Classification:
k-Nearest Neighbor &

Parametric vs. Nonparametric

Key difference:

- Parametric models assume that the data can be characterized via some fixed set of parameters θ. Given this set of parameters, our future predictions are independent of the data $\mathcal{D}$, i.e., $P(x|\theta, \mathcal{D}) = P(x|\theta)$.
  - Often simpler and faster to learn, but can sometimes be a poor fit
- Nonparametric models instead assume that the model features depend on the data $\mathcal{D}$. The number of features tends to grow with the size of the dataset.
  - More complex and expensive, but can learn more flexible patterns
- Both parametric and non-parametric methods can be used for either regression or classification.

Ex: Iris data (click here for all data)

- 4 features
- 3 classes

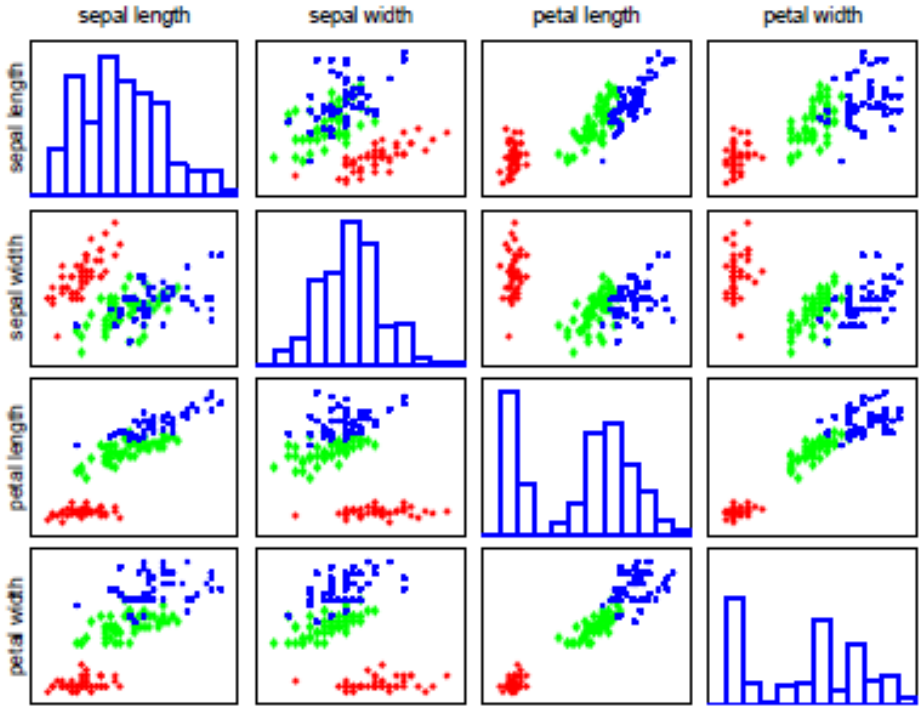
Fisher's *Iris* Data

Sepal length	Sepal width	Petal length	Petal width	Species
5.1	3.5	1.4	0.2	*I. setosa*
4.9	3.0	1.4	0.2	*I. setosa*
4.7	3.2	1.3	0.2	*I. setosa*
4.6	3.1	1.5	0.2	*I. setosa*
5.0	3.6	1.4	0.2	*I. setosa*
5.4	3.9	1.7	0.4	*I. setosa*
4.6	3.4	1.4	0.3	*I. setosa*
5.0	3.4	1.5	0.2	*I. setosa*
4.4	2.9	1.4	0.2	*I. setosa*
4.9	3.1	1.5	0.1	*I. setosa*

Visualization of data helps to identify the right learning model

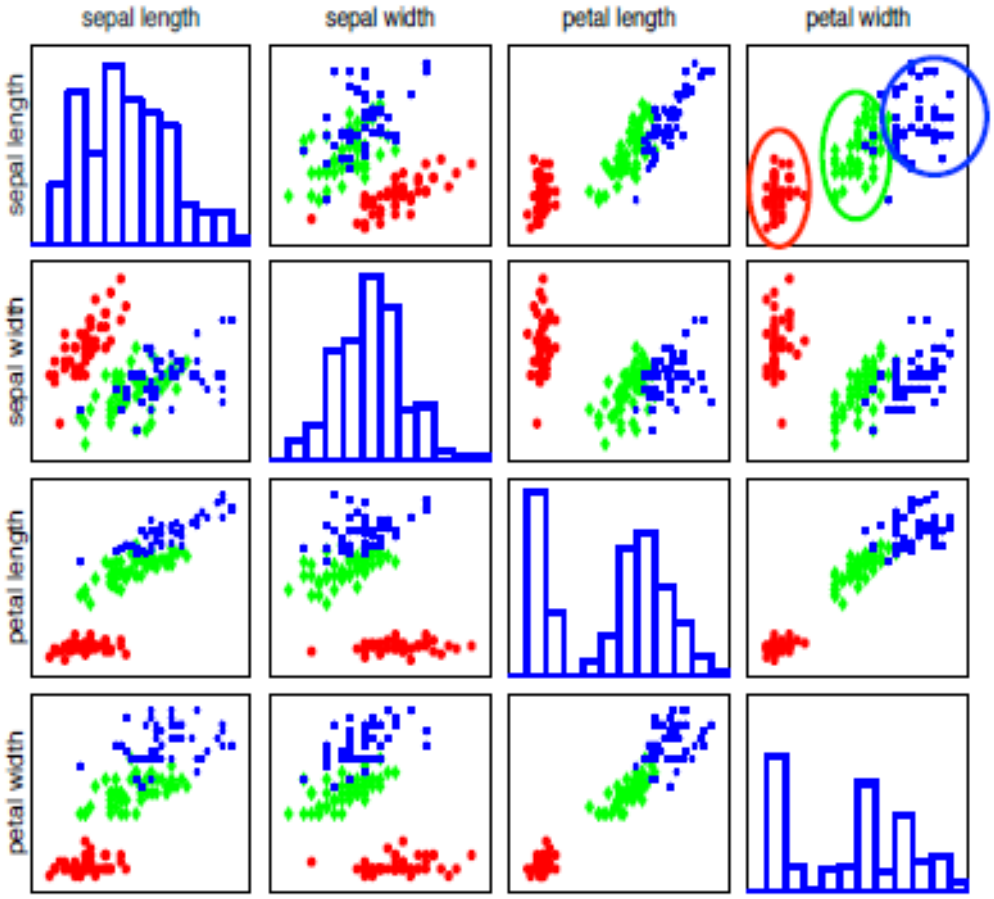
Which combination of features separates the three classes?

Figure 1: Each colored point is a flower specimen: setosa, versicolor, virginica

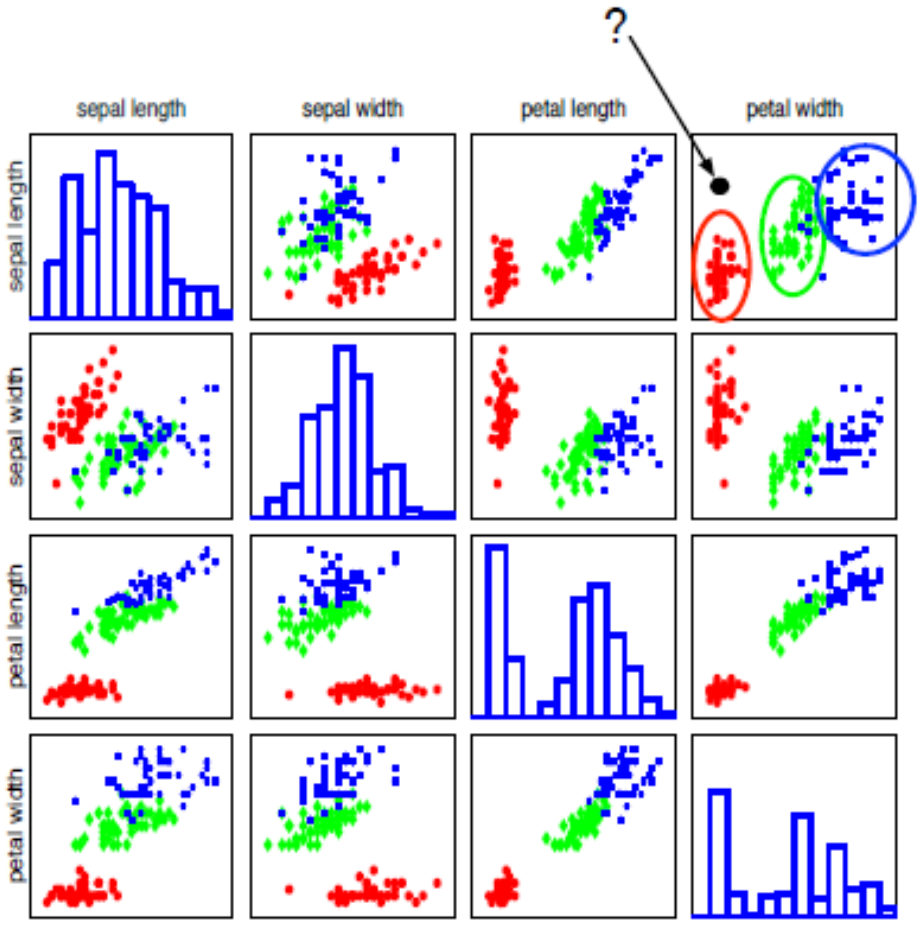


Different types seem well-clustered and separable

Using two features: petal width and sepal length



Labeling an unknown flower type

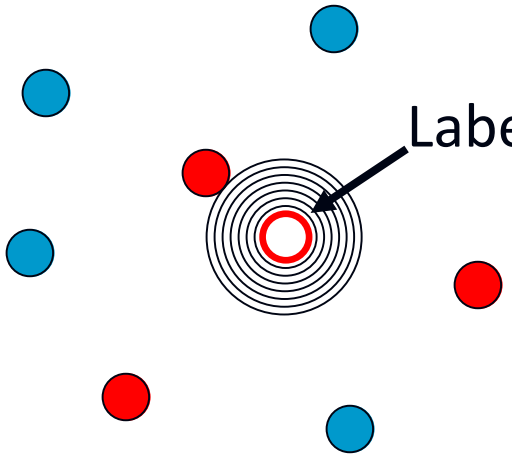


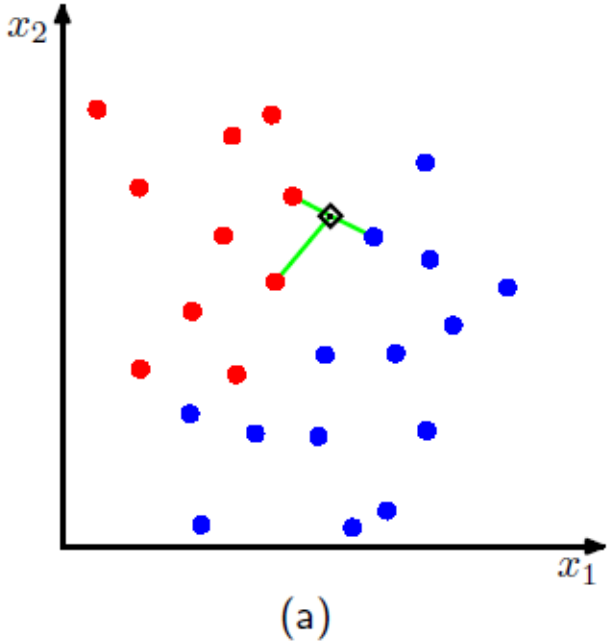
Closer to red cluster: so labeling it as setosa

1-Nearest Neighbor

- One of the simplest of all machine learning classifiers
- Simple idea: label a new point the same as the closest known point

In this 2-dimensional example, the nearest point to x is a red training instance, thus, *x* will be labeled as red.





How to Find Nearest Point

Nearest neighbor of a (test) data point

$$\boldsymbol{x}(1) = \boldsymbol{x}_{\mathrm{nn}(\boldsymbol{x})}$$

where $\mathrm{nn}(\boldsymbol{x}) \in [\mathrm{N}] = \{1, 2, \cdots, \mathrm{N}\}$, i.e., the index to one of the training instances

$$\mathrm{nn}(\boldsymbol{x}) = \mathrm{argmin}_{n \in [\mathrm{N}]} \|\boldsymbol{x} - \boldsymbol{x}_n\|_2^2 = \mathrm{argmin}_{n \in [\mathrm{N}]} \sum_{d=1}^{\mathrm{D}} (x_d - x_{nd})^2$$

Classification rule

$$y = f(\boldsymbol{x}) = y_{\mathrm{nn}(\boldsymbol{x})}$$

Example: if $\mathrm{nn}(\boldsymbol{x}) = 2$, then

$$y_{\mathrm{nn}(\boldsymbol{x})} = y_2,$$

which is the label of the 2nd data point.

Training data

ID (n)	petal width (x_1)	sepal length (x_2)	category (y)
1	0.2	5.1	setosa
2	1.4	7.0	versicolor
3	2.5	6.7	virginica

Flower with unknown category

petal width $= 1.8$ and sepal length $= 6.4$

Calculating distance from (x_1, x_2) to (x_{n1}, x_{n2}): $(x_1 - x_{n1})^2 + (x_2 - x_{n2})^2$

ID	distance
1	4.25
2	0.52
3	0.58

Thus, the predicted category is 2 (*versicolor*)

Distance Metrics

Previously, we used the Euclidean distance

$$\text{nn}(\boldsymbol{x}) = \text{argmin}_{n \in [N]} \|\boldsymbol{x} - \boldsymbol{x}_n\|_2^2$$

We can also use alternative distances

E.g., the following L_1 distance (i.e., city block distance, or Manhattan distance)

$$\begin{aligned} \text{nn}(\boldsymbol{x}) &= \text{argmin}_{n \in [N]} \|\boldsymbol{x} - \boldsymbol{x}_n\|_1 \\ &= \text{argmin}_{n \in [N]} \sum_{d=1}^{D} |x_d - x_{nd}| \end{aligned}$$

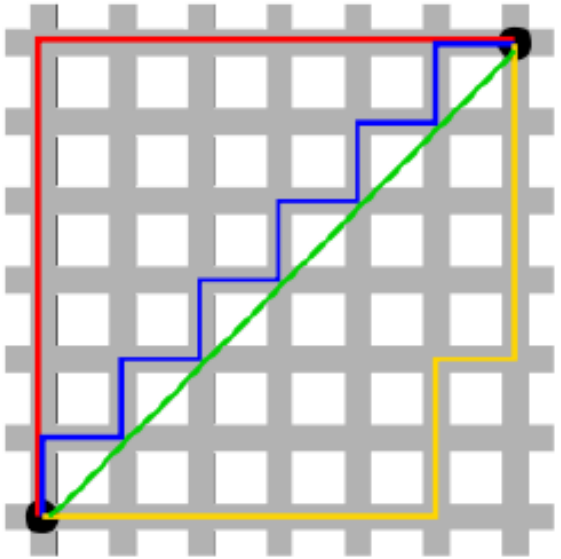
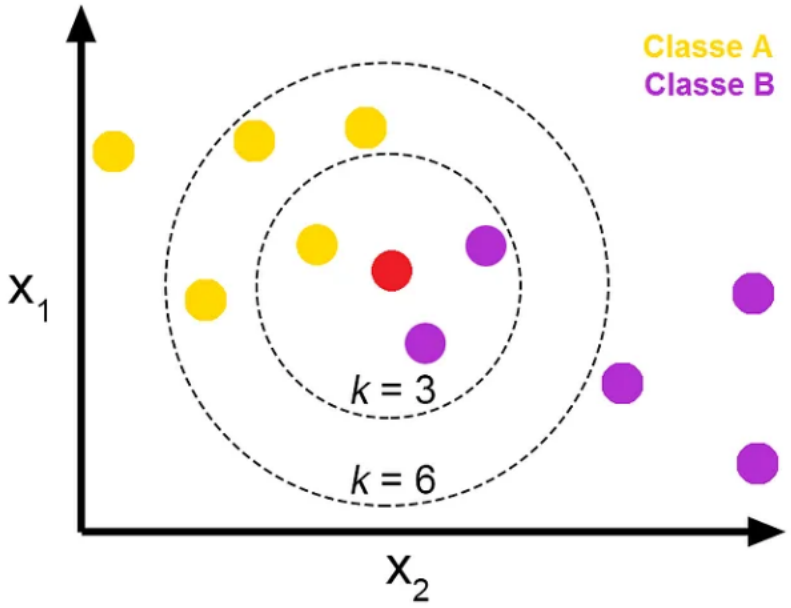


Figure 2: Green line is Euclidean distance. Red, Blue, and Yellow lines are L_1 distance

Choosing from K-nearest Neighbor



Increase the number of nearest neighbors to use?

- 1-nearest neighbor: $\text{nn}_1(\boldsymbol{x}) = \text{argmin}_{n\in[\text{N}]} \|\boldsymbol{x} - \boldsymbol{x}_n\|_2^2$
- 2nd-nearest neighbor: $\text{nn}_2(\boldsymbol{x}) = \text{argmin}_{n\in[\text{N}]-\text{nn}_1(\boldsymbol{x})} \|\boldsymbol{x} - \boldsymbol{x}_n\|_2^2$
- 3rd-nearest neighbor: $\text{nn}_2(\boldsymbol{x}) = \text{argmin}_{n\in[\text{N}]-\text{nn}_1(\boldsymbol{x})-\text{nn}_2(\boldsymbol{x})} \|\boldsymbol{x} - \boldsymbol{x}_n\|_2^2$

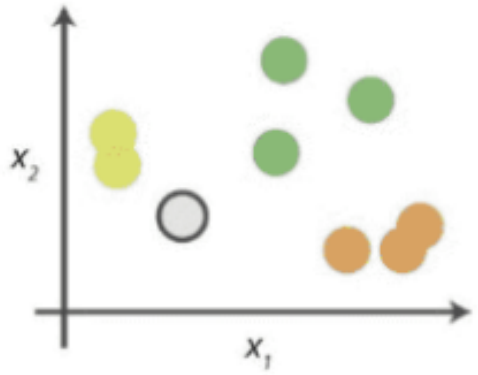
The set of K-nearest neighbors

$$\text{knn}(\boldsymbol{x}) = \{\text{nn}_1(\boldsymbol{x}), \text{nn}_2(\boldsymbol{x}), \cdots, \text{nn}_K(\boldsymbol{x})\}$$

Let $\boldsymbol{x}(k) = \boldsymbol{x}_{\text{nn}_k(\boldsymbol{x})}$, then

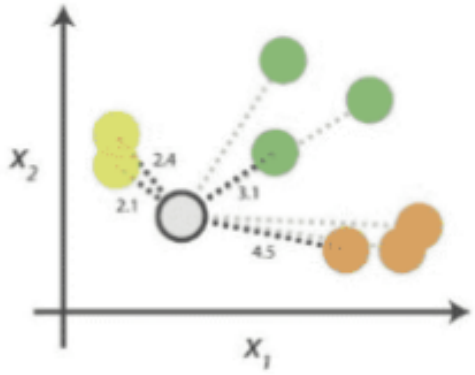
$$\arg\max(\|\boldsymbol{x} - \boldsymbol{x}(1)\|^2{}_2 \leq \|\boldsymbol{x} - \boldsymbol{x}(2)\|^2{}_2 \cdots \leq \|\boldsymbol{x} - \boldsymbol{x}(K)\|^2{}_2)$$

0. Look at the data



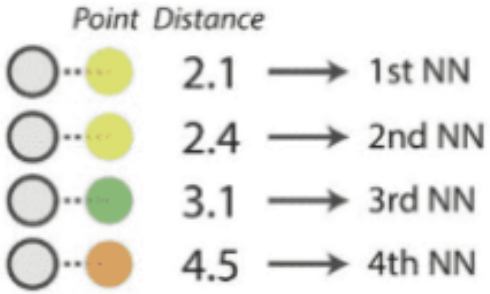
Say you want to classify the grey point into a class. Here, there are three potential classes - lime green, green and orange.

1. Calculate distances



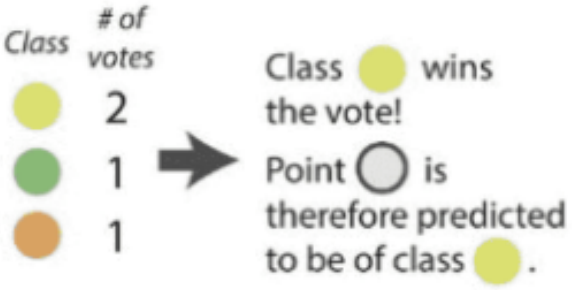
Start by calculating the distances between the grey point and all other points.

2. Find neighbours



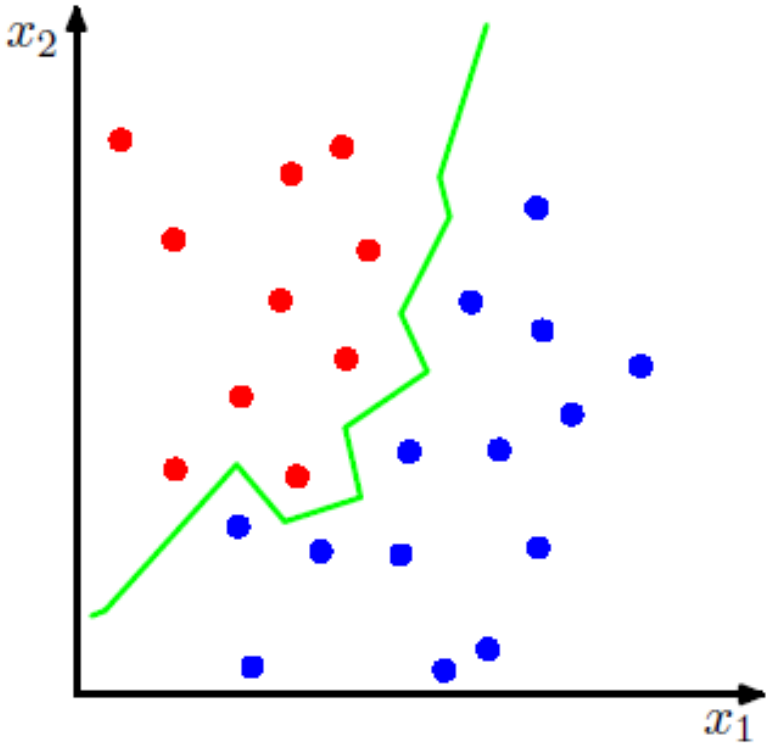
Next, find the nearest neighbours by ranking points by increasing distance. The nearest neighbours (NNs) of the grey point are the ones closest in dataspace.

