



# Applied Machine Learning

## Supervised machine learning (Part 1)

Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

# Learning objectives

- Understand how a number of different supervised learning algorithms learn by estimating their parameters from data to make new predictions.
- Understand the strengths and weaknesses of particular supervised learning methods.



# Learning objectives

- Learn how to apply specific supervised machine learning algorithms in Python with scikit-learn.
- Learn about general principles of supervised machine learning, like overfitting and how to avoid it.

# Review of important terms

- Feature representation
- Data instances/samples/examples (X)
- Target value (y)

fruits							
	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	0.60
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79
5	2	mandarin	mandarin	80	5.8	4.3	0.77
6	2	mandarin	mandarin	80	5.9	4.3	0.81
7	2	mandarin	mandarin	76	5.8	4.0	0.81
8	1	apple	braeburn	178	7.1	7.8	0.92
9	1	apple	braeburn	172	7.4	7.0	0.89
10	1	apple	braeburn	166	6.9	7.3	0.93
11	1	apple	braeburn	172	7.1	7.6	0.92
12	1	apple	braeburn	154	7.0	7.1	0.88
13	1	apple	golden_delicious	164	7.3	7.7	0.70
14	1	apple	golden_delicious	152	7.6	7.3	0.69
15	1	apple	golden_delicious	156	7.7	7.1	0.69
16	1	apple	golden_delicious	156	7.6	7.5	0.67
17	1	apple	golden_delicious	168	7.5	7.6	0.73
18	1	apple	cripps_pink	162	7.5	7.1	0.83

# Review of important terms

- **Training and test sets**
- **Model/Estimator**
  - *Model fitting* produces a 'trained model'.
  - *Training is the process of estimating model parameters.*
- **Evaluation method**

```
%matplotlib notebook
import numpy as np
import pandas as pd
import seaborn as sn
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

fruits = pd.read_table('fruit_data_with_colors.txt')

X = fruits[['height', 'width', 'mass', 'color_score']]
y = fruits['fruit_label']

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
_____

knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train, y_train)
print("Accuracy of K-NN classifier on test set: ", knn.score(X_test, y_test))

example_fruit = [[5.5, 2.2, 10, 0.70]]
print("Predicted fruit type for ", example_fruit, " is ", knn.predict(example_fruit))
_____
```



# Applied Machine Learning

## Overfitting and Underfitting

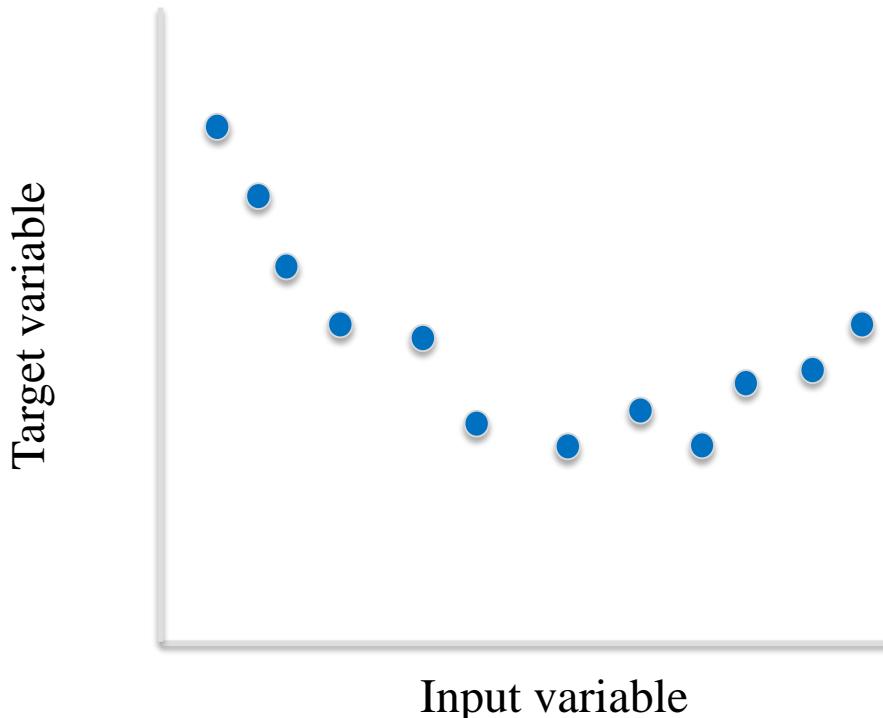
Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

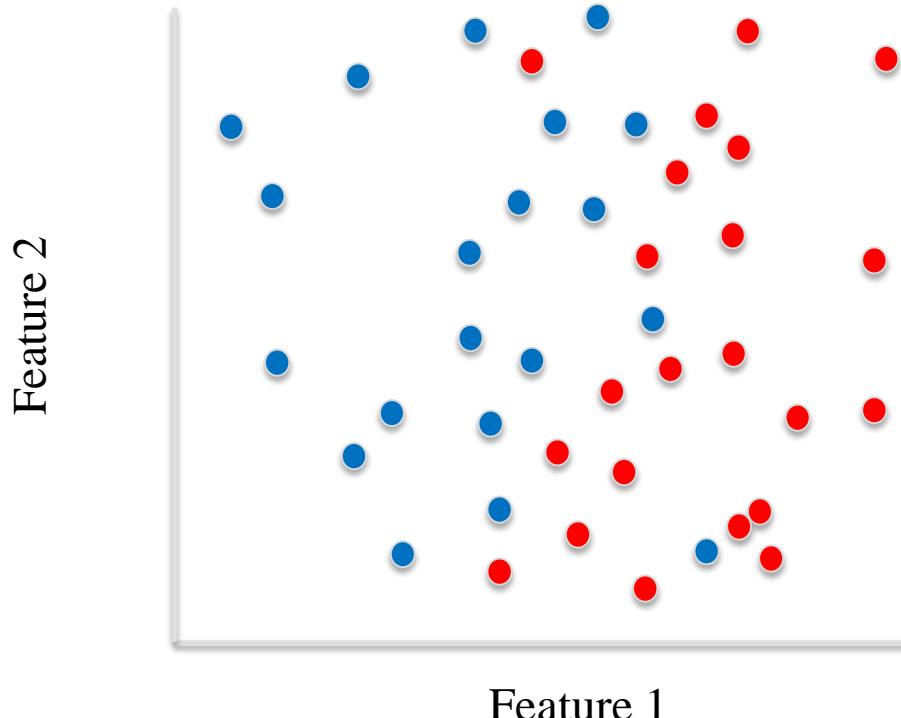
# Generalization, Overfitting, and Underfitting

- **Generalization ability refers to an algorithm's ability to give accurate predictions for new, previously unseen data.**
- Assumptions:
  - Future unseen data (test set) will have the same properties as the current training sets.
  - Thus, models that are accurate on the training set are expected to be accurate on the test set.
  - But that may not happen if the trained model is tuned too specifically to the training set.
- Models that are too complex for the amount of training data available are said to overfit and are not likely to generalize well to new examples.
- Models that are too simple, that don't even do well on the training data, are said to underfit and also not likely to generalize well.

# Overfitting in regression

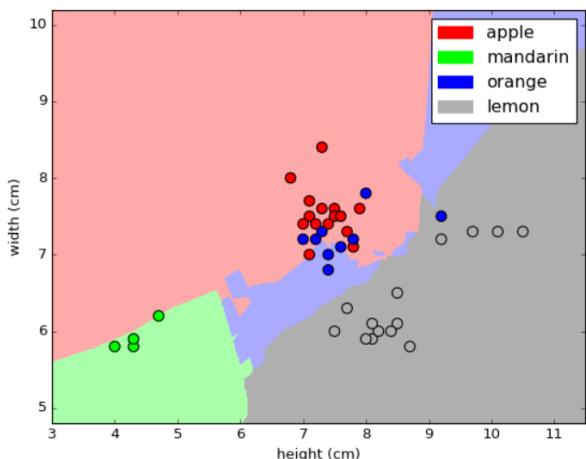


# Overfitting in classification

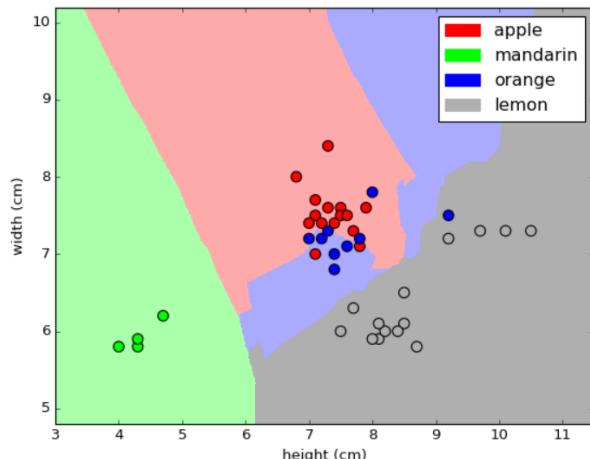


# Overfitting with k-NN classifiers

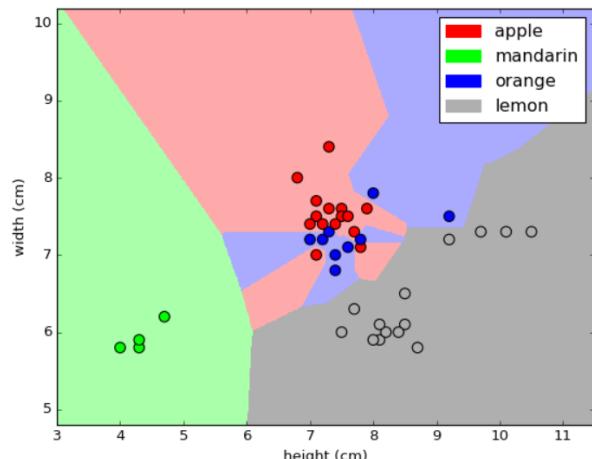
decrease k, overfitting



$K=10$



$K=5$



$K=1$



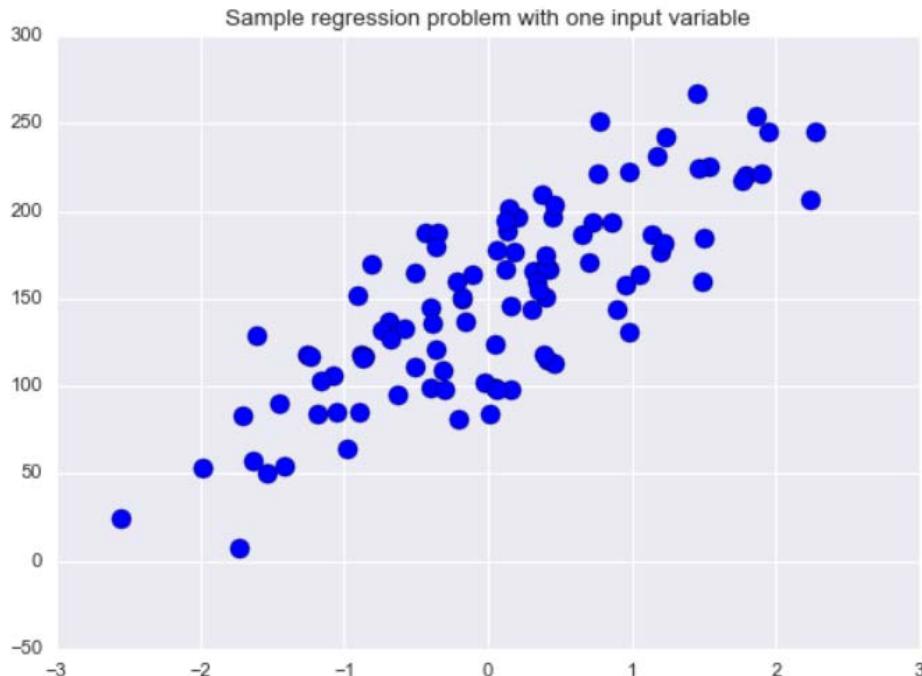
# Applied Machine Learning

## Supervised learning: Datasets

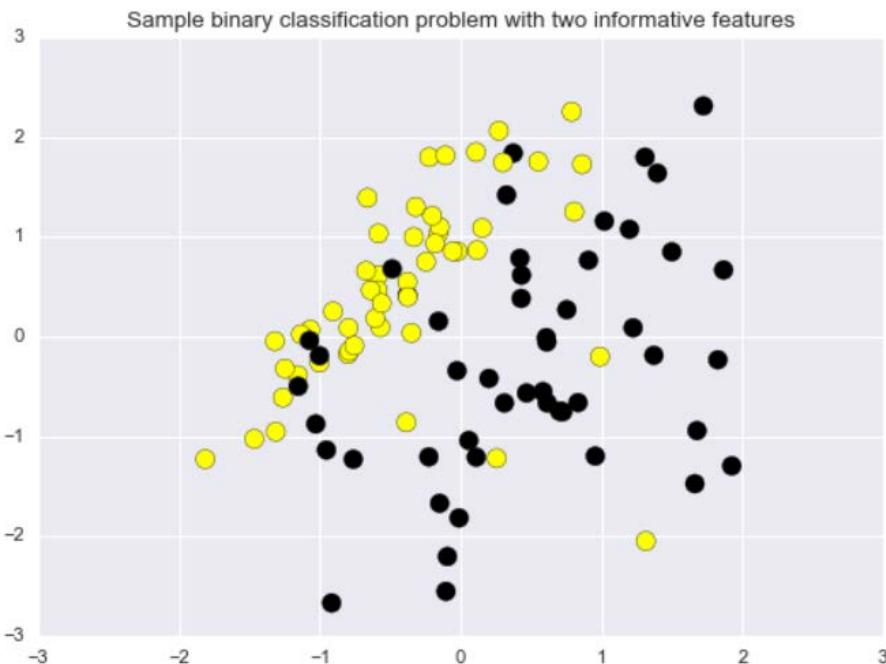
Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

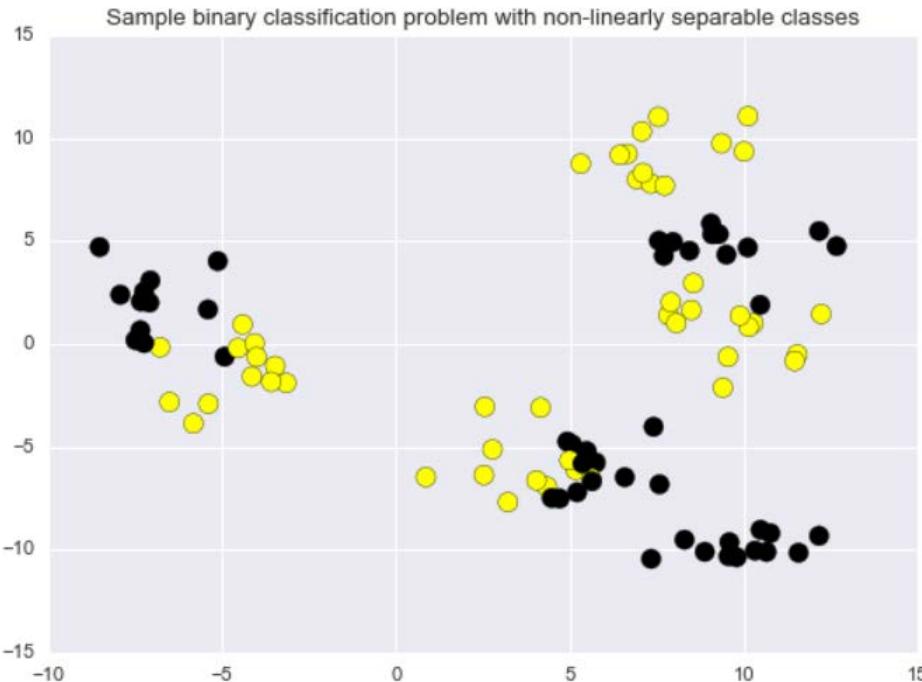
# Simple Regression Dataset



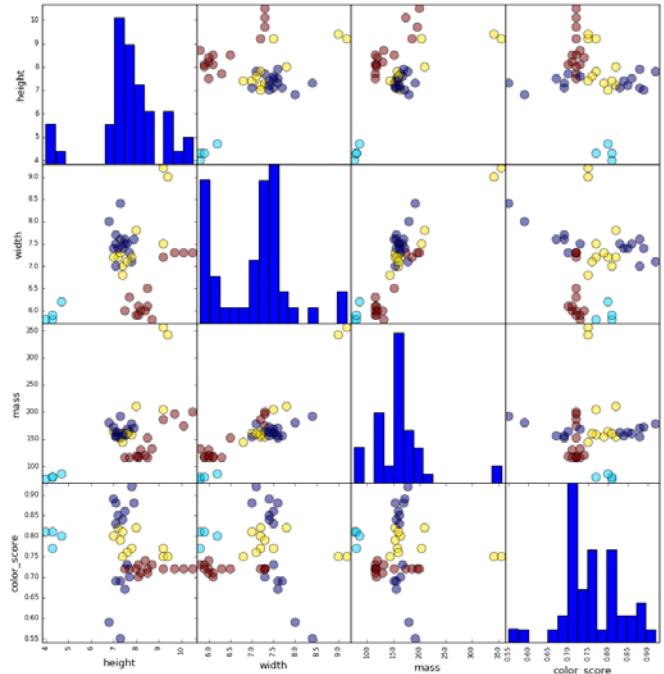
# Simple Binary Classification Dataset



# Complex Binary Classification Dataset



# Fruits Multi-class Classification Dataset



## Features

- `width`
- `height`
- `mass`
- `color_index`

## Classes

- 0: apple
- 1: mandarin orange
- 2: orange
- 3: lemon

# Communities and Crime Dataset

	population	householdsize	agePct12t21	agePct12t29	agePct16t24	agePct65up	numbUrban	pctUrban	medIncome	pctWWage	...	MedRentF
0	11980	3.10	12.47	21.44	10.93	11.33	11980	100.0	75122	89.24	...	23.8
1	23123	2.82	11.01	21.30	10.48	17.18	23123	100.0	47917	78.99	...	27.6
2	29344	2.43	11.36	25.88	11.01	10.28	29344	100.0	35669	82.00	...	24.1
3	16656	2.40	12.55	25.20	12.19	17.57	0	0.0	20580	68.15	...	28.7
5	140494	2.45	18.09	32.89	20.04	13.26	140494	100.0	21577	75.78	...	26.4
6	28700	2.60	11.17	27.41	12.76	14.42	28700	100.0	42805	79.47	...	24.4
7	59459	2.45	15.31	27.93	14.78	14.60	59449	100.0	23221	71.60	...	26.3
8	74111	2.46	16.64	35.16	20.33	8.58	74115	100.0	25326	83.69	...	25.2
9	103590	2.62	19.88	34.55	21.62	13.12	103590	100.0	17852	74.20	...	29.6
10	31601	2.54	15.73	28.57	15.16	14.26	31596	100.0	24763	73.92	...	23.8

Input features:  
socio-economic data by location  
from U.S. Census

Target variable:  
Per capita violent crimes

Derived from the original UCI dataset at:

<https://archive.ics.uci.edu/ml/datasets/Communities+and+Crime+Unnormalized>

```
from adspy_shared_utilities import load_crime_dataset
crime = load_crime_dataset()
```



# Applied Machine Learning

## K-Nearest Neighbors: Classification and Regression

Kevyn Collins-Thompson

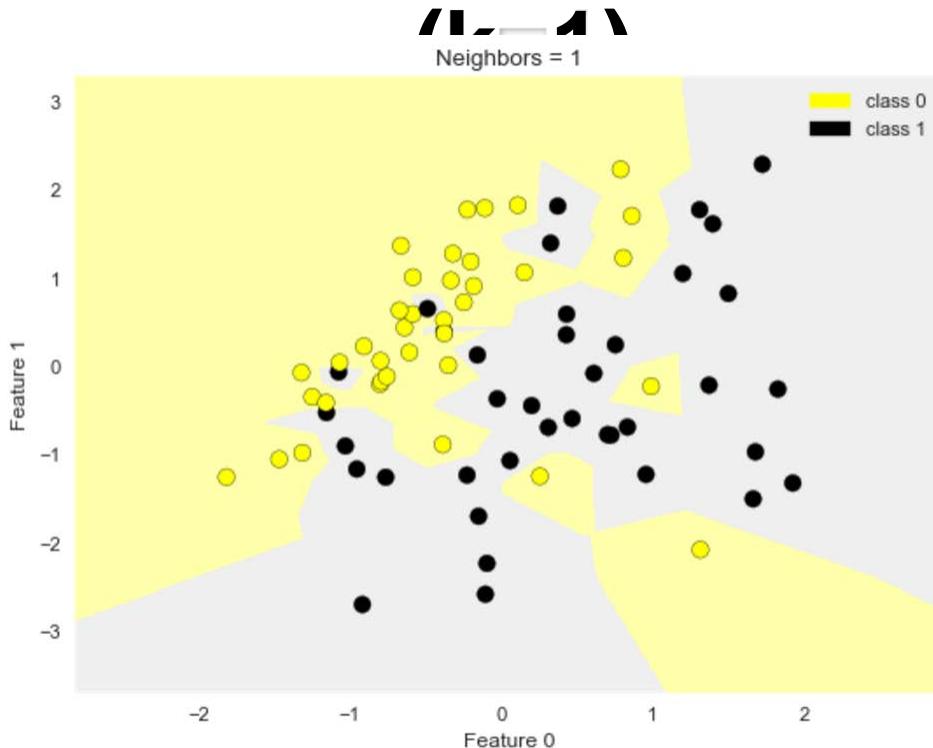
Associate Professor of Information & Computer Science  
University of Michigan

# The k-Nearest Neighbor (k-NN) Classifier Algorithm

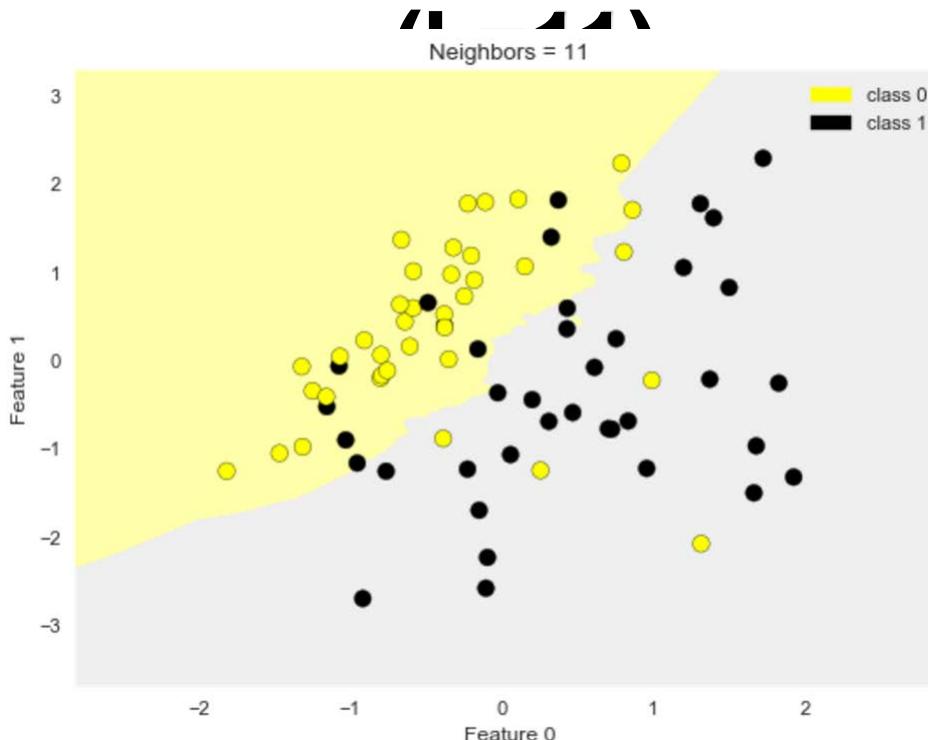
Given a training set  $X_{\text{train}}$  with labels  $y_{\text{train}}$ , and given a new instance  $x_{\text{test}}$  to be classified:

