CS/ECE 148 –

Data Science Fundamentals

k-NN Classification and Support Vector Machines

UCLA Computer Science

Lecture Outline

- *k*-NN review
- *k*-NN for Classification
- Support Vector Machines (SVMs)
  - Classifying Linear Separable Data
  - Classifying Linear Non-Separable Data
  - Kernel Trick

k-Nearest Neighbors

We’ve already seen the *k*-NN method for predicting a quantitative response (it was the very first method we introduced). How was *k*-NN implemented in the Regression setting (quantitative response)?

The approach was simple: to predict an observation’s response, use the **other** available observations that are most similar to it.

For a specified value of *k*, each observation’s outcome is predicted to be the **average** of the *k*-closest observations as measured by some distance of the predictor(s).

With one predictor, the method was easily implemented.

Review: Choice of k

How well the predictions perform is related to the choice of *k*.

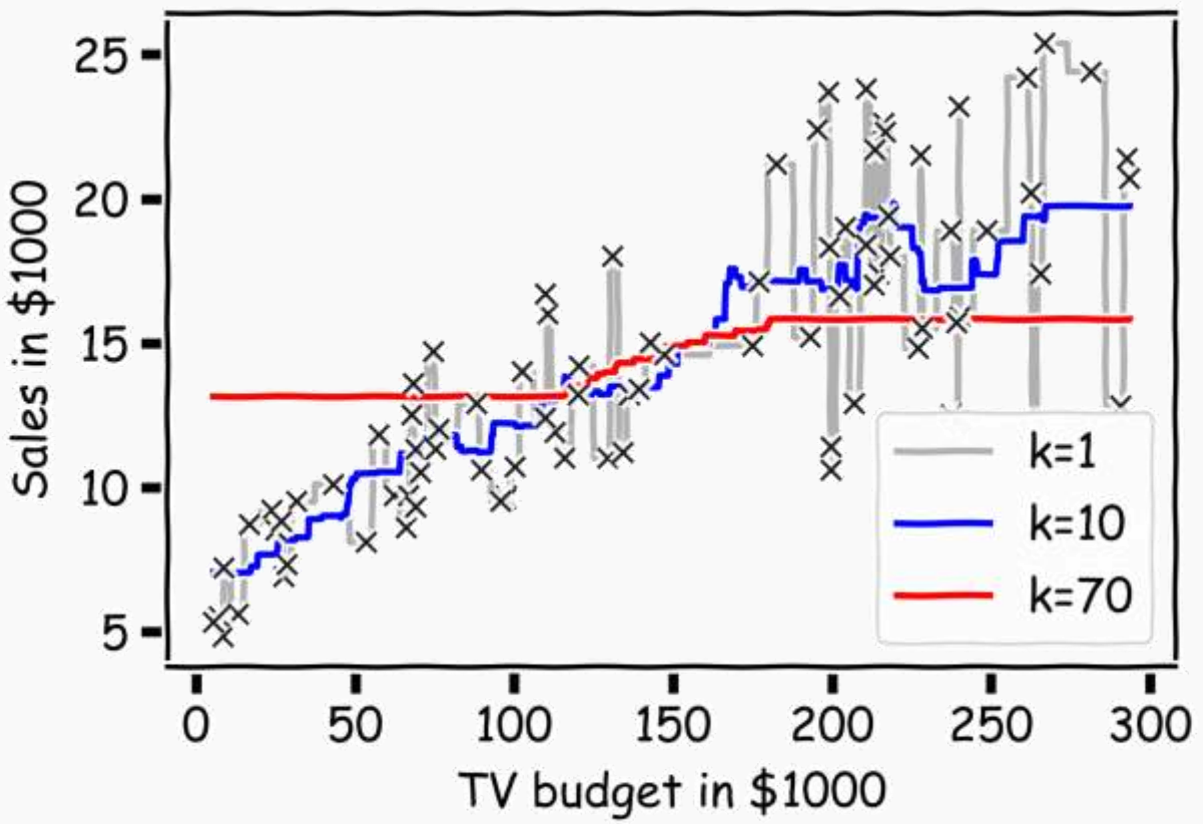
What will the predictions look like if *k* is very small? What if it is very large?

More specifically, what will the predictions be for new observations if *k* = *n*?

$$\bar{Y}$$

A picture is worth a thousand words...

Choice of *k* matters



k-NN for Classification

k-NN for Classification

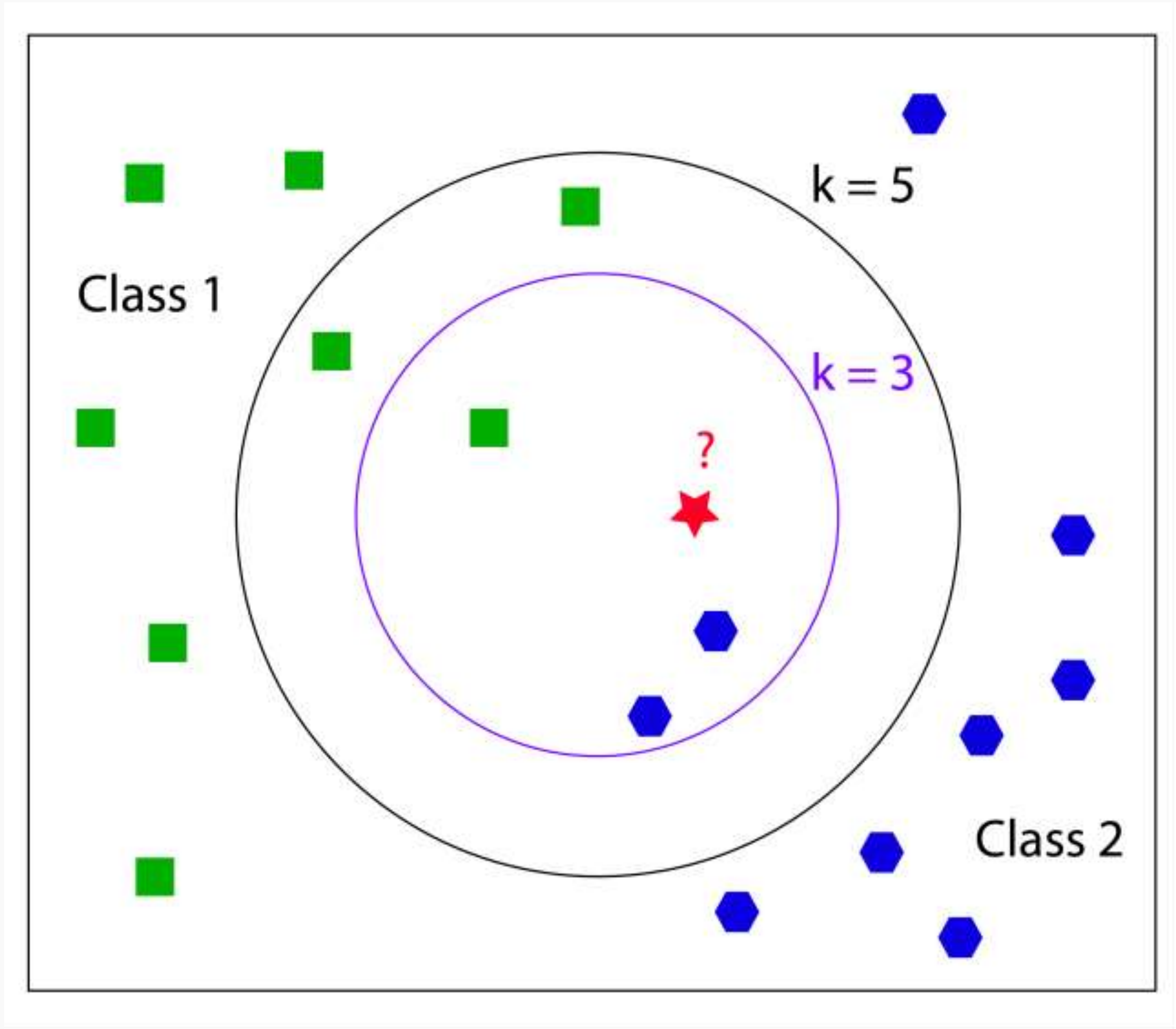
How can we modify the *k*-NN approach for classification?

The approach here is the same as for *k*-NN regression: use the other available observations that are most similar to the observation we are trying to predict (classify into a group) based on the predictors at hand.

How do we classify which category a specific observation should be in based on its nearest neighbors?

