Chapter 3

A Tour of Machine Learning Classifiers Using Scikit-learn

September 16, 2021

Choosing a classification algorithm

- No classifier works best across all scenarios ("no free lunch" theorem)
- Always need to consider the specifics of the problem
- Solving a problem within supervised ML framework:
  1. Select features
  2. Choose performance metrics
  3. Choose classifier and optimization algorithm
  4. Evaluate performance of the model
  5. Tune the classifier

Perceptron implementation

- iPython notebook on github

Modeling class probabilities

- What happens if the classes are not linearly separable?
- Weights never stop updating as long as there is at least one misclassified example in each epoch
- Logistic regression is a better option
- Note that despite the name this is a classification model

Logistic regression model

- This is a "go to" model for classification
- Designed for binary classification but can be extended to multiclass
- Odds ratio

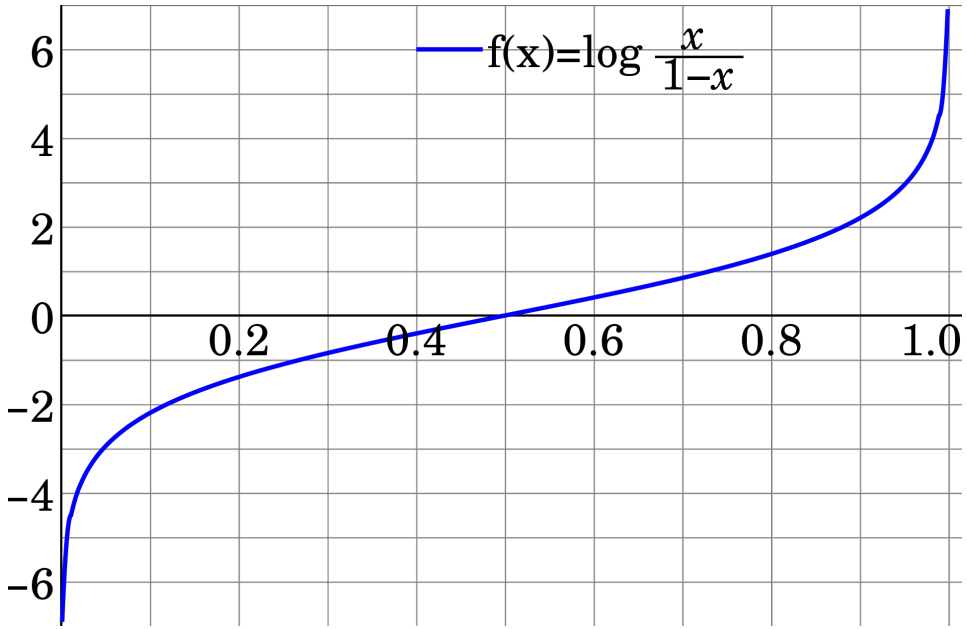
$$\frac{p}{(1-p)}$$

  Where p is the probability of the positive class (class label $y = 1$). E.g. the probability that a patient has a certain disease.

- Logit function

$$logit(p) = \log \frac{p}{1-p}$$

Logit function



- Mathematically, the logit function is the inverse of the logistic function
- An inverse function is a function that 'reverses' another function
- I.e. if the function f applied to an input x gives a result of y, then applying its inverse function g to y gives the result x

Modeling logit function

- We model the logit function as a linear combination of features (dot product of feature values and weights)

$$logit(p(y=1|\mathbf{x})) = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \sum_{i=0}^{m} w_ix_i = \mathbf{w}^T\mathbf{x}.$$

  Where $p(y=1|\mathbf{x})$ s the conditional probability that a particular sample belongs to class 1 given its features $\mathbf{x}$

- This is equivalent to expressing p as

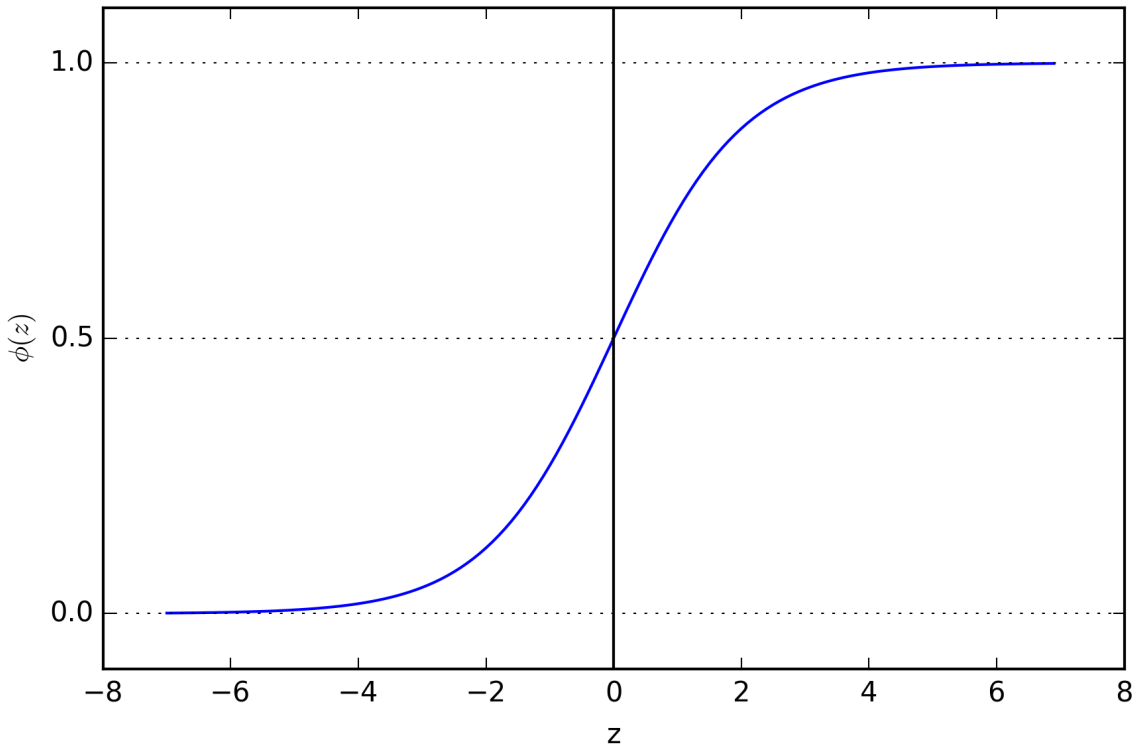
$$p(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$