1. Find the most similar instances (let's call them  $X_{\text{NN}}$ ) to  $x_{\text{test}}$  that are in  $X_{\text{train}}$ .
2. Get the labels  $y_{\text{NN}}$  for the instances in  $X_{\text{NN}}$
3. Predict the label for  $x_{\text{test}}$  by combining the labels  $y_{\text{NN}}$   
e.g. simple majority vote

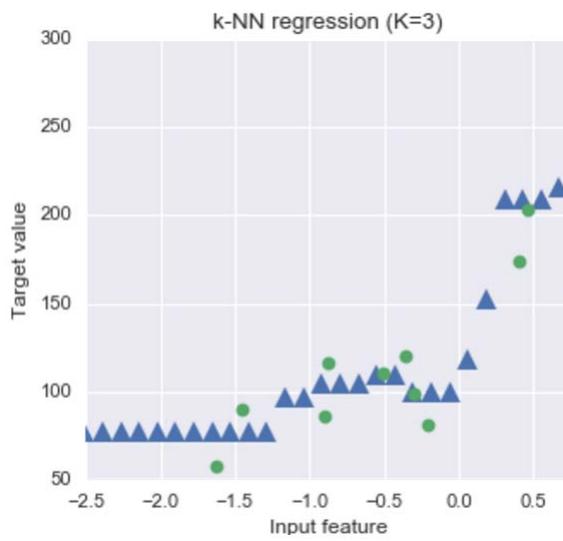
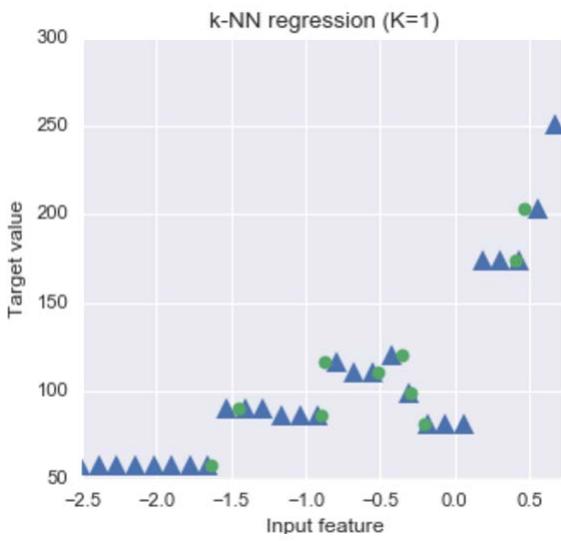
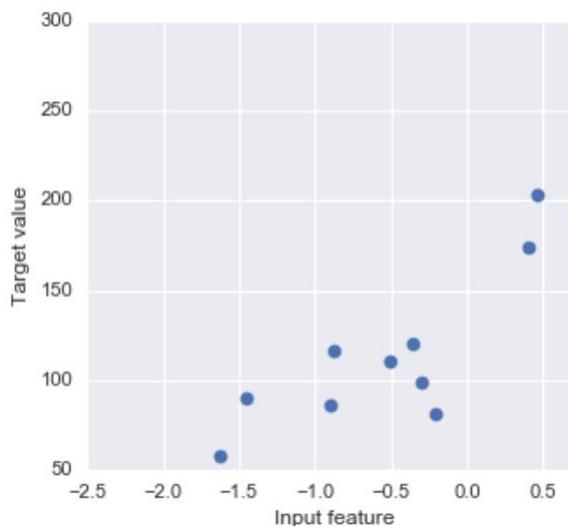
# Nearest Neighbors Classification



# Nearest Neighbors Classification



# k-Nearest Neighbors Regression



# The $R^2$ ("r-squared") Regression Score

- **Measures how well a prediction model for regression fits the given data.**
- **The score is between 0 and 1:**
  - *A value of 0 corresponds to a constant model that predicts the mean value of all training target values.*
  - *A value of 1 corresponds to perfect prediction*
- **Also known as "coefficient of determination"**

small k result in model with higher complexity

larger k result in simple model with lower complexity, train  $R^2$   
smaller, but test  $R^2$  larger

# KNeighborsClassifier and KNeighborsRegressor: important parameters

## Model complexity

- ***n\_neighbors*** : number of nearest neighbors (k) to consider
  - Default = 5

## Model fitting

- ***metric***: *distance function between data points*
  - Default: Minkowski distance with power parameter  $p = 2$  (Euclidean)



# Applied Machine Learning

## Linear Regression: Least-Squares

Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

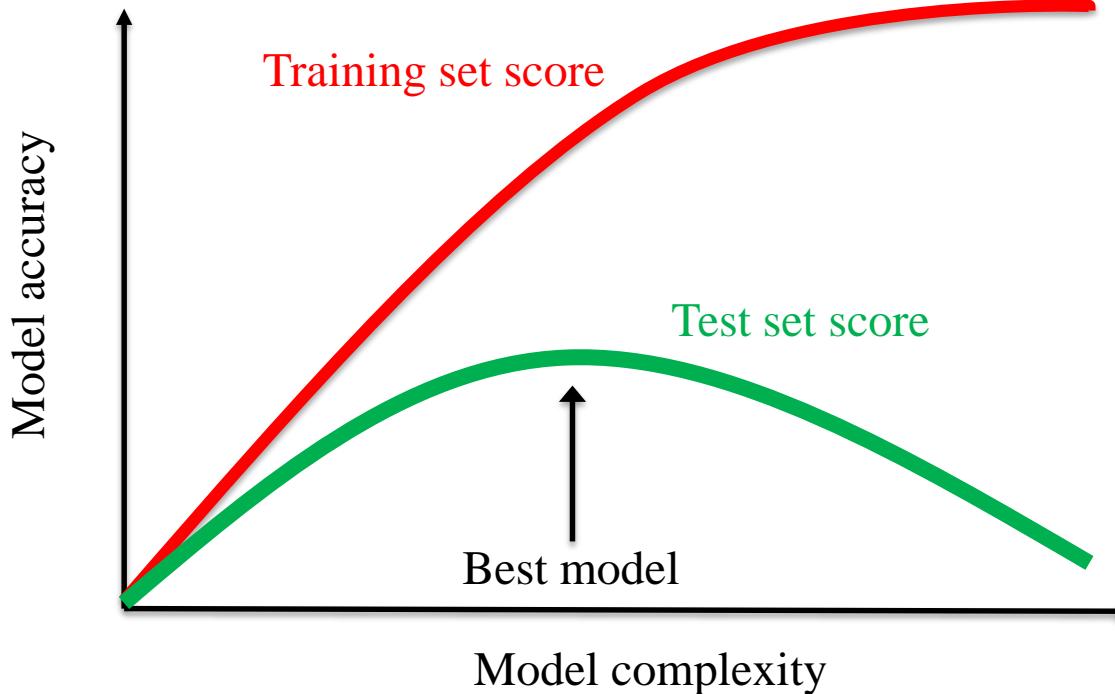
# Supervised learning methods: Overview

- **To start with, we'll look at two simple but powerful prediction algorithms:**
  - *K-nearest neighbors (review from week 1, plus regression)*
  - *Linear model fit using least-squares*
- **These represent two complementary approaches to supervised learning:**
  - *K-nearest neighbors makes few assumptions about the structure of the data and gives potentially accurate but sometimes unstable predictions (sensitive to small changes in the training data).*
  - *Linear models make strong assumptions about the structure of the data and give stable but potentially inaccurate predictions.*

# Supervised Learning Methods: Overview

- We'll cover a number of widely-used supervised learning methods for classification and regression.
- For each supervised learning method we'll explore:
  - *How the method works conceptually at a high level.*
  - *What kind of feature preprocessing is typically needed.*
  - *Key parameters that control model complexity, to avoid under- and over-fitting.*
  - *Positives and negatives of the learning method.*

# The relationship between model complexity and training/test performance



# Linear Models

- A linear model is a sum of weighted variables that predicts a target output value given an input data instance. Example: *predicting housing prices*

- House features: taxes per year ( $X_{TAX}$ ), age in years ( $X_{AGE}$ )

$$\widehat{Y_{PRICE}} = 212000 + 109 X_{TAX} - 2000 X_{AGE}$$

- A house with feature values ( $X_{TAX}, X_{AGE}$ ) of (10000, 75) would have a predicted selling price of:

$$\widehat{Y_{PRICE}} = 212000 + 109 \cdot 10000 - 2000 \cdot 75 = 1,152,000$$

# Linear Regression is an Example of a Linear Model

**Input instance – feature vector:**  $\mathbf{x} = (x_0, x_1, \dots, x_n)$

**Predicted output:**

$$\hat{y} = \widehat{w}_0 x_0 + \widehat{w}_1 x_1 + \cdots + \widehat{w}_n x_n + \hat{b}$$

**Parameters to estimate:**

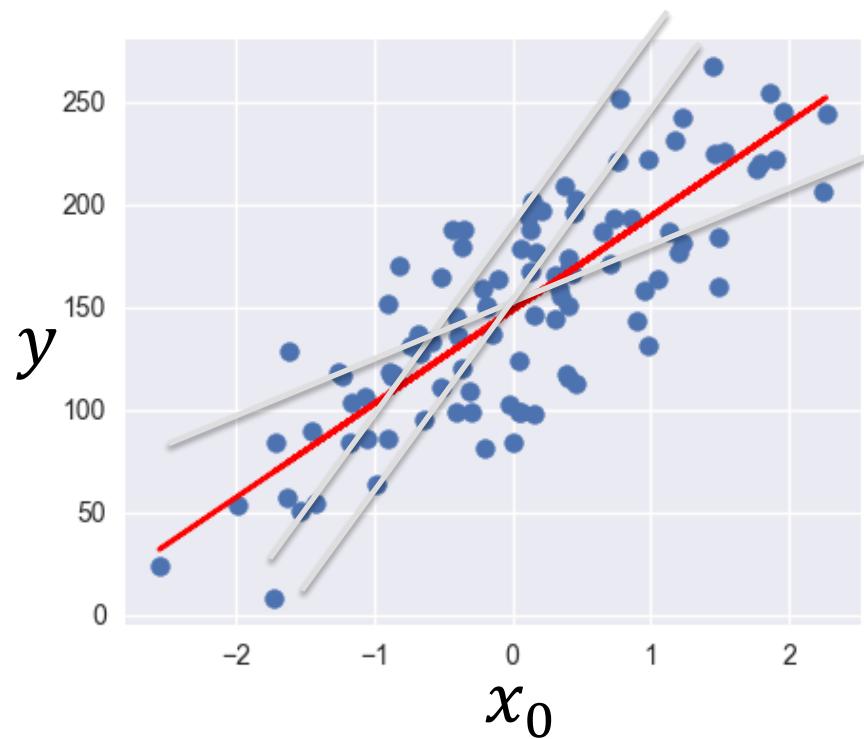
$$\left\{ \begin{array}{l} \widehat{\mathbf{w}} = (\widehat{w}_0, \dots, \widehat{w}_n): \text{feature weights/} \\ \qquad \qquad \qquad \text{model coefficients} \\ \widehat{b}: \text{constant bias term / intercept} \end{array} \right.$$

# A Linear Regression Model with one Variable (Feature)

**Input instance:**  $x = (x_0)$

**Predicted output:**  $\hat{y} = \widehat{w}_0 x_0 + \hat{b}$

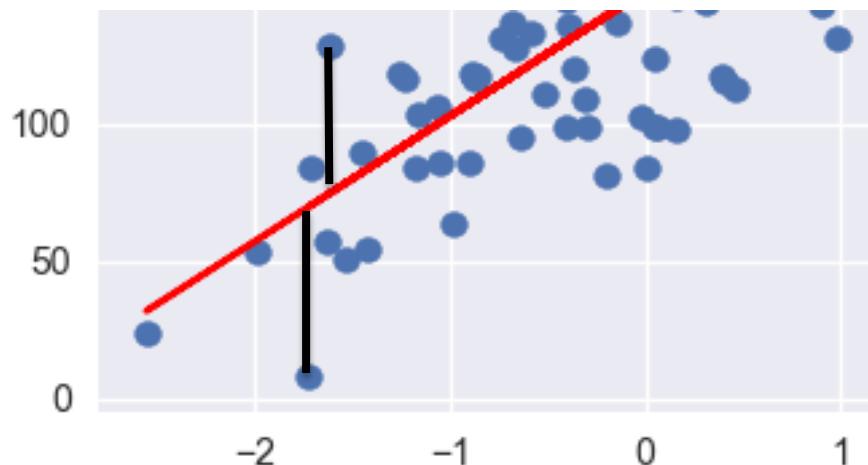
**Parameters**  $\widehat{w}_0$  (slope)  
**to estimate:**  $\hat{b}$  (y-intercept)



# Least-Squares Linear Regression

## ("Ordinary least-squares")

- Finds  $w$  and  $b$  that minimizes the mean squared error of the linear model: the sum of squared differences between predicted target and actual target values.
- No parameters to control model complexity.



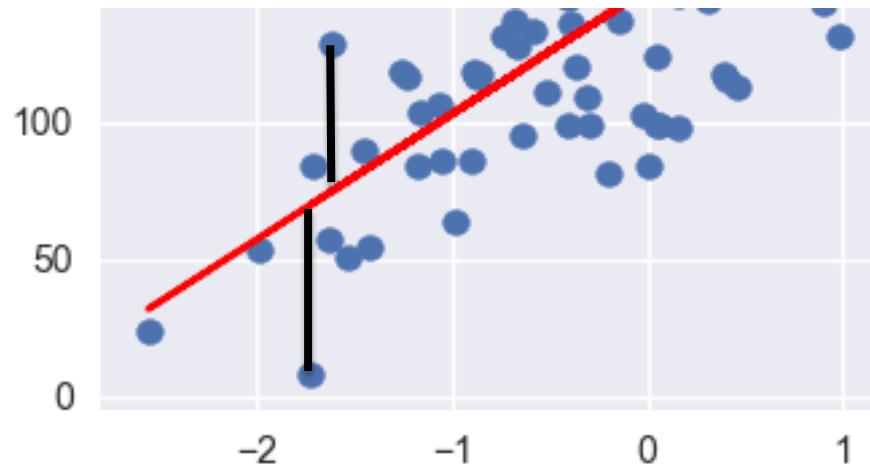
# How are Linear Regression Parameters $w$ , $b$ Estimated?

- Parameters are estimated from training data.
- There are many different ways to estimate  $w$  and  $b$ :
  - *Different methods correspond to different "fit" criteria and goals and ways to control model complexity.*
- The learning algorithm finds the parameters that optimize an objective function, typically to minimize some kind of loss function of the predicted target values vs. actual target values.

# Least-Squares Linear Regression

## ("Ordinary least-squares")

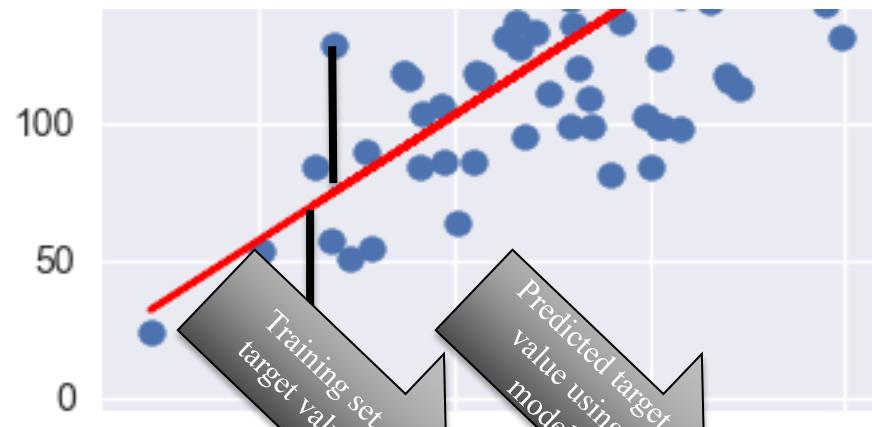
- Finds  $w$  and  $b$  that minimizes the sum of squared differences (RSS) over the training data between predicted target and actual target values.
- a.k.a. mean squared error of the linear model
- No parameters to control model complexity.



$$RSS(w, b) = \sum_{\{i=1\}}^N (y_i - (w \cdot x_i + b))^2$$

# Least-Squares Linear Regression ("Ordinary Least-Squares")

- Finds  $w$  and  $b$  that minimizes the sum of squared differences (RSS) over the training data between predicted target and actual target values.
- a.k.a. mean squared error of the linear model
- No parameters to control model complexity.



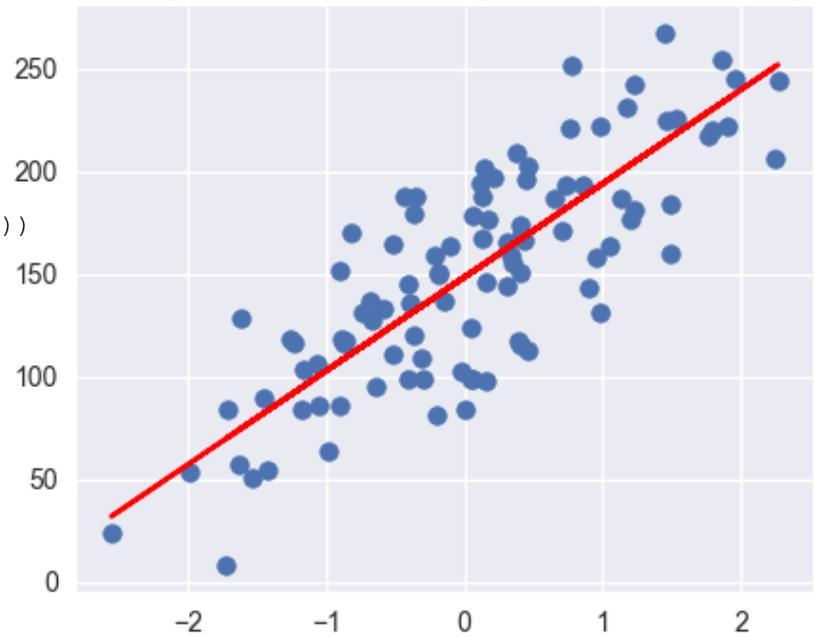
$$RSS(w, b) = \sum_{\{i=1\}} (y_i - (w \cdot x_i + b))^2$$

# Least-Squares Linear Regression in Scikit-Learn

```
from sklearn.linear_model import LinearRegression  
  
X_train, X_test, y_train, y_test =  
    train_test_split(X_R1, y_R1, random_state = 0)  
  
linreg = LinearRegression().fit(X_train, y_train)  
  
print("linear model intercept (b): {}".format(linreg.intercept_))  
print("linear model coeff (w): {}".format(linreg.coef_))
```

$$\hat{y} = w_0 x_0 + b$$

linreg.coef\_      linreg.intercept\_



# Least-Squares Linear Regression in Scikit-Learn

```
from sklearn.linear_model import LinearRegression

X_train, X_test, y_train, y_test =
    train_test_split(X_R1, y_R1, random_state = 0)

linreg = LinearRegression().fit(X_train, y_train)

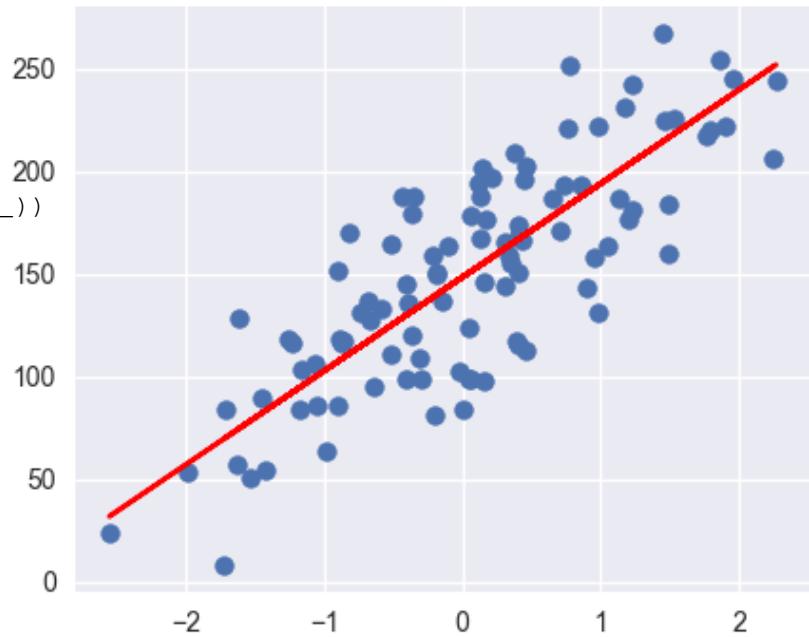
print("linear model intercept (b): {}".format(linreg.intercept_))
print("linear model coeff (w): {}".format(linreg.coef_))
```

Underscore denotes a quantity derived from training data, as opposed to a user setting.

linreg.coef\_

linreg.intercept\_

$$\hat{y} = w_0 x_0 + b$$



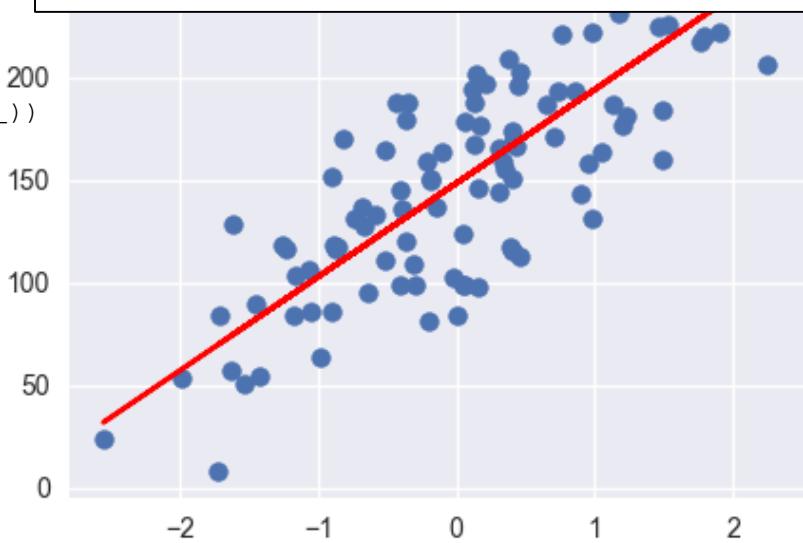
# Least-Squares Linear Regression in Scikit-Learn

```
from sklearn.linear_model import LinearRegression  
  
X_train, X_test, y_train, y_test =  
    train_test_split(X_R1, y_R1, random_state = 0)  
  
linreg = LinearRegression().fit(X_train, y_train)  
  
print("linear model intercept (b): {}".format(linreg.intercept_))  
print("linear model coeff (w): {}".format(linreg.coef_))
```

$$\hat{y} = w_0 x_0 + b$$

linreg.coef\_      linreg.intercept\_

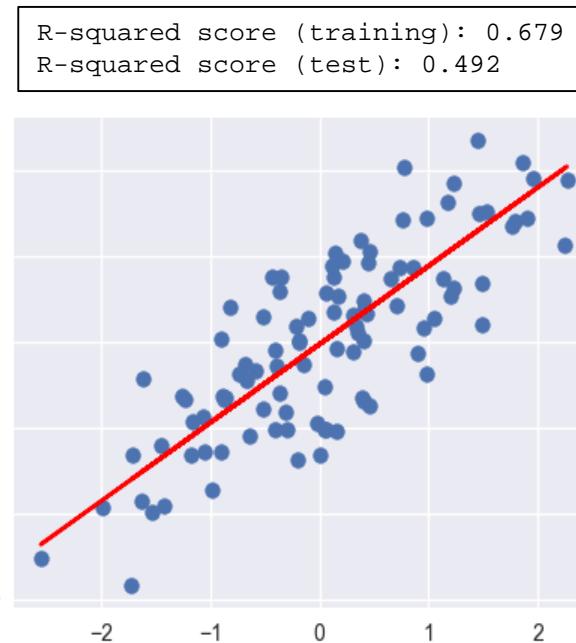
```
linear model coeff (w): [ 45.70870465]  
linear model intercept (b): 148.44575345658873  
R-squared score (training): 0.679  
R-squared score (test): 0.492
```



# K-NN Regression vs Least-Squares Linear Regression

k-NN ( $k=7$ )

LS linear





# Applied Machine Learning

## Linear Regression: Ridge, Lasso, and Polynomial Regression

Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

# Ridge Regression

- Ridge regression learns  $w$ ,  $b$  using the same least-squares criterion but adds a penalty for large variations in  $w$  parameters

$$RSS_{RIDGE}(w, b) = \sum_{\{i=1\}}^N (y_i - (w \cdot x_i + b))^2 + \alpha \sum_{\{j=1\}}^p w_j^2$$

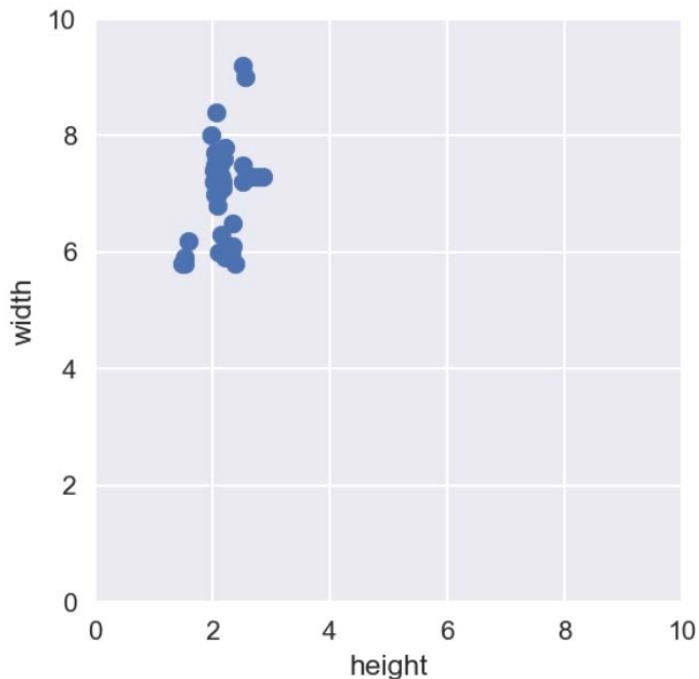
- Once the parameters are learned, the ridge regression prediction formula is the same as ordinary least-squares.
- The addition of a parameter penalty is called regularization. Regularization prevents overfitting by restricting the model, typically to reduce its complexity.
- Ridge regression uses L2 regularization: minimize sum of squares of  $w$  entries
- The influence of the regularization term is controlled by the  $\alpha$  parameter.
- Higher alpha means more regularization and simpler models.

