Lecture 3: Non-Parametric Models

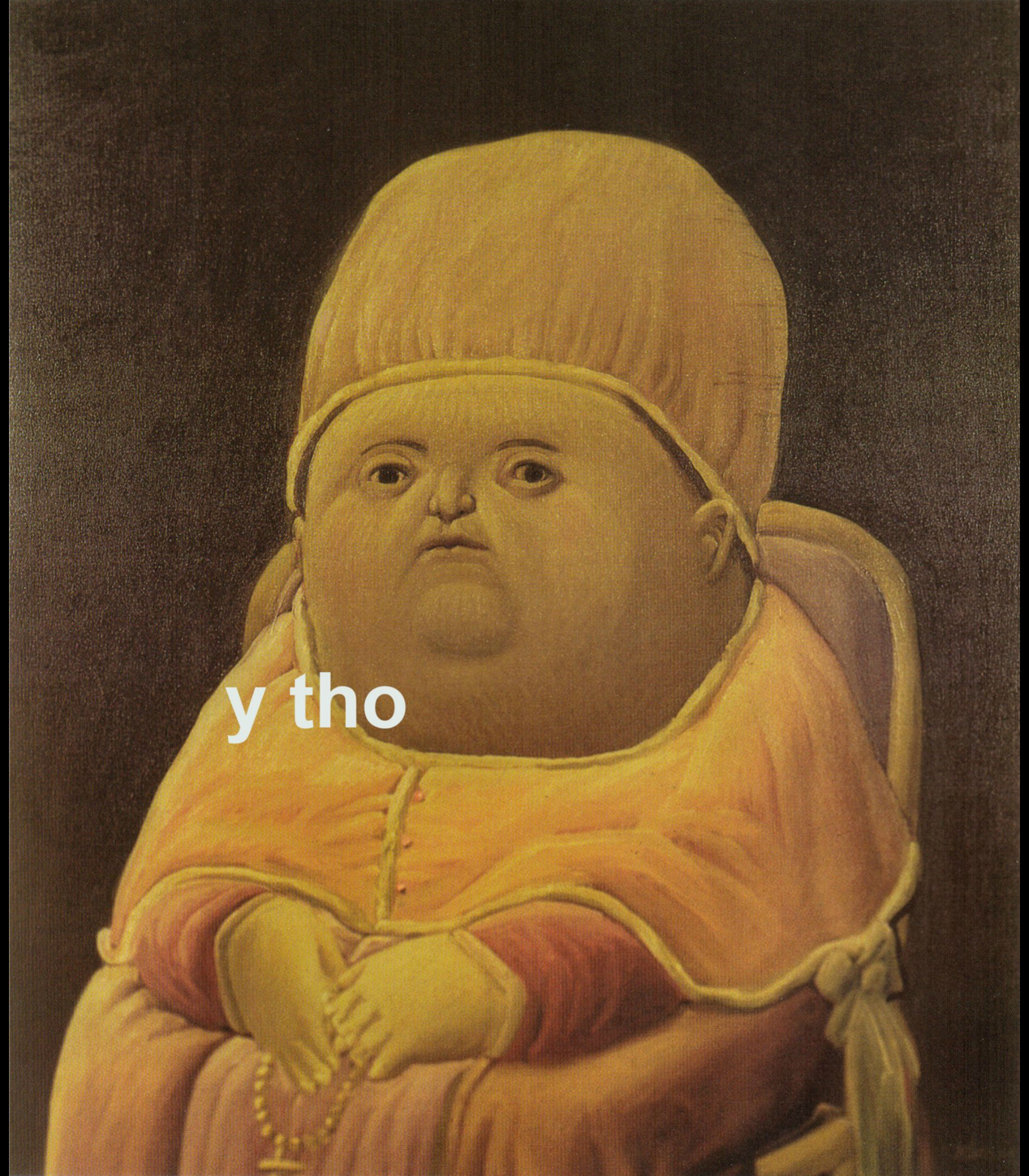
Matthew Caldwell

COMP0088 Introduction to Machine Learning • UCL Computer Science

Contents

3.1: Non What?

COMP0088 Introduction to Machine Learning · UCL Computer Science


y tho

Unhelpful terminology strikes again

- Non-parametric models do have parameters
- The term refers to the fact that they don't assume an *a priori* parameter structure — or assume a parametric distribution
  - → ie, that the data has a particular "shape"
  - → Not doing this often makes them hard to interpret probabilistically
- One common rule of thumb:
  - → A model is "non-parametric" if the number of parameters scales with the number of data points, rather than the number of feature dimensions

Parameters are (often) efficient

- Parametric models distill the “shape” of the data into a compact form
- Having more data is (usually, up to a point) better
  - → Improves estimation of parameter values
  - → Cost of storage and evaluation remain the same
- For a non-parametric model, there can be a downside to having more data
  - → Supports more complex & nuanced predictions
    - ▸ May or may not be desirable depending on the problem
  - → But model may become correspondingly bigger and slower

Partitioning feature space

- Though not necessarily true for all non-parametric models, the models we're looking at this week can be usefully understood as chopping up the feature space into disjoint regions
- Each region is then associated with a particular output value
  - → For our cases, this will be a constant, although it could also be a function
- Regions are effectively independent: there's no prescribed meaning to adjacency or distance between them
- (cf. piecewise fitting examples in last week's basis expansion lecture)

Classification vs Regression

- All this week's methods can be used for both
- But — because of partitioning — it's natural to think in terms of classification
- We'll focus on that, but point out differences for regression
  - → Typically: different loss, aggregate by averaging

3.2: k-Nearest Neighbours

COMP0088 Introduction to Machine Learning · UCL Computer Science

Platonic ideal of a non-parametric model

- Principle: the best estimate of the data distribution is the data itself
- Assume test data is “like” training data
  - → This is really the weakest assumption for learning to be possible at all
- “Training” consists solely of memorising the training set
  - → No attempt to summarise or distill or model it
- Prediction consists of comparing the test sample with all training samples and taking a vote or average of the *k* most similar ones

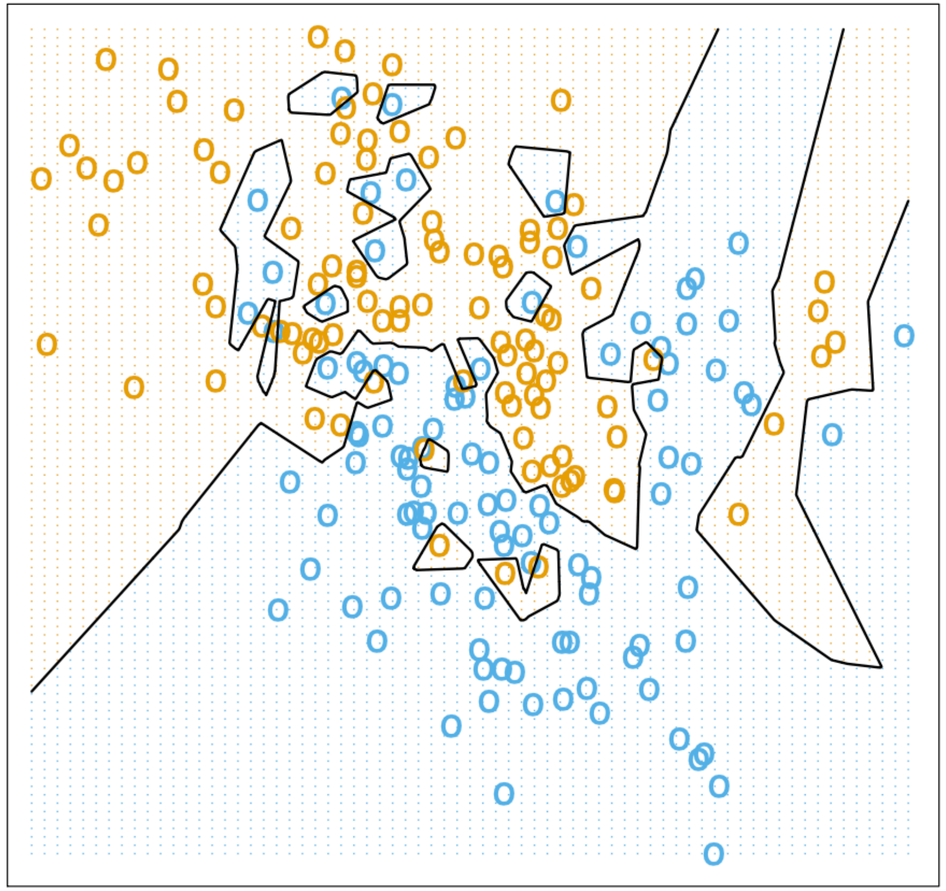
What is “like”?

- Requires a distance (or similarity) metric
  - → And a tie-break policy
- May lose value in high dimensions
- May be sensitive to scale disparities

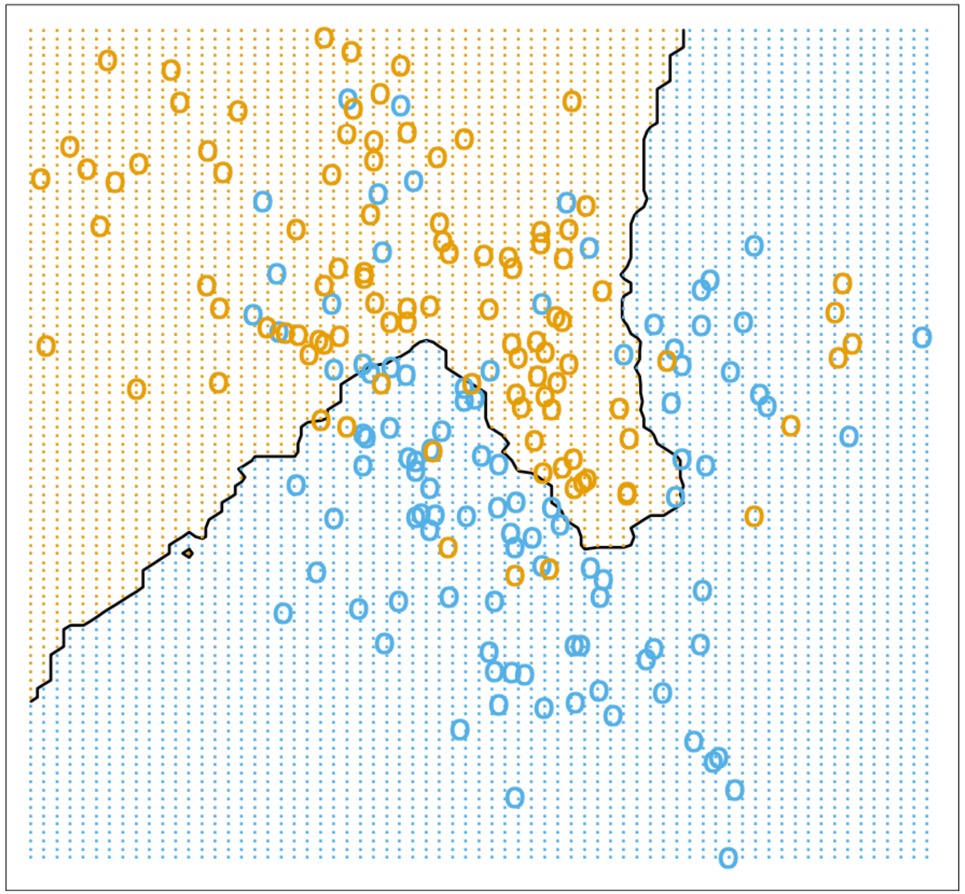
Consequences of k

- With k=1, any bad data point contaminates a neighbourhood
- As k increases, boundaries become less “wiggly”
- If k=n, model reduces to winner takes all (or global average)
- In general increasing k reduces variance

