Supervised Learning

Advanced methods

Danilo Montesi
Stefano Giovanni Rizzo

We have seen so far...

- Until now we **have seen basic** but powerful methods based on the linear combination of input variables.
- Logistic and linear regression differ from the methods we will present here because:
  - They are obtained from minimizing a global loss function through weights tuning.
  - They results in a "**model**": a simple **mathematical function** from the input variables to the output variable
  - The obtained model (classifier or regressor) does not need particular data, algorithms or tricks to work: only the weights.
- We will go further to show tools, algorithms and tricks that have brought great results to the machine learning scene in the last decades.

Advanced methods

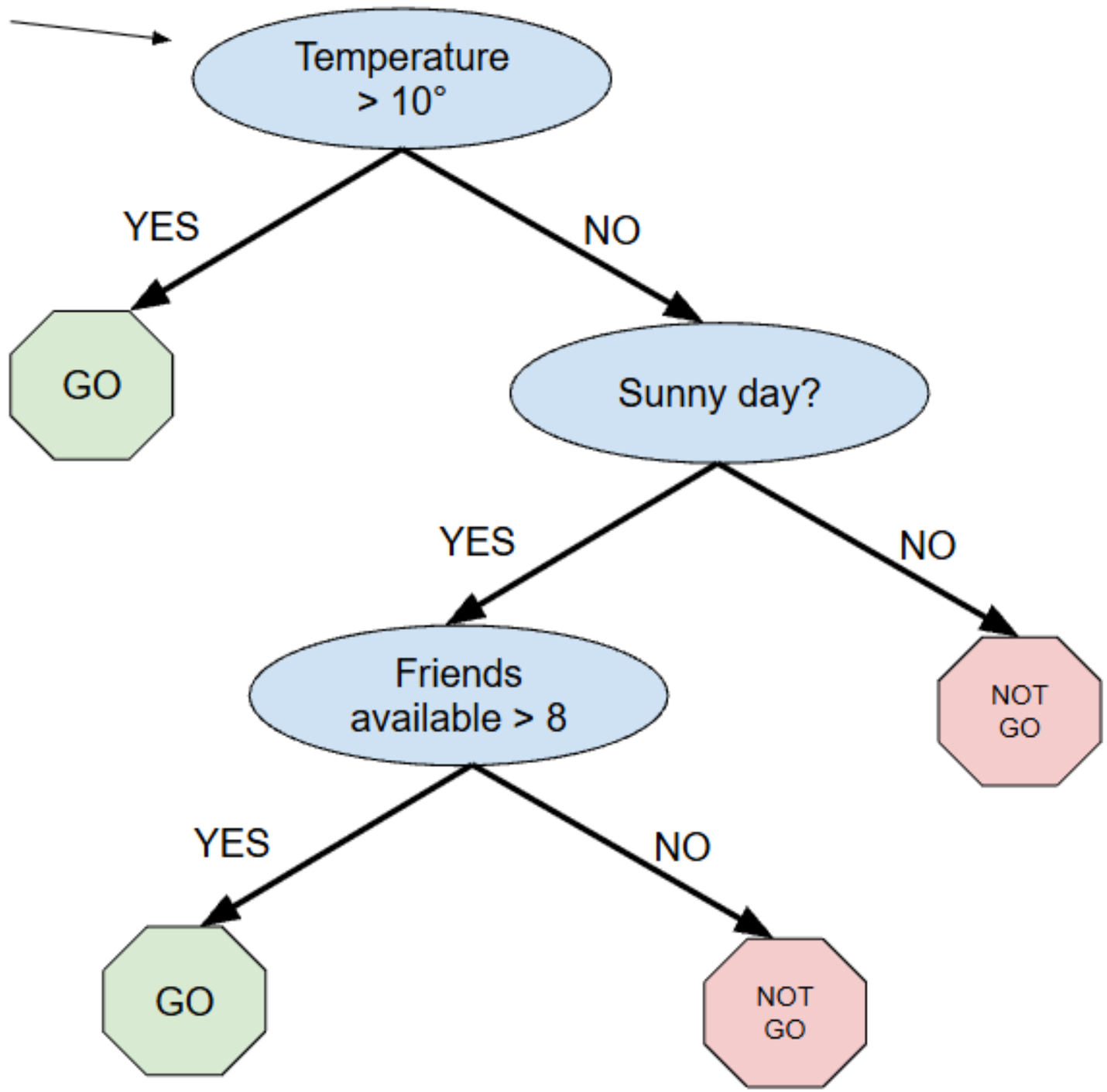
- We will go through the main classes of supervised learning methods, by their main principles of work:
  - **Decision trees**: powerful decision support tools. We will see how they can be generated using algorithms such as CART.
  - **k-Nearest Neighbors (k-NN)**: algorithm based on similarity search, does not need a training phase, use all the data to predict new examples. It's able to classify non linearly separable classes.
  - **Support Vector Machine (SVM)**: divides the space in two and classify depending on the position of new examples. It uses the **kernel trick**: like k-NN considers distance between examples, making the model non linear.

Decision tree

- Decision trees are quite easy to grasp for a human mind, because they are **a natural way of reasoning**.
- Let's say we must decide whether to go play golf this afternoon or not.
- We should take different variable into account (our **features X**) in order to decide whether to **go** or **not go** (our binary **output variable y**).
- The features may be:
  - The temperature outside (real value)
  - The weather: if it's sunny or raining (categorical)
  - How many friends are available to play (integer)
  - …
- Let's see the structure of a decision tree needed to make this decision.

Decision tree structure

Decision node: is this predicate satisfied? Yes or no.



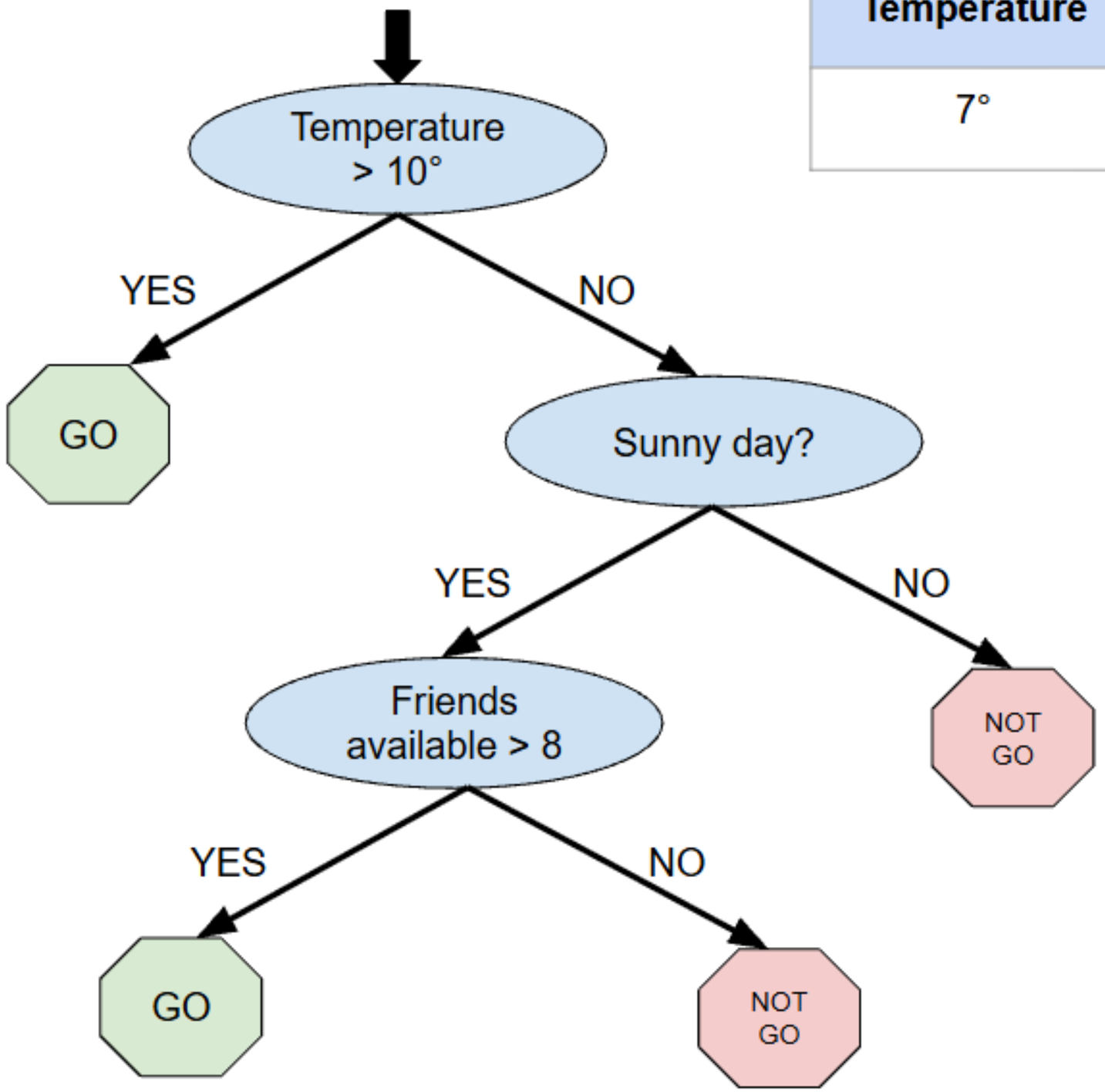
Leaf node: predict a value (or class)

Prediction process 1/2

- We have an example x_i with all its features, for example:
  - Temperature
  - Sunny or not
  - Number of friends available
- We just “drop” the example x_i down the tree until it hits a leaf node.
- The leaf node will be the predicted class or value: a class if we are doing a classification, a value if we are doing a regression.