# The Need for Feature Normalization

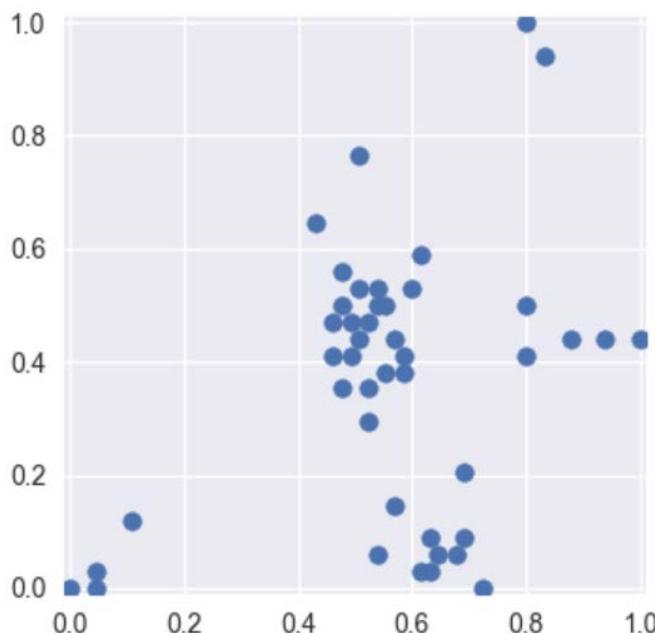
- Important for some machine learning methods that all features are on the same scale (e.g. faster convergence in learning, more uniform or 'fair' influence for all weights)
  - e.g. regularized regression, k-NN, support vector machines, neural networks, ...
- Can also depend on the data. More on feature engineering later in the course. For now, we do MinMax scaling of the features:
  - For each feature  $x_i$  : compute the min value  $x_i^{MIN}$  and the max value  $x_i^{MAX}$  achieved across all instances in the training set.
  - For each feature: transform a given feature  $x_i$  value to a scaled version  $x'_i$  using the formula

$$x'_i = (x_i - x_i^{MIN}) / (x_i^{MAX} - x_i^{MIN})$$

# Feature Normalization with MinMaxScaler



Unnormalized data points



Normalized with MinMaxScaler

# Using a scaler object: fit and transform methods

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
clf = Ridge().fit(X_train_scaled, y_train)
r2_score = clf.score(X_test_scaled, y_test)
```

**Tip:** It can be more efficient to do fitting and transforming together on the training set using the `fit_transform` method.

```
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

# Feature Normalization: The test set must use identical scaling to the training set

- Fit the scaler using the training set, then apply the same scaler to transform the test set.
- Do not scale the training and test sets using different scalers: this could lead to random skew in the data.
- Do not fit the scaler using any part of the test data: referencing the test data can lead to a form of *data leakage*. More on this issue later in the course.

Lasso regression is another form of regularized linear regression that uses an L1 regularization penalty for training (instead of ridge's L2 penalty)

- L1 penalty: Minimize the sum of the absolute values of the coefficients

$$RSS_{LASSO}(\mathbf{w}, b) = \sum_{\{i=1\}}^N (y_i - (\mathbf{w} \cdot \mathbf{x}_i + b))^2 + \alpha \sum_{\{j=1\}}^p |w_j|$$

- This has the effect of setting parameter weights in  $\mathbf{w}$  to zero for the least influential variables. This is called a sparse solution: a kind of feature selection
- The parameter  $\alpha$  controls amount of L1 regularization (default = 1.0).
- The prediction formula is the same as ordinary least-squares.
- When to use ridge vs lasso regression:
  - Many small/medium sized effects: use ridge.
  - Only a few variables with medium/large effect: use lasso.

# Lasso Regression on the Communities and Crime Dataset

For alpha = 2.0, 20 out of 88 features have non-zero weight.

Top features (sorted by abs. magnitude):

```
PctKidsBornNeverMar, 1488.365 # percentage of kids born to people who never married
PctKids2Par, -1188.740 # percentage of kids in family housing with two parents
HousVacant, 459.538 # number of vacant households
PctPersDenseHous, 339.045 # percent of persons in dense housing (more than 1 person/room)
NumInShelters, 264.932 # number of people in homeless shelters
```

# Polynomial Features with Linear Regression

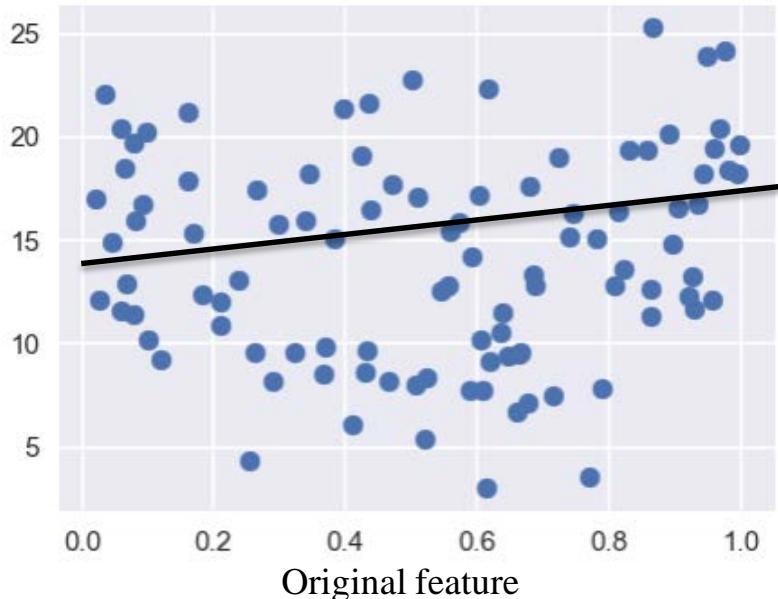
$$x = (x_0, x_1) \rightarrow x' = (x_0, x_1, x_0^2, x_0x_1, x_1^2)$$

$$\hat{y} = \hat{w}_0 x_0 + \hat{w}_1 x_1 + \hat{w}_{00} x_0^2 + \hat{w}_{01} x_0 x_1 + \hat{w}_{11} x_1^2 + b$$

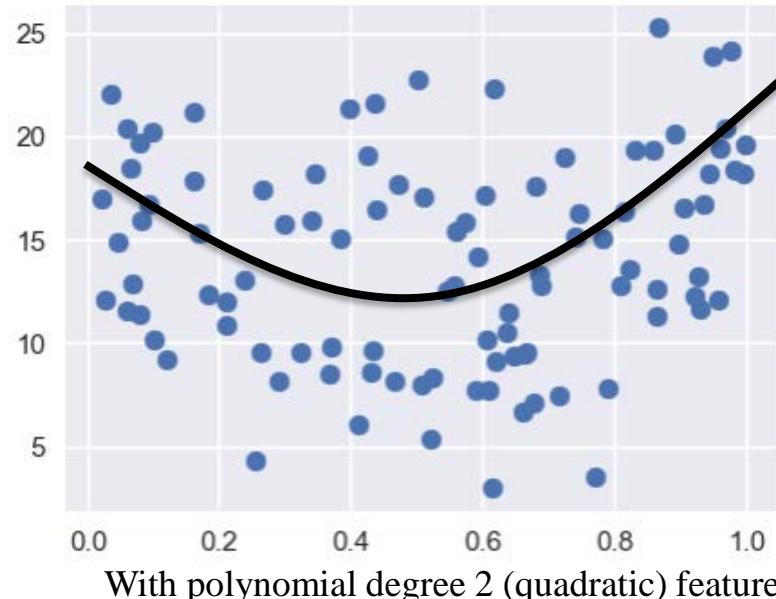
- Generate new features consisting of all polynomial combinations of the original two features  $(x_0, x_1)$ .
- The *degree* of the polynomial specifies how many variables participate at a time in each new feature (above example: degree 2)
- This is still a weighted linear combination of features, so it's a linear model, and can use same least-squares estimation method for  $w$  and  $b$ .

# Least-Squares Polynomial Regression

Complex regression problem with one input variable



Complex regression problem with one input variable



# Polynomial Features with Linear Regression

- **Why would we want to transform our data this way?**
  - *To capture interactions between the original features by adding them as features to the linear model.*
  - *To make a classification problem easier (we'll see this later).*
- **More generally, we can apply other non-linear transformations to create new features**
  - *(Technically, these are called non-linear basis functions)*
- **Beware of polynomial feature expansion with high as this can lead to complex models that overfit**
  - *Thus, polynomial feature expansion is often combined with a regularized learning method like ridge regression.*



# Applied Machine Learning

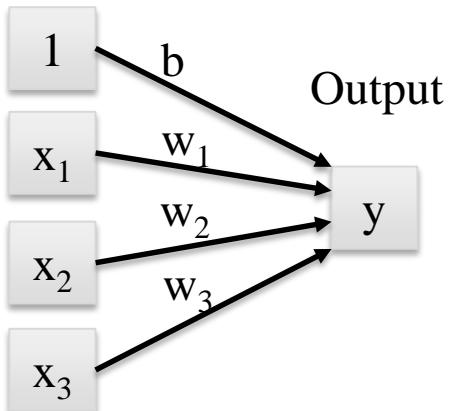
## Logistic regression

Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

# Linear Regression

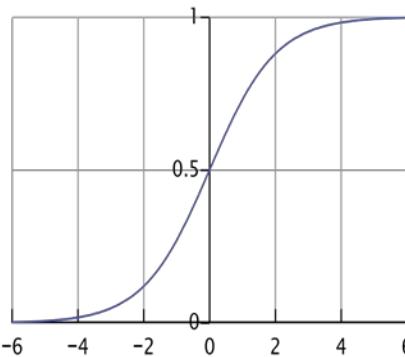
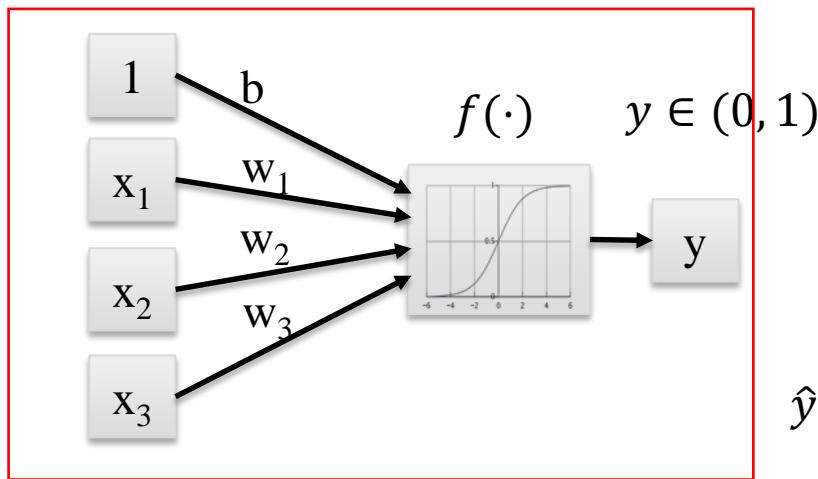
Input features



$$\hat{y} = \hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n$$

# Linear models for classification: Logistic Regression

Input features

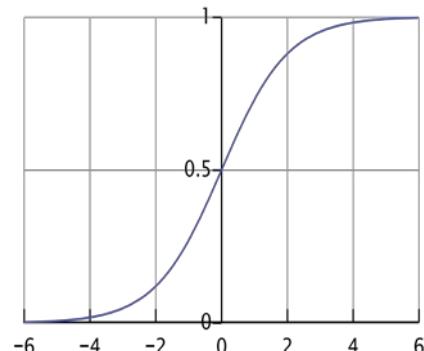
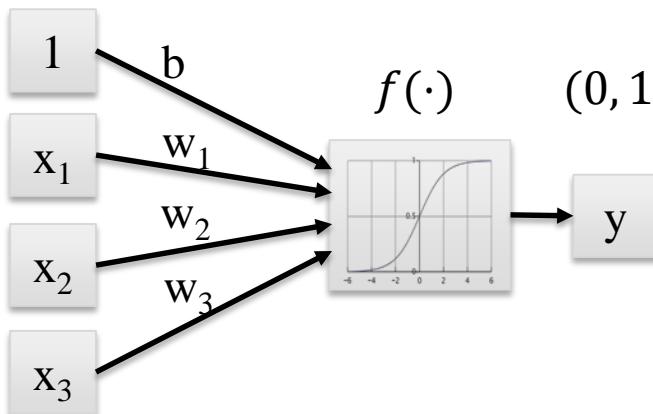


$$\hat{y} = \text{logistic}(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)$$

$$= \frac{1}{1 + \exp [-(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)]}$$

# Linear models for classification: Logistic Regression

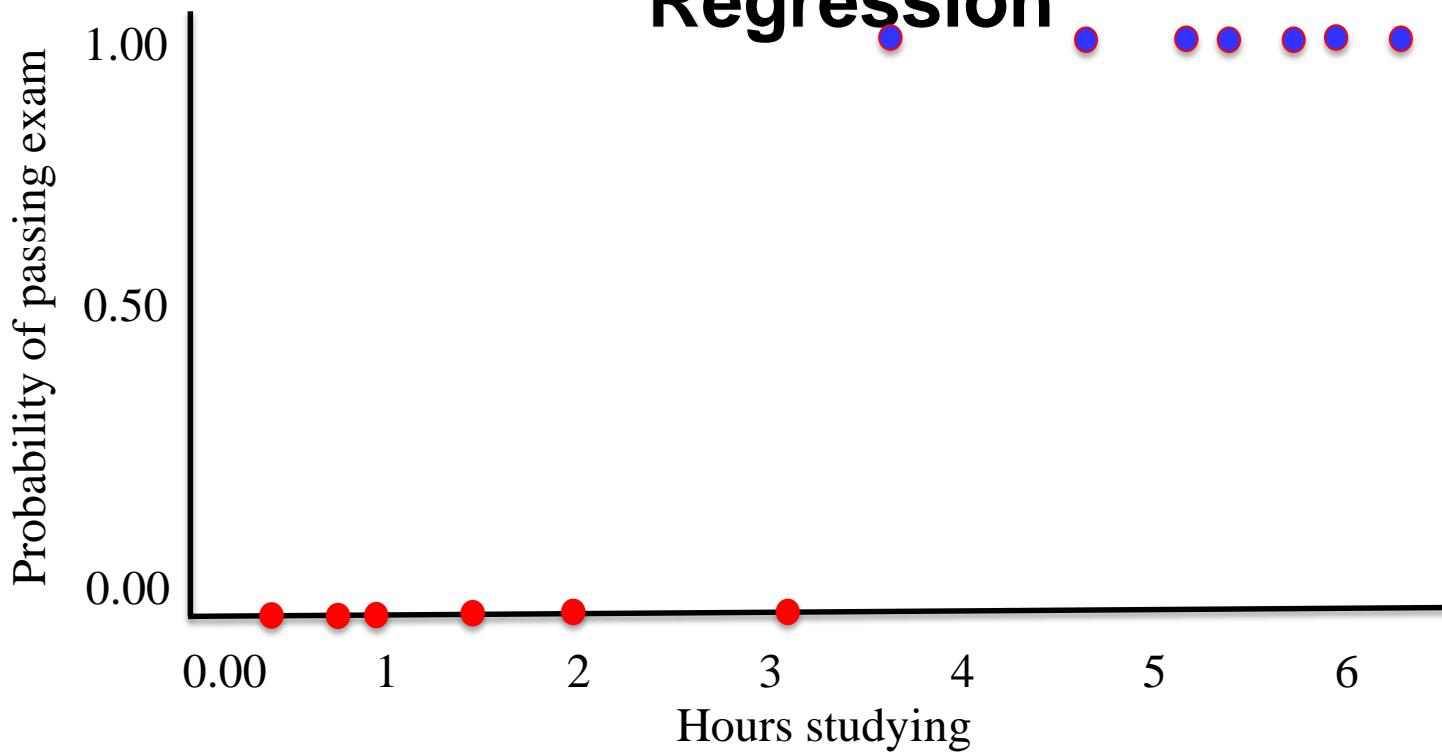
Input features



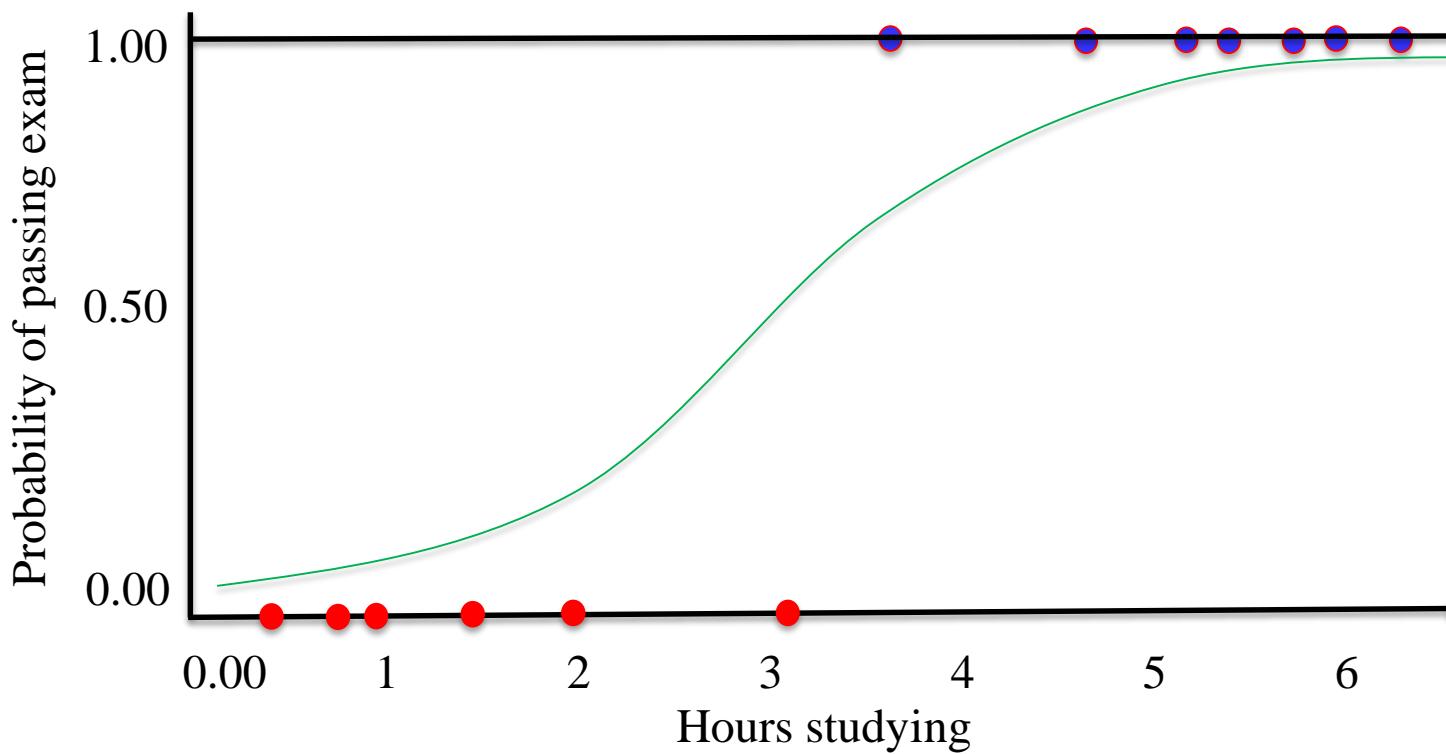
The logistic function transforms real-valued input to an output number  $y$  between 0 and 1, interpreted as the probability the input object belongs to the positive class, given its input features  $(x_0, x_1, \dots, x_n)$

$$\begin{aligned}\hat{y} &= \text{logistic}(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n) \\ &= \frac{1}{1 + \exp [-(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)]}\end{aligned}$$

