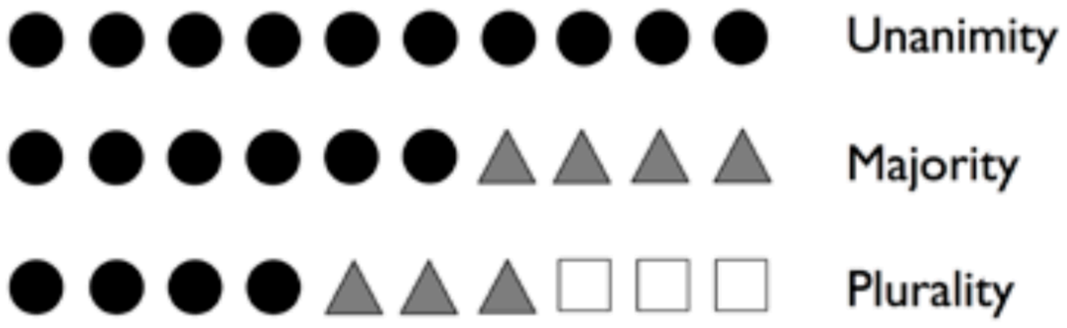
Week 7 - ML

Start Date	@15 December 2025
Weeks	Week 7

Combining Different Models For Ensemble Learning

Learning With Ensemble

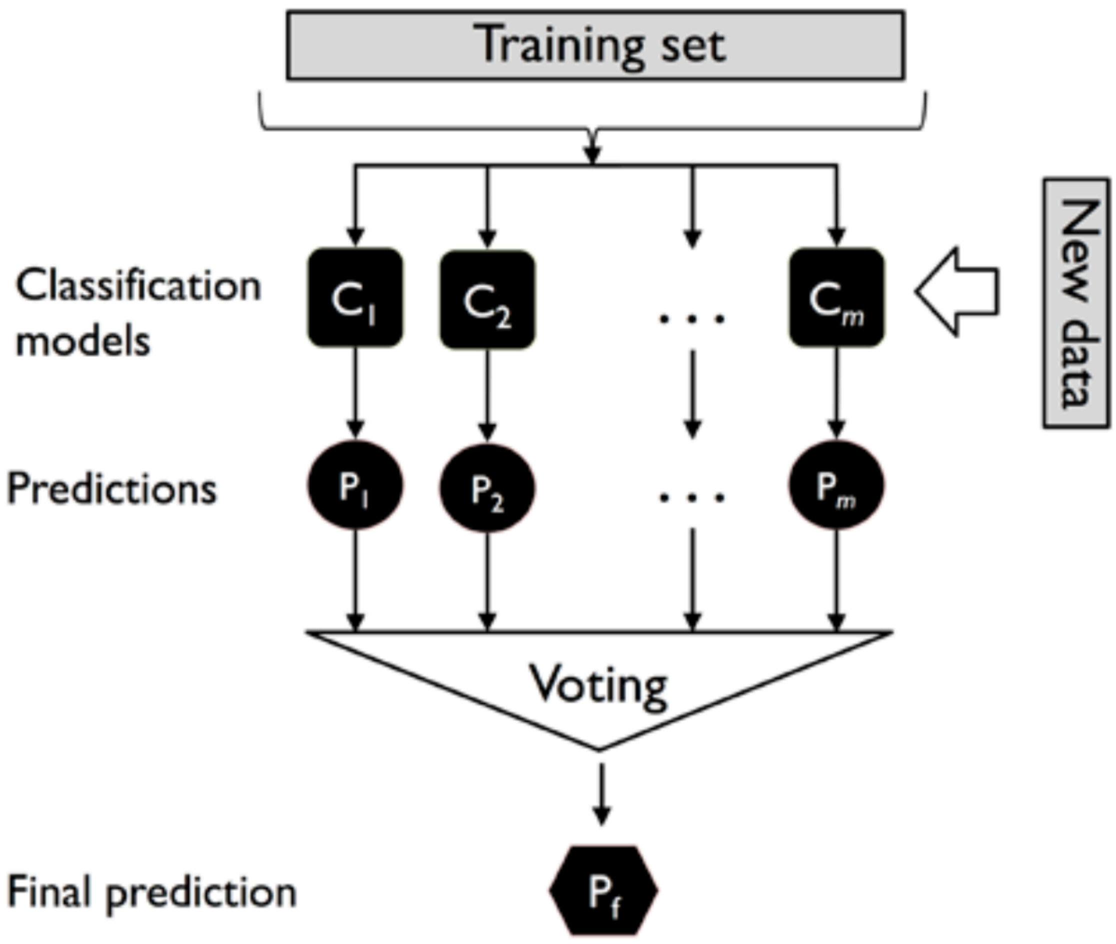
- Ensemble method → Combine classifiers to meta-classifier → Better generalisation performance
- Majority Voting Principle → Class label with more than 50% of votes → refers to binary class settings only
    - Generalise to multi-class settings → Plurality Voting (UK → 'Absolute' + 'Relative' Voting) → Class bale with most votes