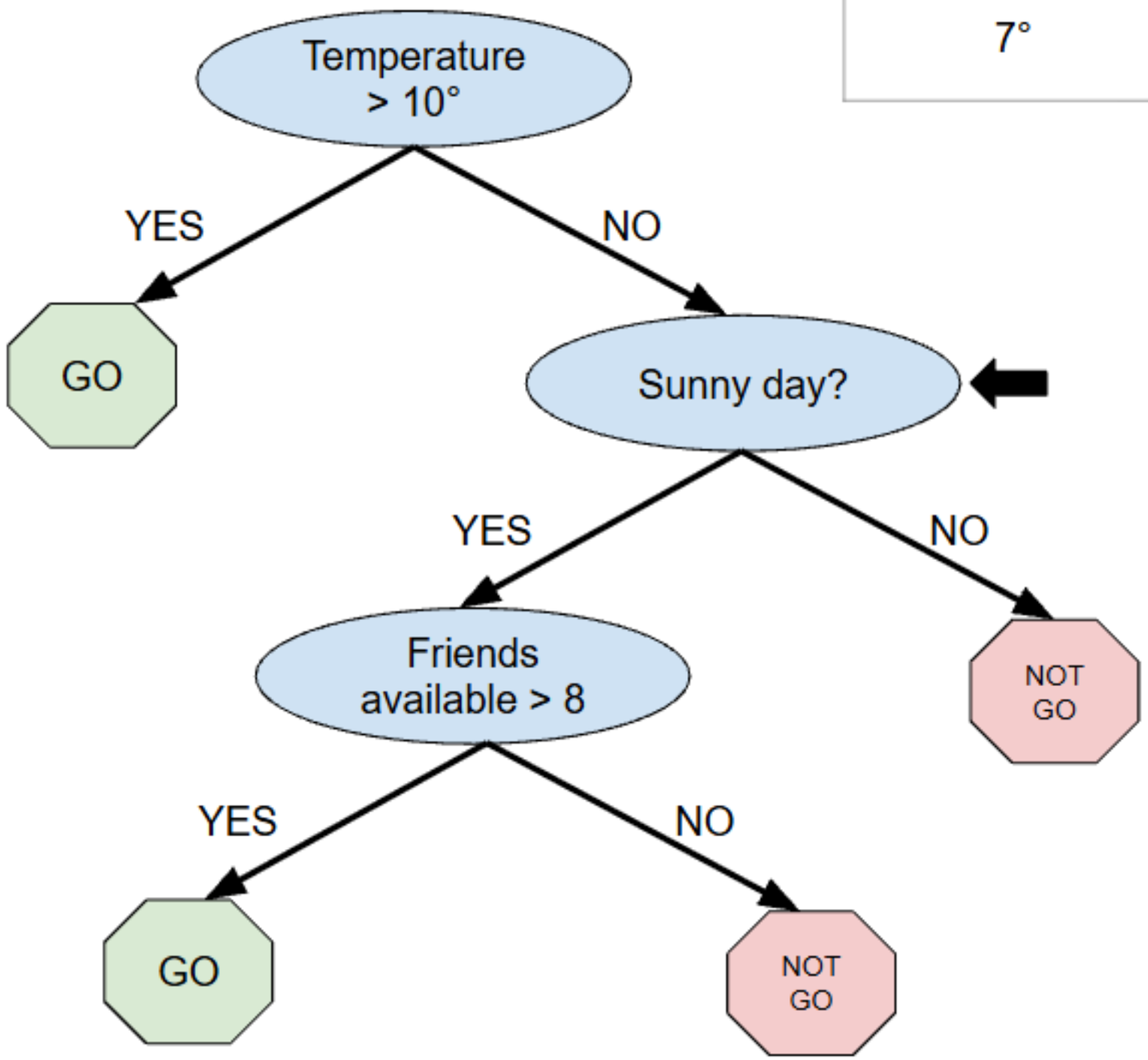
Prediction process 2/2

Temperature	Sunny	Friends
7°	YES	5



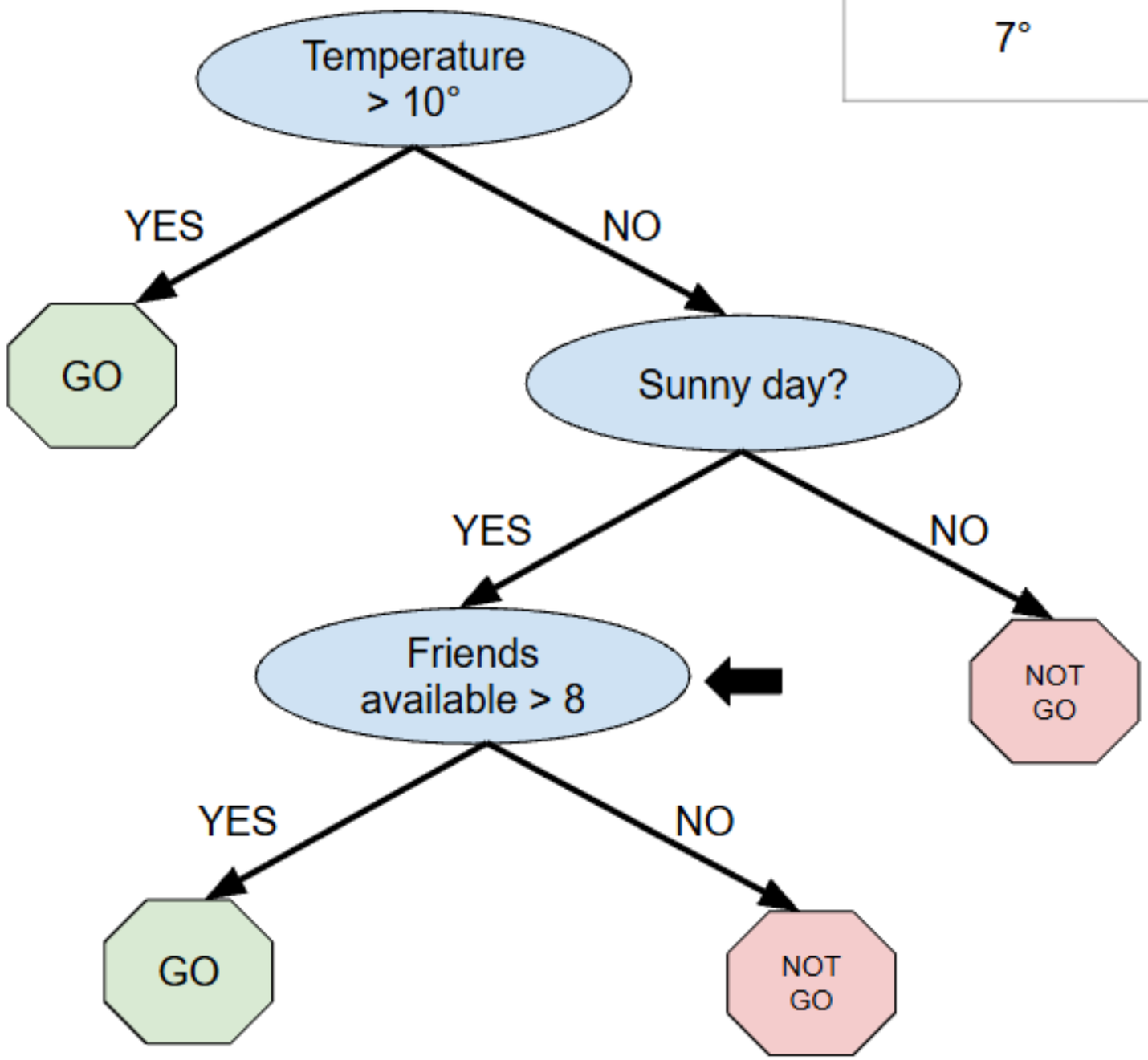
Prediction process 2/2

Temperature	Sunny	Friends
7°	YES	5



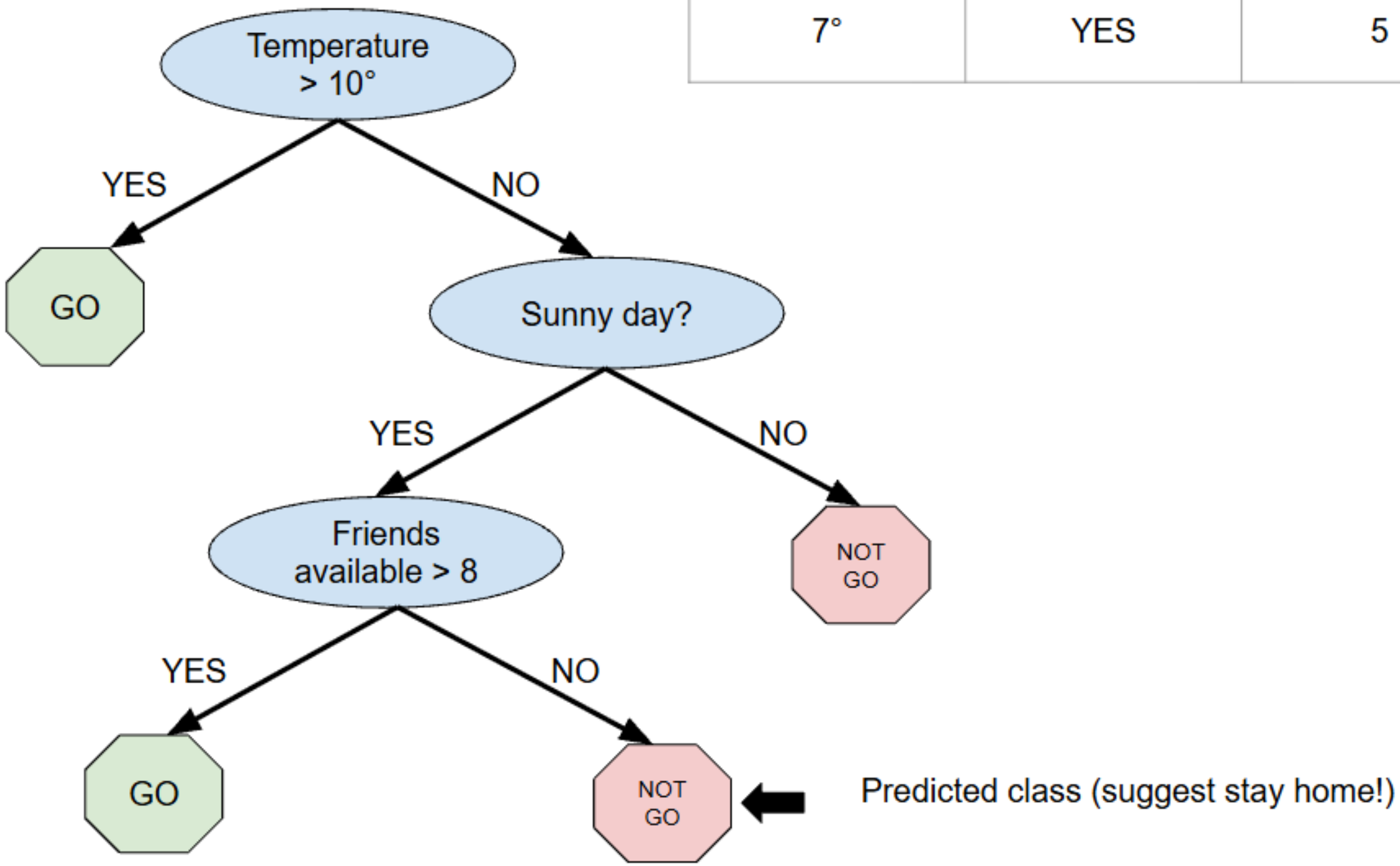
Prediction process 2/2

Temperature	Sunny	Friends
7°	YES	5



Prediction process 2/2

Temperature	Sunny	Friends
7°	YES	5

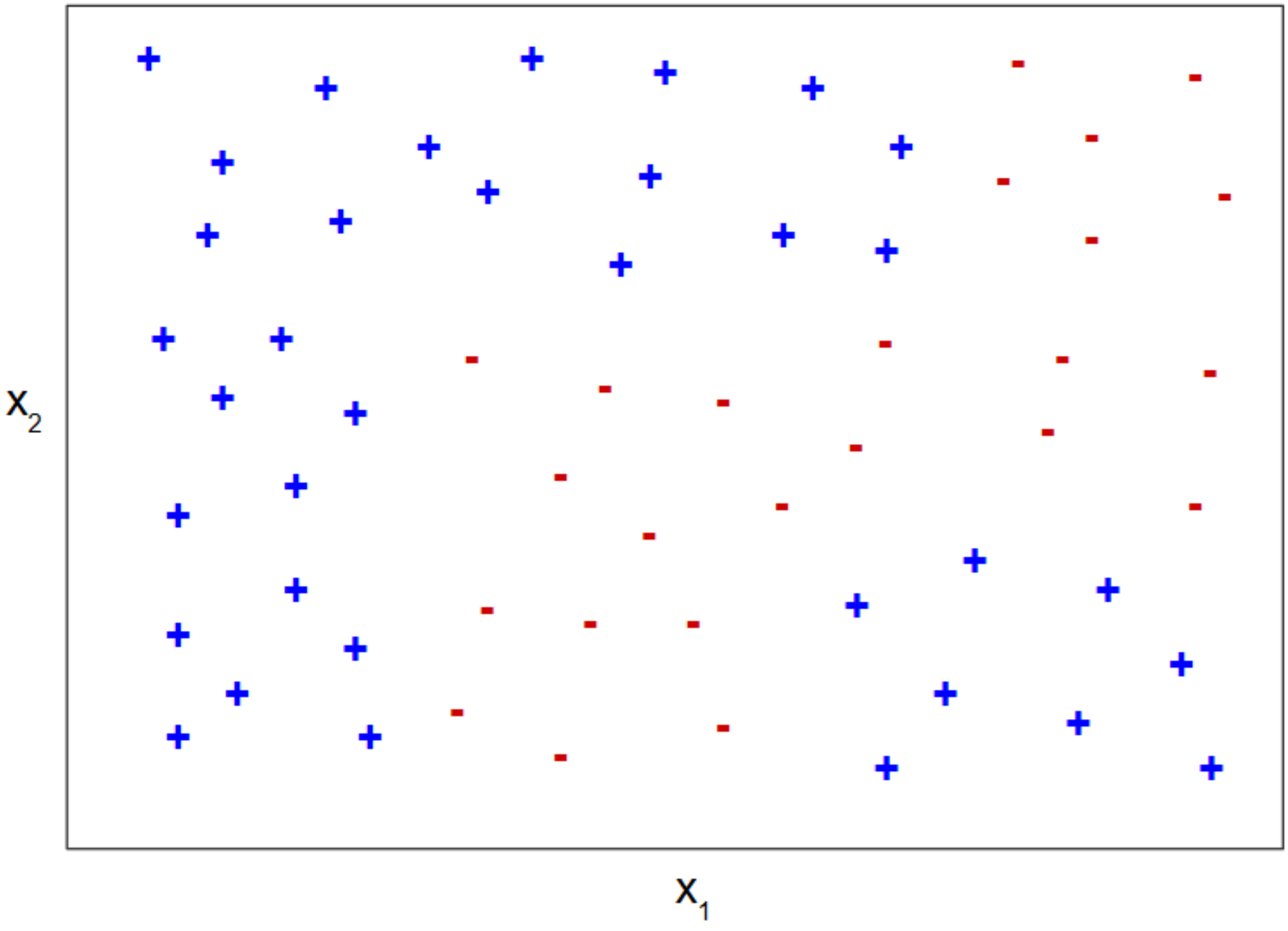


Decision boundaries

- As always, remind that we represent our data in a vector space.
  - The space has a dimension for each feature in the dataset.
  - The last example had 3 dimension : temperature, sunny, number of friends.
  - Each record in the dataset is a point in the vector space.
- A decision tree describe a set of decision boundaries on the vector space.
  - The decision tree divides the space in smaller regions, given by the condition nodes.
  - Each region correspond to a leaf node.
  - Each record will belong to a particular region, thus will be classified as the corresponding leaf node.

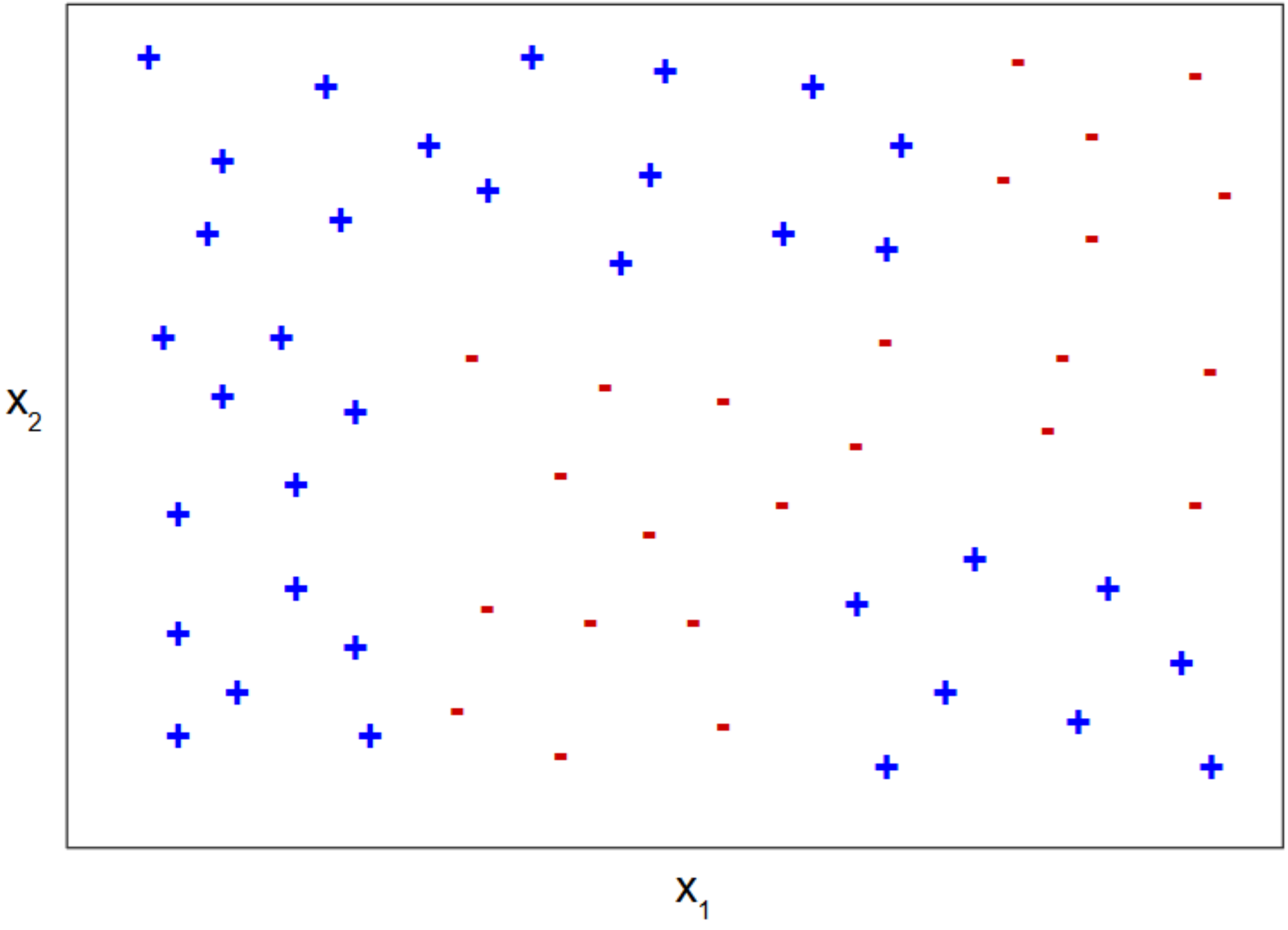
Training data 1/2

- This training data has two dimensions (features)
- Output class is binary (either "+" or "-")