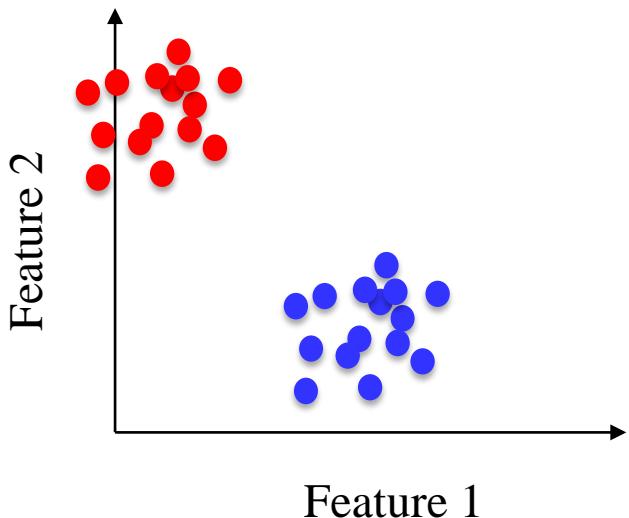
# Linear models for classification: Logistic Regression



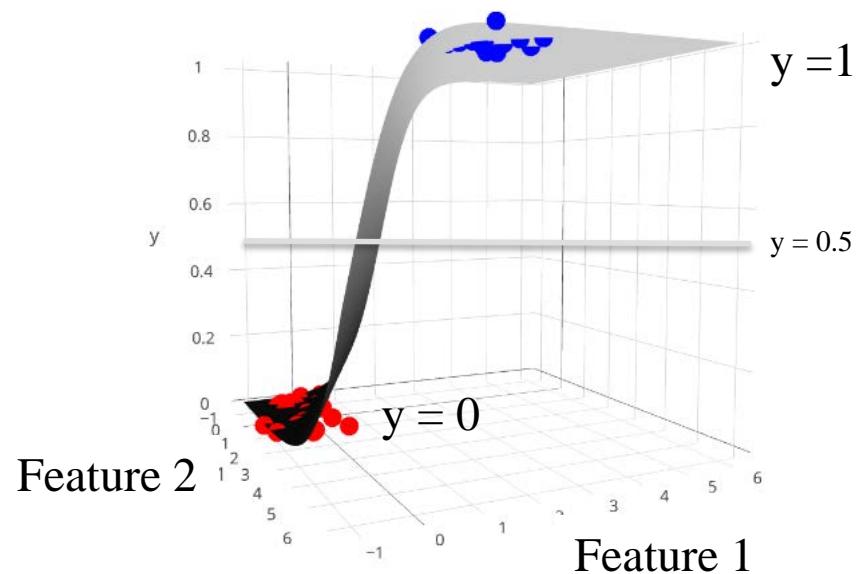
# Linear models for classification: Logistic Regression



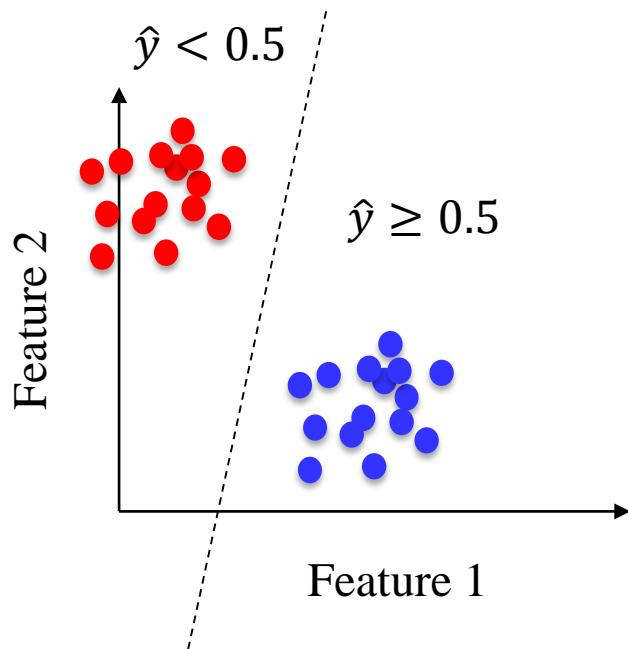
# Logistic Regression for binary classification



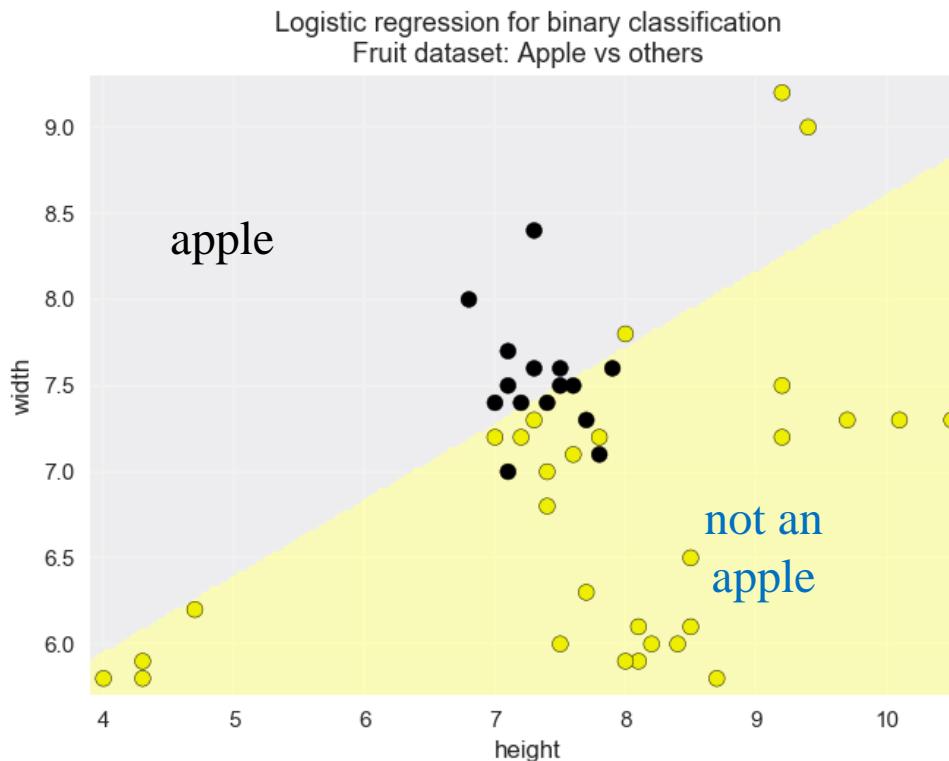
# Logistic Regression for binary classification



# Logistic Regression for binary classification

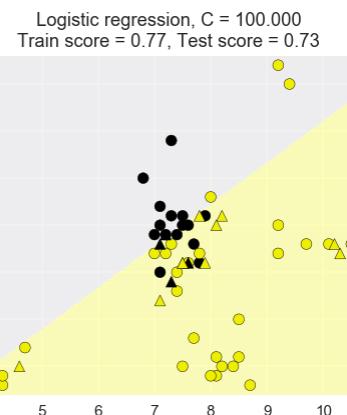
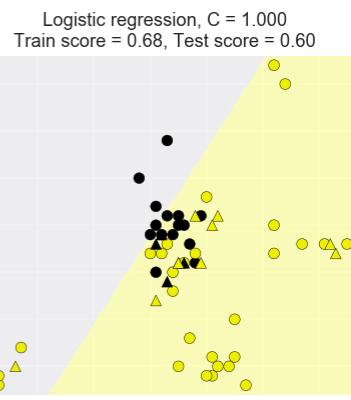
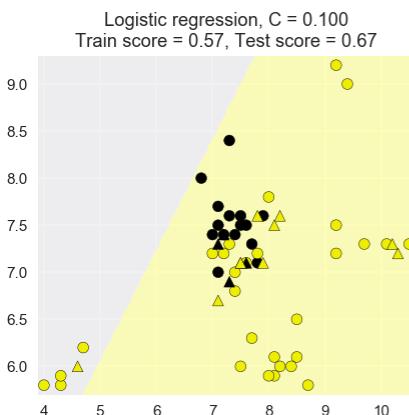


# Simple logistic regression problem: two-class, two-feature version of the fruit dataset



# Logistic Regression: Regularization

- L2 regularization is 'on' by default (like ridge regression)
- Parameter C controls amount of regularization (default 1.0)
- As with regularized linear regression, it can be important to normalize all features so that they are on the same scale.



high c, less regulation, better training  
data fit



# Applied Machine Learning

## Linear Classifiers: Support Vector Machines

Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

# Linear classifiers: how would you separate these two groups of training examples with a straight line?

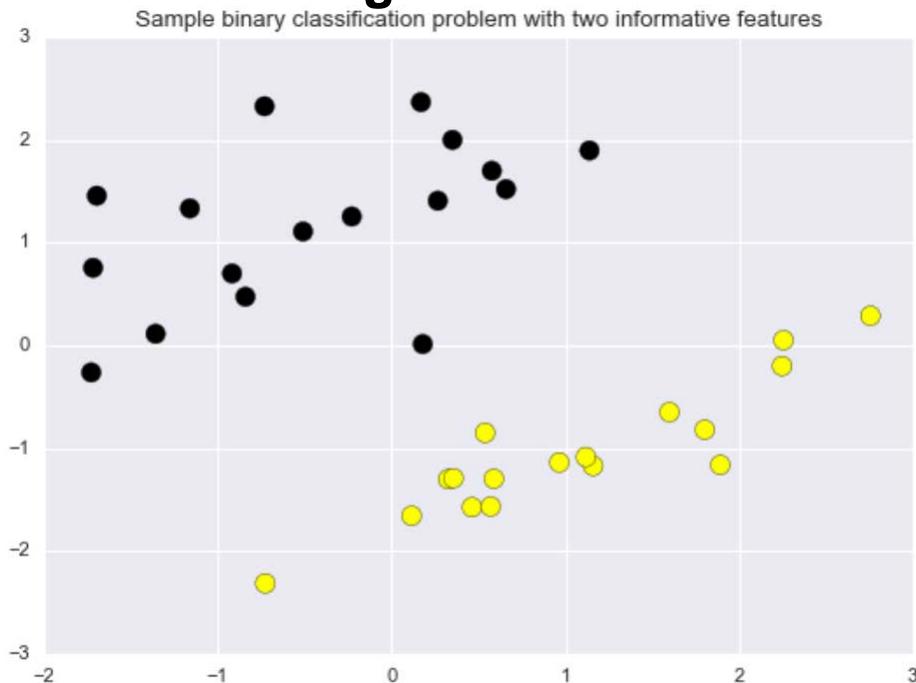
Feature  
vector

Class value

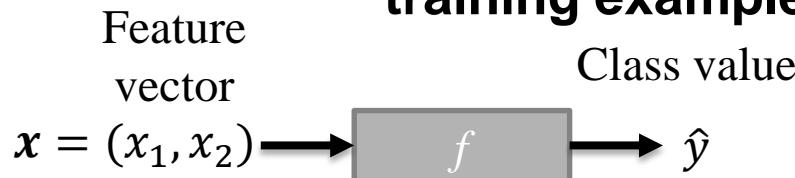
$$x \rightarrow f \rightarrow \hat{y}$$

$$f(x, w, b) = \text{sign}(w \circ x + b)$$

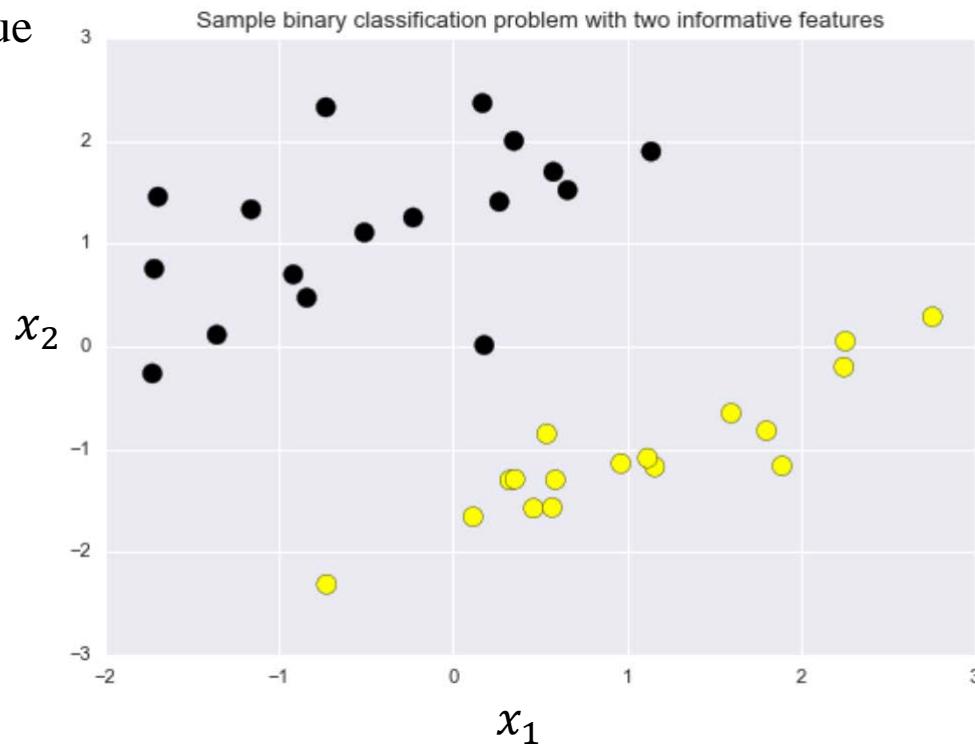
$$= \text{sign} (\sum w[i]x[i] + b)$$



# Linear classifiers: how would you separate these two groups of training examples with a straight line?



A linear classifier is a function that maps an input data point  $x$  to an output class value  $y$  (+1 or -1) using a linear function (with weight parameters  $w$  of the input point's features).



# Linear classifiers: how would you separate these two groups of training examples with a line?

Feature vector                      Class value

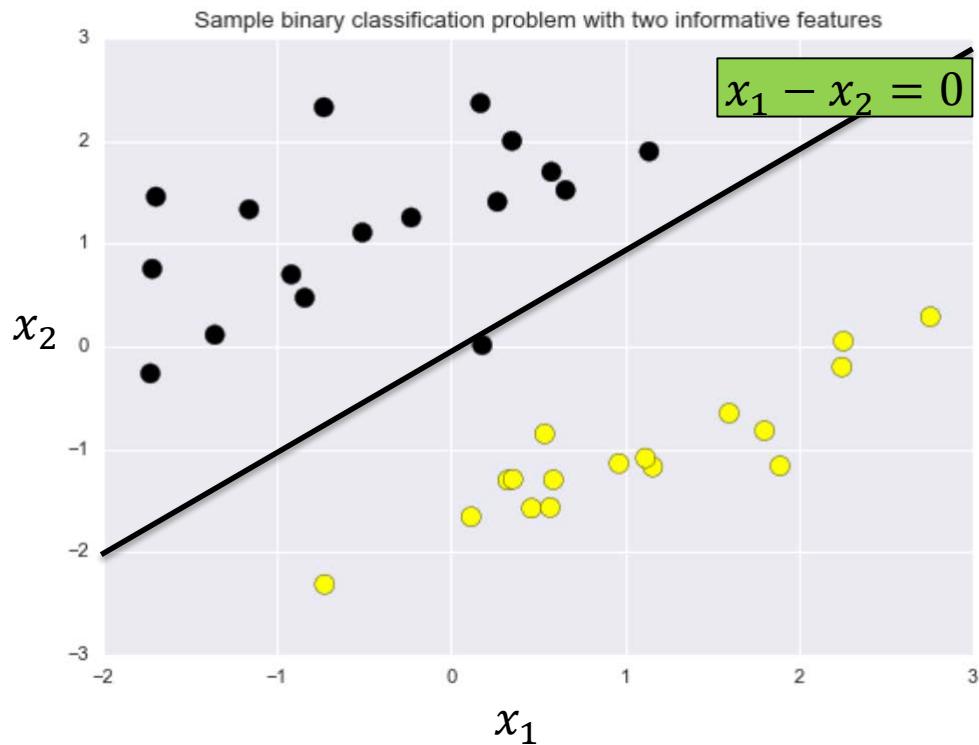


$$f(x, w, b) = \text{sign}(w \circ x + b)$$

$$x_1 - x_2 = 0$$

$$w = [1, -1]$$

$$b = 0$$



# Linear classifiers: how would you separate these two groups of training examples with a line?

Feature vector

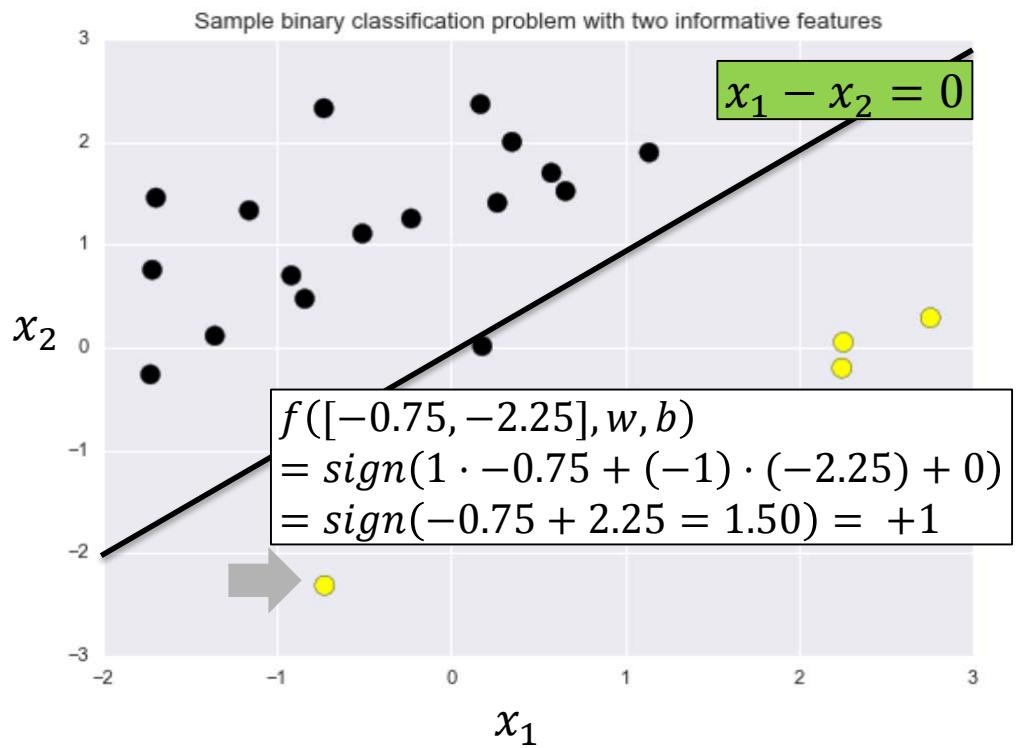


$$f(x, w, b) = \text{sign}(w \circ x + b)$$

$$x_1 - x_2 = 0$$

$$w = [1, -1]$$

$$b = 0$$



# Linear classifiers: how would you separate these two groups of training examples with a line?

Feature vector

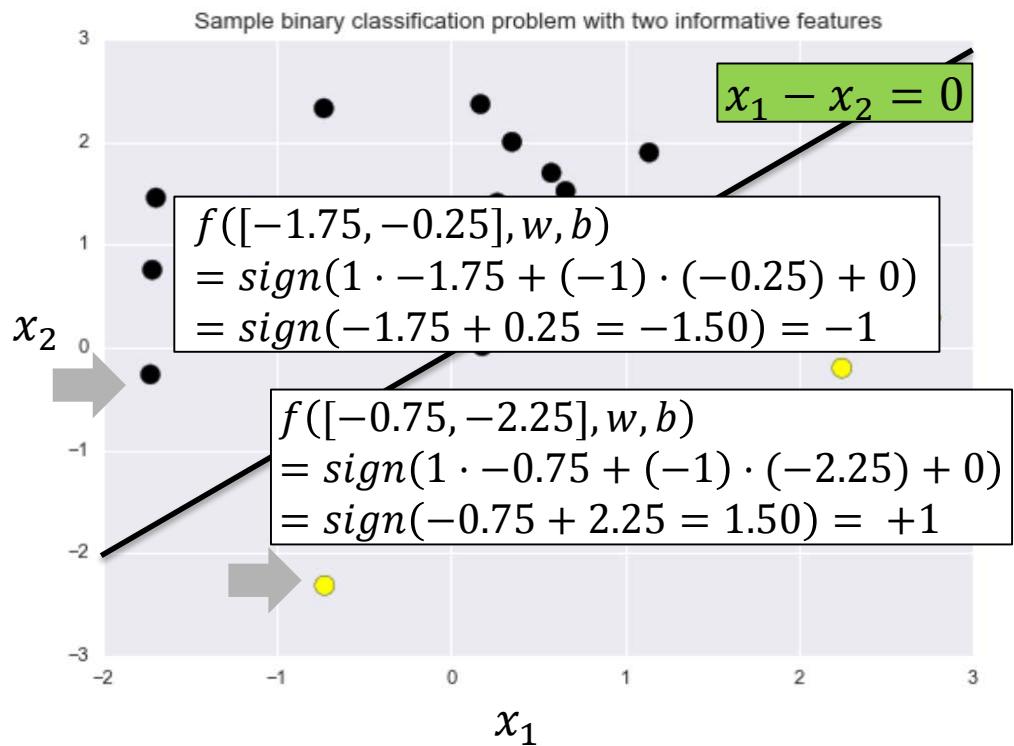


$$f(x, w, b) = \text{sign}(w \circ x + b)$$

$$x_1 - x_2 = 0$$

$$w = [1, -1]$$

$$b = 0$$



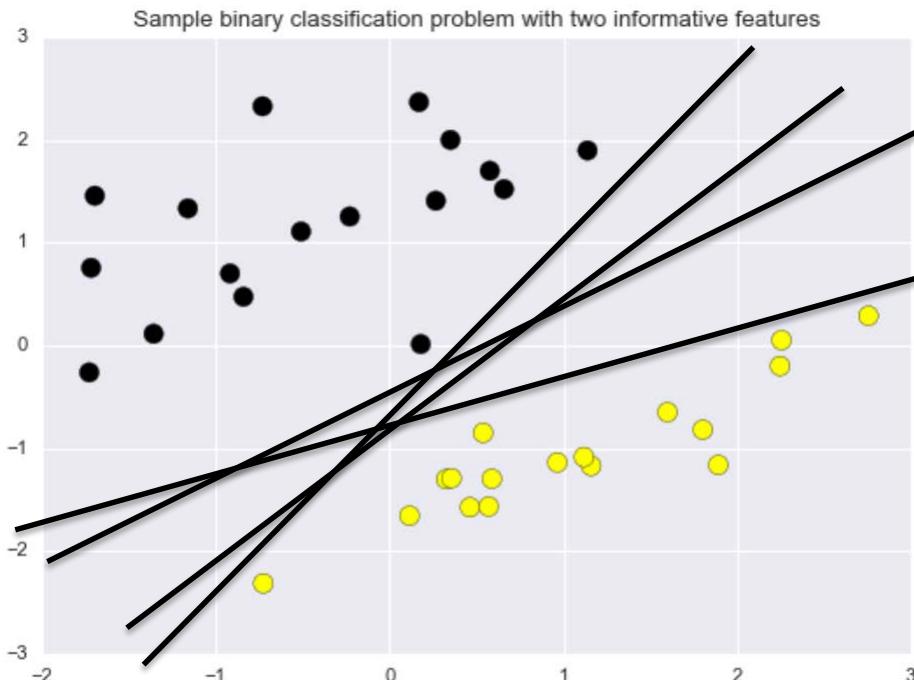
# Linear Classifiers



$$f(x, w, b) = \text{sign}(w \circ x + b)$$

There are many possible linear classifiers that could separate the two classes.