3. Vote on labels

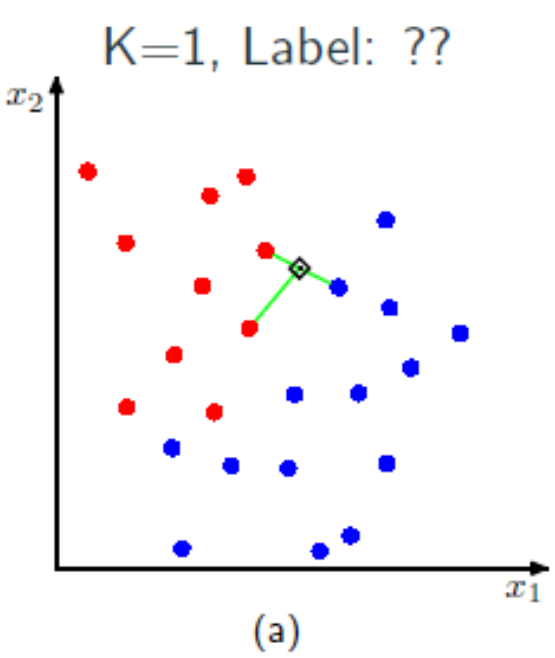


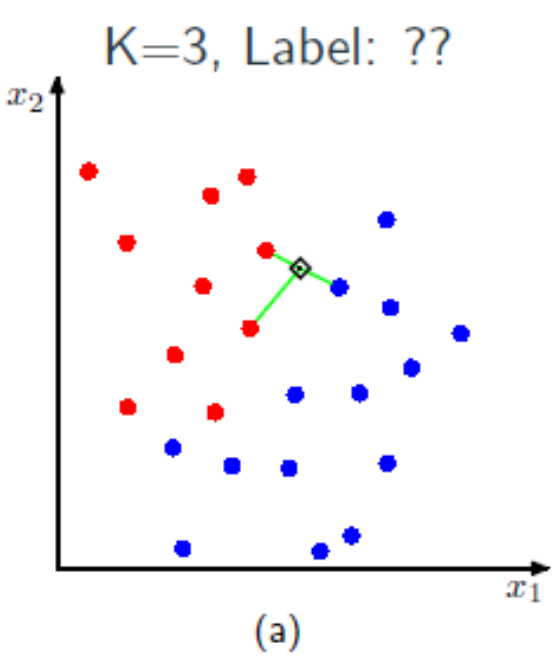
Vote on the predicted class labels based on the classes of the k nearest neighbours. Here, the labels were predicted based on the k=3 nearest neighbours.

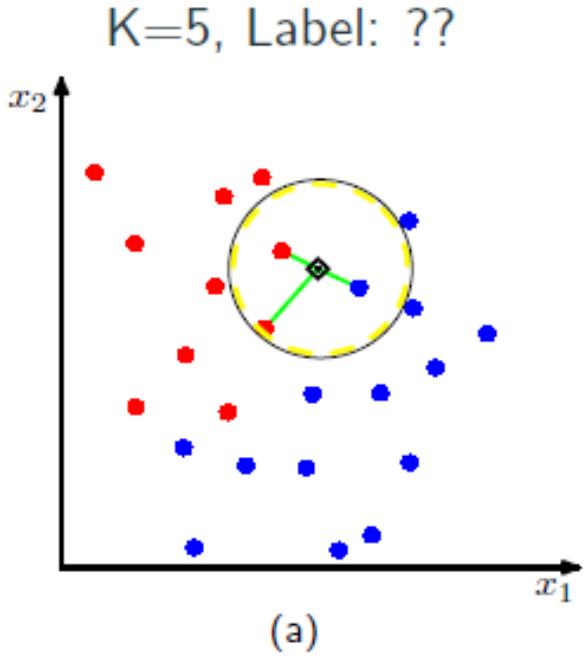
For every point in the space, we can determine its label using the NNC rule. This gives rise to a *decision boundary* that partitions the space into different regions.



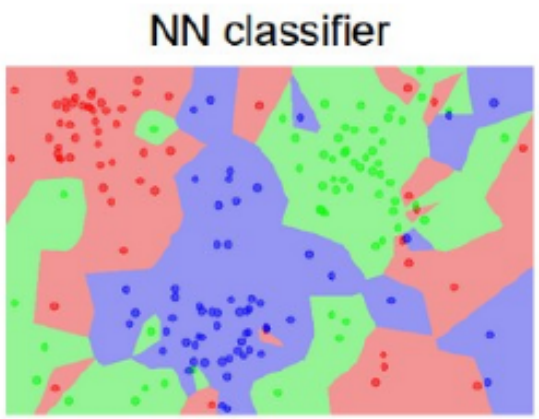
(b)

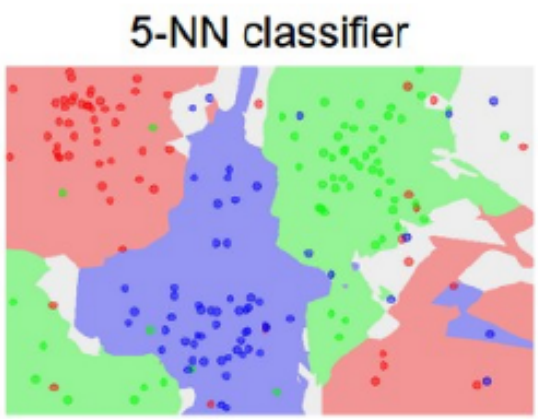

K=1, Label: ??
x_2
x_1
(a)


K=3, Label: ??
x_2
x_1
(a)


K=5, Label: ??
x_2
x_1
(a)


the data


NN classifier


5-NN classifier

Nearest Neighbor Classifier

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in xrange(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

Memorize training data

Nearest Neighbor Classifier

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in xrange(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

For each test image:

Find nearest training image

Return label of nearest image

Advantages of NNC

- Computationally, simple and easy to implement – just compute distances, no optimization required
- Can learn complex decision boundaries

Disadvantages of NNC

- Computationally intensive for large-scale problems:
  |
- We need to "carry" the training data around. Without it, we cannot do classification. This type of method is called *nonparametric*.
- Choosing the right distance measure and K can be difficult.

Decision Trees and Ensembles

Non-linear Classifiers

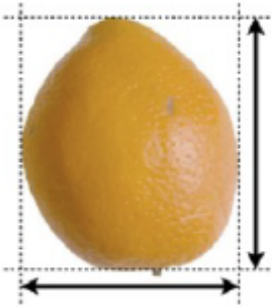
Dealing with non-linear decision boundary

1. add "non-linear" features
   to a linear model (e.g., logistic regression)
2. **use non-linear learners**
   (nearest neighbors, decision trees, artificial neural nets, ...)

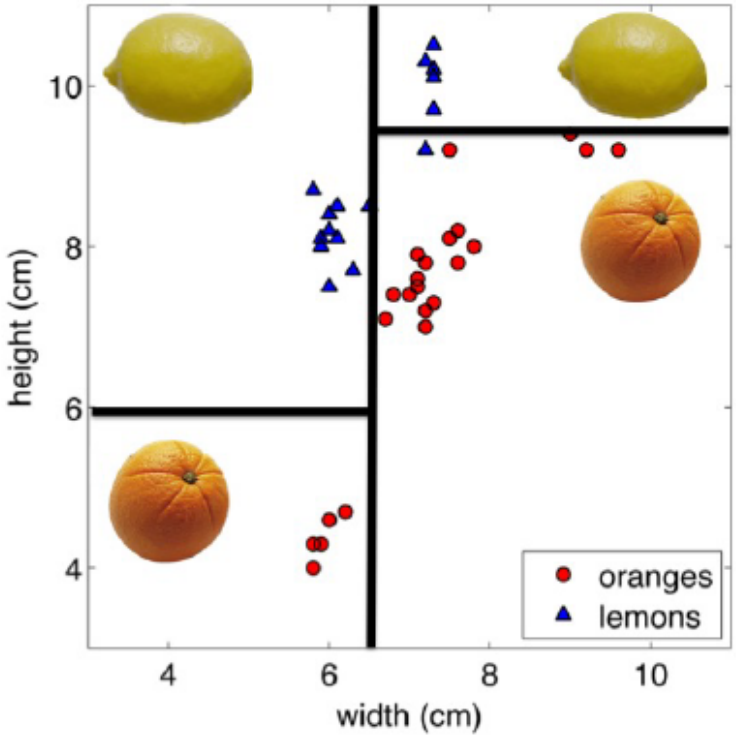
k-Nearest Neighbor Classifier

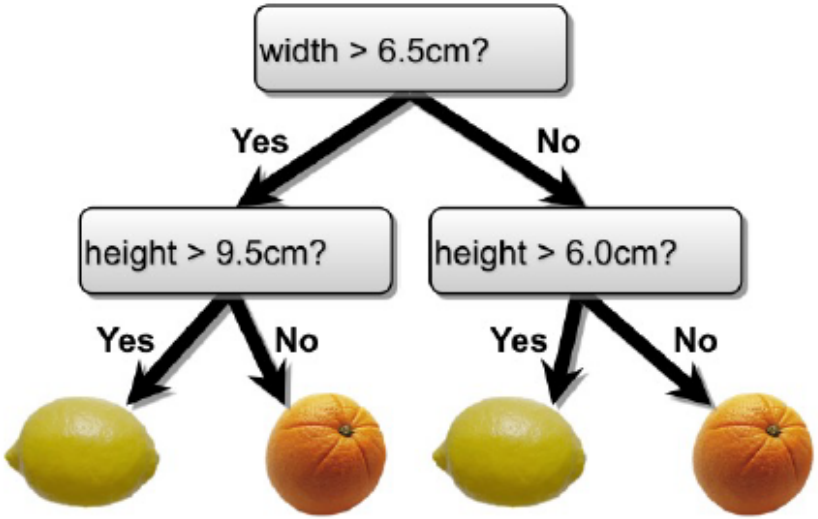
- simple, often a good baseline
- can approximate arbitrary boundary: **non-parametric**
- **downside:** stores all the data

Test example



- Decision trees make predictions by recursively splitting on different attributes according to a tree structure.





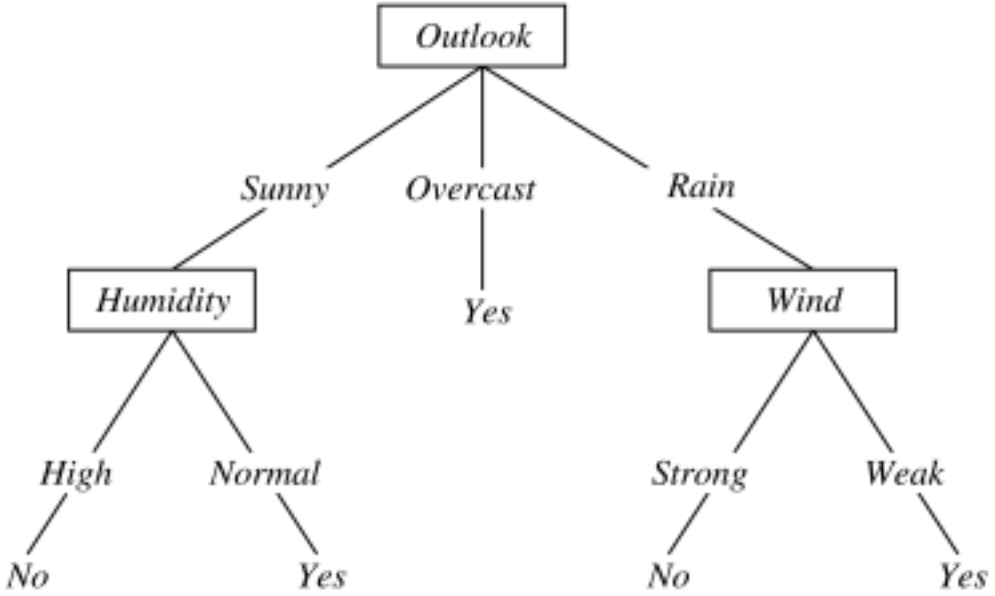
Sample Dataset (was Tennis Played?)

- Columns denote features $\mathrm{X_i}$
- Rows denote labeled instances $\langle \boldsymbol{x}_i, y_i \rangle$
- Class label denotes whether a tennis game was played

$\langle \boldsymbol{x}_i, y_i \rangle$

Predictors				Response
Outlook	**Temperature**	**Humidity**	**Wind**	**Class**
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

A Decision Tree for *PlayTennis*



Each internal node: test one feature X_j

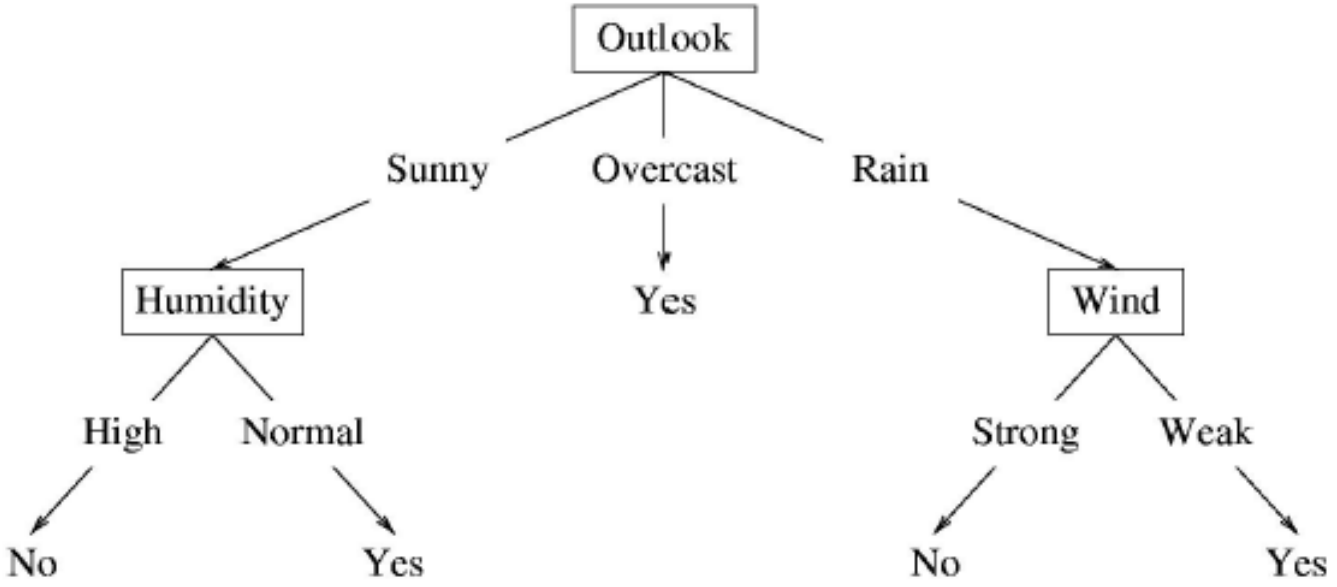
Each branch from a node: select one value for X_j

Each leaf node node: predict Y

or $P(Y \mid X \in \text{leaf})$

Decision Tree

- A possible decision tree for the data:

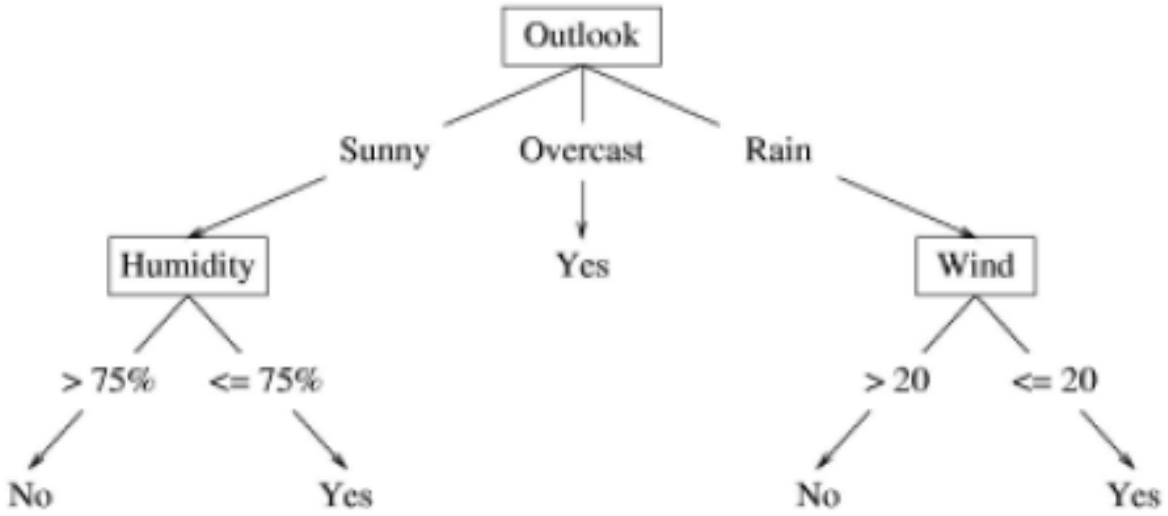


- What prediction would we make for

  <outlook=sunny, temperature=hot, humidity=high, wind=weak> ?

Decision Tree

- If features are continuous, internal nodes can test the value of a feature against a threshold



Example

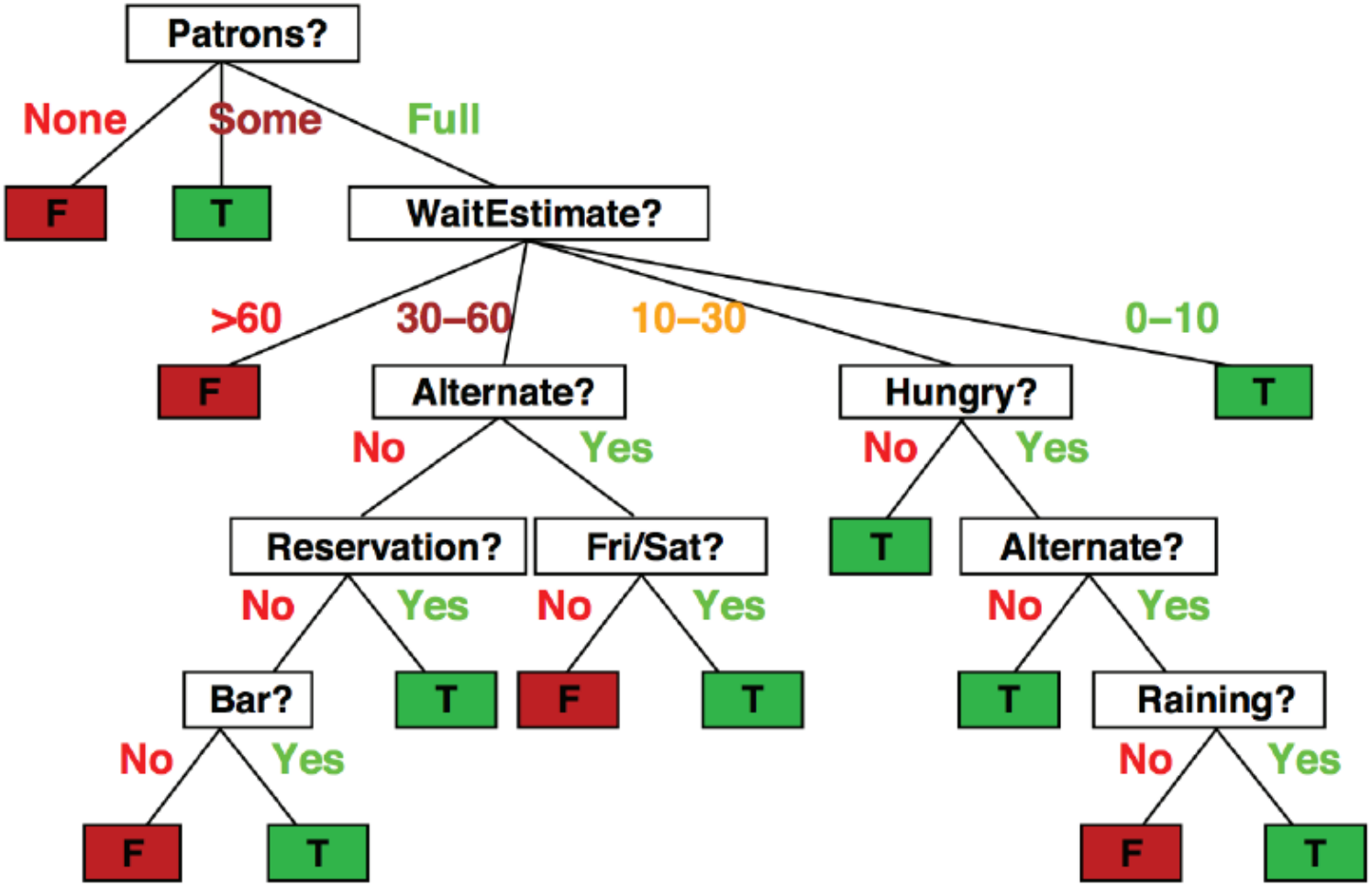
- What if the attributes are discrete?

Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 =$ *Yes*
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 =$ *No*
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 =$ *Yes*
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 =$ *Yes*
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 =$ *No*
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 =$ *Yes*
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 =$ *No*
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 =$ *Yes*
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 =$ *No*
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} =$ *No*
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} =$ *No*
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} =$ *Yes*

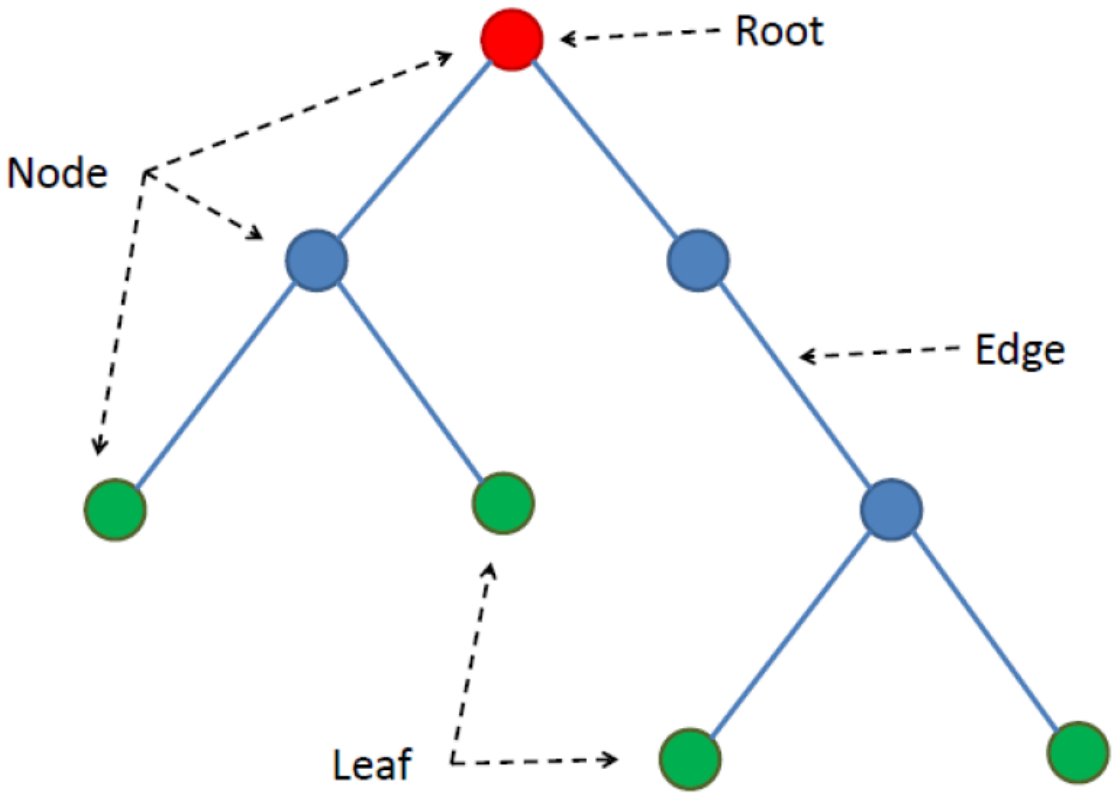
Attributes:

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. Hungry: whether we are hungry.
5. Patrons: how many people are in the restaurant (values are None, Some, and Full).
6. Price: the restaurant's price range ($, $$, $$$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (French, Italian, Thai or Burger).
10. WaitEstimate: the wait estimated by the host (0-10 minutes, 10-30, 30-60, >60).

- The tree to decide whether to wait (T) or not (F)



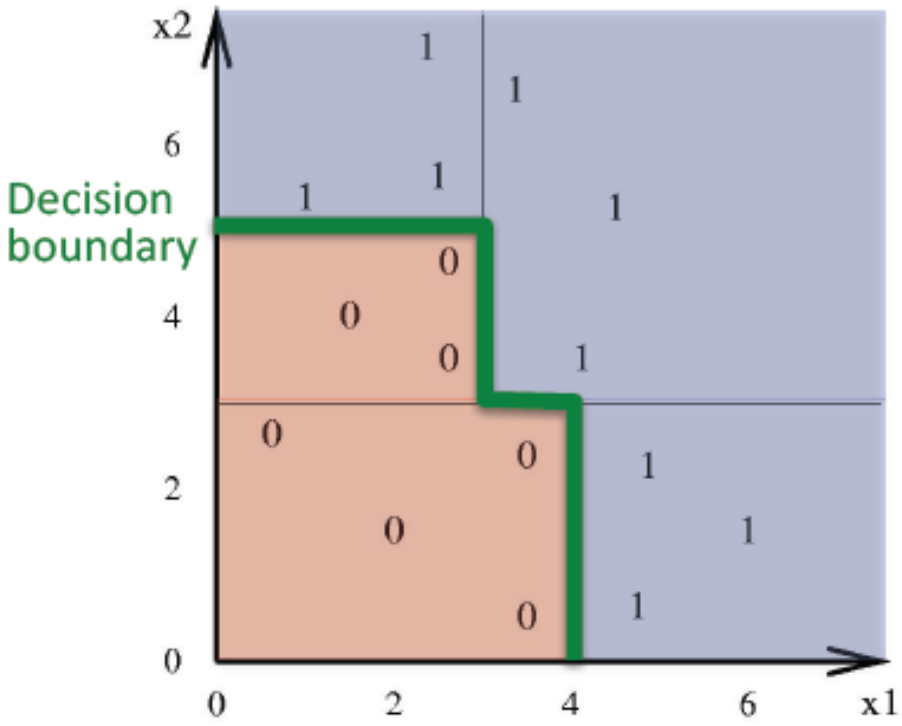
Special names for nodes in a tree Node

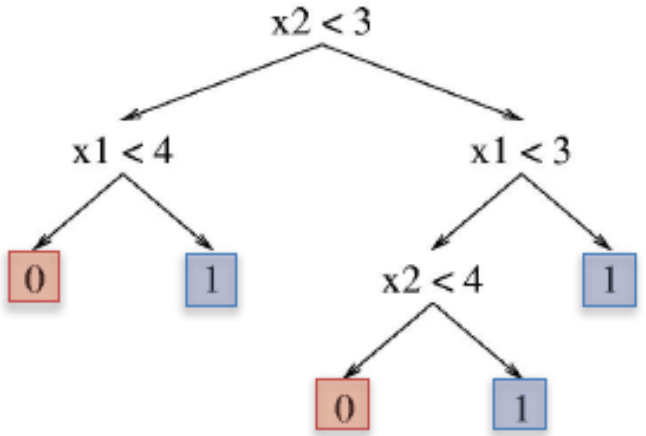


- Internal nodes test attributes
- Branching is determined by attribute value
- Leaf nodes are outputs (predictions)

Decision Tree – Decision Boundary

- Decision trees divide the feature space into axis-parallel (hyper-)rectangles
- Each rectangular region is labeled with one label
  - or a probability distribution over labels





How do we learn decision trees?

We want the *smallest* tree that is consistent with the training data

(i.e. that assigns the correct labels to training items)

But we can't enumerate all possible trees.

$|\mathcal{H}|$ is exponential in the number of features

- In Restaurant example can start from any attribute.
- Any Combination can be used so number of Hypothesis are many/

We use a heuristic: greedy top-down search

This is guaranteed to find a consistent tree,
and is biased towards finding smaller trees

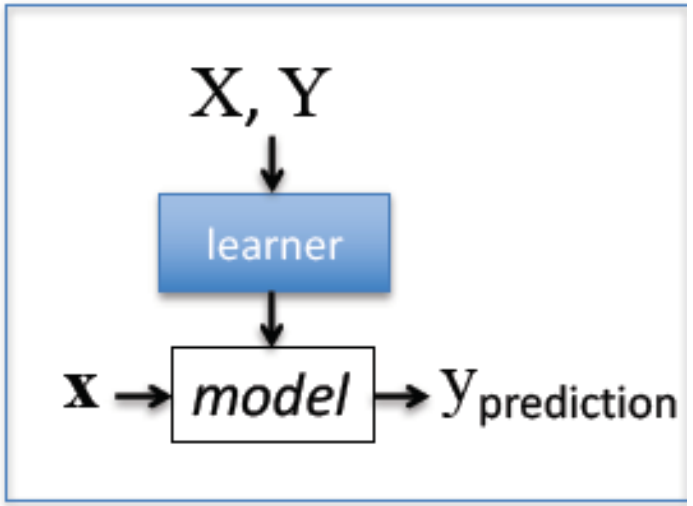
Stages of (Batch) Machine Learning

Given: labeled training data $X, Y = \{\langle \boldsymbol{x}_i, y_i \rangle\}_{i=1}^{n}$

- Assumes each $\boldsymbol{x}_i \sim \mathcal{D}(\mathcal{X})$ with $y_i = f_{target}(\boldsymbol{x}_i)$

Train the model:

model ← *classifier*.train(X, Y)



Apply the model to new data:

- Given: new unlabeled instance $\boldsymbol{x} \sim \mathcal{D}(\mathcal{X})$

$\mathrm{y}_{\text{prediction}}$ ← *model*.predict(**x**)

Basic Algorithm for Top-Down Learning of Decision Trees

[ID3, C4.5 by Quinlan]

node = root of decision tree

Main loop:

1. $A \leftarrow$ the "best" decision attribute for the next node.
2. Assign *A* as decision attribute for *node*.
3. For each value of *A*, create a new descendant of *node*.
4. Sort training examples to leaf nodes.
5. If training examples are perfectly classified, stop. Else, recurse over new leaf nodes.

How do we choose which attribute is best?

Choosing the Best Attribute

Key problem: choosing which attribute to split a given set of examples

- Some possibilities are:
  - **Random:** Select any attribute at random
  - **Least-Values:** Choose the attribute with the smallest number of possible values
  - **Most-Values:** Choose the attribute with the largest number of possible values
  - **Max-Gain:** Choose the attribute that has the largest expected *information gain*

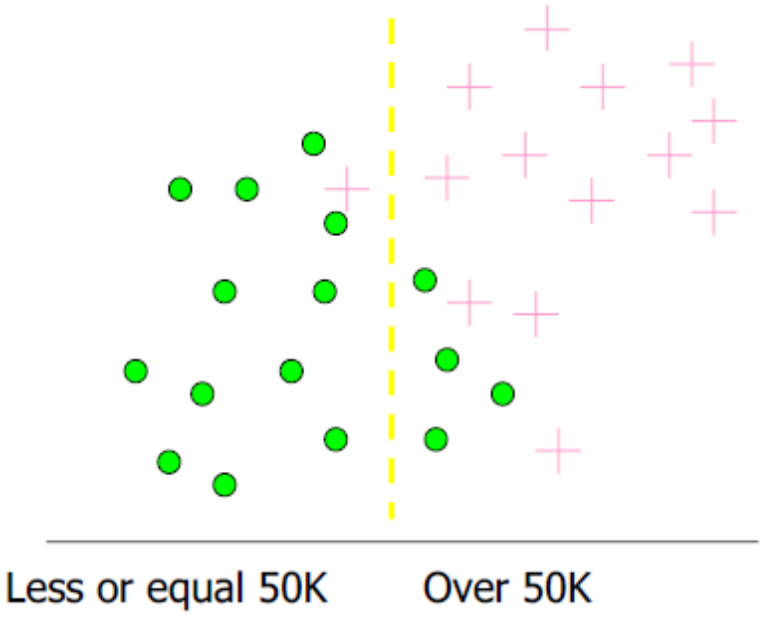
- The ID3 algorithm uses the Max-Gain method of selecting the best attribute

Information Gain

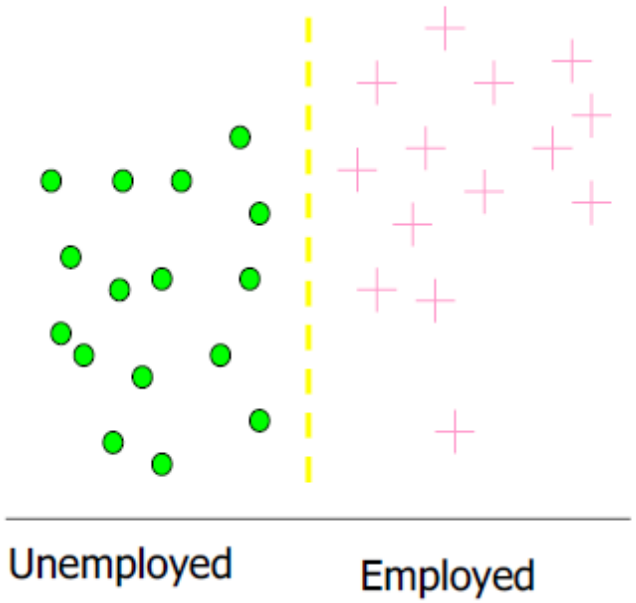
Which test is more informative?

Loan Example

Split over whether Balance exceeds 50K



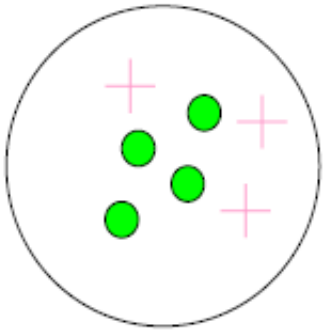
Split over whether applicant is employed



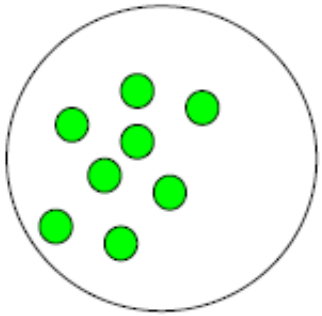
Information Gain

Impurity/Entropy (informal)

– Measures the level of **impurity** in a group of examples

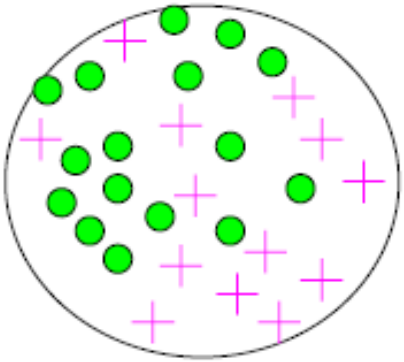


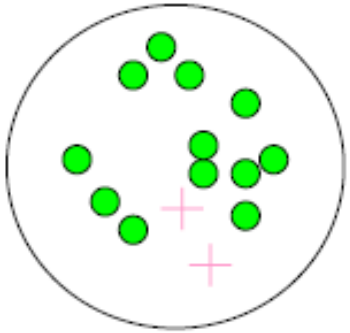
Very impure group

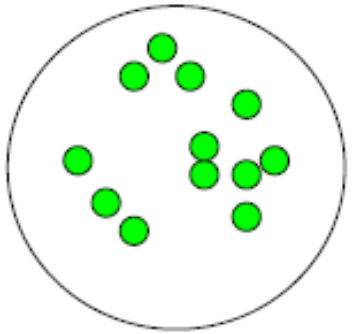


Less impure

Minimum impurity



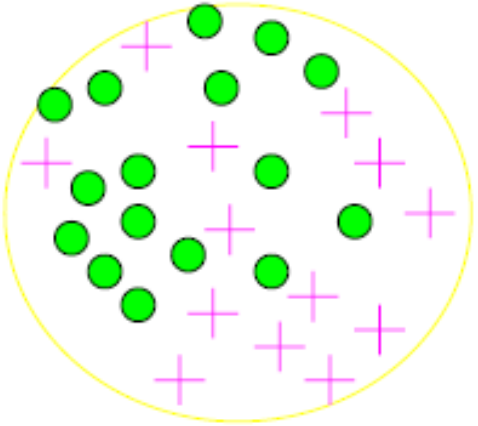




Entropy: a common way to measure impurity

- Entropy = $\sum_i - p_i \log_2 p_i$

  p_i is the probability of class i



- Entropy comes from information theory. The higher the entropy the more the information content.

What does that mean for learning from examples?

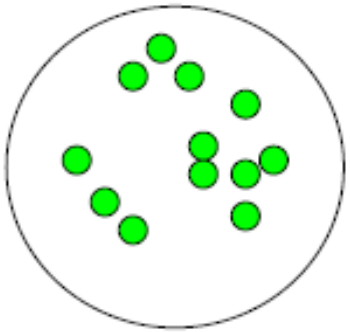
2-Class Cases:

Entropy $$H(x) = -\sum_{i=1}^{n} P(x=i)\log_2 P(x=i)$$

- What is the entropy of a group in which all examples belong to the same class?
  - entropy = - 1 $\log_2 1 = 0$

not a good training set for learning

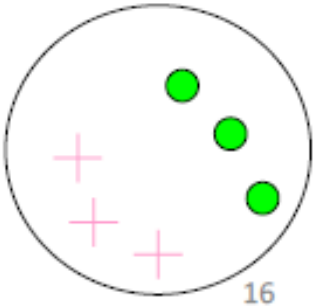
Minimum impurity



- What is the entropy of a group with 50% in either class?
  - entropy = -0.5 $\log_2 0.5 - 0.5 \log_2 0.5 = 1$

good training set for learning

Maximum impurity



Information Gain

- We want to determine which attribute in a given set of training feature vectors is most useful for discriminating between the classes to be learned.
- Information gain tells us how important a given attribute of the feature vectors is.
- We will use it to decide the ordering of attributes in the nodes of a decision tree.

From Entropy to Information Gain

Entropy $H(X)$ of a random variable X

$$H(X) = -\sum_{i=1}^{n} P(X=i)\log_2 P(X=i)$$

Specific conditional entropy $H(X|Y=v)$ of X given $Y=v$:

$$H(X|Y=v) = -\sum_{i=1}^{n} P(X=i|Y=v)\log_2 P(X=i|Y=v)$$

Conditional entropy $H(X|Y)$ of X given Y :

$$H(X|Y) = \sum_{v \in values(Y)} P(Y=v)H(X|Y=v)$$

Mututal information (aka Information Gain) of X and Y :

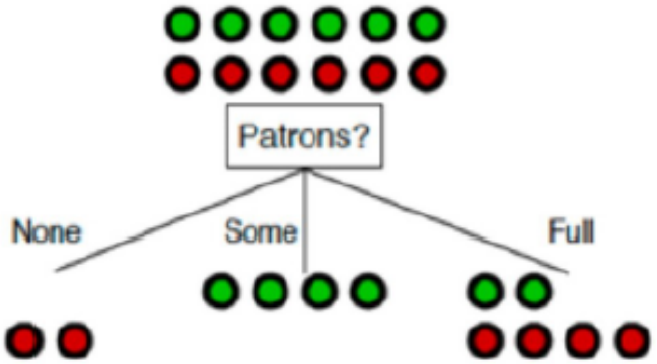
$$I(X,Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

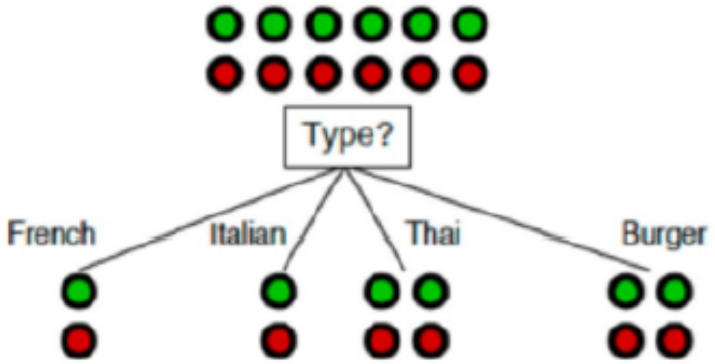
Restaurant Example to find Information Gain

Attributes										Target
Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
T	F	F	T	Some	$$$	F	T	French	0–10	T
T	F	F	T	Full	$	F	F	Thai	30–60	F
F	T	F	F	Some	$	F	F	Burger	0–10	T
T	F	T	T	Full	$	F	F	Thai	10–30	T
T	F	T	F	Full	$$$	F	T	French	>60	F
F	T	F	T	Some	$$	T	T	Italian	0–10	T
F	T	F	F	None	$	T	F	Burger	0–10	F
F	F	F	T	Some	$$	T	T	Thai	0–10	T
F	T	T	F	Full	$	T	F	Burger	>60	F
T	T	T	T	Full	$$$	F	T	Italian	10–30	F
F	F	F	F	None	$	F	F	Thai	0–10	F
T	T	T	T	Full	$	F	F	Burger	30–60	T

Use the attributes to decide whether to wait (T) or not wait (F)

Restaurant Example to find Information Gain





Patron vs. Type?