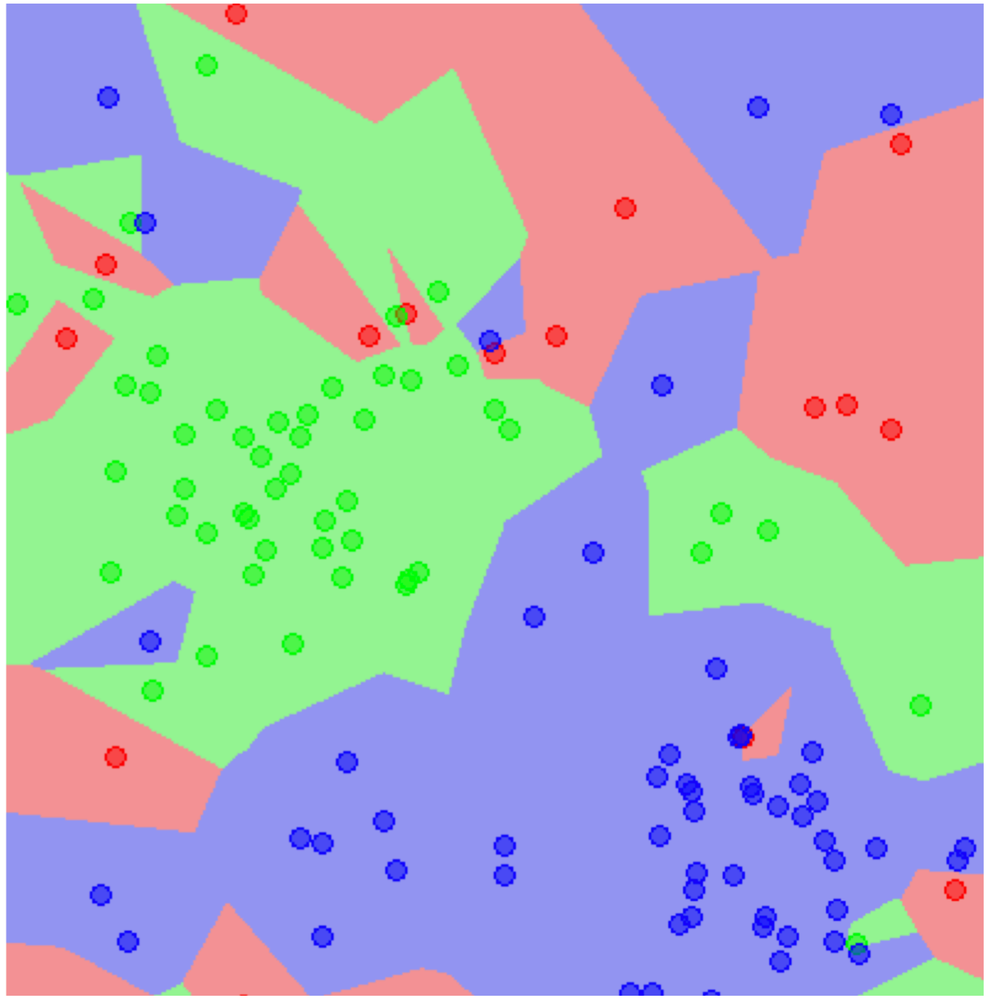
  → But k-NN by its nature is sensitive to vagaries of training data distribution



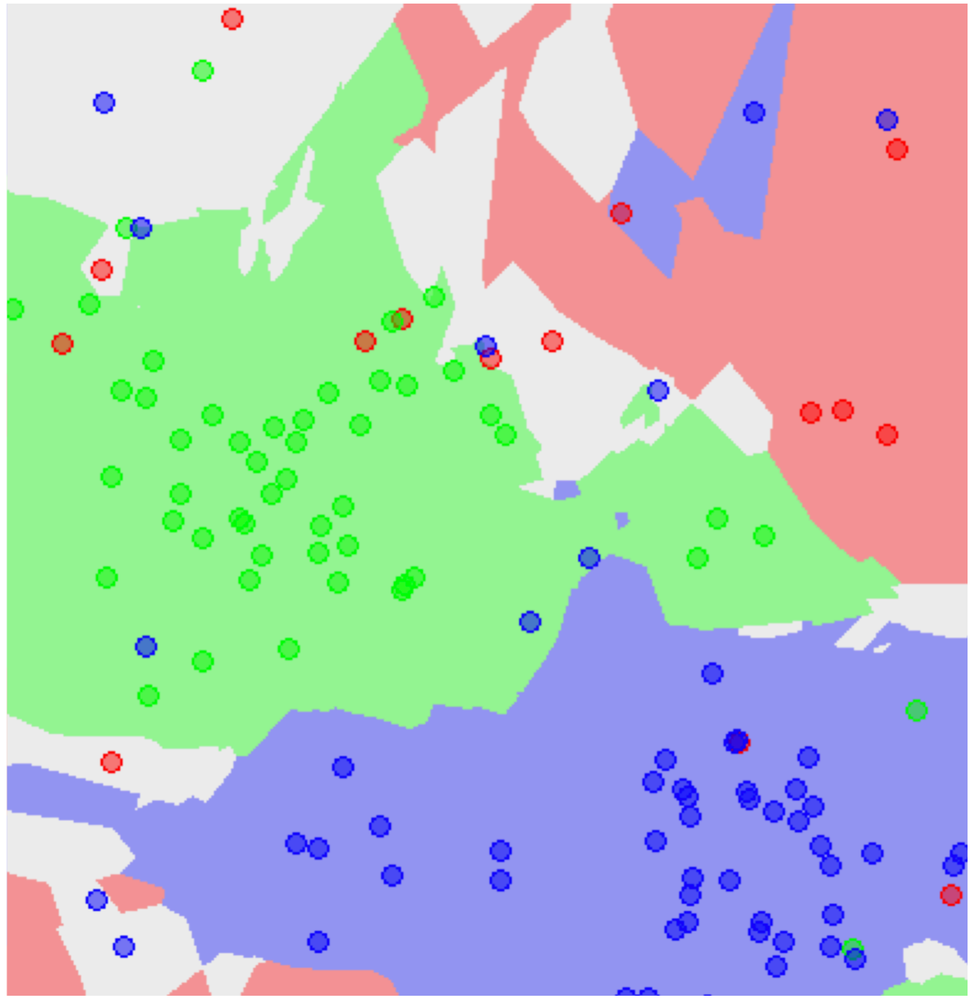
k=1



k=15



k=1



k=3

Pros and cons

- Able to learn complex non-linear decision boundaries (or regression landscapes)
- Easy to understand and implement
- Need to choose *k*
- Risks overfitting
- May be hard to interpret
- Vulnerable to differences in feature scaling and importance
- Potentially very slow to evaluate for large training sets

Scope for improvement

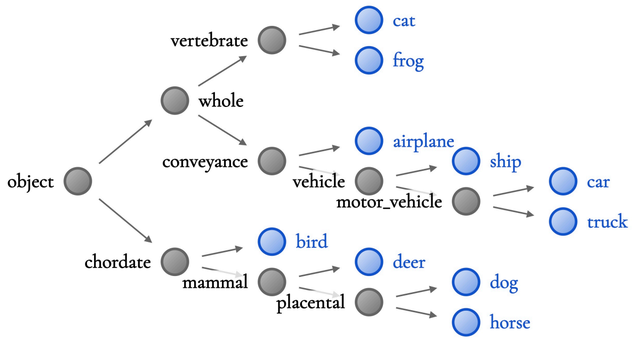
- Data reduction
  - → Many points may not contribute to boundary — prune them
- Dimensionality reduction
  - → Drop or combine features — more on this in week 7
- Weighted averaging/voting
  - → Nearest nearest neighbours contribute more

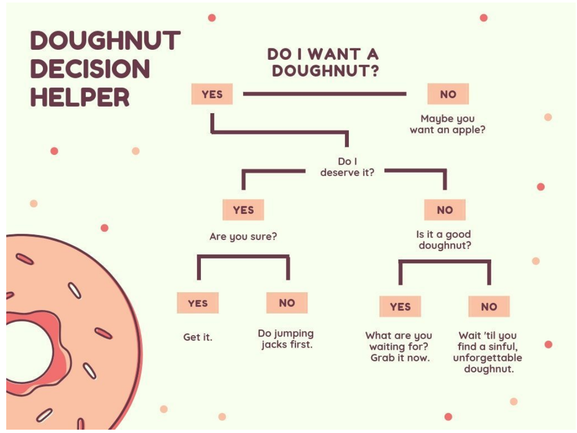
3.3: Decision Trees

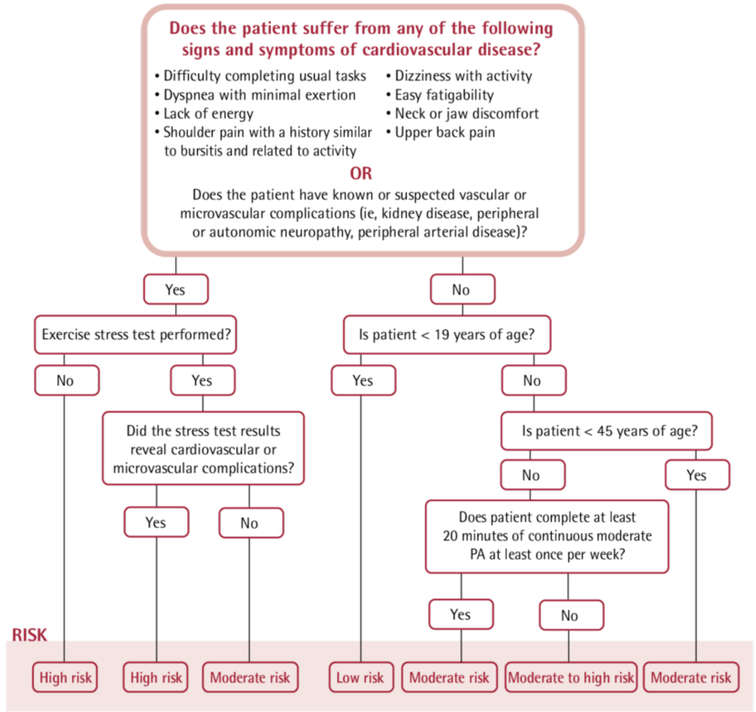
COMP0088 Introduction to Machine Learning · UCL Computer Science

Choose your own adventure

- Decision tree concept is ancient and familiar
  - → Long predates ML as a way to encode procedural or diagnostic “knowledge”
- Algorithms, forms, recipes, flowcharts, games
- Natural graphical representation
  - → Trees are graphs



