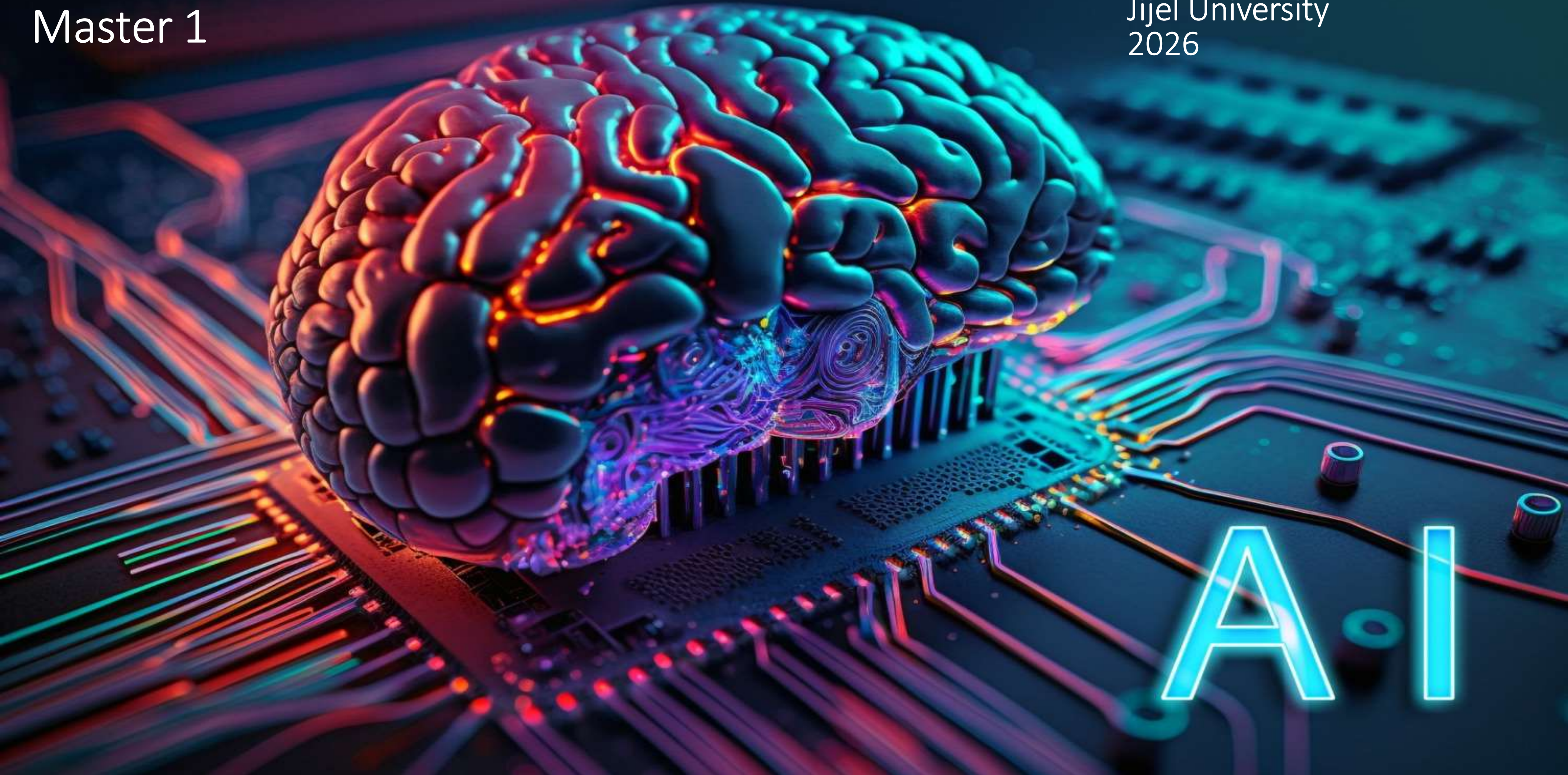
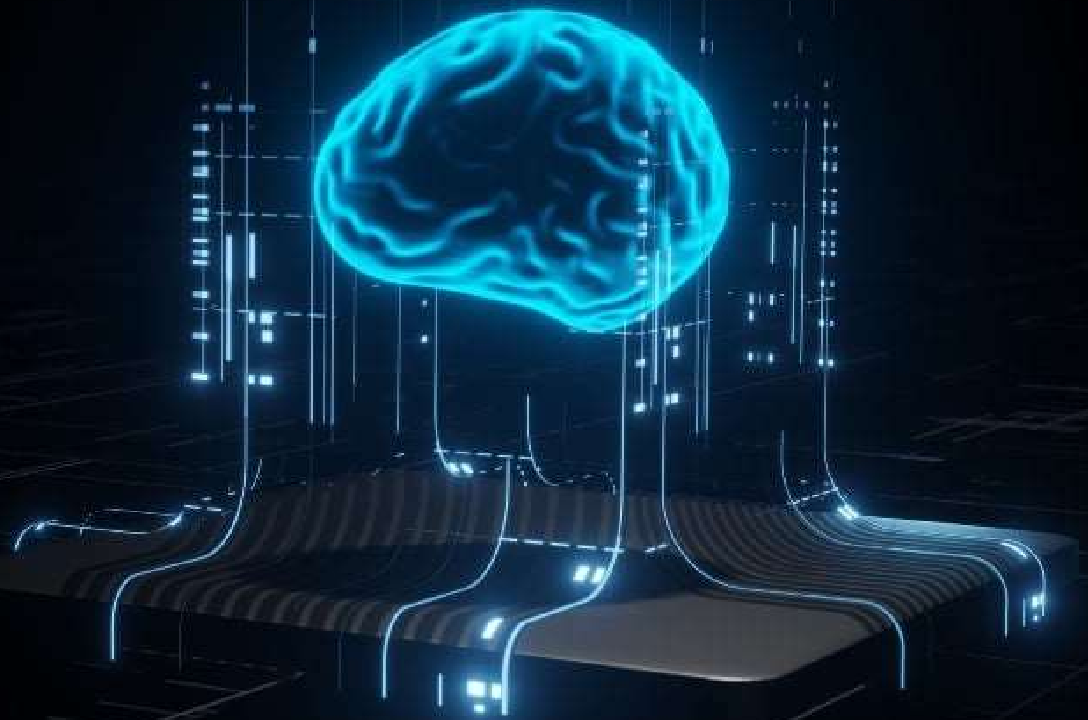
Applied Artificial intelligence

Master 1

By Dr. Nafa Fares
Department of automation
Jijel University
2026



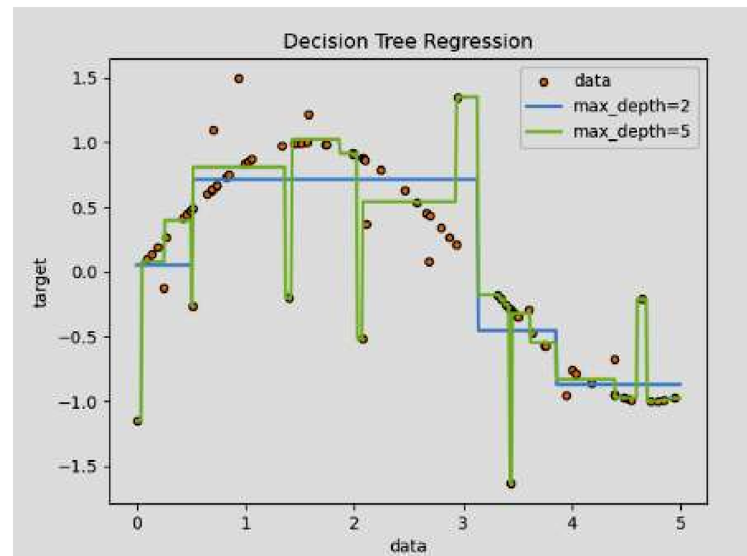
Chapter 4- Machine learning



1. Supervised classification

Decision Trees

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.
For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.



1. Supervised classification

Advantages of decision trees are

- Simple to understand and to interpret. Trees can be visualized.
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Some tree and algorithm combinations support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. However, the scikit-learn implementation does not support categorical variables for now. Other techniques are usually specialized in analyzing datasets that have only one type of variable. See algorithms for more information.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

1. Supervised classification

Disadvantages of decision trees include

- Decision-tree learners can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximations as seen in the above figure. Therefore, they are not good at extrapolation.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree

1. Supervised classification

Classification

Decision Tree Classifier is a class capable of performing multi-class classification on a dataset.
As with other classifiers, **DecisionTreeClassifier** takes as input two arrays: an array X, sparse or dense, of shape (n_samples, n_features) holding the training samples, and an array Y of integer values, shape (n_samples,), holding the class labels for the training samples:

```
>>> from sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, Y)
```

After being fitted, the model can then be used to predict the class of samples:

```
>>> clf.predict([[2., 2.]])
array([1])
```

In case that there are multiple classes with the same and highest probability, the classifier will predict the class with the lowest index amongst those classes.
As an alternative to outputting a specific class, the probability of each class can be predicted, which is the fraction of training samples of the class in a leaf:

```
>>> clf.predict_proba([[2., 2.]])
array([[0., 1.]])
```