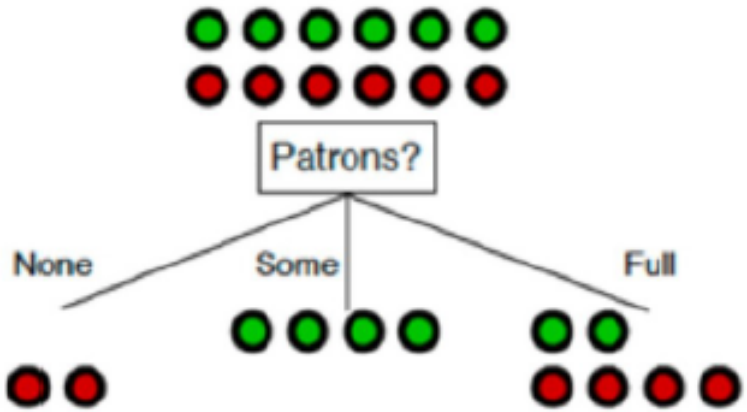
- Let us compute the information gain $I(X;Y) = H[Y] - H[Y|X]$ for Patron and Type

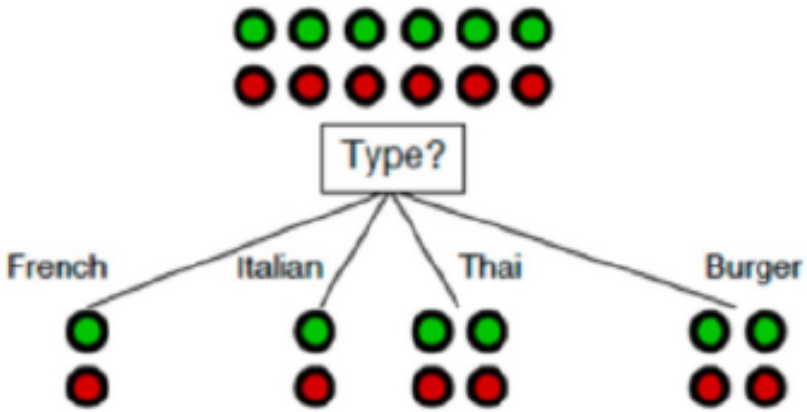
- $H(Y) = -\frac{6}{12}\log\frac{6}{12} - \frac{6}{12}\log\frac{6}{12} = 1$ bit
- $H(Y|X = none) = 0$
- $H(Y|X = some) = 0$
- $H(Y|X = full) = -\left(\frac{2}{2+4}\log\frac{2}{2+4} + \frac{4}{2+4}\log\frac{4}{2+4}\right) \approx 0.9$ bits
- Thus the conditional entropy is

$$H(Y|X) = (\frac{2}{12} \times 0 + \frac{4}{12} \times 0 + \frac{6}{12} \times 0.9) = 0.45 \text{ bits}$$

- Information Gain $I(X;Y) = 1 - 0.45 = 0.55$ bits

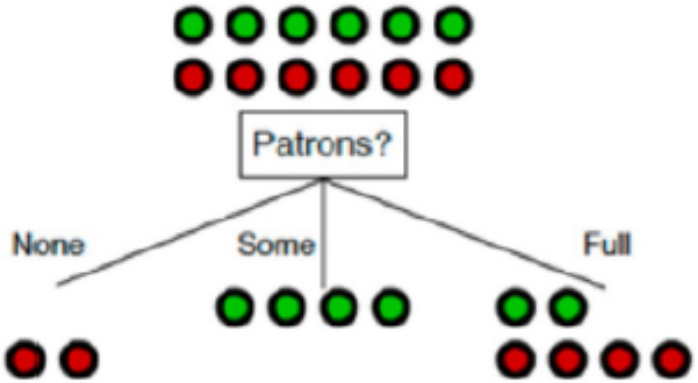
Restaurant Example to find Information Gain

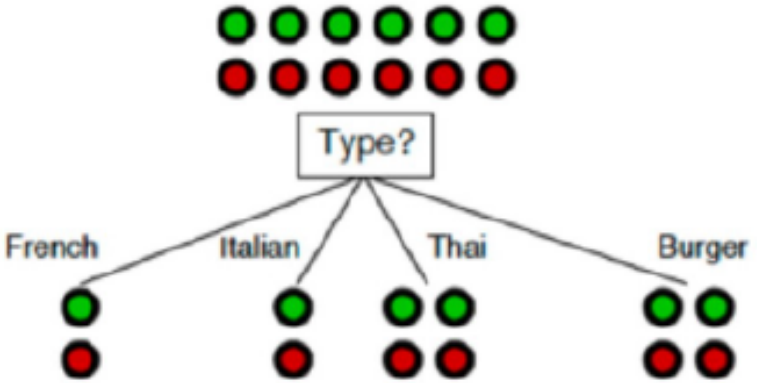




- $H(Y) = -\frac{6}{12}\log\frac{6}{12} - \frac{6}{12}\log\frac{6}{12} = 1$ bit
- $H(Y|X = \textit{french}) = \log 2 = 1$ bit
- $H(Y|X = \textit{italian}) = \log 2 = 1$ bit
- $H(Y|X = \textit{thai}) = \log 2 = 1$ bit
- $H(Y|X = \textit{burger}) = \log 2 = 1$ bit
- Thus the conditional entropy is
  $H(Y|X) = \frac{2}{12} \times 1 + \frac{2}{12} \times 1 + \frac{4}{12} \times 1 + \frac{4}{12} \times 1 = 1$ bit
- Information Gain $I(X; Y) = 1 - 1 = 0$ bits

Restaurant Example to find Information Gain





- Information gain from "Patron" is 0.55 bits.
- Information gain from "Type" is 0 bits.

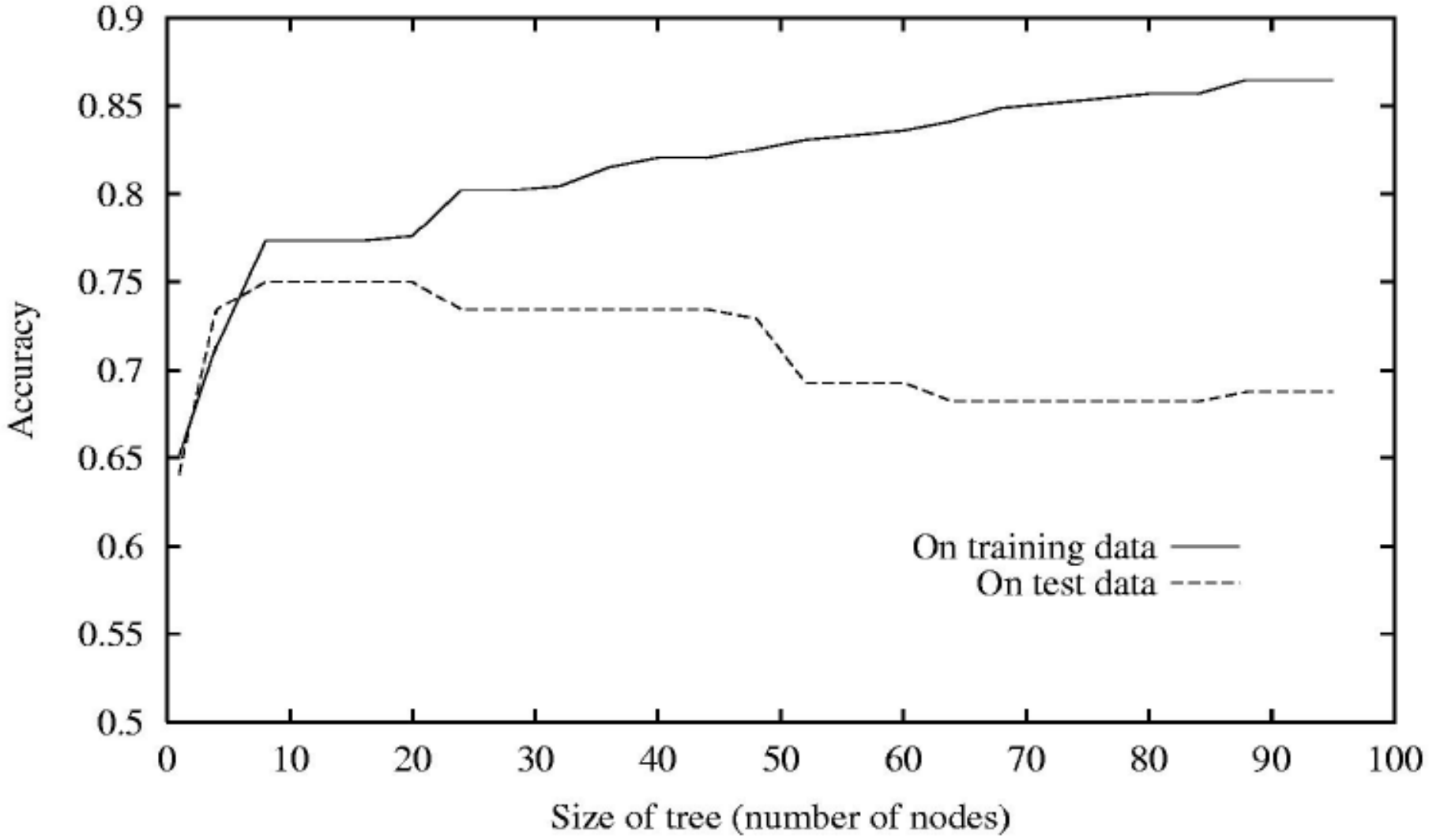
Thus, we should split on "Patron" and not "Type" (higher information gain). This is consistent with our intuition.

Overfitting in Decision Trees

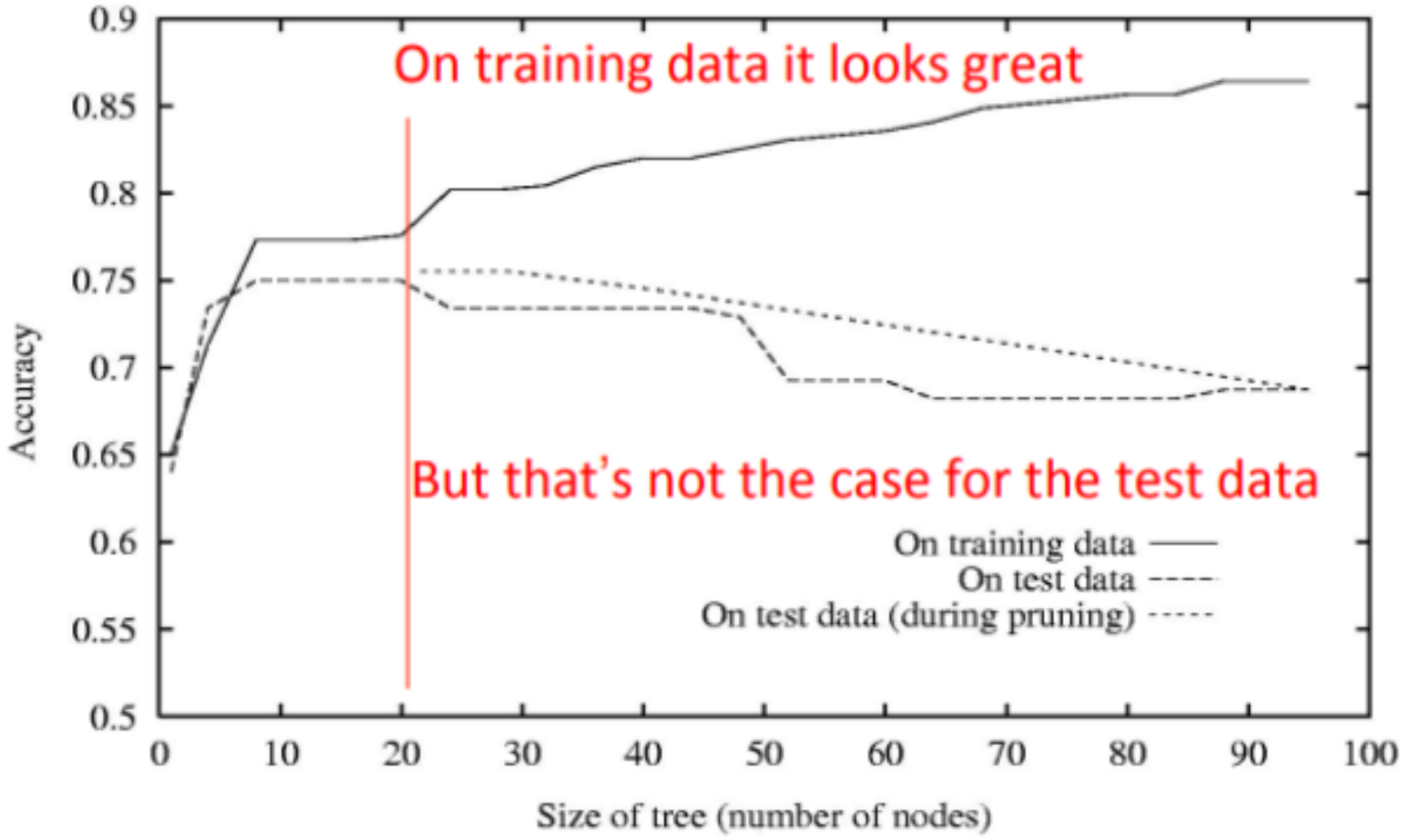
- Irrelevant attributes can result in *overfitting* the training example data
  - If hypothesis space has many dimensions (large number of attributes), we may find **meaningless regularity** in the data that is irrelevant to the true, important, distinguishing features

- If we have too little training data, even a reasonable hypothesis space will 'overfit'

Overfitting in Decision Trees



Effect of Reduced-Error Pruning



The tree is pruned back to the red line where it gives more accurate results on the test data

Limitations of Decision Tree Models

Decision trees models are highly interpretable and fast to train, using our greedy learning algorithm.

However, in order to capture a complex decision boundary (or to approximate a complex function), we need to use a large tree (since each time we can only make axis aligned splits).

We've seen that large trees have high variance and are prone to overfitting.

For these reasons, in practice, decision tree models often underperforms when compared with other classification or regression methods.

Bagging

One way to adjust for the high variance of the output of an experiment is to perform the experiment multiple times and then average the results.

The same idea can be applied to high variance models:

1. **(Bootstrap)** we generate multiple samples of training data, via bootstrapping. We train a full decision tree on each sample of data.

2. **(Aggregate)** for a given input, we output the averaged outputs of all the models for that input.

   For classification, we return the class that is outputted by the plurality of the models.

This method is called ***Bagging*** (Breiman, 1996), short for, of course, Bootstrap Aggregating.

Bagging

Original Data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

Bootstrap Sample 1

X	Y
X_4	y_4
X_{14}	y_{14}
X_{11}	y_{11}
X_2	y_2
X_{35}	y_{35}
⋮	⋮
X_k	y_k

Decision Tree 1

Used and unused data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

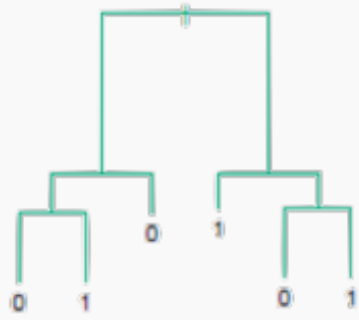
Original Data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

Bootstrap Sample 2

X	Y
X_5	y_5
X_3	y_3
X_{12}	y_{12}
X_{43}	y_{43}
X_1	y_1
⋮	⋮
X_k	y_k

Decision Tree 2



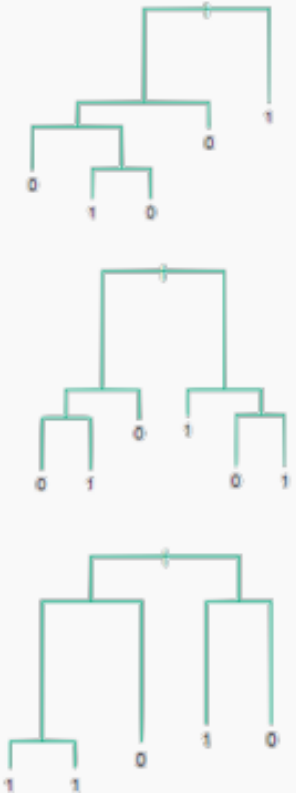
Used and unused data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

Prediction

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
⋮	⋮
X_i	y_i
⋮	⋮
X_n	y_n

B Trees that did not see $\{X_i, y_i\}$



Classification

$$\hat{y}_{i,pw} = \text{majority}(\hat{y}_i)$$

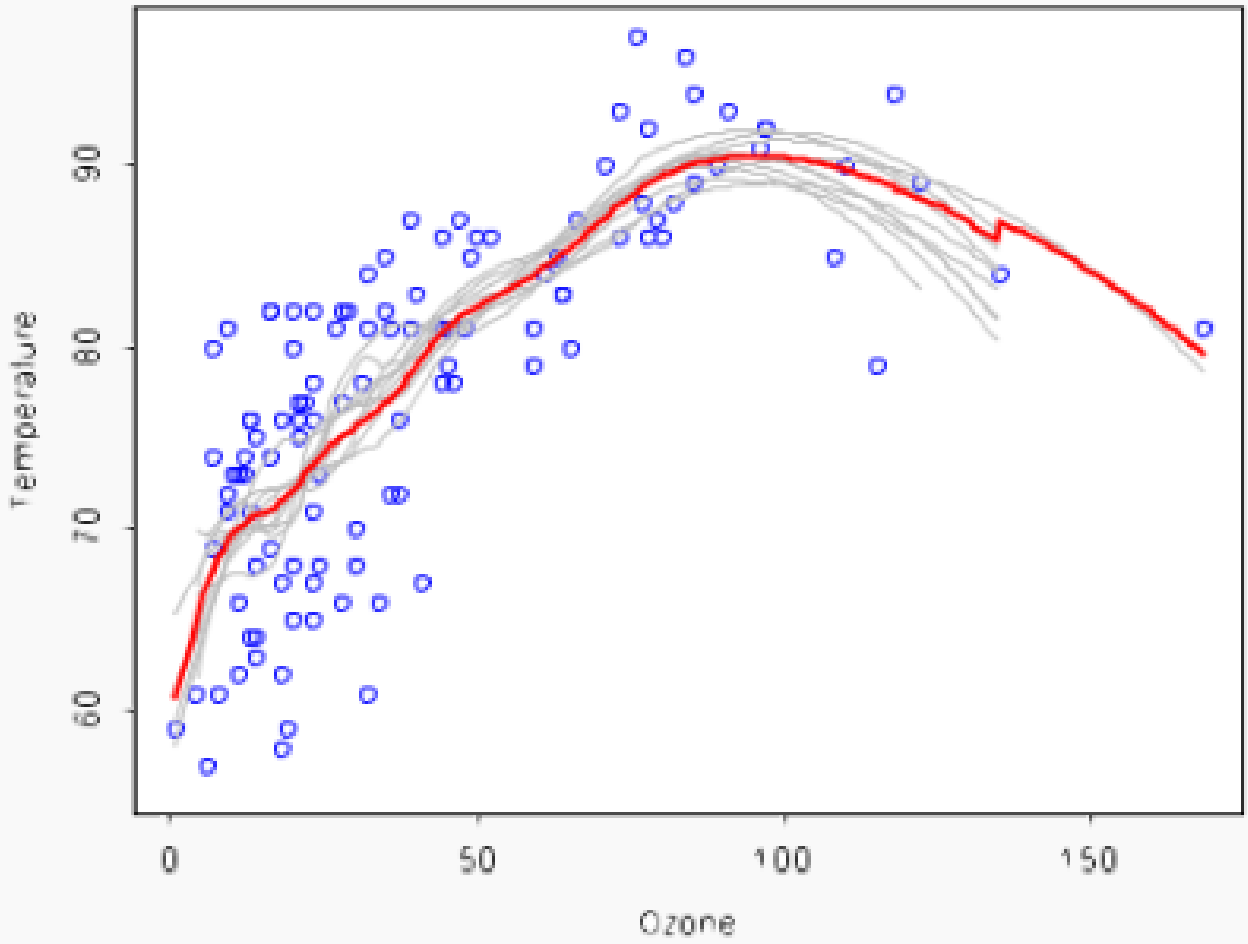
$$e_i = \mathbb{I}(\hat{y}_{i,pw} \neq y_i)$$

Regression

$$\hat{y}_{i,pw} = \sum_{j \in B} \hat{y}_{i,j}$$

$$e_i = (y_i - \hat{y}_{i.pw})^2$$

Bagging



Bagging

Question: Do you see any problems?

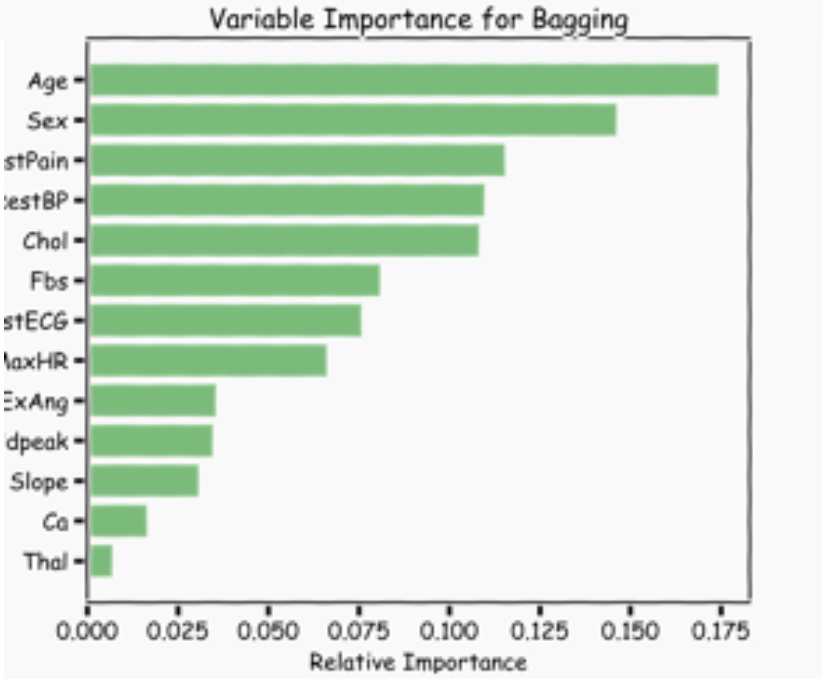
- Still some overfitting if the trees are too large.
- If trees are too shallow it can still underfits.
- Interpretability:

  The **major drawback** of bagging (and other ***ensemble methods*** that we will study) is that the averaged model is no longer easily interpretable - i.e. one can no longer trace the 'logic' of an output through a series of decisions based on predictor values!