Logistic Sigmoid

- Logistic function (aka sigmoid function)

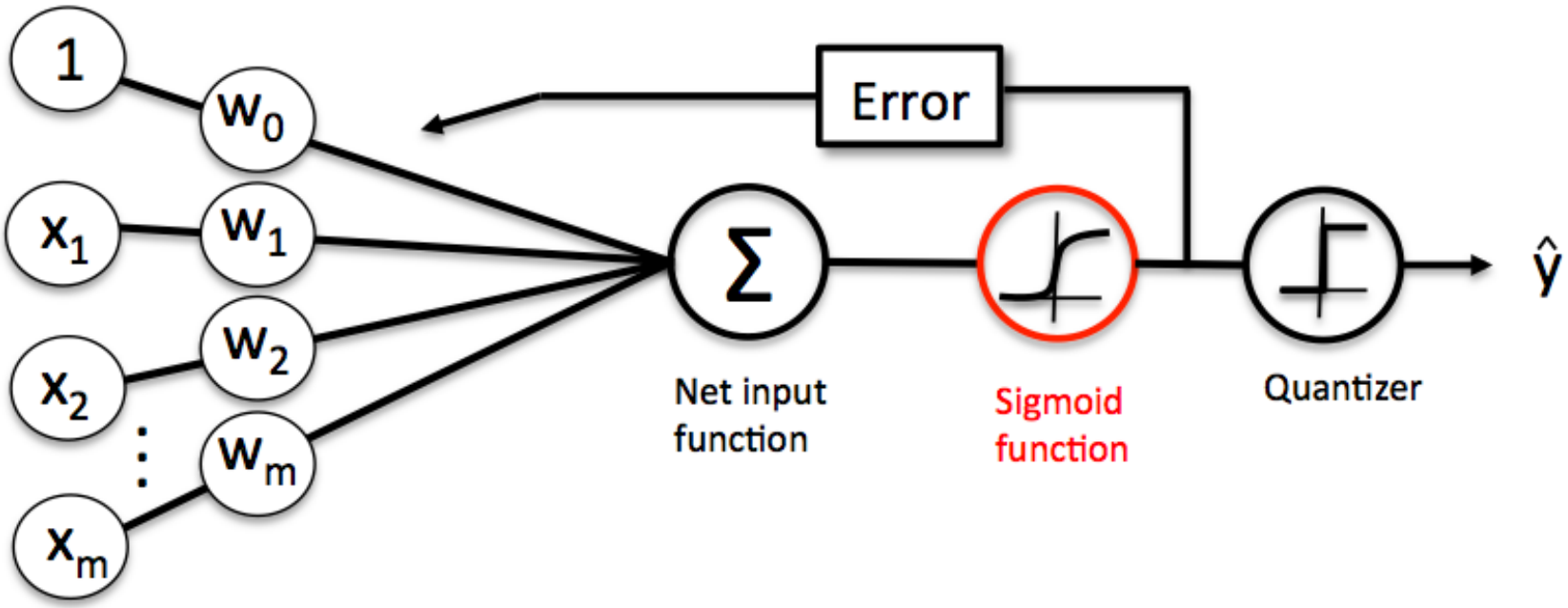
$$\phi(z) = \frac{1}{1 + e^{-z}}.$$

- S-shaped curve



Relationship with Adaline

- In Adaline, we used the identify function as the activation function
- In logistic regression, we instead use the sigmoid function



Probability distribution over classes

- Output of the sigmoid often interpreted as probability
- E.g. $P(y = 1|\mathbf{x}; \mathbf{w}) = 0.8$
- Probability can be converted to a binary outcome (quantizer)

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

- Which is equivalent to the following

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

- For many applications (e.g. weather forecasting), we want the probability

Learning the weights

- Previously we minimized the sum-squared-error cost function

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left(\phi\left(z^{(i)}\right) - y^{(i)}\right)^2$$

- Now we need to derive the cost function for logistic regression
- Define the likelihood L

$$L(\mathbf{w}) = P(\mathbf{y}|\mathbf{x};\mathbf{w}) = \prod_{i=1}^{n} P\left(y^{(i)}|x^{(i)};\mathbf{w}\right)$$

$$L(\mathbf{w}) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

Log-likelihood function

- Maximize the likelihood function

$$L(\mathbf{w}) = P(\mathbf{y}|\mathbf{x};\mathbf{w})$$

$$L(\mathbf{w}) = \prod_{i=1}^{n} P\left(y^{(i)}|x^{(i)};\mathbf{w}\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1-\phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

- In practice easier to deal with the natural log of this equation

$$l(\mathbf{w}) = \log L(\mathbf{w})$$

$$l(\mathbf{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1-y^{(i)}\right) \log\left(1-\phi\left(z^{i()}\right)\right) \right]$$

- Easier to take derivative + fewer numerical underflow issues

Cost function

- Rewrite likelihood as a cost function

$$J(\mathbf{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log \left(\phi\left(z^{(i)}\right) \right) - \left(1 - y^{(i)}\right) \log \left(1 - \phi\left(z^{i()}\right) \right) \right]$$