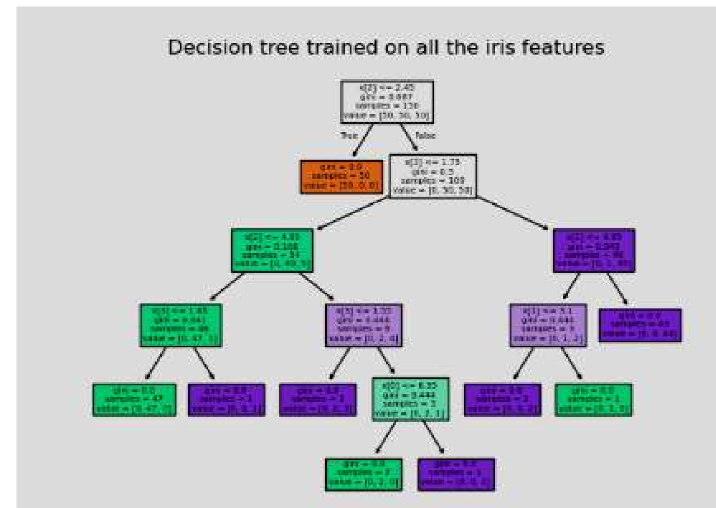
1. Supervised classification

DecisionTreeClassifier is capable of both binary (where the labels are [-1, 1]) classification and multiclass (where the labels are [0, …, K-1]) classification. Using the Iris dataset, we can construct a tree as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, y)
```

Once trained, you can plot the tree with the plot_tree function:

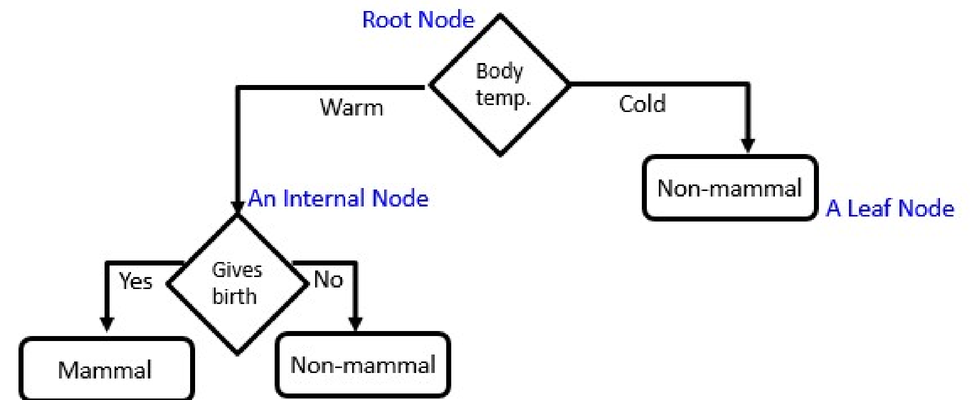
```
>>> tree.plot_tree(clf)
[...]
```



2. Decision Tree

- A Decision Tree (DT) defines a hierarchy

of rules to make a prediction/



- Root and internal nodes test rules. Leaf nodes make pre
- Decision Tree (DT) learning is about learning such a

tree from labeled data

- A decision tree friendly problem : Loan approval prediction

ID	Age	Has_Job	Own_House	Credit_Rating	Class
1	young	false	false	fair	**No**
2	young	false	false	good	**No**
3	young	true	false	good	**Yes**
4	young	true	true	fair	**Yes**
5	young	false	false	fair	**No**
6	middle	false	false	fair	**No**
7	middle	false	false	good	**No**
8	middle	true	true	good	**Yes**
9	middle	false	true	excellent	**Yes**
10	middle	false	true	excellent	**Yes**
11	old	false	true	excellent	**Yes**
12	old	false	true	good	**Yes**
13	old	true	false	good	**Yes**
14	old	true	false	excellent	**Yes**
15	old	false	false	fair	**No**

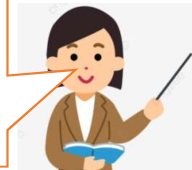
2. Decision Tree: Learning Decision Trees with Supervision

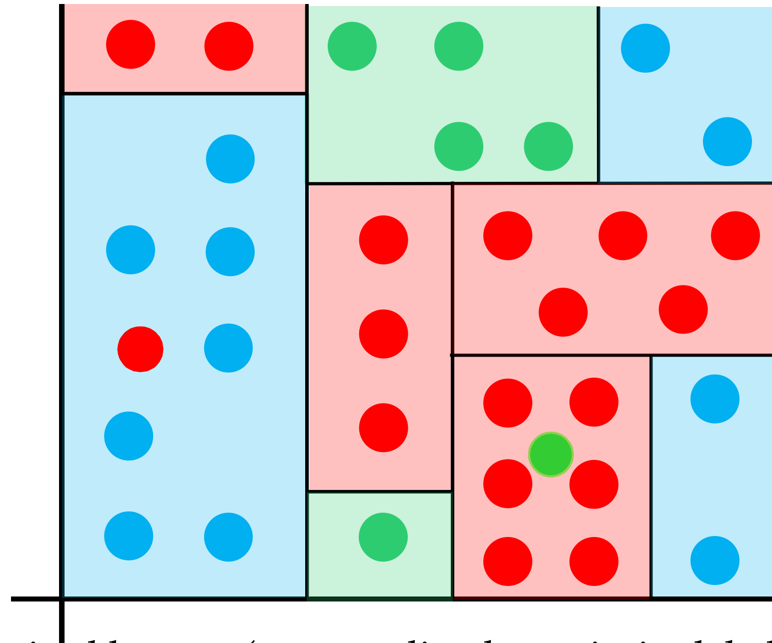
- The basic idea is very simple

- Recursively partition the training data into homogeneous regions

What do you mean by "homogeneous" regions?

A homogeneous region will have all (or a majority of) training inputs with the same/similar outputs

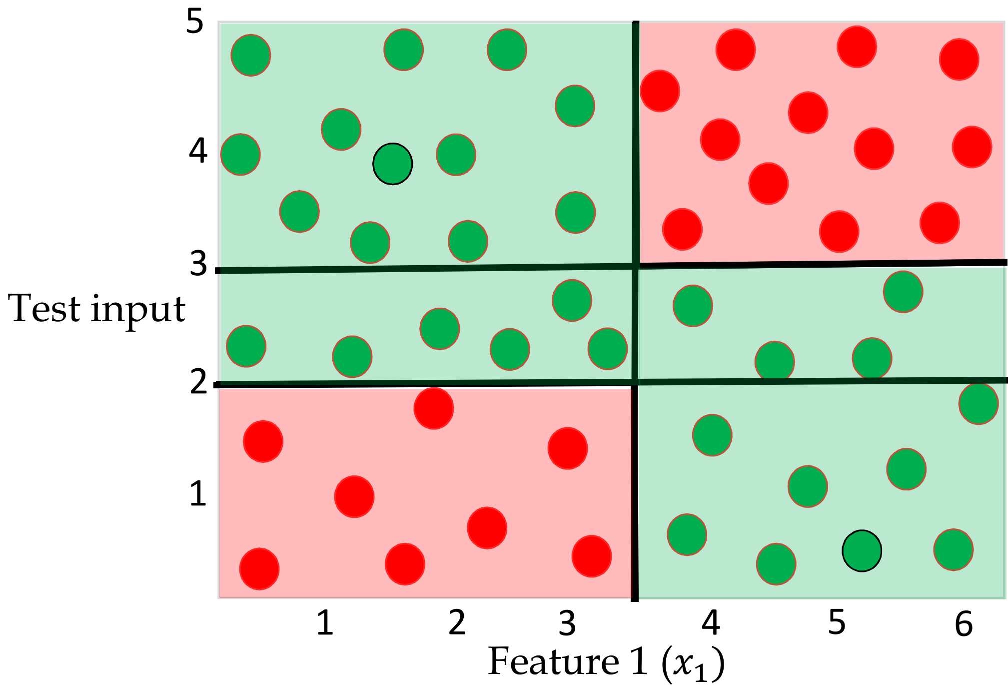


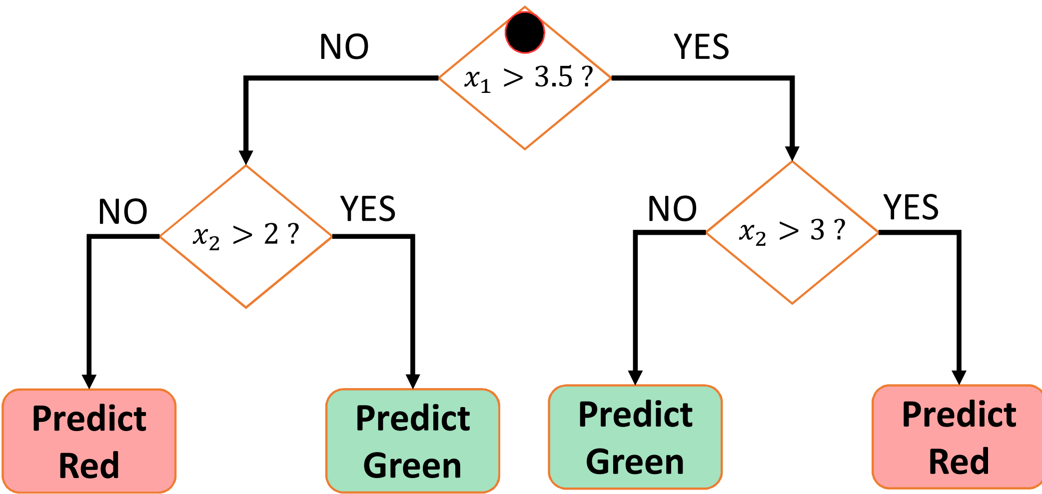


Even though the rule within each group is simple, we are able to learn a fairly sophisticated model overall (note in this example, each rule is a simple horizontal/vertical classifier but the overall decision boundary is rather sophisticated)

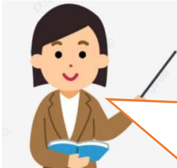
- Within each group, fit a simple supervised learner (e.g., predict the majority label)

2. DecisionTrees for Classification





Remember: Root node contains all training inputs
Each leaf node receives a subset of training inputs



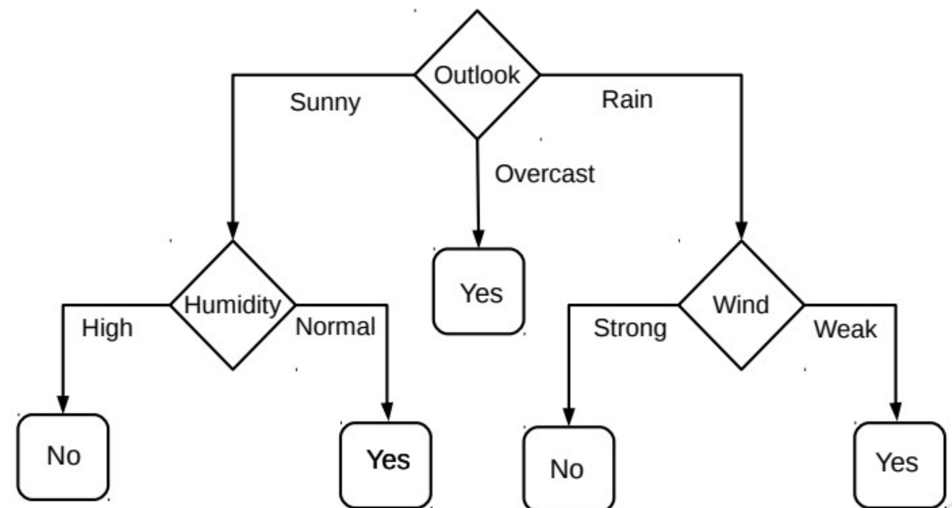
DT is very efficient at test time: To predict the label of a test point, nearest neighbors will require computing distances from 48 training inputs. DT predicts the label by doing just 2 feature-value comparisons! Way faster!

2. DecisionTrees for Classification: another example

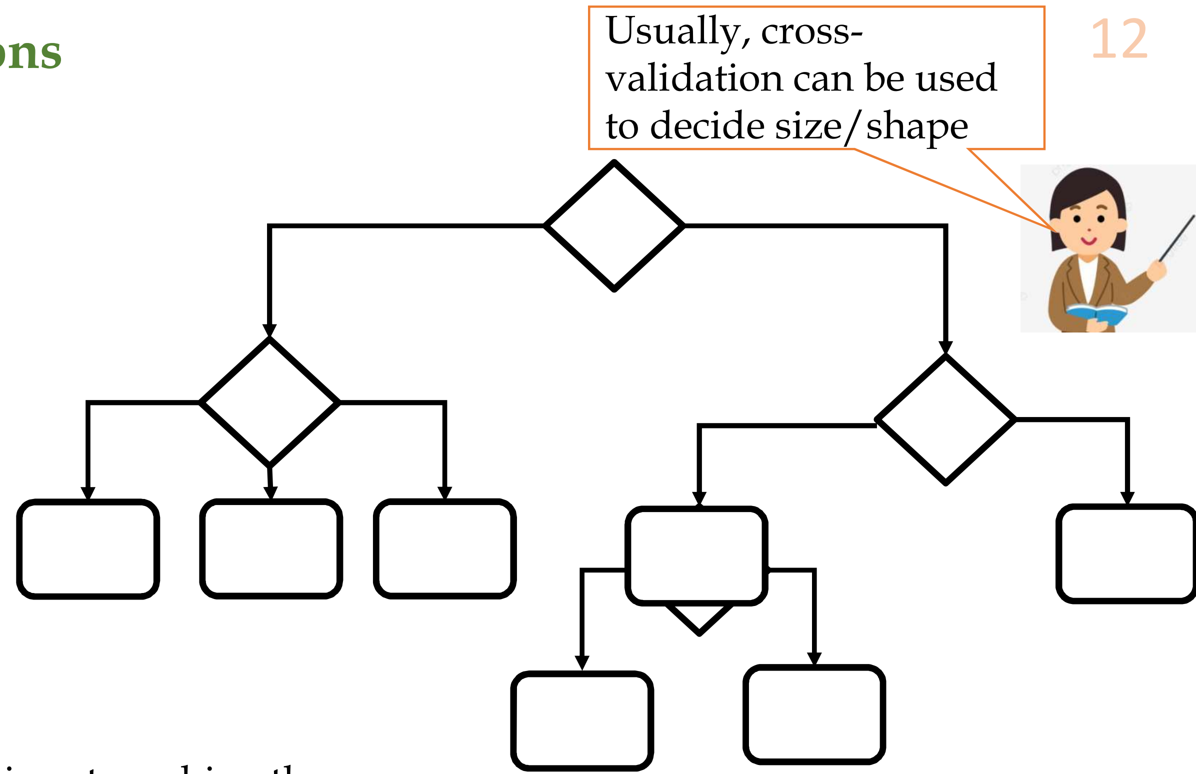
- Deciding whether to play or not to play Tennis on a Saturday
- Each input (Saturday) has 4 categorical features: Outlook, Temp., Humidity, Wind
- A binary classification problem (play vs no-play)
- Below Left: Training data, Below Right: A decision tree constructed using this data

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no

Example credit: Tom Mitchell



4. Decision Trees: Some Considerations



- What should be the size/shape of the DT?
  - Number of internal and leaf nodes
  - Branching factor of internal nodes
  - Depth of the tree

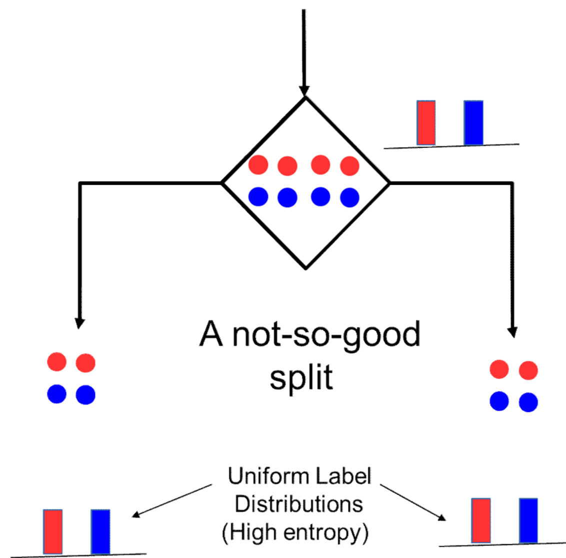
- Split criterion at internal nodes
  - Use another classifier?
  - Or maybe by doing a simpler test?

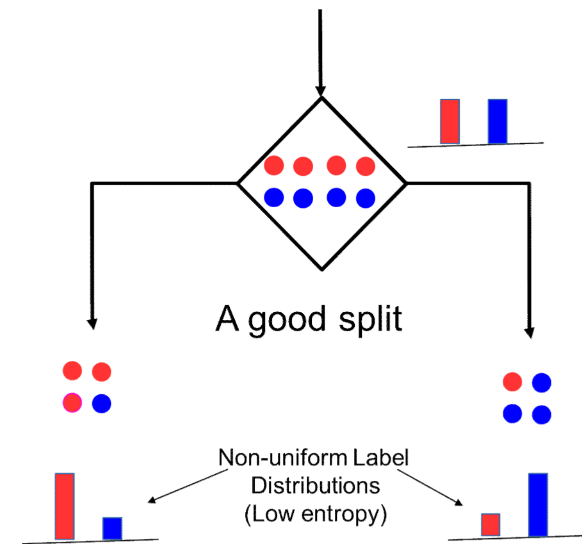
- What to do at the leaf node? Some options:
  - Make a constant prediction for each test input reaching there
  - Use a nearest neighbor based prediction using training inputs at that leaf node
  - Train and predict using some other sophisticated supervised learner on that node

Usually, constant prediction at leaf nodes used since it will be very fast

How to Split at Internal Nodes?

- Recall that each internal node receives a subset of all the training inputs
- Regardless of the criterion, the split should result in as "pure" groups as possible