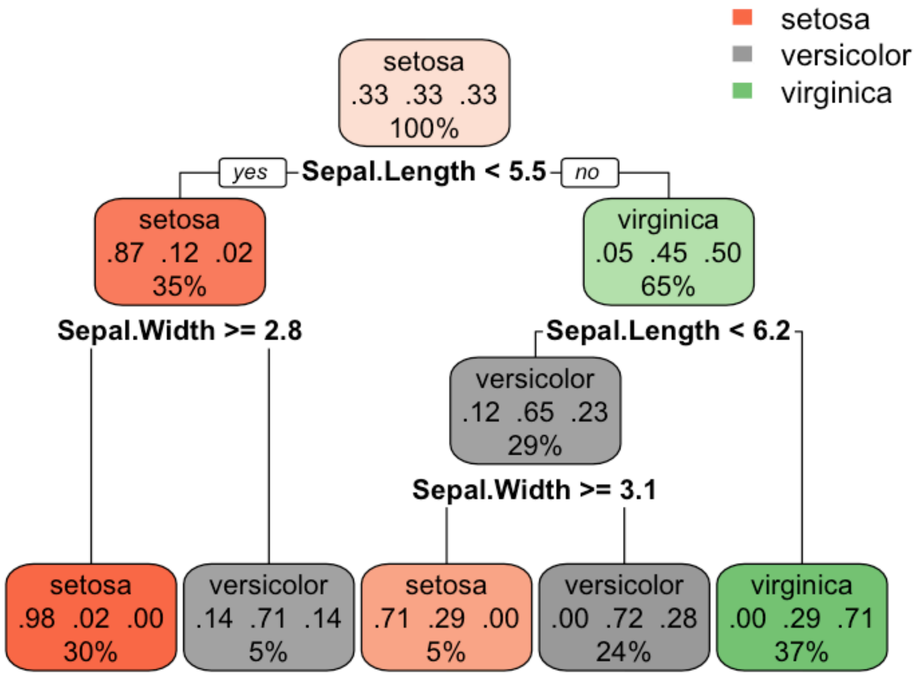


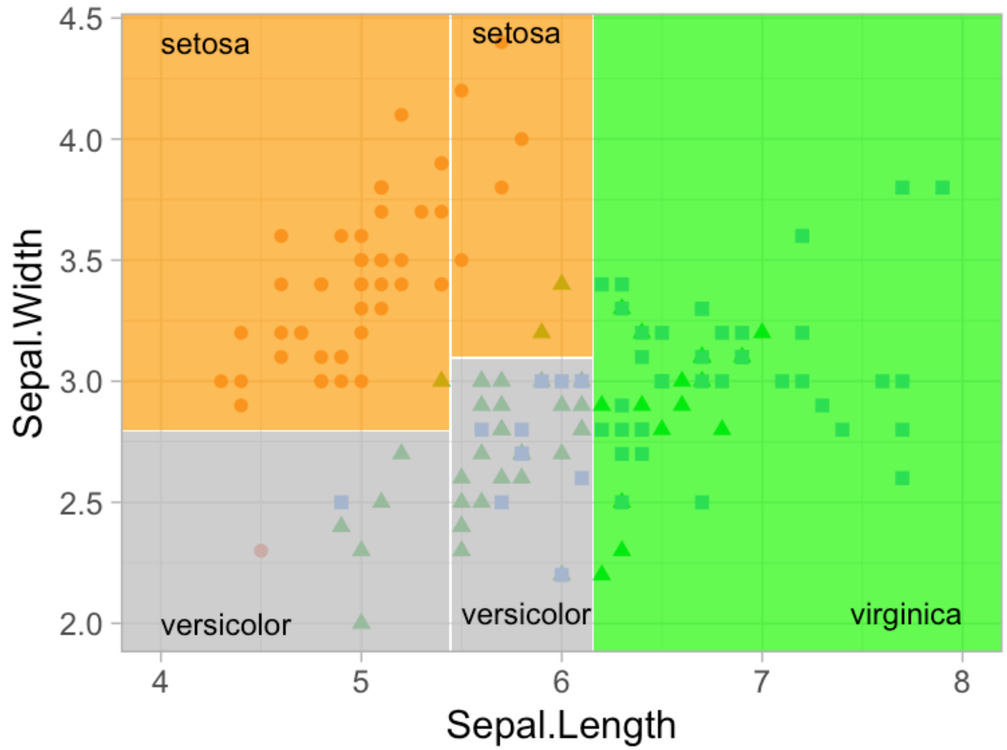


Garden of forking paths

- In a classic flowchart choices are free form, posed in natural language
- Decision could depend on any arbitrary question
  - → How long until the bus arrives?
  - → Napoleon Solo or Ilya Kuryakin?
  - → If I asked the other guard which door had the tiger, what would he say?
- For ML we want to learn from the data, so limit to questions about features
  - → Usually to simple equalities or inequalities on feature values: is $x_5 \geq 2.4$?
  - → Splits don't have to be binary, but we almost always restrict them to be so
    - ▸ A multi-way decision can always be represented as a sequence of binary ones

- Tree represents a recursive partitioning of the feature space
- Each decision splits the current subspace into disjoint parts
- So an alternative visualisation (for 2D features, at least) is something like:





How to grow a tree?

- Recursively
- Greedily
  - → Potentially suboptimal, but avoids combinatorial explosion
- By brute force
  - → Check each feature value present in subset
  - → Can instead use systematic search if training data really large
- But how to choose?
  - → General principle: minimise (weighted) sum of child node losses
  - → Optionally: halt if weighted sum of child losses is no less than loss of unsplit parent

Loss functions for classification trees

- For the current region of input space, *R*, define:

$$p_k = \frac{1}{n_R} \sum_{i \in R} \mathbf{1}(y_i = k)$$

  → ie, proportion of examples in *R* that are of class *k*

- Then the prediction for all examples in *R* is:

$$\hat{y}_R = \operatorname*{argmax}_k p_k$$

  → ie, we just predict the most common class in R

Loss functions for classification trees

- Misclassification error (equivalent to 0-1 loss discussed last week)

$$\frac{1}{n_R} \sum_{i \in R} \mathbf{1}(y_i \neq \hat{y}_R)$$

- [Cross-]Entropy

$$-\sum_{k \in R} p_k \log p_k$$

- Gini Impurity

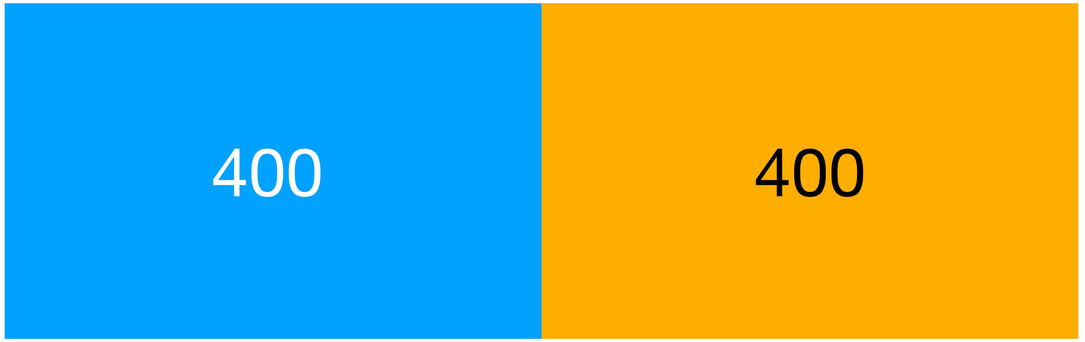
$$\sum_{k \in R} p_k (1 - p_k)$$

NB: don't depend on the prediction!

Loss functions for classification trees

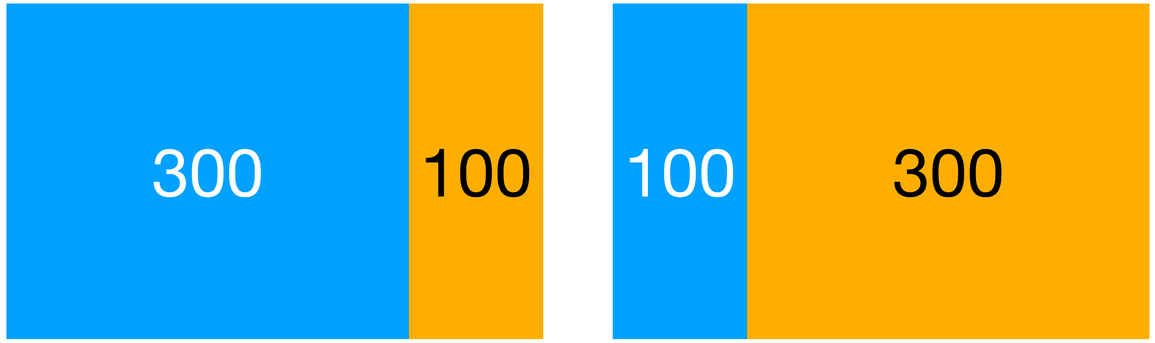
- All three losses will be zero for a pure node (ie, containing only one class)
  - → Pure nodes are good, because we can stop recursing
- Cross-entropy and Gini impurity are differentiable
  - → We are only considering brute force optimisation here but gradient-based methods may be used in other contexts
- Cross-entropy and Gini impurity are also more sensitive to node probabilities
  - → example from Hastie: [400, 400] → {[300, 100], [100, 300]} vs {[400, 200], [200, 0]}
  - → both splits have the same misclassification error: 0.25
  - → but second creates a pure node, which is probably preferable — Gini & Cross-Entropy are both lower in this case

Parent



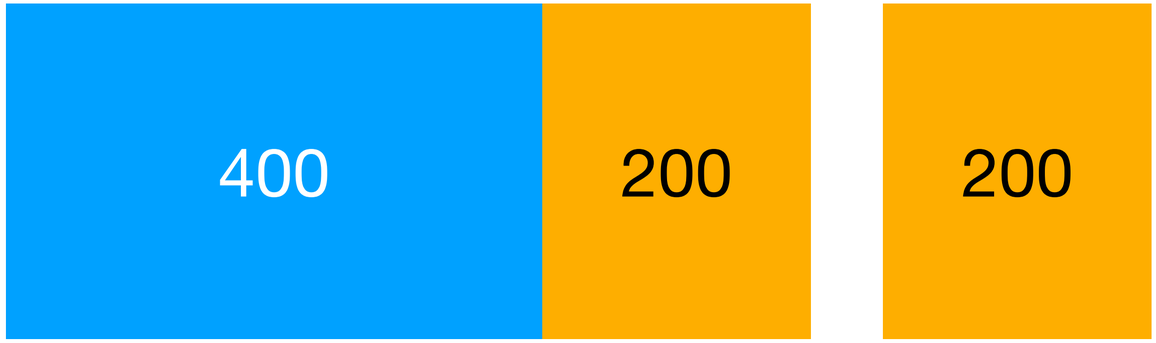
Misclassification: 0.5
Gini: 0.5
Entropy: 0.693

Split Option 1



Misclassification: 0.25
Gini: 0.375
Entropy: 0.56

Split Option 2



Misclassification: 0.25
Gini: 0.333
Entropy: 0.477

Regression trees

- Predict mean of all samples in node
- Typically use a mean-squared error loss to choose splits

Limiting tree growth

- Growing without bounds might eventually produce a tree with a node for every training sample
  - → Roughly equivalent to 1-Nearest Neighbour
  - → Unlikely to get to this point if there's any local structure at all
  - → But still may overfit significantly
- Limits can be imposed (as a kind of regularisation) on:
  - → Maximum tree depth
  - → Maximum overall nodes in tree
  - → Minimum number of data points in node
  - → Minimum improvement in loss

Pruning

- Pre-emptively stopping growth can lead to underfitting
  - → Eg, if best choice at later node made possible by weak choice earlier
- An alternative is to grow the tree to full depth and then iteratively prune the least beneficial branches

Pros and cons

- Fast to evaluate
- Easy to understand and visualise
- Potentially interpretable
- Non-smooth
  - → Basically a discrete method that we can fudge for regression
- Non-additive
  - → Basically: they don’t learn diagonals
- High variance
  - → Minor data differences that alter early splits can result in very different structures

3.4: Ensembles & Random Forests

COMP0088 Introduction to Machine Learning · UCL Computer Science

Fighting variance with aggregation

- Experimental measurements are often noisy
- We might take multiple readings and average them to reduce this
- Similarly, we can often mitigate model variance by fitting multiple models and aggregating their predictions
  - → Obviously, models must be different — can't use a deterministic learning algorithm repeatedly on the same task, something must change
  - → Each model may be wrong in a different way, but some of that averages out
- There are a number of ways to do this, including methods that throw together a whole bunch of completely dissimilar models
- We will focus on two structured approaches: boosting and bagging

Boosting

- Boosting combines multiple weak learners to improve the performance
- Training is sequential
  - → Learner k+1 is contingent on learners 1…k
- Each new learner is trained on a different task
  - → Task is modified at each step to address weaknesses in the ensemble so far
- There are several variants, but the most popular is probably AdaBoost, which we will describe in detail in §3.5

Bagging

- Bagging combines multiple strong learners to reduce the variance
- The learners are trained independently — potentially in parallel
- All learners learn the same task — but training on different randomised subsets of the training data
- Bagging is a contraction of “bootstrap aggregation”
  - → The “bootstrap” comes from statistics — it is a tactic to improve estimates about a population for which there is limited data by resampling that data with replacement
- Easily the most popular bagging method is Random Forests, which we’ll discuss shortly

Meta-algorithms

- Bagging and boosting do not prescribe a particular model or learning algorithm for their ensemble members
- They are (families of) “meta-algorithms”, which specify how to combine other models for improved results
- It is possible to use them to aggregate many different kinds of machine learning model
- However, decision trees are probably the most common choice for both
  - → NB: they are “weak” in one and “strong” in the other because of different hyperparameterisation