- Can now be minimized using gradient descent

▸ iPython notebook on github

Weight update derivation

Calculate the partial derivative of the log-likelihood function with respect to the jth weight:

$$\frac{\partial}{\partial w_j} l(\mathbf{w}) = \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

Partial derivative of the sigmoid function:

$$\frac{\partial}{\partial z} \phi(z) = \frac{\partial}{\partial z} \frac{1}{1+e^{-1}} = \frac{1}{\left(1+e^{-z}\right)^2} e^{-z} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}} \right)$$

$$= \phi(z)(1-\phi(z)).$$

Weight update derivation

Resubstitute $\frac{\partial}{\partial z}\phi(z) = \phi(z)(1-\phi(z))$ to obtain:

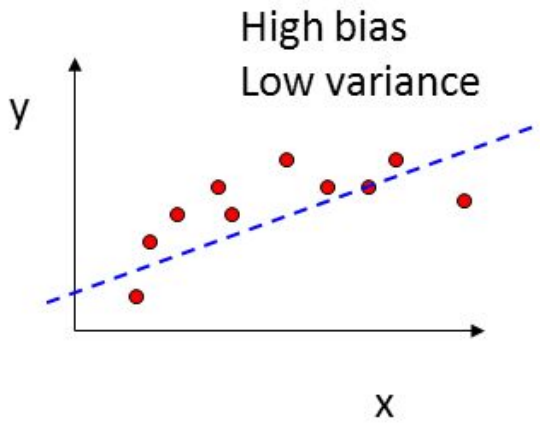
$$\begin{aligned}
&\left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial}{\partial w_j}\phi(z) \\
&= \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\phi(z)\big(1-\phi(z)\big)\frac{\partial}{\partial w_j}z \\
&= \Big(y\big(1-\phi(z)\big) - (1-y)\phi(z)\Big)x_j \\
&= \big(y-\phi(z)\big)x_j
\end{aligned}$$

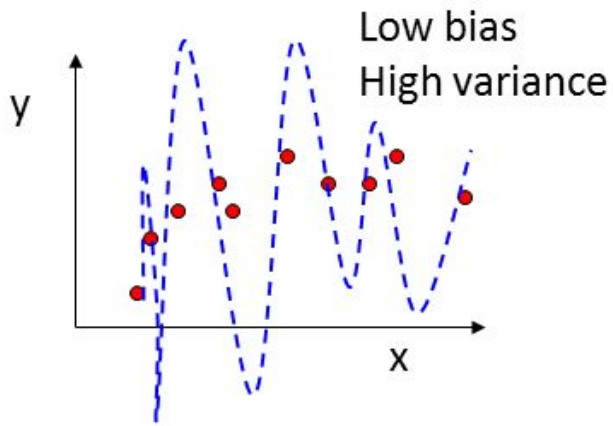
Overfitting

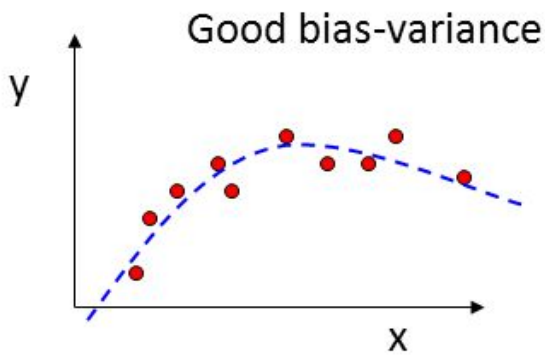
- Sometimes model performs well on training data but does not generalize well to unseen data (test data)
- This is overfitting
- If a model suffers from overfitting, the model has a high variance
- This is often caused by a model that's too complex
- Underfitting can also occur (high bias)
- Underfitting is caused by a model's not being complex enough
- Both suffer from low performance on unseen data

Bias-variance tradeoff

Bias-Variance Examples



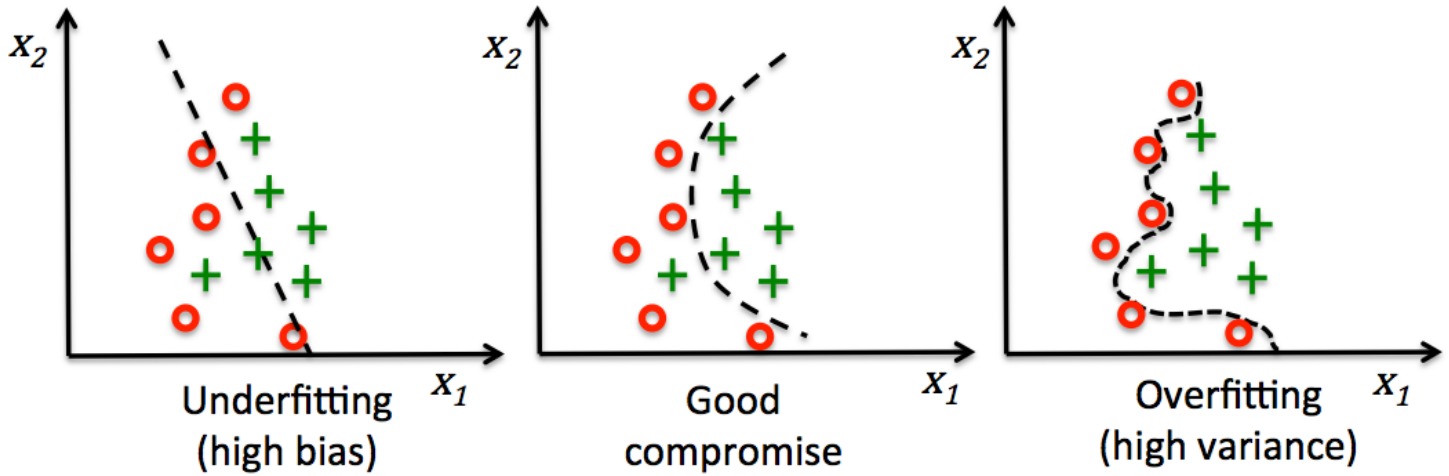




Here the red circles represent training data and the blue curves are models fitted to the data

Regularization

Regularization: adding information in order to solve an ill-posed problem or to prevent overfitting. An example of an ill-posed problem is a problem where the solution does not exist or is not unique.