The category that shows up the most among the nearest neighbors.

k-NN for Classification



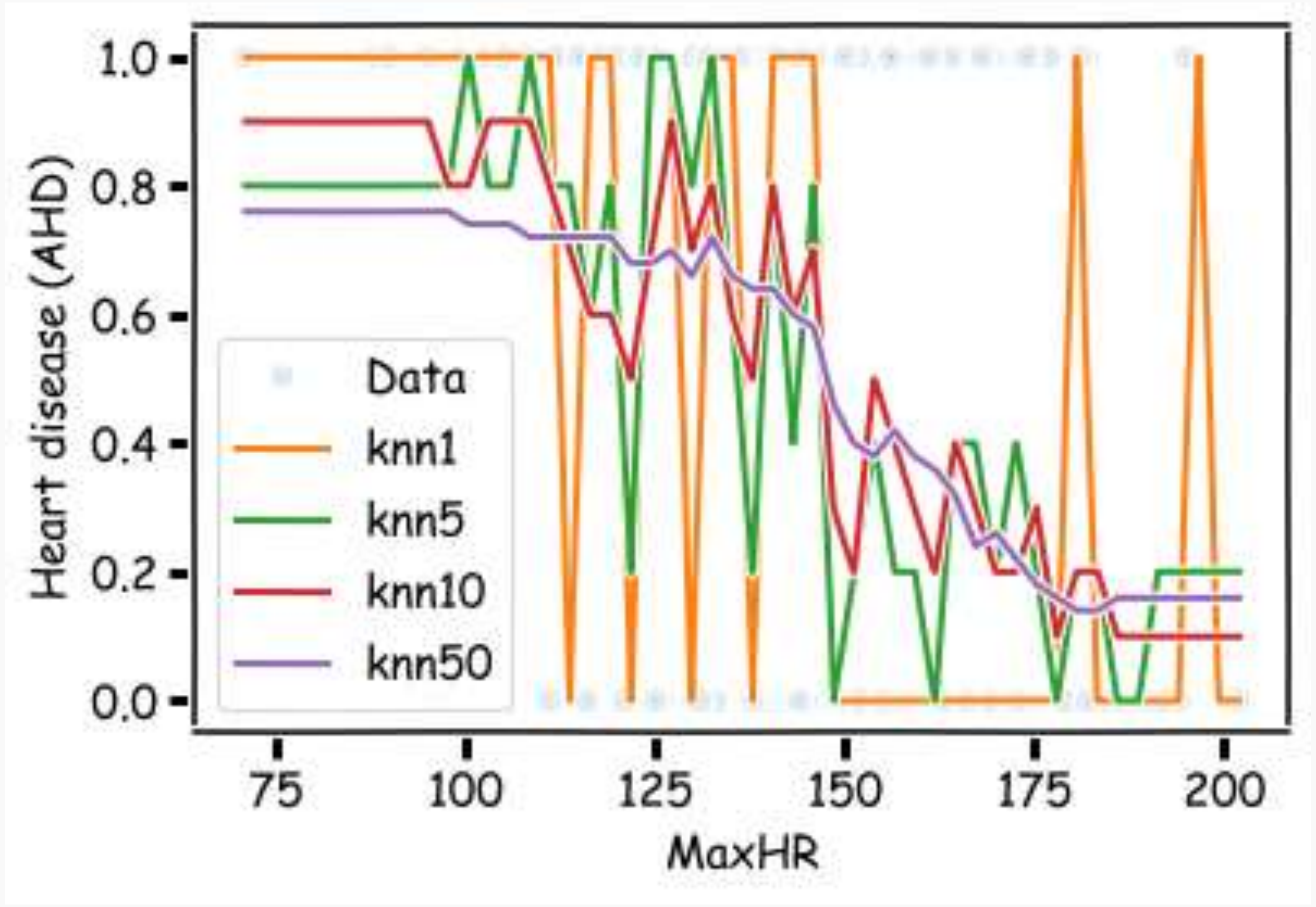
k-NN for Classification: formal definition

The *k*-NN classifier first identifies the *k* points in the training data that are closest to x_0, represented by $\mathcal{N}_0$. It then estimates the conditional probability for class *j* as the fraction of points in $\mathcal{N}_0$ whose response values equal *j*:

$$P(Y = j | X = x_0) = \frac{1}{k} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

Then, the *k*-NN classifier applies Bayes rule and classifies the test observation, x_0, to the class with largest estimated probability.

Estimated Probabilities in *k*-NN Classification



k-NN for Classification (cont.)

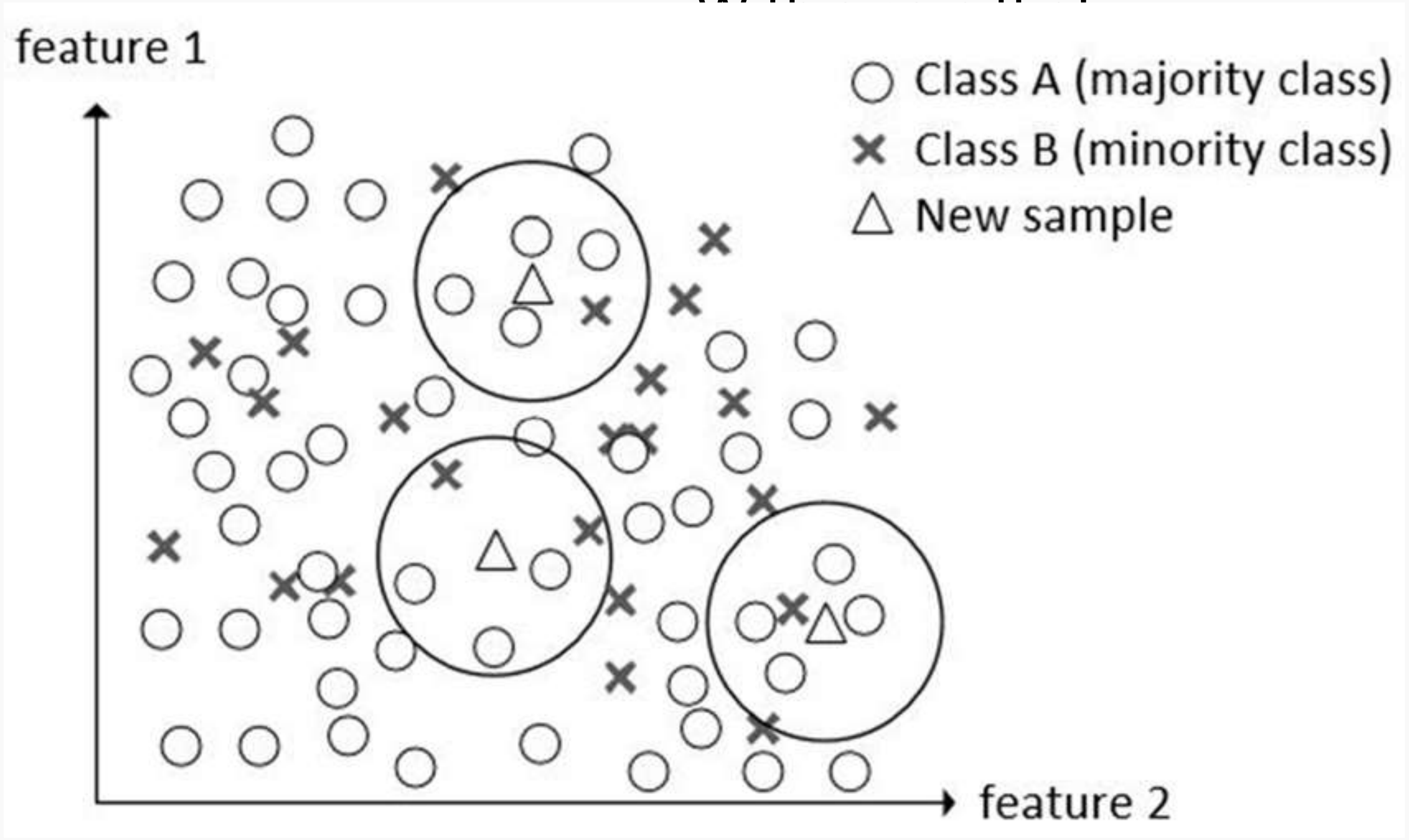
There are some issues that may arise:

- How can we handle a tie? With a coin flip!
- What could be a major problem with always classifying to the most common group amongst the neighbors?

  If one category is much more common than the others then all the predictions may be the same!
- How can we handle this?

  Rather than classifying with the most likely group, use a biased coin flip to decide which group to classify to.

k-NN for Classification (cont.)



k-NN with Multiple Predictors

How could we extend *k*-NN (both regression and classification) when there are multiple predictors?

We would need to define a measure of distance for observations in order to which are the most similar to the observation we are trying to predict.

Euclidean distance is a good option. To measure the distance of a new observation, $\boldsymbol{x}_0$ from each observation in the data set, $\boldsymbol{x}_i$:

$$D^2(\boldsymbol{x}_i, \boldsymbol{x}_0) = \sum_{j=1}^{P} \left(x_{i,j} - x_{0,j}\right)^2$$

k-NN with Multiple Predictors (cont.)

But what must we be careful about when measuring distance?