  - A pure group means that the majority of the inputs have the same label/output



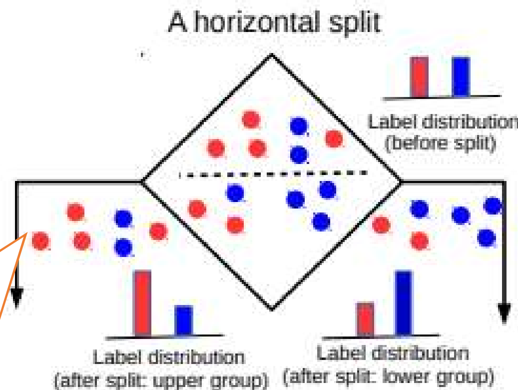


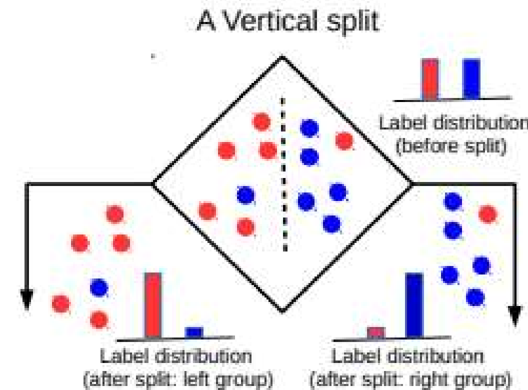
- For classification problems (discrete outputs), entropy is a measure of purity
  - Low entropy $\Rightarrow$ high purity (less uniform label distribution)
  - Splits that give the largest reduction (before split vs after split) in entropy are preferred (this reduction is also known as "information gain")

Techniques to Split at Internal Nodes

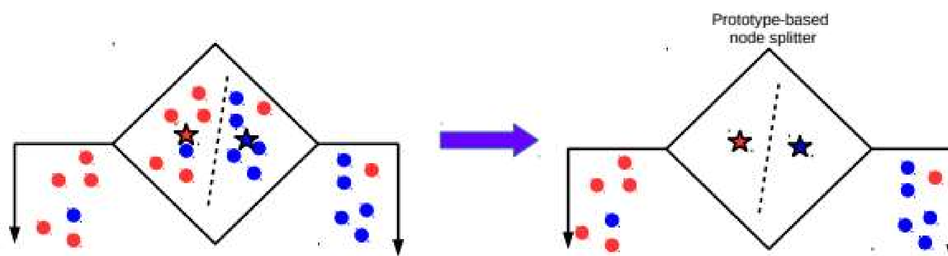
- Each internal node decides which outgoing branch an input should be sent to
- This decision/split can be done using various ways, e.g.,
  - Testing the value of a single feature at a time (such internal node called "Decision Stump")
  - See here for more details
  - Learning a classifier (e.g., LwP or some more sophisticated classifier)

With this approach, all features and all possible values of each feature need to be evaluated in selecting the feature to be tested at each internal node
(can be slow but can be made faster using some tricks)





DT methods based on testing a single feature at each internal node are faster and more popular (e.g., ID3, C4.5 algos)



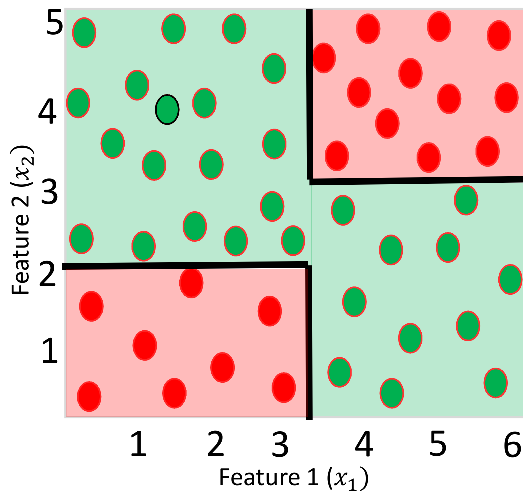
DT methods based on learning and using a separate classifier at each internal node are less common. But this approach can be very powerful and are sometimes used in some advanced DT methods

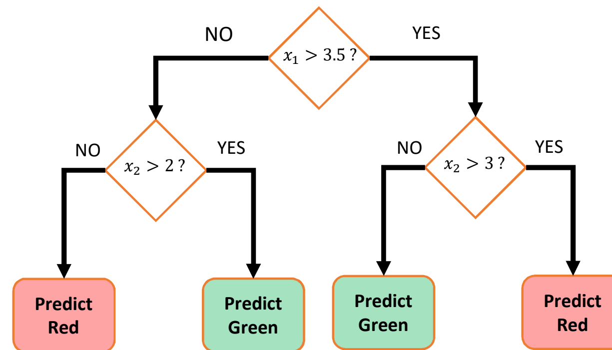
Constructing Decision Trees

Given some training data, what's the "optimal" DT?

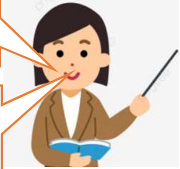
How to decide which rules to test for and in what order?

How to assess informativeness of a rule?





In general, constructing DT is an intractable problem (NP-hard)



Often we can use some "greedy" heuristics to construct a "good" DT

The rules are organized in the DT such that most informative rules are tested first

Hmm.. So DTs are like the "20 questions" game (ask the most useful questions first)

Informativeness of a rule is of related to the extent of the purity of the split arising due to that rule. More informative rules yield more pure splits

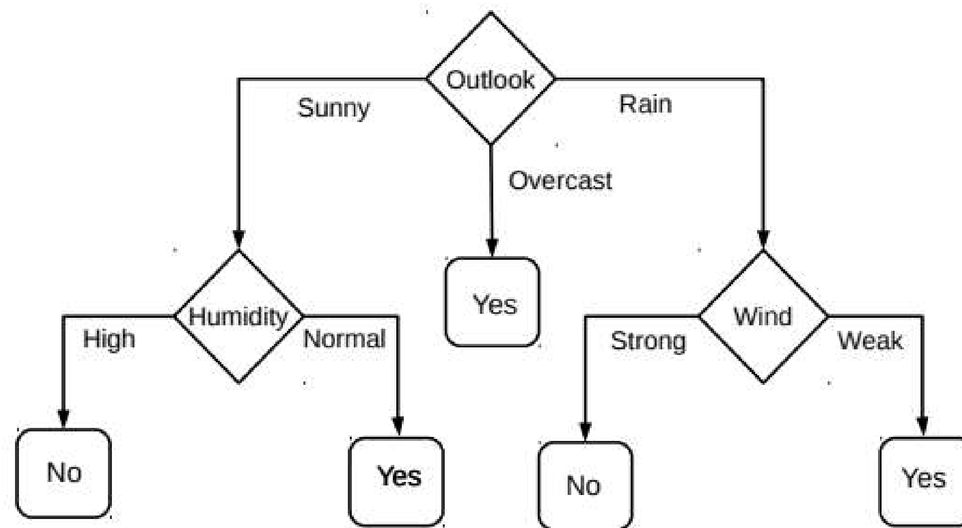
To do so, we use the training data to figure out which rules should be tested at each node

The same rules will be applied on the test inputs to route them along the tree until they reach some leaf node where the prediction is made

Decision Tree Construction: An Example

- Let's consider the playing Tennis example
- Assume each internal node will test the value of one of the features

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



- Question: Why does it make more sense to test the feature "outlook" first?
- Answer: Of all the 4 features, it's the most informative
  - It has the highest information gain as the root node

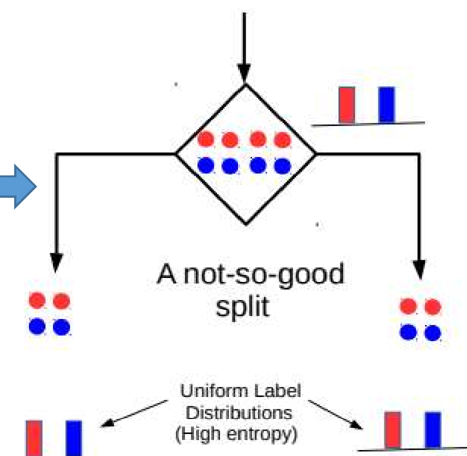
Entropy and Information Gain

- Assume a set of labelled inputs $\boldsymbol{S}$ **from** C classes, p_c as fraction of class c inputs
- **Entropy** of the set $\boldsymbol{S}$ is defined as $\mathrm{H}(\boldsymbol{S}) = -\sum_{c\in C} p_c \log p_c$
- Suppose a rule splits $\boldsymbol{S}$ into two smaller disjoint sets $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
- Reduction in entropy after the split is called information gain

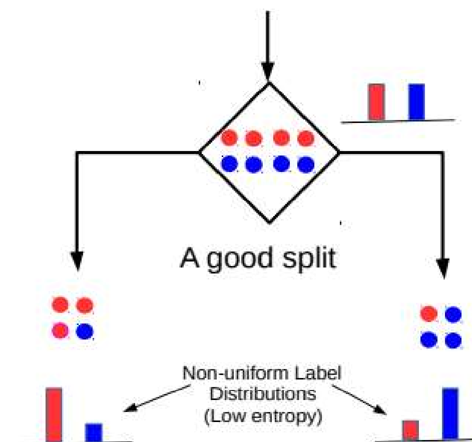
Uniform sets (all classes roughly equally present) have high entropy; skewed sets low

$$IG = H(S) - \frac{|S_1|}{|S|} H(S_1) - \frac{|S_2|}{|S|} H(S_2)$$

This split has a low IG (in fact zero IG)



VS



This split has higher IG

Entropy and Information Gain

- Let's use IG based criterion to construct a DT for the Tennis example
- At root node, let's compute IG of each of the 4 features
- Consider feature "wind". Root contains all examples S = [9+,5-]

$H(S) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.94$

$S_{\text{weak}} = [6+, 2-] \Rightarrow H(S_{\text{weak}}) = 0.811$