- Regularization is a way to tune the complexity of the model
- Regularization helps to filter out noise from training data
- As a result, regularization prevents overfitting

L2 regularization

The most common form of regularization is the so-called L2 regularization (sometimes also called L2 shrinkage or weight decay):

$$\frac{\lambda}{2}\|\mathbf{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

Where λ is the so-called regularization parameter. To apply regularization, we add the regularization term to the cost function, which shrinks the weights (adding a zero mean gaussian prior over the weights):

$$J(\mathbf{w}) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi(z^{(i)})\right) - \left(1-y^{(i)}\right)\log\left(1-\phi(z^{(i)})\right)\right] + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

Weight decay

Note that the derivative of w^2 with respect to w is $2w$, so:

$$w_i = w_i - \eta \frac{\partial J}{\partial w_i} - \eta \lambda w_i$$

The new term $(\eta \lambda w_i)$ coming from the regularization causes the weight to decay in proportion to its size.

Regularization parameter

- We control how well we fit the training data via the regularization parameter λ
- By increasing λ, we increase the strength of regularization
- Sometimes (e.g in scikit-learn), SVM terminology is used

$$C = \frac{1}{\lambda}$$

- I.e we rewrite the regularized cost function of logistic regression:

$$C\left[\sum_{i=1}^{n}\left(-y^{(i)}\log\left(\phi(z^{(i)}\right)-\left(1-y^{(i)}\right)\right)\log\left(1-\phi(z^{(i)})\right)\right]+\frac{1}{2}\|\mathbf{w}\|^2$$

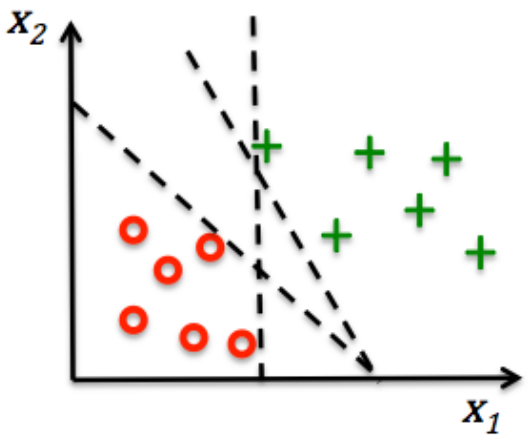
Regularization illustrated

- Decreasing the value of C means increasing the regularization strength
- Can be visualized by plotting L2 regularization path for two weights
- Display weights across multiple C values
- As you see, weights shrink to zero as C decreased
- ▸ iPython notebook on github

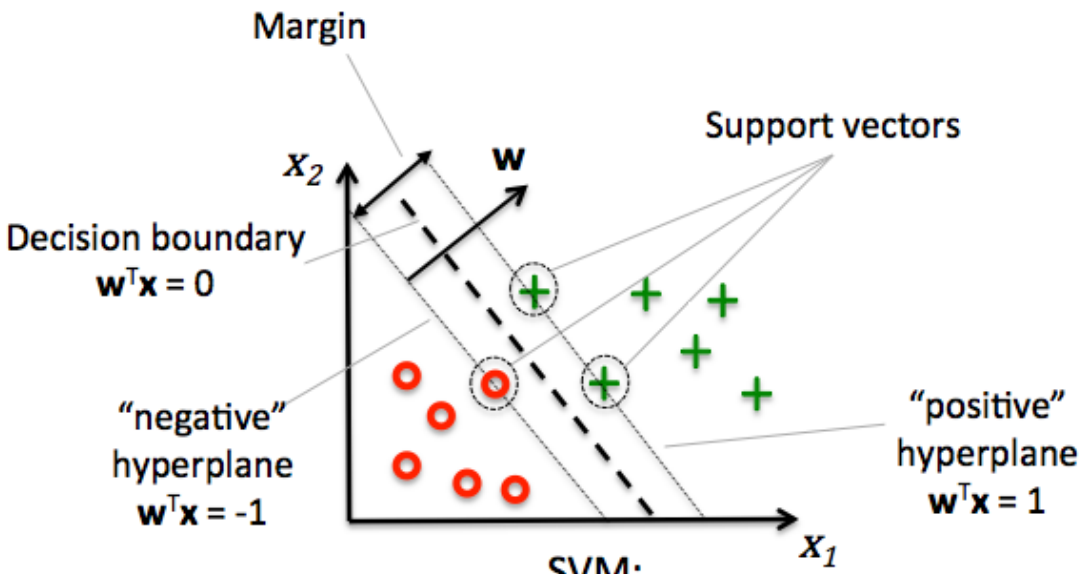
Support Vector Machines

- In SVMs, the optimization objective is to maximize the **margin**
- The margin is defined as the distance between the separating hyperlane and the training samples that are closest to this hyperplane (**support vectors**)
- Intuitively, the larger the margin, the lower generalization error
- Models with small margin prone to overfitting

Maximum margin classification



Which hyperplane?