1. Differences in variability in our predictors!
2. Having a mixture of quantitative and categorical predictors.

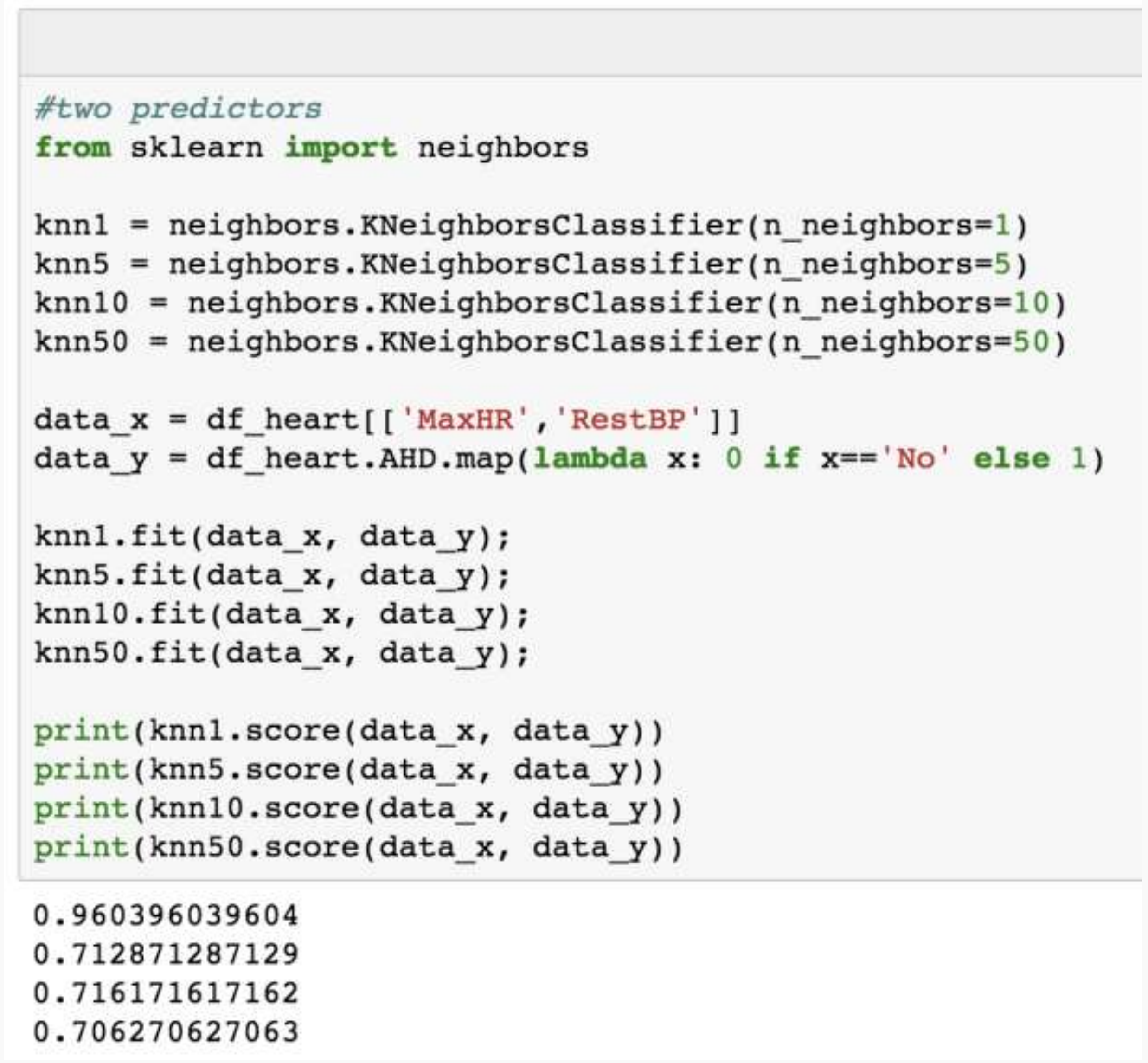
So what should be good practice? To determine closest neighbors when $p > 1$, you should first standardize the predictors! And you can even standardize the binaries if you want to include them.

How else could we determine closeness in this multi-dimensional setting?

k-NN Classification in Python

Performing kNN classification in python is done via **`KNeighborsClassifier`** in `sklearn.neighbors`.

An example:



```
#two predictors
from sklearn import neighbors

knn1 = neighbors.KNeighborsClassifier(n_neighbors=1)
knn5 = neighbors.KNeighborsClassifier(n_neighbors=5)
knn10 = neighbors.KNeighborsClassifier(n_neighbors=10)
knn50 = neighbors.KNeighborsClassifier(n_neighbors=50)

data_x = df_heart[['MaxHR','RestBP']]
data_y = df_heart.AHD.map(lambda x: 0 if x=='No' else 1)

knn1.fit(data_x, data_y);
knn5.fit(data_x, data_y);
knn10.fit(data_x, data_y);
knn50.fit(data_x, data_y);

print(knn1.score(data_x, data_y))
print(knn5.score(data_x, data_y))
print(knn10.score(data_x, data_y))
print(knn50.score(data_x, data_y))
```

```
0.960396039604
0.712871287129
0.716171617162
0.706270627063
```

Support Vector Machines (SVMs)

Outline

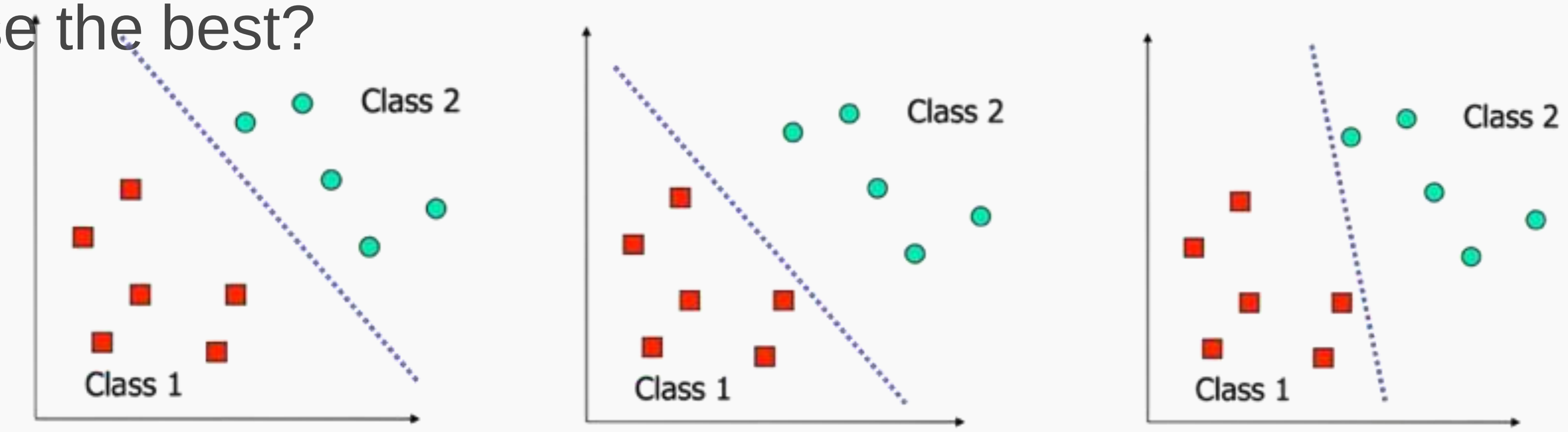
- Classifying Linear Separable Data
- Classifying Linear Non-Separable Data
- Kernel Trick

Decision Boundaries Revisited

In logistic regression, we learn a ***decision boundary*** that separates the training classes in the feature space.

When the data can be perfectly separated by a linear boundary, we call the data ***linearly separable***.

In this case, multiple decision boundaries can fit the data. How do we choose the best?



Question: What happens to our logistic regression model when training on linearly separable datasets?

Decision Boundaries Revisited (cont.)