$S_{\text{strong}} = [3+, 3-] \Rightarrow H(S_{\text{strong}}) = 1$

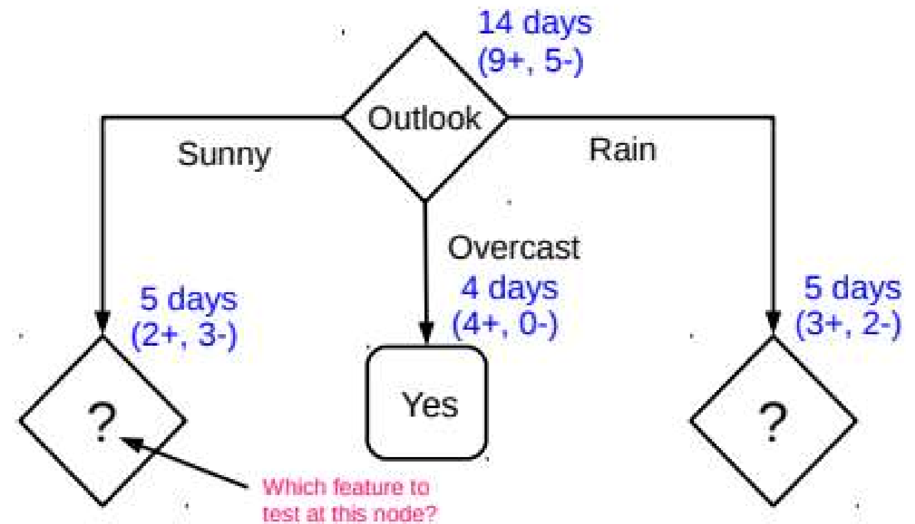
day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no

$$IG(S, wind) = H(S) - \frac{|S_{\text{weak}}|}{|S|} H(S_{\text{weak}}) - \frac{|S_{\text{strong}}|}{|S|} H\left(S_{\text{strong}}\right) = 0.94 - 8/14 * 0.811 - 6/14 * 1 = 0.048$$

- Likewise, at root: IG(S, outlook) = 0.246, IG(S, humidity) = 0.151, IG(S,temp) = 0.029
- Thus we choose "outlook" feature to be tested at the root node
- Now how to grow the DT, i.e., what to do at the next level? Which feature to test next?
- Rule: Iterate - for each child node, select the feature with the highest IG

Growing the tree

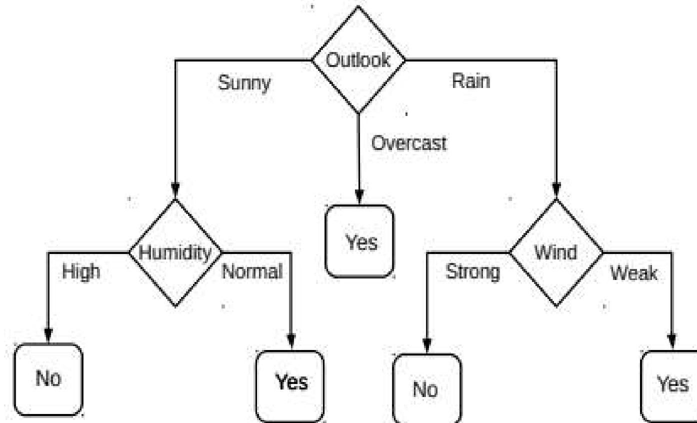
day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no

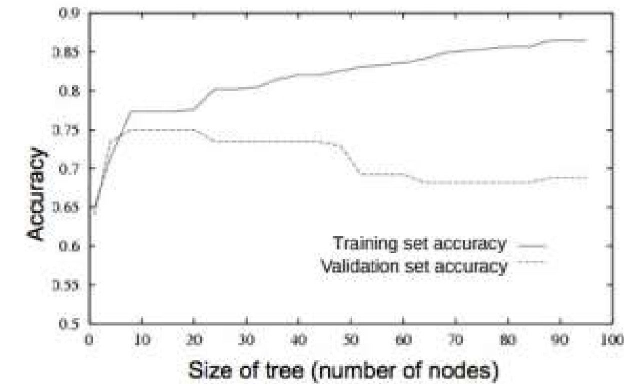


- Proceeding as before, for level 2, left node, we can verify that
  - IG(S,temp) = 0.570, IG(S, humidity) = 0.970, IG(S, wind) = 0.019
- Thus humidity chosen as the feature to be tested at level 2, left node
- No need to expand the middle node (already “pure” - all “yes” training examples)
- Can also verify that wind has the largest IG for the right node
- Note: If a feature has already been tested along a path earlier, we don’t consider it again

When to stop growing the tree?

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no

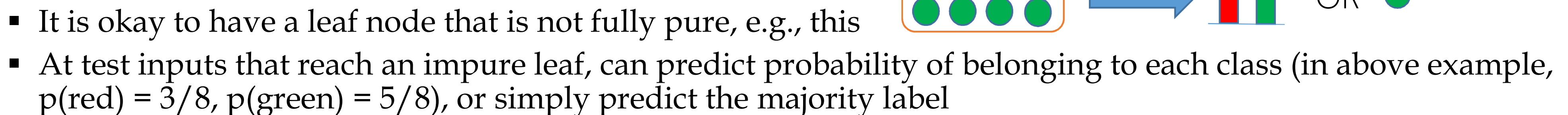




- Stop expanding a node further (i.e., make it a leaf node) when
  - It consist of all training examples having the same label (the node becomes "pure")
  - We run out of features to test along the path to that node
  - The DT starts to overfit (can be checked by monitoring the validation set accuracy)

To help prevent the tree from growing too much!

- **Important:** No need to obsess too much for purity
  - It is okay to have a leaf node that is not fully pure, e.g., this



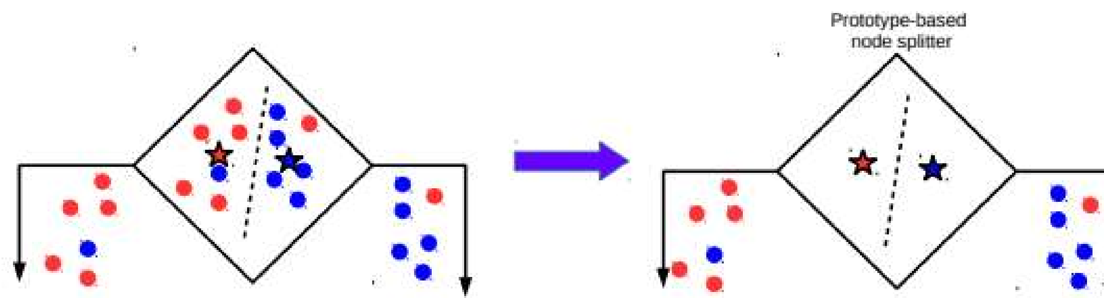
  - At test inputs that reach an impure leaf, can predict probability of belonging to each class (in above example, p(red) = 3/8, p(green) = 5/8), or simply predict the majority label

Avoiding Overfitting in DTs

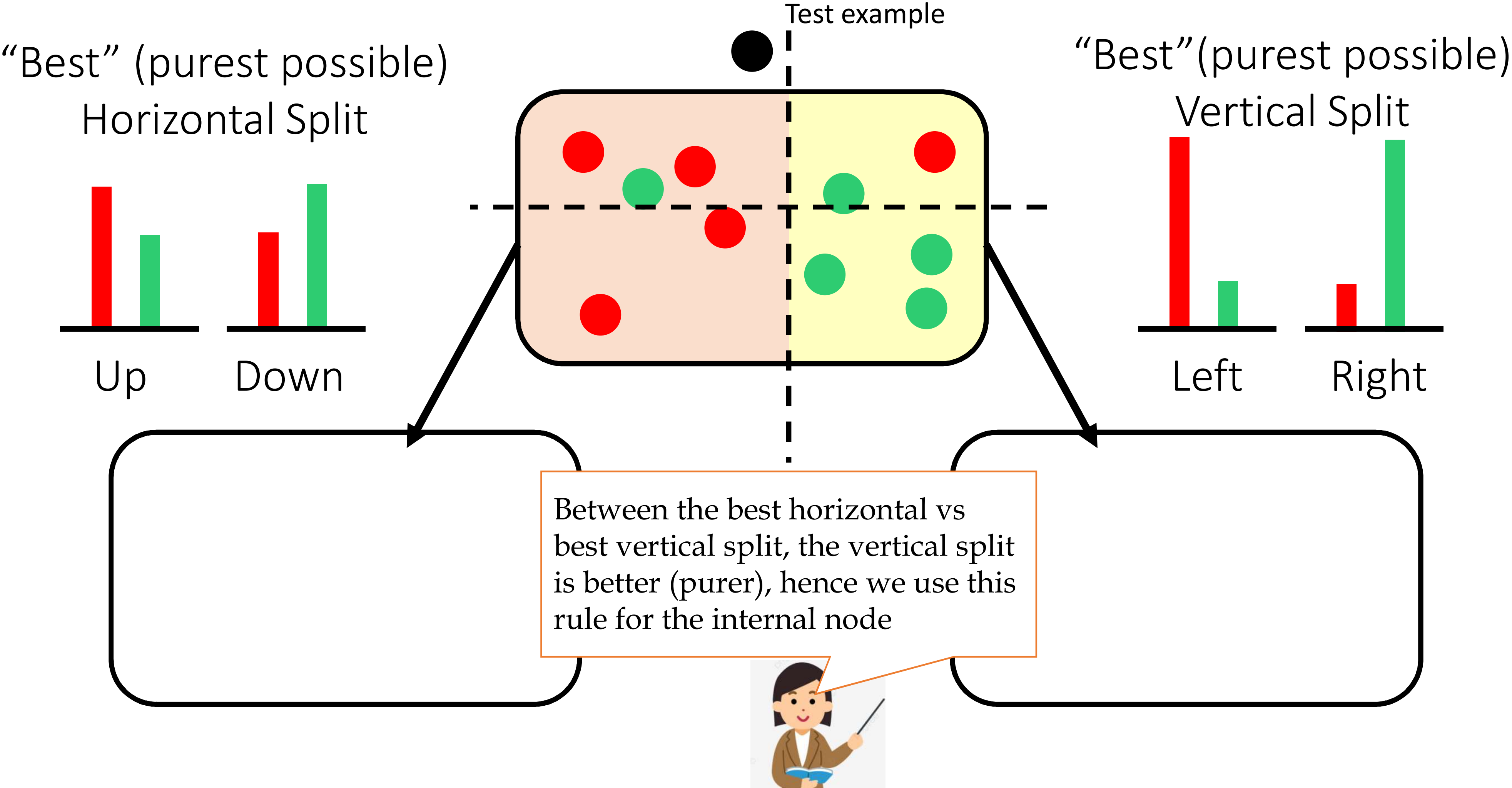
- Desired: a DT that is not too big in size, yet fits the training data reasonably
- Note: An example of a very simple DT is "decision-stump"
  - A decision-stump only tests the value of a single feature (or a simple rule)
  - Not very powerful in itself but often used in large ensembles of decision stumps
- Mainly two approaches to prune a complex DT
  - Prune while building the tree (stopping early)
  - Prune after building the tree (post-pruning)
- Criteria for judging which nodes could potentially be pruned
  - Use a validation set (separate from the training set)
    - Prune each possible node that doesn't hurt the accuracy on the validation set
    - Greedily remove the node that improves the validation accuracy the most
    - Stop when the validation set accuracy starts worsening
  - Use model complexity control, such as Minimum Description Length (will see later)

Decision Trees: Some Comments

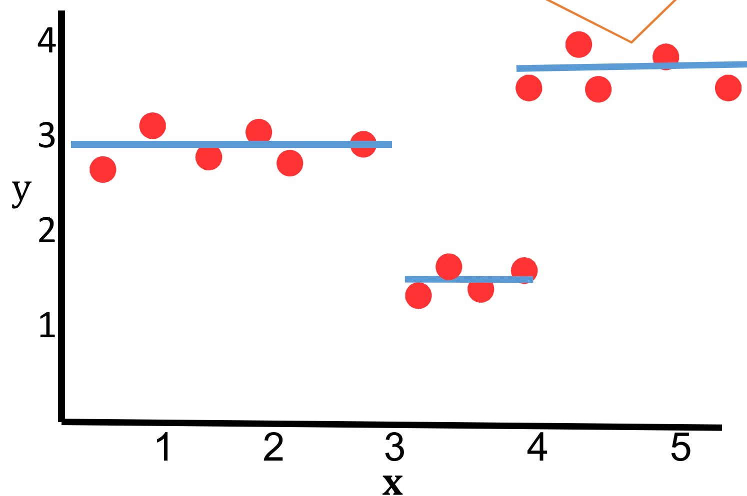
- Gini-index defined as $\sum_{c=1}^{C} p_c(1 - p_c)$ can be an alternative to IG
- For DT regression[1], variance in the outputs can be used to assess purity
- When features are real-valued (no finite possible values to try), things are a bit more tricky
  - Can use tests based on thresholding feature values (recall our synthetic data examples)
  - Need to be careful w.r.t. number of threshold points, how fine each range is, etc.
- More sophisticated decision rules at the internal nodes can also be used
  - Basically, need some rule that splits inputs at an internal node into homogeneous groups
  - The rule can even be a machine learning classification algo (e.g., LwP or a deep learner)
  - However, in DTs, we want the tests to be fast so single feature based rules are preferred
- Need to take care handling training or test inputs that have some features missing