Different voting concepts

- Training dataset → Training m different classifiers
- Depending on technique
    - Use different classification algorithm
    - Decision Trees, SVM, Logistic Regression

- Can also use same base classification algorithm, which fits different subsets of training datasets



General ensemble approach

- Predict class label with Majority or Plurality voting, then combine class labels classifier, C_j + selected class label, $\hat{y}$

$$\hat{y} = mode\,\{C_1(x), C_2(x), \cdots, C_m(x)\}$$

- This shows the most frequent event. (mode from stats)
- Binary Classification + Majority Vote:

$$C(x) = sign\left[\sum_{j=1}^{m} C_j(x)\right] = \begin{cases} 1, & \text{if } \sum_{j=1}^{m} C_j(x) \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

- Ensemble method works better than individual classifiers → Proven with combinatorics

- Assumptions for Binary Classification task:
  - n-base classifiers → equal error rate, ε
  - Classifiers independent + error rate not correlated

$$P(y_g \geq k) = \sum_{k}^{n} \binom{n}{k} \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{\text{ensemble}}$$

- $\binom{n}{k}$ = Binomial Coefficient
- Error probability of ensemble of base classifiers as probability mass function of binomial distribution
- Example = 11 Classifiers ($n = 11$) + Error rate 0.25 ($\varepsilon = 0.25$)

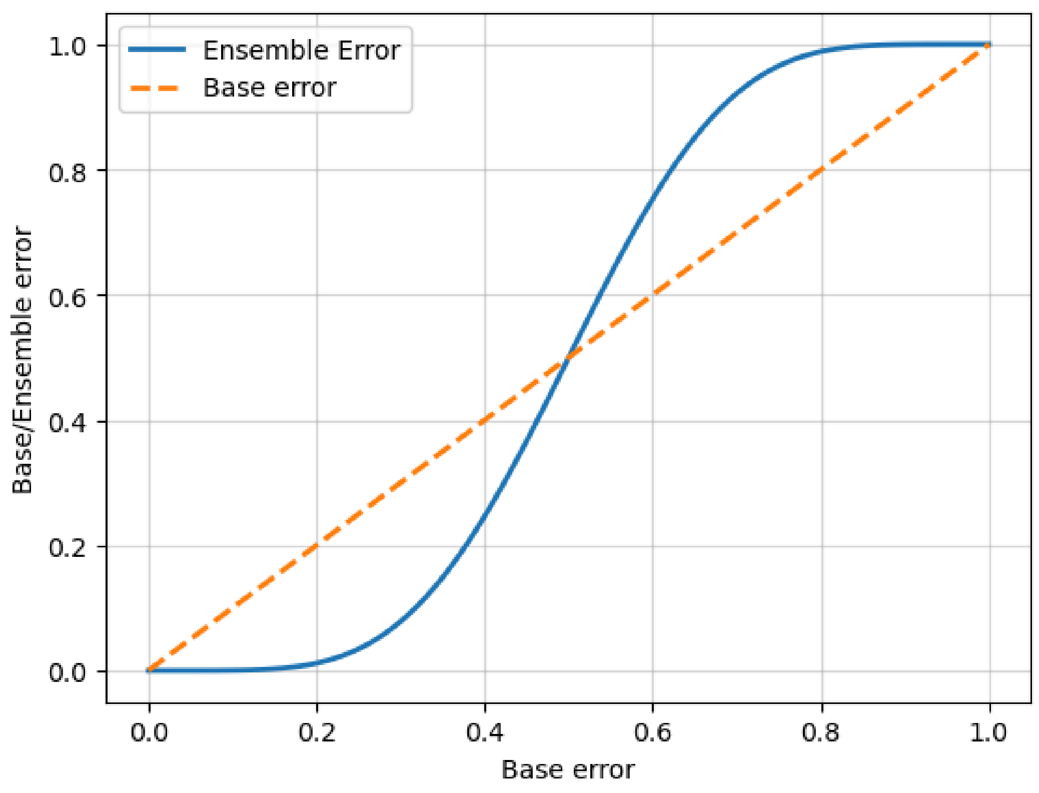
$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1-0.25)^{11-k} = 0.034$$

- Error rate lower than of each individual classifier

Binomial Coefficient

- Number of ways we choose k from set of size n → "n choose k"

$$\frac{n!}{(n-k)!k!}$$



Plot of the ensemble error versus the base error

- Error probability of ensemble always better

Combining Classifiers via Majority Vote

Single Majority Vote Classifier

$$\hat{y} = arg_i max \sum_{j=1}^{m} w_j \chi_A \left(C_j(x) = i\right)$$

- w_j = Weight associated with base classifier, C_j
- $\hat{y}$ = Predicted class label of the ensemble
- A = Set of unique class labels
- χ_A = Characteristic or inidicator function