Constraints on the decision boundary:

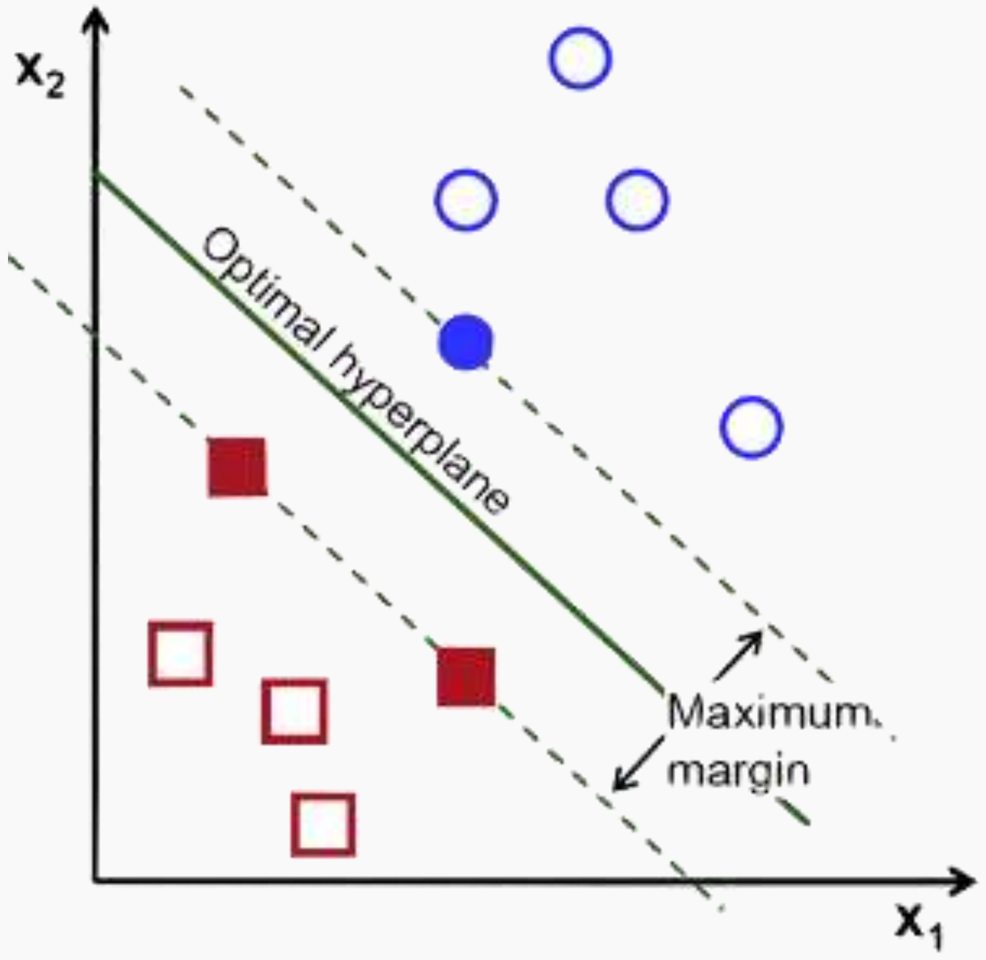
- In logistic regression, we typically learn an $\ell 1$ or $\ell 2$ regularized model.
- So, when the data is linearly separable, we choose a model with the ‘smallest coefficients’ that still separate the classes.
- The purpose of regularization is to prevent overfitting.

Decision Boundaries Revisited (cont.)

Constraints on the decision boundary:

- We can consider alternative constraints that prevent overfitting.
- For example, we may prefer a decision boundary that does not ‘favor’ any class (esp. when the classes are roughly equally populous).
- Geometrically, this means choosing a boundary that maximizes the distance or ***margin*** between the boundary and both classes.

Illustration of an SVM

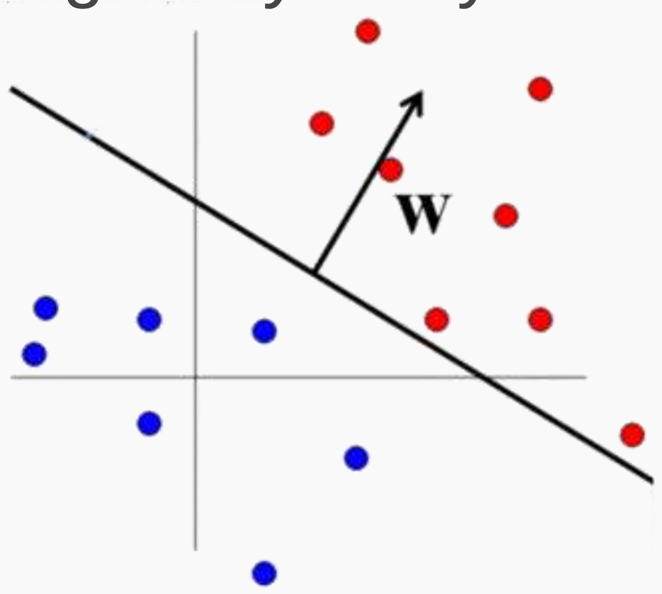


Geometry of Decision Boundaries

Recall that the decision boundary is defined by some equation in terms of the predictors. A linear boundary is defined by:

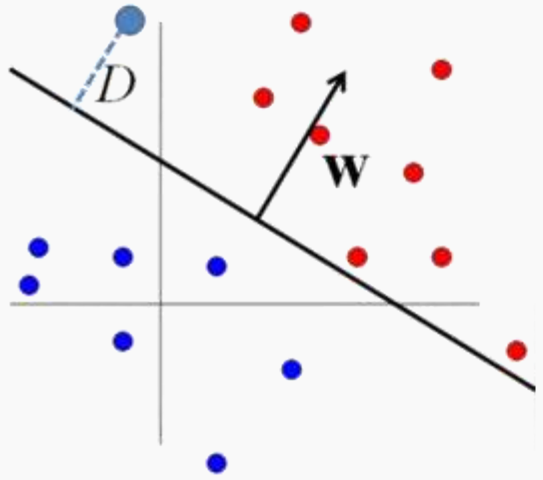
$$w^\top x + b = 0 \text{ (General equation of a hyperplane)}$$

Recall that the non-constant coefficients, w, represent a ***normal vector***, pointing orthogonally away from the plane



Geometry of Decision Boundaries (cont.)

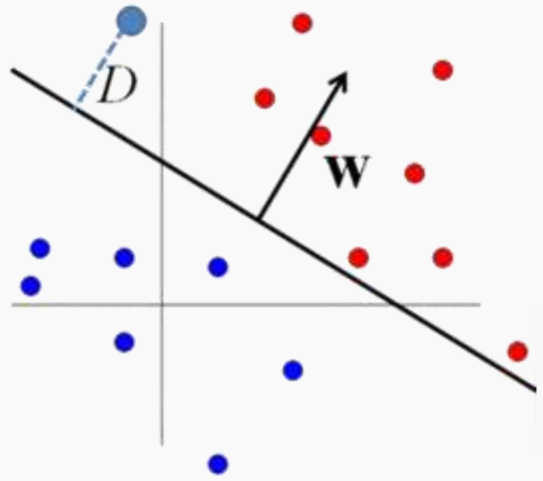
Now, using some geometry, we can compute the distance between any point to the decision boundary using *w* and *b*.

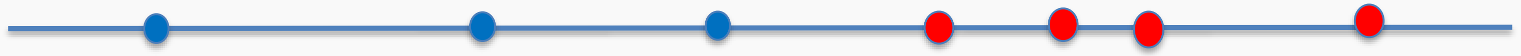


The signed distance from a point $x \in \mathbb{R}^n$ to the decision boundary is

$$D(x) = \frac{w^\top x + b}{||w||}$$ (Euclidean Distance Formula)

Linear points





Maximizing Margins

Now we can formulate our goal - find a decision boundary that maximizes the distance to both classes - as an optimization problem:

$$\begin{cases} \max\limits_{w,b} M \\ \text{such that } |D(x_n)| = \frac{y_i(w^\top x_n + b)}{||w||} \geq M,\ n = 1, \ldots, N \end{cases}$$

where *M* is a real number representing the width of the 'margin' and $y_i = \pm 1$. The inequalities $|D(x_n)| \geq M$ are called ***constraints***.

The constrained optimization problem as present here looks tricky. Let's simplify it with a little geometric intuition.

Maximizing Margins (cont.)

Notice that maximizing the distance of ***all points*** to the decision boundary, is exactly the same as maximizing the distance to the ***closest points***.

The points closest to the decision boundary are called ***support vectors***.

For any plane, we can always scale the equation:

$$w^\top x + b = 0$$

so that the support vectors lie on the planes:

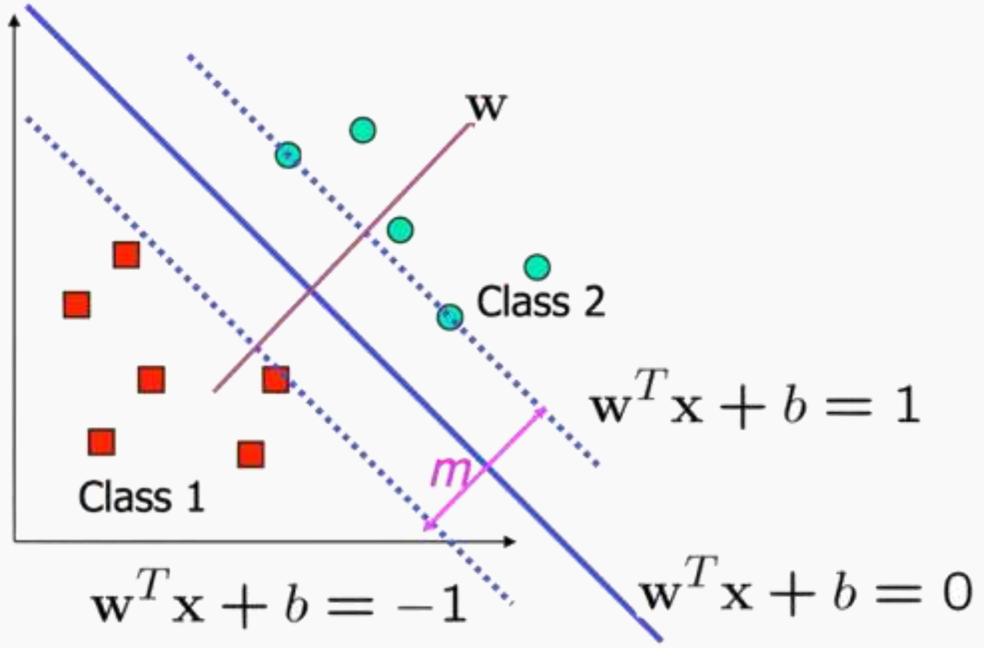
$$w^\top x + b = \pm 1,$$

depending on their classes.