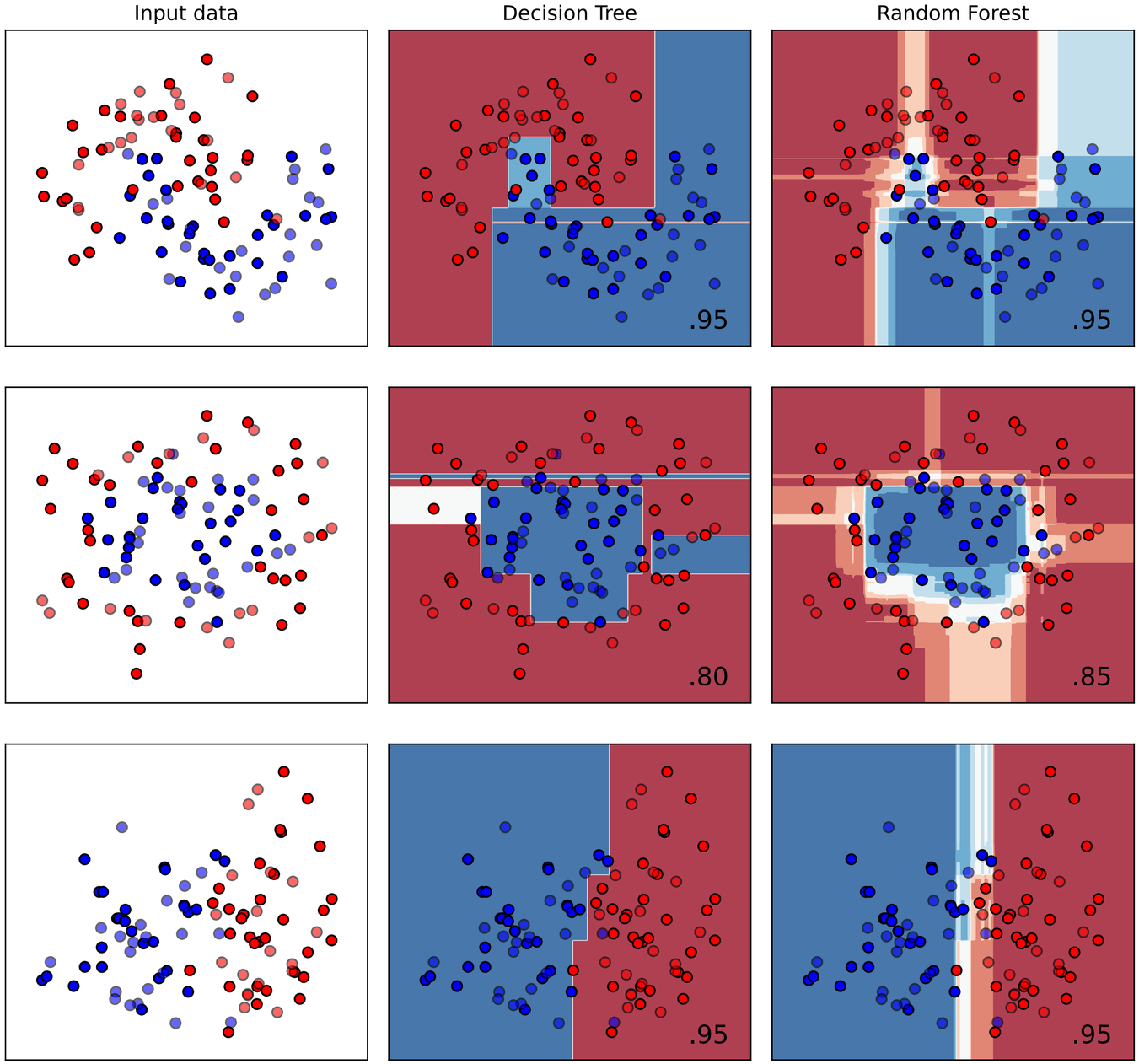
Random forests

- Decision trees are (or can be) able to fit complex decision functions
- But they are high variance and prone to overfitting
- We can combat this by aggregating over an ensemble of trees — a forest

- Fun fact: “random forest” is a registered trademark!
  - → The algorithm itself is public domain
  - → The circumstances where you would care about this are rare
  - → …but still, it’s a bit weird

Diversification

- Fitting decision trees is deterministic, so we need to vary it somehow
- One way is bagging: fitting to bootstrap samples of the training data
  - → Randomly resample from the training set with replacement
  - → Any given sample may appear once, multiple times or not at all
  - → Each bootstrap sample is a different approximation of the true distribution
- Random forests add a further source of variation by randomly selecting different feature subsets for each tree
- Extremely randomised trees is an alternative approach that randomises the fitting procedure to increase the diversity of the ensemble, instead of bootstrapping the data


Input data
Decision Tree
Random Forest
.95
.95
.80
.85
.95
.95

Pros and cons

- Easy to use
- Good performance with reduced variance and less overfitting
- Uninterpretable
  - → Sacrifices clear decision structure of single trees
- Strong learners can be slower to train
- Hyperparameters
  - → Number of trees
  - → Tree depth
  - → Feature subsetting

3.5: AdaBoost

COMP0088 Introduction to Machine Learning · UCL Computer Science

Adaptive boosting

- Probably most popular boosting meta-algorithm
- Combines weak learners to build a strong learner
- We'll focus on the binary classification case
  - → Extends to multiclass and regression but in slightly fiddly ways that would be distracting here
- Different labelling convention: classes are {-1, 1} rather than {0, 1}
  - → Gives a neat way of calculating update direction: $y\hat{y}$ or $yf(\mathbf{x})$ ← "margin"
  - → Makes the final aggregation simpler: we'll want to make a weighted sum where opposing classifications have magnitude in the relevant direction, rather than zeroing out the negative ones

Weak learners

- There isn’t really a set definition of “weakness”
- Must perform better than chance — otherwise nothing learned at all
  - → Performance constraint for classic Adaboost is actually > 50%
  - → This is one of the obstacles to extending to multiclass
- But might be only a bit better
- Weak learners can be cheap to train, making for efficient ensembles
- It is possible to boost strong learners, but less worthwhile

Decision stumps

- The classic weak learner used for AdaBoost
- A decision tree of depth 1
  - → ie, just a single split with no recursion
- Fast to train
- But very weak
  - → Individual stump performance likely to be terrible for almost all problems

Sample weighting

- Each learner is chosen to minimise a weighted misclassification error

$$\epsilon = \sum_i w_i \mathbf{1}(y_i \neq \hat{y}_i)$$

- Weights define the importance attached to each observation
- All weights are initially equal, set to 1/n

Predictor weighting

- After a learner is fitted, it is assigned a prediction weight (α) based on its weighted prediction error:

$$\alpha = \frac{1}{2} \log \left(\frac{1 - \epsilon}{\epsilon} \right)$$

- Note the error is for the predictions of this learner alone, not the whole ensemble
- Information accumulates in the loss weights w_i

Sample re-weighting

- The weights are adjusted according to predictions
  - → Two different formulations are common, using only errors or all predictions:

$$w_i \leftarrow w_i \, e^{\alpha \, \mathbf{1}(y_i \neq \hat{y}_i)} \qquad\qquad w_i \leftarrow w_i \, e^{-\alpha y_i \hat{y}_i}$$

- Either way, we then normalise adjusted weights to sum to 1:

$$w_i = \frac{w_i}{\sum_j w_j}$$

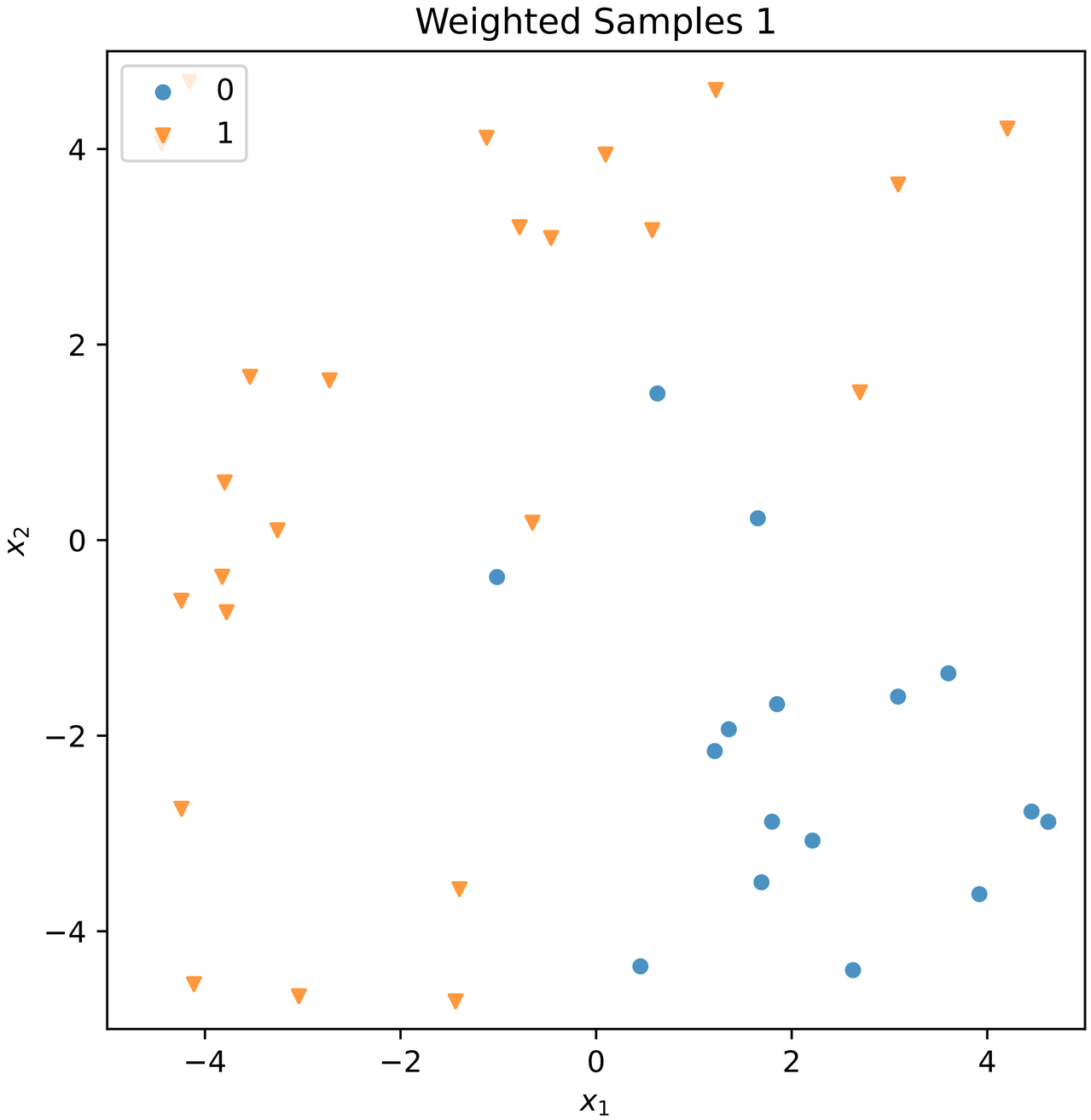
- Overall effect is to upweight wrongly classified samples, downweight correct

