



Computer
Science

CSC380: Principles of Data Science

Nonlinear Models 1

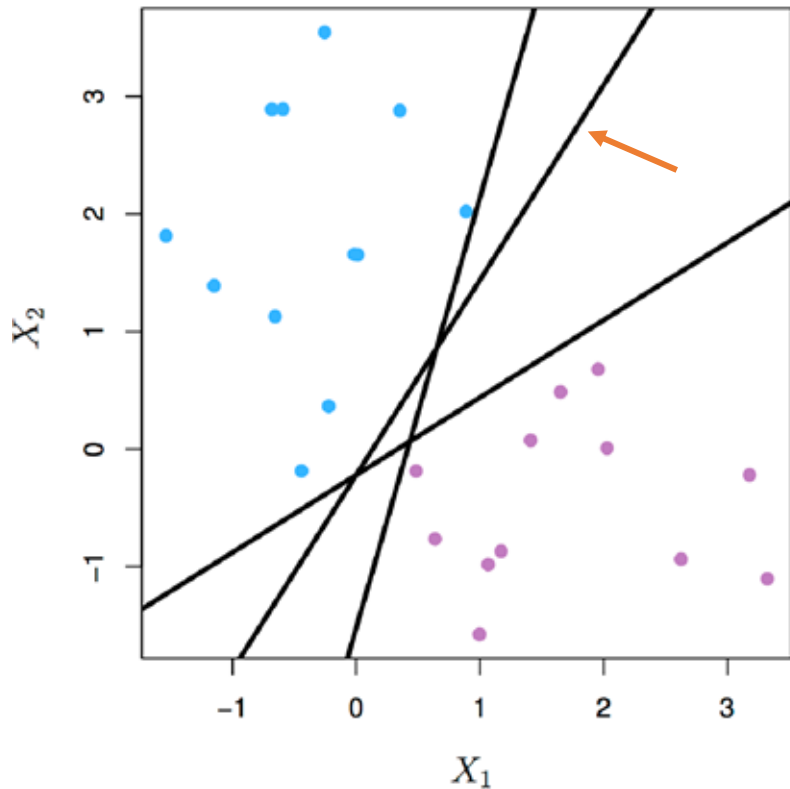
Xinchen Yu

- Support Vector Machine

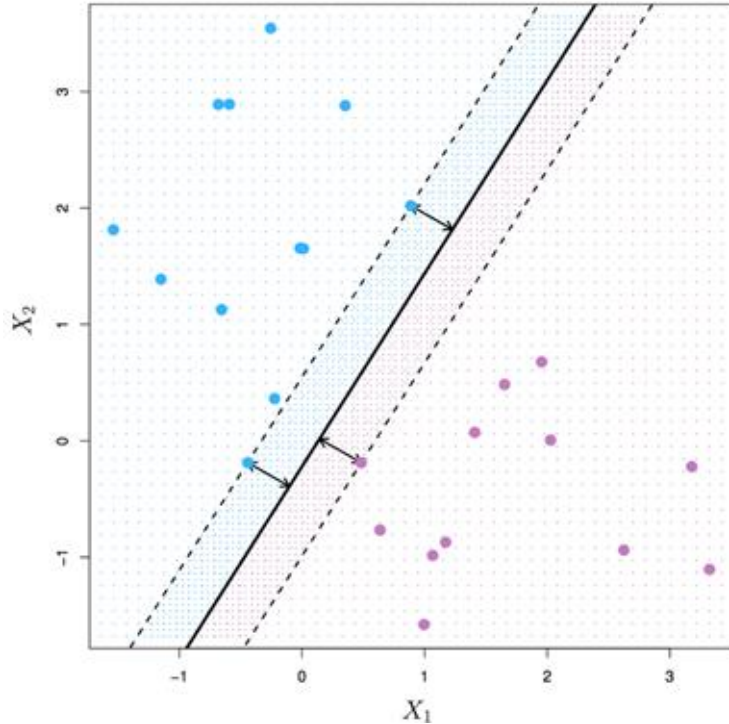
Linear Decision Boundary

Forget about the 'regression' point of view for now..

At the end of the day, we just want a line that separates the two classes well.



Q: but if you have to choose one, which one will you choose?



The **margin** measures minimum distance between each class and the decision boundary

Observation Decision boundaries with larger margins are more likely to generalize to unseen data

Idea Learn the classifier with the largest margin that still separates the data...

...we call this a **max-margin classifier**

A linear discriminant function in D dimensions is given by a hyperplane, defined as follows:

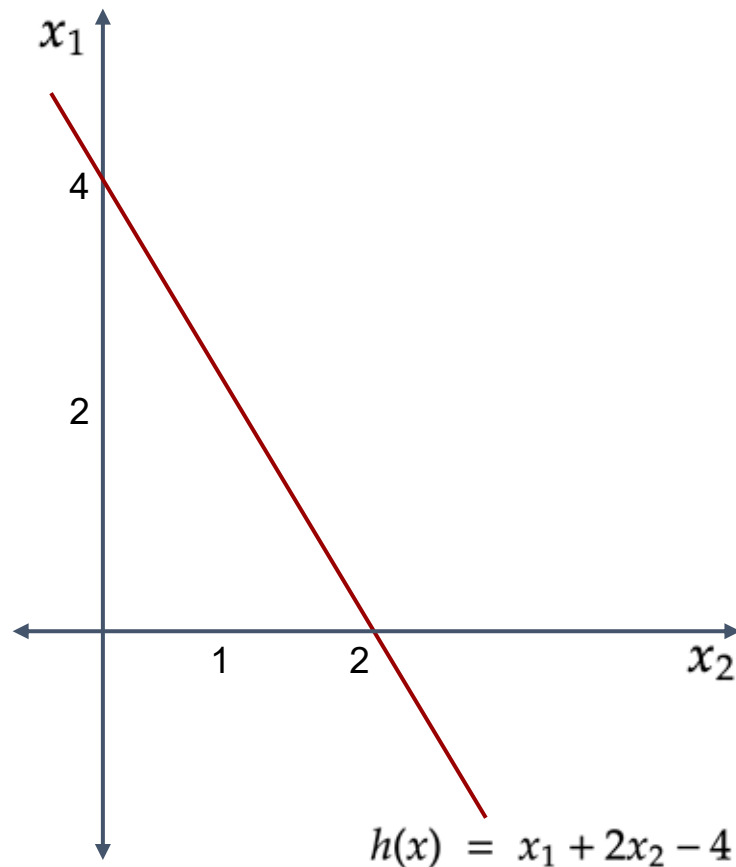
$$\begin{aligned}h(\mathbf{x}) &= \mathbf{w}^T \mathbf{x} + b \\ &= w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b\end{aligned}$$