- For equal weights → simplified equation:

$$\hat{y} = mode\left\{C_1(x), C_2(x), \cdots, C_m(x)\right\}$$

- Example:
    - $C_1(x) = 0, C_2(x) = 0, C_3(x) = 1$
    - $\hat{y} = mode\,\{0, 0, 1\} = 0$
    - Weights = $C_1, C_2 = 0.2, C_3 = 0.6$

$$\hat{y} = arg_i max[0.2 \times i_0 + 0.2 \times i_0, 0.6 \times i_1] = 1$$

- $3 \times 0.2 = 0.6$ ⇒ C_3 has three times the weight than C_1 C_2 → $\hat{y} = mode\,\{0, 0, 1, 1, 1\} = 1$
- Modified version of Majority Vote for predicting class labels (using `predict_proba` method from Logistic Regression):

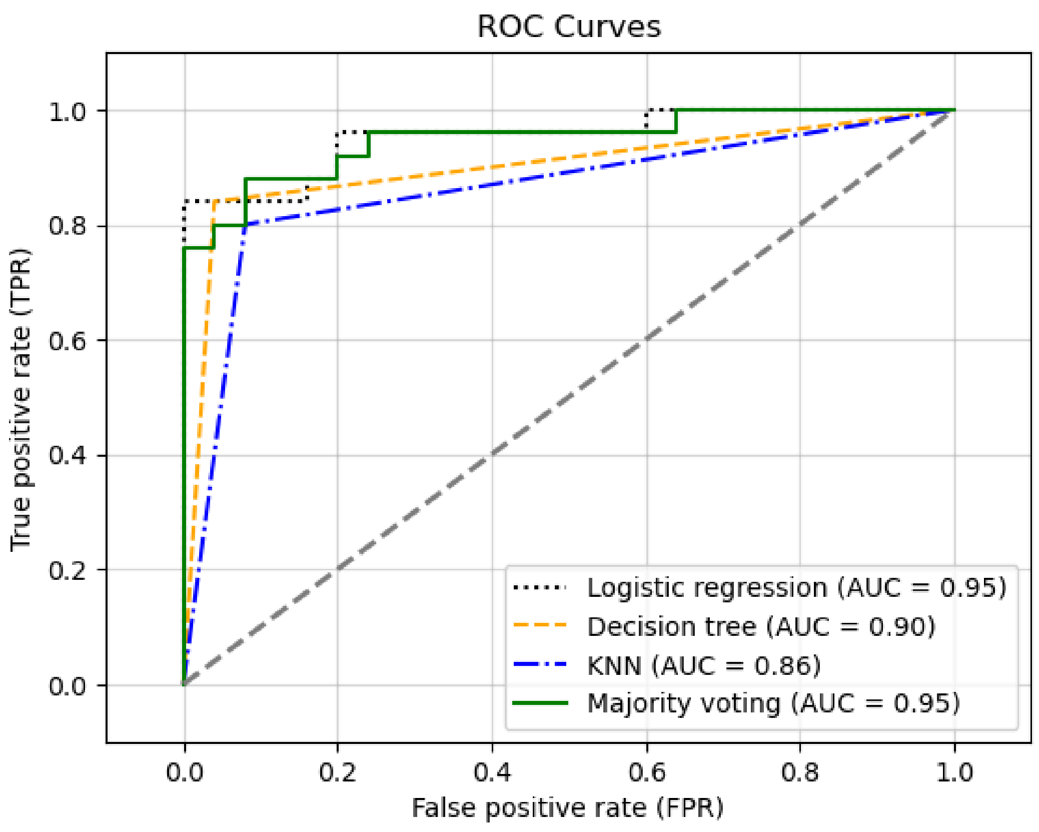
$$\hat{y} = arg_i max \sum_{j=1}^{m} w_j P_{ij}$$