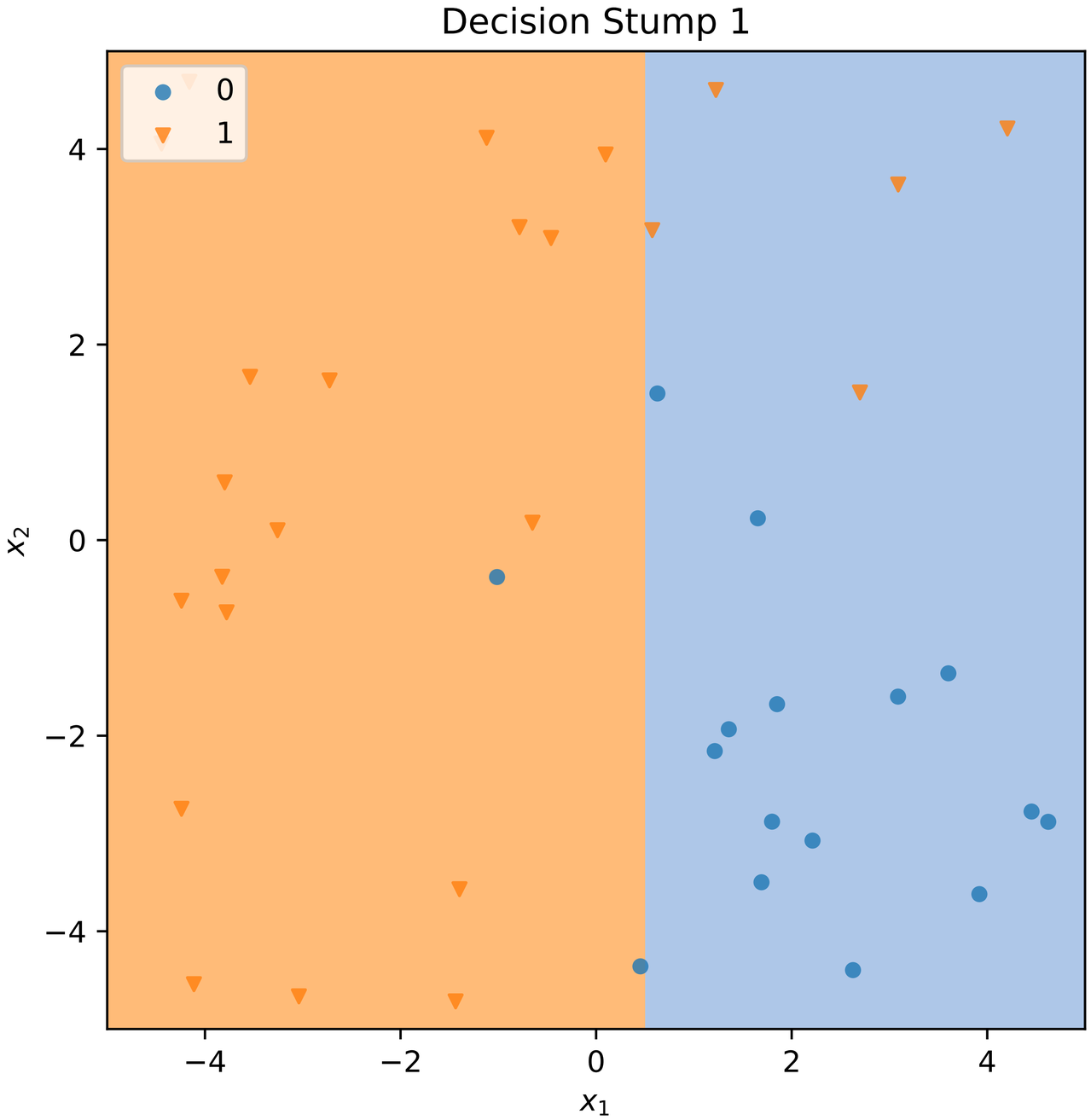
Ensemble prediction

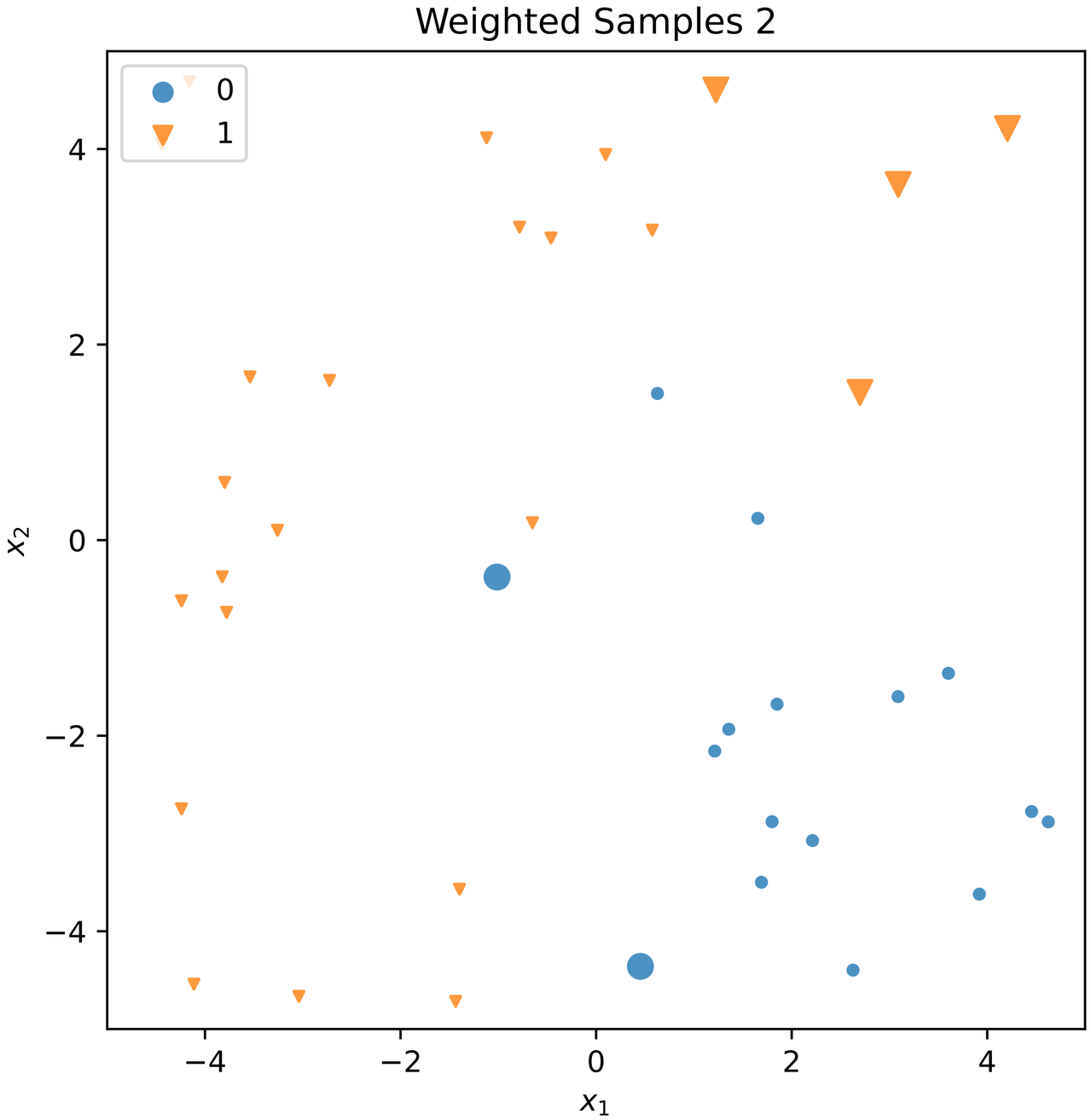
- Having trained k learners, $h_1 \ldots h_k$, each with associated weight $\alpha_1 \ldots \alpha_k$, an unseen test sample **x** can be predicted as:

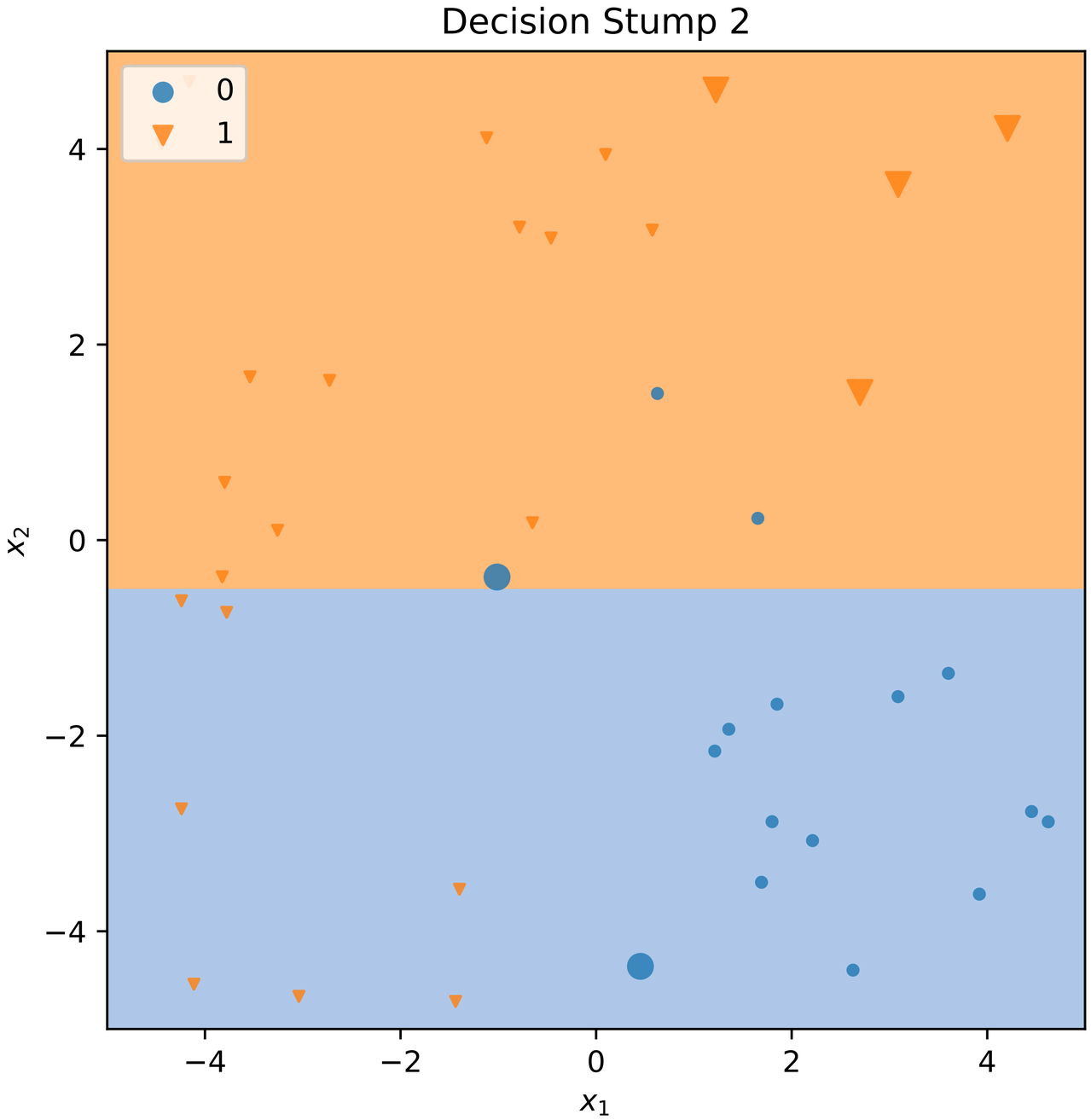
$$\hat{y} = \operatorname{sgn} \sum_{t=1}^{k} \alpha_t h_t(\mathbf{x})$$

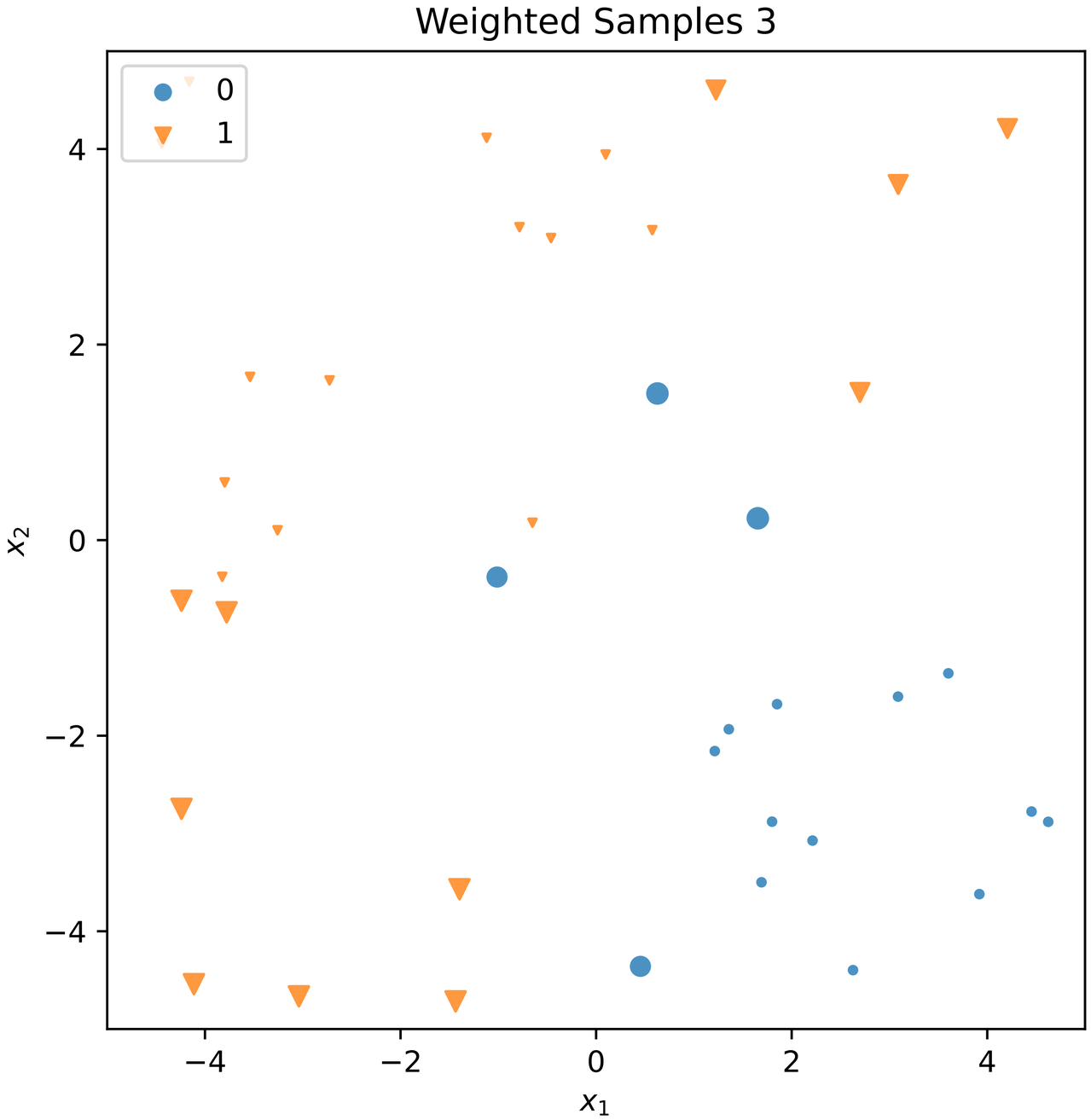
→ Sample weights w_i are used only in training and discarded afterwards