Variable Importance for Bagging

Bagging improves prediction accuracy at the expense of interpretability.

Calculate the total amount that the MSE (for regression) or Gini index (for classification) is decreased due to splits over a given Feature, averaged over all B trees.



Improving on Bagging

In practice, the ensembles of trees in Bagging tend to be highly correlated.

Suppose we have an extremely strong Feature, x_j, in the training set amongst moderate Features. Then the greedy learning algorithm ensures that most of the models in the ensemble will choose to split on x_j in early iterations.

Each tree in the ensemble is identically distributed, with the expected output of the averaged model the same as the expected output of any of the trees.

Random Forests

Random Forest is a modified form of bagging that creates ensembles of independent decision trees.

To de-correlate the trees, we:

1. train each tree on a separate bootstrap sample of the full training set (same as in bagging)

2. for each tree, at each split, we ***randomly*** select a set of J' Features from the full set of Features.

From amongst the J' Features, we select the optimal Feature and the optimal corresponding threshold for the split.

Scenario

- Task: Predict whether a person will buy a car (Yes/No) based on features like income, age, and location.

Income ($)	Age	Location	Buy Car (Yes/No)
50,000	25	Urban	No
60,000	35	Suburban	Yes
45,000	23	Rural	No
80,000	45	Suburban	Yes
70,000	40	Urban	Yes

Bagging

1. Create multiple bootstrapped datasets by sampling with replacement.
   - Example:
     - Dataset 1: Row 1, Row 3, Row 5, Row 1, Row 4
     - Dataset 2: Row 2, Row 4, Row 3, Row 5, Row 2
2. Train a separate decision tree on each dataset.
3. Use majority vote for classification or average prediction for regression.

- Example Outcome:
  - Tree 1 predicts: No, No, Yes, Yes, Yes → Final Vote: Yes
  - Tree 2 predicts: Yes, Yes, Yes, No, No → Final Vote: Yes
  - Aggregated Result: Majority vote → Yes (Buys Car)

Random Forest

1. Create bootstrapped datasets (same as Bagging).
2. For each decision tree:
   - At every split, select a random subset of features.
   - Example:
     - Tree 1: Split on Income and Age
     - Tree 2: Split on Income and Location
     - Tree 3: Split on Age and Location
3. Use majority vote for classification or average prediction for regression.

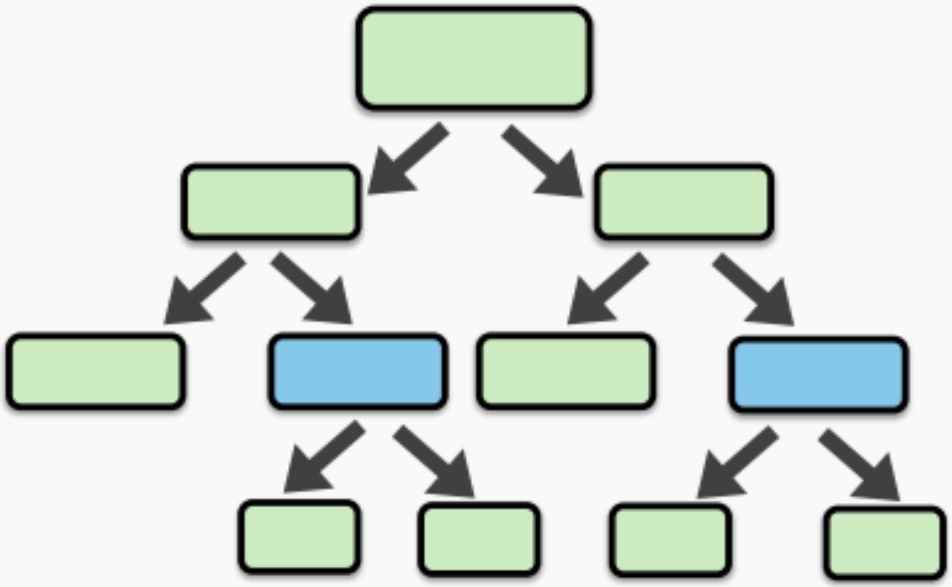
Example Outcome:

- Tree 1 uses Income and Age → Predicts Yes
- Tree 2 uses Income and Location → Predicts No
- Tree 3 uses Age and Location → Predicts Yes
- Aggregated Result: Majority vote → Yes (Buys Car)

Random Forest reduces both **variance** and **overfitting** by decorrelating trees through feature randomness.

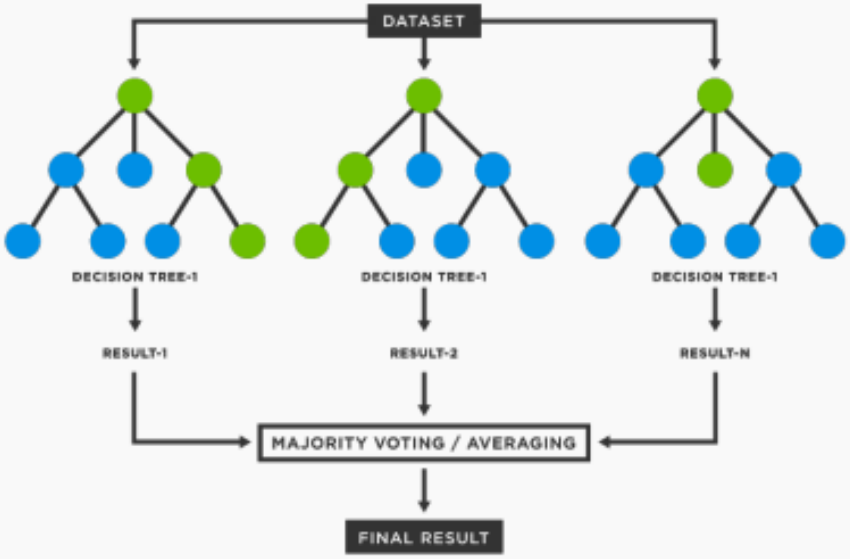
DECSION TREE ISSUES?

- Shallow trees:
  - Shallow trees (with very few leaves) suffer from high bias and do not train well.
- Deep Trees:
  - Deep trees (with large number of nodes and leaves) have low bias, but suffer from high variance leading to



RF ISSUES?

- Variance:
  - Although variance reduction is better than bagging, the generalization error is still high
- Speed:
  - Large number of trees can make the algorithm very slow and ineffective for real-time predictions



Motivation for Boosting

Question: Could we address the shortcomings of single decision trees models in some other way?

For example, rather than performing variance reduction on complex trees, can we decrease the bias of simple trees - make them more expressive?

Can we learn from our mistakes?

A solution to this problem, making an expressive model from simple trees, is another class of ensemble methods called ***boosting***.

Boosting

NEW IDEA 💡

- **Boosting** methods are general algorithms which combine several "**weak learners**" to produce a strong rule.
- The first implementation of Boosting was '**Adaboost**' invented by Robert Schapire and Yoav Freund in 1996.
- Boosting algorithms are fast, easy to compute and very accurate and are the de-facto optimization tree algorithms.

Get Good understanding of Boosting Concept to Get Grade in CS-351

OPTION #1

1. Steal the time-stone from Dr. Strange.
2. Go back to 1996 and meet Rob Schapire and Yoav Freund.
3. Follow their work for at least a decade to understand everything about boosting.
4. Return to the present and nail the test.
5. Repeat for another test

OPTION #2

STEP #1:

Go to the library and get previous year question papers.

STEP #2:

Find a helpful student and ask her to give you a "rule of thumb" to get at least some answers right.

DOES THE "RULE" WORK ?

Test out the rule.

It worked 60% of the time. Not bad!!

STEP #3:

Find a TA and ask him to also give you a "rule of thumb" to get at least some answers right.

make sure to focus on the ones you got wrong before.

DOES THE "RULE" WORK ?

Test out the new rule.

It works well on difficult problems! But a few problems persist.

STEP #4:

Call your favorite professor and focus on the ones you got wrong before!

OPTION #2

DOES THE "RULE" WORK ?

Test out the new rule.

The new rule works well on the difficult problems!

STEP #5:

Combine the rules, but pay more attention to the ones that were more often right

Accuracy: 60%

Accuracy: 64%

Accuracy: 70%

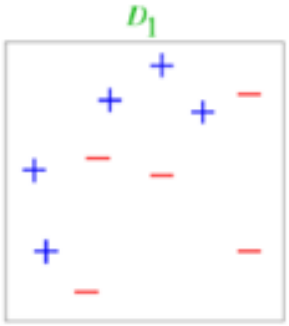
$$Strategy = \alpha * Rule_1 + \beta * Rule_2 + \gamma * Rule_3$$

FINAL STEP:

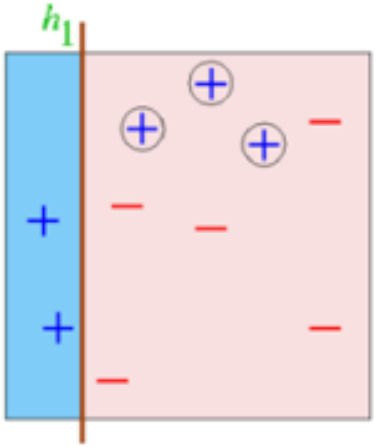
Take the test with these approximate rules, weighted by how well each rule performed.

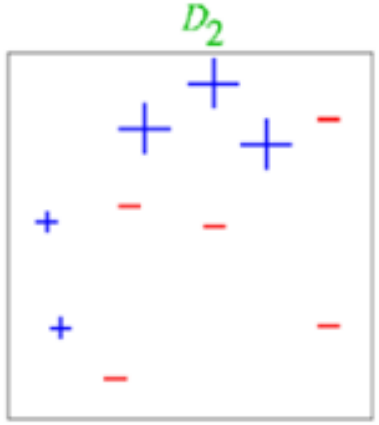
A+

Boosting



First classifier

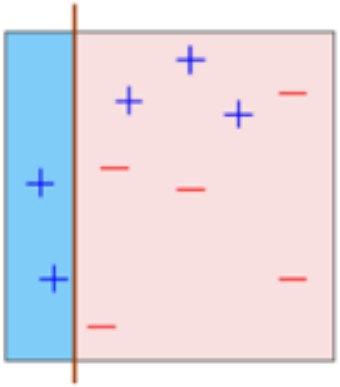


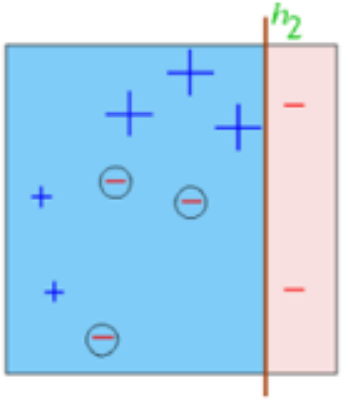


$\varepsilon_1 = 0.30$

$\alpha_1 = 0.42$

First 2 classifiers



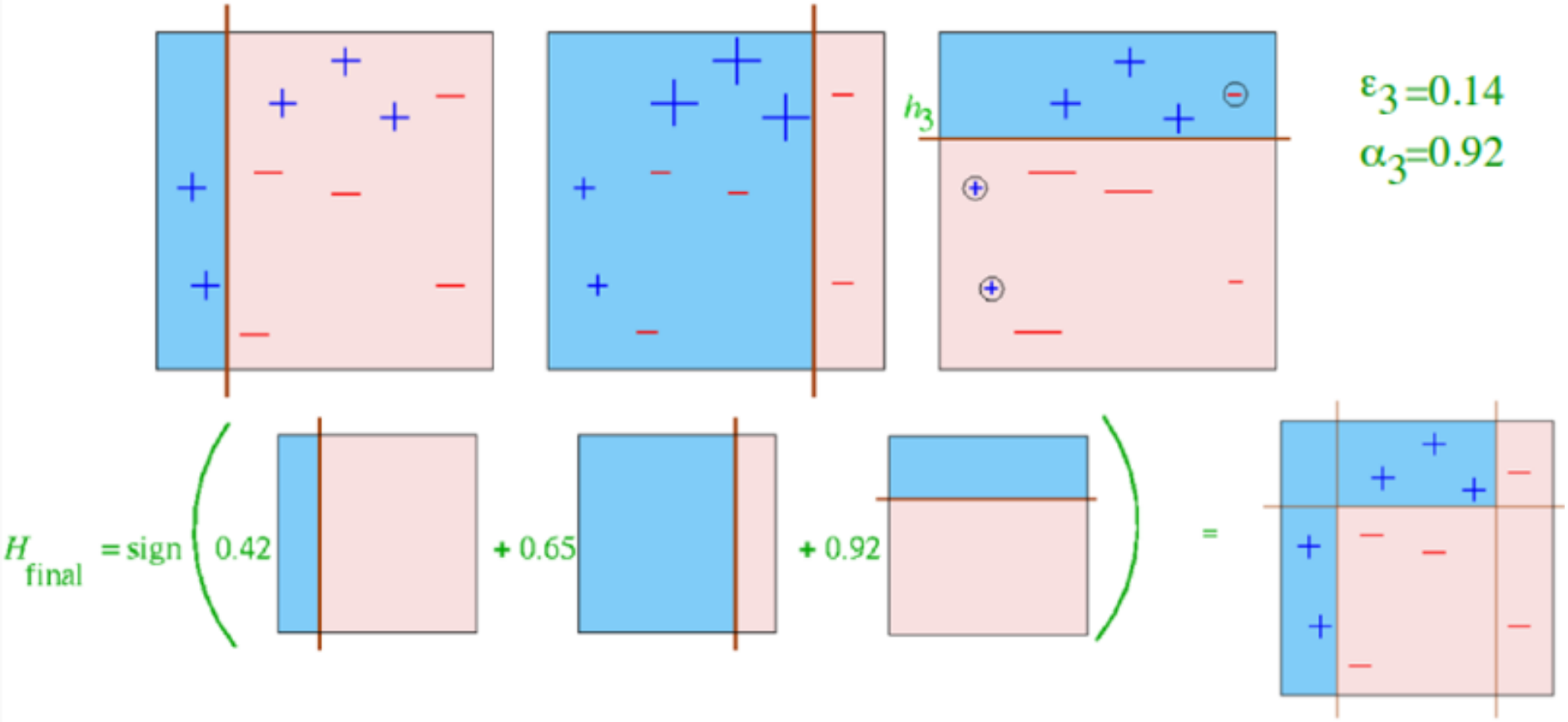


$\varepsilon_2 = 0.21$

$\alpha_2 = 0.65$

Adaboost combines multiple weak classifiers into a strong classifier by weighting their outputs. The final decision is a weighted sum of these weak classifiers.

Boosting



Suppose we have training data $\{(x_i, y_i)\}_{i=1}^{N}$, $y_i \in \{1, -1\}$

- Initialize equal weights for all observations $w_i = 1/N$

- At each iteration t:
  1. **Train a stump** G_t using data weighted by w_i
  2. **Compute the misclassification error** adjusted by w_i
  3. **Compute the weight of the current tree** α_t
  4. **Reweight each observation** based on prediction accuracy

- Calculate the weighted misclassification error

$$err_t = \frac{\sum_{i=1}^{N} w_i \, I(y_i \neq G_t(x_i))}{\sum_{i=1}^{N} w_i}$$

- Use the error score to weight the current tree in the final classifier:

$$\alpha_t = log\left(\frac{1 - err_t}{err_t}\right)$$

A classifier with 50% accuracy is given a weight of zero;

- Use misclassification error and tree weight to reweight the training data:

$$w_i \leftarrow w_i exp[\alpha_t I(y_i \neq G_t(x_i))]$$

- **Final prediction is a weighted sum of the predictions from each stump:**

$$G(x) = sign\left[\sum_{t=1}^{T} \alpha_t \, G_t(x)\right]$$

More accurate trees are weighted higher in the final model.

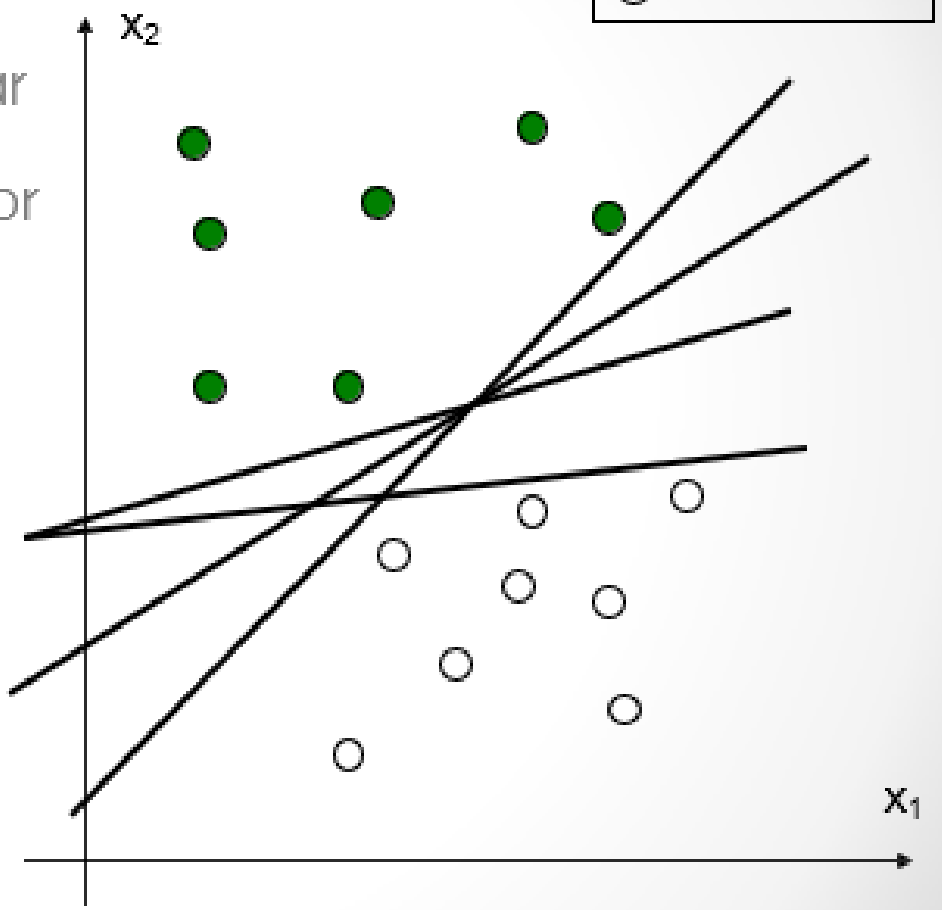
Support Vector Machines

Linear Discriminant Function

- How would you classify these points using a linear discriminant function in order to minimize the error rate?

- Infinite number of answers!

- Which one is the best?