**Which one is best?**



# Classifier Margin



$$f(x, w, b) = \text{sign}(w \circ x + b)$$

## Classifier margin

Defined as the maximum width the decision boundary area can be increased before hitting a data point.



# Maximum margin linear classifier: Linear Support Vector Machines



$$f(x, w, b) = \text{sign}(w \circ x + b)$$

## Maximum margin classifier

The linear classifier with maximum margin is a Linear Support Vector Machine (LSVM).



# Regularization for SVMs: the C parameter

- The strength of regularization is determined by C
- Larger values of C: less regularization
  - *Fit the training data as well as possible*
  - *Each individual data point is important to classify correctly*
- Smaller values of C: more regularization
  - *More tolerant of errors on individual data points*

# Linear Models: Pros and Cons

## Pros:

- Simple and easy to train.
- Fast prediction.
- Scales well to very large datasets.
- Works well with sparse data.
- Reasons for prediction are relatively easy to interpret.

## Cons:

- For lower-dimensional data, other models may have superior generalization performance.
- For classification, data may not be linearly separable (more on this in SVMs with non-linear kernels)

# linear\_model: Important Parameters

## Model complexity

- **alpha:** weight given to the L1 or L2 regularization term in regression models
  - *default* = 1.0
- **C:** regularization weight for LinearSVC and LogisticRegression classification models
  - *default* = 1.0



# Applied Machine Learning

## Multi-Class Classification

**Kevyn Collins-Thompson**

**Associate Professor of Information & Computer Science  
University of Michigan**

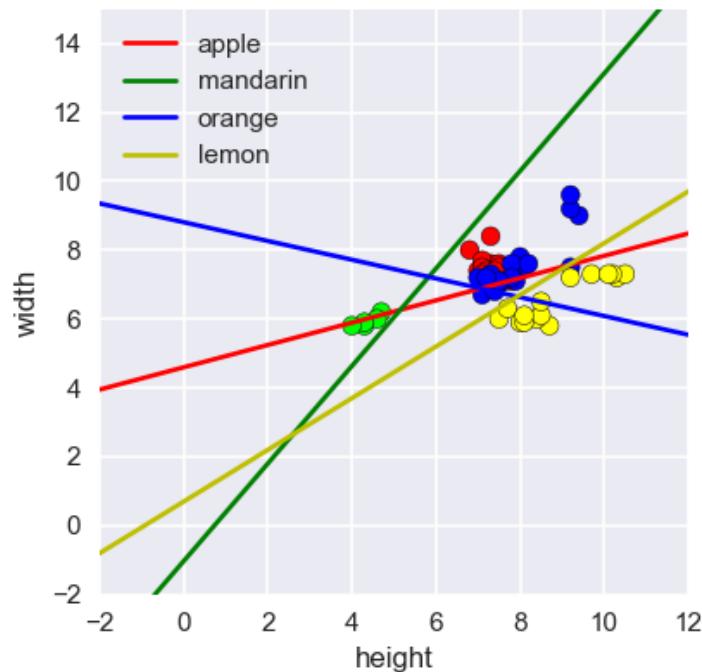
# Multi-Class classification with linear models

```
clf = LinearSVC(C=5, random_state = 67)
clf.fit(X_train, y_train)

print(clf.coef_)

[[-0.23401135  0.72246132]
 [-1.63231901  1.15222281]
 [ 0.0849835   0.31186707]
 [ 1.26189663 -1.68097    ]]

print(clf.intercept_)
[-3.31753728  1.19645936 -2.7468353  1.16107418]
```



# Multi-Class classification with linear models

```
clf = LinearSVC(C=5, random_state = 67)
clf.fit(X_train, y_train)
```

```
print(clf.coef_)
```

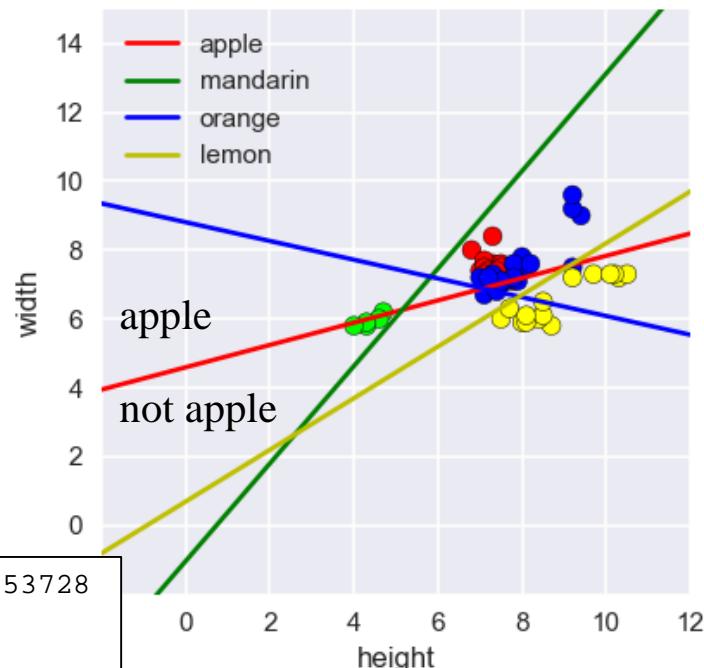
```
[[-0.23401135  0.72246132]
 [-1.63231901  1.15222281]
 [ 0.0849835   0.31186707]
 [ 1.26189663 -1.68097    ]]
```

```
print(clf.intercept_)
```

```
[-3.31753728  1.19645936 -2.7468353  1.16107418]
```

```
y_apple = -0.23401135 * height + 0.72246132 * width - 3.31753728
```

```
height=2, width=6: y_apple = + 0.549 (>= 0: predict apple)
height=2, width=2: y_apple = - 2.340 (< 0: predict other)
```





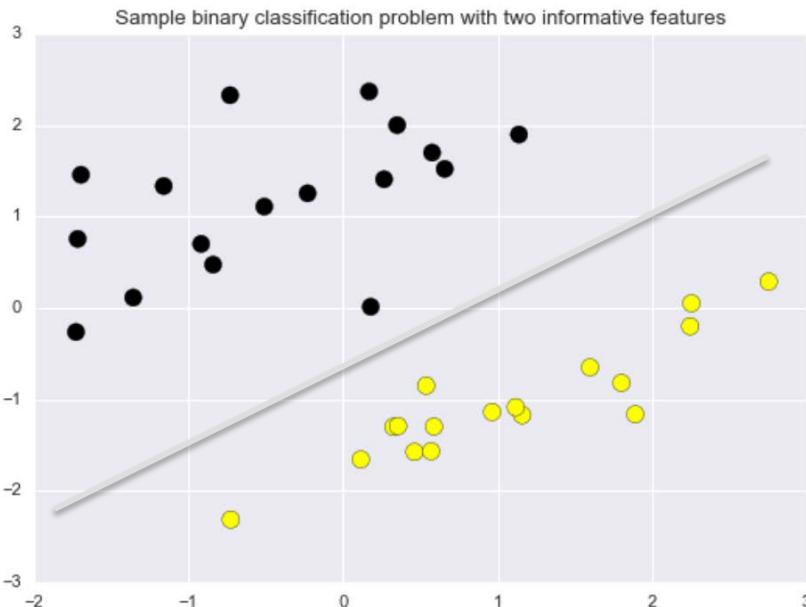
# Applied Machine Learning

## Kernelized Support Vector Machines

Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

We saw how linear support vector classifiers could effectively find a decision boundary with maximum margin

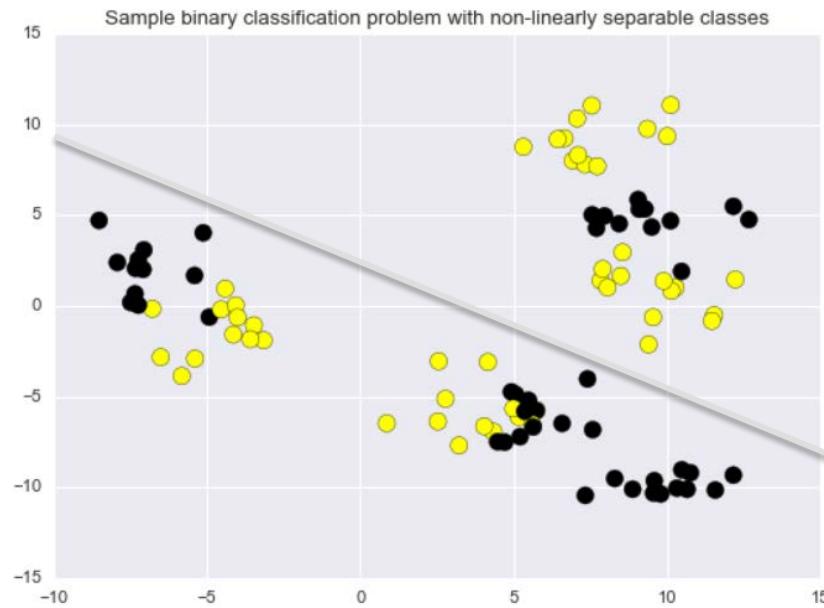


Easy for a linear classifier

# But what about more complex binary classification problems?

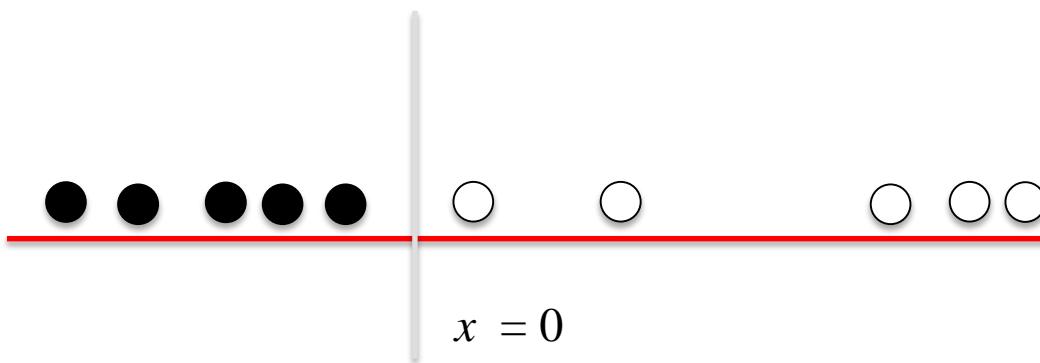


Easy for a linear classifier



Difficult/impossible for a linear classifier

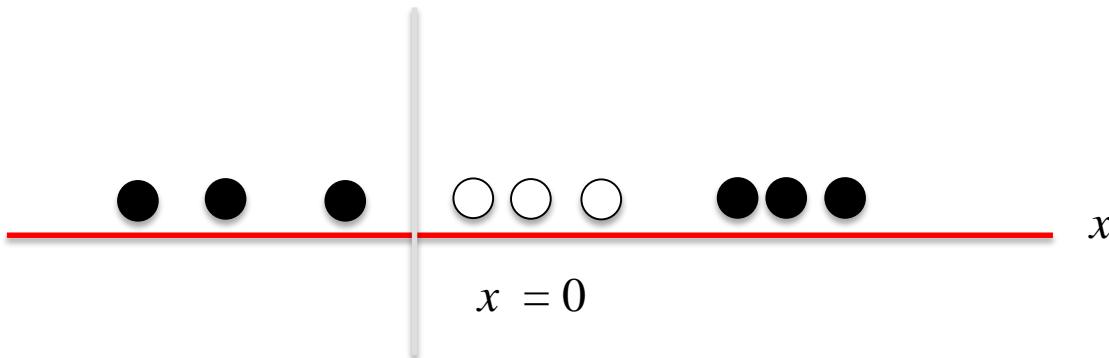
# A simple 1-dimensional classification problem for a linear classifier



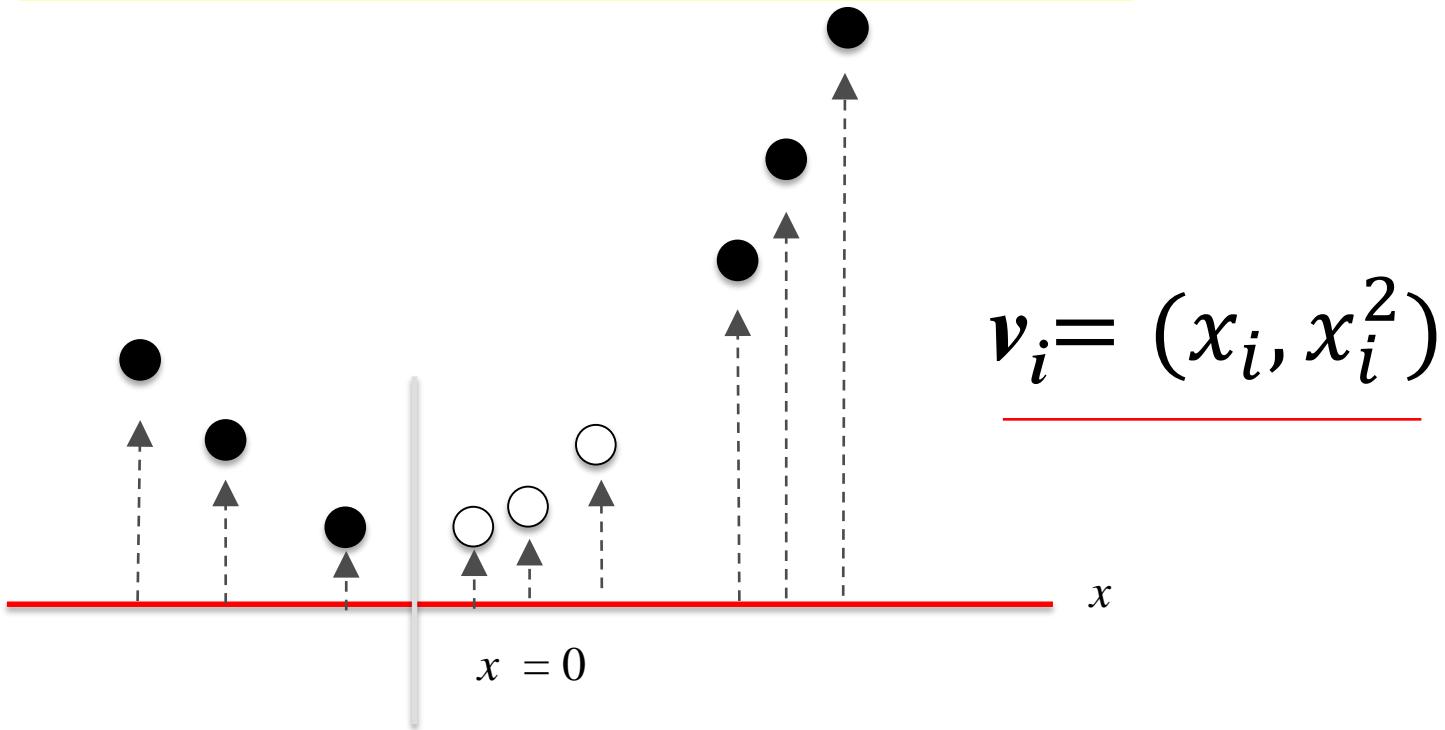
# A more perplexing 1-d classification problem for a linear classifier



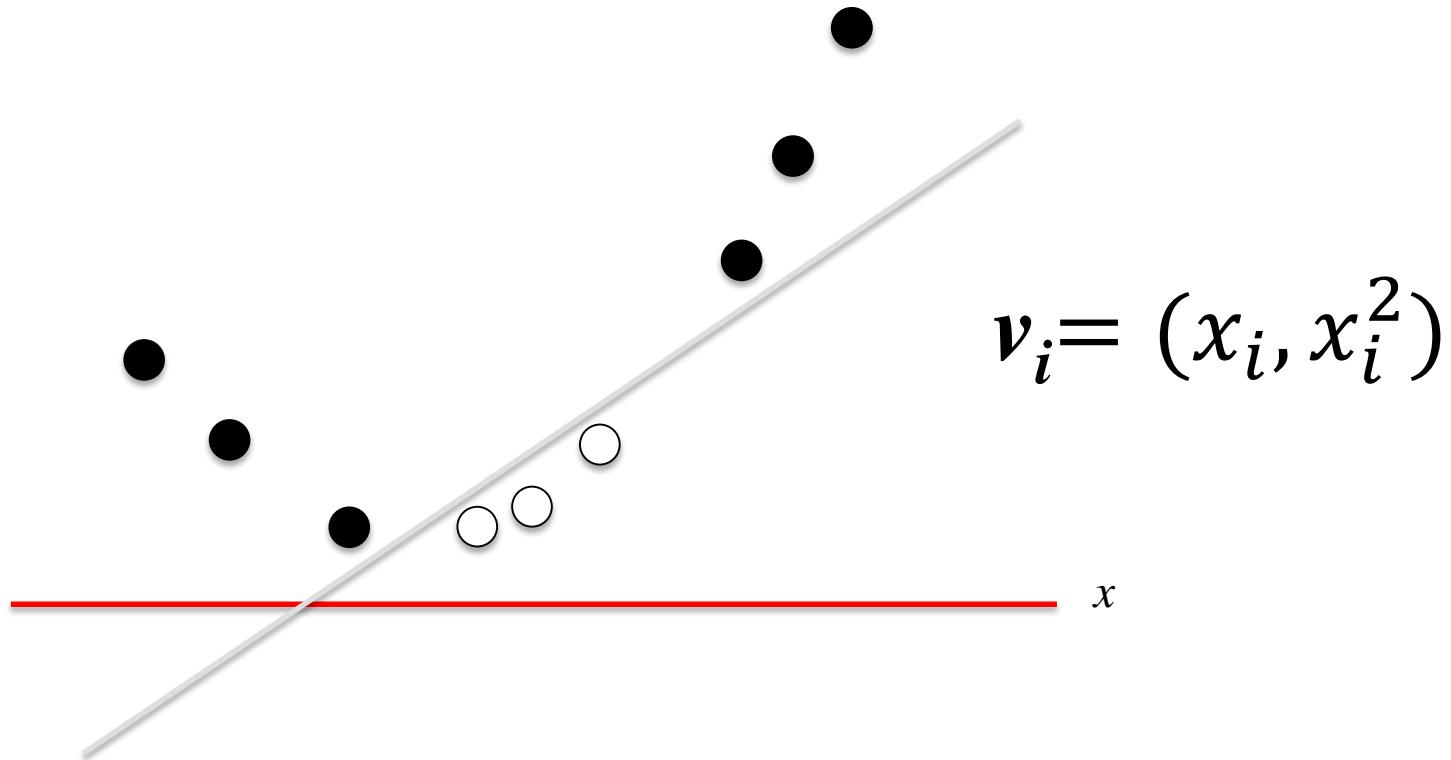
# A more perplexing 1-d classification problem for a linear classifier



Let's transform the data by adding a second dimension/feature  
(set to the squared value of the first feature)