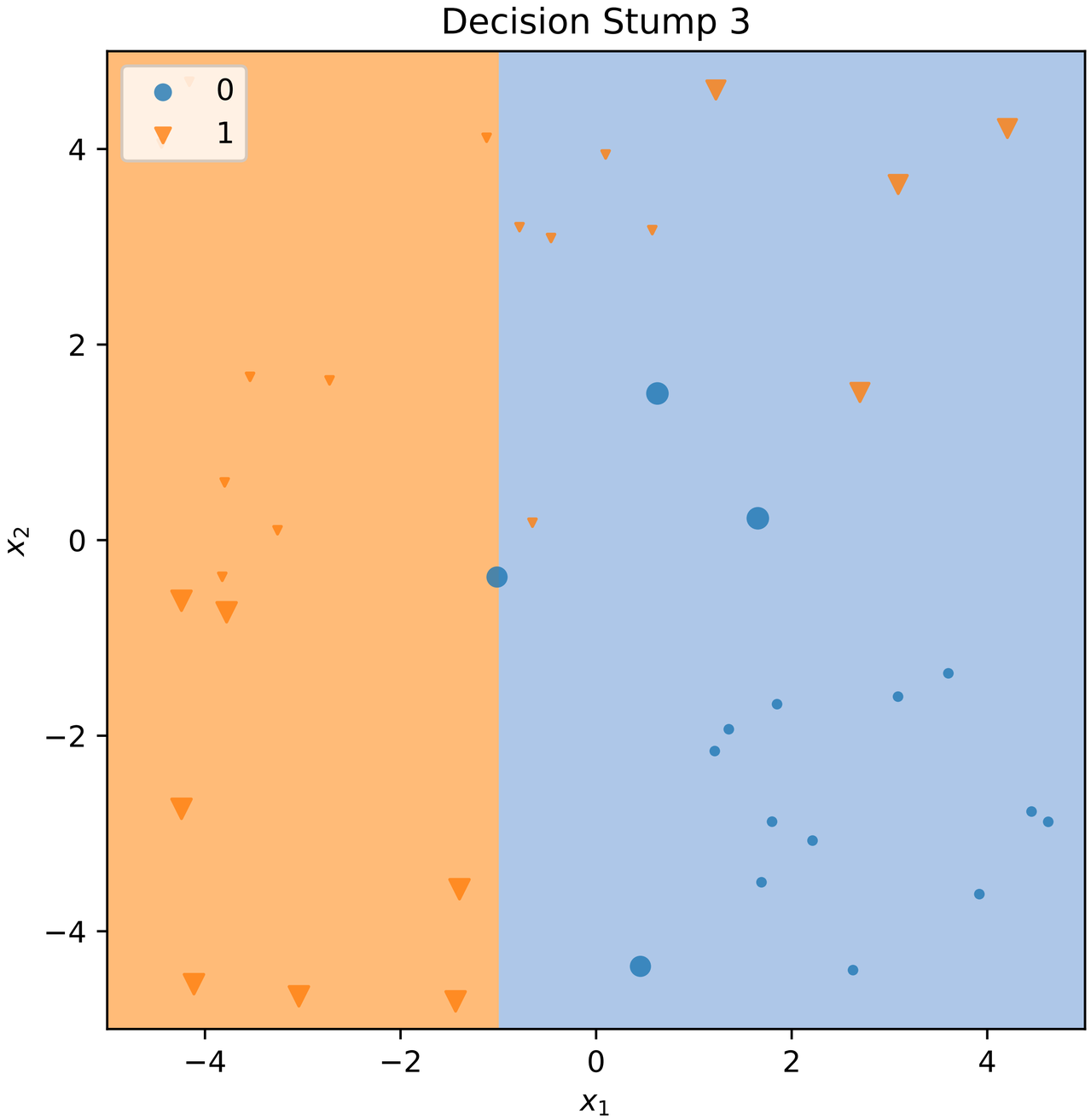

Weighted Samples 1
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1

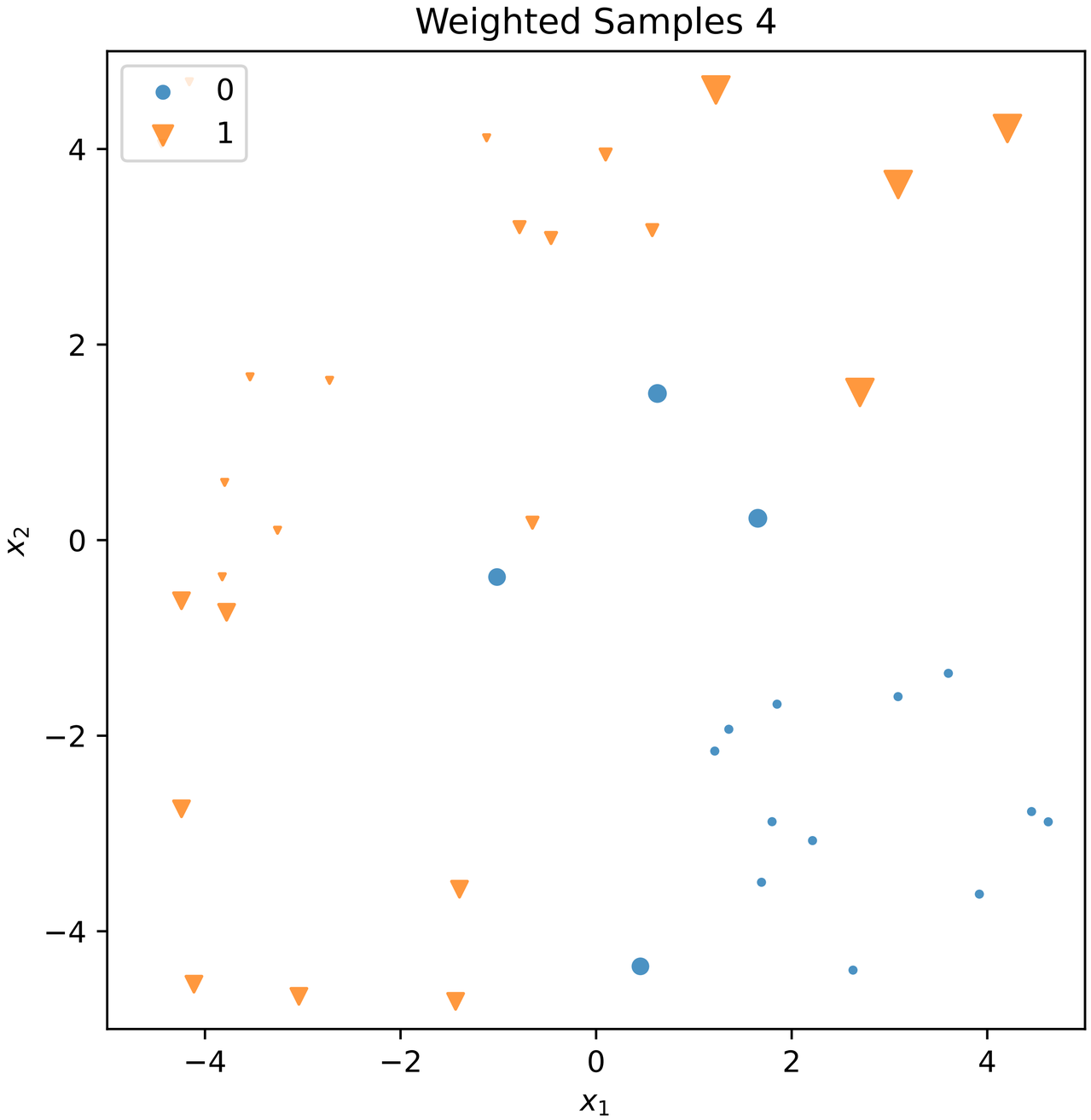

Decision Stump 1
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1

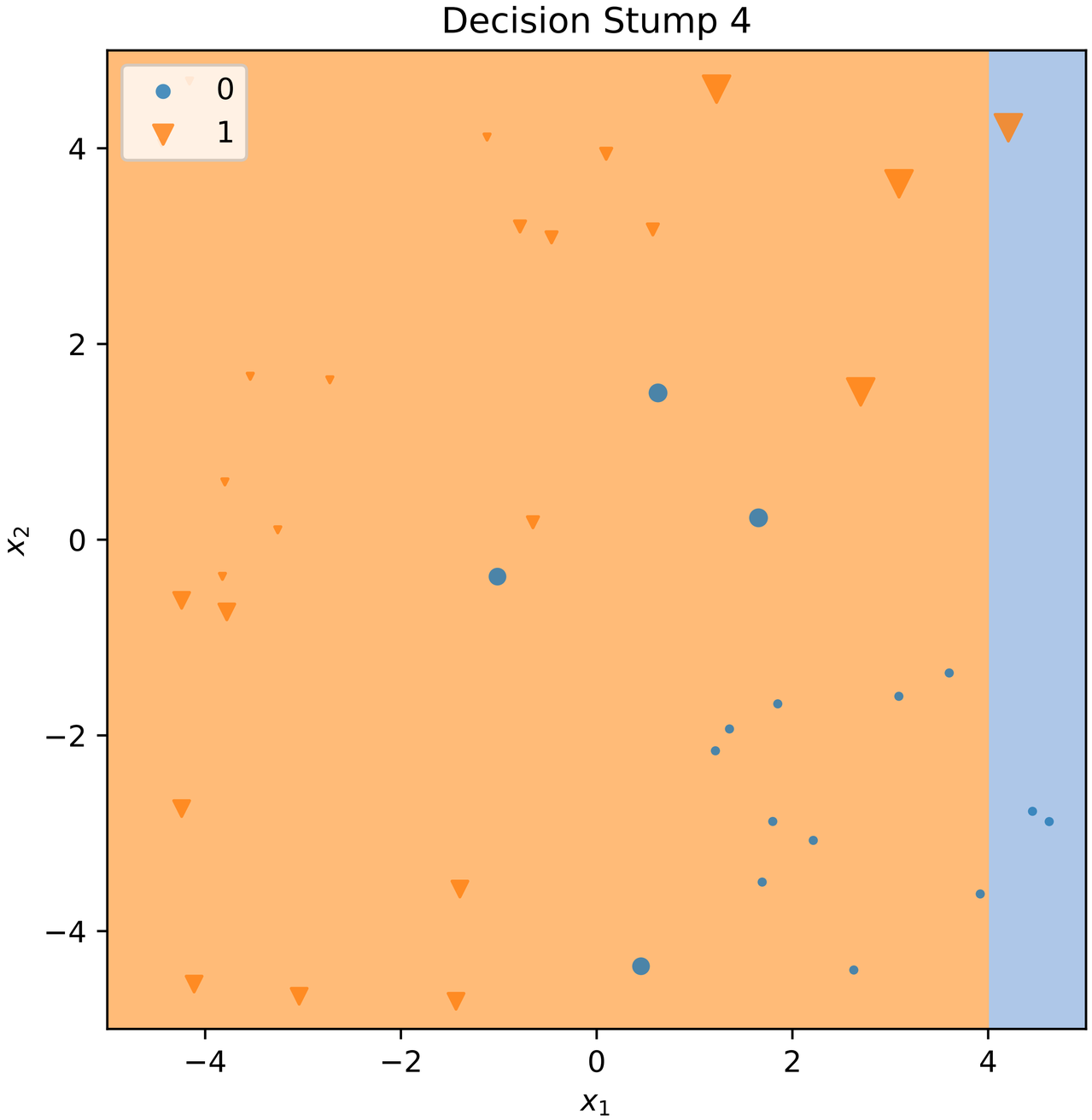

Weighted Samples 2
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1

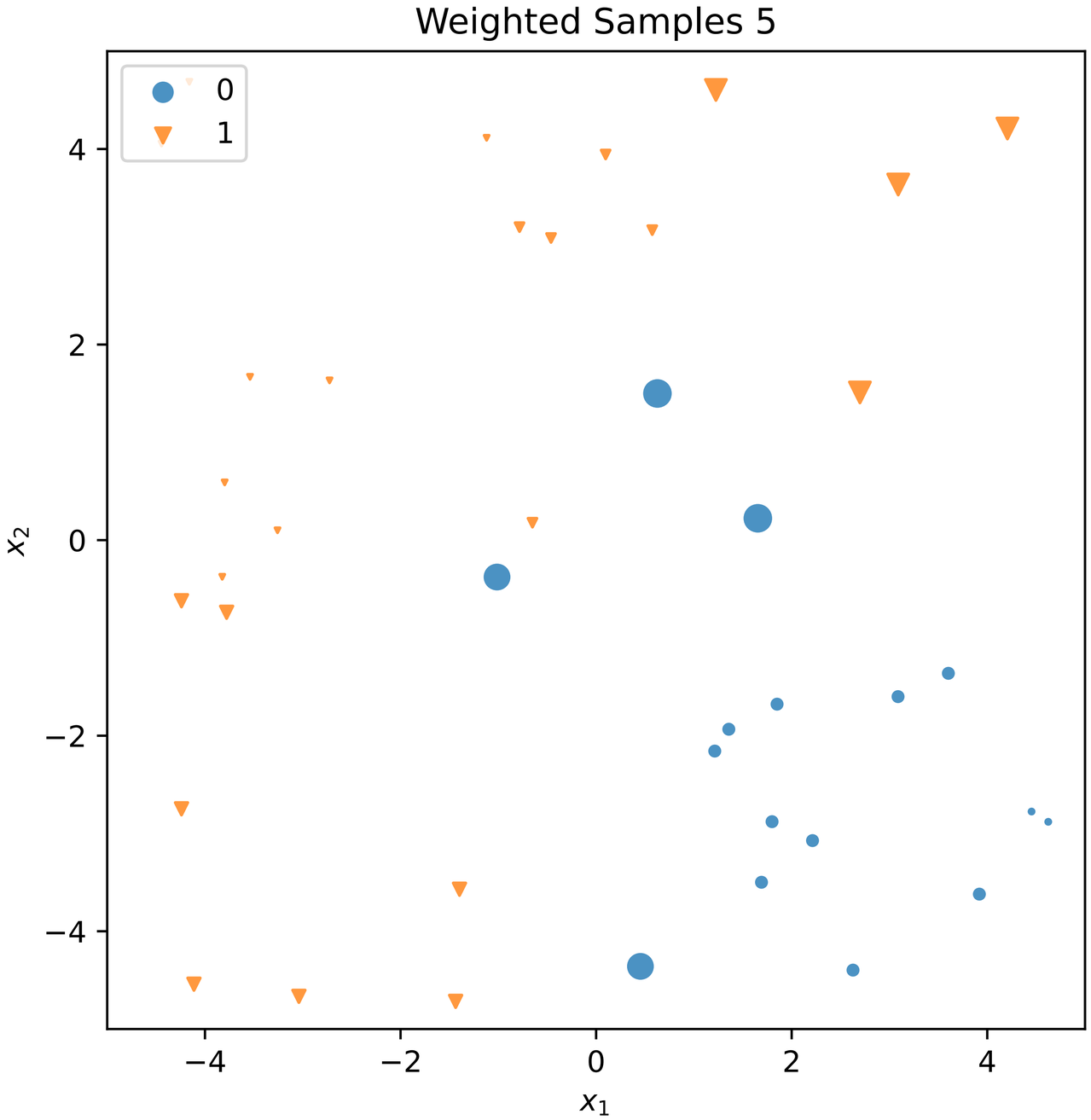

Decision Stump 2
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1

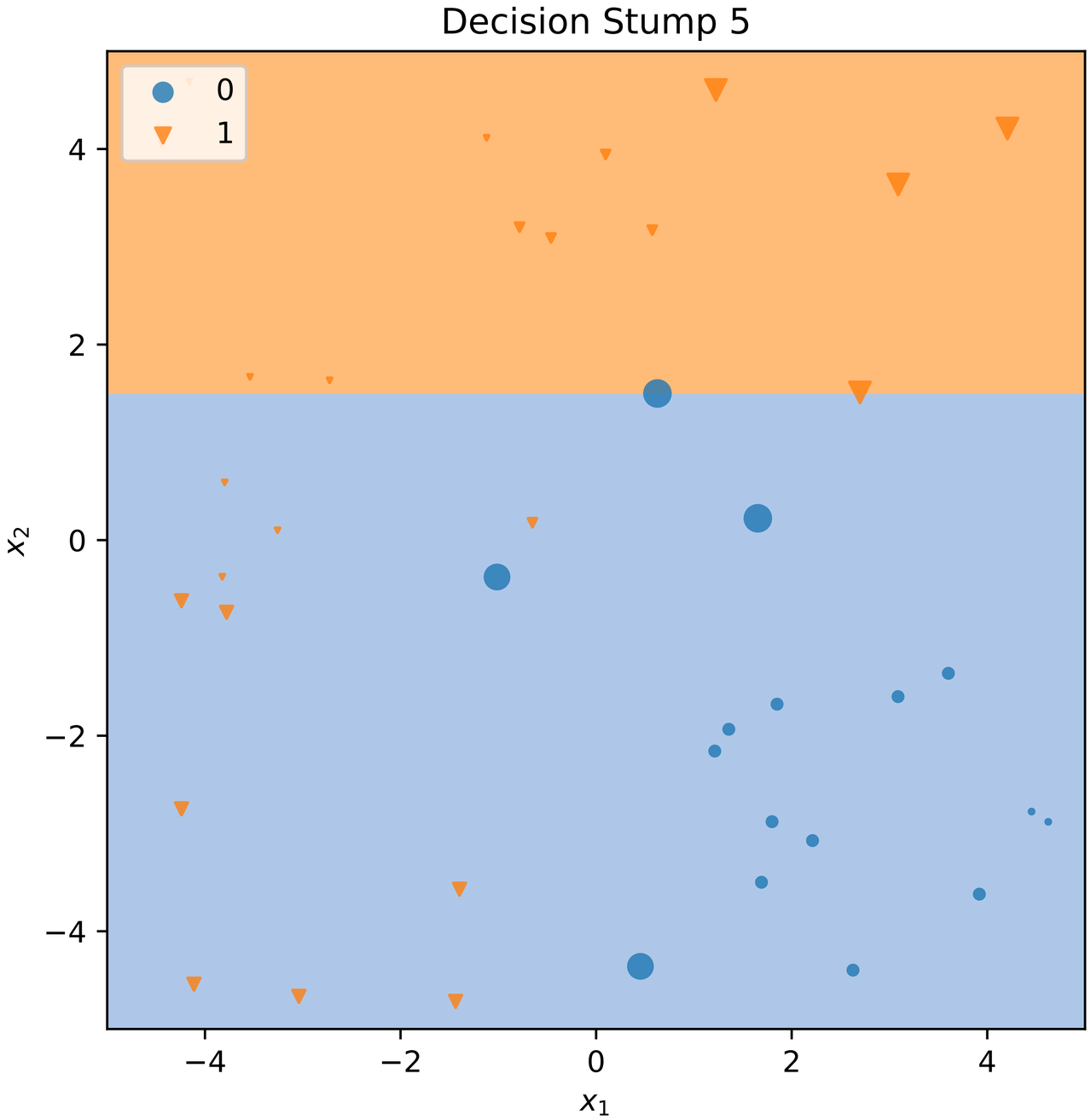

Weighted Samples 3