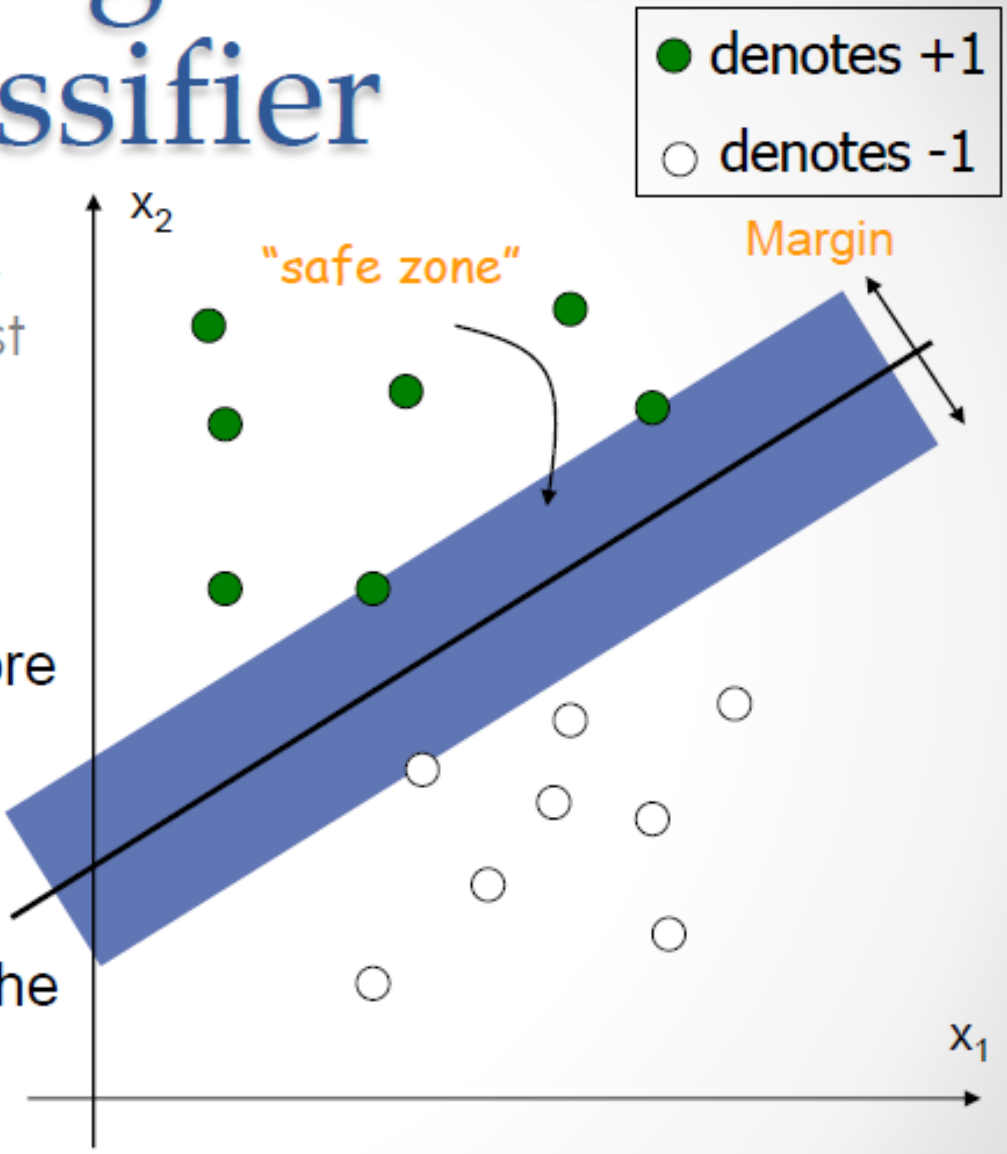


Large Margin Linear Classifier

- The linear discriminant function (classifier) with the maximum margin is the best

- Margin is defined as the width that the boundary could be increased by before hitting a data point

- Why it is the best?
  - The larger the margin the better generalization
  - Robust to outliers



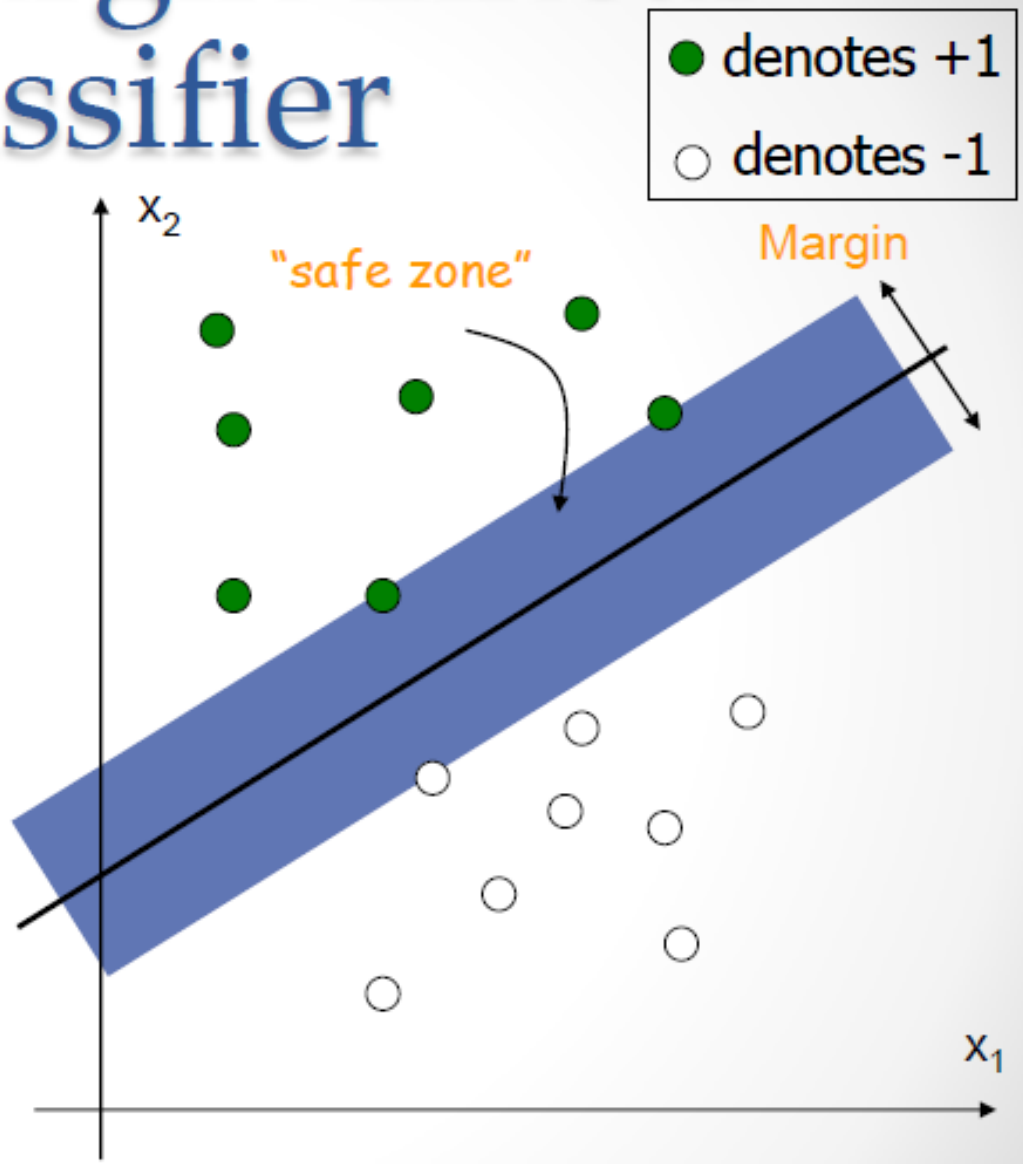
Large Margin Linear Classifier

- Aim: Learn a large margin classifier.
- Given a set of data points, define:

$$\text{For } y_i = +1, \quad \mathbf{w}^T \mathbf{x}_i + b \geq 1$$

$$\text{For } y_i = -1, \quad \mathbf{w}^T \mathbf{x}_i + b \leq -1$$

- Give an algebraic expression for the width of the margin.



Algebraic Expression for Width of a Margin

Given 2 parallel lines with equations

$$ax + by + c_1 = 0$$

and

$$ax + by + c_2 = 0$$

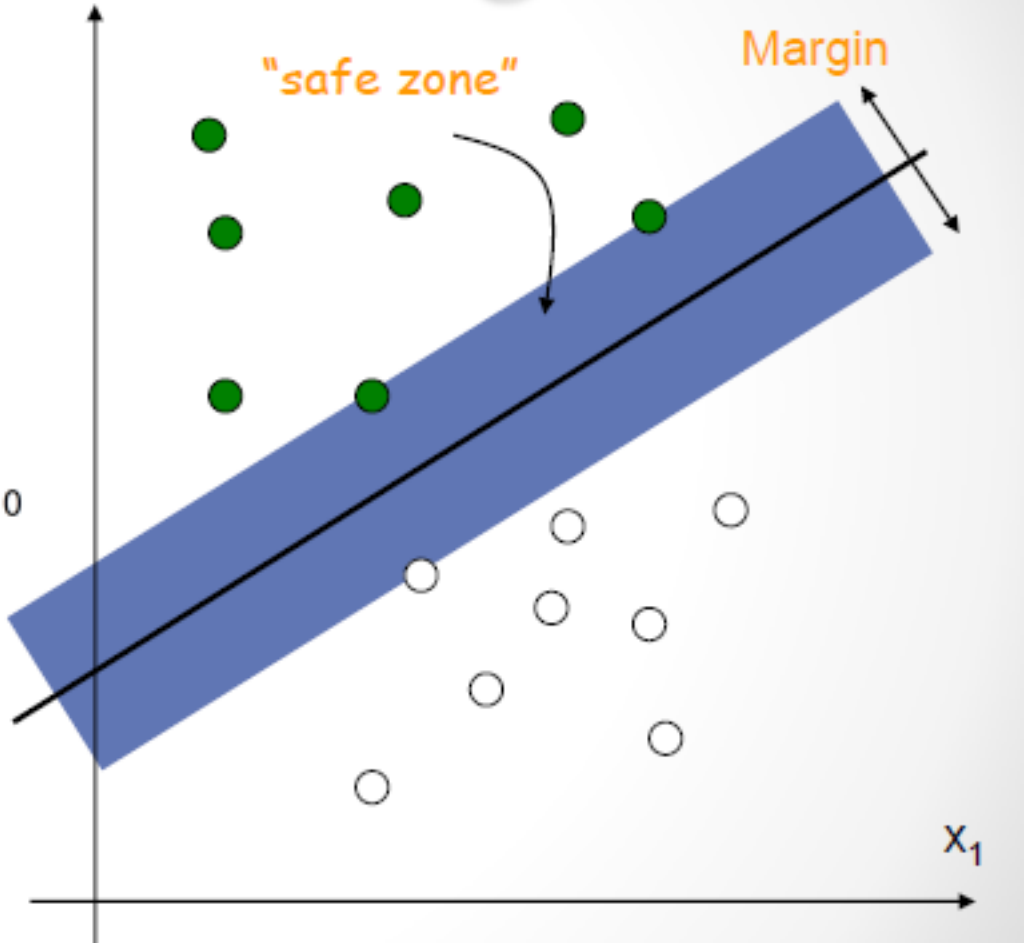
the distance between them is given by:

$$d = \frac{|c_2 - c_1|}{\sqrt{a^2 + b^2}}$$

Our lines in 2-D are:

$w_1 x_1 + w_2 x_2 + b - 1 = 0$ and $w_1 x_1 + w_2 x_2 + b + 1 = 0$

$$Distance = \frac{|b - 1 - b - 1|}{\sqrt{w_1^2 + w_2^2}} = \frac{2}{||\mathbf{w}||}$$



Large Margin Linear Classifier

- Aim: Learn a large margin classifier
- Mathematical Formulation:

$$\text{maximize} \quad \frac{2}{\|\mathbf{w}\|}$$

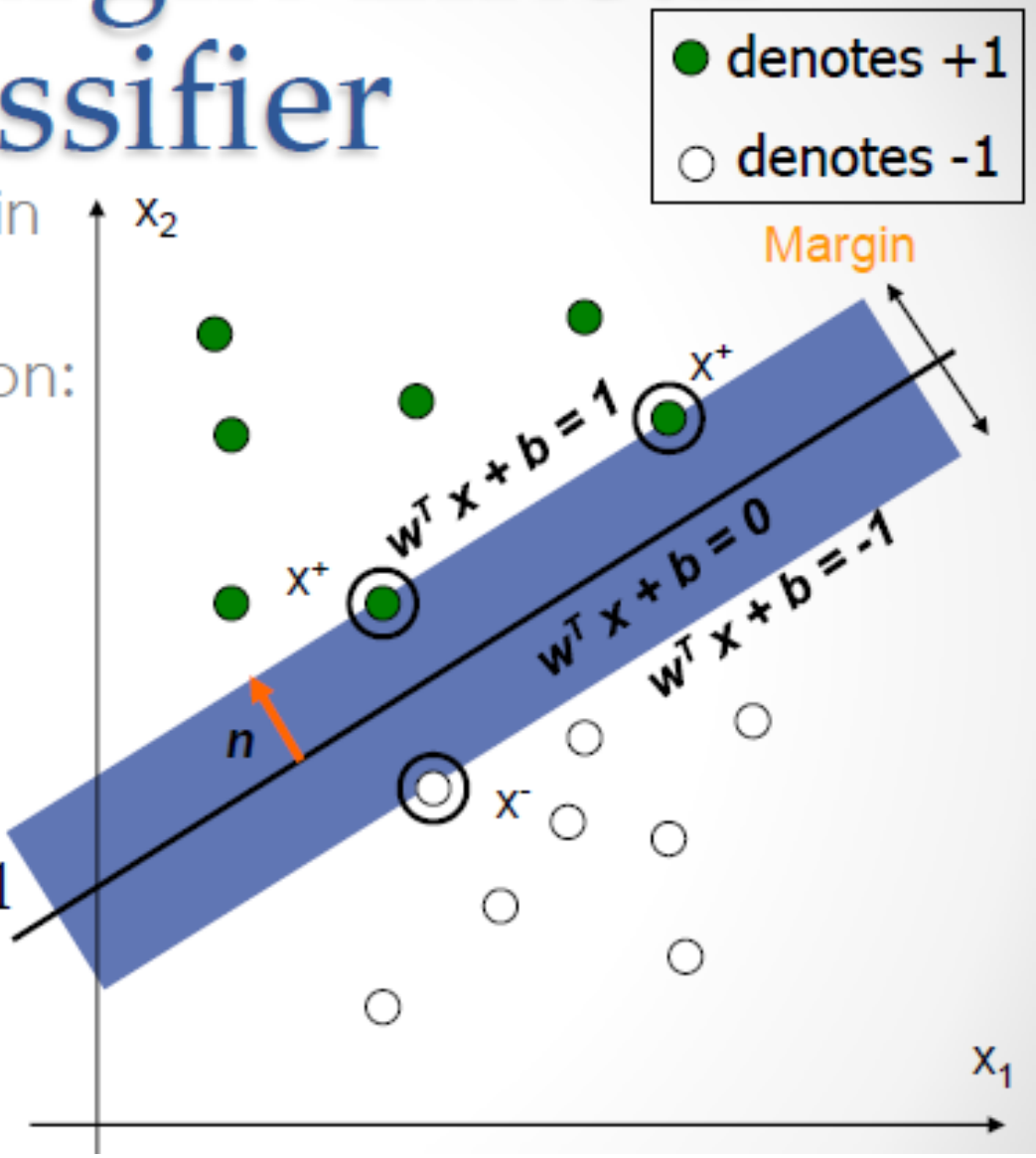
such that

$$\text{For } y_i = +1, \quad \mathbf{w}^T\mathbf{x}_i + b \geq 1$$

$$\text{For } y_i = -1, \quad \mathbf{w}^T\mathbf{x}_i + b \leq -1$$

such that

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$$



Large Margin Linear Classifier

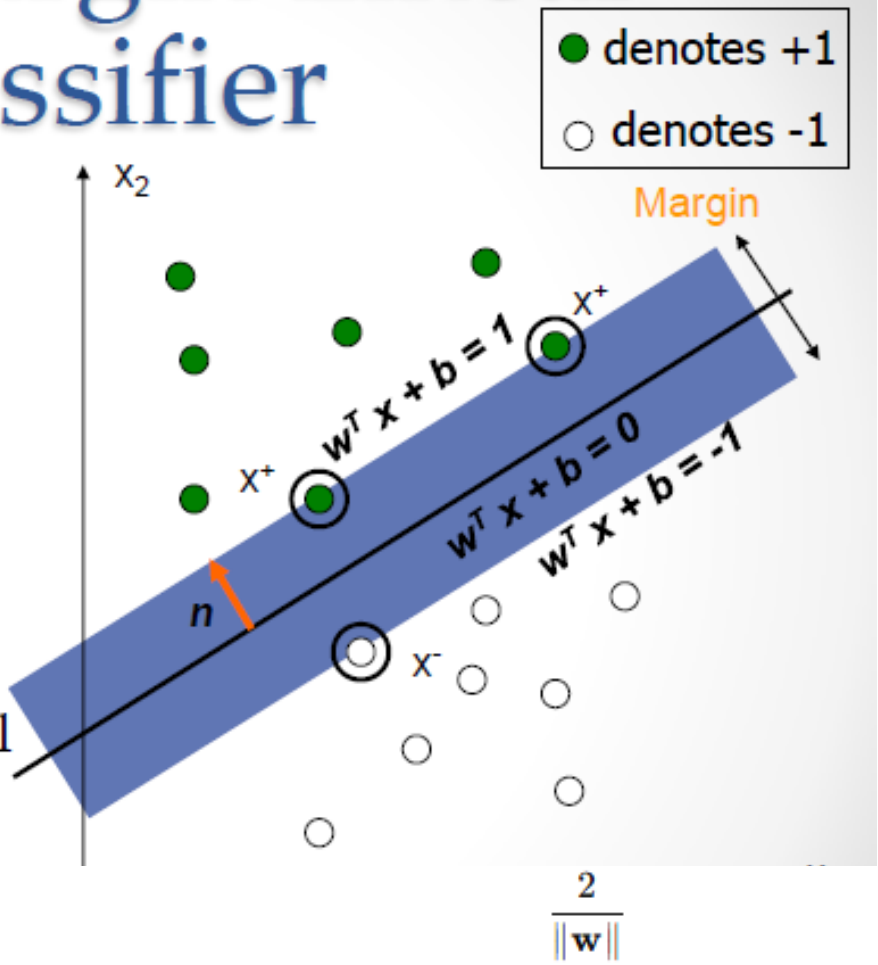
- Formulation:

$$\text{minimize} \quad \frac{1}{2}\|\mathbf{w}\|^2$$

such that

$$\text{For } y_i = +1, \quad \mathbf{w}^T\mathbf{x}_i + b \geq 1$$

$$\text{For } y_i = -1, \quad \mathbf{w}^T\mathbf{x}_i + b \leq -1$$



This represents a fraction where 2 is a constant and $\|\mathbf{w}\|$ is the norm of the weight vector.

- As $\|\mathbf{w}\|$ decreases, the denominator becomes smaller, making the fraction larger.
- Conversely, as $\|\mathbf{w}\|$ increases, the fraction decreases.

Thus, **maximizing** $\frac{2}{\|\mathbf{w}\|}$ **corresponds to minimizing** $\|\mathbf{w}\|$.

Solving the Optimization Problem

$$\text{minimize } L_p(\mathbf{w}, b, \alpha_i) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i \left(y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 \right)$$

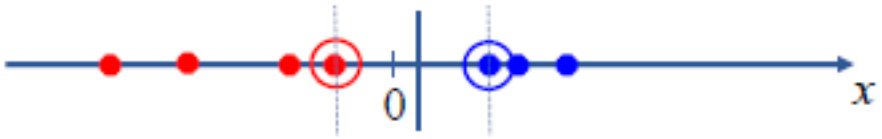
$$\text{s.t.} \quad \alpha_i \geq 0$$

$$\frac{\partial L_p}{\partial \mathbf{w}} = 0 \quad \Rightarrow \quad \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

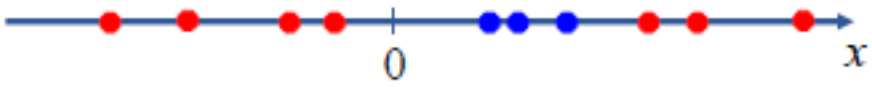
$$\frac{\partial L_p}{\partial b} = 0 \quad \Rightarrow \quad \sum_{i=1}^{n} \alpha_i y_i = 0$$

Non-linear SVMs

- Datasets that are linearly separable with noise work out great:

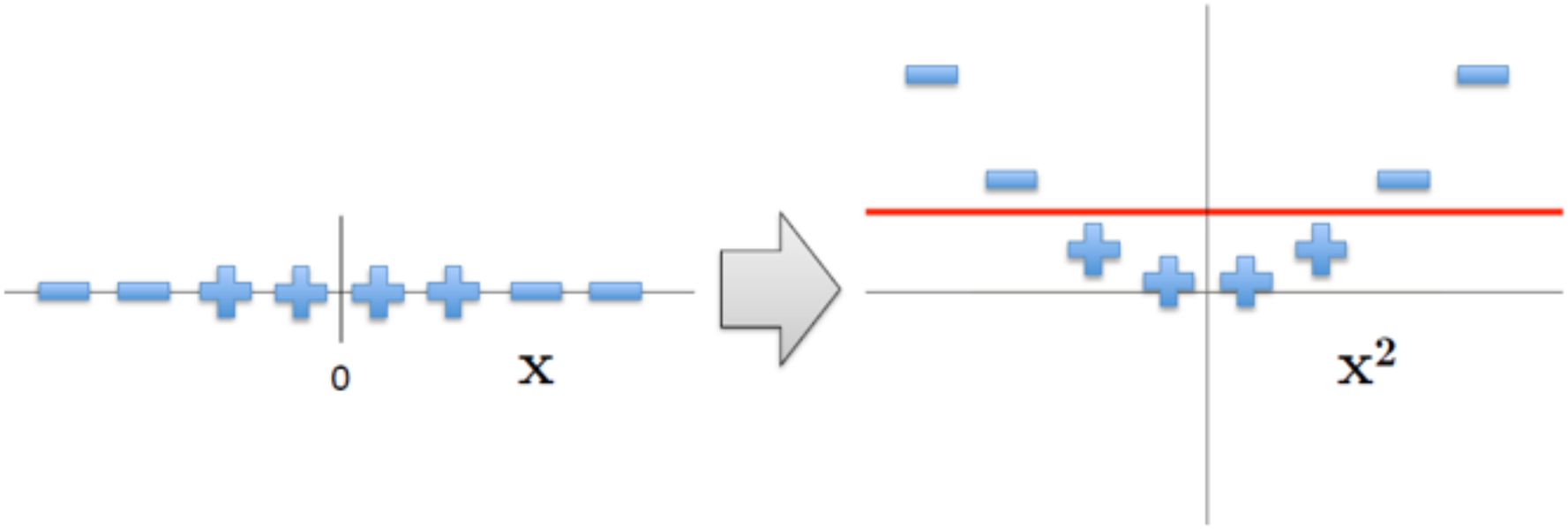


- But what are we going to do if the dataset is just too hard?