For points that lie on the hyperplane, we have:

$$h(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b = 0$$

Example:

$$h(x) = x_1 + 2x_2 - 4$$

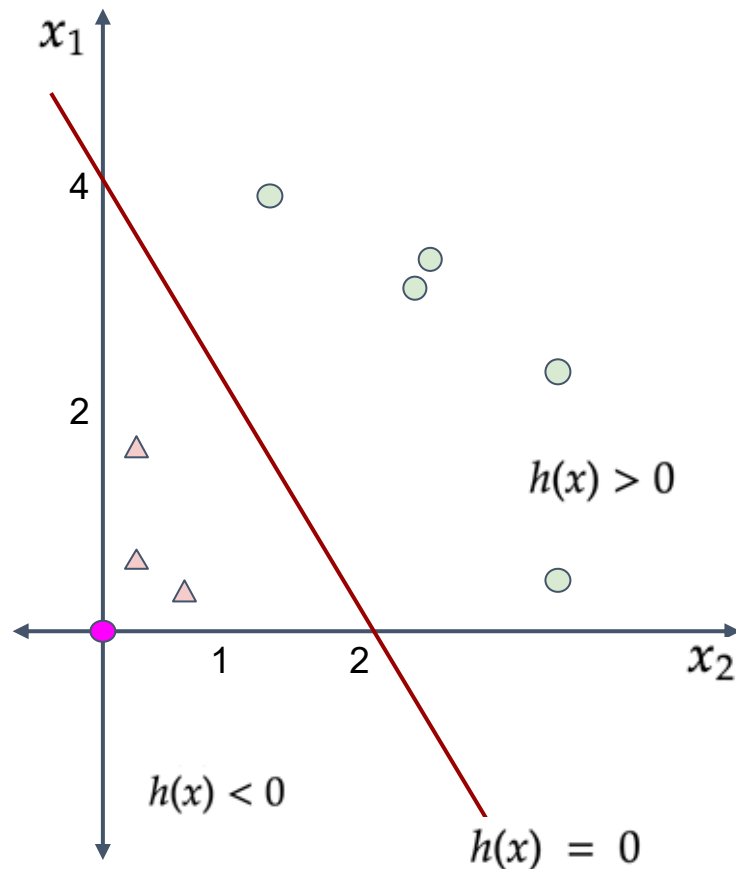


A hyperplane $h(x)$ splits the original d -dimensional space into two half-spaces.
If the input dataset is linearly separable:

$$y = \begin{cases} +1 & \text{if } h(\mathbf{x}) > 0 \\ -1 & \text{if } h(\mathbf{x}) < 0 \end{cases}$$

Q: what is the label for the origin?

$$h(x) = x_1 + 2 \cdot x_2 - 4 = 0 + 2 \cdot 0 - 4 = -4$$
$$y = -1$$



Separating Hyperplane: weight vector

7

Let \mathbf{a}_1 and \mathbf{a}_2 be two arbitrary points that lie on the hyperplane, we have:

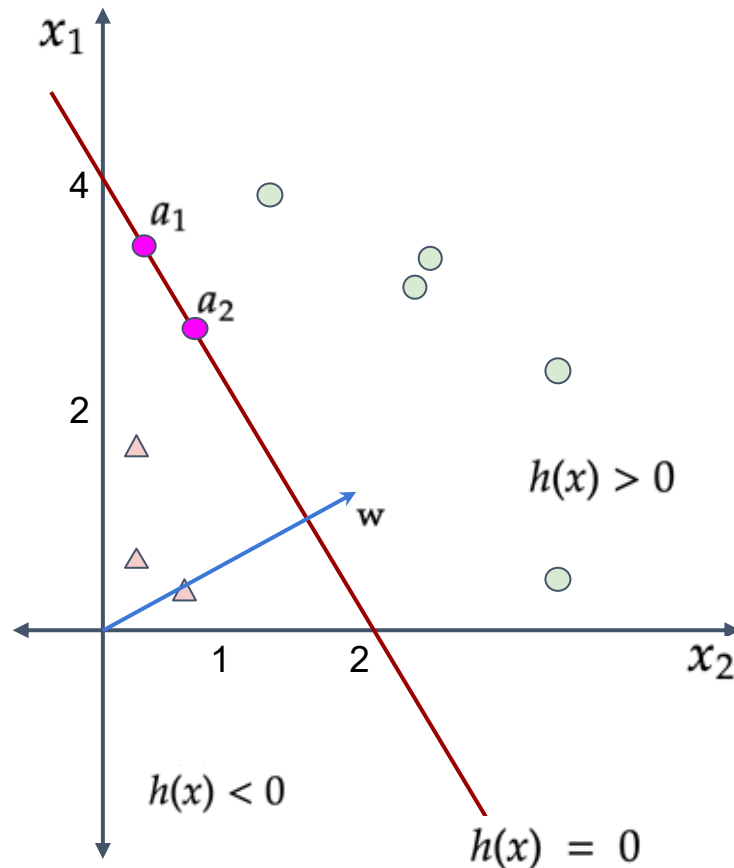
$$h(\mathbf{a}_1) = \mathbf{w}^T \mathbf{a}_1 + b = 0$$

$$h(\mathbf{a}_2) = \mathbf{w}^T \mathbf{a}_2 + b = 0$$

Subtracting one from the other:

$$\mathbf{w}^T (\mathbf{a}_1 - \mathbf{a}_2) = 0$$

The weight vector \mathbf{w} is orthogonal to the hyperplane.



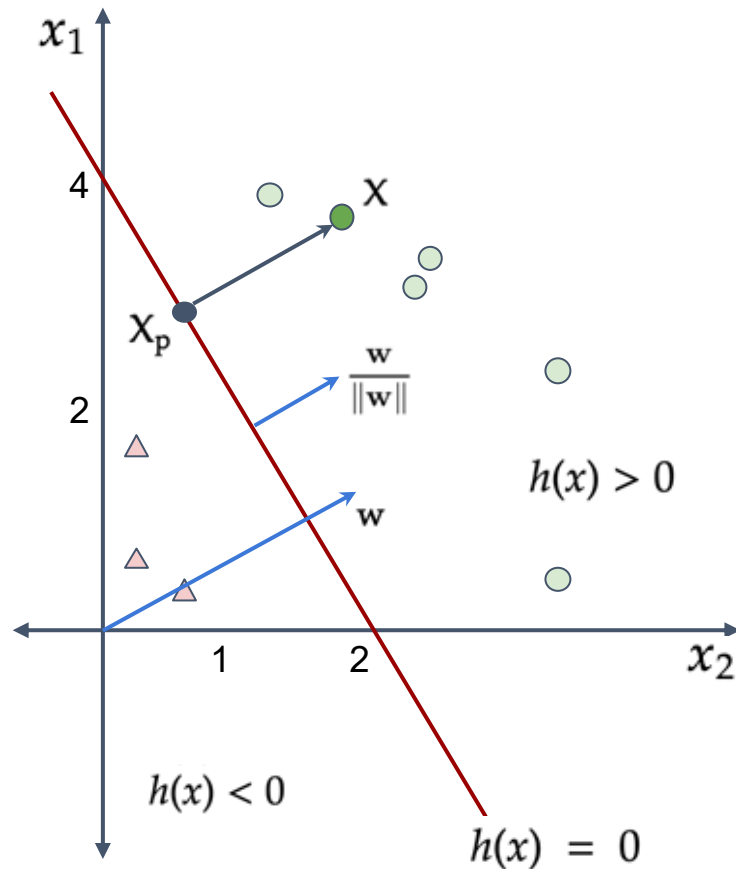
Distance of a Point to the Hyperplane

9

Consider a point X not on the hyperplane. Let X_p be the projection of X on the hyperplane.