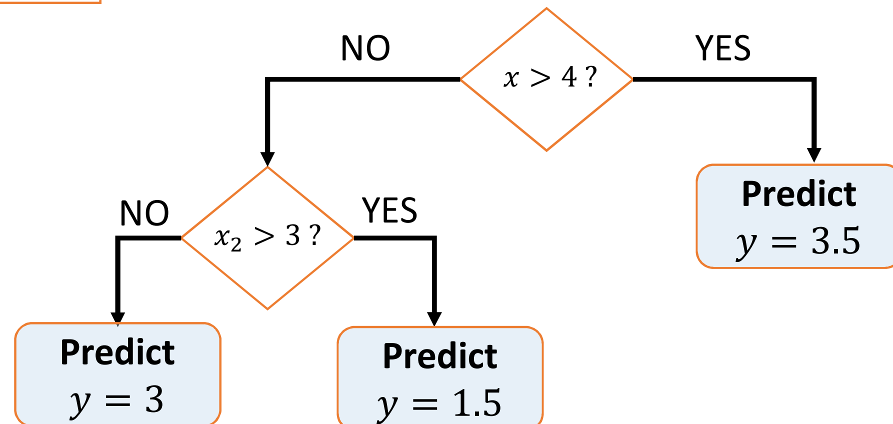
An Illustration: DT with Real-Valued Features

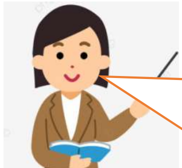


Decision Trees for Regression



Another simple option can be to predict the average output of the training inputs in this region



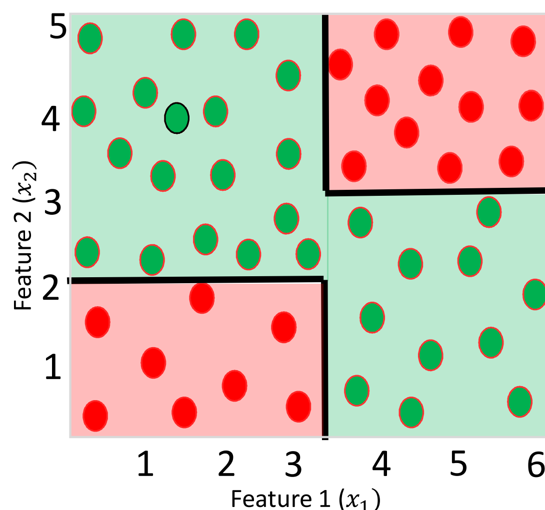


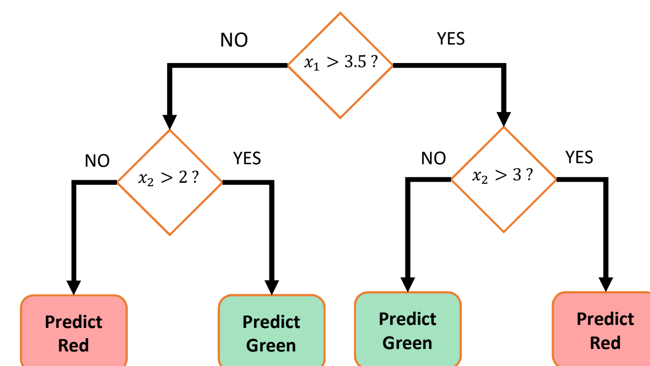
To predict the output for a test point, nearest neighbors will require computing distances from 15 training inputs. DT predicts the label by doing just at most feature-value comparisons! Way faster!

Decision Trees: A Summary

Some key strengths:

- Simple and easy to interpret
- Nice example of "divide and conquer" paradigm in machine learning
- Easily handle different types of features (real, categorical, etc.)
- Very fast at test time





- Multiple DTs can be combined via ensemble methods: more powerful
- Used in several real-world ML applications, e.g., recommender systems, gaming (Kinect)

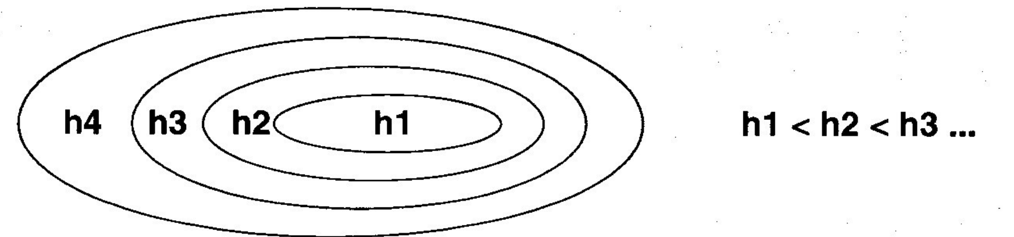
Some key weaknesses:

- Learning optimal DT is (NP-hard) intractable. Existing algos mostly greedy heuristics
- Can sometimes become very complex unless some pruning is applied

Support Vector Machines

A hierarchy of model classes

- Some model classes can be arranged in a hierarchy of increasing complexity.
- How do we pick the best level in the hierarchy for modeling a given dataset?
- We want to get a low error rate on unseen data.
  - This is called "structural risk minimization"



How to choose model calss

- It would be really helpful if we could get a guarantee of the following form:

  Test error rate =< train error rate + f(N, h, p)

  Where N = size of training set,

  h = measure of the model complexity,

  p = the probability that this bound fails

  We need p to allow for really unlucky test sets.

- Then we could choose the model complexity that minimizes the bound on the test error rate.

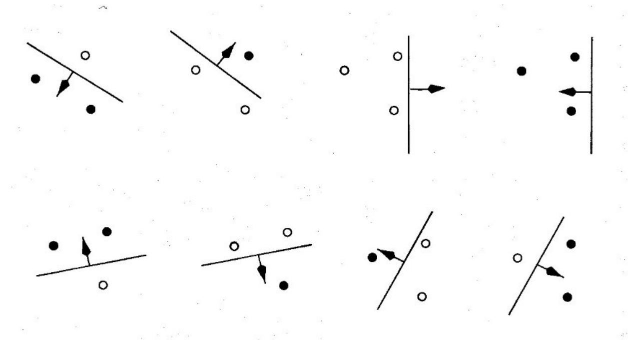
A weird measure of model complexity

- Suppose that we pick n datapoints and assign labels of + or – to them at random. If our model class (e.g. a neural net with a certain number of hidden units) is powerful enough to learn any association of labels with the data, its too powerful!
- Maybe we can characterize the power of a model class by asking how many datapoints it can "shatter" i.e. learn perfectly for all possible assignments of labels.
  - This number of datapoints is called the Vapnik-Chervonenkis dimension.
  - The model does not need to shatter all sets of datapoints of size h. One set is sufficient.
    - For planes in 3-D, h=4 even though 4 co-planar points cannot be shattered.

An example of VC dimension

- Suppose our model class is a hyperplane.
- In 2-D, we can find a plane (i.e. a line) to deal with any

labeling of three points. A 2-D hyperplane shatters 3 points



But we cannot deal with some of the possible labelings of four points. A 2-D hyperplane (i.e. a line) does not shatter 4 points.

Some examples of VC dimension

- The VC dimension of a hyperplane in 2-D is 3.
    - In k dimensions it is k+1.
- Its just a coincidence that the VC dimension of a hyperplane is almost identical to the number of parameters it takes to define a hyperplane.
- A sine wave has infinite VC dimension and only 2 parameters! By choosing the phase and period carefully we can shatter any random collection of one-dimensional datapoints (except for nasty special cases).

The probabilistic guarantee

$$E_{test} \leq E_{train} + \left(\frac{h + h \log(2N/h) - \log(p/4)}{N} \right)^{\frac{1}{2}}$$

where N = size of training set

h = VC dimension of the model class

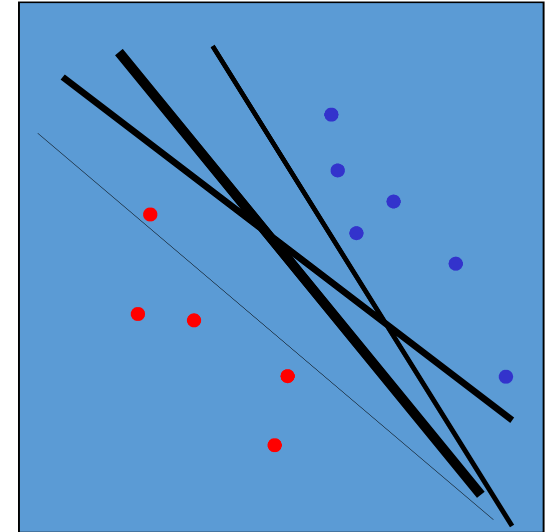
p = upper bound on probability that this bound fails

So if we train models with different complexity, we should pick the one that minimizes this bound

Actually, this is only sensible if we think the bound is fairly tight, which it usually isn't. The theory provides insight, but in practice we still need some witchcraft.