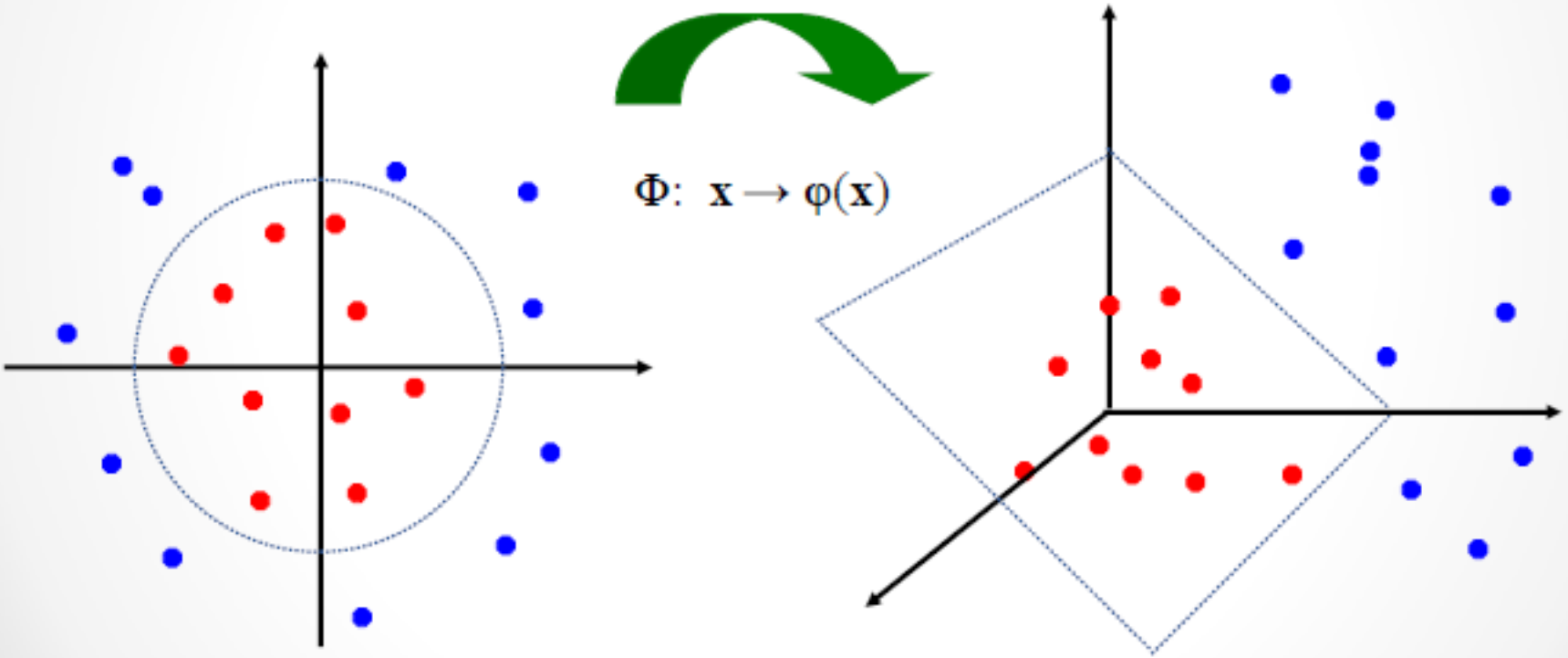
- Kernel Trick!!!
  - SVM = Linear SVM + Kernel Trick

When Linear Separators Fail



Non-linear SVMs: Feature Space

- General idea: the original input space can be mapped to some higher-dimensional feature space where the training set is separable:



Kernel Trick Motivation

- **Linear classifiers** are well understood, widely-used and efficient.
- How to use linear classifiers to build non-linear ones?

- **Kernels:**
  - Map the problem from the input space to a new higher-dimensional space (called the feature space) by doing a non-linear transformation using a special function called the kernel.
  - Then use a linear model in this new high-dimensional feature space. The linear model in the feature space corresponds to a non-linear model in the input space.

A Few Good Kernels...

- Linear Kernel $K(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$
- Polynomial kernel $K(\mathbf{x}_i, \mathbf{x}_j) = (\langle \mathbf{x}_i, \mathbf{x}_j \rangle + c)^d$
  - $c \geq 0$ trades off influence of lower order terms
- Gaussian kernel $K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{2\sigma^2}\right)$
- Sigmoid kernel $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh\left(\alpha \mathbf{x}_i^\intercal \mathbf{x}_j + c\right)$

Unsupervised Learning:

Unsupervised Learning

- Supervised learning used labeled data pairs $(\mathbf{x}, \mathrm{y})$ to learn a function $f : \mathrm{X} \rightarrow \mathrm{Y}$
  - But, what if we don't have labels?

- No labels = **unsupervised learning**

Clustering

Clustering: group together similar points and represent them with a single token

Key Challenges:

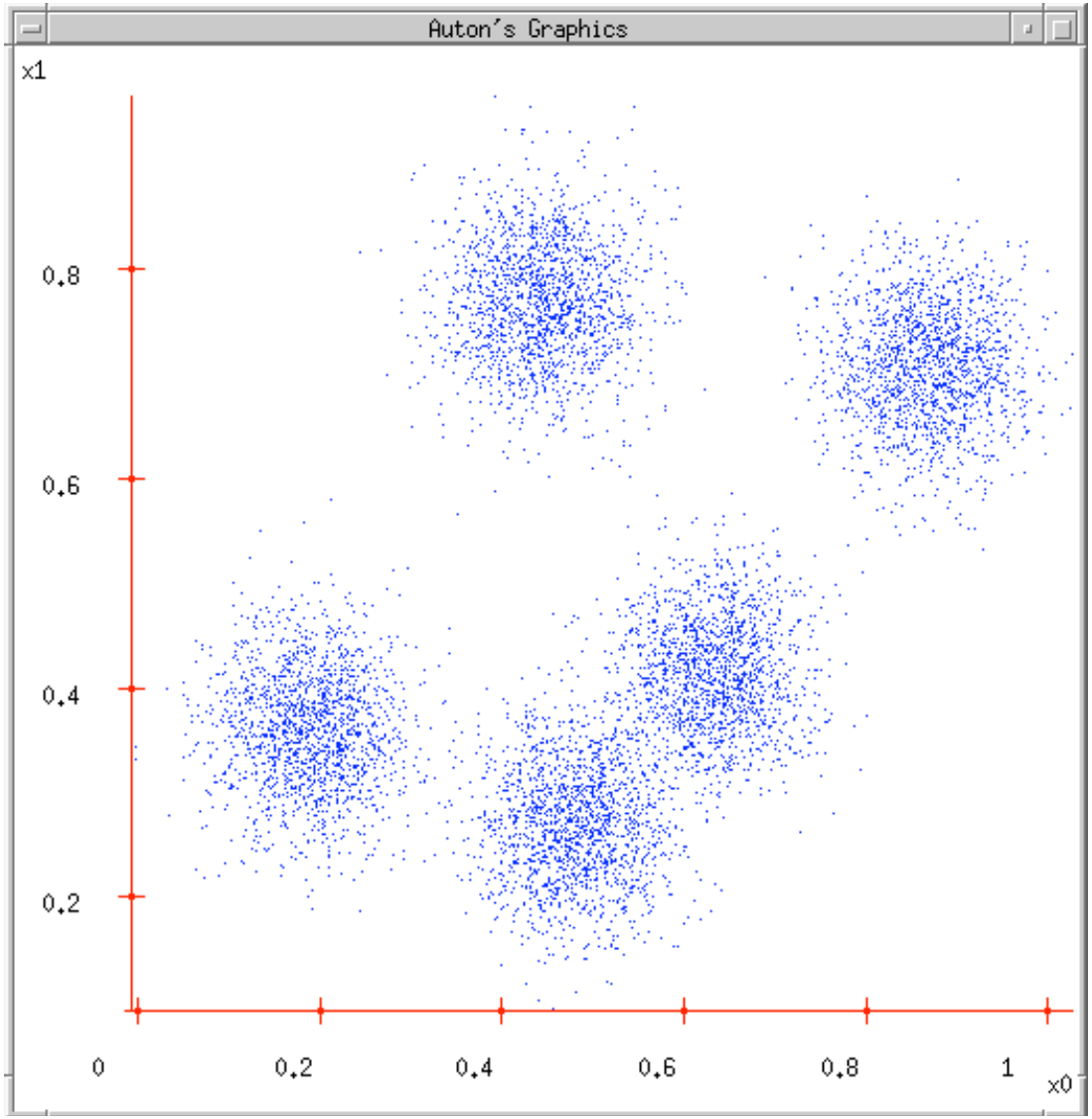
1) What makes two data points similar?

2) How do we compute an overall grouping from pairwise similarities?

How might we cluster?

- K-means
  - Iteratively re-assign points to the nearest cluster center
- Agglomerative clustering
  - Start with each point as its own cluster and iteratively merge the closest clusters

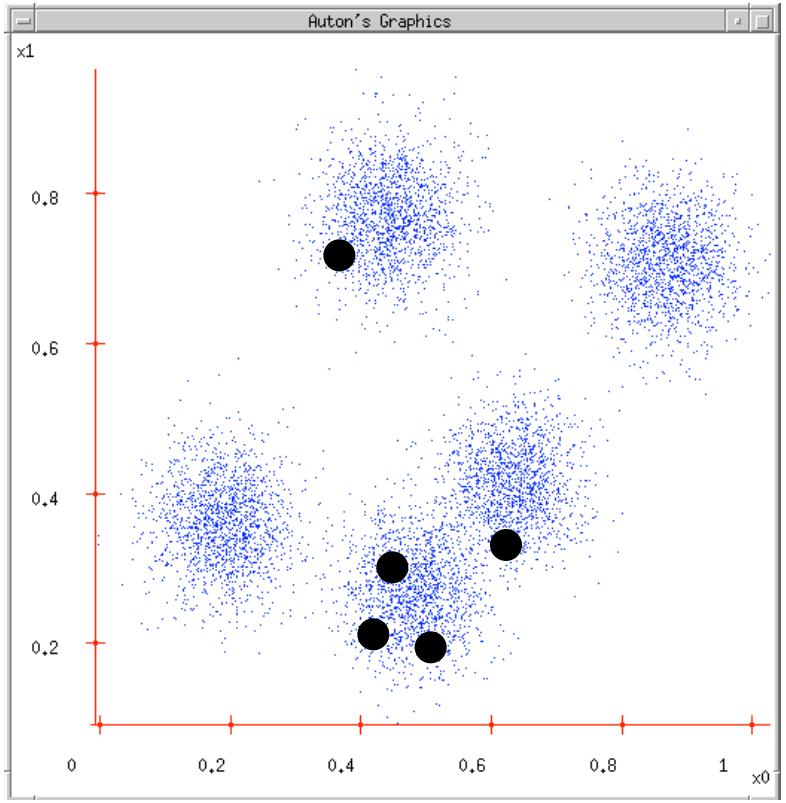
Clustering Data



K-Means Clustering

K-Means (k , X)

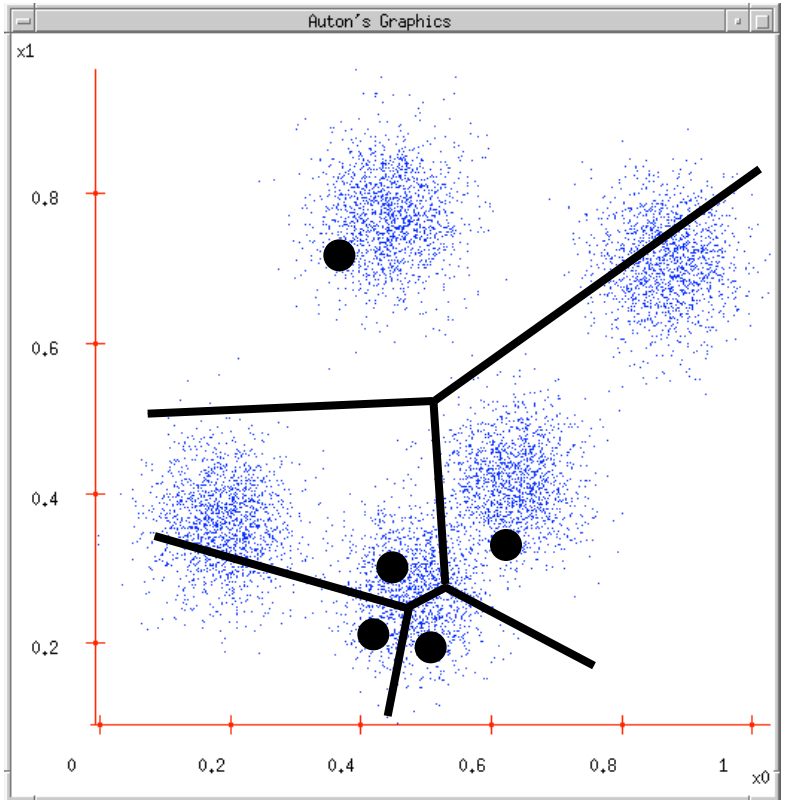
- Randomly choose k cluster center locations (centroids)
- Loop until convergence
  - Assign each point to the cluster of the closest centroid
  - Re-estimate the cluster centroids based on the data assigned to each cluster



K-Means Clustering

K-Means (k , X)

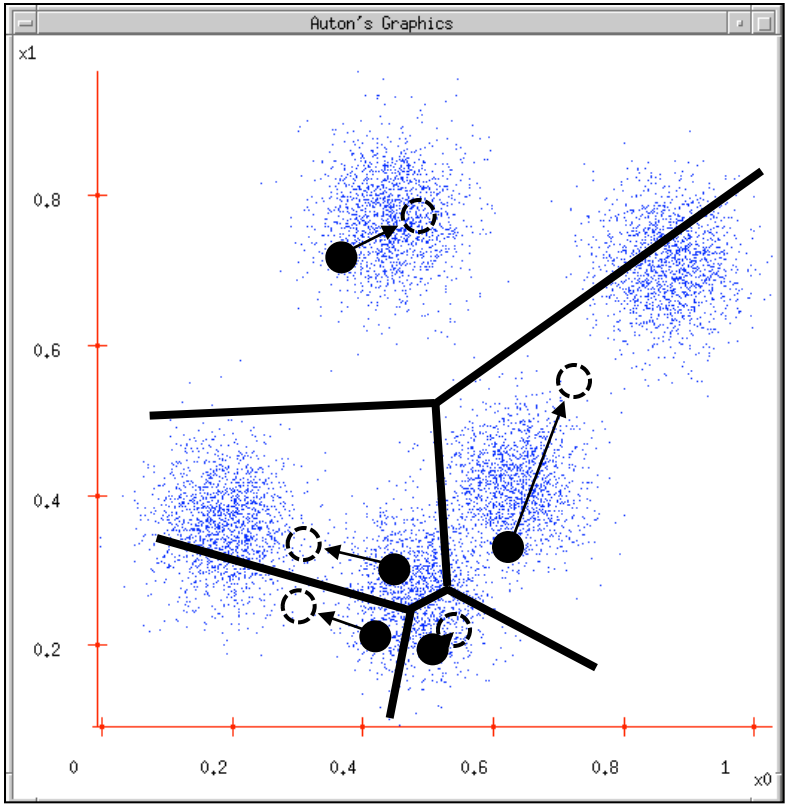
- Randomly choose k cluster center locations (centroids)
- Loop until convergence
  - Assign each point to the cluster of the closest centroid
  - Re-estimate the cluster centroids based on the data assigned to each cluster



K-Means Clustering

K-Means (k , X)

- Randomly choose k cluster center locations (centroids)
- Loop until convergence
  - Assign each point to the cluster of the closest centroid
  - Re-estimate the cluster centroids based on the data assigned to each cluster



K-Means Objective Function

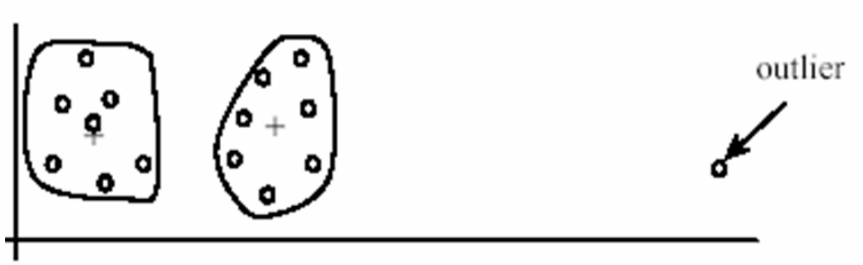
- K-means finds a local optimum of the following objective function:

$$\arg\min_{\boldsymbol{\mathcal{S}}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in \mathcal{S}_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|_2^2$$

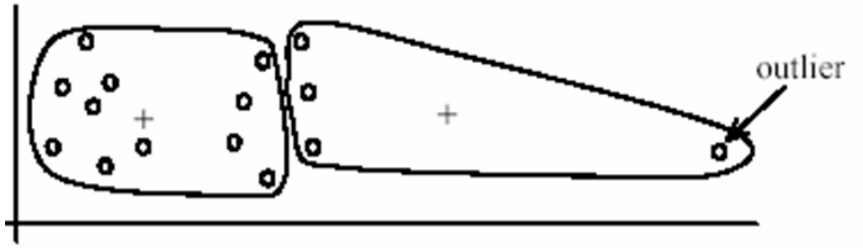
where $\boldsymbol{\mathcal{S}} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$ is a partitioning over $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ s.t. $X = \bigcup_{i=1}^{k} \mathcal{S}_i$ and $\boldsymbol{\mu}_i = \text{mean}(\mathcal{S}_i)$

K-Means pros and cons

- Pros
  - Finds cluster centers that minimize conditional variance (good representation of data)
  - Easy to implement
- Cons
  - Need to choose K
  - Sensitive to outliers
  - Prone to local minima
  - All clusters have the same parameters (e.g., distance measure is non-adaptive)



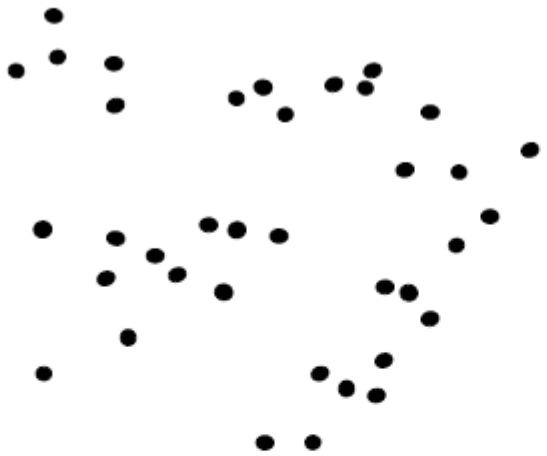
(B): Ideal clusters



How might we cluster?

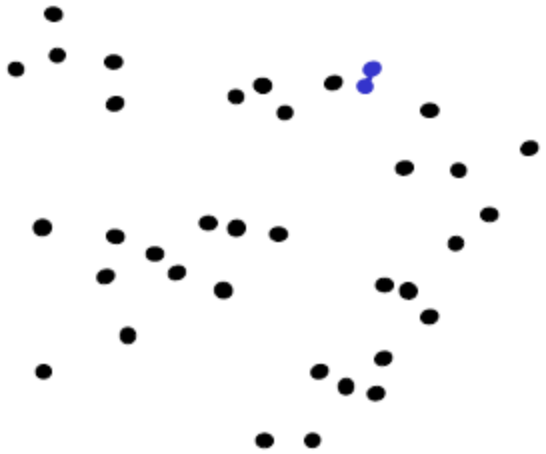
- K-means
  - Iteratively re-assign points to the nearest cluster center
- Agglomerative clustering
  - Start with each point as its own cluster and iteratively merge the closest clusters

Agglomerative clustering



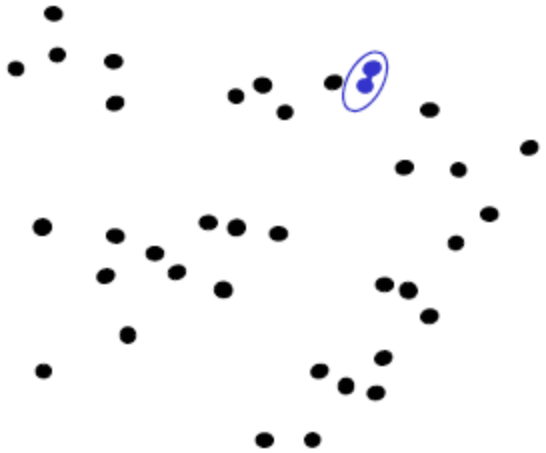
1. Say "Every point is its own cluster"

Agglomerative clustering



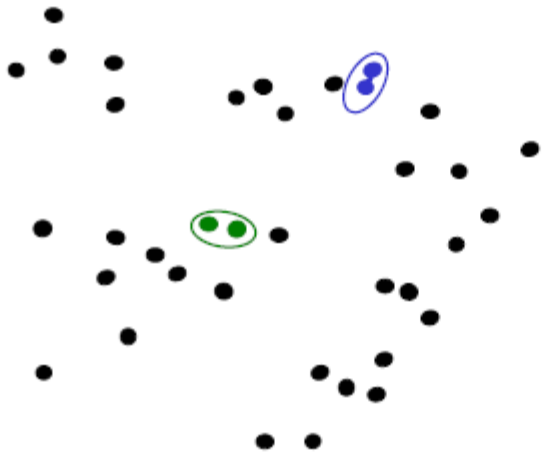
1. Say "Every point is its own cluster"
2. Find "most similar" pair of clusters

Agglomerative clustering



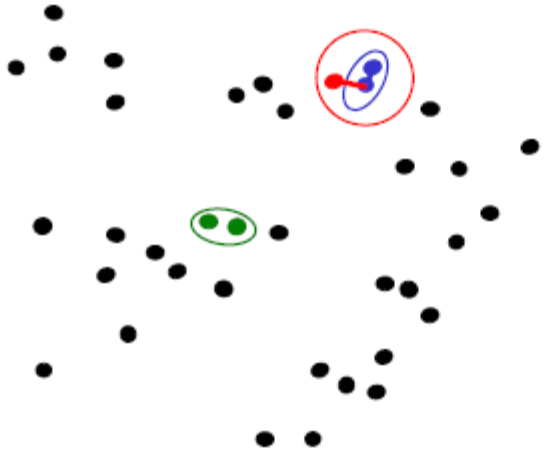
1. Say "Every point is its own cluster"
2. Find "most similar" pair of clusters
3. Merge it into a parent cluster