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1

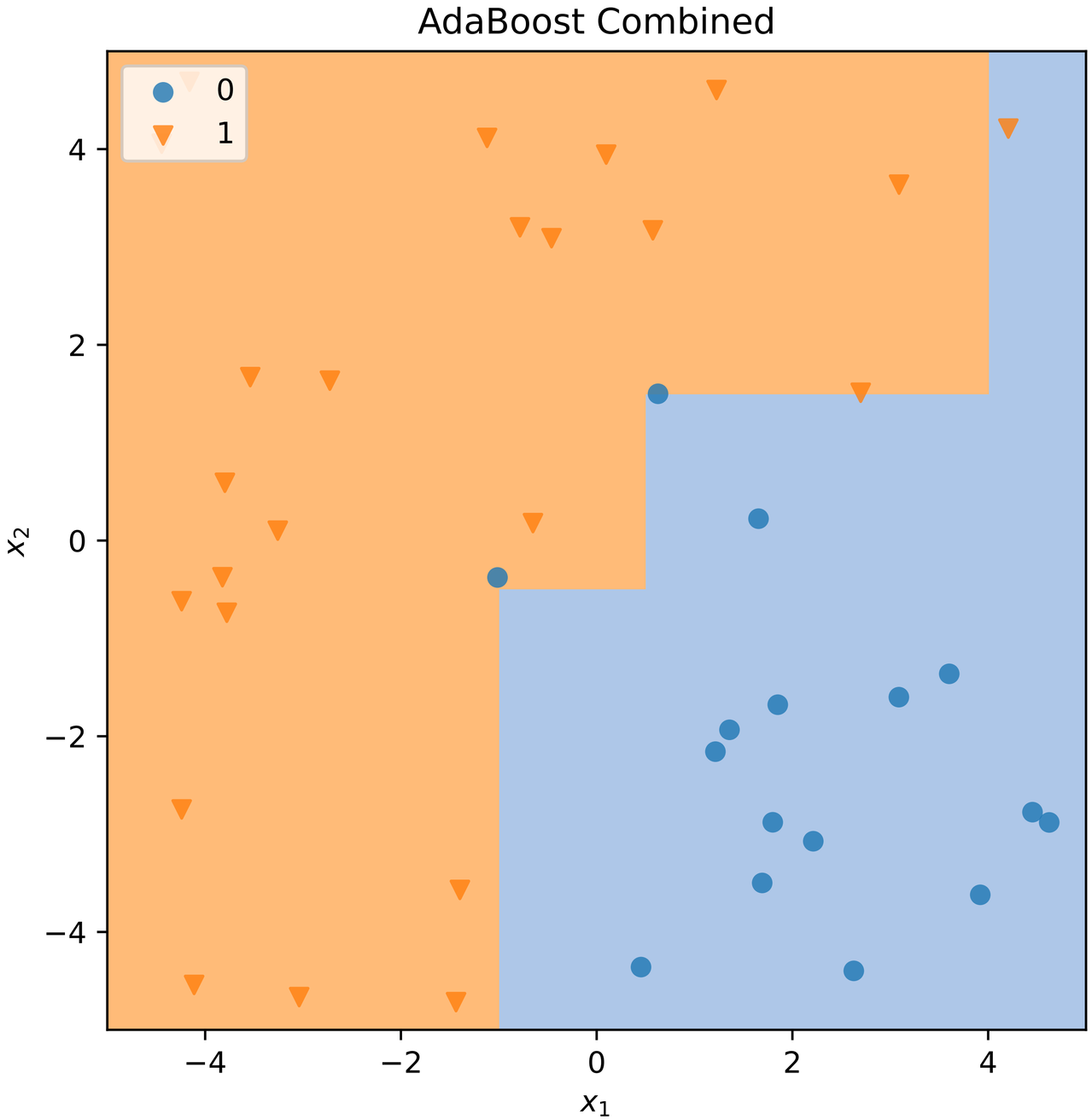

Decision Stump 3
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1


Weighted Samples 4
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1


Decision Stump 4
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1


Weighted Samples 5
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1


Decision Stump 5
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1


AdaBoost Combined
0
1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1

Why does it work?