Let r be the steps need to walk from X_p to X .

$$\begin{aligned}h(\mathbf{x}) &= h\left(\mathbf{x}_p + r \frac{\mathbf{w}}{\|\mathbf{w}\|}\right) \\&= \mathbf{w}^T \left(\mathbf{x}_p + r \frac{\mathbf{w}}{\|\mathbf{w}\|} \right) + b \\&= \underbrace{\mathbf{w}^T \mathbf{x}_p + b}_{h(\mathbf{x}_p)} + r \frac{\mathbf{w}^T \mathbf{w}}{\|\mathbf{w}\|} \\&= \underbrace{h(\mathbf{x}_p)}_0 + r \|\mathbf{w}\| \\&= r \|\mathbf{w}\| \qquad r = \frac{h(\mathbf{x})}{\|\mathbf{w}\|}\end{aligned}$$



Q: What is the direct distance from origin ($x=0$) to the hyperplane?

$$r = \frac{h(\mathbf{x})}{\|\mathbf{w}\|} \quad r = \frac{h(\mathbf{0})}{\|\mathbf{w}\|} = \frac{\mathbf{w}^T \mathbf{0} + b}{\|\mathbf{w}\|} = \frac{b}{\|\mathbf{w}\|}$$

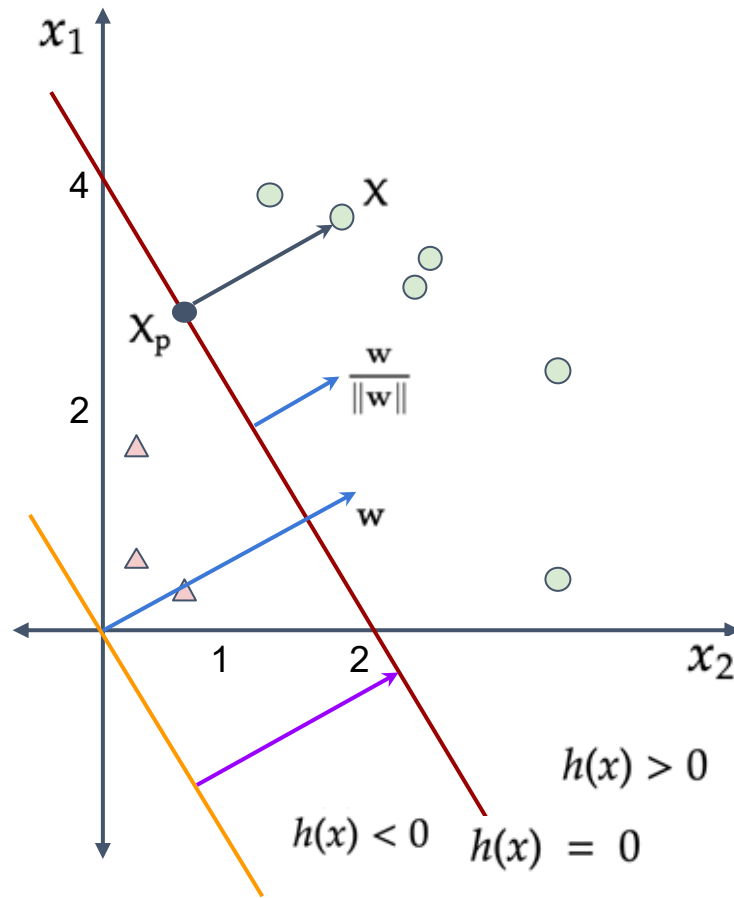
Example:

$$h(x) = x_1 + 2x_2 - 4$$

$$\mathbf{w}^T \mathbf{x} + b = (1 \ 2) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 4$$

$$\frac{b}{\|\mathbf{w}\|} = -\frac{4}{\sqrt{5}}$$

Q: how to deal with negative distance?



Q: How to deal with negative distance?

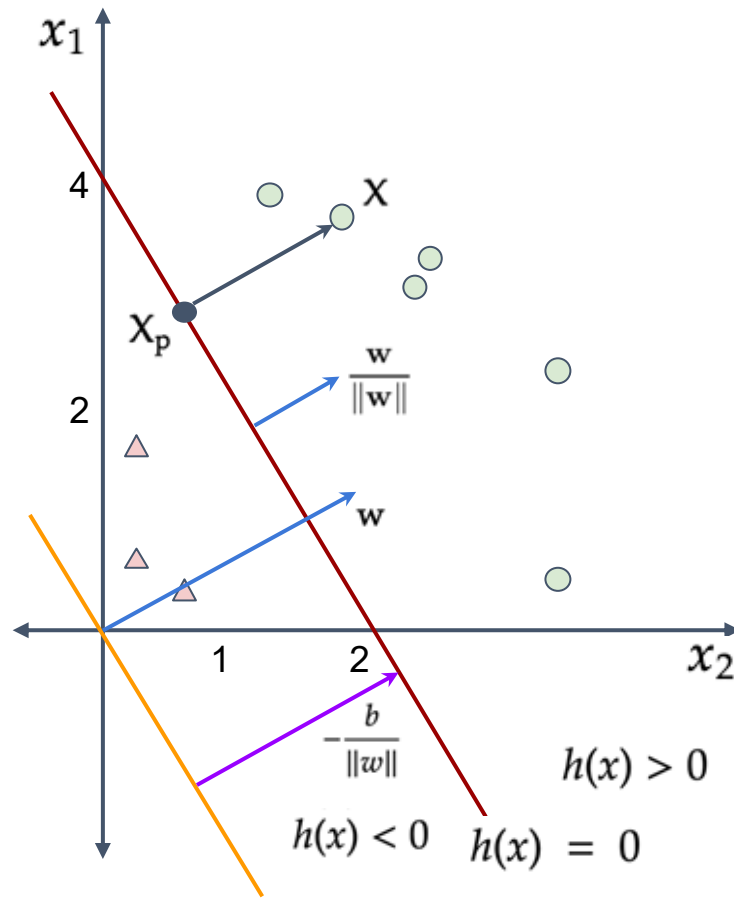
$$r = \frac{h(\mathbf{x})}{\|\mathbf{w}\|}$$

$$y = \begin{cases} +1 & \text{if } h(\mathbf{x}) > 0 \\ -1 & \text{if } h(\mathbf{x}) < 0 \end{cases}$$

$$\delta = y r = \frac{y h(\mathbf{x})}{\|\mathbf{w}\|}$$

Example (when point is the origin):

$$(-1) \cdot \frac{b}{\|\mathbf{w}\|} = \frac{4}{\sqrt{5}}$$

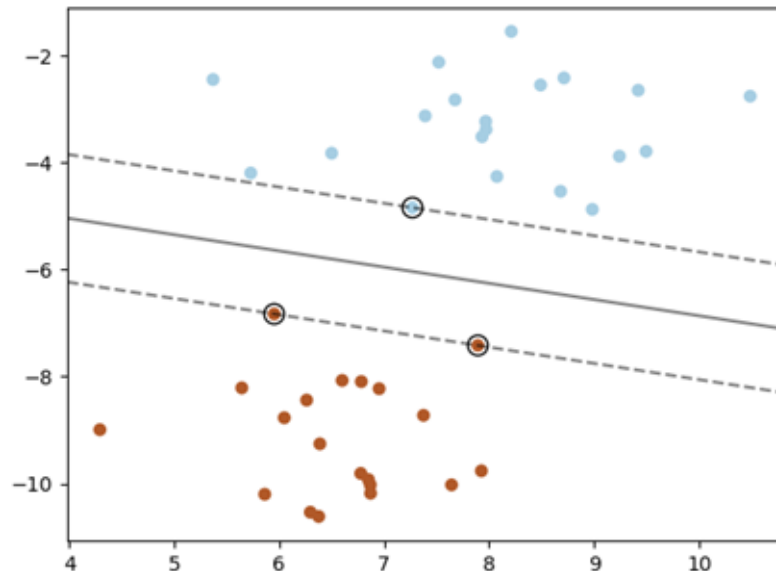


$$\delta = y r = \frac{y h(\mathbf{x})}{\|\mathbf{w}\|}$$

Over all the n points, the **margin** of the linear classifier is the minimum distance of a point from the separating hyperplane:

$$\delta^* = \min_{\mathbf{x}_i} \left\{ \frac{y_i (\mathbf{w}^T \mathbf{x}_i + b)}{\|\mathbf{w}\|} \right\}$$

All the points that achieve this minimum distance are called **support vectors**.



For training data $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$, a classifier $f(x) = w^T x + b$ with 0 train error will satisfy

$$y^{(i)} f(x^{(i)}) = y^{(i)} (w^T x^{(i)} + b) > 0$$

↓ negative margin when misclassifying it!