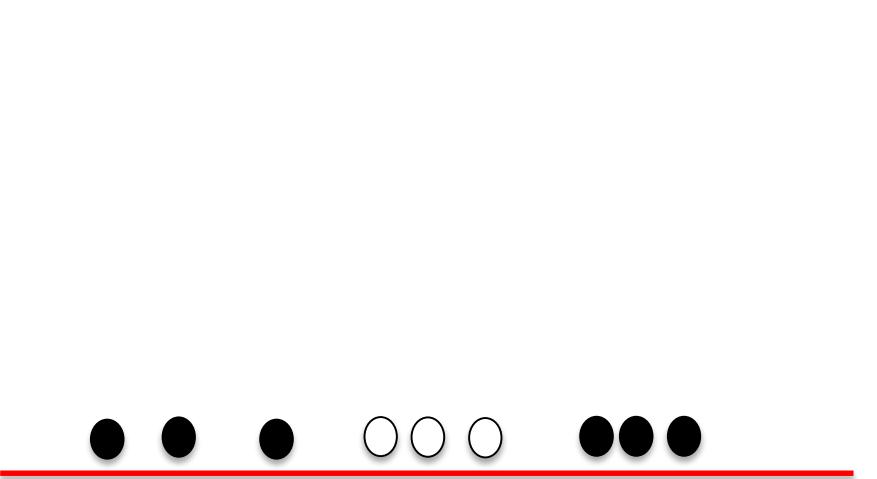


The data transformation makes it possible to solve this with a linear classifier

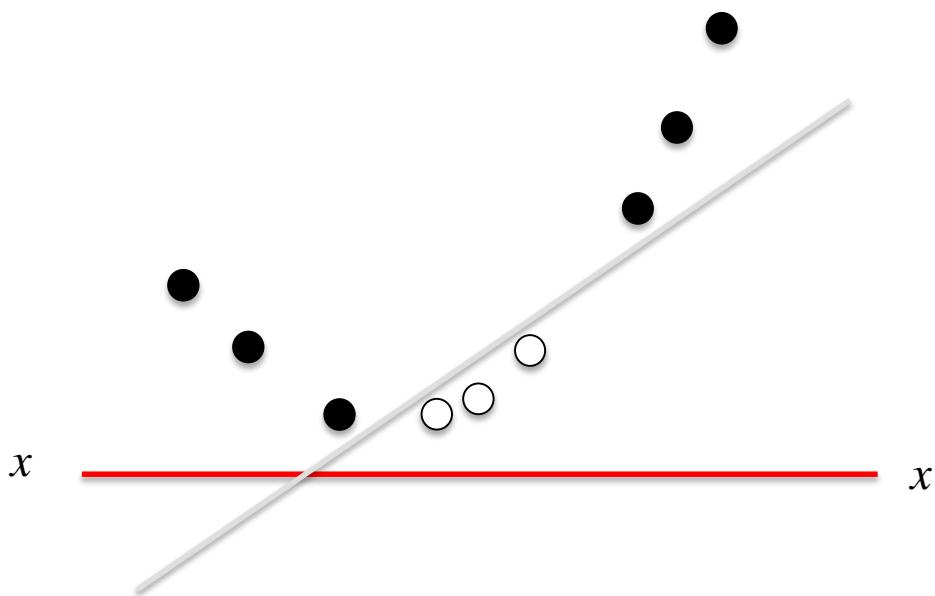


# What does the linear decision boundary in feature space correspond to in the original input space?

Original input space

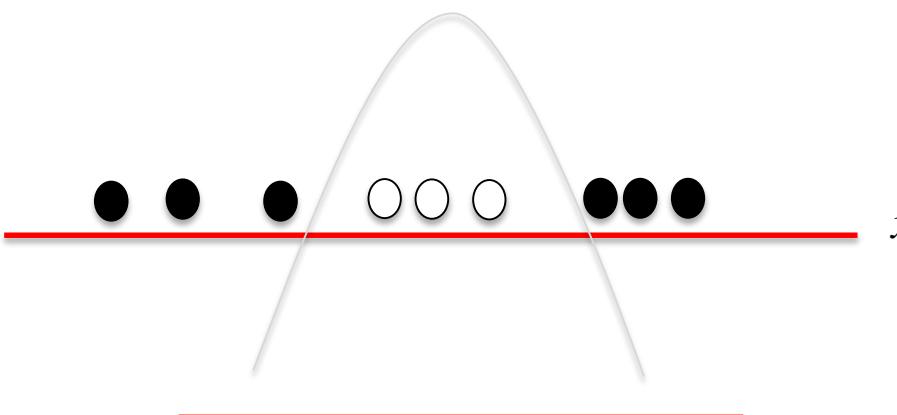


Feature space

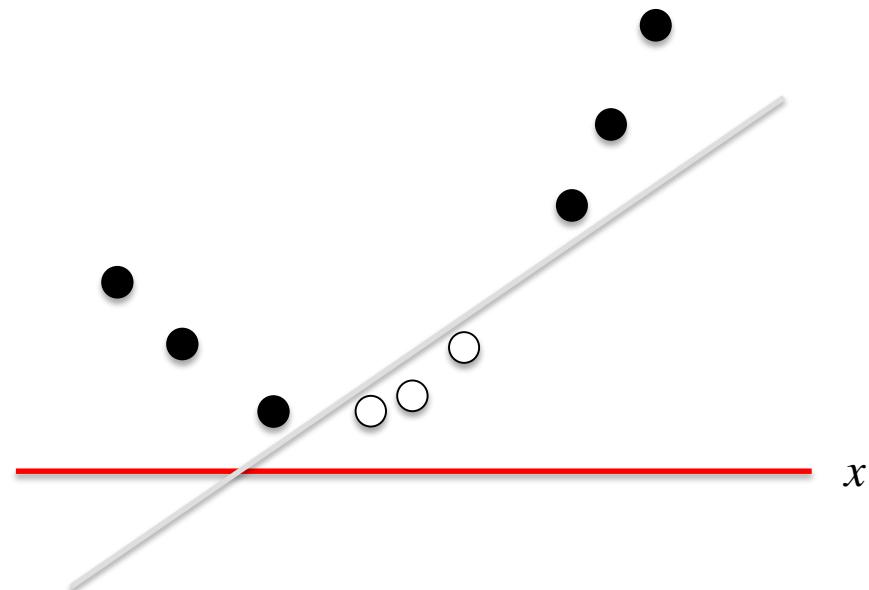


# What does the linear decision boundary correspond to in the original input space?

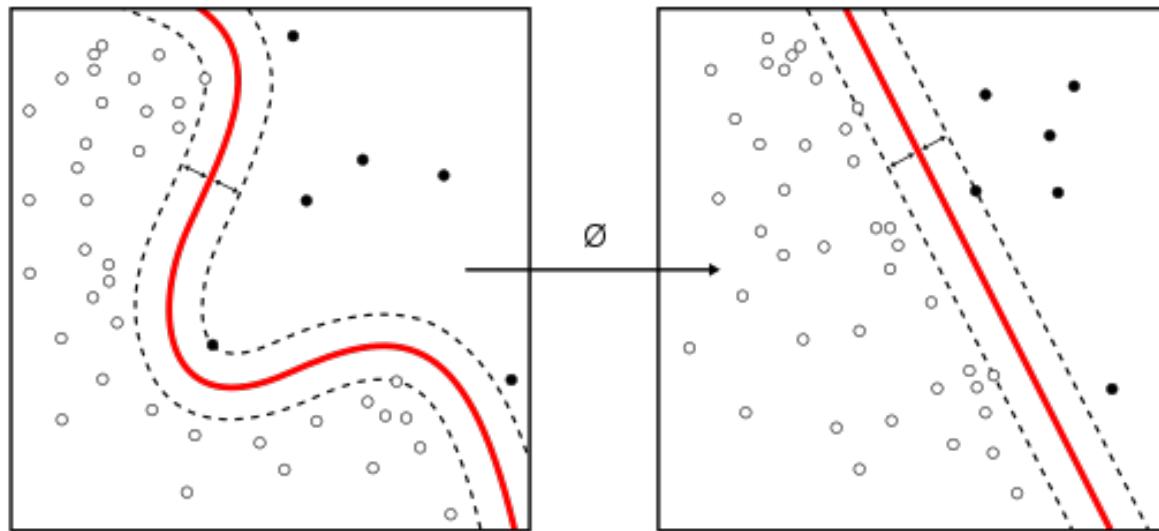
Original input space



Feature space



# Transforming the data can make it much easier for a linear classifier.



Original input space

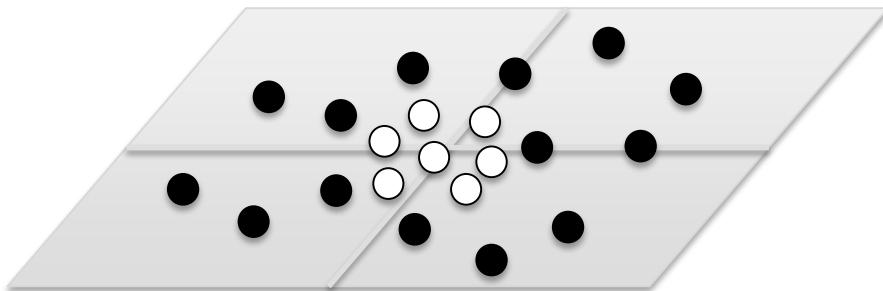
Feature space

Source: Wikipedia "Kernel Machine" article.

<https://commons.wikimedia.org/w/index.php?curid=47868867>

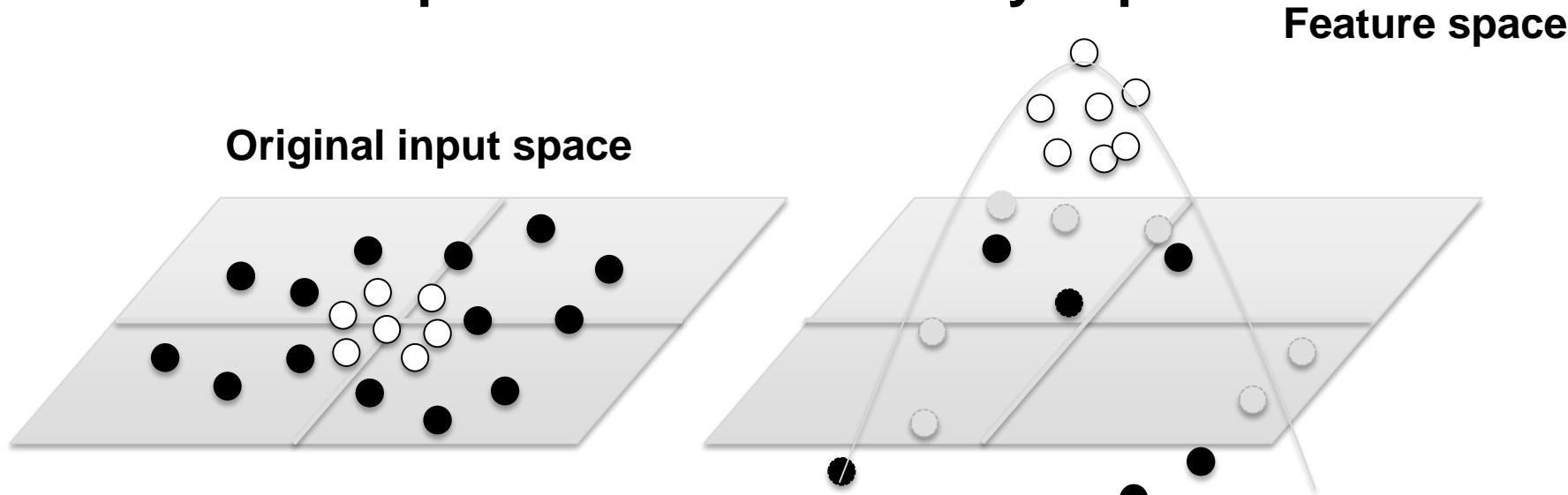
# Example of mapping a 2D classification problem to a 3D feature space to make it linearly separable

Original input space



$$x_i = (x_0, x_1)$$

# Example of mapping a 2D classification problem to a 3D feature space to make it linearly separable

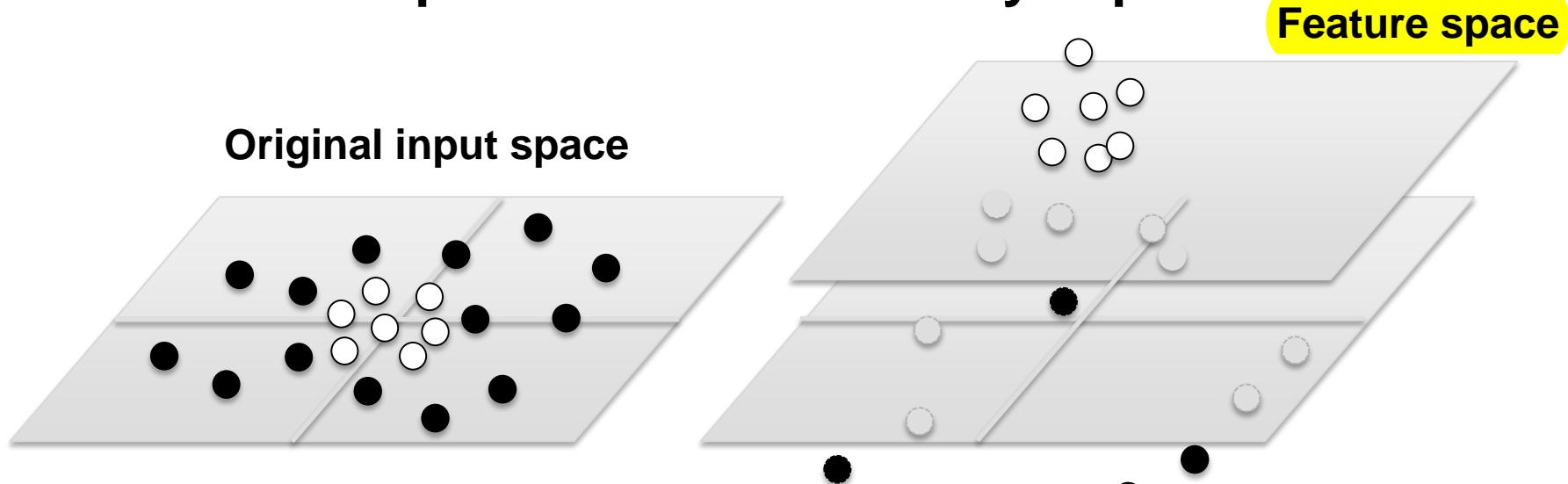


$$\mathbf{x}_i = (x_0, x_1)$$

$$\mathbf{v}_i = (x_0, x_1, 1 - (x_0^2 + x_1^2))$$

---

# Example of mapping a 2D classification problem to a 3D feature space to make it linearly separable

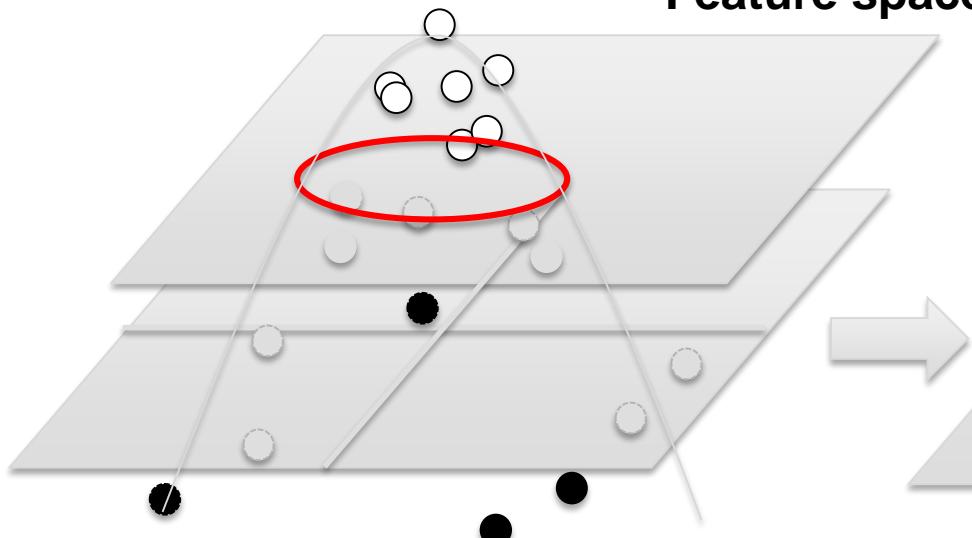


$$\mathbf{x}_i = (x_0, x_1)$$

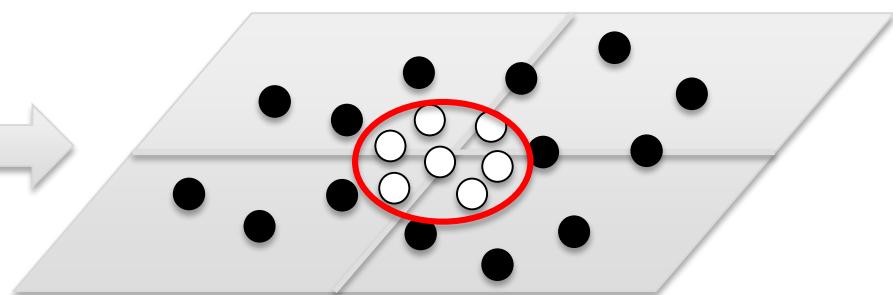
$$\mathbf{v}_i = (x_0, x_1, 1 - (x_0^2 + x_1^2))$$

# Example of mapping a 2D classification problem to a 3D feature space to make it linearly separable

Feature space



Original input space

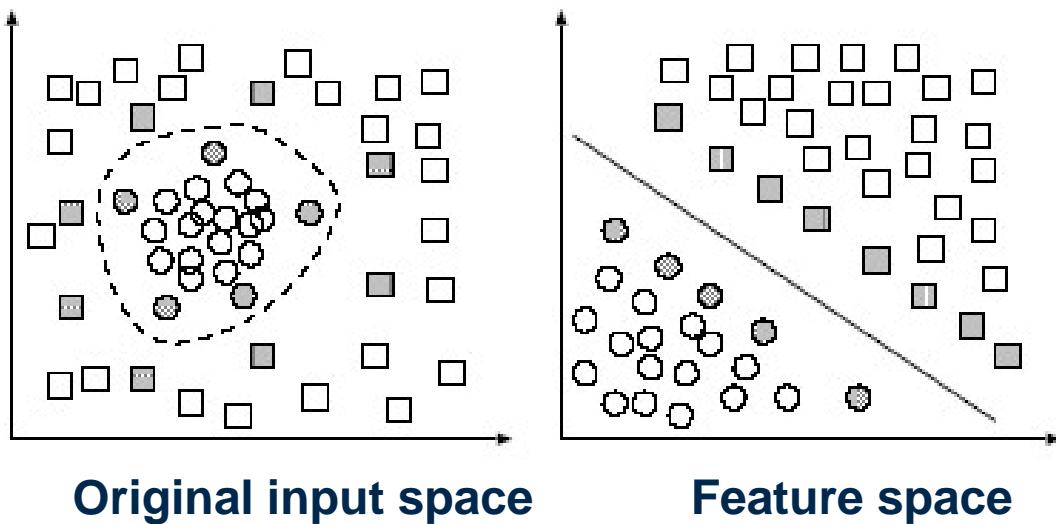


$$\nu_i = (x_0, x_1, 1 - (x_0^2 + x_1^2))$$

$$x_i = (x_0, x_1)$$

# Radial Basis Function Kernel

$$K(x, x') = \exp [-\gamma \cdot \|x - x'\|^2]$$

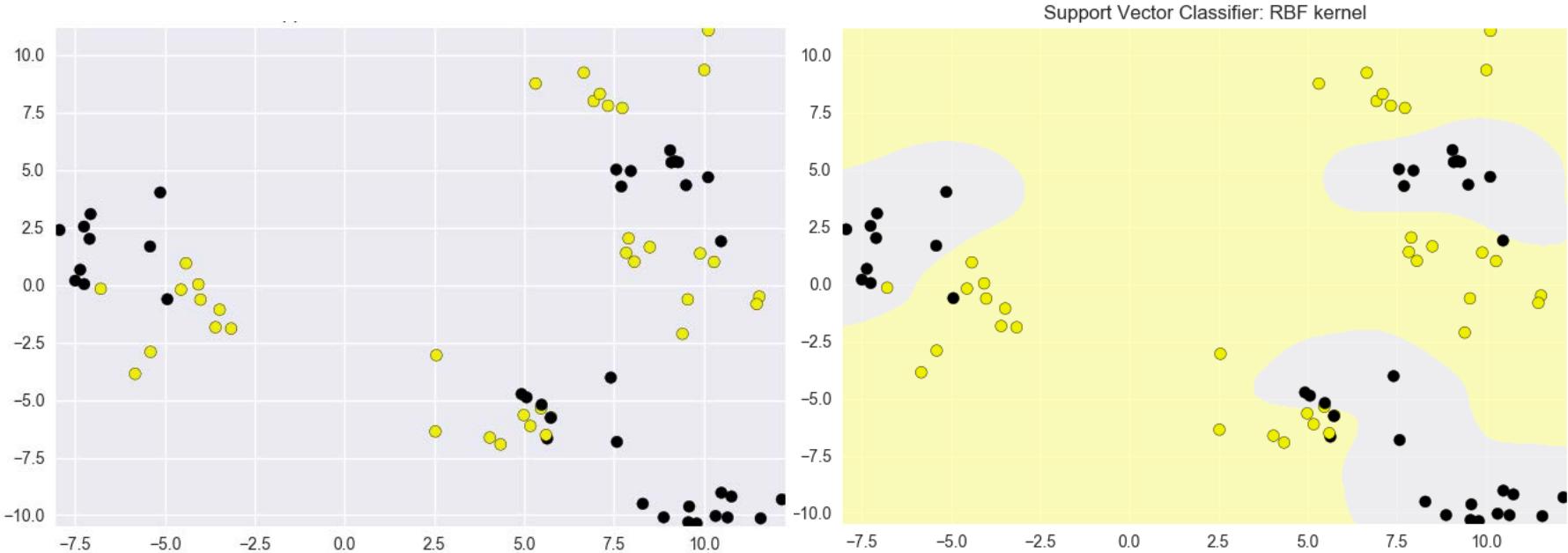


Original input space

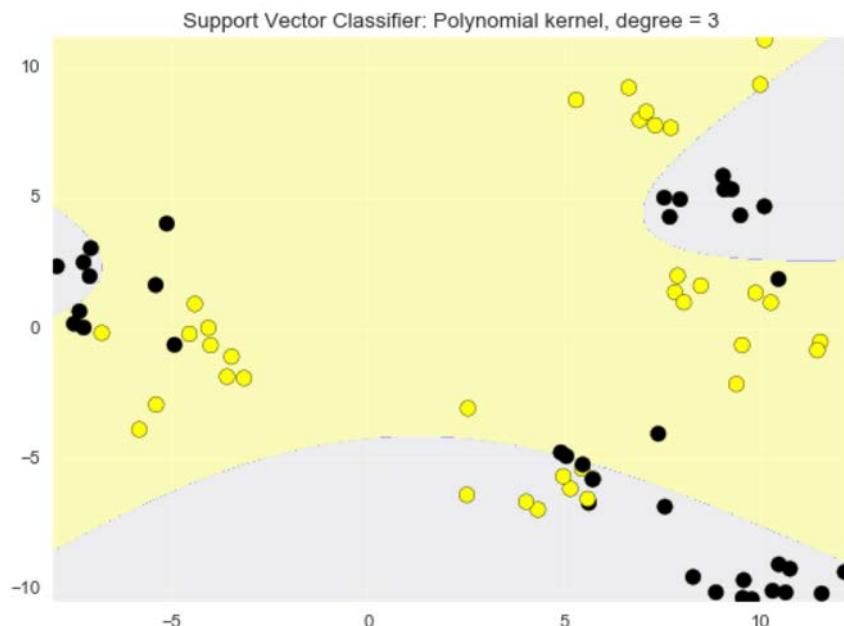
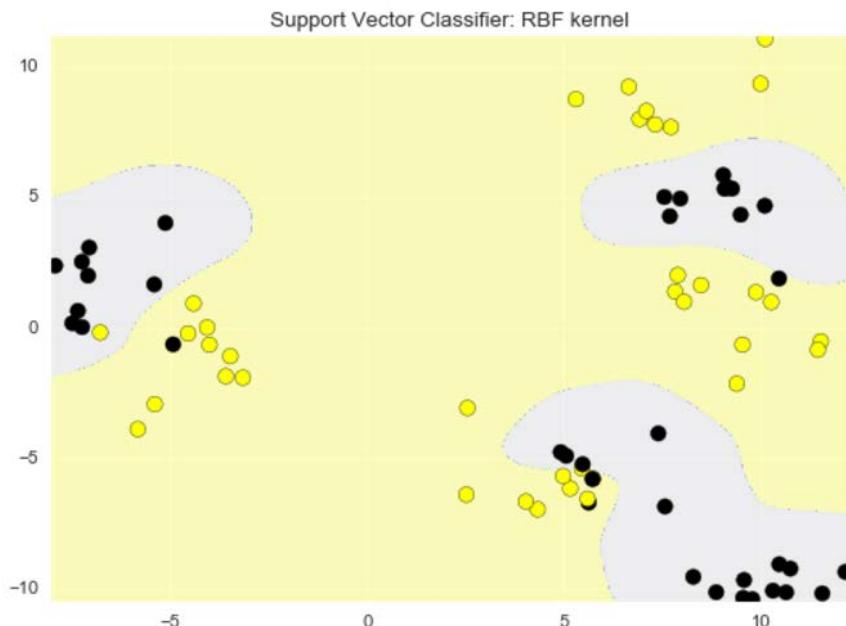
Feature space

A kernel is a similarity measure (modified dot product) between data points

# Applying the SVM with RBF kernel



# Radial Basis Kernel vs Polynomial Kernel

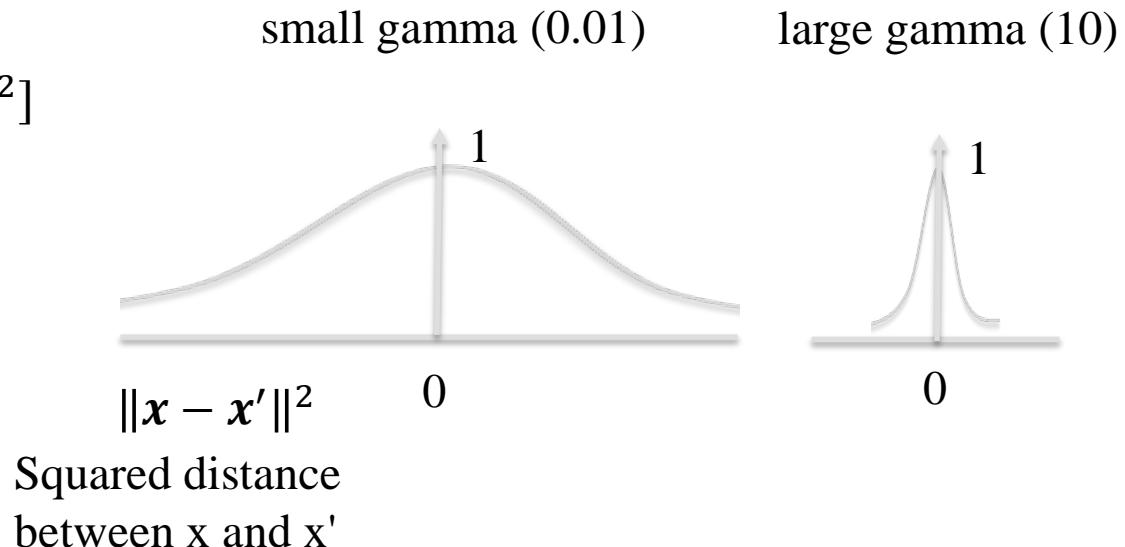


# Radial Basis Function kernel: Gamma Parameter

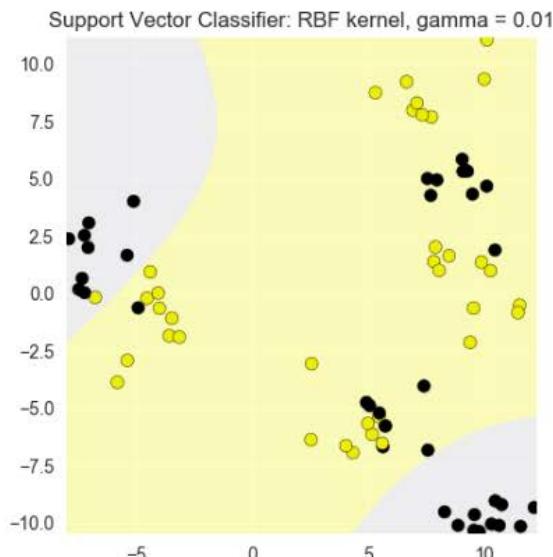
$$K(x, x') = \exp [-\gamma \cdot \|x - x'\|^2]$$