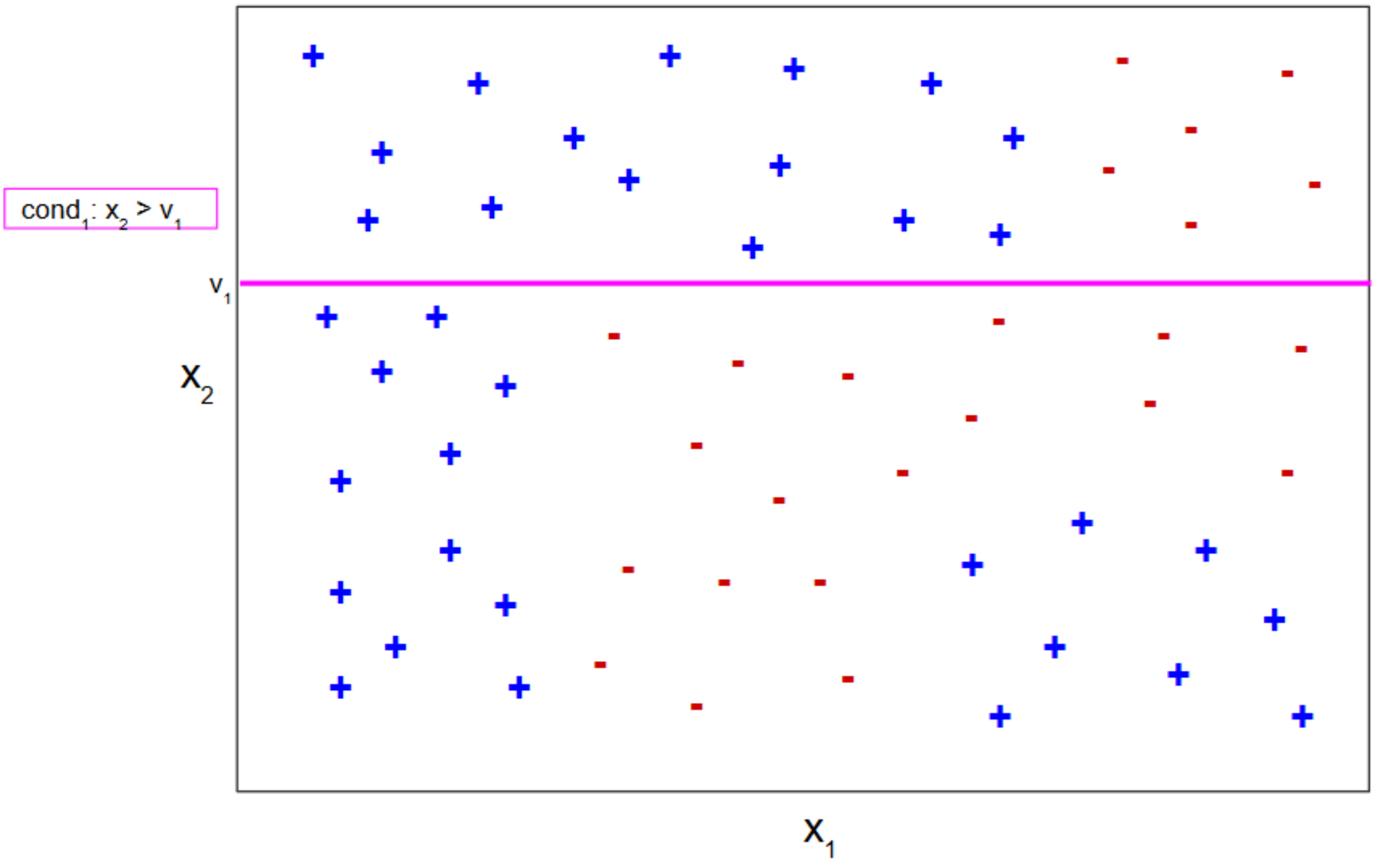
Training data 2/2

- The training algorithm finds the best condition every time to divide the space into **homogeneous** regions (having points of the same class)



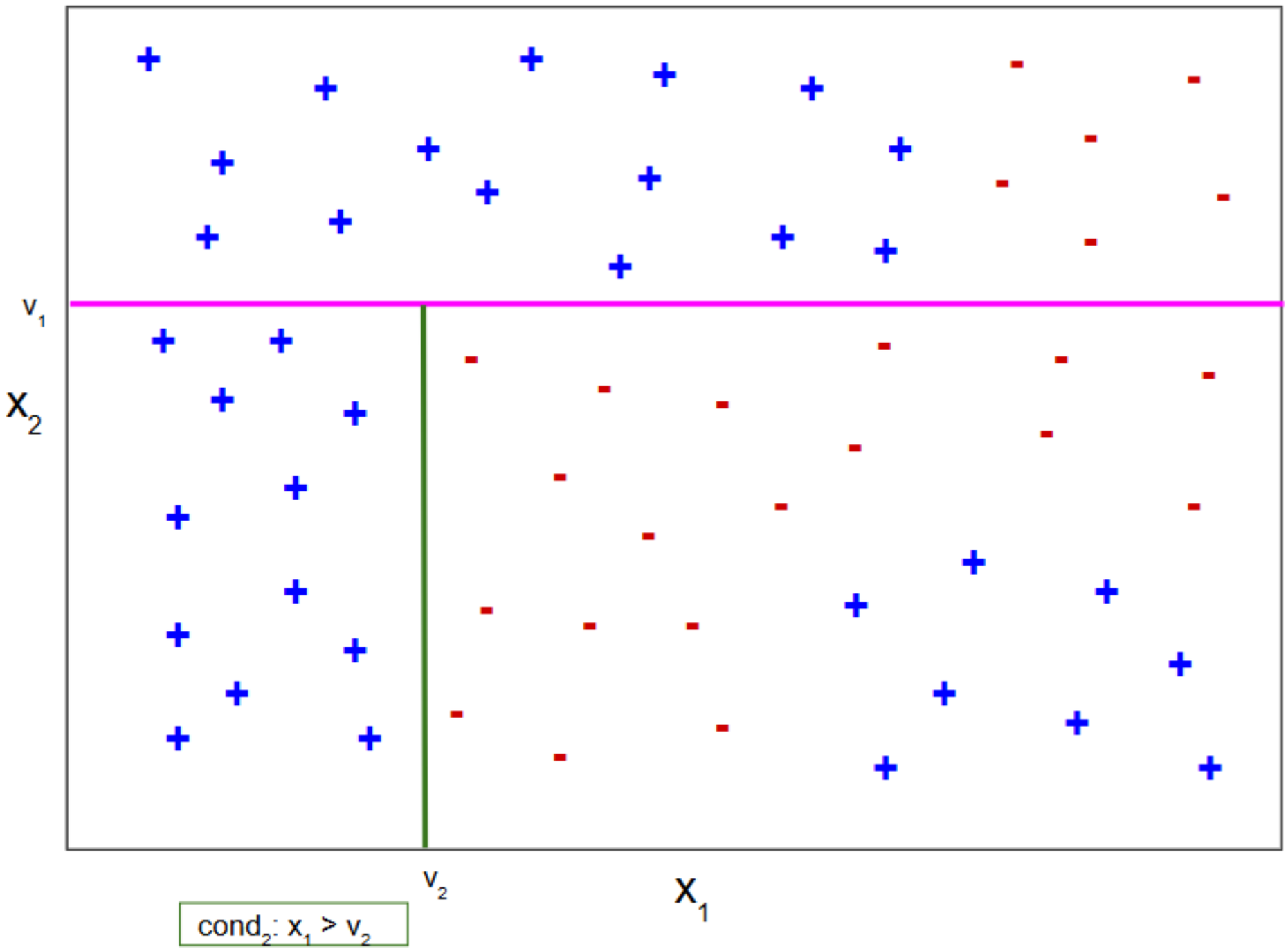
Training data 2/2

- The training algorithm finds the best condition every time to divide the space into **homogeneous** regions (having points of the same class)



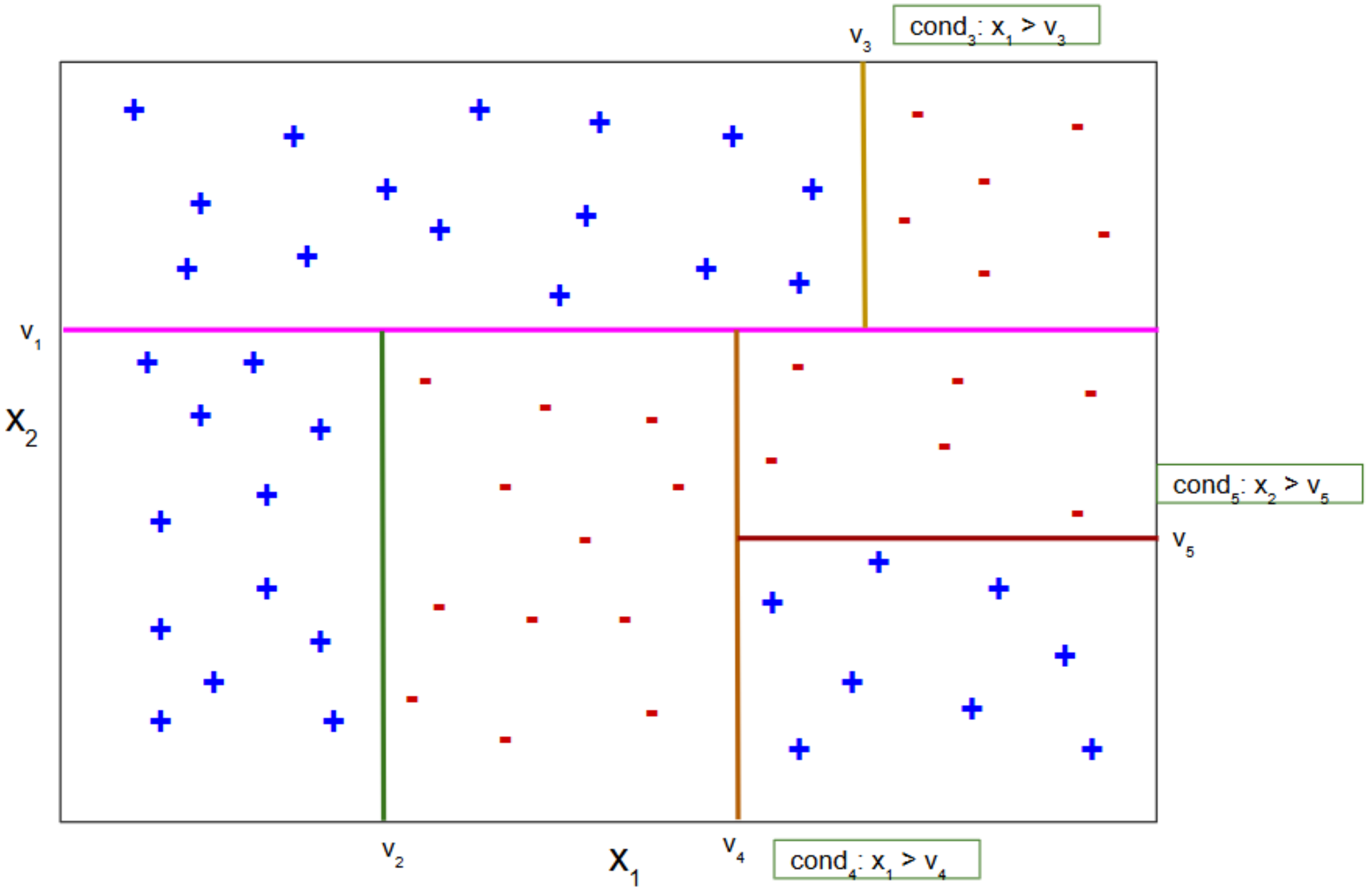
Training data 2/2

- The training algorithm finds the best condition every time to divide the space into **homogeneous** regions (having points of the same class)



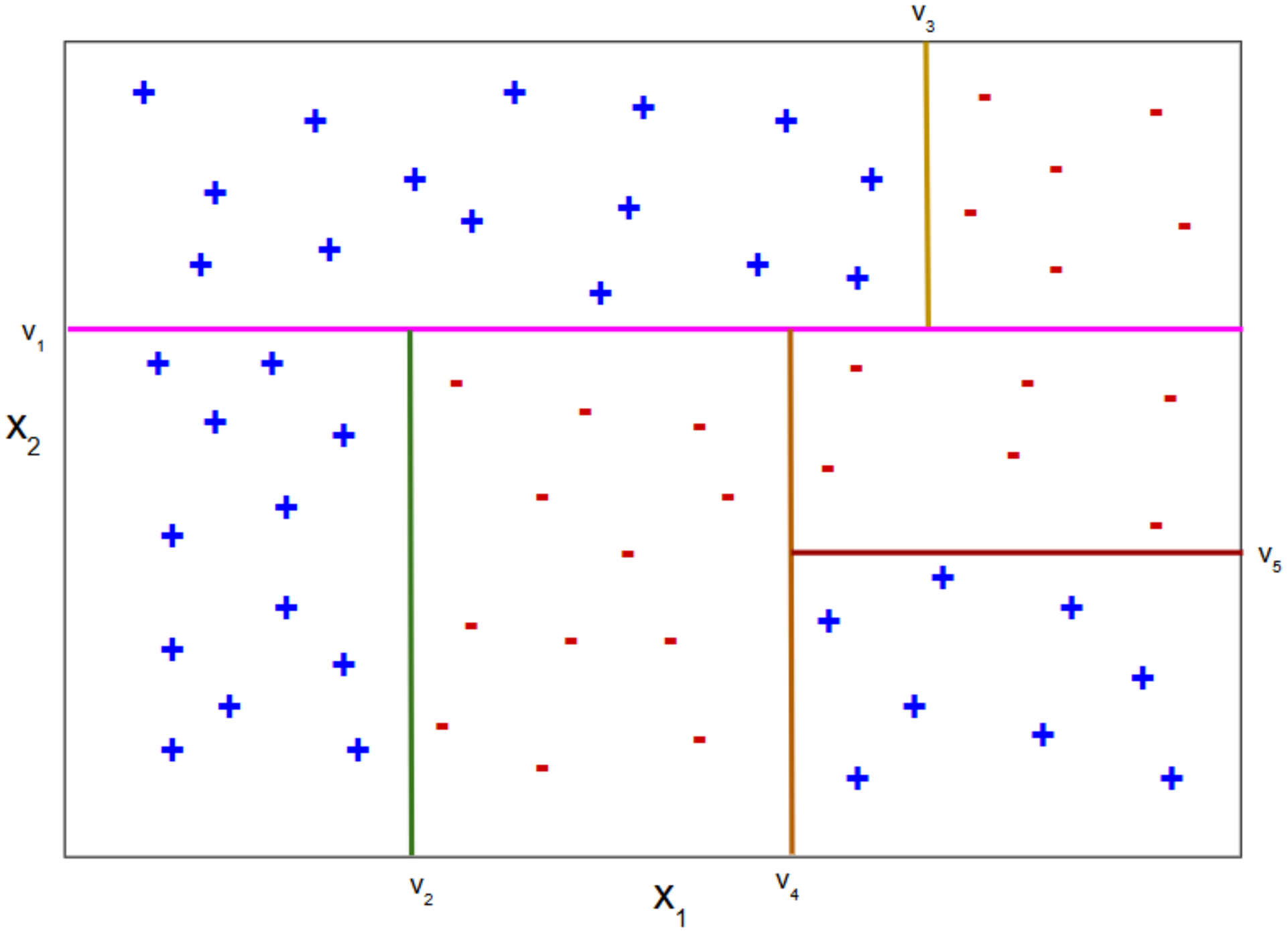
Training data 2/2

- The training algorithm finds the best condition every time to divide the space into **homogeneous** regions (having points of the same class)



Training data 2/2

- The training algorithm finds the best condition every time to divide the space into **homogeneous** regions (having points of the same class)



Learning a decision tree

- The space splitting representation is equivalent to a decision tree, in which each **boundary line** is the **condition** of a decision node.
- Basically, the learning process has the following recursive steps:
  - Find the **best attribute** to split the current data
  - Create a **branch for each value** in the selected attribute
  - If branch has more than one class, **recursively** apply the two steps above
- We will see now the specific case of the simple **ID3 algorithm** to better understand the process.

ID3

- ID3 is a simple algorithm for building decision trees from data, and also one of the first (1975).
- ID3 is the precursor to the popular C4.5 algorithm, which introduces additional improvements.
- It employs a **top-down**, **greedy** search to generate the tree nodes:
  - top-down means it start from the whole dataset (root node split) down to the leafs, that represent subsets of the dataset with the same class
  - greedy means it picks the best attribute at each step, it never looks back to reconsider earlier choices that may lead to a better tree.
- It works on **categorical data only** (no numerical attributes).