Agglomerative clustering

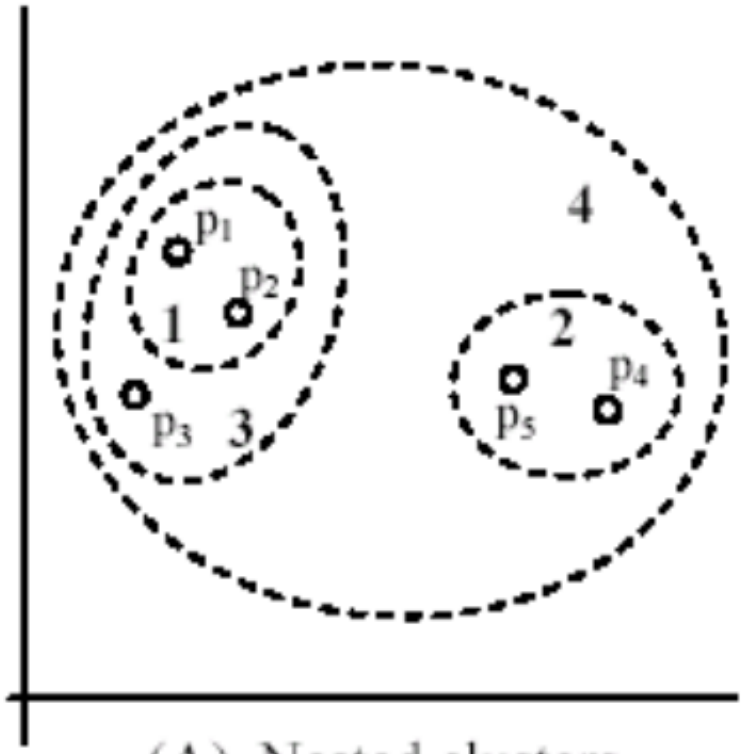


1. Say "Every point is its own cluster"
2. Find "most similar" pair of clusters
3. Merge it into a parent cluster
4. Repeat

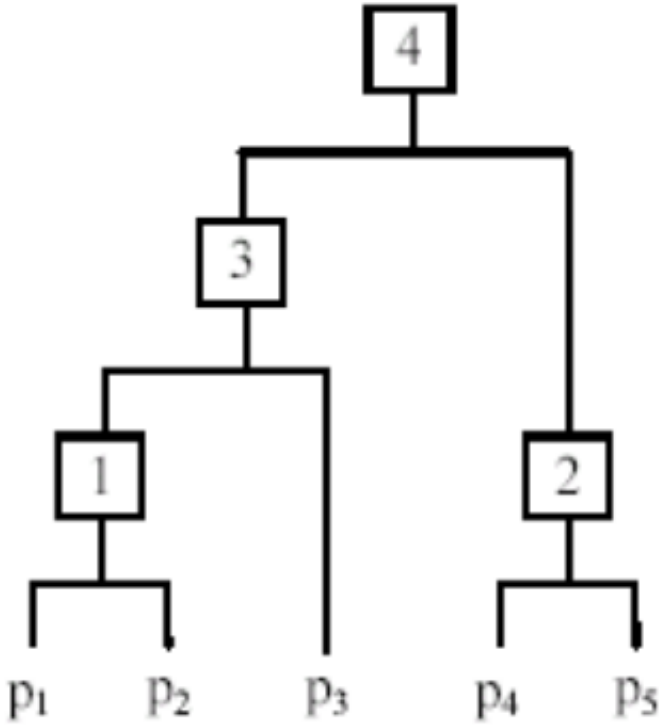
Agglomerative clustering



1. Say "Every point is its own cluster"
2. Find "most similar" pair of clusters
3. Merge it into a parent cluster
4. Repeat



(A). Nested clusters



(B) Dendrogram

Performance metrics

1. Metrics

2. Precision and recall

3. Receiver Operating Characteristic (ROC) curves

4. Worked example

Metrics

It is extremely important to use **quantitative metrics** for evaluating a machine learning model

- Until now, we relied on the **cost function value** for regression and classification

- Other metrics can be used to **better evaluate** and understand the model

- **For classification**
  - ✓ Accuracy/Precision/Recall/F1-score, ROC curves,…
- **For regression**
  - ✓ Normalized RMSE, Normalized Mean Absolute Error (NMAE),…

Confusion matrix

Estiamted class	Actual class: 1 (p)	Actual class: 0 (n)
1 (Y)	True positive (TP)	False positive (FP)
0 (N)	False negative (FN)	True negative (TN)

Accuracy

Accuracy is a measure of **how close** a given set of guessing from our model are closed to their true value.

$$\text{Accuracy} = \frac{\#\ \text{Correct classifications}}{\#\ \text{All classifications}}$$

- If a classifier make 10 predictions and 9 of them are correct, the accuracy is 90%.

- Accuracy is a measure of **how well** a binary classifier correctly identifies or excludes a condition.
- It's the **proportion of correct predictions among the total number of cases examined.**

• consider a fraud detection model for credit card transactions. Suppose we have a dataset with 1000 transactions, where 990 are legitimate (majority class) and 10 are fraudulent (minority class).

Model Performance:

- The model correctly identifies 980 legitimate transactions (true negatives) and 8 fraudulent transactions (true positives).
- However, it misclassifies 2 fraudulent transactions as legitimate (false negatives) and 10 legitimate transactions as fraudulent (false positives).

Accuracy Calculation:

- Total correct predictions: 980 (true negatives) + 8 (true positives) = 988

  Accuracy: 988/1000 = 0.988 or 98.8%

The problem:

- The accuracy metric suggests the model is performing extremely well (98.8%), but it's actually performing poorly on the minority class (fraudulent transactions).
- The model only detected 8 out of 10 fraudulent transactions (80% recall) and misclassified 2 fraudulent transactions as legitimate (false negatives).

Precision and recall

Suppose that $y = 1$ in presence of a **rare class** that we want to detect

Precision *(How much we are precise in the detection)*

Of all patients where we classified $y = 1$, what fraction actually has the disease?

$$\frac{\text{True Positive}}{\text{\# Estimated Positive}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

Recall *(How much we are good at detecting)*

Of all patients that actually have the disease, what fraction did we correctly detect as having the disease?

$$\frac{\text{True Positive}}{\text{\# Actual Positive}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

Confusion matrix

Estiamted class \ Actual class	1 (p)	0 (n)
1 (Y)	True positive (TP)	False positive (FP)
0 (N)	False negative (FN)	True negative (TN)

Trading off precision and recall

Logistic regression: $0 \leq s(\boldsymbol{\varphi}^{\mathsf{T}}\boldsymbol{\theta}) \leq 1$

- Classify 1 if $s(\boldsymbol{\varphi}^{\mathsf{T}}\boldsymbol{\theta}) \geq 0.5$
- Classify 0 if $s(\boldsymbol{\varphi}^{\mathsf{T}}\boldsymbol{\theta}) < 0.5$

These thresholds can be different from 0.5!

→ *At different thresholds, correspond different confusion matrices!*

Suppose we want to classify $y = 1$ (disease) only if very confident

- Increase threshold → Higher precision, lower recall

Suppose we want to avoid missing too many cases of disease (avoid false negatives)

- Decrease threshold → Higher recall, lower precision

F1-score

It is usually better to compare models by means of one number only. The $\mathbf{F1-score}$ can be used to **combine precision and recall**

	Precision(P)	Recall (R)	Average	F_1 Score
Algorithm 1	0.5	0.4	0.45	0.444
Algorithm 2	0.7	0.1	0.4	0.175
Algorithm 3	0.02	1.0	0.51	0.0392

The best is Algorithm 1

Algorithm 3 classifies always 1

Average says not correctly that Algorithm 3 is the best

$$\text{Average} = \frac{P + R}{2} \qquad F_1\text{score} = 2\frac{P \cdot R}{P + R}$$

- $P = 0$ or $R = 0 \Rightarrow F_1\text{score} = 0$
- $P = 1$ and $R = 1 \Rightarrow F_1\text{score} = 1$

Summaries of the confusion matrix

Different metrics can be computed from the confusion matrix, depending on the class of interest *(https://en.wikipedia.org/wiki/Precision_and_recall)*

		True condition				
	Total population	Condition positive	Condition negative	Prevalence = $\frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$	Accuracy (ACC) = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$	
Predicted condition	Predicted condition positive	**True positive**, Power	**False positive**, Type I error	Positive predictive value (PPV), Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Predicted condition positive}}$	
	Predicted condition negative	**False negative**, Type II error	**True negative**	False omission rate (FOR) = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Predicted condition negative}}$	
		True positive rate (TPR), Recall, Sensitivity, probability of detection = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR−}}$	F_1 score = $\frac{1}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}{2}$
		False negative rate (FNR), Miss rate = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	Negative likelihood ratio (LR−) = $\frac{\text{FNR}}{\text{TNR}}$		

Ranking instead of classifying

Classifiers such as logistic regression can output a **probability** of belonging to a class (or something similar)

- We can use this to **rank** the different istances and take actions on the cases at top of the list

- We may have a **budget**, so we have to target most promising individuals

- Ranking enables to use different techniques for **visualizing** model performance

Ranking instead of classifying

Instance description	True class	Score
……	1	0,99
……	1	0,98
……	0	0,96
……	0	0,90
……	1	0,88
……	1	0,87
……	0	0,85
……	1	0,80
……	0	0,70

	p	n
Y	0	0
N	100	100

	p	n
Y	1	0
N	99	100

	p	n
Y	2	0
N	98	100

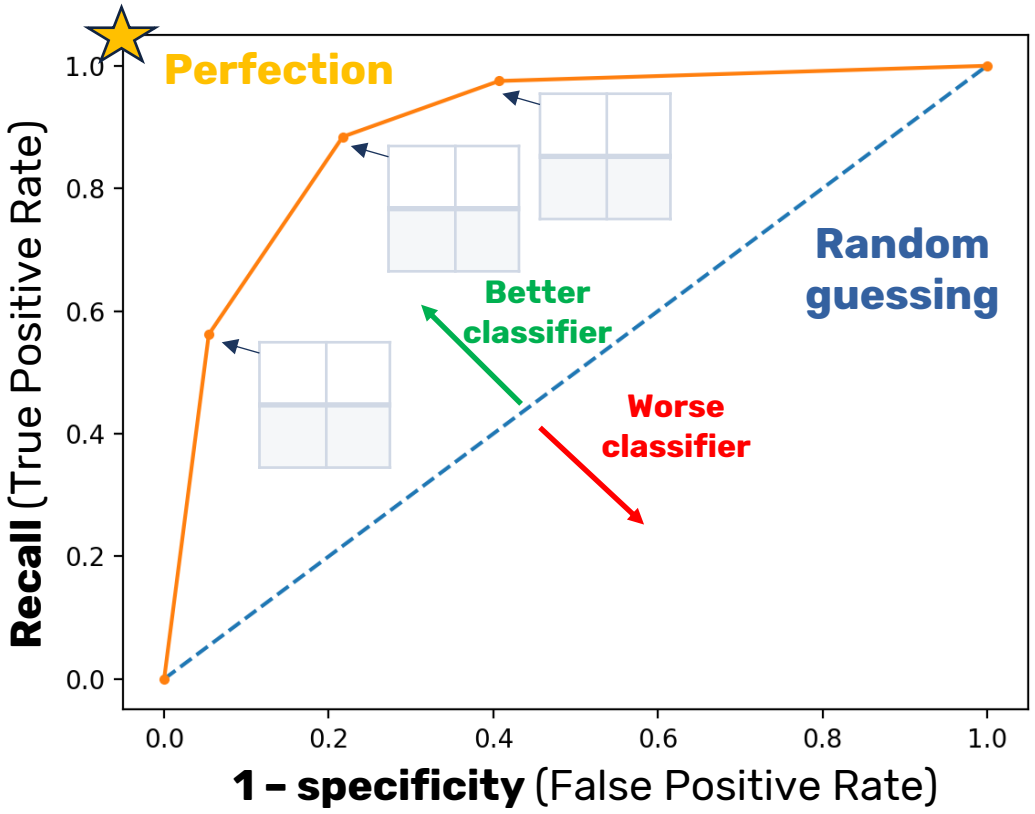
	p	n
Y	2	1
N	98	99

	p	n
Y	6	4
N	94	96

Different confusion matrices by changing the **threshold**

Ranking instead of classifying

ROC curves are a very general way to **represent and compare** the performance of different models (on a binary classification task)

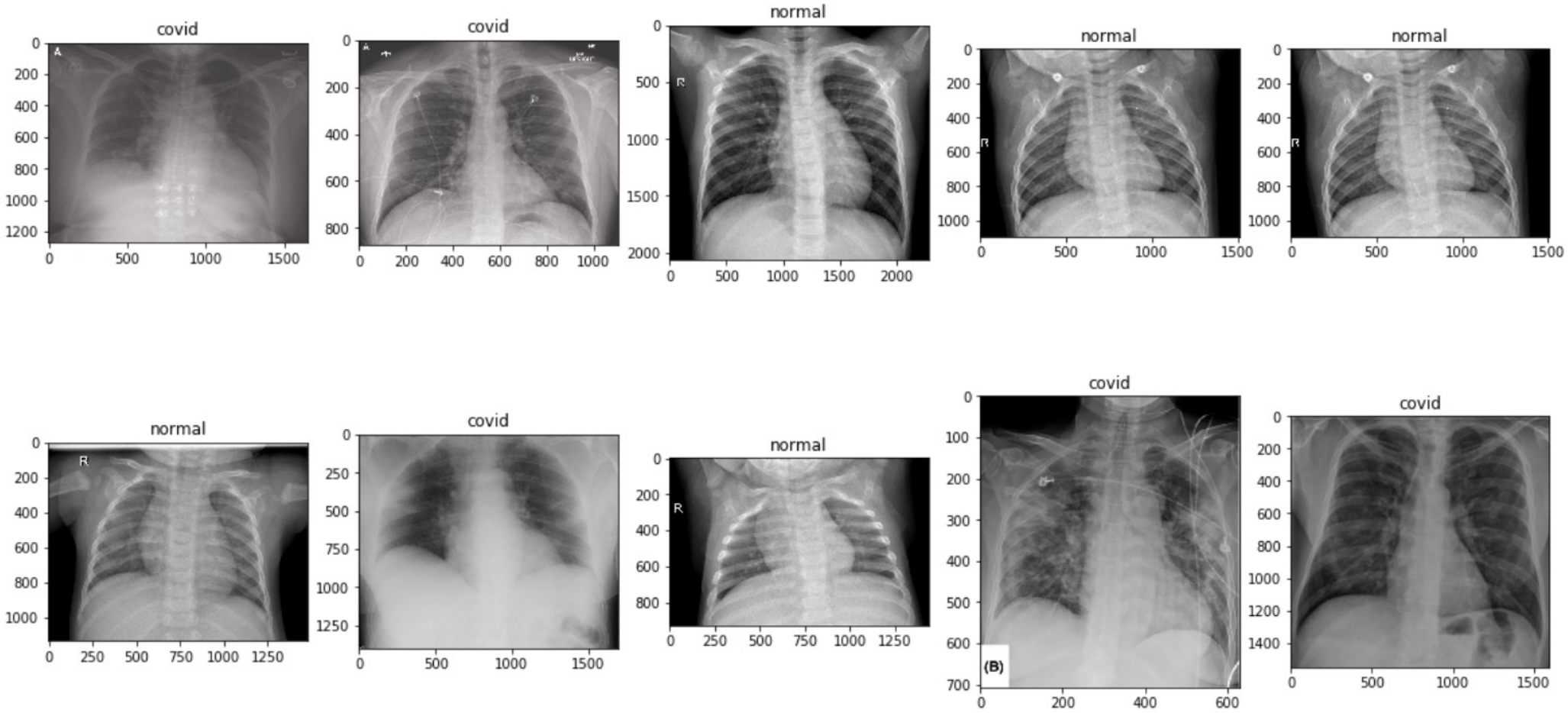


Observations

- $(0,0)$: classify always negative
- $(1,1)$: classify always positive
- Diagonal line: random classifier
- Below diagonal line: worse than random classifier
- Different classifiers can be compared
- **Area Under the Curve (AUC):** probability that a randomly chosen positive instance will be ranked ahead of randomly chosen negative instance

Pneumonia detection

Suppose to have at disposal X-ray images of lungs: **Healthy** people - **Covid-19 disease** patients



Acknowledgments

We want to use a classifier to perform classification:

- **Healthy** patients: class 0
- Patients with a **disease**: class 1

The input data are directly the X-ray **images**

For these computer vision tasks, the state of the art algorithm are the **Convolutional Neural Networks:**

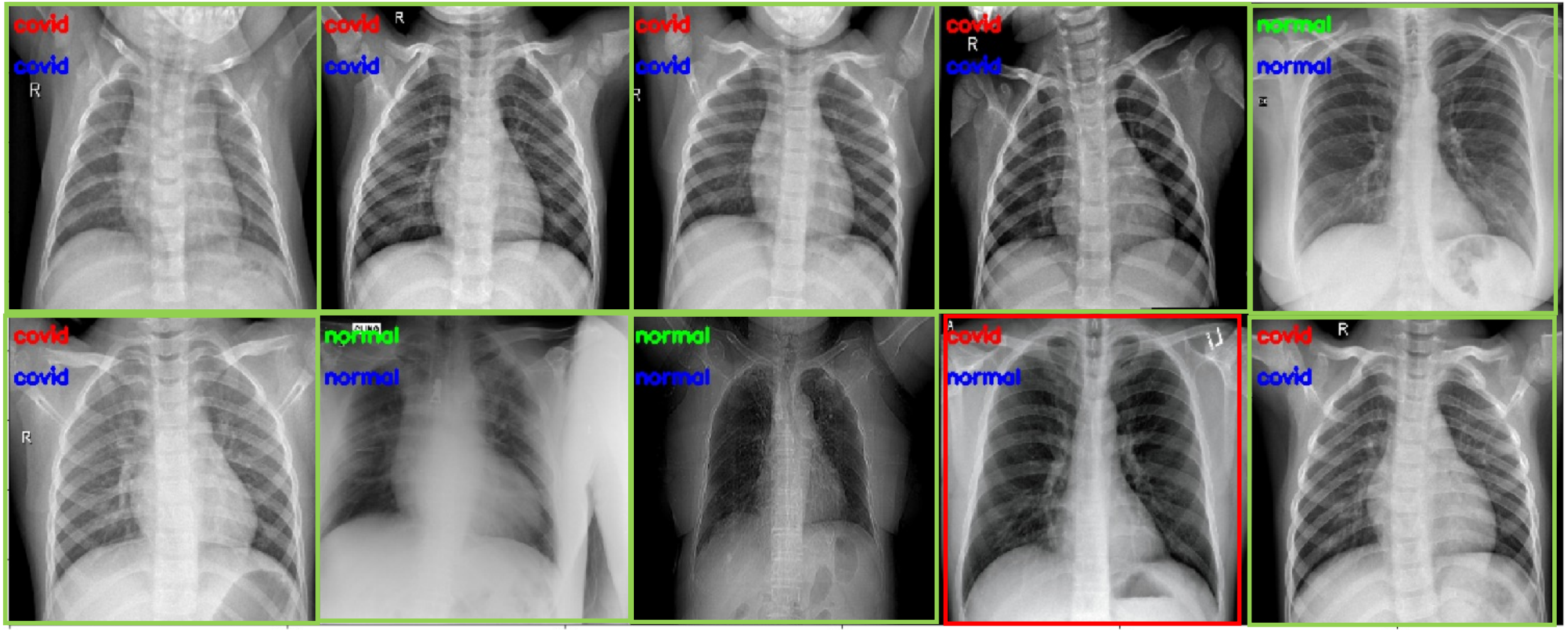
- we can use them to classify the images into **healthy** and **disease**

Pneumonia detection

True label

Estimated covid label

Estimated healthy label



Pneumonia detection

Classification results on test set

Sensitivity (recall, true positive rate)

$$\frac{\text{True Positive}}{\text{\# Actual Positive}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} = 0.92$$

Specificity (true negative rate)

$$\frac{\text{True Negative}}{\text{\# Actual Negative}} = \frac{\text{True Negative}}{\text{False Positive} + \text{True Negative}} = 1$$

Estimated class	Actual class: 1 (p)	Actual class: 0 (n)
1 (Y)	True positive 11	False positive 0
0 (N)	False negative 1	True negative 11

- **Accuracy**: ≈ 96%

Pneumonia detection

Classification results on test set

Sensitivity (recall, true positive rate)

$$\frac{\text{True Positive}}{\text{\# Actual Positive}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} = 0.92$$

Specificity (true negative rate)

$$\frac{\text{True Negative}}{\text{\# Actual Negative}} = \frac{\text{True Negative}}{\text{False Positive} + \text{True Negative}} = 1$$

- **Sensitivity:** of patients that ***do have*** COVID-19 (i.e., *true positives*), we could accurately identify them as "COVID-19 positive" 92% of the time using our model

- **Specificity:** of patients that ***do not have*** COVID-19 (i.e., true negatives), we could accurately identify them as "COVID-19 negative" 100% of the time using our model.

Pneumonia detection

Classification results on test set

Sensitivity (recall, true positive rate)

$$\frac{\text{True Positive}}{\text{\# Actual Positive}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} = 0.92$$

Specificity (true negative rate)

$$\frac{\text{True Negative}}{\text{\# Actual Negative}} = \frac{\text{True Negative}}{\text{False Positive} + \text{True Negative}} = 1$$

- Being able to **accurately detect healthy patients** with 100% accuracy is great. We do not want to quarantine someone for nothing

- …but **we don't want to classify someone as «healthy» when they are «COVID-19 positive»**, since it could infect other people without knowing

Summary

Balancing sensitivity and specificity is incredibly challenging when it comes to medical applications

The results should **always be validated** with another pool of people

Furthermore, we need to be **concerned of what the model is actually learning:**

- Does the results align with the medical knowledge?
- Was the dataset well representative of the population or there was selection bias?