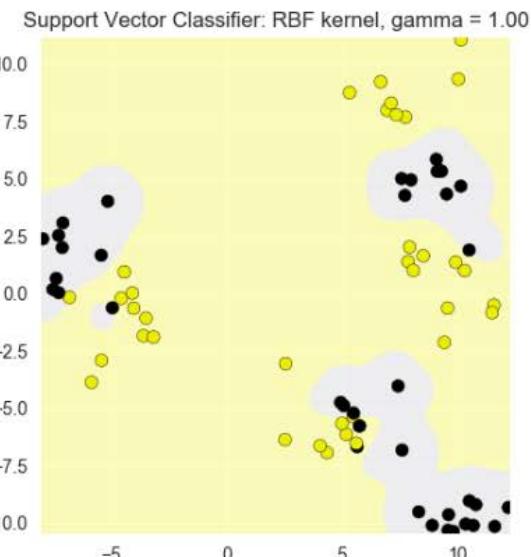
gamma ( $\gamma$ ): kernel width  
parameter



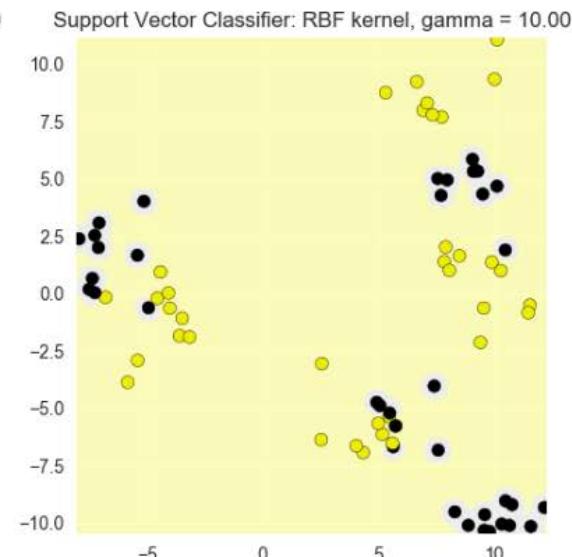
# The effect of the RBF gamma parameter on decision boundaries



gamma = 0.01

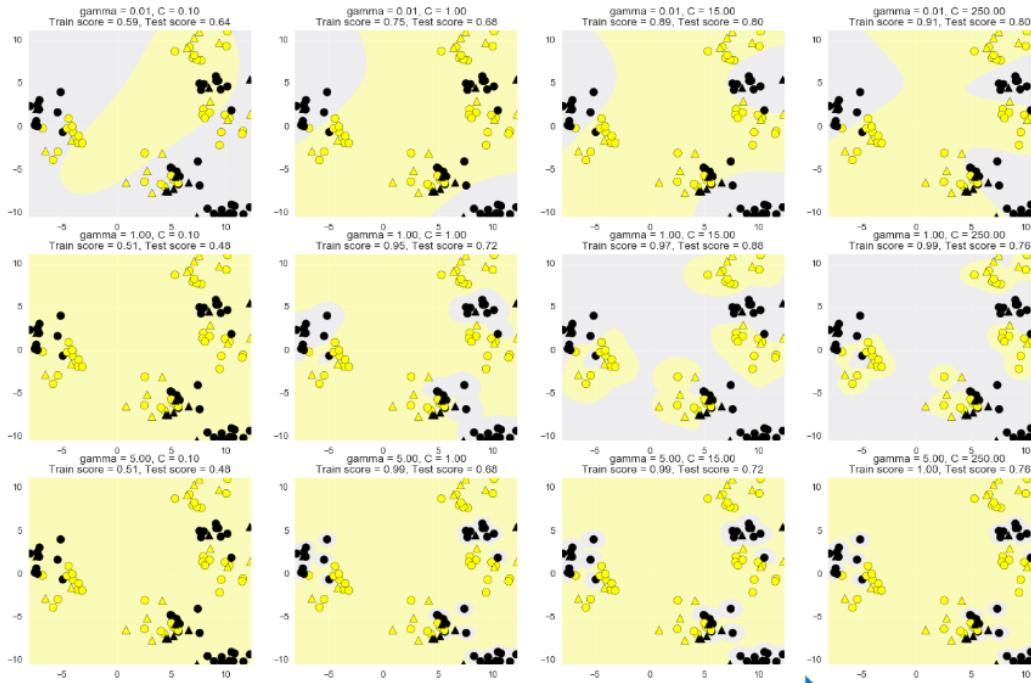


gamma = 1.0



gamma = 10

larger gamma, less effect of the C



Increasing gamma

Increasing C

## Reminder: Using a scaler object: fit and transform methods

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
clf = SVC().fit(X_train_scaled, y_train)
accuracy = clf.score(X_test_scaled, y_test)
```

**Tip:** It can be more efficient to do fitting and transforming together on the training set using the `fit_transform` method.

```
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

# Kernelized Support Vector Machines: pros and cons

## Pros:

- Can perform well on a range of datasets.
- Versatile: different kernel functions can be specified, or custom kernels can be defined for specific data types.
- Works well for both low- and high-dimensional data.

## Cons:

- Efficiency (runtime speed and memory usage) decreases as training set size increases (e.g. over 50000 samples).
- Needs careful normalization of input data and parameter tuning.
- Does not provide direct probability estimates (but can be estimated using e.g. Platt scaling).
- Difficult to interpret why a prediction was made.

# Kernelized Support Vector Machines (SVC): Important parameters

## Model complexity

- **kernel:** Type of kernel function to be used
  - Default = 'rbf' for radial basis function
  - Other types include 'polynomial'
- **kernel parameters**
  - gamma ( $\gamma$ ): RBF kernel width
- **C:** regularization parameter
- **Typically C and gamma are tuned at the same time.**



# Applied Machine Learning

## Cross-validation

Kevyn Collins-Thompson

Associate Professor of Information & Computer Science  
University of Michigan

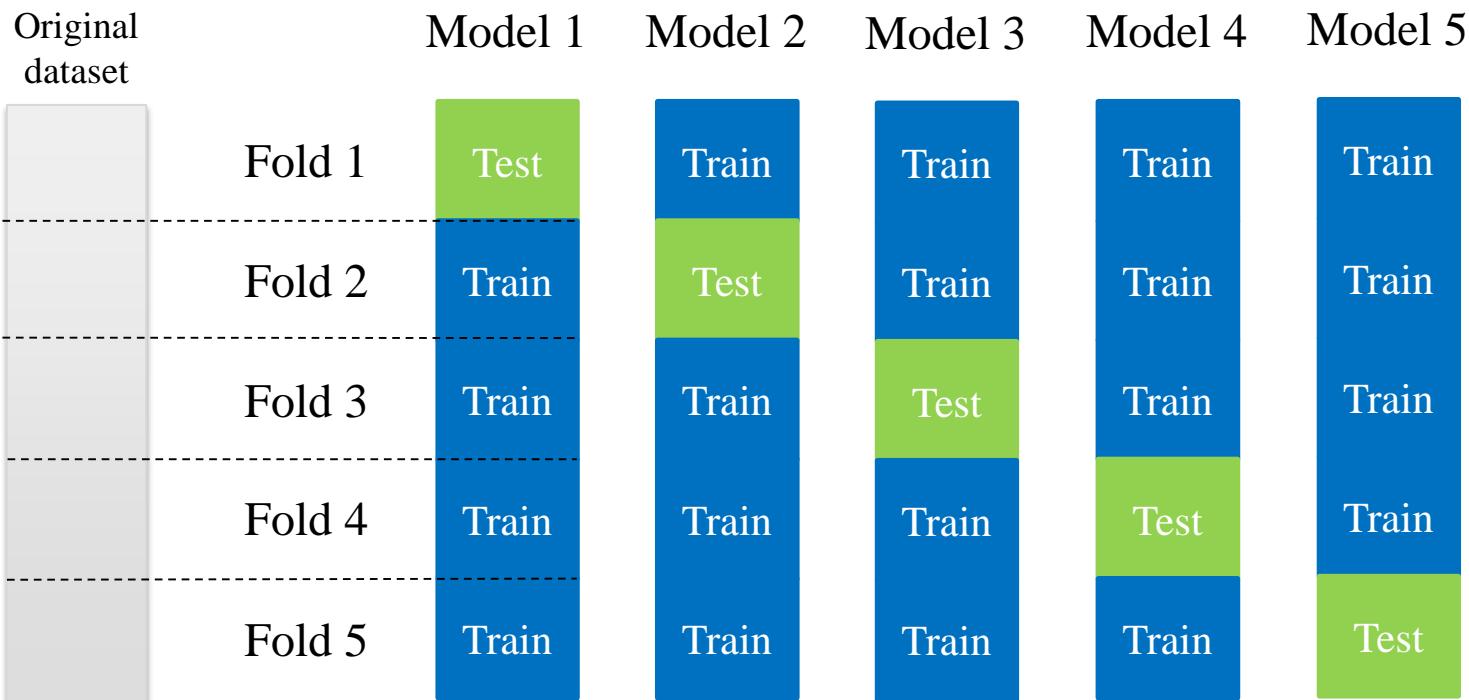
# Cross-validation

- **Uses multiple train-test splits, not just a single one**
- **Each split used to train & evaluate a separate model**
- **Why is this better?**
  - *The accuracy score of a supervised learning method can vary, depending on which samples happen to end up in the training set.*
  - *Using multiple train-test splits gives more stable and reliable estimates for how the classifier is likely to perform on average.*
  - *Results are averaged over multiple different training sets instead of relying on a single model trained on a particular training set.*

random_state	Test set accuracy
0	1.00
1	0.93
5	0.93
7	0.67
10	0.87

Accuracy of k-NN classifier (k=5) on fruit data test set for different random\_state values in train\_test\_split.

# Cross-validation Example (5-fold)



# Stratified Cross-validation

(Folds and dataset shortened for illustration purposes.)

fruit_label	fruit_name
1	Apple
2	Mandarin
...	...
3	Orange
...	...
4	Lemon

Example has 20 data samples  
= 4 classes with 5 samples each.

5-fold CV: 5 folds of 4 samples each.

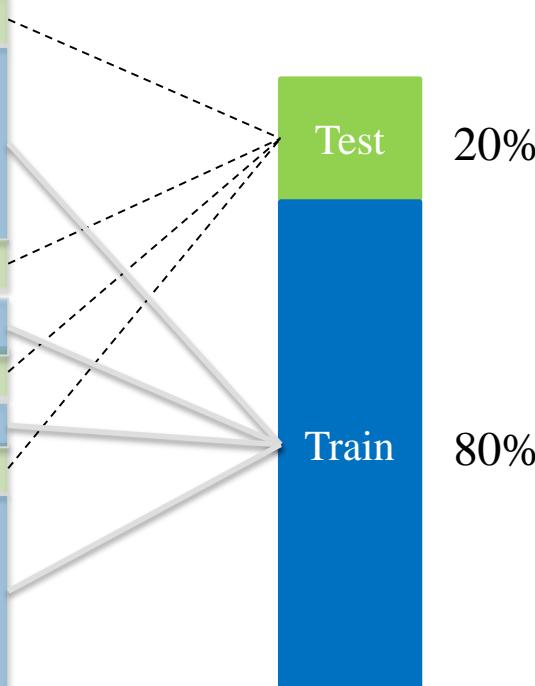
Fold 1 uses the first 20% of the dataset as the test set,  
which only contains samples from class 1.

Classes 2, 3, 4 are missing entirely from test set and so  
will be missing from the evaluation.

# Stratified Cross-validation

fruit_label	fruit_name
1	Apple
2	Mandarin
...	...
3	Orange
...	...
4	Lemon

Fold 1



20%

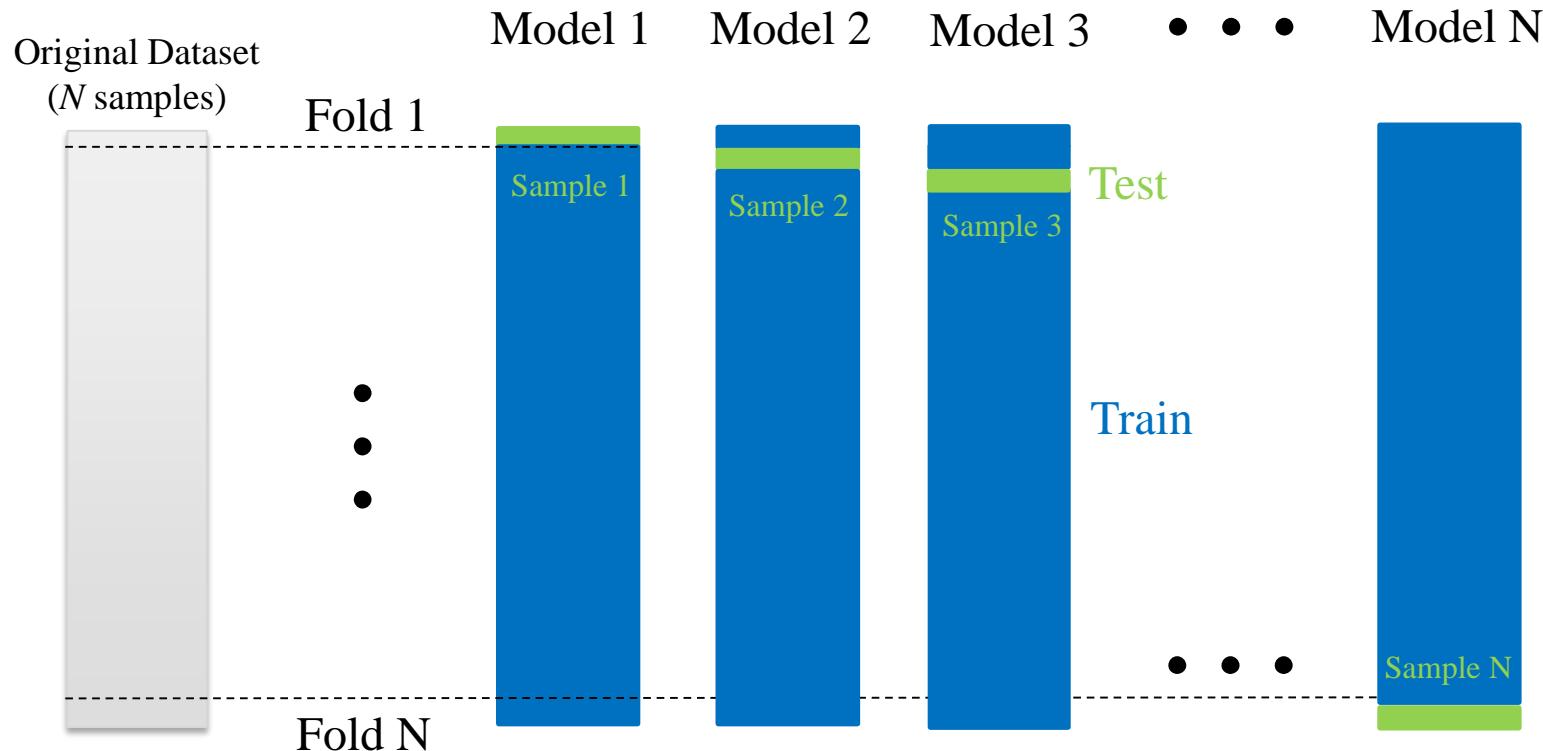
Test

Train

80%

Stratified folds each contain a proportion of classes that matches the overall dataset. Now, all classes will be fairly represented in the test set.

# Leave-one-out cross-validation (with $N$ samples in dataset)



# Validation curves show sensitivity to changes in an important parameter

```
from sklearn.svm import SVC
from sklearn.model_selection import validation_curve
param_range = np.logspace(-3, 3, 4)
train_scores, test_scores
= validation_curve(SVC(), X, y, param_name="gamma",
param_range=param_range, cv=5)
```

```
: print(train_scores)
```

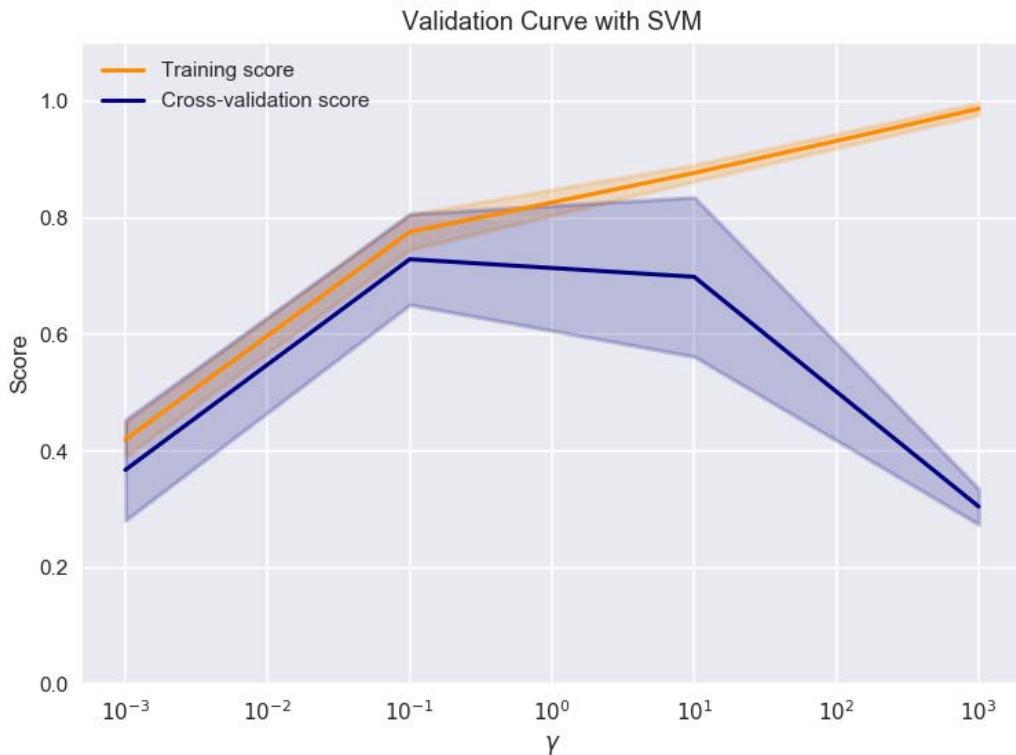
```
[[ 0.48648649  0.425      0.41463415]
 [ 0.83783784  0.725      0.75609756]
 [ 0.91891892  0.9        0.92682927]
 [ 1.          1.          0.97560976]]
```

```
: print(test_scores)
```

```
[[ 0.45454545  0.31578947  0.33333333]
 [ 0.81818182  0.68421053  0.61111111]
 [ 0.40909091  0.84210526  0.66666667]
 [ 0.36363636  0.21052632  0.38888889]]
```

One row per parameter sweep value,  
One column per CV fold.

# Validation Curve Example



The validation curve shows the mean cross-validation accuracy (solid lines) for training (orange) and test (blue) sets as a function of the SVM parameter ( $\gamma$ ). It also shows the variation around the mean (shaded region) as computed from  $k$ -fold cross-validation scores.



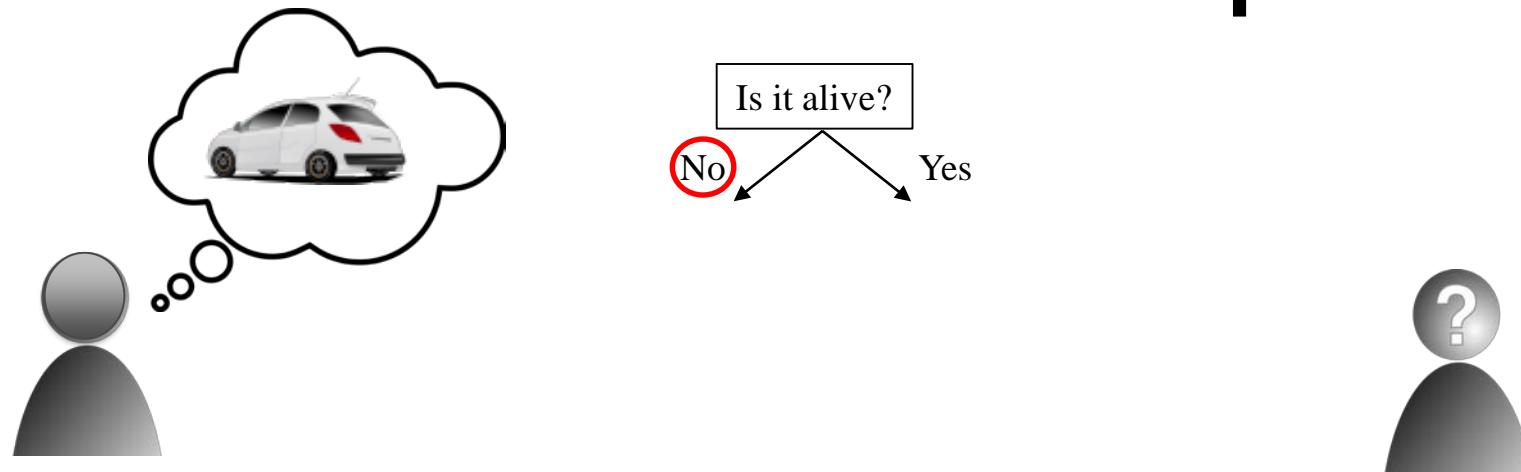
# Applied Machine Learning

## Decision Trees

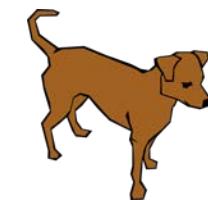
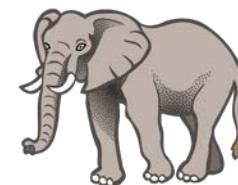
**Kevyn Collins-Thompson**