ID3 Algorithm

D = Training set of examples
Classes = two classes ‘YES’ or ‘NO’
Atts = Set of attributes that may be tested by the learned decision tree
function ID3(D,Classes, Atts){
create Root node for the tree
if D are all ‘YES’ then Root = ‘YES’ and **exit**
else if D are all ‘NO’ then Root = ‘NO’ and **exit**
if Attrs = ∅ **then** Root = most common class in D
else
A = best decision attribute from Atts
Root = A
for each possible value v_i of A:
add a new tree branch with A=v_i for A
Dv_i = subset of D that have value v_i for A
if Dv_i = ∅ **then add** leaf = most common class in D
else add the subtree ID3(Dv_i, Classes, Atts-{A})
}

- “Best” in which terms?
- How do we define which one is the best attribute?
- We need a measure.

Entropy

- The **best attribute** is the one that, splitting on its values, it produces the most **homogeneous** subsets.
- In order to quantify the homogeneity we need the **entropy** measure.
- Entropy is a formula to calculate the **homogeneity** of a sample D considering all its classes c in the Classes set.

$$Entropy(Classes) = \sum_{c \in Classes} -p(c) log_2 p(c)$$

- **Example:** S is a collection with 2 classes (Play golf YES or NO) and 14 examples, of which 9 YES and 5 NO

Entropy(D) = - (9/14 $\log_2$ 9/14) - (5/14 $\log_2$ 5/14) = 0.94

Information gain

- Now we want to test a specific attribute to measure the homogeneity that it produces if we split on it.
  - We split the dataset using one branch for each different value of the attribute
  - The entropy for the subset of each branch is calculated, then it is added proportionally to the sizes of the branches to get the total entropy for the split
    - The resulting entropy is subtracted from the entropy before the split.
- This difference is the **Information Gain (IG)**, or decrease in entropy.
- **The attribute that yields the largest IG is chosen for the decision node.**
- Let's see an example.

Dataset

Outlook	Temp	Humidity	Windy	Play Golf
Rainy	Hot	High	False	No
Rainy	Hot	High	True	No
Overcast	Hot	High	False	Yes
Sunny	Mild	High	False	Yes
Sunny	Cool	Normal	False	No
Sunny	Cool	Normal	True	Yes
Overcast	Cool	Normal	True	No
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Sunny	Mild	Normal	False	Yes
Rainy	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Sunny	Mild	High	True	No

Dataset entropy

- We first compute the entropy for the whole dataset following the entropy formula

$$Entropy(Classes) = \sum_{c \in Classes} -p(c) log_2 p(c)$$

Play Golf	
Yes	No
9	5

Entropy(PlayGolf) = Entropy (5,9)
= Entropy (0.36, 0.64)
= - (0.36 $\log_2$ 0.36) - (0.64 $\log_2$ 0.64)
= 0.94

Attribute entropy

- We compute the entropy of the classes (yes and no) with respect to each value v (Sunny, Overcast, Rainy) of the attribute A (Overcast)

$$Entropy(Classes, A) = \sum_{v \in A} -p(v)Entropy(Classes)$$

		Play Golf		
		Yes	No	
Outlook	Sunny	3	2	5
	Overcast	4	0	4
	Rainy	2	3	5
				14

E(PlayGolf, Outlook) = **P**(Sunny)*__E__(3,2) + **P**(Overcast)*__E__(4,0) + **P**(Rainy)*__E__(2,3)

= (5/14)*0.971 + (4/14)*0.0 + (5/14)*0.971

= 0.693

Attribute IG

- The Information Gain is the decrease in entropy obtained splitting the dataset using the attribute.
- We obtain the IG subtracting the attribute's entropy from the dataset entropy:

$$Gain(Classes, A) = Entropy(Classes) - Entropy(Classes, A)$$

G(PlayGolf, Outlook) = **E**(PlayGolf) – **E**(PlayGolf, Outlook)

= 0.940 – 0.693 = 0.247

Best attribute

- The attribute with the highest information gain is chosen as the best attribute
- It is used as a decision node in the ID3 algorithm

Best attribute is Outlook

		Play Golf	
		Yes	No
Outlook	Sunny	3	2
	Overcast	4	0
	Rainy	2	3
Gain = 0.247			

		Play Golf	
		Yes	No
Temp.	Hot	2	2
	Mild	4	2
	Cool	3	1
Gain = 0.029			

		Play Golf	
		Yes	No
Humidity	High	3	4
	Normal	6	1
Gain = 0.152			

		Play Golf	
		Yes	No
Windy	False	6	2
	True	3	3
Gain = 0.048			

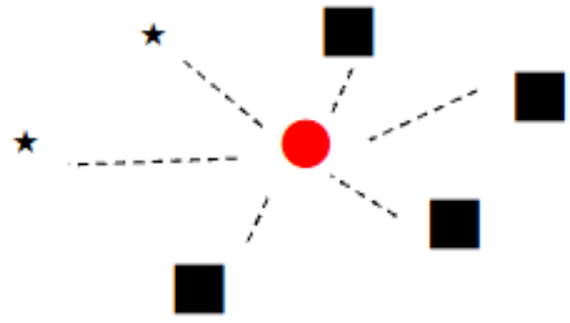
Decision Trees wrap up

- Decision trees are trees of conditions over the dataset attributes.
- Given a new example, we follow the branches down to the leaf using the example's values. The leaf gives us the predicted class.
- We have seen how to build a simple decision tree for categorical attributes using the ID3 algorithm.
- We presented entropy and information gain in order to select the best attribute to split the dataset in a decision tree algorithm.
- Decision trees work very well with discrete and categorical data, where classes are separated in fixed rectangular regions.
- When classes spread heterogeneously across the space in unusually shaped regions, we may want to consider a distance notion instead.

Machine Learning methods
based on **distances**

k-NN

- k-Nearest Neighbors (k-NN in short) is a *lazy* method (it does not learning a model from data):
  - The algorithm **doesn't actually learn** anything
  - It just **stores in memory** all the (data) examples and run a **similarity search**
- Despite its "laziness" it is quite powerful method as we will see.

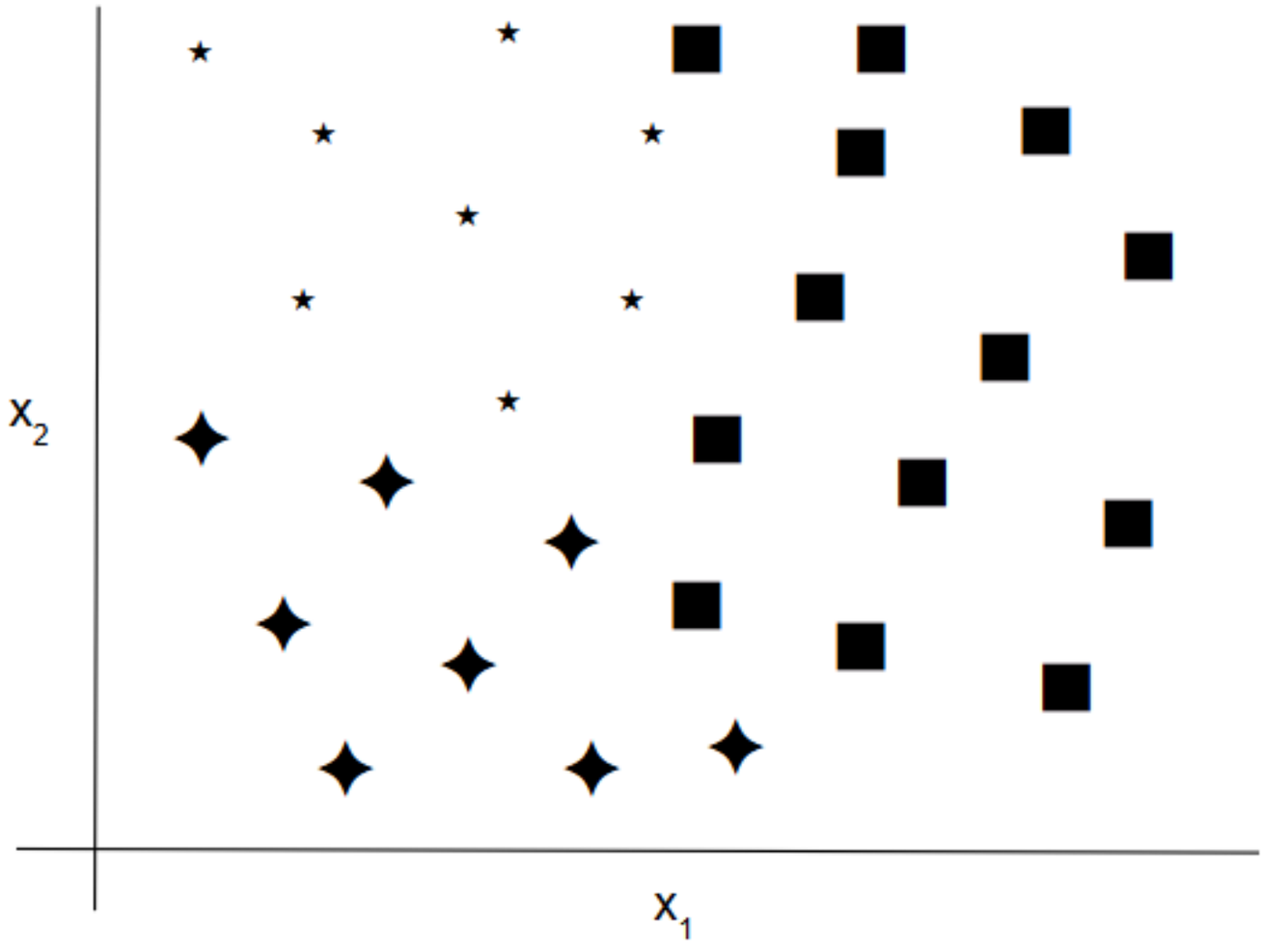


k-NN idea

- As always, each example (e.g. a document) is represented as a vector of *n* dimension
  - A training example has therefore a position in the vector space depending on the values in its elements.
- When a new, unseen example arrives:
  - Search for the k nearest (training set) examples in the vector space
  - Look at their class
  - Assign to the new example the most frequent class among the k documents

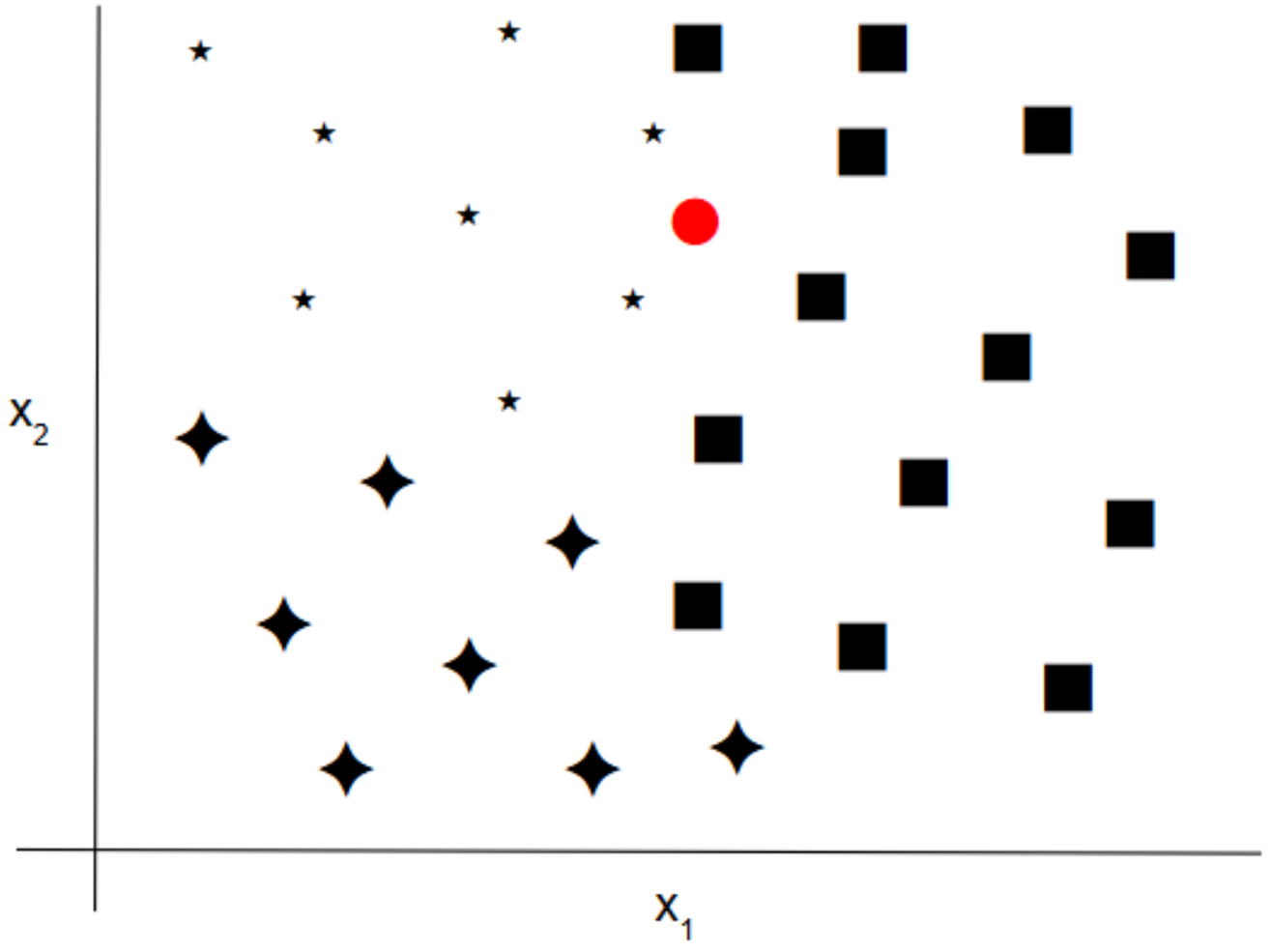
k-NN: 2-D example 1/3

- Suppose for simplicity that the vector space has only two dimensions and 3 classes: ⋆ ✦ ■