SVM:
Maximize the margin

Mathematical intuition

Positive and *negative* hyperplanes that are parallel to the decision boundary, which can be expressed as follows:

$$w_0 + \mathbf{w}^T \mathbf{x}_{pos} = 1$$

$$w_0 + \mathbf{w}^T \mathbf{x}_{neg} = -1$$

Distance between these two planes (prove it!), i.e. the margin:

$$\frac{2}{\|\mathbf{w}\|}$$

Where the length of the vector $\mathbf{w}$ is defined as follows:

$$\|\mathbf{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

Constrained optimization problem

Minimize:

$$\frac{1}{2}\|\mathbf{w}\|^2$$

Subject to constraints that the samples are classified correctly:

$$w_0 + \mathbf{w}^T\mathbf{x}^{(i)} \geq 1 \text{ if } y^{(i)} = 1$$

$$w_0 + \mathbf{w}^T\mathbf{x}^{(i)} < -1 \text{ if } y^{(i)} = -1$$

These equations say that all negative and positive samples should fall respectively on one side of the negative and positive hyperplanes. This can be written more compactly:

$$y^{(i)}\left(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}\right) \geq 1 \quad \forall_i$$

SVM Solution

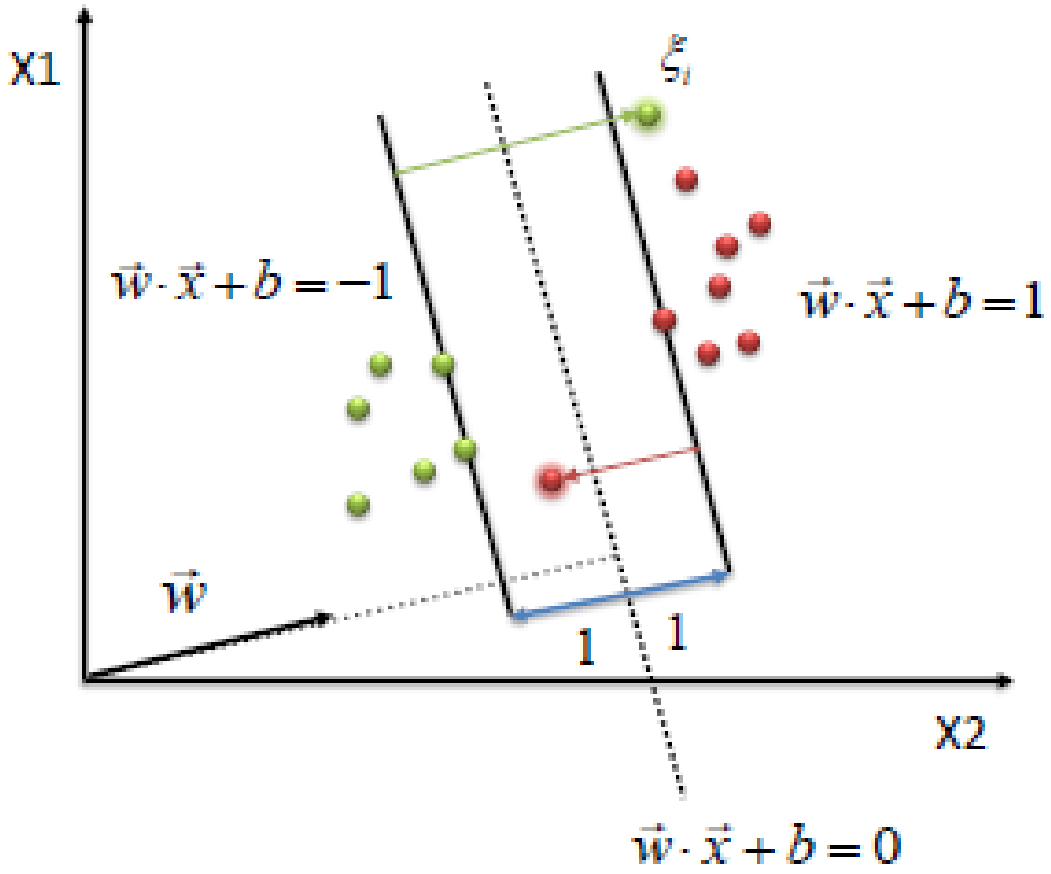
Classsifier

$$f(\mathbf{x}) = sgn(\mathbf{w}^T\mathbf{x} + w_0)$$

Weights

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i$$

Slack variables / soft margin SVM



slack variable:

$$\xi_i$$

Allow some instances to fall off the margin, but penalize them

Source: http://www.saedsayad.com/support_vector_machine.htm

Extending SVM to non-linearly separable cases

- Need to relax the linear constraints
- To ensure convergence in presense of misclassifications
- Introduce slack variables ξ

$$\mathbf{w}^T\mathbf{x}^{(i)} \geq 1 - \xi^{(i)} \text{ if } y^{(i)} = 1$$

$$\mathbf{w}^T\mathbf{x}^{(i)} < -1 + \xi^{(i)} \text{ if } y^{(i)} = -1$$

- New objective to be minimized:

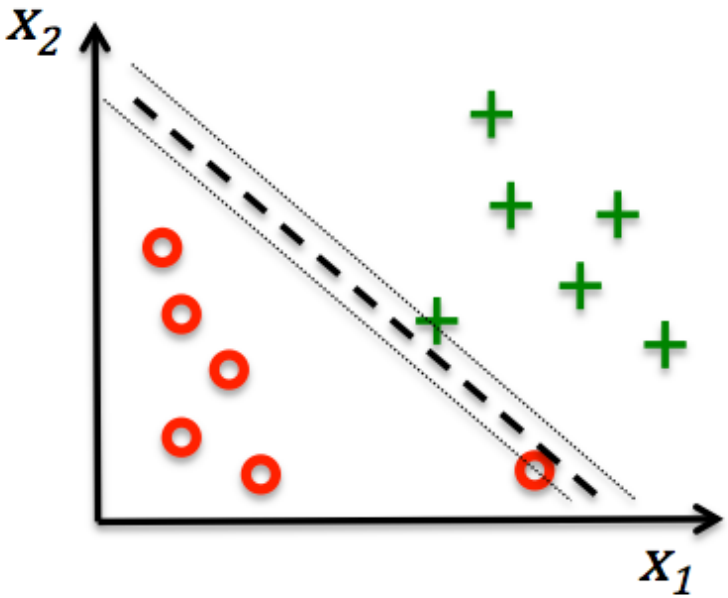
$$\frac{1}{2}\|\mathbf{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