**Associate Professor of Information & Computer Science  
University of Michigan**

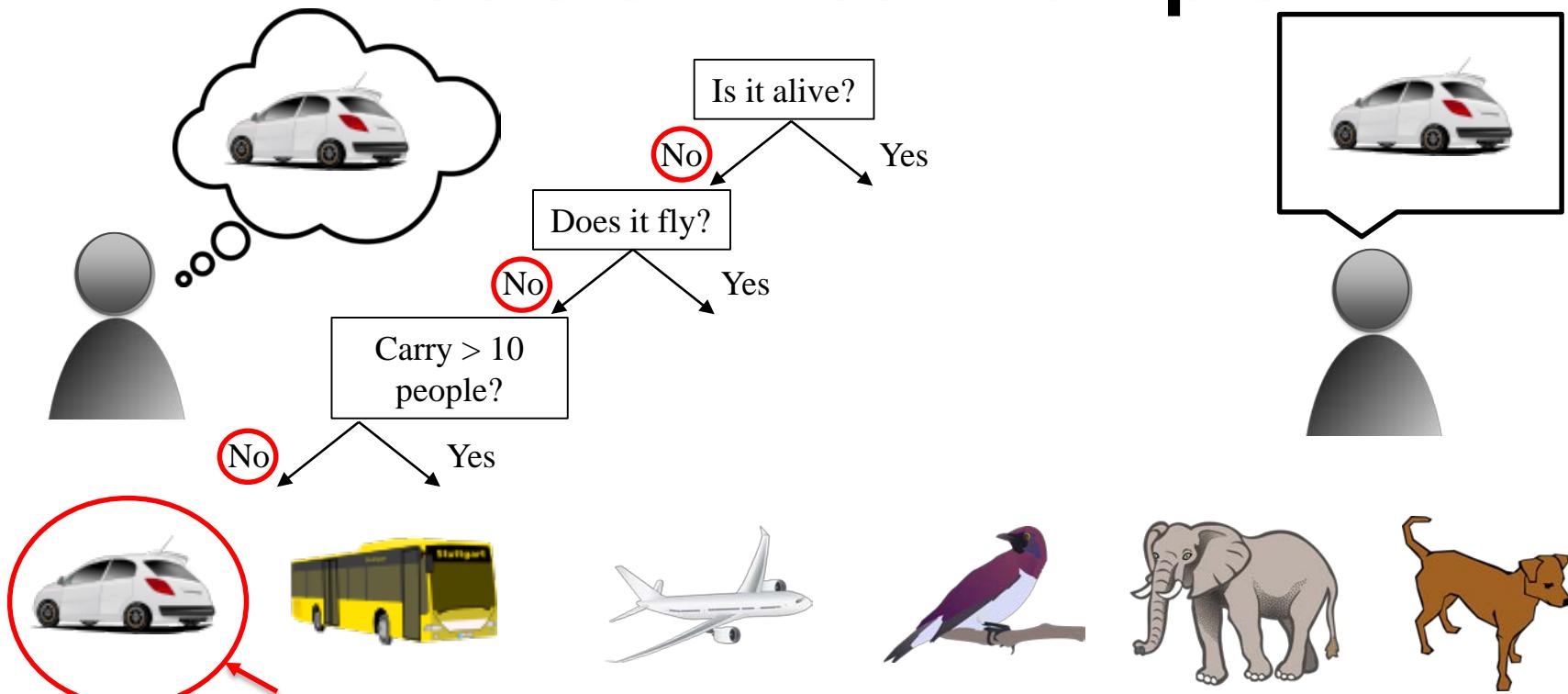
# Decision Tree Example



Objects with Alive == No



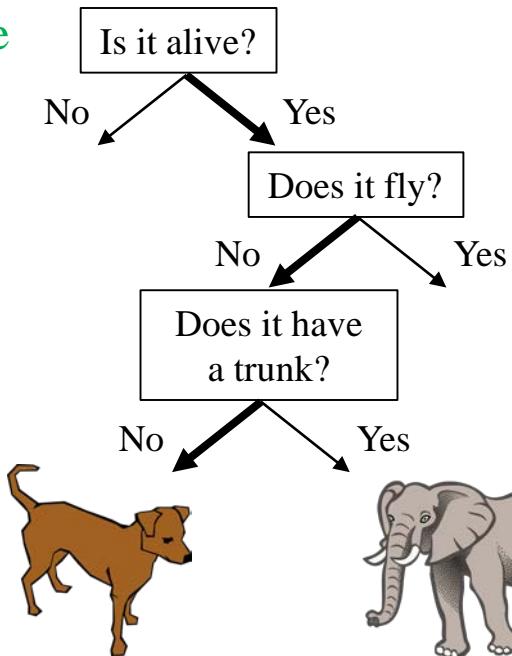
# Decision Tree Example



Objects with: Alive == No AND Fly == No AND Carry > 10 == No

# Decision Tree Example

Root node



Alive == Yes  
Fly == No  
Trunk == No

Leaf node

Alive == Yes  
Fly == No  
Trunk == Yes

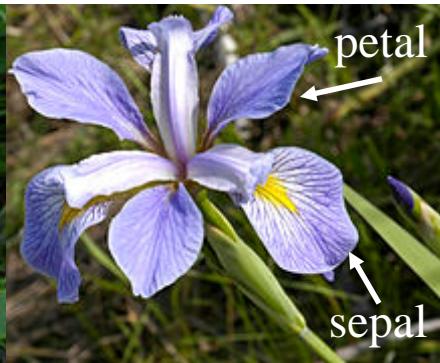
# The Iris Dataset



*Iris setosa*



*Iris versicolor*



*Iris virginica*

150 flowers  
3 species  
50 examples/species

# Decision Tree Splits

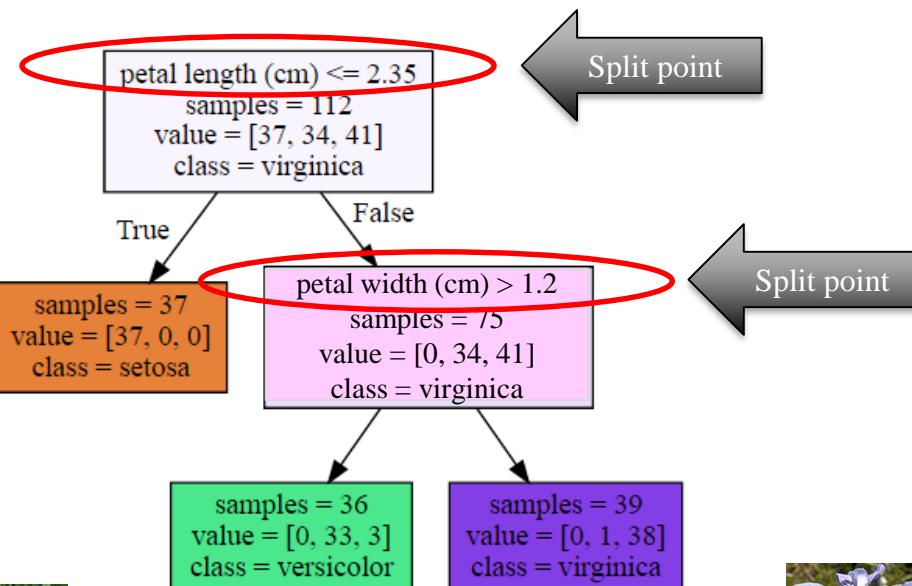
samples at this leaf have:  
**petal length <= 2.35**



*setosa*



*versicolor*



samples at this leaf have:  
**petal length > 2.35**  
**AND petal width <= 1.2**

samples at this leaf have:  
**petal length > 2.35**  
**AND petal width > 1.2**

*virginica*

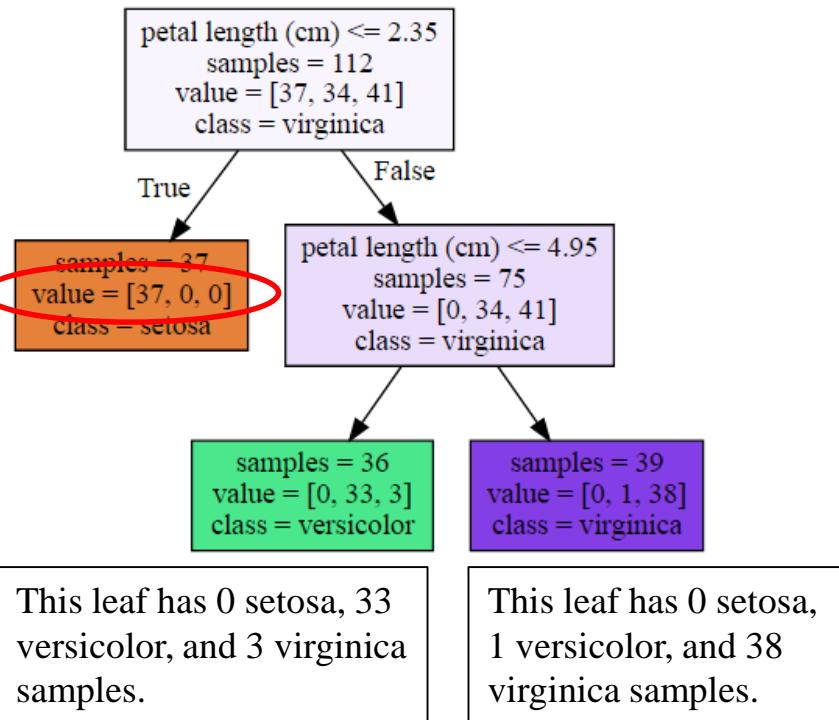


# Informativeness of Splits

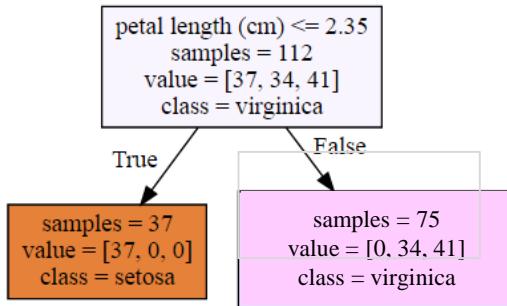
The *value* list gives the number of samples of each class that end up at this leaf node during training.

The iris dataset has 3 classes, so there are three counts.

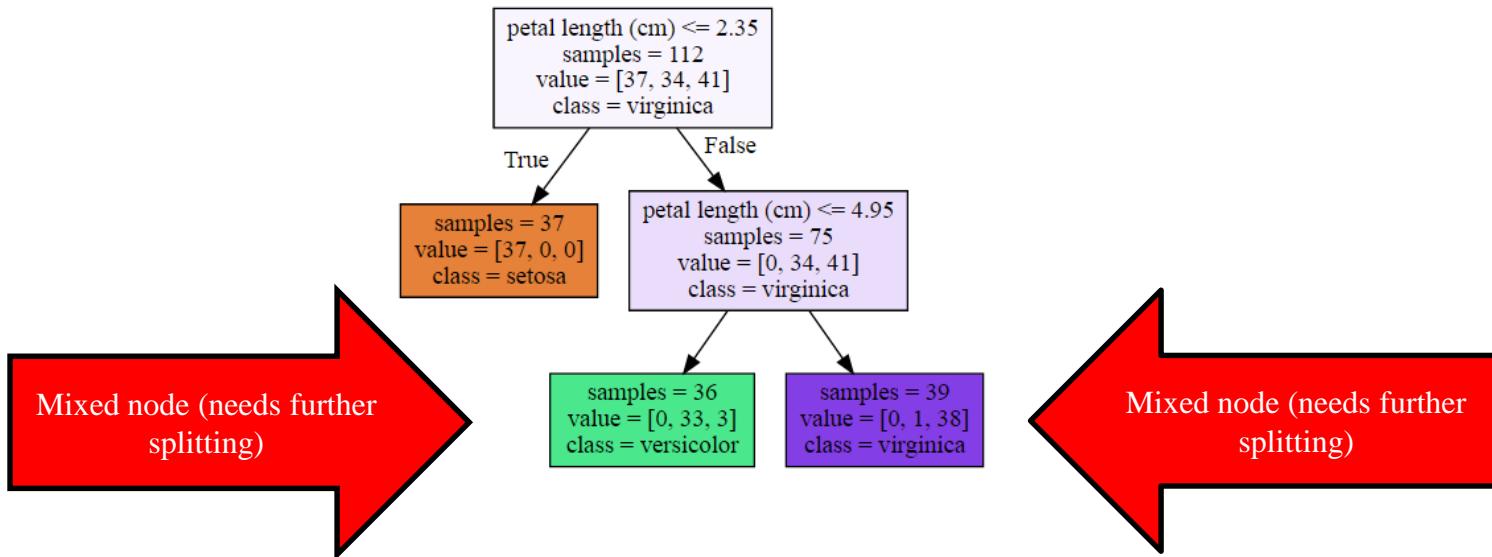
This leaf has 37 setosa samples, zero versicolor, and zero virginica samples.



Pure node (all one class:  
perfect classification)



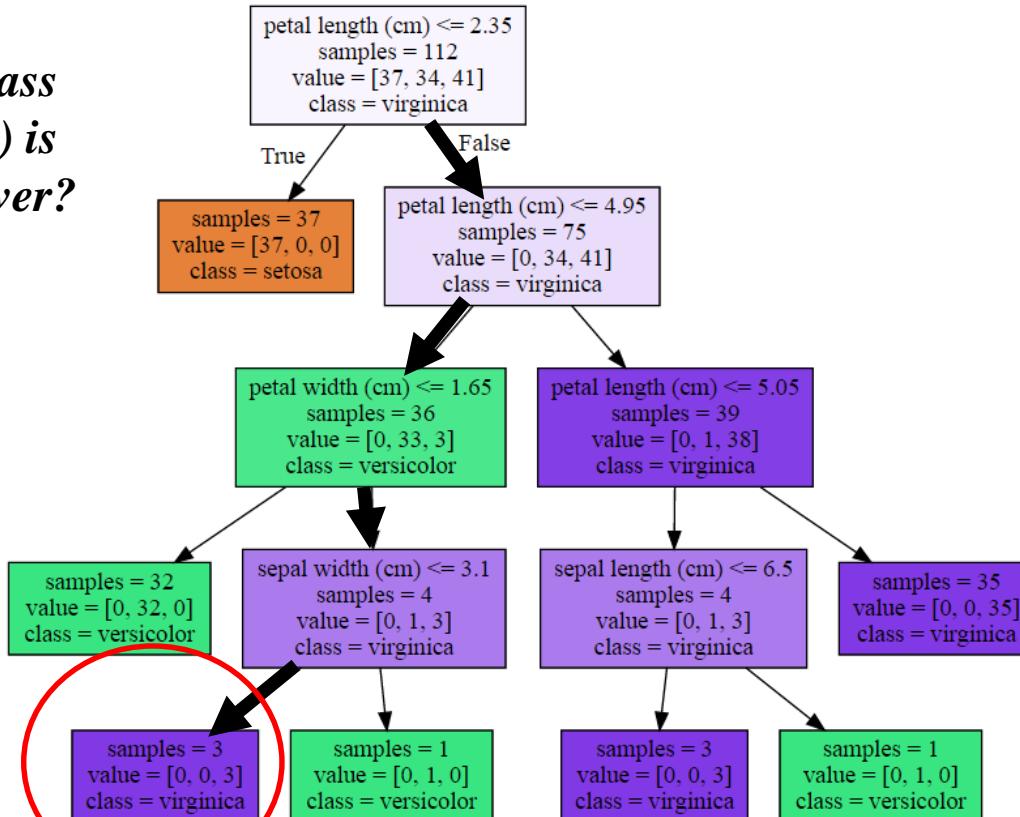
Mixed node (mixture of  
classes, still needs further  
splitting)





*What class  
(species) is  
this flower?*

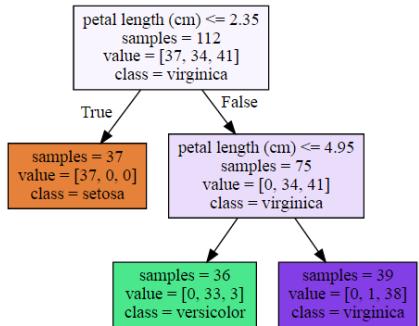
petal length: 3.0  
petal width: 2.0  
sepal width: 2.0  
sepal length: 4.2



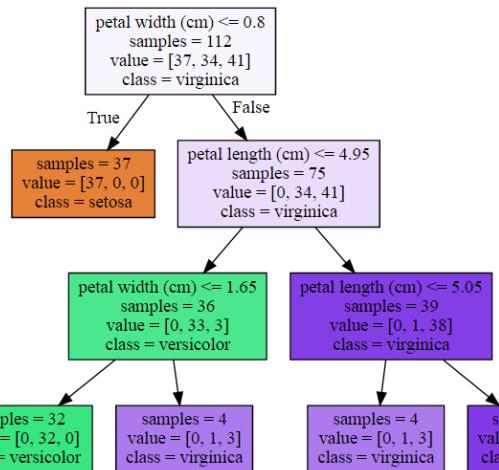
Leaf counts are: setosa = 0, versicolor = 0, virginica = 3  
Predicted class is majority class at this leaf: **virginica**

# Controlling the Model Complexity of Decision Trees

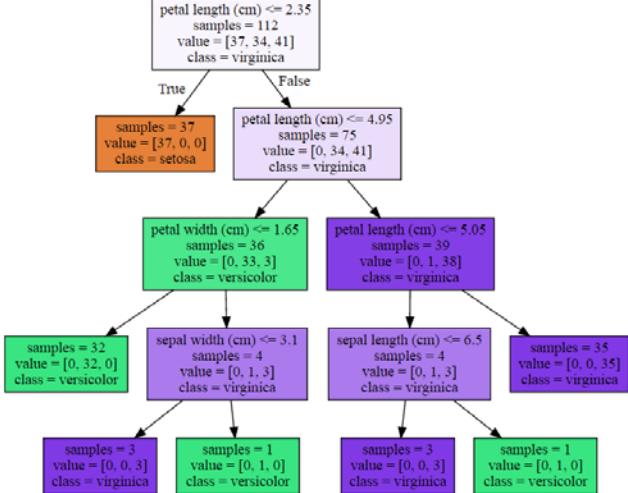
`max_depth = 2`



`max_depth = 3`



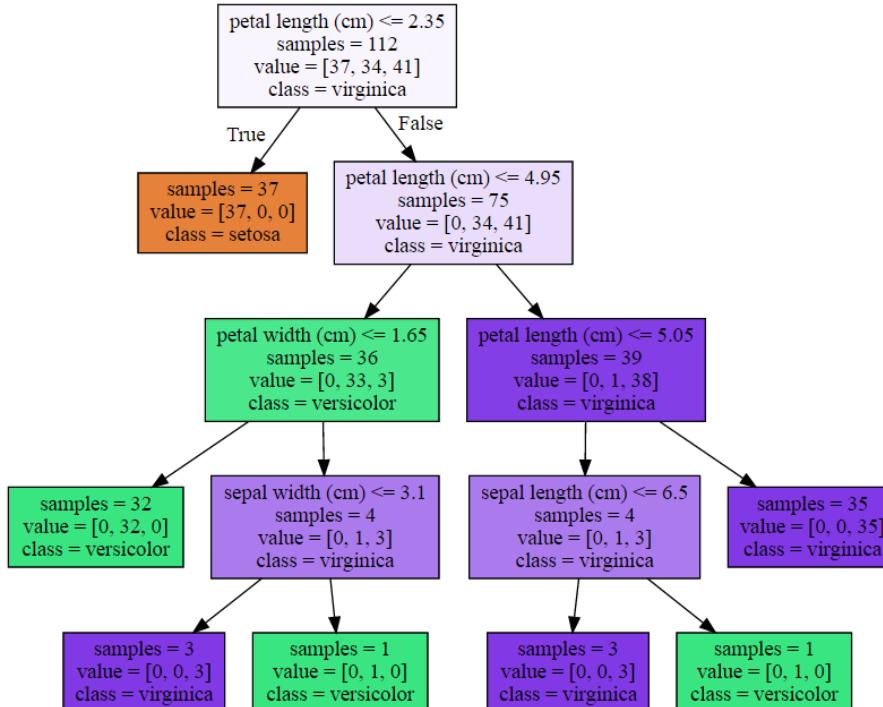
`max_depth = 4`



Other parameters: Max. # of leaf nodes: `max_leaf_nodes`

Min. samples to consider splitting: `min_samples_leaf`

# Visualizing Decision Trees

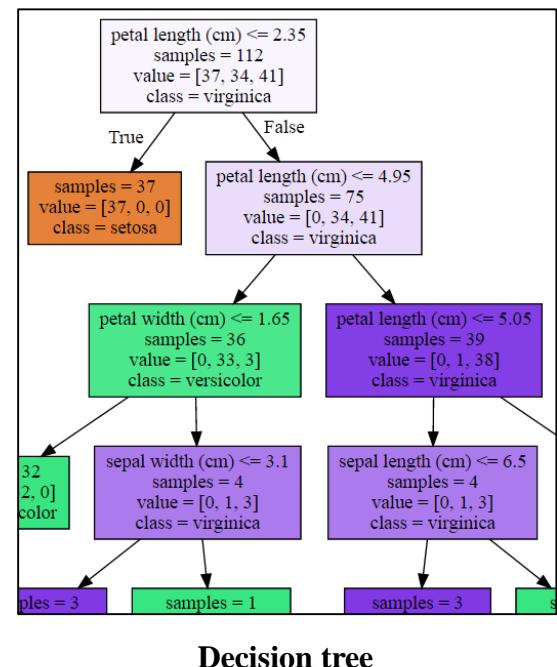
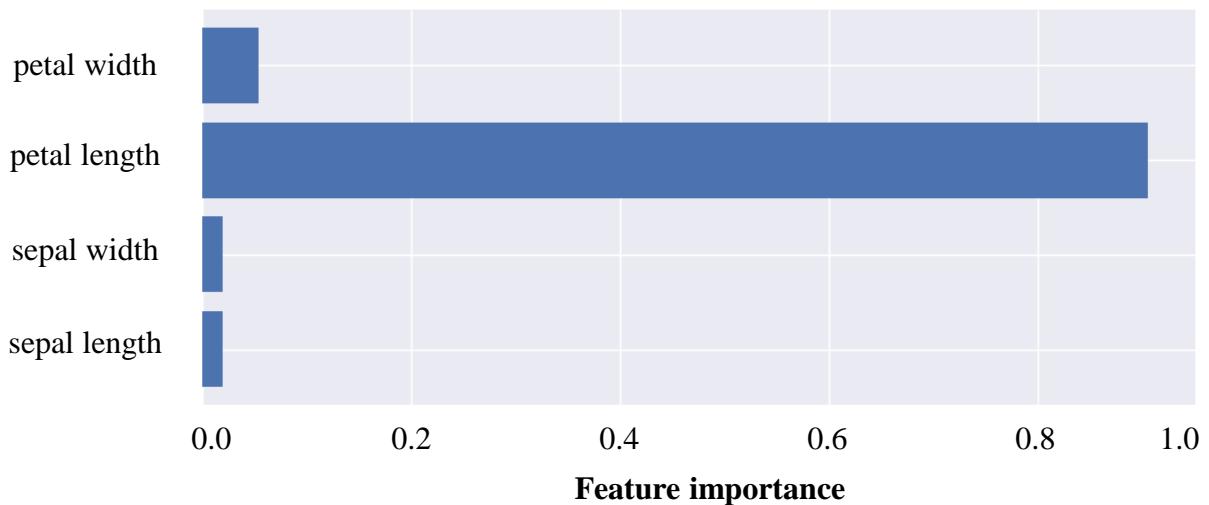


See: `plot_decision_tree()` function in `adspy_shared_utilities.py` code

# Feature Importance: How important is a feature to overall prediction accuracy?

- A number between 0 and 1 assigned to each feature.
- Feature importance of 0 → the feature was not used in prediction.  
→
- Feature importance of 1 → the feature predicts the target perfectly.
- All feature importances are normalized to sum to 1.

# Feature Importance Chart



See: `plot_feature_importances()` function in `adspy_shared_utilities.py` code

# Decision Trees: Pros and Cons

## Pros:

- Easily visualized and interpreted.
- No feature normalization or scaling typically needed.
- Work well with datasets using a mixture of feature types (continuous, categorical, binary)

## Cons:

- Even after tuning, decision trees can often still overfit.
- Usually need an ensemble of trees for better generalization performance.

# Decision Trees: DecisionTreeClassifier Key Parameters

- `max_depth`: controls maximum depth (number of split points). Most common way to reduce tree complexity and overfitting.
- `min_samples_leaf`: threshold for the minimum # of data instances a leaf can have to avoid further splitting.
- `max_leaf_nodes`: limits total number of leaves in the tree.
- In practice, adjusting only one of these (e.g. `max_depth`) is enough to reduce overfitting.