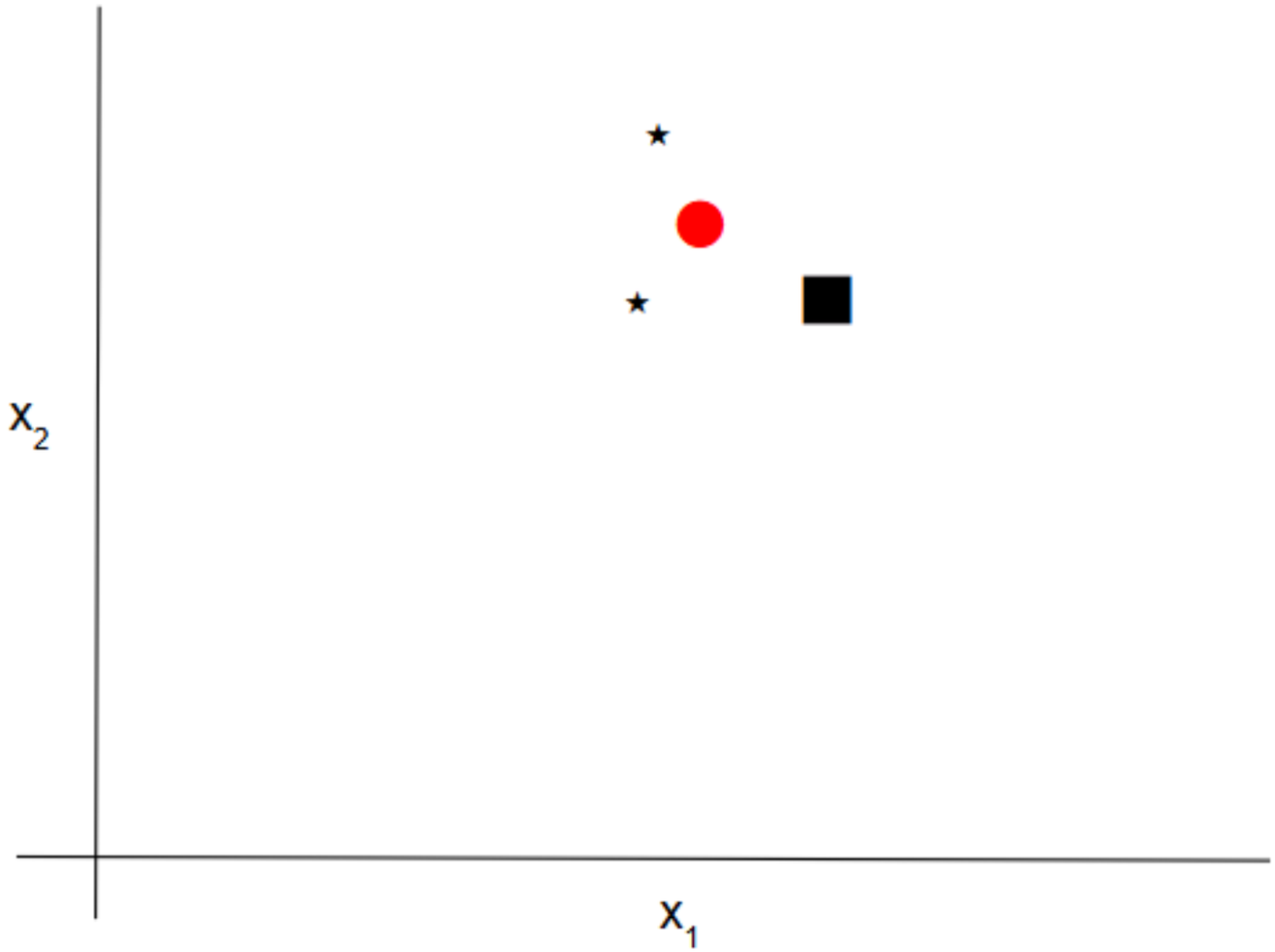
k-NN: 2-D example 2/3

- Now a new unseen example arrives that we want to classify under one of the 3 categories: ●



k-NN: 2-D example 3/3

- With k=3, we consider only the 3 nearests examples
- Most occurrent class is ⋆
- Changing the hyperparameter k the result may change



k-NN: effectiveness and overfitting

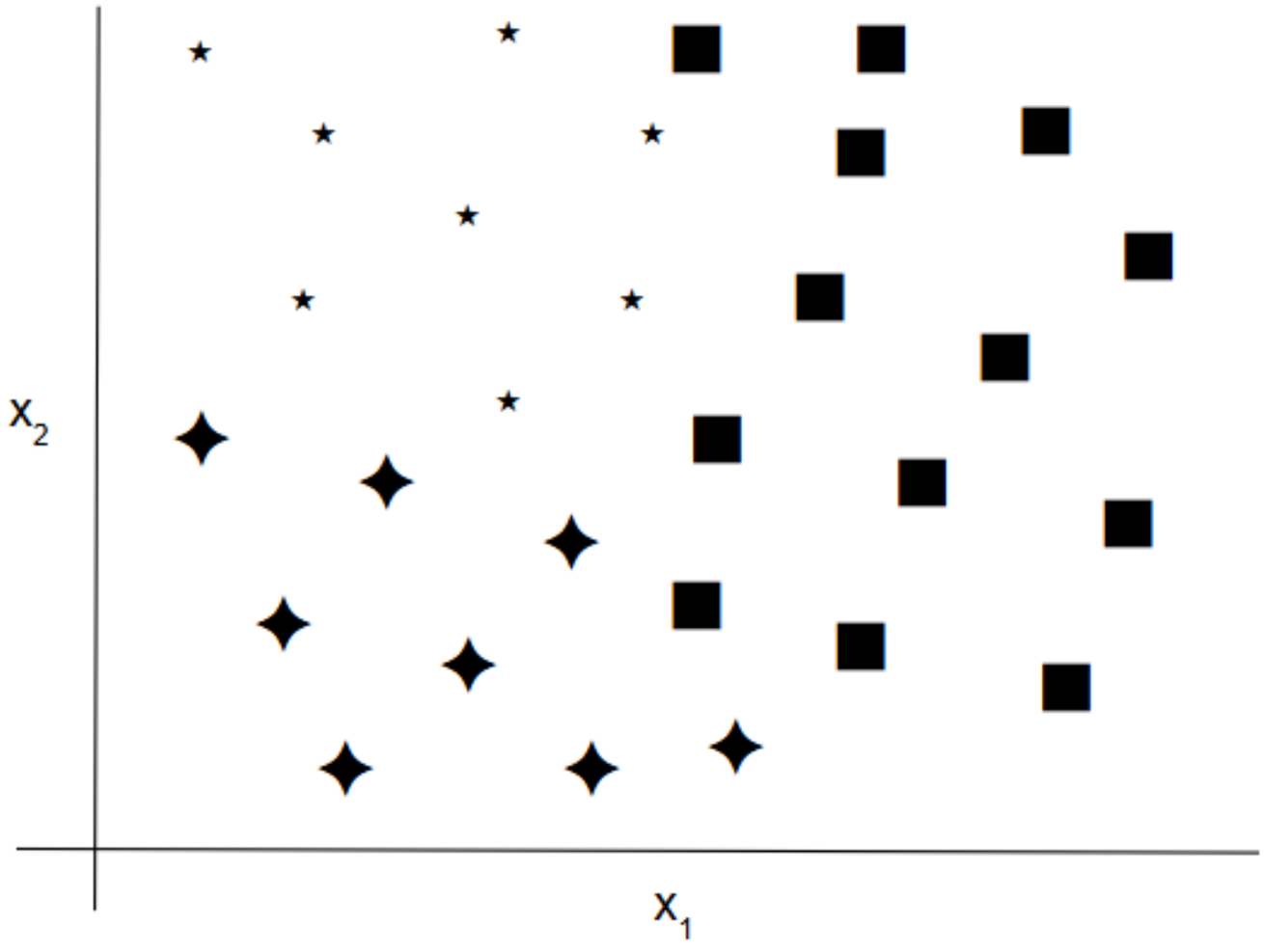
- k-NN does not use any straight boundary lines to decide the class of previous example, instead it employs complex decision boundaries depending on the classes of examples around a point.
- Because of this it is able to learn **both linear and non-linear** functions, **accurately fitting the training set**.
- The fitting of the training set is evaluated by using the **training for testing**, where k-NN can achieve 100% of accuracy:
  - Remind that it stores all the training example with the related classes
  - When k=1, the nearest neighbor of a training sample is just itself, so the answer is always right.
- Using greater k (from 3 to 30) is however more advisable to avoid overfitting:
  - that is, **perfectly fitting** the training set but not being able to **generalize** on unseen examples

Centroid classifier

- Similarly to k-NN, in the centroid classifier we look at distances in the vector space
- The difference is that:
  - we don't consider each single training example
  - instead we group the training examples from the same class in a *centroid*
- Given a new unseen example, the nearest centroid's class is taken as the predicted class.

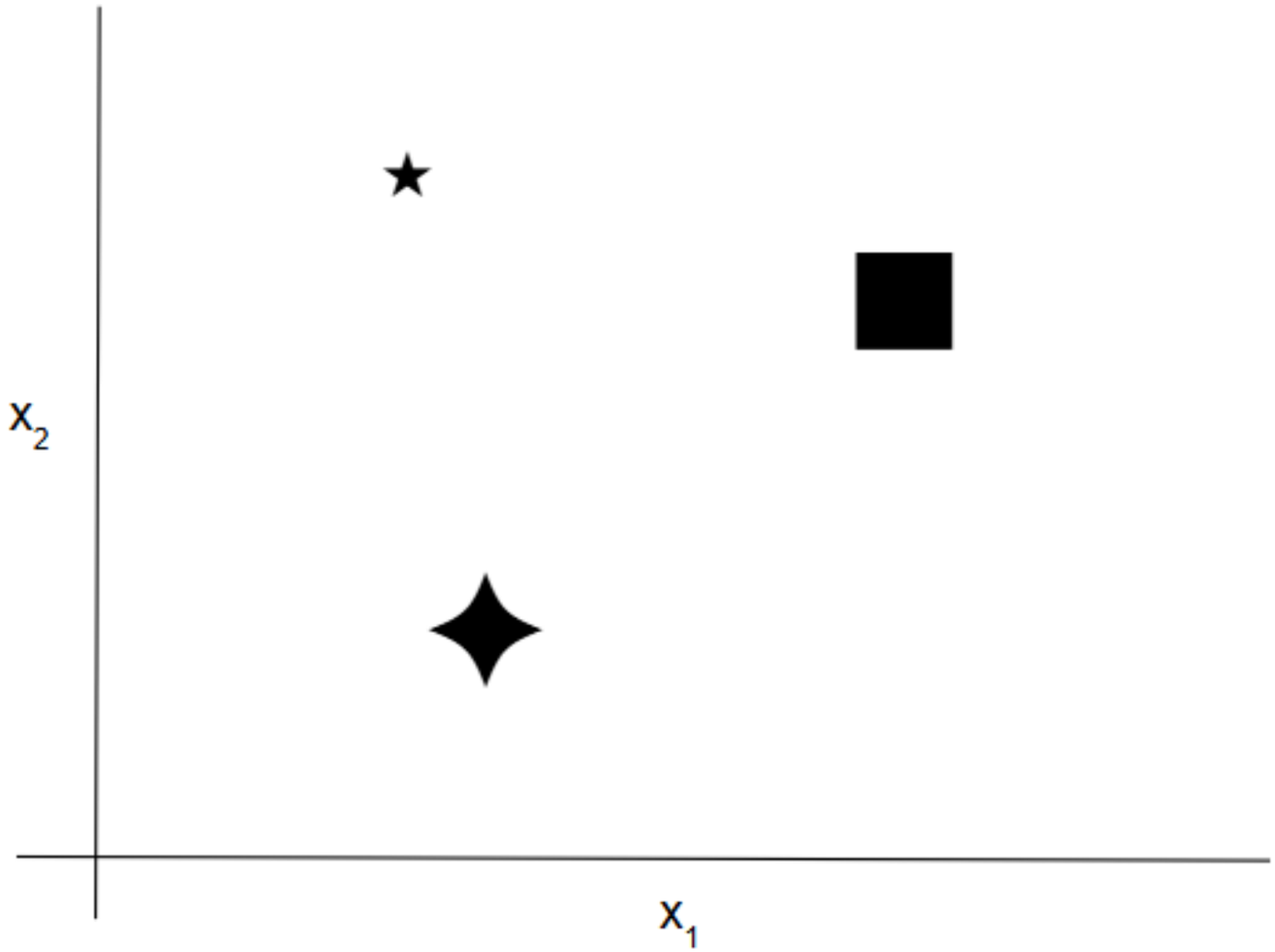
Centroid: 2-D example 1/3

- Suppose for simplicity that the vector space has only two dimensions and 3 classes: ⋆ ✦ ■