Regularization in SVMs

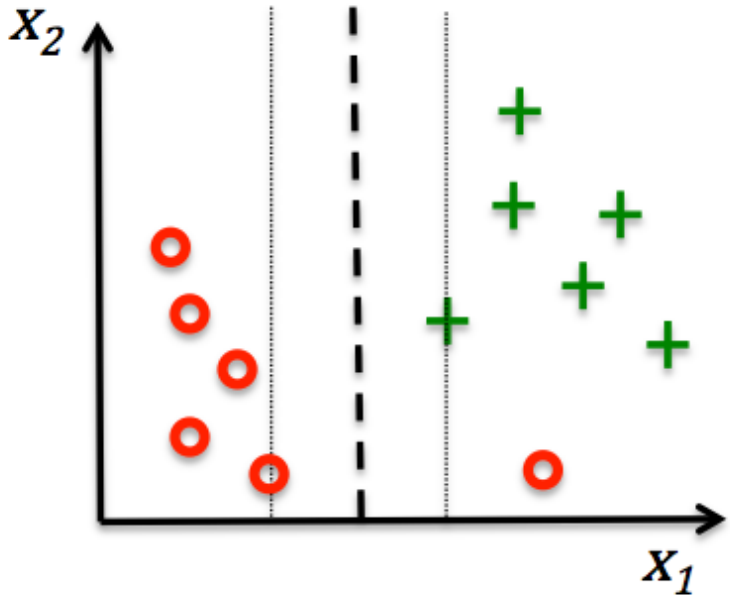
$$\frac{1}{2}\|\mathbf{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

- Large values of C - large error penalties
- Small values of C - less strict about misclassifications
- Parameter C controls width of the margin
- I.e. C is a way to do regularization in SVMs