Maximizing Margins Illustration

For points on planes $w^T x + b = \pm 1$, their distance to the decision boundary is ±1/‖w‖.

So we can define the ***margin*** of a decision boundary as the distance to its support vectors, $m = 2/\|w\|$.


w
Class 2
Class 1
m
$\mathbf{w}^T\mathbf{x} + b = 1$
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$

Support Vector Classifier: Hard Margin

Finally, we can reformulate our optimization problem - find a decision boundary that maximizes the distance to both classes - as the maximization of the margin, *m*, ***while maintaining zero misclassifications***,

$$\begin{cases} \max\limits_{w,b} \dfrac{2}{||w||} \\ \text{such that } y_n(w^\top x_n + b) \geq 1,\ n = 1, \ldots, N \end{cases}$$

The classifier learned by solving this problem is called ***hard margin support vector classification***.

Often SVC is presented as a minimization problem:

$$\begin{cases} \min\limits_{w,b} ||w||^2 \\ \text{such that } y_n(w^\top x_n + b) \geq 1,\ n = 1, \ldots, N \end{cases}$$

SVC and Convex Optimization

As a convex optimization problem SVC has been extensively studied and can be solved by a variety of algorithms:

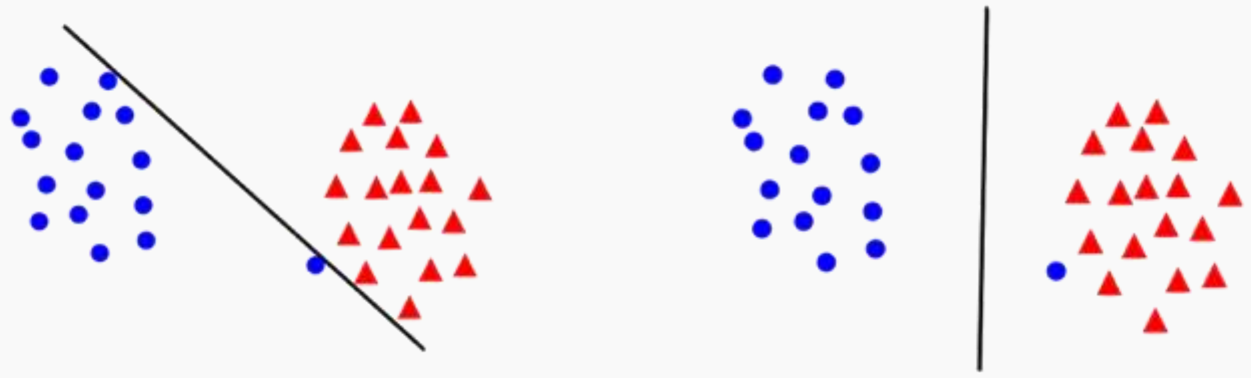
- **(Stochastic)** libLinear
  Fast convergence, moderate computational cost
- **(Greedy)** libSVM
  Fast convergence, moderate computational cost
- **(Stochastic)** Stochastic Gradient Descent Slow convergence, low computational cost per iteration
- **(Greedy)** Quasi-Newton Method
  Very fast convergence, high computational cost

Classifying Linear Non-Separable Data

Geometry of Data

Maximizing the margin is a good idea as long as we assume that the underlying classes are linear separable and that the data is noise free.

If data is noisy, we might be sacrificing generalizability in order to minimize classification error with a very narrow margin:



With every decision boundary, there is a trade-off between maximizing margin and minimizing the error.

Support Vector Classifier: Soft Margin

Since we want to balance maximizing the margin and minimizing the error, we want to use an objective function that takes both into account:

$$\begin{cases} \min\limits_{w,b} \|w\|^2 + \lambda \mathsf{Error}(w,b) \\ \mathsf{such\ that}\ y_n(w^\top x_n + b) \geq 1,\ n = 1, \ldots, N \end{cases}$$

where λ is an intensity parameter.

So just how should we compute the error for a given decision boundary?

Support Vector Classifier: Soft Margin (cont.)

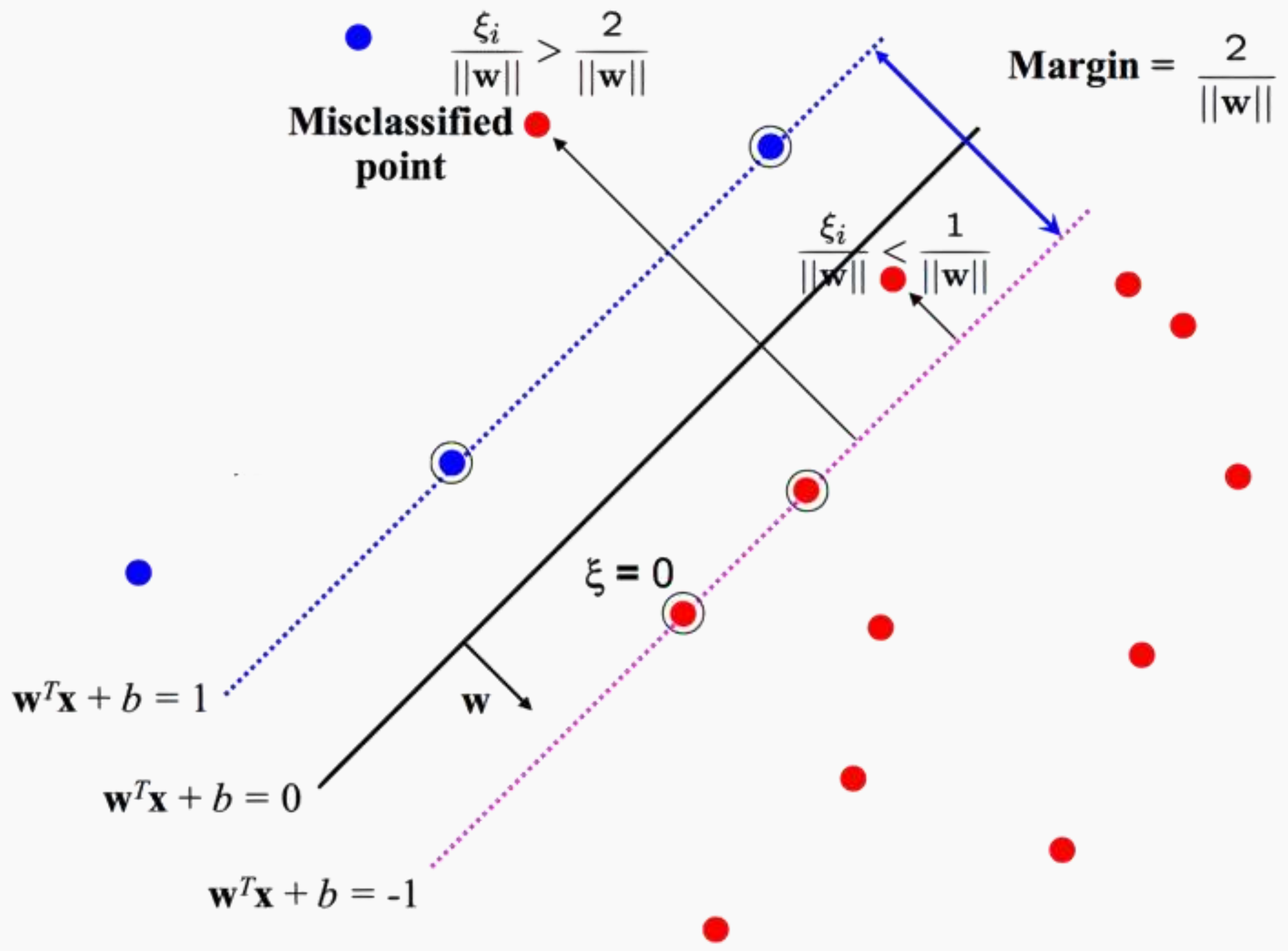
We want to express the error as a function of distance to the decision boundary.

Recall that the support vectors have distance 1/‖*w*‖ to the decision boundary. We want to penalize two types of ‘errors’

- **(margin violation)** points that are on the correct side of the boundary but are inside the margin. They have distance ξ /‖*w*‖, where $0 < \xi < 1$.
- **(misclassification)** points that are on the wrong side of the boundary. They have distance ξ/‖w‖, where $\xi > 1$.