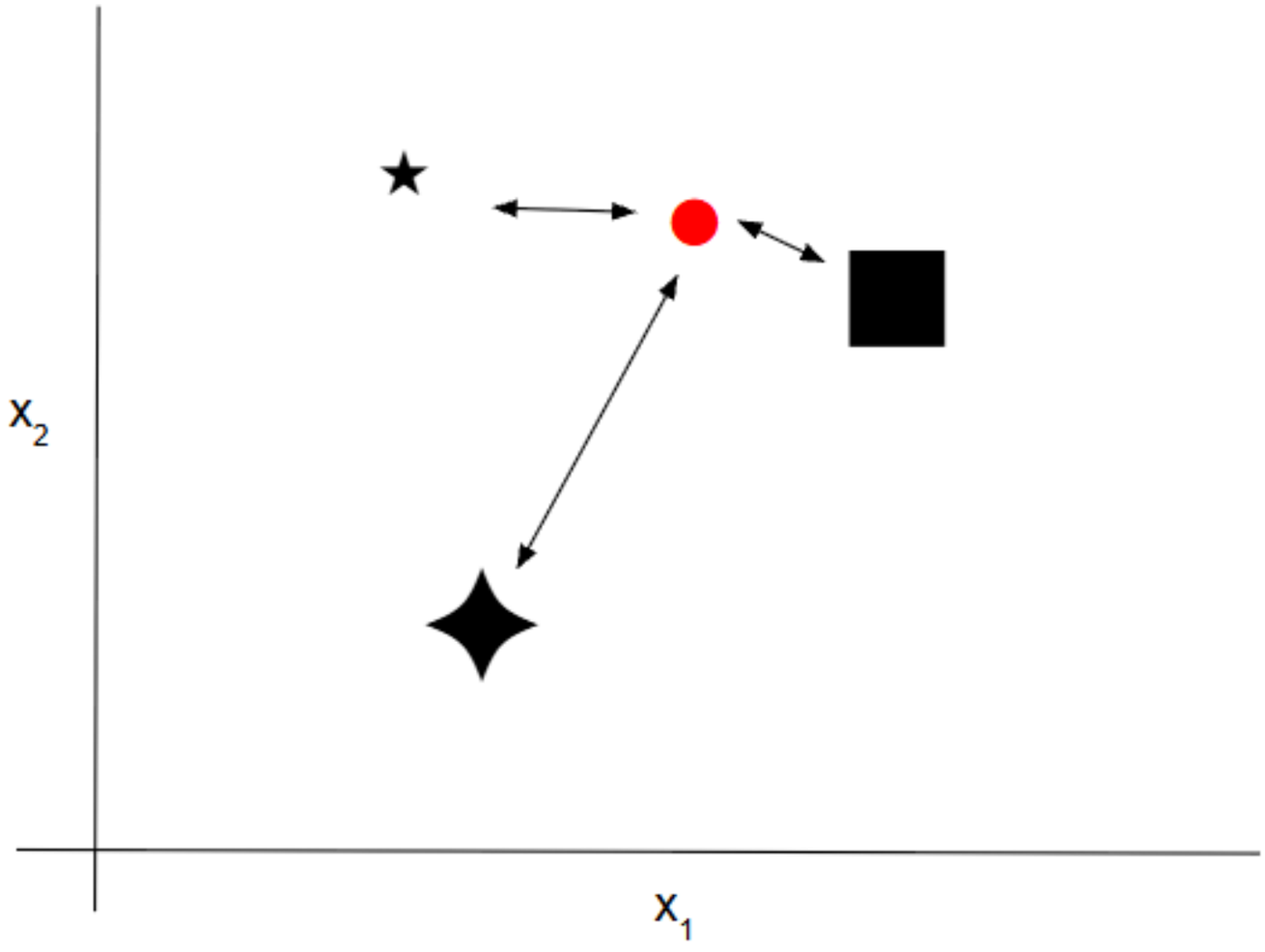
Centroid: 2-D example 2/3

- By simply averaging each coordinate we produce one centroid for each class



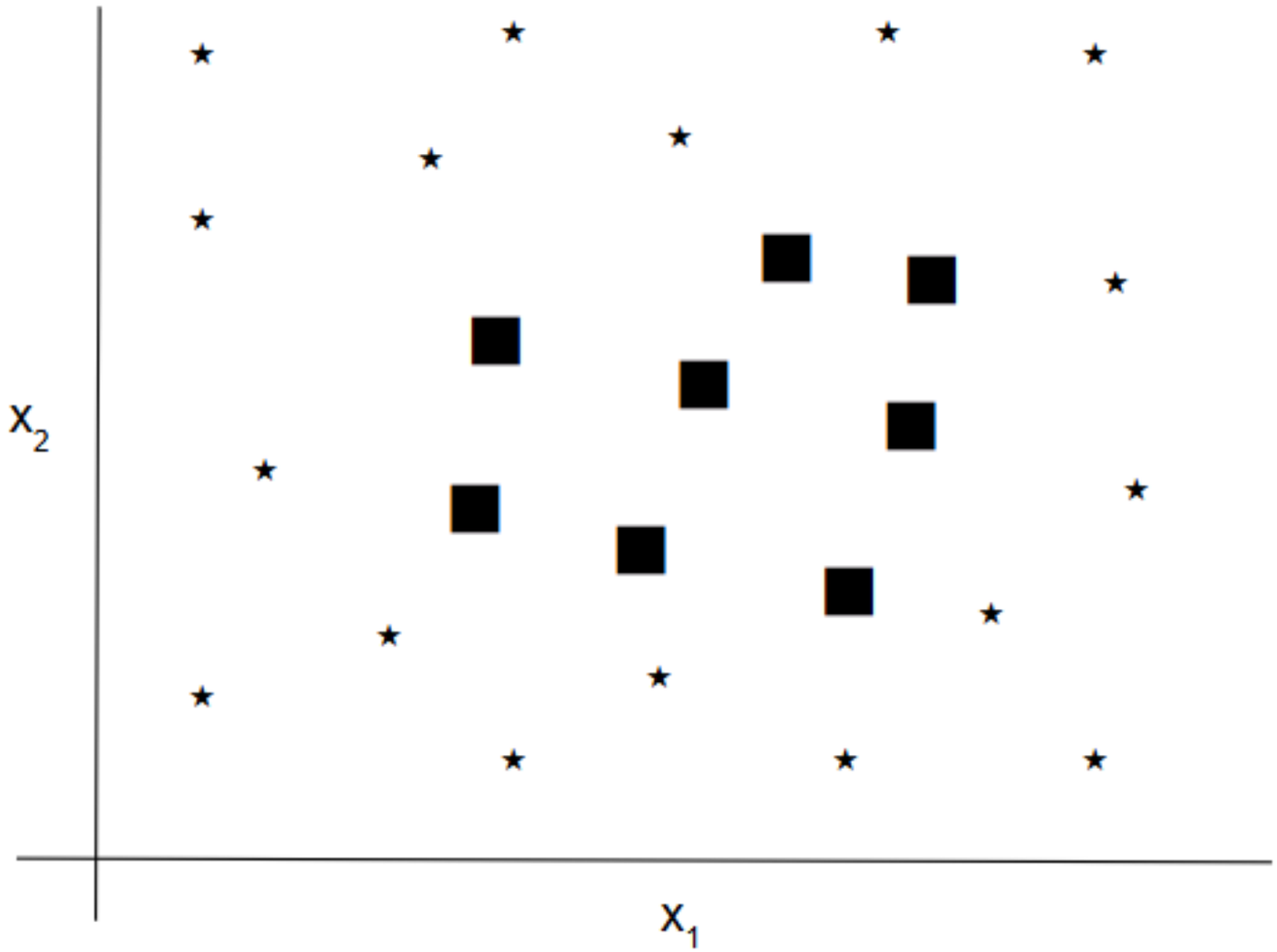
Centroid: 2-D example 3/3

- Given a new unseen example, we compute the distance to each centroid, the nearest centroid is ■
- Distance can be chosen (euclidean, manhattan…)



Non linear data

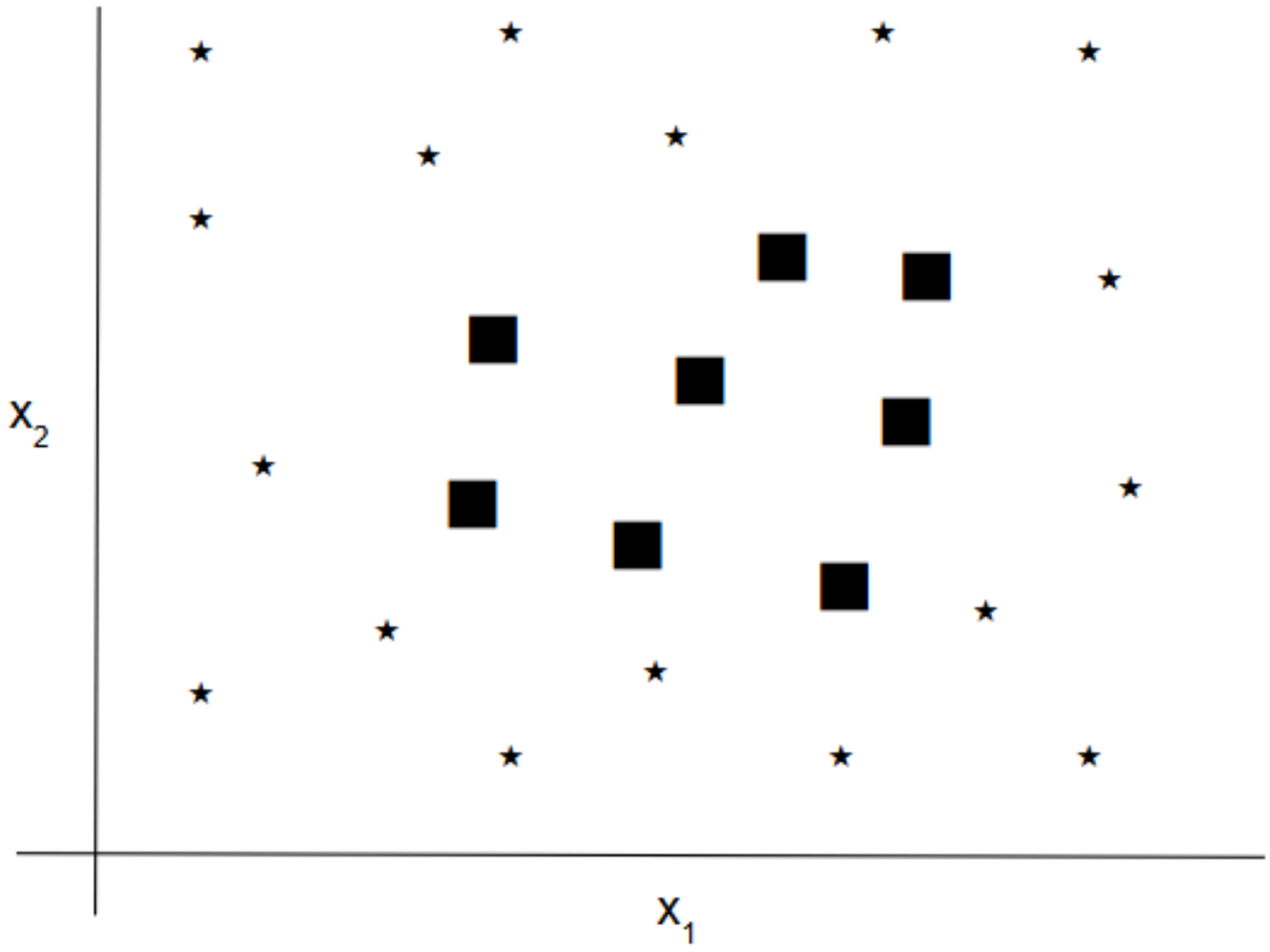
- Would a centroid classifier work for this dataset?



- No: both stars and squares have their centroid in the center, making the model outputting one or the other randomly

Non linear data

- Would a k-NN work for this dataset?



- Yes: take 3-NN as an example, most points will be accurately predicted as squares if in the center and as stars if in the external position

k-NN and Centroid classifiers

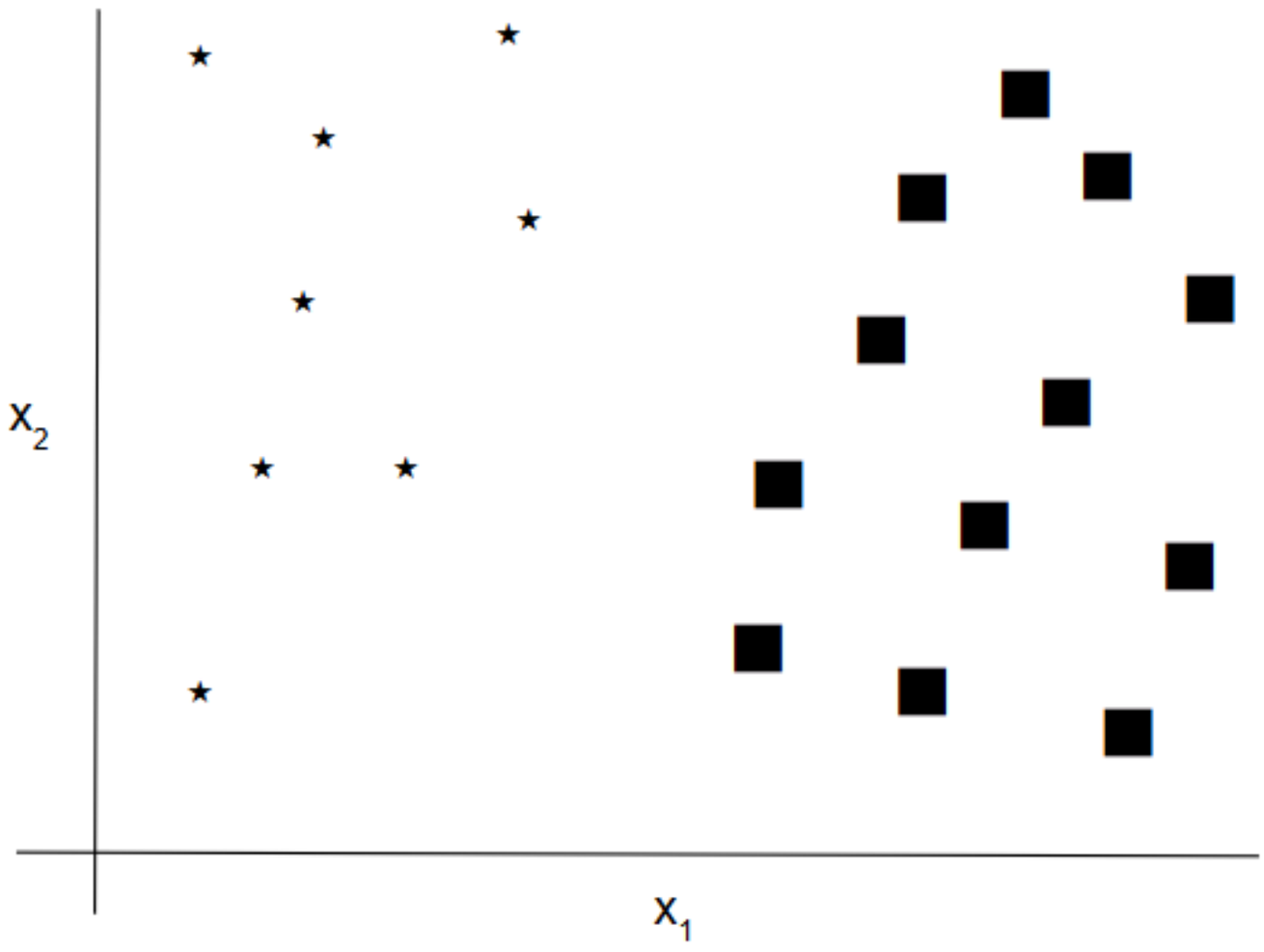
- k-NN and Centroid are based on metric distances between examples in the vector spaces.
- k-NN:
  - Keep in memory all the training examples.
  - Output the most frequent class around the unseen example, considering the k nearest neighbors in the training examples.
- Centroid classifier:
  - Keep in memory, for each class, the centroid of its training examples
  - Output the class of the centroid closest to the unseen example.
- We will now see a very accurate model, which combines ideas from logistic regression, distance models and the notion of *large margin* for boundary lines.

A better solution: SVM

- Support Vector Machine (SVM) can solve the problem of non-linear separation by using the “kernel trick”
  - SVM is actually a linear classifier
  - It can achieve non-linear classification by transforming the features into distances from “landmarks”, as we will see
  - Without the kernel trick, SVM is similar to logistic regression with the except that SVM is a “**large margin classifier**”
    - The SVM decision boundary is better at **generalization** (working good on unseen examples)
- Let’s see some examples for decision boundaries

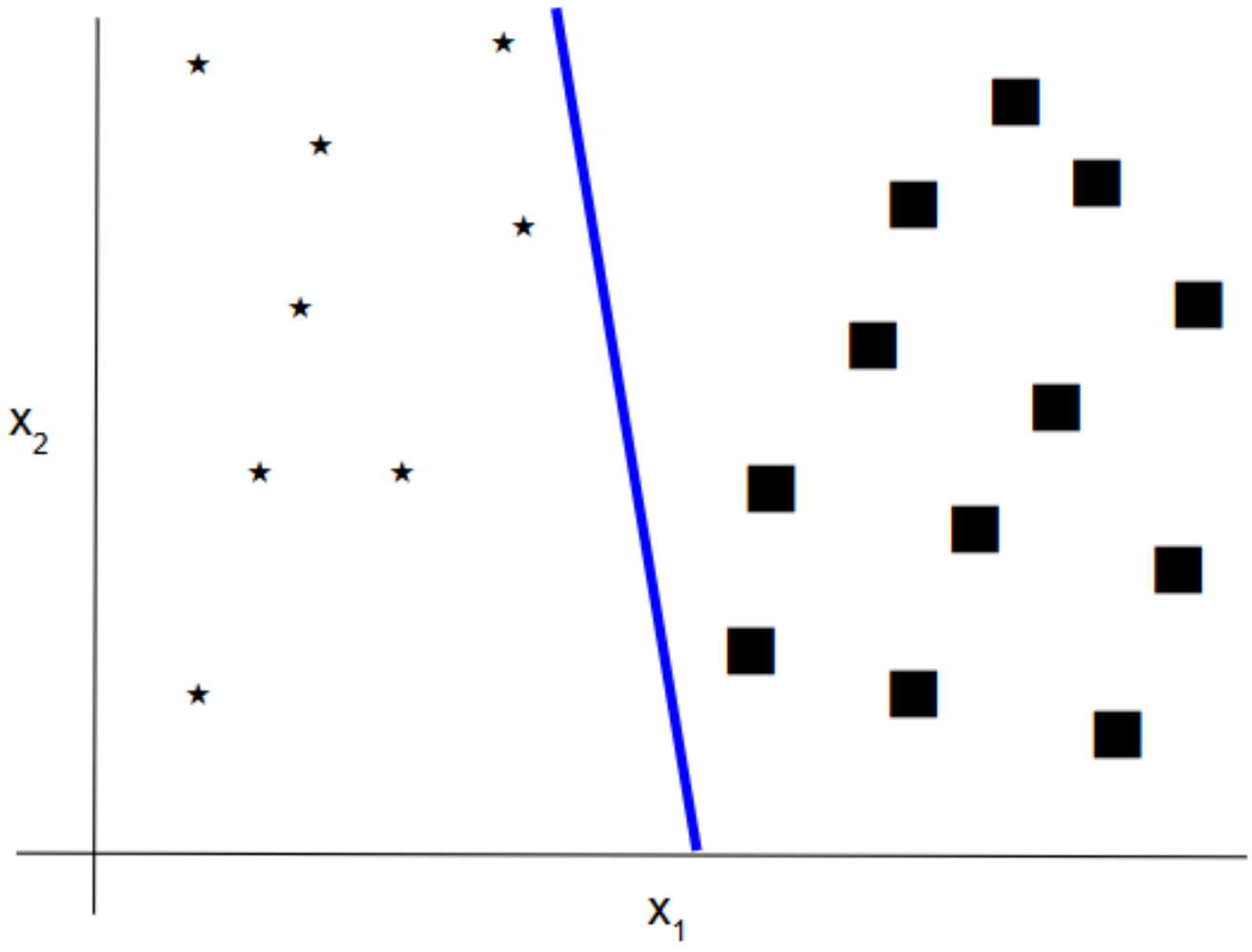
Example: linearly separable

- The following data can be linearly separated
- What would be the best decision boundary that separates the two classes?