The distance for $(x^{(i)}, y^{(i)})$ to separating hyperplane

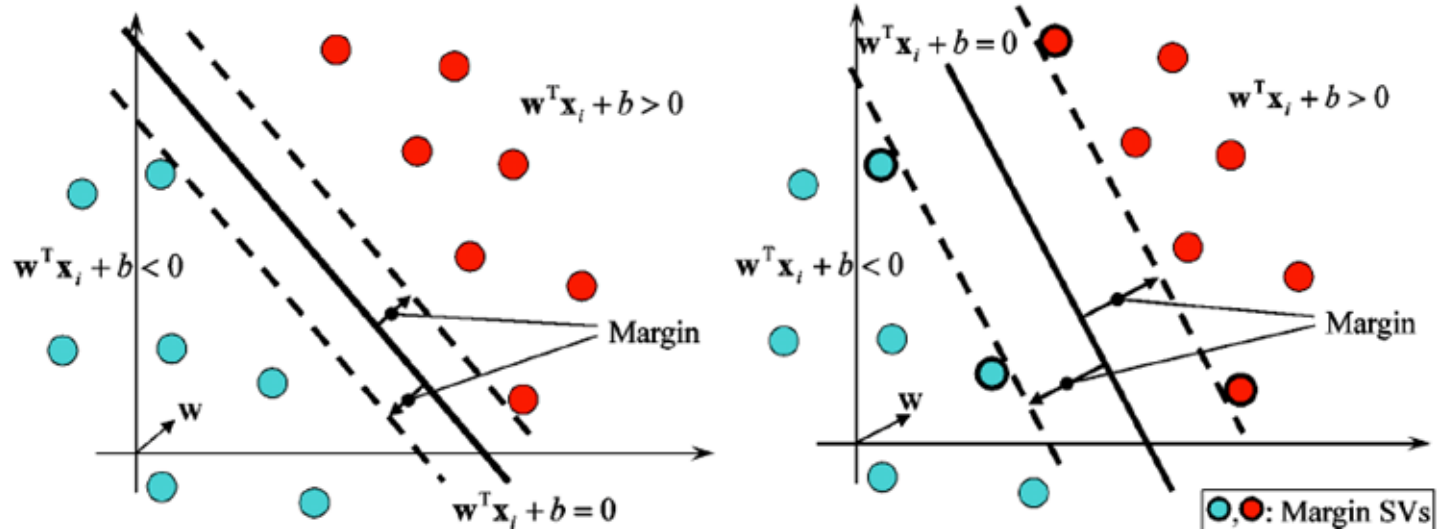
$$\frac{y^{(i)} (w^T x^{(i)} + b)}{\|w\|}$$

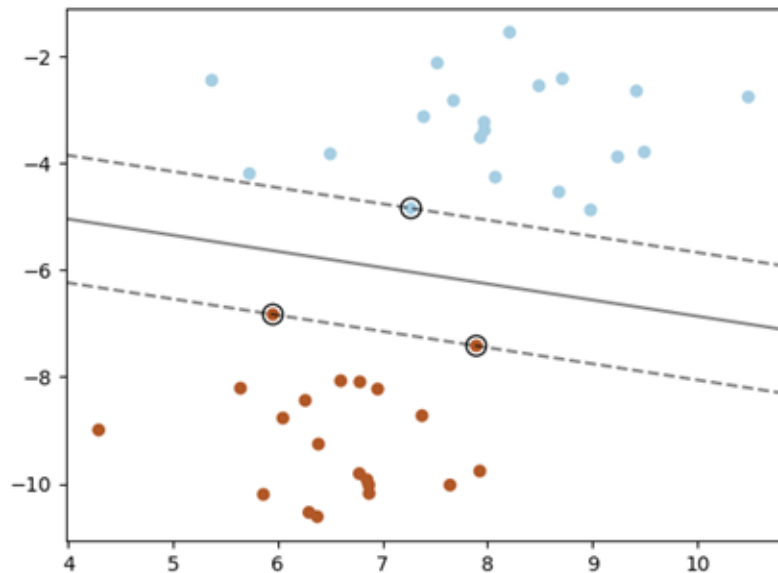
The margin of a classifier $f(x)$ is

$$\min_i \frac{y^{(i)} (w^T x^{(i)} + b)}{\|w\|}$$

Find f that maximize margin

$$\arg \max_{w,b} \min_i \frac{y^{(i)} (w^T x^{(i)} + b)}{\|w\|}$$





Maximize the minimum margin

$$\arg \max_{w,b} \min_i \frac{y^{(i)} (w^\top x^{(i)} + b)}{\|w\|}$$

Minimum margin over all training data

Find the parameters (w,b) that **maximize** the **smallest margin** over all the training data

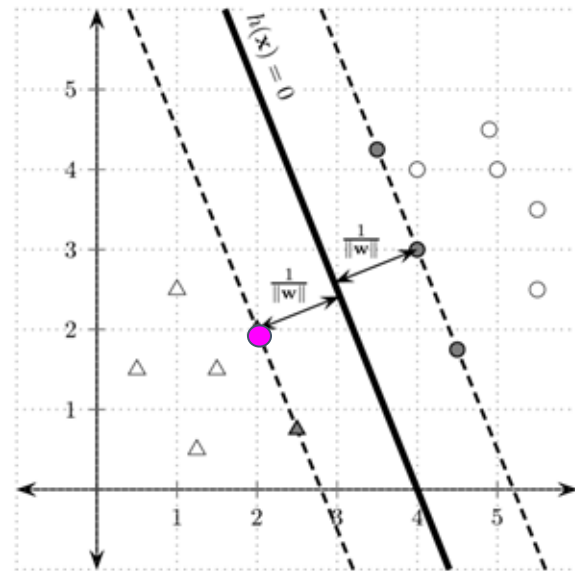
$$\arg \max_{w,b} \min_i \boxed{y^{(i)}(w^\top x^{(i)} + b)} \longrightarrow 1 \text{ for support vectors}$$

Issue: infinite equivalent hyperplanes result in infinite solutions

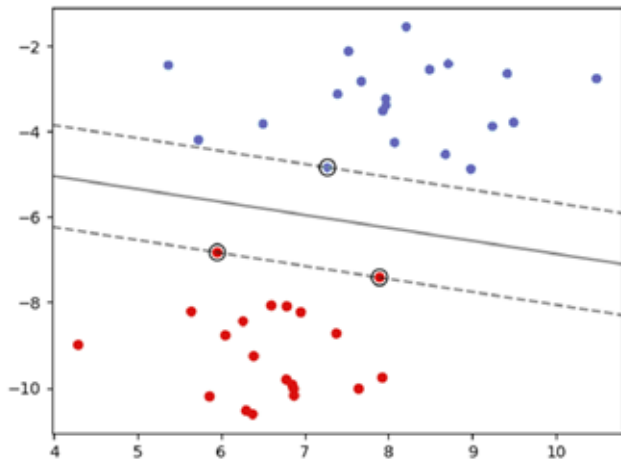
Solution: choose the scalar s such that the absolute distance of a **support vector** from the hyperplane is $\frac{1}{\|w\|}$

Margin: $\delta^* = \frac{1}{\|w\|}$

Max margin: $h^* = \arg \max_h \{\delta_h^*\} = \arg \max_{w,b} \left\{ \frac{1}{\|w\|} \right\}$