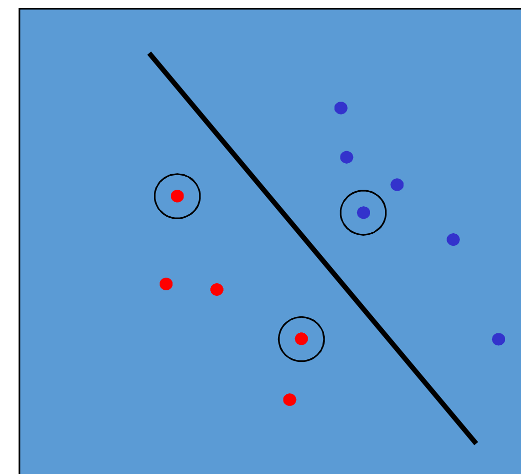
Preventing overfitting when using big sets of features

- Suppose we use a big set of features to ensure that the two classes are linearly separable. What is the best separating line to use?
- The Bayesian answer is to use them all (including ones that do not quite separate the data.)
- Weight each line by its posterior probability (i.e. by a combination of how well it fits the data and how well it fits the prior).
- Is there an efficient way to approximate the correct Bayesian answer?



Support Vector Machines

- The line that maximizes the minimum margin is a good bet.
  - The model class of “hyper-planes with a margin of m” has a low VC dimension if m is big.
- This maximum-margin separator is determined by a subset of the datapoints.
  - Datapoints in this subset are called “support vectors”.
  - It will be useful computationally if only a small fraction of the datapoints are support vectors, because we use the support vectors to decide which side of the separator a test case is on.



The support vectors are indicated by the circles around them.

Training a linear SVM

To find the maximum margin separator, we have to solve the following optimization problem:

$$\mathbf{w}.\mathbf{x}^c + b > +1 \quad \textit{for positive cases}$$

$$\mathbf{w}.\mathbf{x}^c + b < -1 \quad \textit{for negative cases}$$

$$\textit{and} \quad \| \mathbf{w} \|^2 \quad \textit{is as small as possible}$$

- This is tricky but it’s a convex problem. There is only one optimum and we can find it without fiddling with learning rates or weight decay or early stopping.
- Don’t worry about the optimization problem. It has been solved. Its called quadratic programming.
- It takes time proportional to N^2 which is really bad for very big datasets
- so for big datasets we end up doing approximate optimization!

Testing a linear SVM

- The separator is defined as the set of points for which:

$$\mathbf{w}.\mathbf{x} + b = 0$$

$$\textit{so if} \quad \mathbf{w}.\mathbf{x}^c + b > 0 \quad \textit{say its a positive case}$$

$$\textit{and if} \quad \mathbf{w}.\mathbf{x}^c + b < 0 \quad \textit{say its a negative case}$$

A Bayesian Interpretation

Using the maximum margin separator often gives a pretty good approximation to using all separators weighted by their posterior probabilities.

What to do if there is no separating plane

- Use a much bigger set of features.
  - This looks as if it would make the computation hopelessly slow, but in the next part of the lecture we will see how to use the "kernel" trick to make the computation fast even with huge numbers of features.
- Extend the definition of maximum margin to allow non-separating planes.
  - This can be done by using "slack" variables

Introducing slack variables

- Slack variables are constrained to be non-negative. When they are greater than zero they allow us to cheat by putting the plane closer to the datapoint than the margin. So we need to minimize the amount of cheating. This means we have to pick a value for lamba (this sounds familiar!)

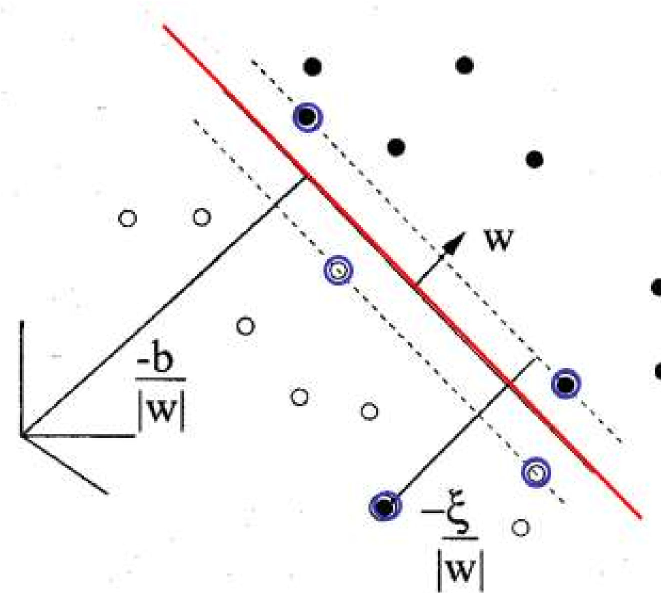
$$\mathbf{w}.\mathbf{x}^c + b \geq +1 - \xi^c \quad \textit{for positive cases}$$

$$\mathbf{w}.\mathbf{x}^c + b \leq -1 + \xi^c \quad \textit{for negative cases}$$

$$\textit{with } \xi^c \geq 0 \quad \textit{for all } c$$

$$\textit{and } \frac{\|\mathbf{w}\|^2}{2} + \lambda \sum_c \xi^c \quad \textit{as small as possible}$$

A picture of the best plane with a slack variable

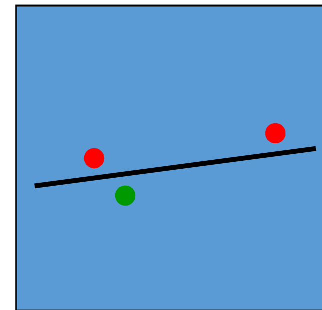


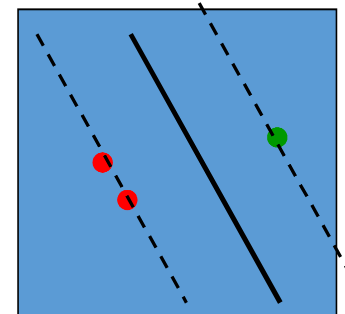
Introducing slack variables

- If we use a large set of non-adaptive features, we can often make the two classes linearly separable.
  - But if we just fit any old separating plane, it will not generalize well to new cases.
- If we fit the separating plane that maximizes the margin (the minimum distance to any of the data points), we will get much better generalization.
