ 1.,  2.])  
  
clf.predict_proba([[1, -1], [4, -6]])  
array([[0.927, 0.038, 0.035],  
       [0.001, 0.501, 0.498]])
```

```
# Model parameters:  
>>> clf.coef_  
array([[-1.23804226,  0.98973963],  
      [-1.37272021, -1.53820378],  
      [ 2.13288304,  0.54858162]])  
>>> clf.intercept_  
array([ 3.79924145, -3.50411623, -5.02047854])
```



# Multiclass Version ②

- The second version is historically older.
- Has a unique solution even without regularization.
- Is not implemented in `scikit-learn`, but can be found in `statsmodels` library.

```
# Training code:  
from statsmodels.api import MNLogit  
  
# statsmodel has a bit different API.  
# In reality, X should zero-mean, but  
# let's simplify the code a bit here.  
clf = MNLogit(y, X)  
clf = clf.fit()
```

```
# Test code:  
>>> clf.predict([[1,-1], [4, -6]])  
array([0.972, 0.019, 0.009],  
     [0.001, 0.471, 0.528]),  
  
# clf.predict() gives probabilities.  
# To get classes, we find the max:  
>>> clf.predict([[1,-1], [4, -6]]).argmax(axis = 1)  
array([0, 2])
```

```
# Model parameters:  
  
>>> clf.params  
array([-0.06122776,  2.82418605],  
      [-2.00590479, -0.4564178 ])  
  
# Note: only two sets of weights.  
# Note2: no intercept; assumes zero mean
```