Regularization in SVMs

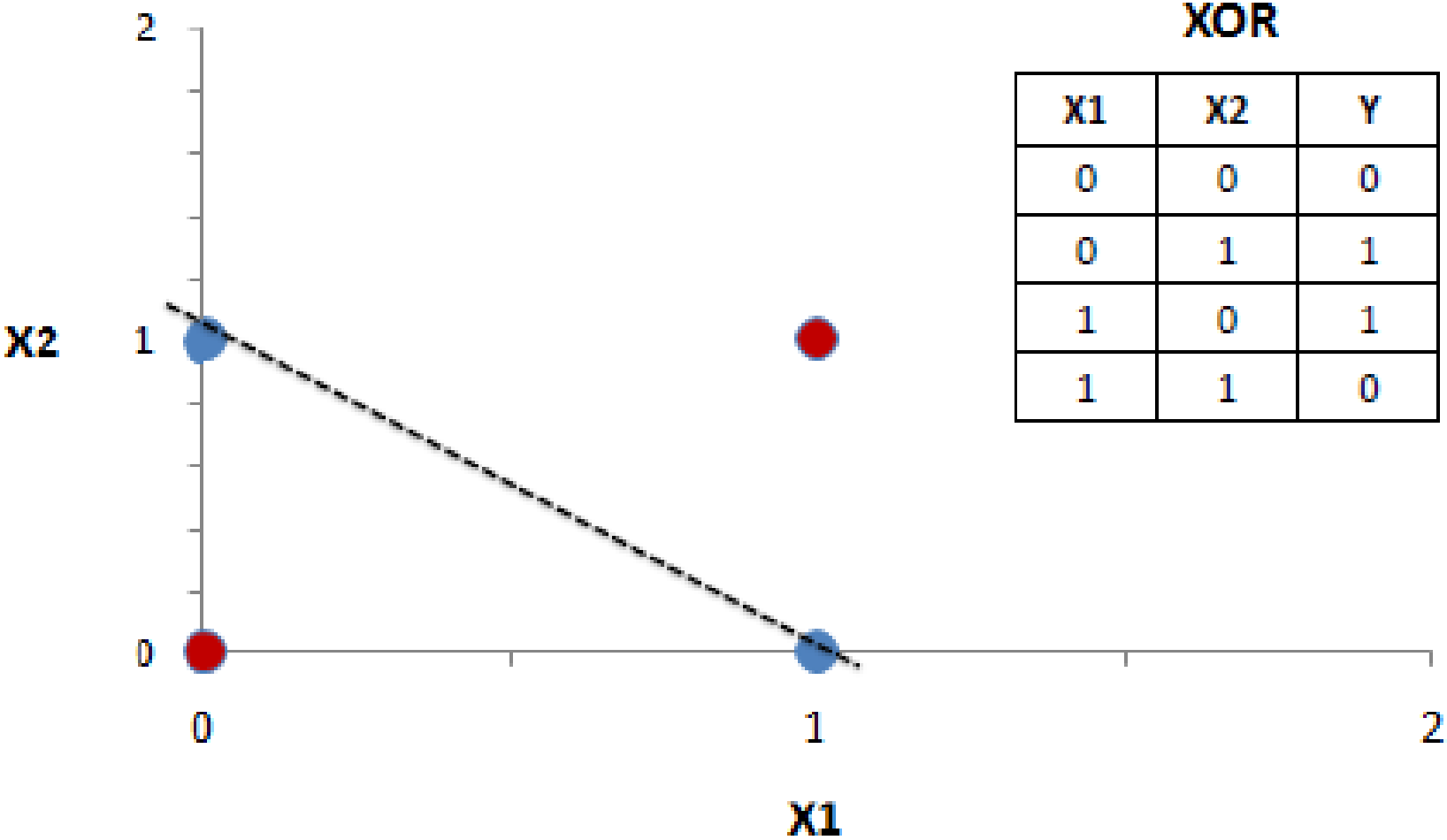


Large value for parameter C



Small value for parameter C

Exclusive OR (XOR) linear separability



XOR

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

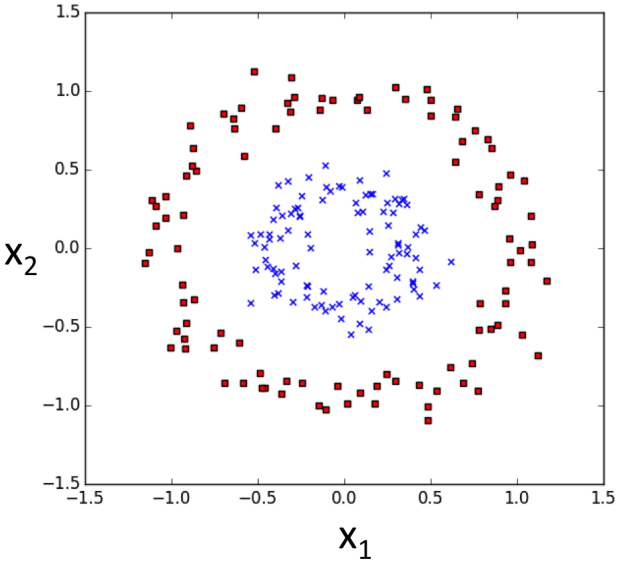
Source: http://www.saedsayad.com/artificial_neural_network_bkp.htm

Generated XOR data

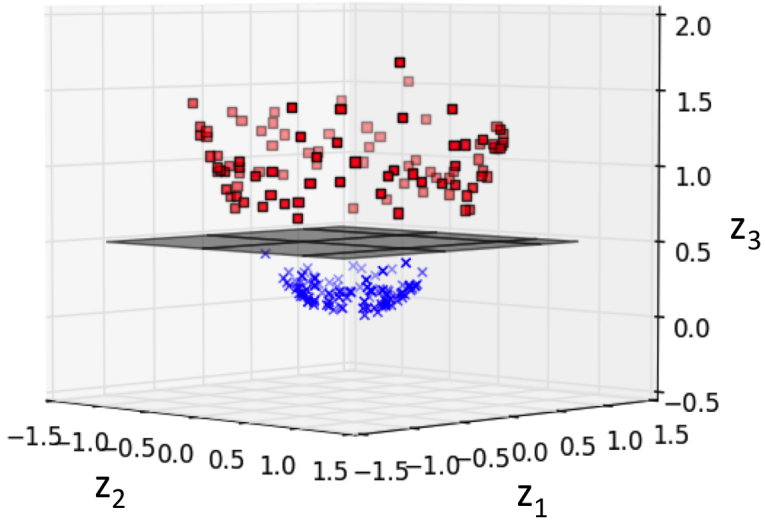
- ▸ iPython notebook on github
- Kernel methods create non-linear combinations of the original features
- Project onto a higher dimensional space where they are separable
- Mapping function $\phi(\cdot)$

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

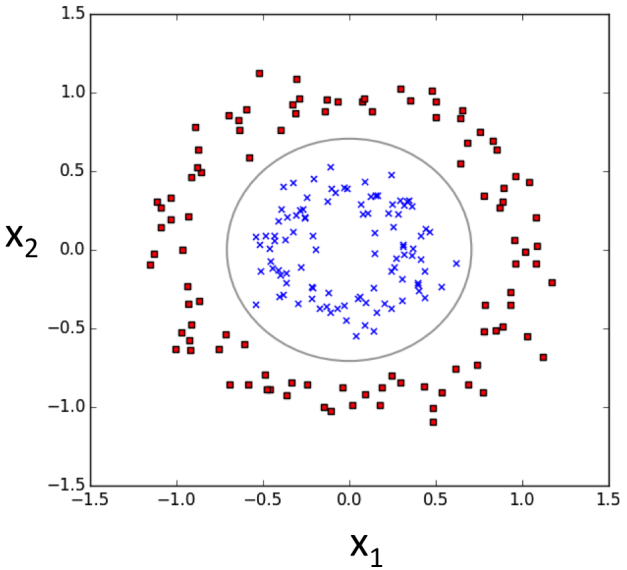
Turn non-separable classess are separable



ϕ



ϕ^{-1}



Kernel trick

General blueprint:

- Transform training data into a higher dimensional space via a mapping function $\phi(\cdot)$
- Train a linear SVM to classify the data in the new feature space
- Use the same mapping function $\phi(\cdot)$ to transform new (unseen) data
- Classify unseen data using the linear SVM model

Problem with explicit mapping

- The construction of the new features is computationally expensive
- Fortunately, we have the *kernel trick*
- Decision boundary relies on dot products in input space
- Need to replace the dot product

$$\mathbf{x}^{(i)\,T}\mathbf{x}^{(j)} \text{ by } \phi\left(\mathbf{x}^{(i)}\right)^T \phi\left(\mathbf{x}^{(j)}\right)$$

- No need to calculate this dot product explicitly
- Instead, we define a kernel function:

$$k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \phi\left(\mathbf{x}^{(i)}\right)^T \phi\left(\mathbf{x}^{(j)}\right)$$