- P_{ij} = Predicted probability of jth class label i

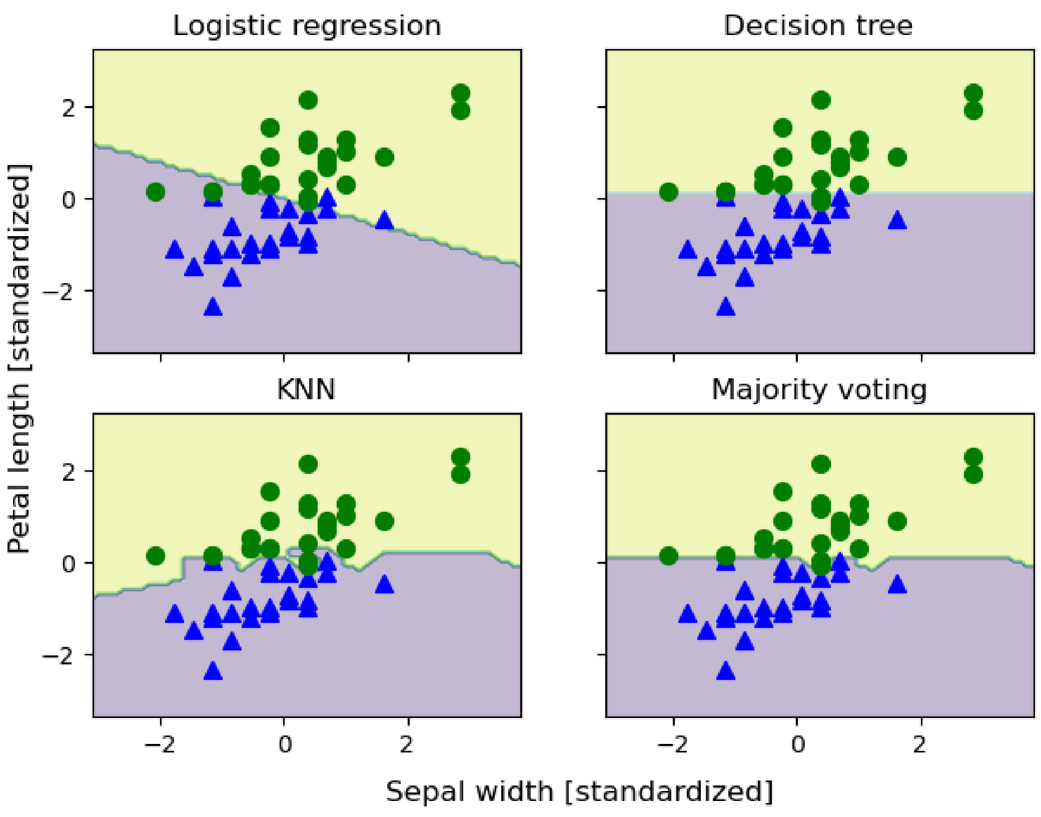
- Binary Classification problem + same weights

$$C_1(x) \rightarrow [0.9, 0.1], \quad C_2(x) \rightarrow [0.8, 0.2], \quad C_3(x) \rightarrow [0.4, 0.6]$$

$$\begin{gathered} p(\hat{y}_0 \mid x) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58 \\ p(\hat{y}_1 \mid x) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42 \\ \hat{y} = \arg\max_i \left[p(\hat{y}_0 \mid x), p(\hat{y}_1 \mid x)\right] = 0 \end{gathered}$$



The ROC curve for the different classifiers



The decision boundaries for the different classifiers

Stacking (Ensemble Methods)

- **David H. Wolpert** → "Stacked Generalization" 1992

Input: Training data $D = \{\chi_i, y_i\}_{i=1}^{n} \, (x_i \in R^n, y_i \in \gamma)$

Output: Ensemble classifier H

Step 1: Learn first-level classifiers

```
for $t \leftarrow 1$ to $T$ do
    Learn a base classifier $h_t$ based on $D$
end for
```

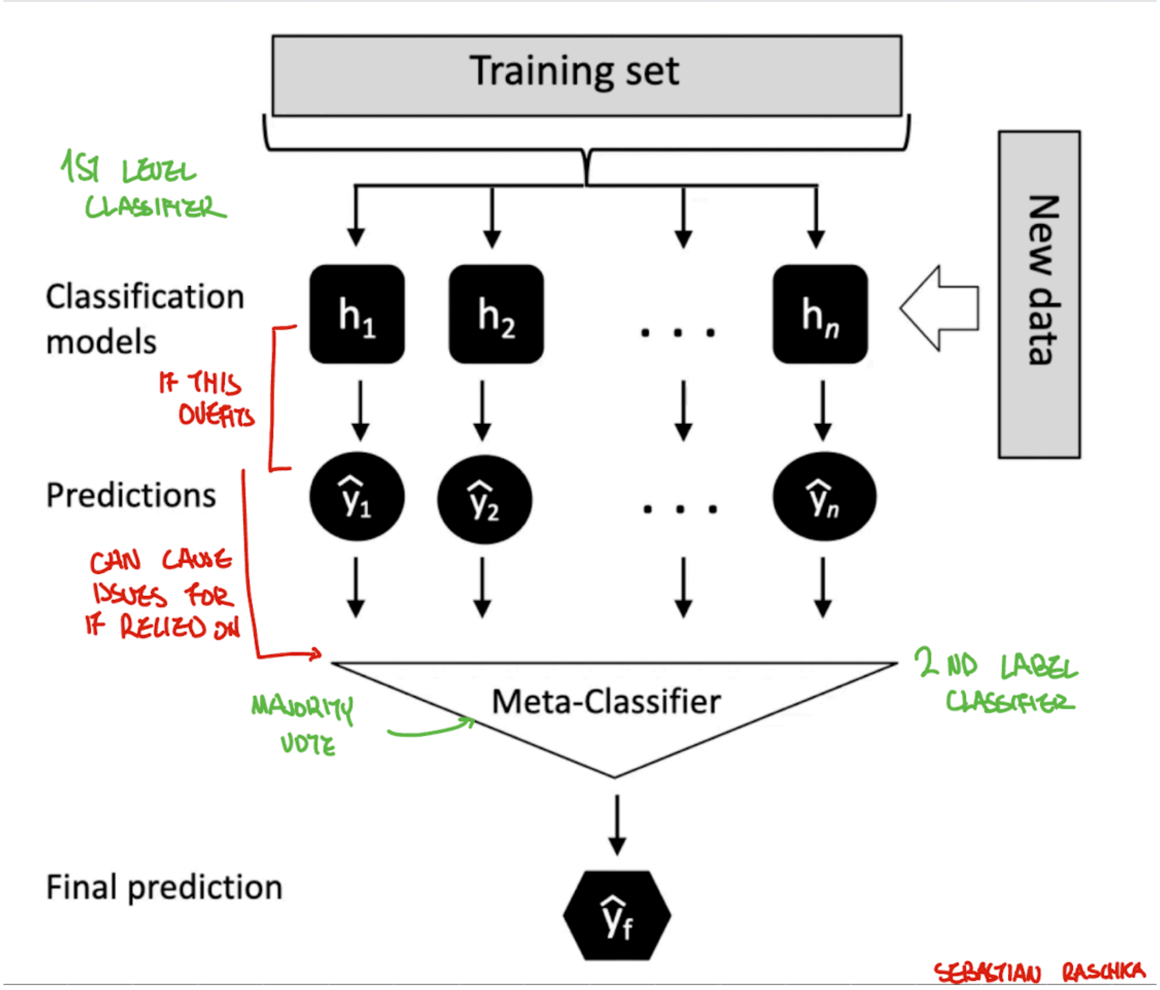
Step 2: Construct new data sets from D

```
for $i \leftarrow 1$ to $n$ do
    Construct a new data set that contains $\{x'_i, y_i\}$ where $x' = \{h_1(x_i), h_2(x_i), \cdots, h_T(x_i)\}$
        • $x'$ = Modified feature vector
        • $\{h_1(x_i), h_2(x_i), \cdots, h_T(x_i)\}$ = predicted class labels via scikit-learn
end for
```