Specifying a nonnegative quantity for ξ_n is equivalent to quantifying the error on the point x_n.

Support Vector Classifier: Soft Margin Illustration


$\frac{\xi_i}{||\mathbf{w}||} > \frac{2}{||\mathbf{w}||}$
Misclassified point
Margin = $\frac{2}{||\mathbf{w}||}$
$\frac{\xi_i}{||\mathbf{w}||} < \frac{1}{||\mathbf{w}||}$
$\xi = 0$
$\mathbf{w}^T\mathbf{x} + b = 1$
$\mathbf{w}$
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$

Support Vector Classifier: Soft Margin (cont.)

Formally, we incorporate error terms ξ_n 's into our optimization problem by:

$$\begin{cases} \min\limits_{\xi_n \in \mathbb{R}^+, w, b} \|w\|^2 + \lambda \sum\limits_{n=1}^{N} \xi_n \\ \text{such that } y_n(w^\top x_n + b) \geq 1 - \xi_n, \; n = 1, \ldots, N \end{cases}$$

The solution to this problem is called ***soft margin support vector classification*** or simply ***support vector classification***.

Tuning SVC

Choosing different values for λ in

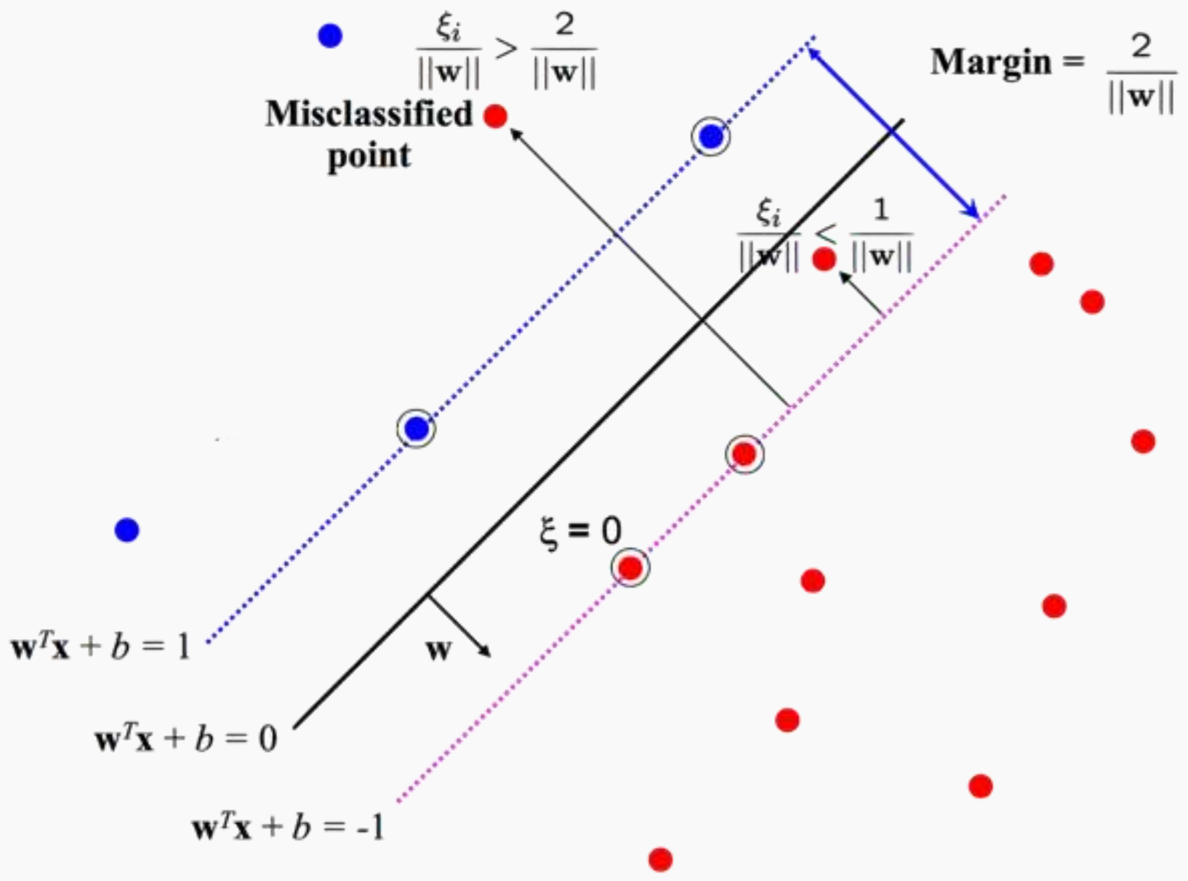
$$\begin{cases} \min\limits_{\xi_n \in \mathbb{R}^+, w, b} \|w\|^2 + \lambda \sum\limits_{n=1}^{N} \xi_n \\ \text{such that } y_n(w^\top x_n + b) \geq 1 - \xi_n, \ n = 1, \ldots, N \end{cases}$$

will give us different classifiers. In general,

- small λ penalizes errors less and hence the classifier will have a large margin
- large λ penalizes errors more and hence the classifier will accept narrow margins to improve classification
- setting $\lambda = \infty$ produces ???

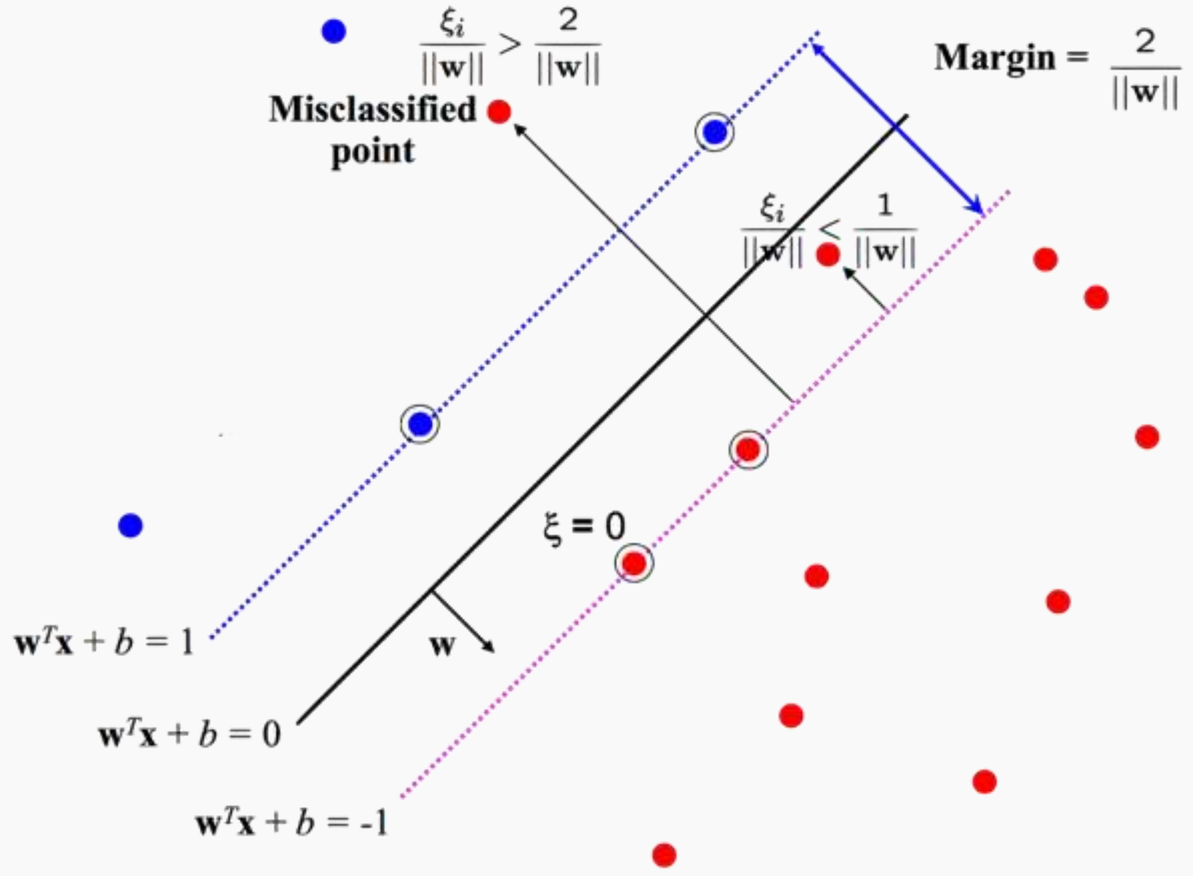
Decision Boundaries and Support Vectors

Recall how the error terms ξ_n's were defined: the points where $\xi_n = 0$ are precisely the support vectors


$\frac{\xi_i}{||\mathbf{w}||} > \frac{2}{||\mathbf{w}||}$
Misclassified point
Margin = $\frac{2}{||\mathbf{w}||}$
$\frac{\xi_i}{||\mathbf{w}||} < \frac{1}{||\mathbf{w}||}$
$\xi = 0$
$\mathbf{w}$
$\mathbf{w}^T\mathbf{x} + b = 1$
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$

Decision Boundaries and Support Vectors

Thus to re-construct the decision boundary, ***only the support vectors are needed!***



$\frac{\xi_i}{||\mathbf{w}||} > \frac{2}{||\mathbf{w}||}$
Margin = $\frac{2}{||\mathbf{w}||}$
Misclassified point
$\frac{\xi_i}{||\mathbf{w}||} < \frac{1}{||\mathbf{w}||}$
$\xi = 0$
$\mathbf{w}^T\mathbf{x} + b = 1$
$\mathbf{w}$
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$

Decision Boundaries and Support Vectors

The decision boundary of an SVC is given by

$$\hat{w}^\top x + \hat{b} = \sum_{x_n \text{ is a support vector}} \hat{\alpha}_n y_n (x_n^\top x_n) + b$$

where $\hat{\alpha}_n$ and the set of support vectors are found by solving the optimization problem.