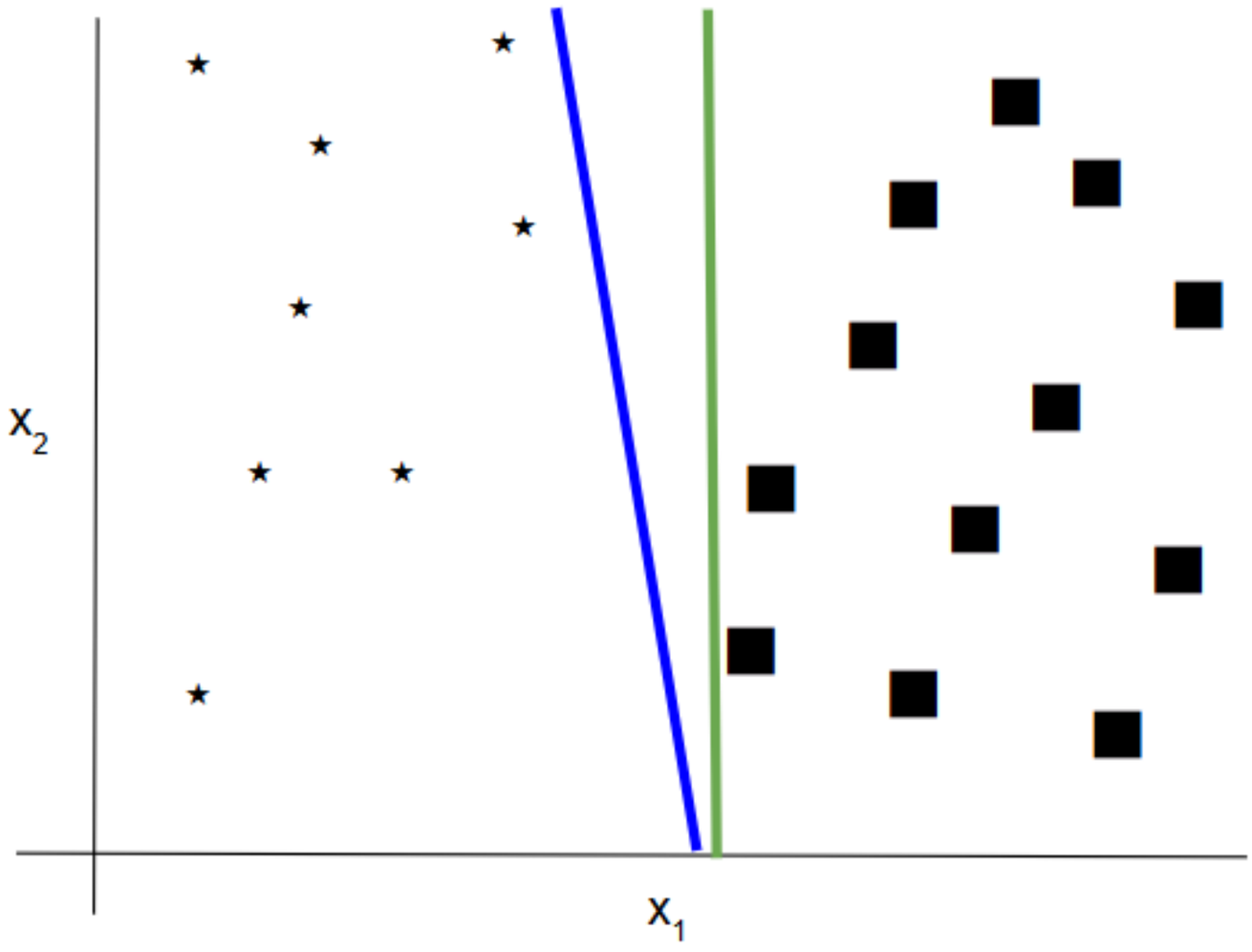
Example: decision boundary 1

- From a training set point of view, **the blue line** perfectly fits the data
- It may be a decision boundary learned using Logistic Regression



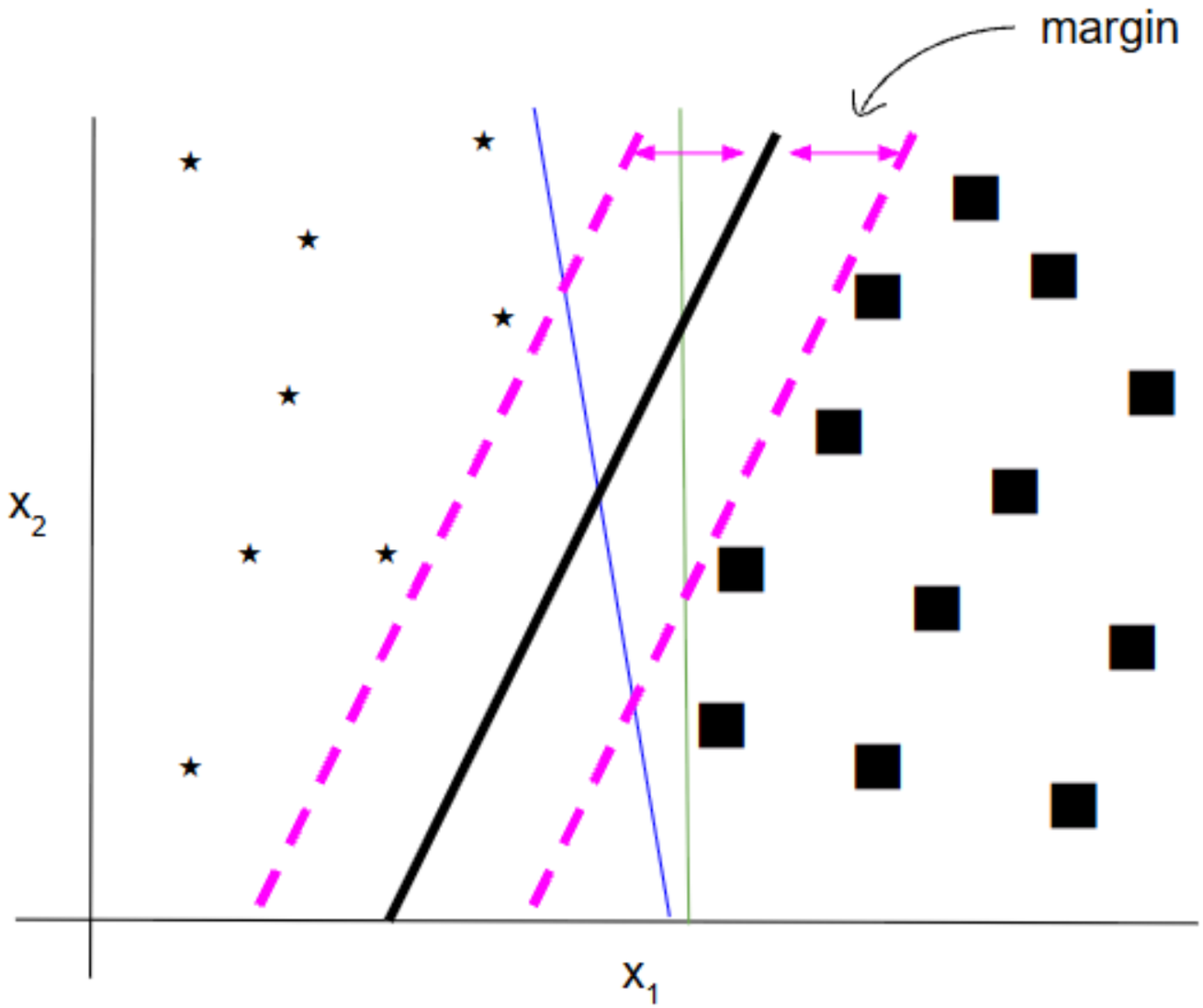
Example: decision boundary 2

- Also the **green line** can be a good decision boundary!
- As the blue line, it “minimizes the cost” in a logistic regression function, because it perfectly fit the training set



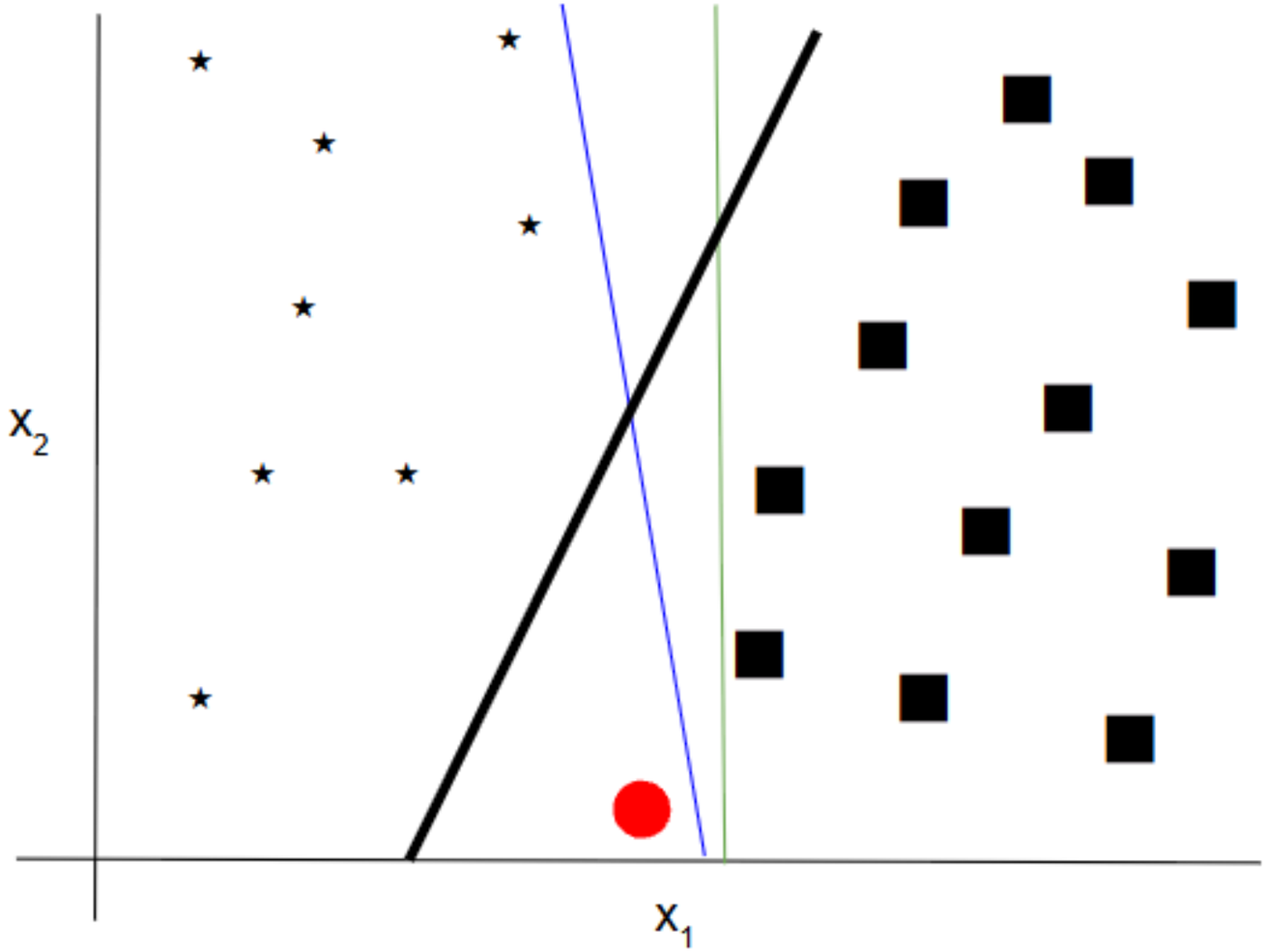
Example: decision boundary SVM

- Purple line is the decision boundary learned using SVM
- It has the largest margin from the two classes



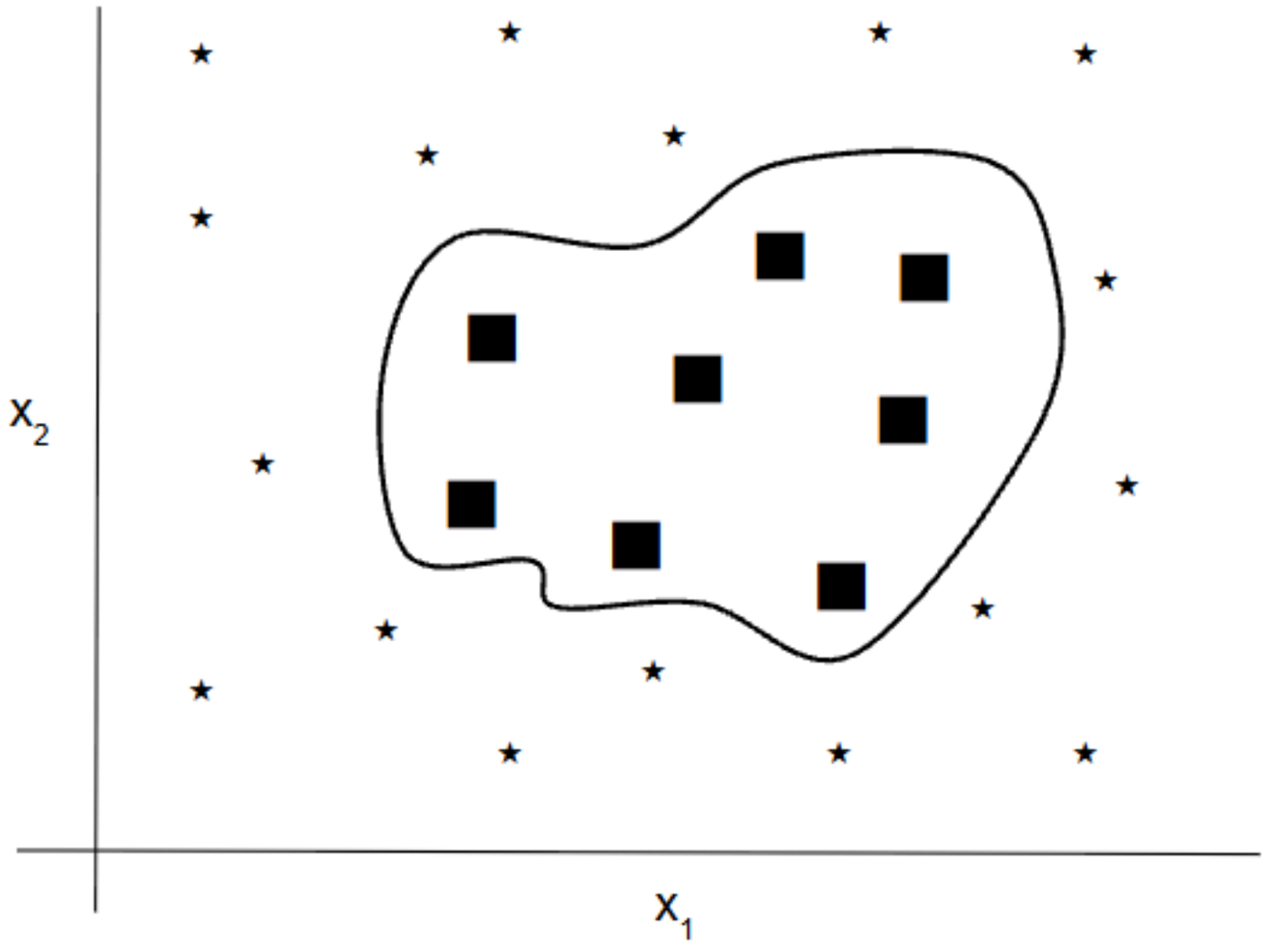
Generalization

- While all three decision boundaries are **correct** for training set (accuracy 100%) they behave differently with new **unseen examples** (in red)
- The intuition is that given a new unseen example (in red) the large margin classifier (black bold) is better, because it follows the shape of the class' subset.
- In other words, it has a **better generalization** on unseen data