... it leads to



$$\min_{w,b} \frac{1}{2} \|w\|^2$$

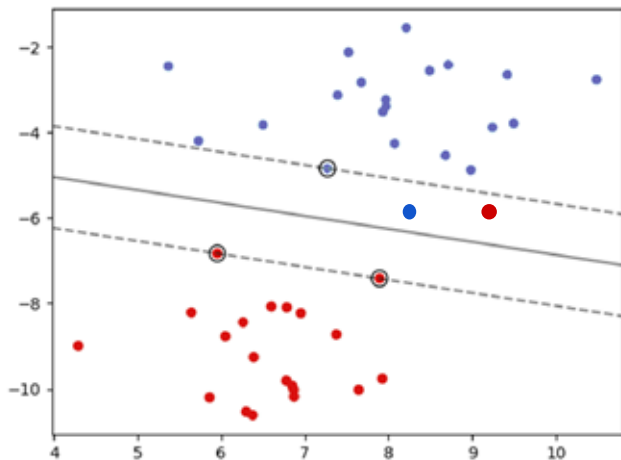
subject to

$$y^{(i)} (w^\top x^{(i)} + b) \geq 1 \quad \text{for } i = 1, \dots, m$$

This is a convex (quadratic) optimization problem that can be solved efficiently

- Data are D -dimensional *vectors*
- Margins determined by nearest data points called **support vectors**
- We call this a *support vector machine* (SVM)

If the data is linearly not separable,



regularization

error

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \cdot \sum_{i=1}^m \xi_i$$

subject to

$$y^{(i)}(w^\top x^{(i)} + b) \geq 1 - \xi_i$$

$$\xi_i \geq 0 \quad \text{for } i = 1, \dots, m$$

Equivalent formulation

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m (1 - y^{(i)}(w^\top x^{(i)} + b))_+$$

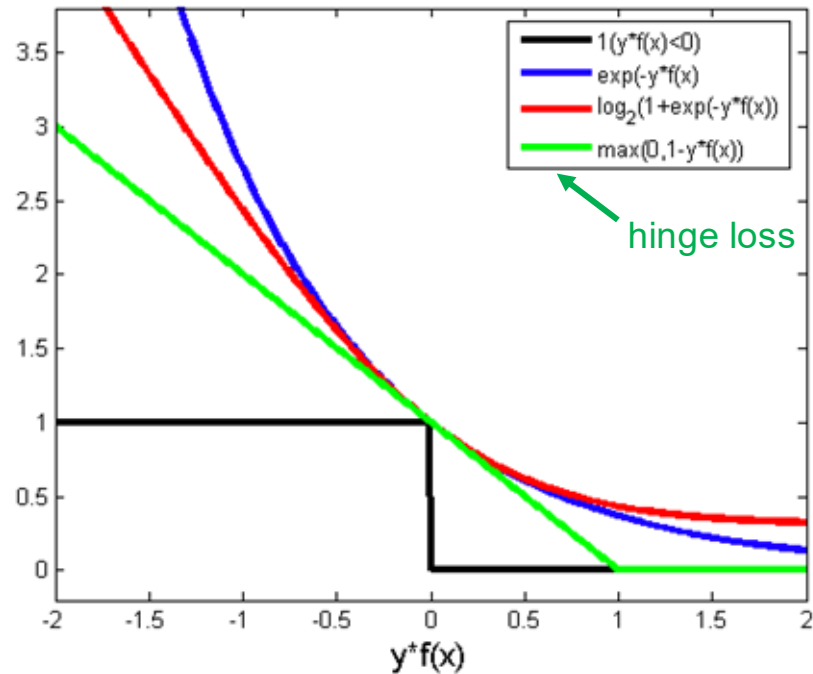
SVM - Soft Margin: hinge loss

Equivalent formulation

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m (1 - y^{(i)}(w^\top x^{(i)} + b))_+$$

$$\ell(f; x^{(i)}, y^{(i)}) = (1 - y^{(i)} f(x^{(i)}))_+$$

$$(X)_+ := \max(X, 0)$$



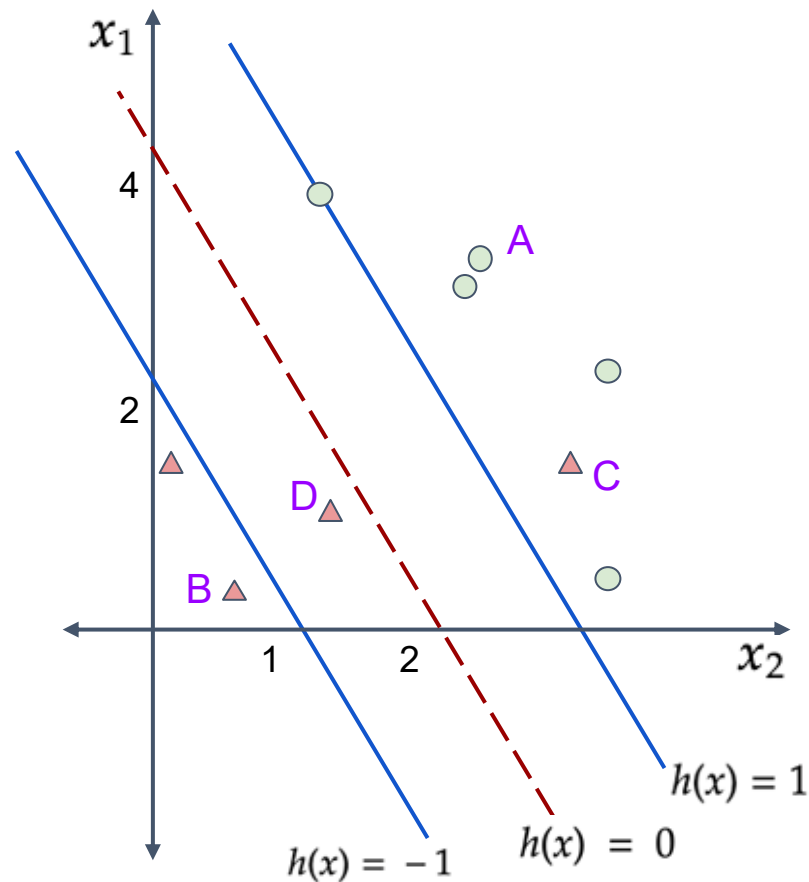
$$\text{Hinge loss} = \max(0, 1 - y_i(w^T x_i + b))$$

$$A: \max(0, 1 - 1 \cdot (> 1)) \rightarrow 0$$

$$B: \max(0, 1 - (-1) \cdot (< -1)) \rightarrow 0$$

$$C: \max(0, 1 - (-1) \cdot (> 1)) \rightarrow > 1$$

$$D: \max(0, 1 - (-1) \cdot (\text{between } [-1, 0])) \\ \rightarrow \text{between } [0, 1]$$



SVM with linear decision boundaries,

`sklearn.svm.LinearSVC`

Call options include...

penalty : {'l1', 'l2'}, default='l2'

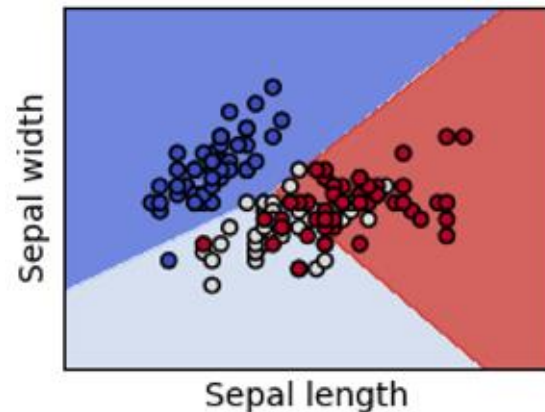
Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to `coef_` vectors that are sparse.

dual : bool, default=True

Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when `n_samples > n_features`.

C : float, default=1.0

Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.



kernel : {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

gamma : {'scale', 'auto'} or float, default='scale'

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses $1 / (n_features * X.var())$ as value of gamma,
- if 'auto', uses $1 / n_features$.

for RBF,

small γ : complex decision boundary

large γ : more like linear decision boundary

max_iter : int, default=-1

Hard limit on iterations within solver, or -1 for no limit.

verbose : bool, default=False

Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

class_weight : dict or 'balanced', default=None

Set the parameter C of class i to `class_weight[i]*C` for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Classify among 3 species of Iris flowers...