  - Intuitively, by maximizing the margin we are squeezing out all the surplus capacity that came from using a high-dimensional feature space.
- This can be justified by a whole lot of clever mathematics which shows that
  - large margin separators have lower VC dimension.
  - models with lower VC dimension have a smaller gap between the training and test error rates.

Why do large margin separators have lower VC dimension

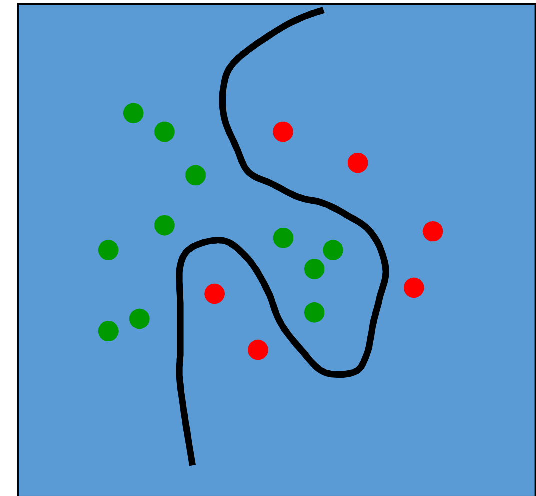
- Consider a random set of N points that all fit inside a unit hypercube.
- If the number of dimensions is bigger than N-2, it is easy to find a separating plane for any labeling of the points.
- So the fact that there is a separating plane doesn't tell us much. It like putting a straight line through 2 data points.
- But there is unlikely to be a separating plane with a margin that is big
- If we find such a plane its unlikely to be a coincidence. So it will probably apply to the test data too.





How to make a plane curved

- Fitting hyperplanes as separators is mathematically easy.
  - The mathematics is linear.
- By replacing the raw input variables with a much larger set of features we get a nice property:
  - A planar separator in the high-dimensional space of feature vectors is a curved separator in the low dimensional space of the raw input variables.



A planar separator in a 20-D feature space projected back to the original 2-D space

- If we map the input vectors into a very high-dimensional feature space, surely the task of finding the maximum-margin separator becomes computationally intractable?
  - The mathematics is all linear, which is good, but the vectors have a huge number of components.

So taking the scalar product of two vectors is very expensive.
The way to keep things tractable is to use ***"the kernel trick"***

The kernel trick makes your brain hurt when you first learn about it, but its actually very simple.

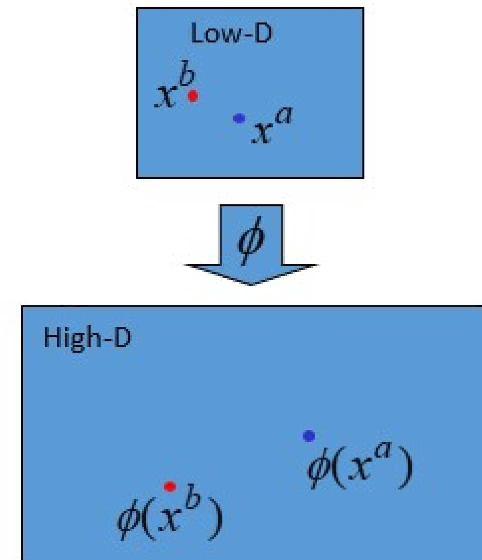
What the kernel trick achieves

- All of the computations that we need to do to find the maximum-margin separator can be expressed in terms of scalar products between pairs of datapoints (in the high-dimensional feature space).
- These scalar products are the only part of the computation that depends on the dimensionality of the high-dimensional space.
  - So if we had a fast way to do the scalar products we would not have to pay a price for solving the learning problem in the high-D space.
- The kernel trick is just a magic way of doing scalar products a whole lot faster than is usually possible.
  - It relies on choosing a way of mapping to the high-dimensional feature space that allows fast scalar products.
- For many mappings from a low-D space to a high-D space, there is a simple operation on two vectors in the low-D space that can be used to compute the scalar product of their two images in the high-D space.

$$K(x^a, x^b) = \phi(x^a) . \phi(x^b)$$

Letting the kernel do the work

doing the scalar product in the obvious way



Dealing with the test data

- If we choose a mapping to a high-D space for which the kernel trick works, we do not have to pay a computational price for the high-dimensionality when we find the best hyper-plane.
  - We cannot express the hyperplane by using its normal vector in the high-dimensional space because this vector would have a huge number of components.
  - Luckily, we can express it in terms of the support vectors.
- But what about the test data. We cannot compute the scalar product $\mathbf{w} \cdot \phi(\mathbf{x})$ because its in the high-D space.
- We need to decide which side of the separating hyperplane a test point lies on and this requires us to compute a scalar product.
- We can express this scalar product as a weighted average of scalar products with the stored support vectors
  - This could still be slow if there are a lot of support vectors .

The classification rule

- The final classification rule is quite simple:

$$bias + \sum_{s \, \varepsilon \, SV} w_s K(x^{test}, x^s) \quad > \; 0$$