Kernel Trick: Example

$$\mathbf{x} = (x_1, x_2),\ \mathbf{z} = (z_1, z_2),\ K(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle^2$$

$$\begin{aligned} K(x, z) = (x_1 z_1 + x_2 z_2)^2 = (x_1^2 z_1^2 + 2 x_1 z_1 x_2 z_2 + x_2^2 z_2^2) = \\ = \langle (x_1^2, \sqrt{2} x_1 x_2, x_2^2) \cdot (z_1^2, \sqrt{2} z_1 z_2, z_2^2) \rangle = \langle \phi(\mathbf{x}) \phi(\mathbf{z}) \rangle \end{aligned}$$

RBF Kernel

One of the most widely used kernels is the *Radial Basis Function kernel* (RBF kernel) or Gaussian kernel:

$$k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right)$$

This is often simplified to:

$$k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left(-\gamma \, \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\right)$$

Here, $\gamma = \frac{1}{2\sigma^2}$ is a free parameter that is to be optimized.

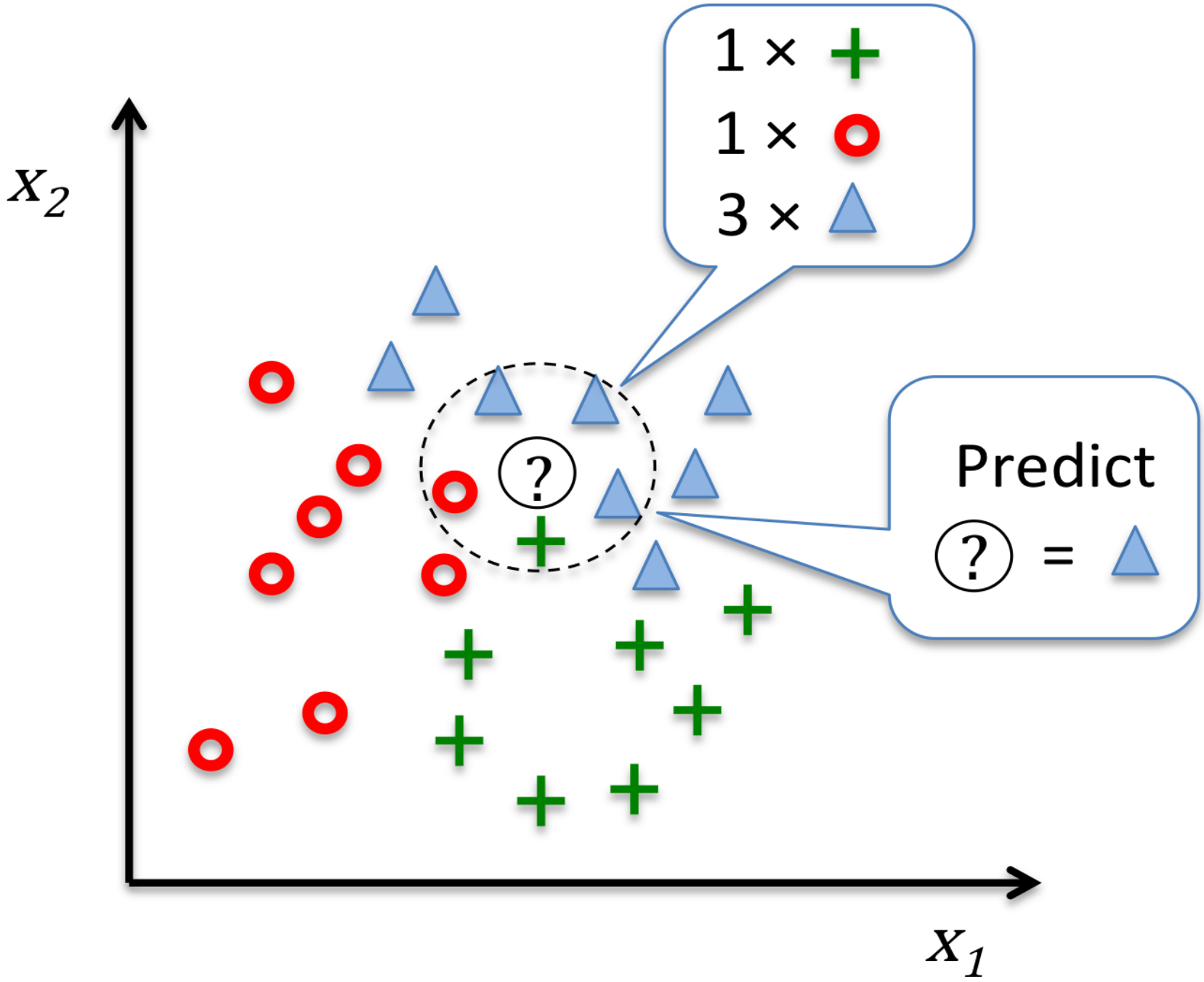
- The term *kernel* can be interpreted as a *similarity function* between a pair of samples
- The minus sign inverts the distance measure into a similarity score (from 0/dissimilar to 1/very similar)
- Use RBF kernel to separate XOR data
- Vary the γ parameter
- ▸ iPython notebook on github

K-nearest neighbors

- KNN is an example of a non-parametric model
- Parametric models learn parameters from training data
- Once training done, the training set not required
- KNN is an instance-based learner

Basic KNN algorithm

- Choose k and a distance metric
- Find k nearest neighbors of the sample to be classified
- Assign the class label by majority vote


1 ×
1 ×
3 ×
Predict
? =
?
x_2
x_1

KNN advantages

- Classifier immediately adapts as we receive new training examples
- But computational complexity grows linearly with the number of samples
- Need efficient data structures such as KD-trees

Distance metrics:

$$d\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} - x_k^{(j)}\right|^p}$$

Euclidean distance if we set the parameter $p = 2$
Manhattan distance if we set the parameter $p = 1$