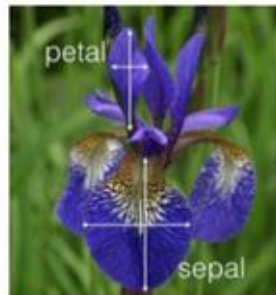
Iris setosa



Iris versicolor

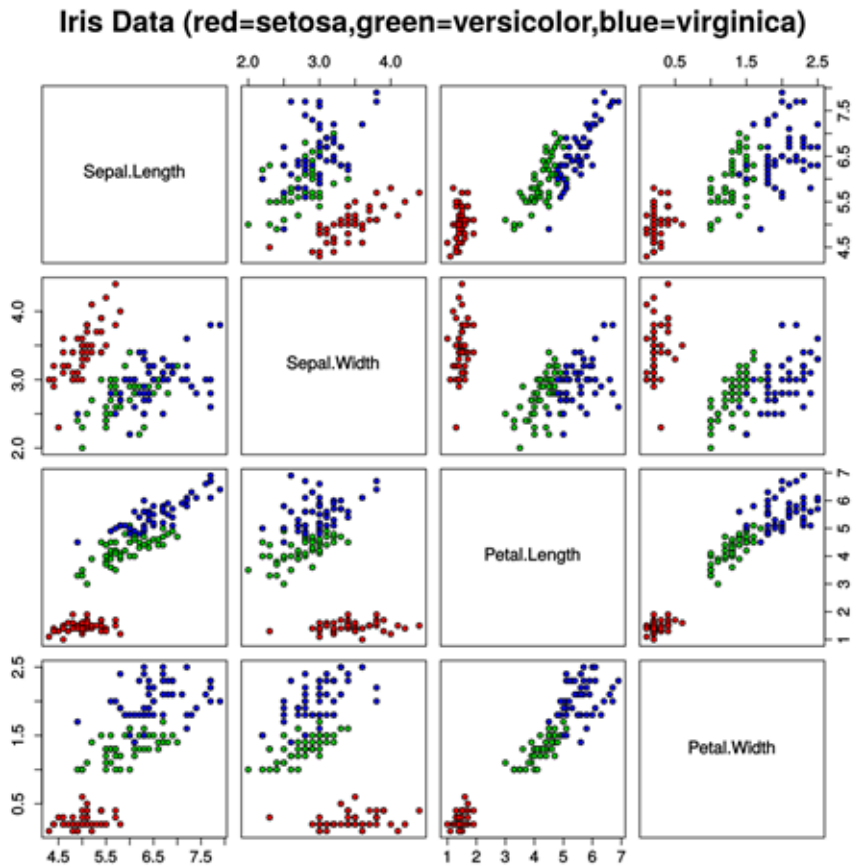


Iris virginica



Four features (in centimeters)

- Petal length / width
- Sepal length / width



*Fairly easy to separate **setosa** from others using a linear classifier*

Need to use nonlinear basis / kernel representation to better separate other classes

Train 8-degree polynomial kernel SVM classifier,

```
from sklearn.svm import SVC
svclassifier = SVC(kernel='poly', degree=8)
svclassifier.fit(X_train, y_train)
```

Generate predictions on held-out test data,

```
y_pred = svclassifier.predict(X_test)
```

Show confusion matrix and classification accuracy,

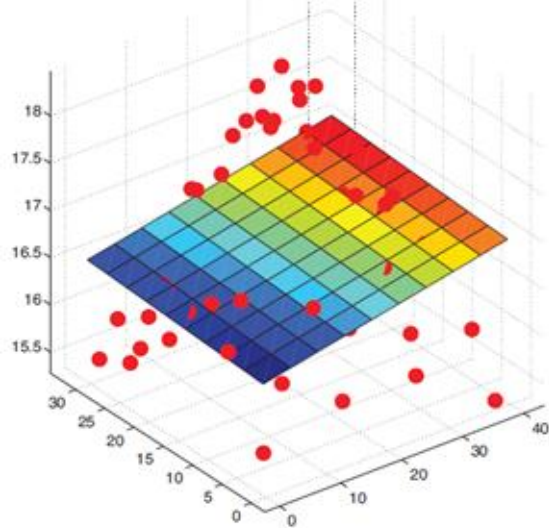
```
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[11  0  0]
 [ 0 12  1]
 [ 0  0  6]]
```

	precision	recall	f1-score	support
Iris-setosa	1.00	1.00	1.00	11
Iris-versicolor	1.00	0.92	0.96	13
Iris-virginica	0.86	1.00	0.92	6
avg / total	0.97	0.97	0.97	30

- Support Vector Machine
- **Basis Functions**

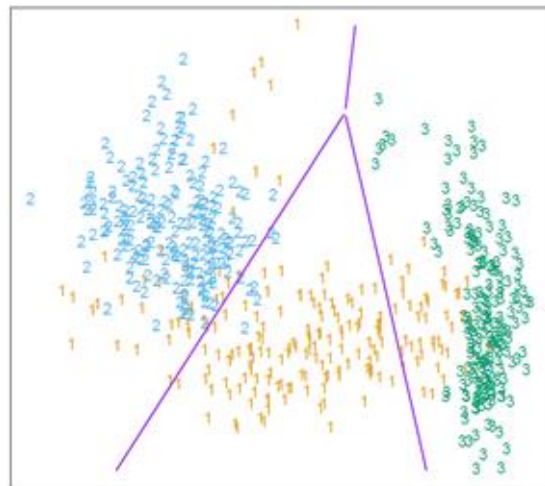
[Image: Murphy, K. (2012)]



Linear Regression Fit a *linear function* to the data,

$$y = w^T x$$

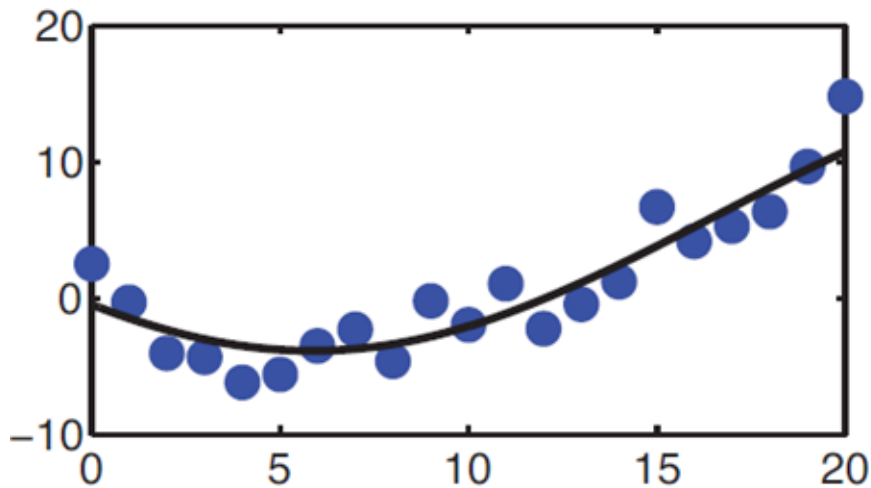
[Image: Hastie et al. (2001)]



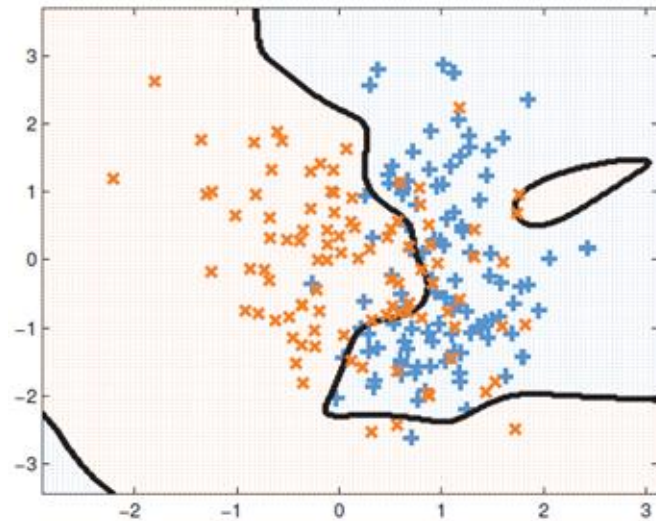
Logistic Regression Learn a decision boundary that is *linear in the data*,

For two classes: $p = \frac{1}{1+e^{w^T x + b}} = 0.5$

$$y = \mathbf{I}\{w^T x \geq 0\}$$



What if our data are *not* well-described by a linear function?