⭐ **The first two steps are the same as Majority Voting**

Step 3: Learn a second-level classifier → Learn new classifier based on predictions

```
Learn a new classifier $h'$ based on newly constructed data set.
return $H(x) = h'(h_1(x), h_2(x), \cdots, h_T(x))$
```



Stacking

- **Problem with stacking??**
    - Prone to overfitting…
- You can improve stacking with cross-validation
- Stacking → K-fold classification for 2nd label classifier
    - Use k predictions for 2nd label classifier

Modified Stacking Algorithm

Input + Output are the same

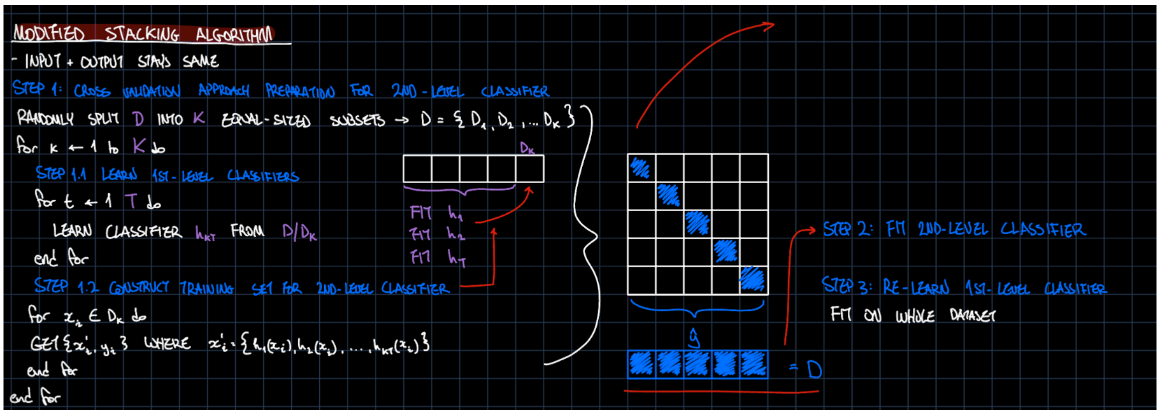
Step 1: Cross validation approach preparation for 2nd-level classifier

Randomly split D into k equal-sized subsets → $D = \{D_1, D_2, \cdots, D_k\}$

for $k \leftarrow 1$ to K do

Step 1.1: Learn 1st-level classifiers

```
        for $t \leftarrow 1\ T$ do
            Learn classifier $h_{KT}$ from $D/D_k$
        end for
        Step 1.2: Construct training set for 2nd-level classifier
        for $x_i \in D_k$ do
            Get $\{x_i', y_i\}$ where $x_i' = \{h_1(x_i), h_2(x_i), \cdots, h_{KT}(x_i)\}$
        end for
    end for
Step 2: Fit 2nd-level classifier
Step 3: Re-learn 1st-level classifier
```