- To classify a test point x_{test}, we predict

$$\hat{y}_{test} = \text{sign}\left(\hat{w}^\top x + \hat{b}\right)$$

SVC as Optimization

With the help of geometry, we translated our wish list into an optimization problem

$$\begin{cases} \min\limits_{\xi_n \in \mathbb{R}^+, w, b} \|w\|^2 + \lambda \sum_{n=1}^{N} \xi_n \\ \text{such that } y_n(w^\top x_n + b) \geq 1 - \xi_n, \ n = 1, \ldots, N \end{cases}$$

where ξ_n quantifies the error at x_n.

The SVC optimization problem is often solved in an alternate form (the dual form):

$$\max_{\alpha_n \geq 0, \sum_n \alpha_n y_n = 0} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m=1}^{N} y_n y_m \alpha_n \alpha_m x_n^\top x_m$$

Later we'll see that this alternate form allows us to use SVC with non-linear boundaries.

Extension to Non-linear Boundaries

Polynomial Regression: Two Perspectives

Given a training set:

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

with a single real-valued predictor, we can view fitting a 2nd degree polynomial model:

$$w_0 + w_1 x + w_2 x^2$$

on the data as the process of finding the best quadratic curve that fits the data. But in practice, we first expand the feature dimension of the training set

$$x_n \mapsto (x_n^0, x_n^1, x_n^2)$$

and train a ***linear model*** on the expanded data

$$\{(x_n^0, x_n^1, x_N^2, y_1), \ldots, (x_N^0, x_N^1, x_N^2, y_N)\}$$

Transforming the Data

The key observation is that training a polynomial model is just training a linear model on data with transformed predictors.

In our previous example, transforming the data to fit a 2nd degree polynomial model requires a map:

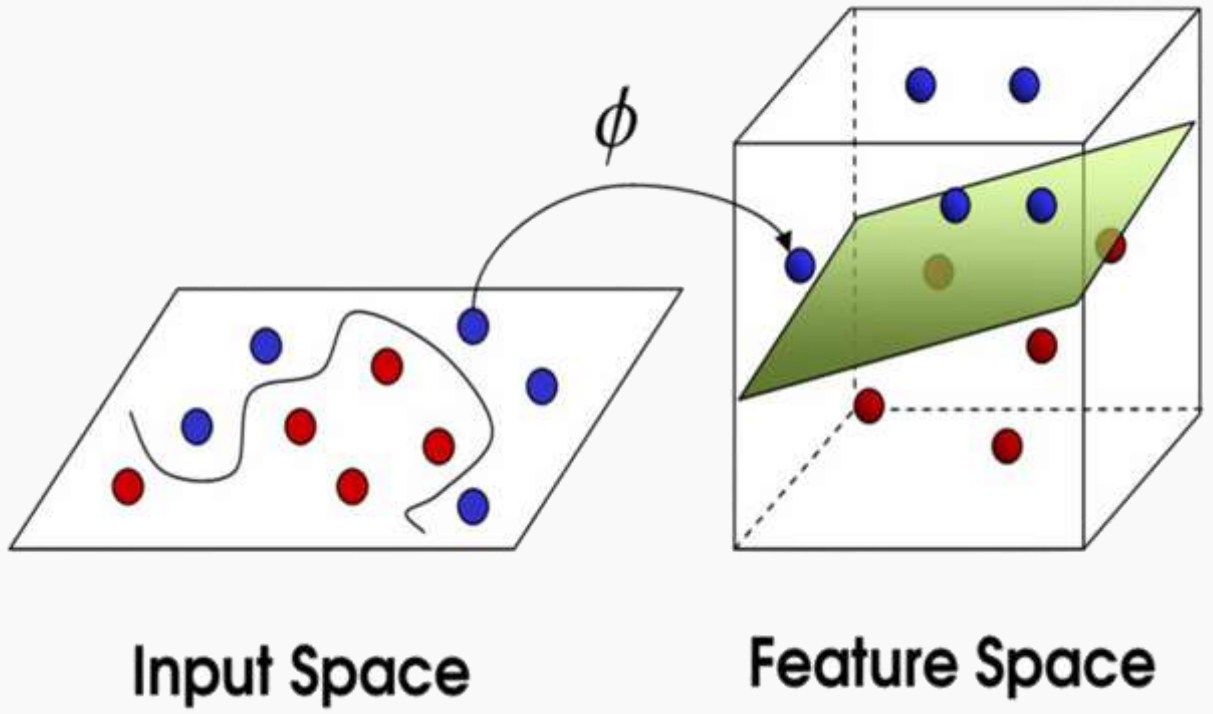
$$\phi : \mathbb{R} \to \mathbb{R}^3$$
$$\phi(x) = (x^0, x^1, x^2)$$

where $\mathbb{R}$ called the ***input space***, $\mathbb{R}^3$ is called the ***feature space***.

While the response may not have a linear correlation in the input space $\mathbb{R}$, it may have one in the feature space $\mathbb{R}^3$.

SVC with Non-Linear Decision Boundaries

The same insight applies to classification: while the response may not be linear separable in the input space, it may be in a feature space after a fancy transformation:



SVC with Non-Linear Decision Boundaries (cont.)

The motto: instead of tweaking the definition of SVC to accommodate non-linear decision boundaries, we map the data into a feature space in which the classes are linearly separable (or nearly separable):

- Apply transform $\phi: \mathbb{R}^J \rightarrow \mathbb{R}^{J'}$ on training data
  $$x_n \rightarrow \phi(x_n)$$
  where typically J' is much larger than J.
- Train an SVC on the transformed data
  $$\{(\phi(x_1), y_1), (\phi(x_2), y_2), \ldots, (\phi(x_N), y_N)\}$$

END

Inner Products

Since the feature space $\mathbb{R}^{J'}$ is potentially extremely high dimensional, computing ϕ explicitly can be costly.

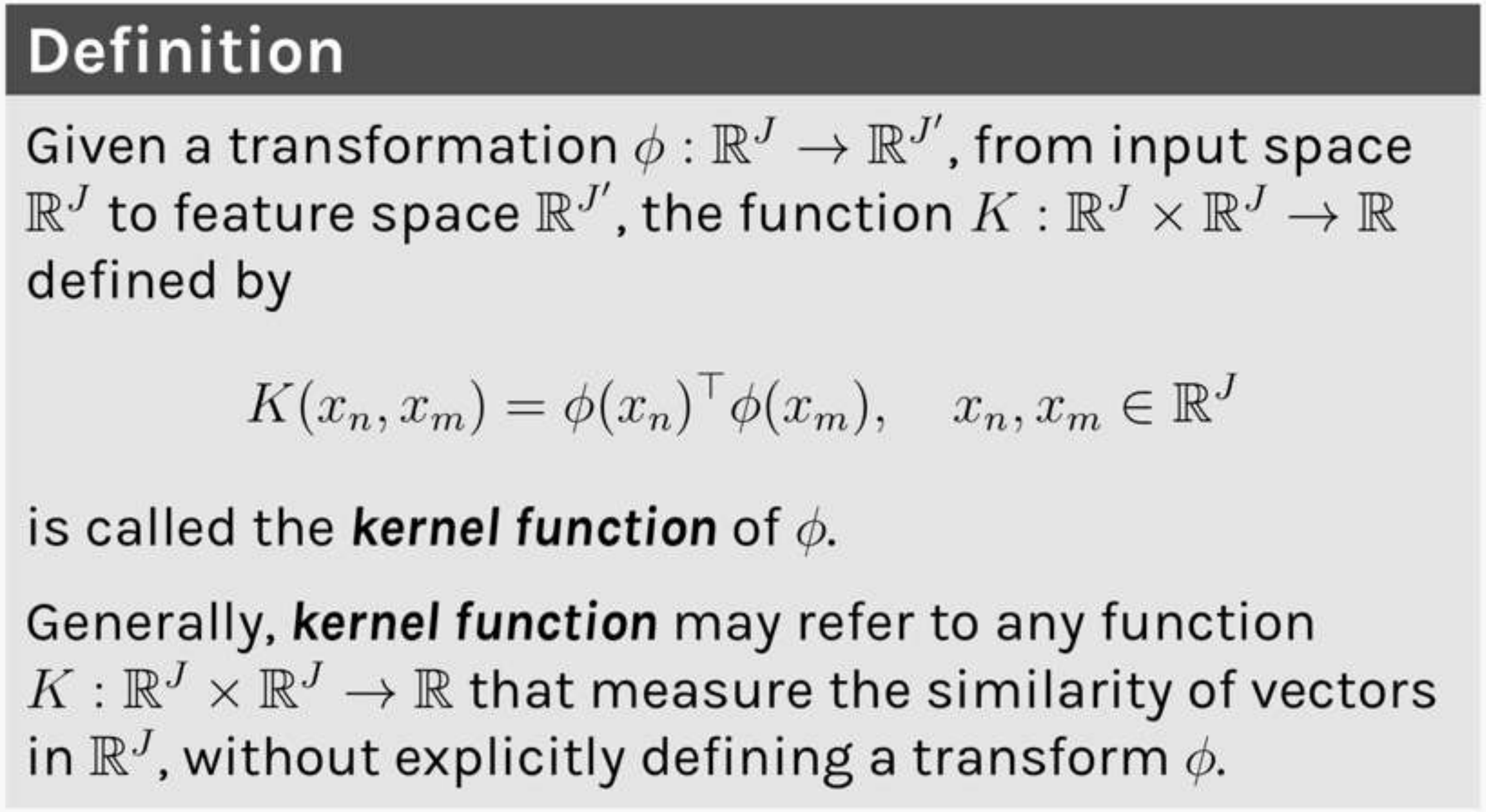
Instead, we note that computing ϕ is unnecessary. Recall that training an SVC involves solving the optimization problem:

$$\max_{\alpha_n \geq 0,\, \sum_n \alpha_n y_n = 0} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m=1}^{N} y_n y_m \alpha_n \alpha_m \phi(x_n)^\top \phi(x_m)$$

In the above, ***we are only interested in computing inner products*** $\phi(x_n)^T\phi(x_m)$***in the feature space*** and not the quantities $\phi(x_n)$ themselves.

The Kernel Trick

The ***inner product*** between two vectors is a measure of the similarity of the two vectors.



Definition

Given a transformation $\phi : \mathbb{R}^J \to \mathbb{R}^{J'}$, from input space $\mathbb{R}^J$ to feature space $\mathbb{R}^{J'}$, the function $K : \mathbb{R}^J \times \mathbb{R}^J \to \mathbb{R}$ defined by

$$K(x_n, x_m) = \phi(x_n)^\top \phi(x_m), \quad x_n, x_m \in \mathbb{R}^J$$

is called the ***kernel function*** of ϕ.

Generally, ***kernel function*** may refer to any function $K : \mathbb{R}^J \times \mathbb{R}^J \to \mathbb{R}$ that measure the similarity of vectors in $\mathbb{R}^J$, without explicitly defining a transform ϕ.

The Kernel Trick (cont.)

For a choice of kernel *K*,

$$K(x_n, x_m) = \phi(x_n)^\top \phi(x_m)$$

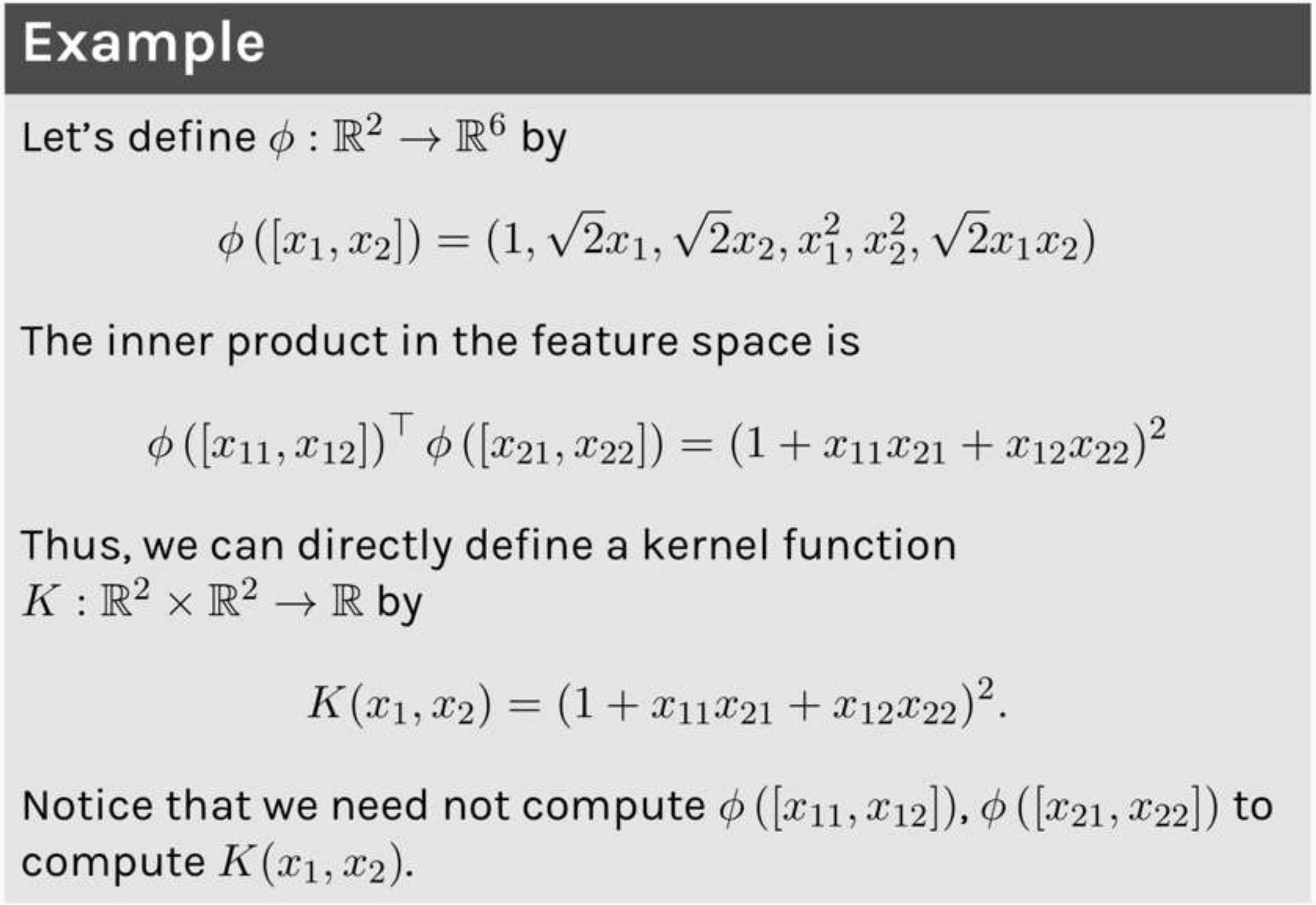
we train an SVC by solving

$$\max_{\alpha_n \geq 0,\, \sum_n \alpha_n y_n = 0} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m=1}^{N} y_n y_m \alpha_n \alpha_m K(x_n, x_m)$$

Computing $K(x_n, x_m)$ can be done without computing the mappings $\phi(x_n)$, $\phi(x_m)$.

This way of training a SVC in feature space without explicitly working with the mapping ϕ is called ***the kernel trick***.

Transforming Data: An Example



Example

Let's define $\phi : \mathbb{R}^2 \to \mathbb{R}^6$ by

$$\phi([x_1, x_2]) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2)$$

The inner product in the feature space is

$$\phi([x_{11}, x_{12}])^\top \phi([x_{21}, x_{22}]) = (1 + x_{11}x_{21} + x_{12}x_{22})^2$$

Thus, we can directly define a kernel function
$K : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ by

$$K(x_1, x_2) = (1 + x_{11}x_{21} + x_{12}x_{22})^2.$$

Notice that we need not compute $\phi([x_{11}, x_{12}])$, $\phi([x_{21}, x_{22}])$ to compute $K(x_1, x_2)$.

Kernel Functions

Common kernel functions include:

- **Polynomial Kernel** (kernel=‘poly’)

$$K(x_1, x_2) = (x_1^\top x_2 + 1)^d$$

  where *d* is a hyperparameter.

- **Radial Basis Function Kernel** (kernel=‘rbf’)

$$K(x_1, x_2) = \mathsf{exp}\left\{-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right\}$$

  where σ is a hyperparameter.

- **Sigmoid Kernel** (kernel=‘sigmoid’)

$$K(x_1, x_2) = \mathsf{tanh}(\kappa x_1^\top x_2 + \theta)$$

  where κ and θ are hyperparameters.