The set of support vectors

- All the cleverness goes into selecting the support vectors that maximize the margin and computing the weight to use on each support vector.
- We also need to choose a good kernel function and we may need to choose a lambda for dealing with non-separable cases.

Some commonly used kernels

For the neural network kernel, there is one "hidden unit" per support vector, so the process of fitting the maximum margin hyperplane decides how many hidden units to use. Also, it may violate Mercer's condition

Polynomial: $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}.\mathbf{y} + 1)^p$

Gaussian radial basis function : $K(\mathbf{x}, \mathbf{y}) = e^{-\|\mathbf{x}-\mathbf{y}\|^2 / 2\sigma^2}$

Neural net: $K(\mathbf{x}, \mathbf{y}) = \tanh(k \, \mathbf{x}.\mathbf{y} - \delta)$

Parameters that the user must choose

Performance

- Support Vector Machines work very well in practice.
  - The user must choose the kernel function and its parameters, but the rest is automatic.
  - The test performance is very good.
- They can be expensive in time and space for big datasets
  - The computation of the maximum-margin hyper-plane depends on the square of the number of training cases.
  - We need to store all the support vectors.
- SVM's are very good if you have no idea about what structure to impose on the task.
- The kernel trick can also be used to do PCA in a much higher-dimensional space, thus giving a non-linear version of PCA in the original space.

Support Vector Machines are Perceptrons!

- SVM's use each training case, x, to define a feature K(x, .) where K is chosen by the user.
  - So the user designs the features.
- Then they do "feature selection" by picking the support vectors, and they learn how to weight the features by solving a big optimization problem.
- So an SVM is just a very clever way to train a standard perceptron.
  - All of the things that a perceptron cannot do cannot be done by SVM's (but it's a long time since 1969 so people have forgotten this).

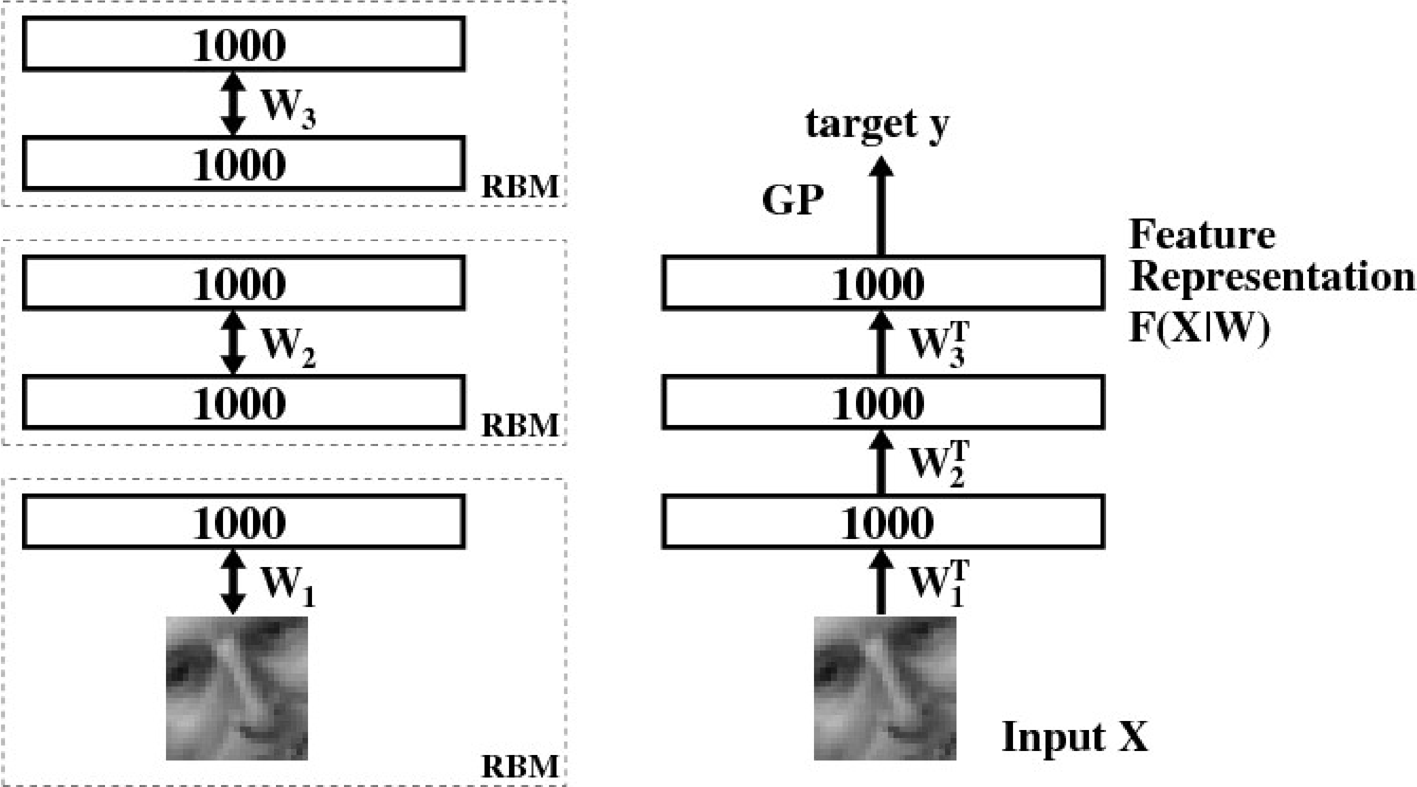
A problem that cannot be solved using a kernel that computes the similarity of a test image to a training case

- Suppose we have images that may contain a tank, but with a cluttered background.
- To recognize which ones contain a tank, it is no good computing a global similarity
  - A non-tank test image may have a very similar background to a tank training image, so it will have very high similarity if the tanks are only a small fraction of the image.
- We need local features that are appropriate for the task. So they must be learned, not pre-specified.
- Its very appealing to convert a learning problem to a convex optimization problem
  - but we may end up by ignoring aspects of the real learning problem in order to make it convex.

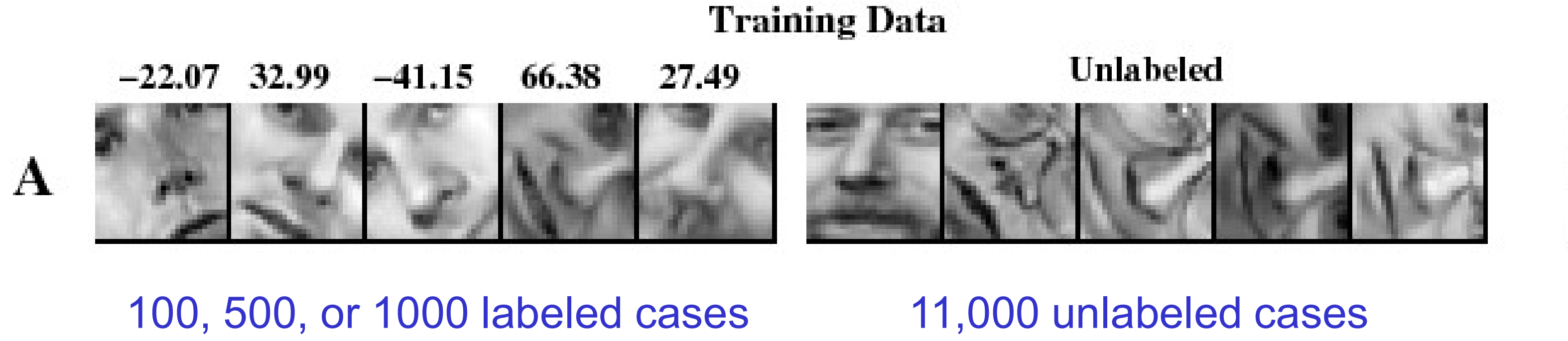
A hybrid approach

- If we use a neural net to define the features, maybe we can use convex optimization for the final layer of weights and then backpropagate derivatives to "learn the kernel".
- The convex optimization is quadratic in the number of training cases. So this approach works best when most of the data is unlabelled.
  - Unsupervised pre-training can then use the unlabelled data to learn features that are appropriate for the domain.
  - The final convex optimization can use these features as well as possible and also provide derivatives that allow them to be fine-tuned.
  - This seems better than just trying lots of kernels and selecting the best ones (which is the current method).

Learning to extract the orientation of a face patch (Ruslan Salakhutdinov)



The training and test sets



The root mean squared error in the orientation when combining GP's with deep belief nets

	GP on the pixels	GP on top-level features	GP on top-level features with fine-tuning
100 labels	22.2	17.9	15.2
500 labels	17.2	12.7	7.2
1000 labels	16.3	11.2	6.4

Conclusion: The deep features are much better than the pixels. Fine-tuning helps a lot.