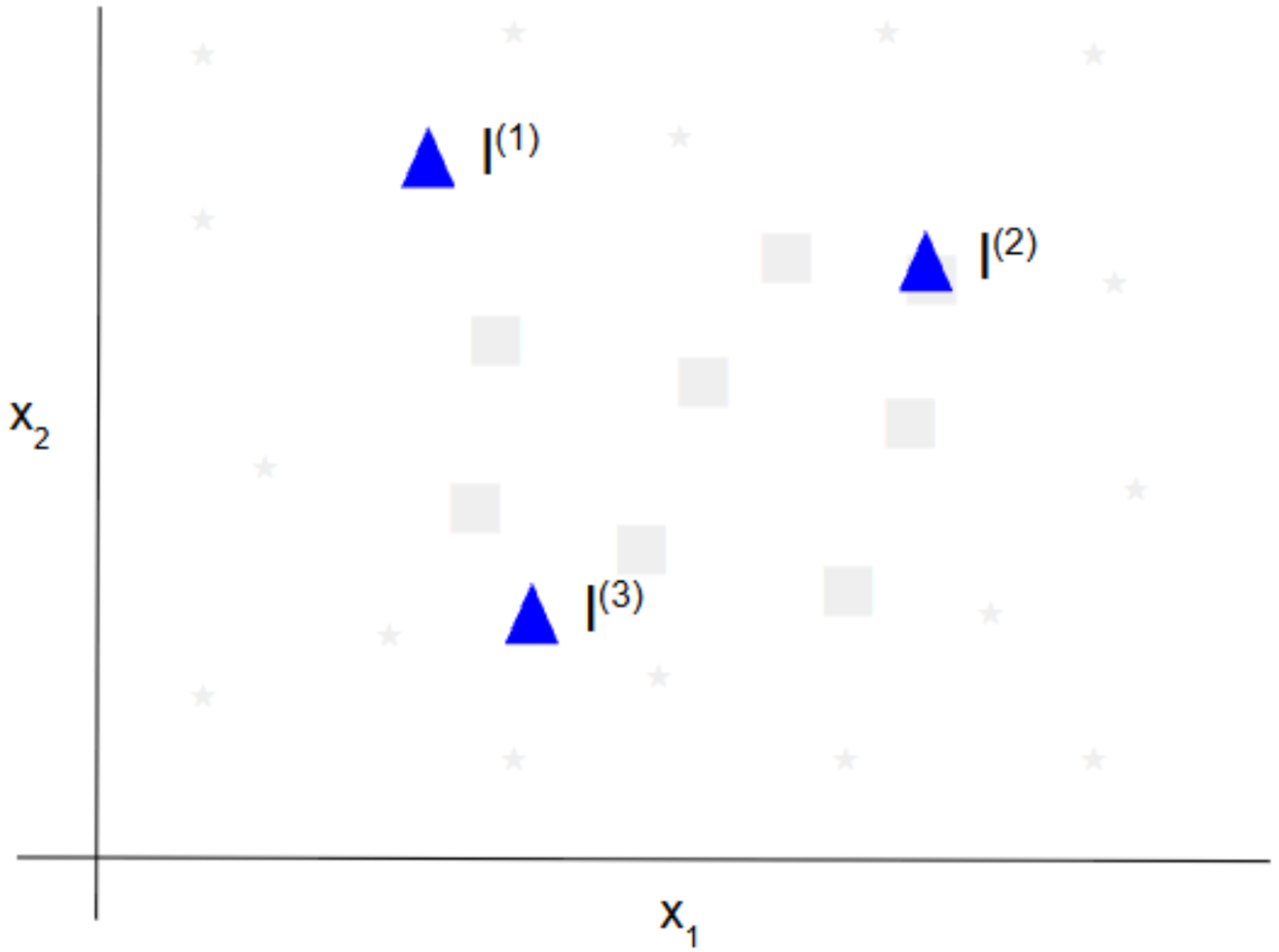
Non-linear functions

- Despite SVM is a linear classifier, it is often used to learn complex non-linear decision boundaries.
- SVM achieve non-linearity by using a technique called kernel trick



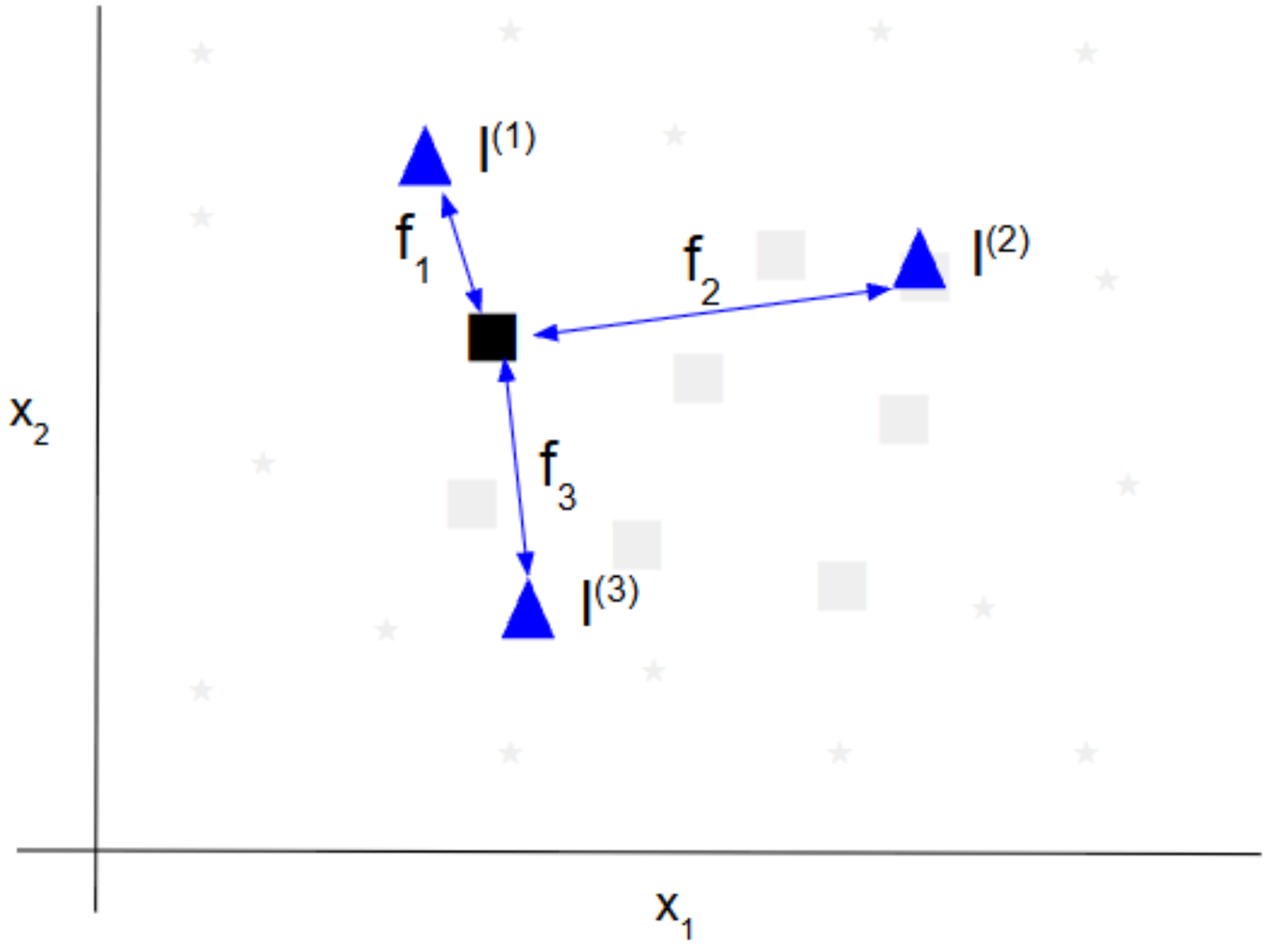
Kernels landmarks

- The kernel trick starts by choosing some “landmarks”
- Landmarks are points in the vector space of the examples
- Example: $l^{(1)}, l^{(2)}, l^{(3)}$ are 3 landmarks in our vector space with 3 examples



Distances from landmarks

1. Given an example, we compute the distances between the example point and the landmarks
2. We transform the distances into similarities
3. The 3 similarities f_1, f_2, f_3 will be our new features instead of x_1 and x_2

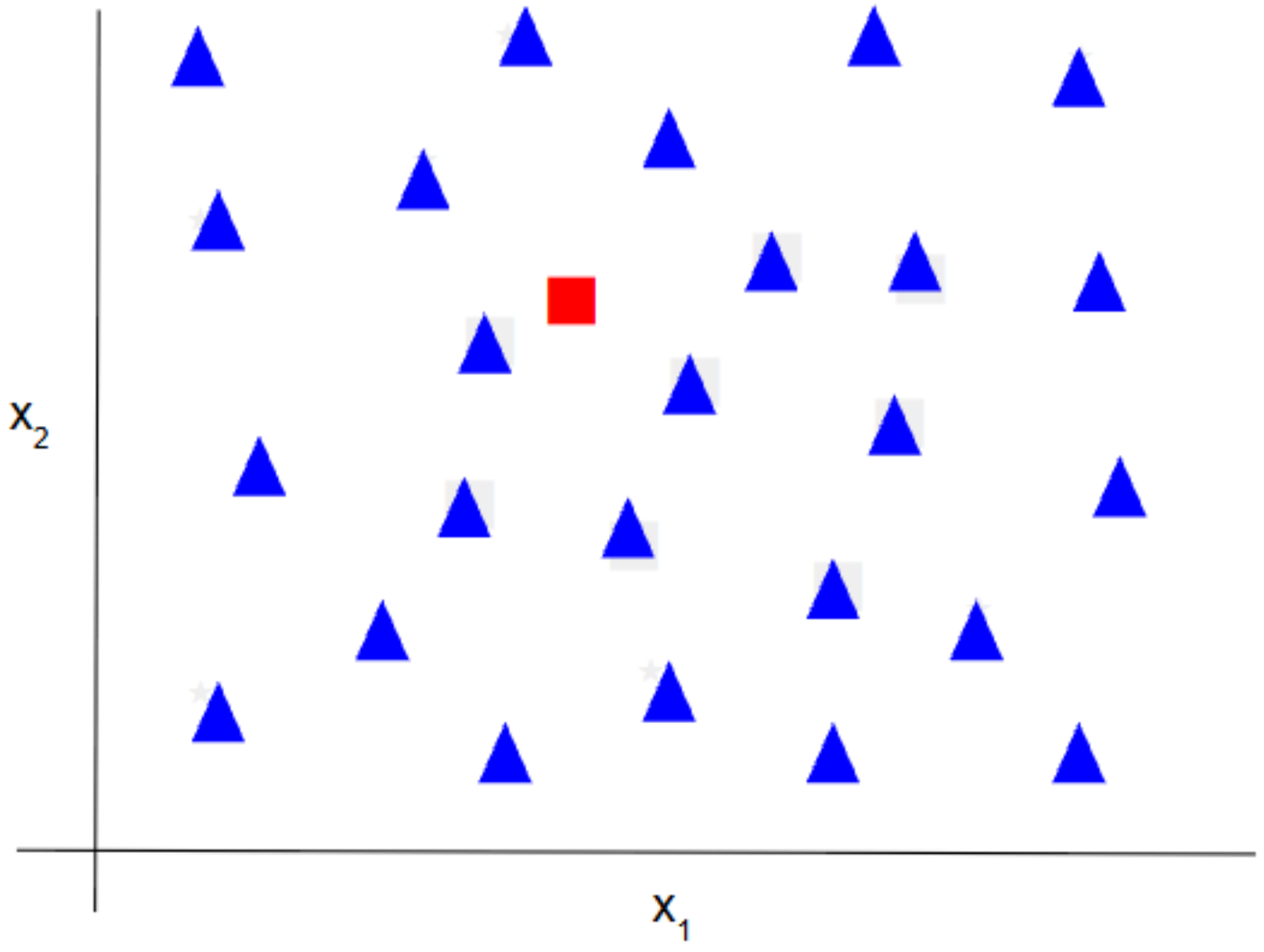


A new (support) vector space

- If we are using 3 landmarks, for each example we can compute the similarity between the vector of the example and the vector of each landmark
  - This results in 3 new features for each document: f_1, f_2, f_3
- Before we represented an example x using an element x_i for each feature i, assuming we have *n* features:
  - $d = (x_1, x_2, ..., x_{n-1}, x_n)$
- Now, we will represent a document d using the distances from the landmarks:
  - $d = (f_1, f_2, f_3)$
- This process is called **features mapping**.

Choosing the landmarks

- How do we choose the landmarks?
- Good way: use the training examples as landmarks!



Note: the red example will be represented as a 25-dimension vector, one distance for each landmark, we can't represent graphically

SVM: Implications

- Every example is described as its distance from any other example
- Each feature is the similarity from the example and another particular example
- The decision boundary is non-linear with respect to the original features (i.e. x_1,x_2,...)
- SVM achieve this goal by looking at the distance from each example in the training set, similarly to k-NN, but…
  - each distance is “weighted” in our hypothesis function (that is a linear combination) to minimize the error
  - No k parameter
  - an hypothesis function is learned that is then applied a new input. (is not lazy)

References

Data Mining

Concepts and Techniques

Authors: Jiawei Han, Micheline Kamber, Jian Pei.

Sections:

- 3.4.4. Attribute Subset Selection
- 9.5.1 k-Nearest Neighbor Classifiers
- 9.3 Support Vector Machines