MODIFIED STACKING ALGORITHM

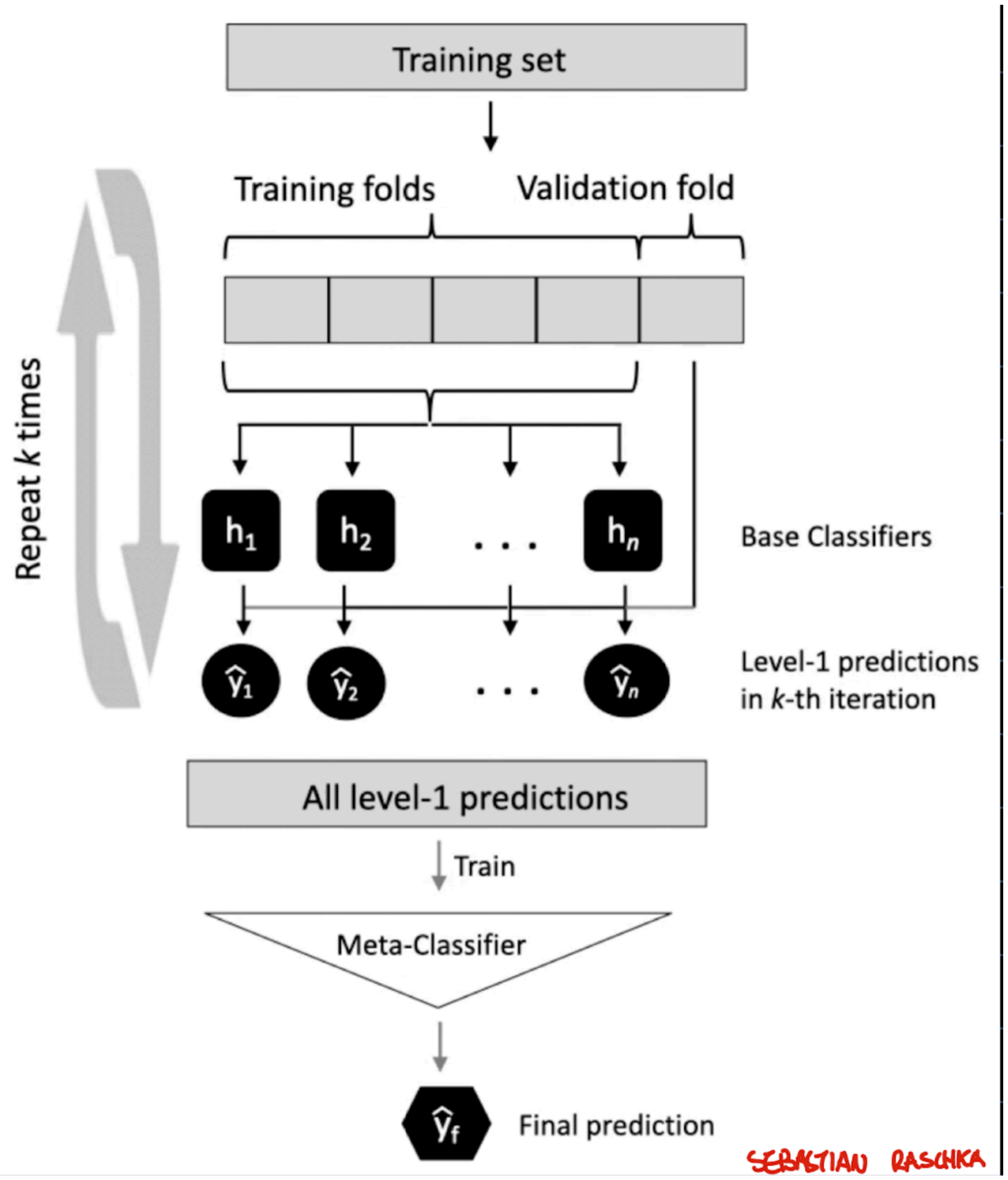
- INPUT + OUTPUT STAYS SAME

```
STEP 1: CROSS VALIDATION APPROACH PREPARATION FOR 2ND-LEVEL CLASSIFIER
RANDOMLY SPLIT D INTO K EQUAL-SIZED SUBSETS → $D = \{D_1, D_2, \dots D_K\}$
for $k \leftarrow 1$ to $K$ do
    STEP 1.1 LEARN 1ST-LEVEL CLASSIFIERS
    for $t \leftarrow 1\ T$ do
        LEARN CLASSIFIER $h_{KT}$ FROM $D/D_K$
    end for
    STEP 1.2 CONSTRUCT TRAINING SET FOR 2ND-LEVEL CLASSIFIER
    for $x_i \in D_K$ do
    GET $\{x_i', y_i\}$ WHERE $x_i' = \{h_1(x_i), h_2(x_i), \dots, h_{KT}(x_i)\}$
    end for
end for
```