- AdaBoost performs a forward stagewise fit of an additive prediction model

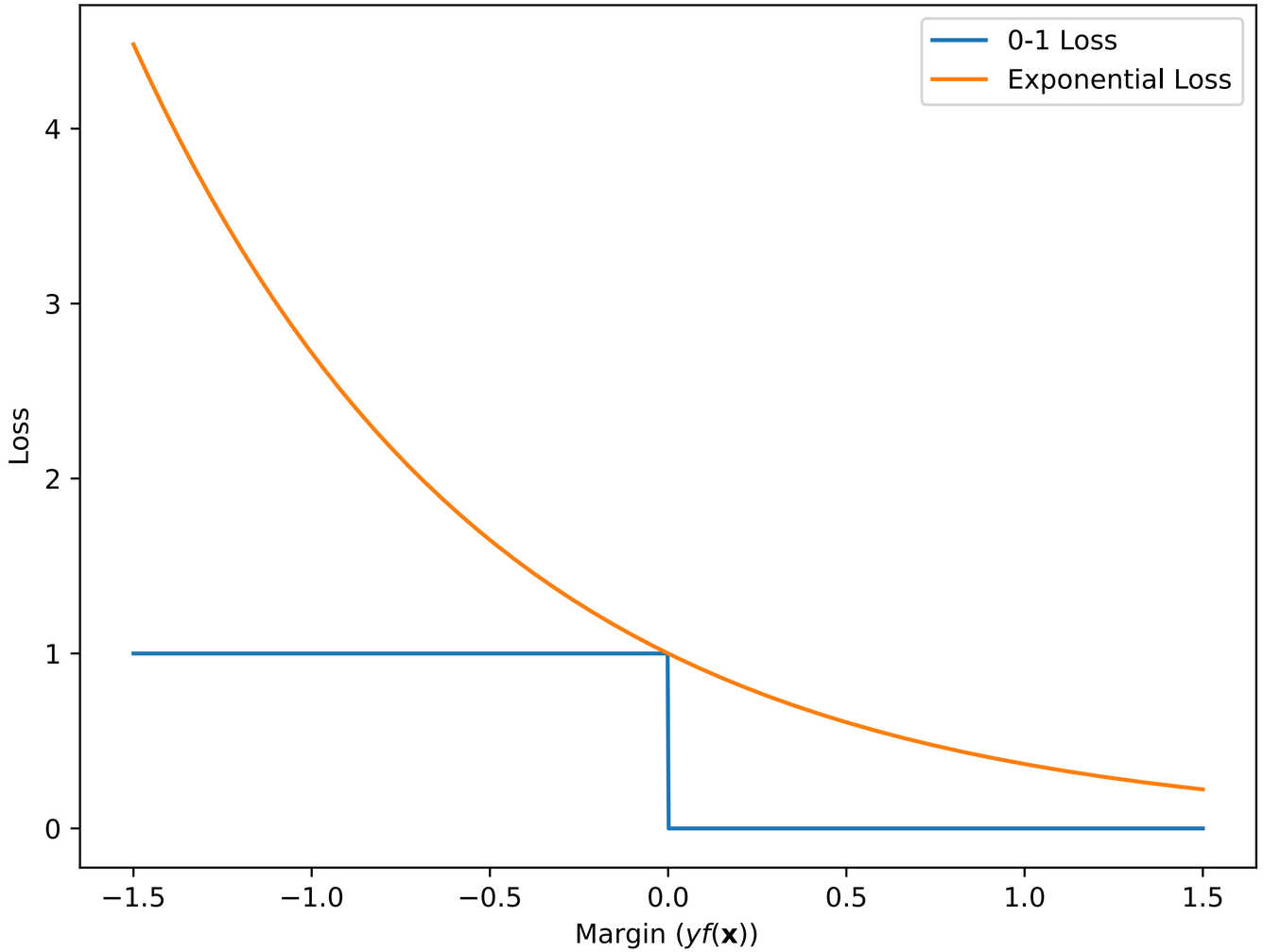
$$\hat{y} = \operatorname{sgn} \sum_{t=1}^{k} \alpha_t h_t(\mathbf{x}) = \operatorname{sgn} f_k(\mathbf{x})$$

- To minimise the exponential loss

$$L(y, f(\mathbf{x})) = e^{-yf(\mathbf{x})}$$

Exponential loss

- (Another) smooth proxy for 0-1 loss
- Sensitive to margin
- More negative margin
  = more firmly wrong
  = higher loss



Adding a learner

- Given existing ensemble

$$f_{k-1}(\mathbf{x}) = \sum_{t=1}^{k-1} \alpha_t h_t(\mathbf{x})$$

- Choose optimal weight α_k and classifier h_k to add such that

$$\alpha_k, h_k = \underset{\alpha,h}{\operatorname{argmin}} \sum_{i}^{n} e^{-y_i(f_{k-1}(\mathbf{x}_i)+\alpha h(\mathbf{x}_i))}$$

Justifying the weights

$$\sum_{i}^{n} e^{-y_i(f_{k-1}(\mathbf{x}_i)+\alpha h(\mathbf{x}_i))}$$

Justifying the weights

$$\sum_{i}^{n} e^{-y_i\left(f_{k-1}(\mathbf{x}_i)+\alpha h(\mathbf{x}_i)\right)}$$

Split sum in exponent to product

Justifying the weights

$$\sum_{i}^{n} e^{-y_i(f_{k-1}(\mathbf{x}_i)+\alpha h(\mathbf{x}_i))}$$

$$= \sum_{i}^{n} e^{-y_i f_{k-1}(\mathbf{x}_i)} e^{-y_i \alpha h(\mathbf{x}_i)}$$

Justifying the weights

$$\sum_i^n e^{-y_i(f_{k-1}(\mathbf{x}_i)+\alpha h(\mathbf{x}_i))}$$

$$= \sum_i^n e^{-y_i f_{k-1}(\mathbf{x}_i)} e^{-y_i \alpha h(\mathbf{x}_i)}$$

Independent of h and α

Justifying the weights

$$\sum_{i}^{n} e^{-y_i(f_{k-1}(\mathbf{x}_i)+\alpha h(\mathbf{x}_i))}$$

$$= \sum_{i}^{n} e^{-y_i f_{k-1}(\mathbf{x}_i)} e^{-y_i \alpha h(\mathbf{x}_i)}$$

$$= \sum_{i}^{n} w_i e^{-y_i \alpha h(\mathbf{x}_i)}$$

where: $w_i = e^{-y_i f_{k-1}(\mathbf{x}_i)}$

Optimal classifier

$$\sum_i^n w_i e^{-y_i \alpha h(\mathbf{x}_i)}$$

assume: $\alpha > 0$

Optimal classifier

$$\sum_{i}^{n} w_i e^{-y_i \alpha h(\mathbf{x}_i)}$$

assume: $\alpha > 0$

Separate the correct and incorrect cases into separate sums

Optimal classifier

$$\sum_{i}^{n} w_i e^{-y_i \alpha h(\mathbf{x}_i)} \qquad \text{assume: } \alpha > 0$$

$$= e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad + \quad e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i$$

Optimal classifier

$$\sum_{i}^{n} w_i e^{-y_i \alpha h(\mathbf{x}_i)} \qquad \text{assume: } \alpha > 0$$

$$= e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad + \quad e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i$$

Add incorrect cases to this sum…

Optimal classifier

$$\sum_i^n w_i e^{-y_i \alpha h(\mathbf{x}_i)} \qquad \text{assume: } \alpha > 0$$

$$= e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad + \quad e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i$$

…and subtract the same weight from those same cases in this sum

Optimal classifier

$$\sum_{i}^{n} w_i e^{-y_i \alpha h(\mathbf{x}_i)} \qquad \text{assume: } \alpha > 0$$

$$= e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad + \quad e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i$$

$$= e^{-\alpha} \sum_{1}^{n} w_i \quad + \quad (e^{\alpha} - e^{-\alpha}) \sum_{1}^{n} w_i \mathbf{1}(y_i \neq h(\mathbf{x}_i))$$

Optimal classifier

$$\sum_i^n w_i e^{-y_i \alpha h(\mathbf{x}_i)} \qquad \text{assume: } \alpha > 0$$

$$= e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad + \quad e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i$$

$$= e^{-\alpha} \sum_1^n w_i \quad + \quad \left(e^{\alpha} - e^{-\alpha}\right) \sum_1^n w_i \, \mathbf{1}(y_i \neq h(\mathbf{x}_i))$$

Don't depend on *h*

Optimal classifier

$$\sum_i^n w_i e^{-y_i \alpha h(\mathbf{x}_i)} \qquad \text{assume: } \alpha > 0$$

$$= e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad + \quad e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i$$

$$= e^{-\alpha} \sum_1^n w_i \quad + \quad (e^{\alpha} - e^{-\alpha}) \sum_1^n w_i \mathbf{1}(y_i \neq h(\mathbf{x}_i))$$

$$h_k = \operatorname*{argmin}_h \sum_1^n w_i \mathbf{1}(y_i \neq h(\mathbf{x}_i))$$

Optimal classifier

$$\sum_i^n w_i e^{-y_i \alpha h(\mathbf{x}_i)} \qquad \text{assume: } \alpha > 0$$

$$= e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad + \quad e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i$$

$$= e^{-\alpha} \sum_1^n w_i \quad + \quad (e^{\alpha} - e^{-\alpha}) \sum_1^n w_i \mathbf{1}(y_i \neq h(\mathbf{x}_i))$$

$$h_k = \underset{h}{\text{argmin}} \sum_1^n w_i \mathbf{1}(y_i \neq h(\mathbf{x}_i))$$

ie: optimal classifier is just the one that minimises the weighted error

Optimal prediction weight

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}} \; e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad + \quad e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i$$

Optimal prediction weight

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}} \left(e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i + e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \right)$$

Differentiate and set to zero

Optimal prediction weight

$$\alpha_k = \underset{\alpha}{\text{argmin}} \; e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad + \quad e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i$$

$$\frac{d(...)}{d\alpha} = e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad - \quad e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad = \quad 0$$

Optimal prediction weight

$$\alpha_k = \underset{\alpha}{\text{argmin}} \; e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad + \quad e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i$$

$$\frac{d(...)}{d\alpha} = e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad - \quad e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad = \quad 0$$

$$e^{\alpha_k} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad = \quad e^{-\alpha_k} \sum_{y_i = h(\mathbf{x}_i)} w_i$$

Optimal prediction weight

$$\alpha_k = \underset{\alpha}{\text{argmin}}\ e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad + \quad e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i$$

$$\frac{d(\ldots)}{d\alpha} = e^{\alpha} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad - \quad e^{-\alpha} \sum_{y_i = h(\mathbf{x}_i)} w_i \quad = \quad 0$$

$$e^{\alpha_k} \sum_{y_i \neq h(\mathbf{x}_i)} w_i \quad = \quad e^{-\alpha_k} \sum_{y_i = h(\mathbf{x}_i)} w_i$$

Take log of both sides

Optimal prediction weight

$$\alpha_k + \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = -\alpha_k + \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right)$$

Optimal prediction weight

$$\alpha_k + \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = -\alpha_k + \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right)$$

$$2\alpha_k = \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right) - \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = \log\left(\frac{\sum_{y_i = h(\mathbf{x}_i)} w_i}{\sum_{y_i \neq h(\mathbf{x}_i)} w_i}\right)$$

Optimal prediction weight

$$\alpha_k + \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = -\alpha_k + \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right)$$

$$2\alpha_k = \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right) - \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = \log\left(\frac{\sum_{y_i = h(\mathbf{x}_i)} w_i}{\sum_{y_i \neq h(\mathbf{x}_i)} w_i}\right)$$

Denominator is just the weighted error rate, ϵ

Optimal prediction weight

$$\alpha_k + \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = -\alpha_k + \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right)$$

Weights sum to 1, so so numerator is 1 - ϵ

$$2\alpha_k = \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right) - \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = \log\left(\frac{\sum_{y_i = h(\mathbf{x}_i)} w_i}{\sum_{y_i \neq h(\mathbf{x}_i)} w_i}\right)$$

Optimal prediction weight

$$\alpha_k + \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = -\alpha_k + \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right)$$

$$2\alpha_k = \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right) - \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = \log\left(\frac{\sum_{y_i = h(\mathbf{x}_i)} w_i}{\sum_{y_i \neq h(\mathbf{x}_i)} w_i}\right)$$

$$\alpha_k = \frac{1}{2}\log\left(\frac{1-\epsilon}{\epsilon}\right)$$

Optimal prediction weight

$$\alpha_k + \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = -\alpha_k + \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right)$$

$$2\alpha_k = \log\left(\sum_{y_i = h(\mathbf{x}_i)} w_i\right) - \log\left(\sum_{y_i \neq h(\mathbf{x}_i)} w_i\right) = \log\left(\frac{\sum_{y_i = h(\mathbf{x}_i)} w_i}{\sum_{y_i \neq h(\mathbf{x}_i)} w_i}\right)$$

$$\alpha_k = \frac{1}{2}\log\left(\frac{1-\epsilon}{\epsilon}\right)$$

NB: we assumed $\alpha > 0$, which requires $\epsilon < 0.5$

If learners are less accurate than 50% then they will make performance worse

Notes

- AdaBoost is efficient and easy to use
- Relatively turnkey — need to choose k, but not super critical
- Pruning can improve efficiency and also reduce overfitting

- This concept of taking a weighted sum over classifiers will crop up again when we talk about neural networks
- Next week we'll see a model that is both linear and non-parametric, which uses basis expansion in new and exciting ways, and which depends crucially on the margin $y\, f(\boldsymbol{x})$