What if classes cannot be well-distinguished by a linear function?

Suppose that we want to predict the number of earthquakes that occur of a certain magnitude.

FIGURE 5-3A: WORLDWIDE EARTHQUAKE FREQUENCIES, JANUARY 1964–MARCH 2012

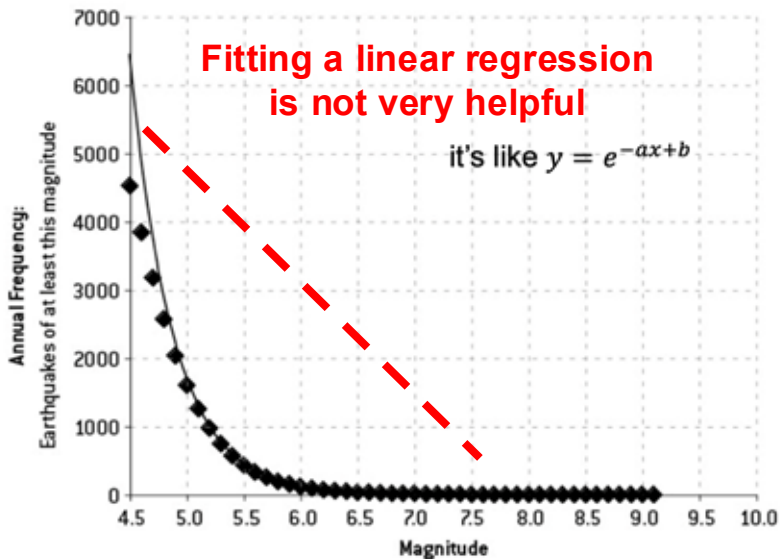
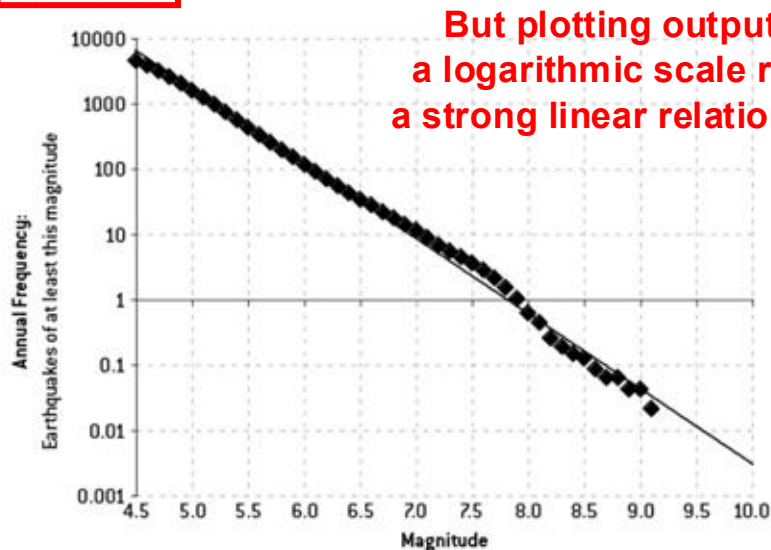


FIGURE 5-3B: WORLDWIDE EARTHQUAKE FREQUENCIES, JANUARY 1964–MARCH 2012,

LOGARITHMIC SCALE



- Recall: for 1d problem, we embedded the feature: $x' = (x, 1) \in \mathbb{R}^2$ so we can encode the intercept term.

$$\phi_0(x) = 1 \quad \phi_1(x) = x \quad y = \mathbf{w}^\top \Phi_{\text{lin}}(x) = \phi_0(x)w_0 + \phi_1(x)w_1 = w_0 + w_1x$$

- Recall: for 1d problem, we embedded the feature: $x' = (x, 1) \in \mathbb{R}^2$ so we can encode the intercept term.

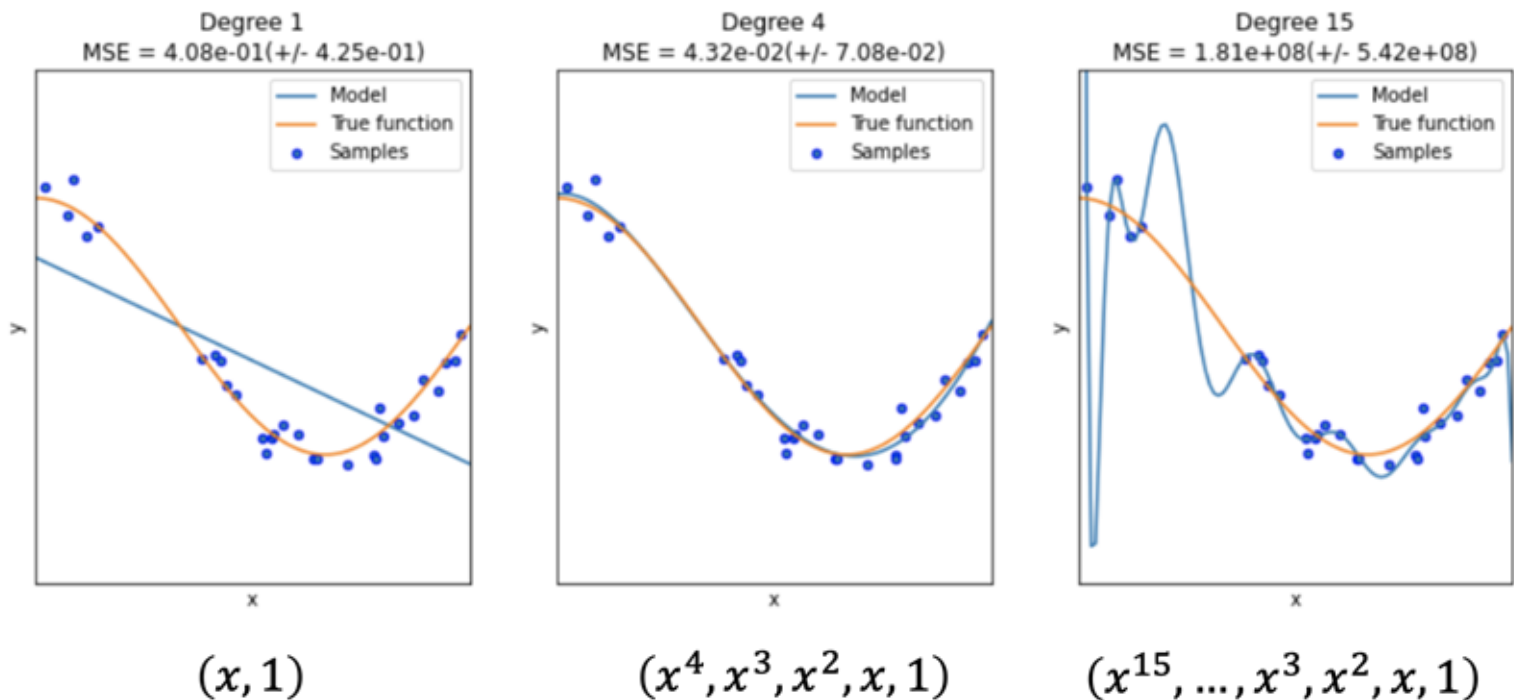
$$\phi_0(x) = 1 \quad \phi_1(x) = x \quad y = \mathbf{w}^\top \Phi_{\text{lin}}(x) = \phi_0(x)w_0 + \phi_1(x)w_1 = w_0 + w_1x$$

- Actually, the embedding trick is stronger.