STEP 2: FIT 2ND-LEVEL CLASSIFIER

STEP 3: RE-LEARN 1ST-LEVEL CLASSIFIER

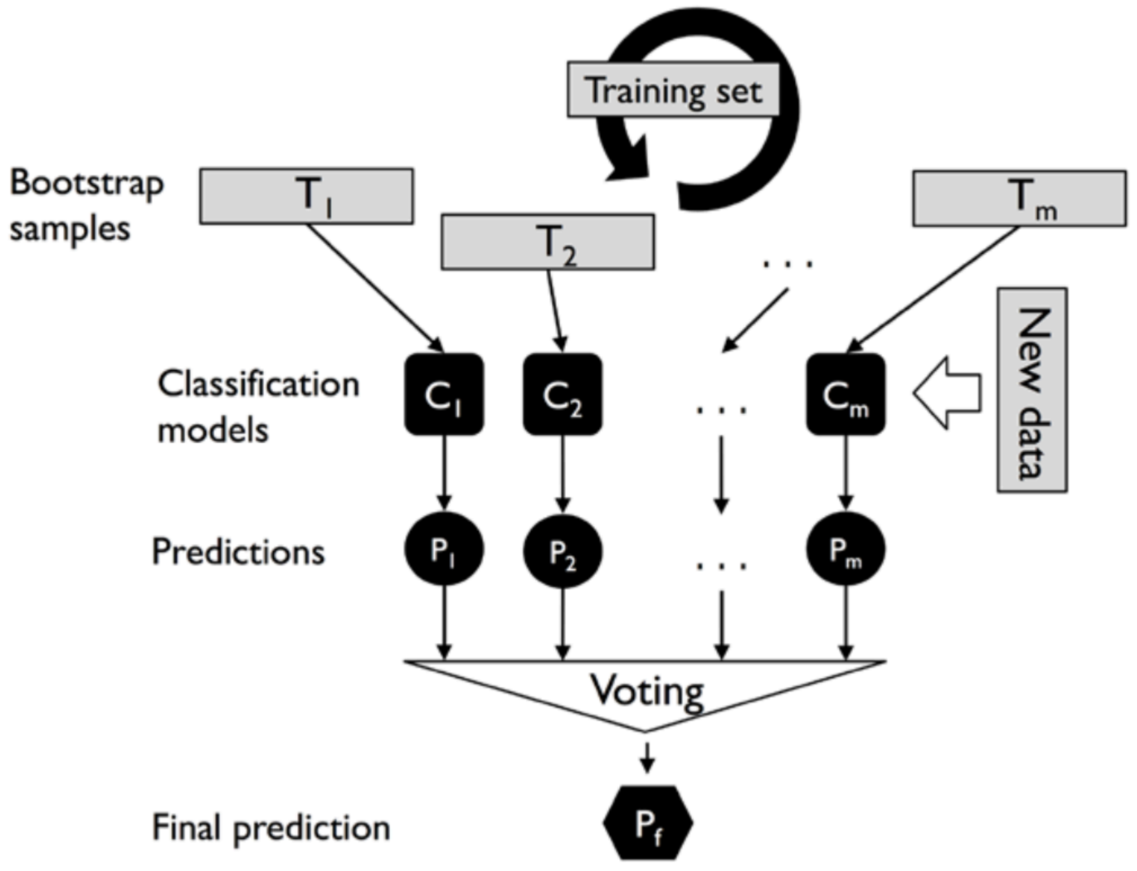
FIT ON WHOLE DATASET



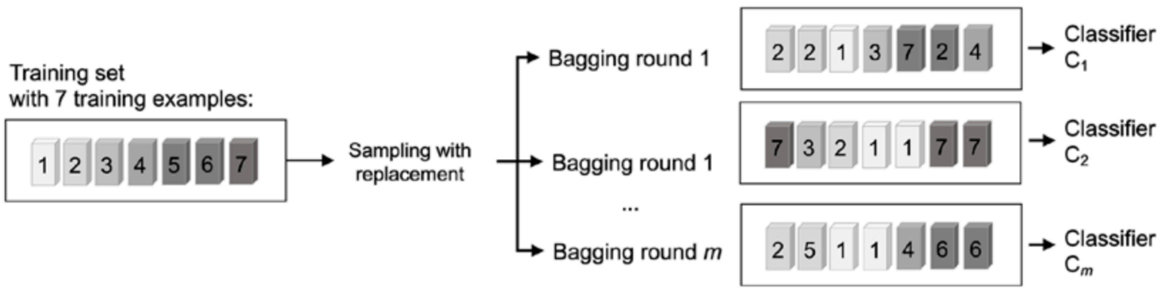
Modified Stacking

Bagging - Ensemble Classifiers From Bootstrap Samples

- Closely related to `MajorityVoteClassifier`
- Instead of same training dataset to fit individual classifiers → Draw bootstrap samples (random samples with replacement) from initial training dataset
  - Called Bootstrap Aggregating



Bagging concept



Example of bagging

- Random samples obtained via Bagging → Bagging round
- Each subset contains some duplicates + some original examples don't appear
  - Sampling with replacement
- Individual classifiers fit to bootstrap samples and then predictions combined with Majority Vote
- Bagging related to Random Forest Classifier