- $(x^2, x, 1)$: 2nd order polynomial with respect to x
- $(x^d, x^{d-1}, \dots, 1)$: d -th order polynomial (= degree d)

$$\phi_0(x) = 1 \quad \phi_1(x) = x \quad \phi_2(x) = x^2$$

$$y = \mathbf{w}^\top \Phi_{\text{lin}}(x) = \phi_0(x)w_0 + \phi_1(x)w_1 + \phi_2(x)w_2 = w_0 + w_1x + w_2x^2$$



higher-order polynomial = higher complexity = prone to overfitting!

- A **basis function** can be any function of the input features **X**
- Define a set of B basis functions $\phi_1(x), \dots, \phi_B(x)$
- Fit a linear regression model in terms of basis functions,

$$y = \sum_{b=1}^B w_b \phi_b(x) = w^T \phi(x)$$

notation:
 $\phi(x) := [\phi_1(x), \dots, \phi_B(x)]$

- The model is *linear* in the transformed basis/induced features $\phi(x)$.
- The model is *nonlinear* in the data **X**

Example

- Two features: $x = (x_1, x_2)$
- Quadratic features: $\phi_1(x) = 1, \phi_2(x) = x_1, \phi_3(x) = x_2, \phi_4(x) = x_1^2, \phi_5(x) = x_2^2, \phi_6(x) = x_1 \cdot x_2$
- For an input: $x = (2, 3)$
- $\phi(x) = (1, 2, 3, 4, 9, 6)$
- $y = w^T \phi(x) = w_1 \times 1 + w_2 \times 2 + w_3 \times 3 + w_4 \times 4 + w_5 \times 9 + w_6 \times 6$

sklearn.preprocessing.PolynomialFeatures

degree : int or tuple (min_degree, max_degree), default=2

If a single int is given, it specifies the maximal degree of the polynomial features. If a tuple (min_degree, max_degree) is passed, then min_degree is the minimum and max_degree is the maximum polynomial degree of the generated features. Note that min_degree=0 and min_degree=1 are equivalent as outputting the degree zero term is determined by include_bias.

interaction_only : bool, default=False

If True, only interaction features are produced: features that are products of at most degree distinct input features, i.e. terms with power of 2 or higher of the same input feature are excluded:

- included: $x[0]$, $x[1]$, $x[0] * x[1]$, etc.
- excluded: $x[0] ** 2$, $x[0] ** 2 * x[1]$, etc.

include_bias : bool, default=True

If True (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).

order : {'C', 'F'}, default='C'

Order of output array in the dense case. 'F' order is faster to compute, but may slow down subsequent estimators.

Create three two-dimensional data points [0,1], [2,3], [4,5]:

```
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Compute quadratic features $(1, x_1, x_2, x_1^2, x_1x_2, x_2^2)$,

```
>>> poly = PolynomialFeatures(degree=2)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

These are now our new data and ready to fit a model...

Create a 3-rd order polynomial (cubic) function,

```
f = lambda x: (x-1)*(x-2)*(x-3)
import numpy.random as ra
ra.seed(20)
train_x = np.arange(5)
train_y = f(train_x) + 1*ra.randn(len(train_x))
train_y
```

✓ 0.3s

```
array([-5.11610689,  0.19586502,  0.35753652, -2.34326191,  4.91516741])
```

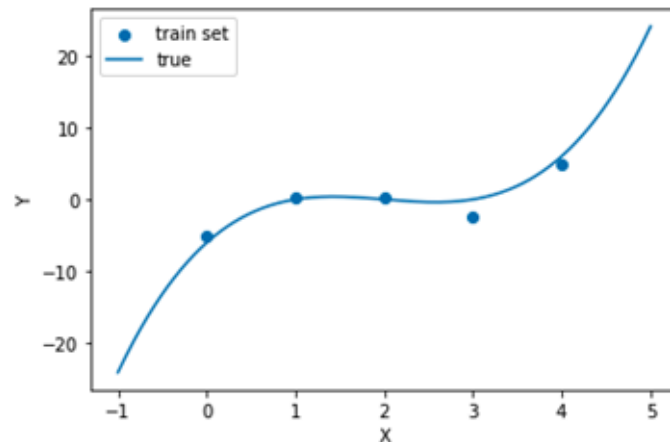
Plot train set and the actual function

```
test_x = np.linspace(-1,5,400)

from matplotlib import pyplot as plt
plt.scatter(train_x,train_y)

plt.plot(test_x, f(test_x))
plt.legend(['train set', 'true'])
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

✓ 0.4s



Create cubic features $(1, x, x^2, x^3)$

```
poly = PolynomialFeatures(degree=3)
train_xx = poly.fit_transform(train_x[:,np.newaxis])
train_xx
```

✓ 0.4s turns train_x (length 5 array) into a matrix (5 by 4 matrix)

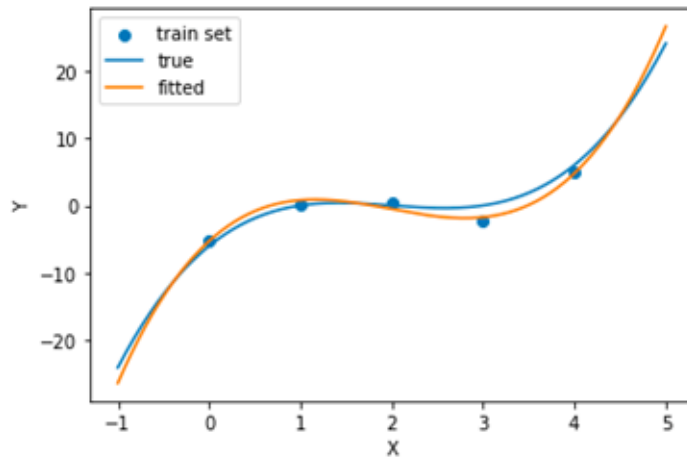
```
array([[ 1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  2.,  4.,  8.],
       [ 1.,  3.,  9., 27.],
       [ 1.,  4., 16., 64.]])
```

Perform linear regression; plot it

```
from matplotlib import pyplot as plt
from sklearn.linear_model import LinearRegression
model = LinearRegression().fit(train_xx, train_y)
test_x = np.linspace(-1,5,400)
test_xx = poly.fit_transform(test_x[:,np.newaxis])
pred_y = model.predict(test_xx)
```

```
plt.scatter(train_x, train_y)
plt.plot(test_x, f(test_x))
plt.plot(test_x, pred_y)
plt.legend(['train set', 'true', 'fitted'])
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

✓ 0.2s



- Generally the first step in data science involves *preprocessing* or transforming data in some way
 - Filling in missing values (imputation)
 - Centering / normalizing / standardizing
 - Etc.
- We then fit our models to this preprocessed data
- One way to view preprocessing is simply as computing some basis function $\phi(x)$, nothing more

PROs

- More flexible modeling that is nonlinear in the original data
- Increases model expressivity

CONs

- Typically requires **more parameters** to be learned
- More sensitive to **overfitting** training data (due to expressivity)
- Requires more **regularization** to avoid overfitting
- Need to find *good* basis functions (feature engineering)