    - Random Forest → Special example → Random feature subsets also used when fitting individual decision trees
- Complex classification tasks + dataset's high dimensionality can lead to overfitting in single decision trees → In these scenarios bagging comes in handy

⭐ **Bagging is an effective approach to reducing variance of model**

- However, it's ineffective in reducing model bias, if models too simple to capture trends in data
    - This is why we want to perform bagging on ensemble of classifiers with low bias → e.g. Unpruned Decision Trees

Adaptive Boosting - Leveraging weak learners

- **Adaptive boosting (AdaBoost)**
    - **Robert E. Shapire (1990)** → 'The Strength of Weak Learnability'
- Boosting - Ensemble consists of very simple base classifiers → Referred to as 'weak learners' → Slight edge over random guessing
    - **Key Concept →** Focus on training examples that are hard to classify
        - Weak learners subsequently learn from misclassified examples to improve performance of ensemble

How Adaptive Boosting works:

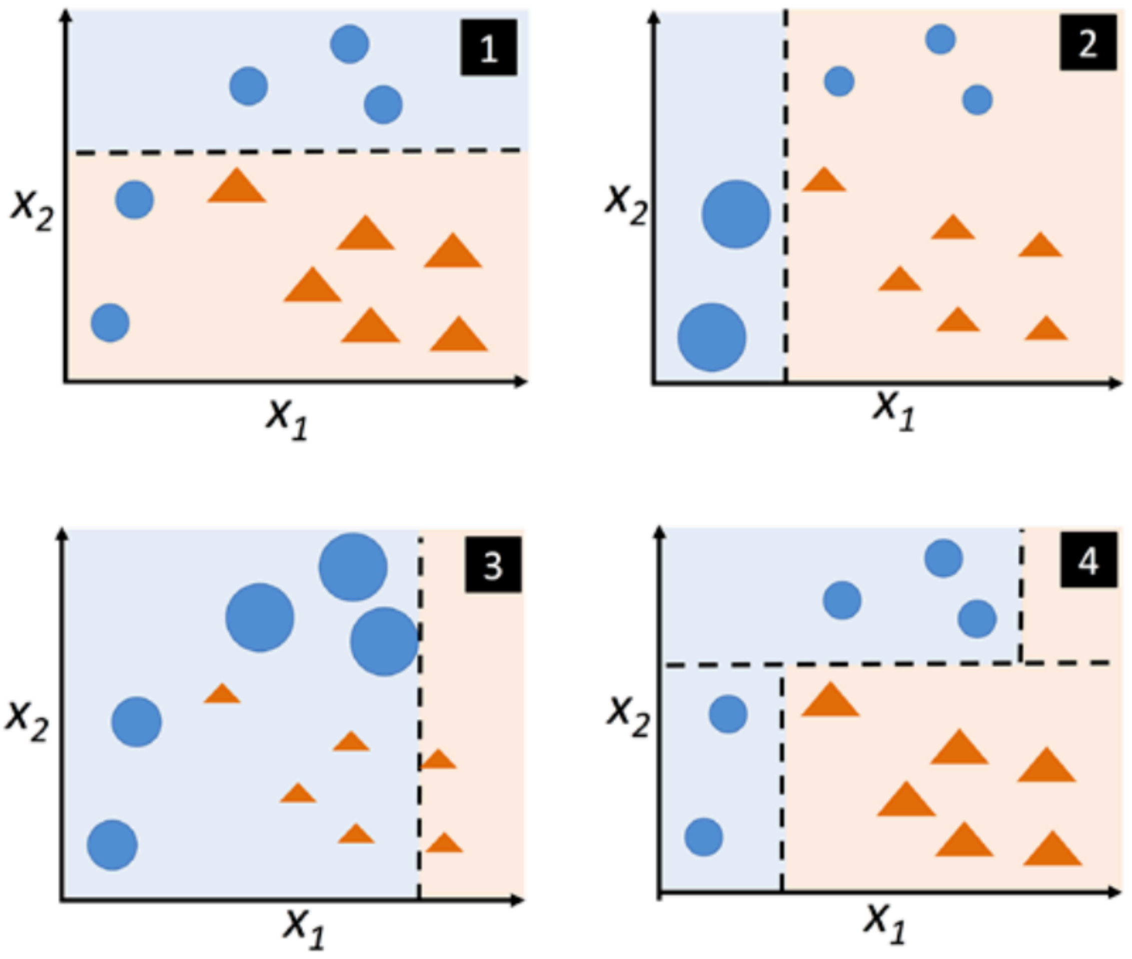
⭐ **Initial formulation - Algorithm uses random subsets of training examples from training dataset without replacement**

Four key steps:

1. Draw random subset (samples) from training examples, d without replacement from training dataset, D for weaker learner C_1
2. Second random training subset, d_2 without replacement and 50% of examples from previously misclassified to train weaker learner C_2

3. Find training examples, d_2 in training dataset, D which $C_1 + C_2$ disagree with, to train C_3
4. Combine C_1, C_2, C_3 via Majority Voting

- **Leo Breiman** → Boosting can lead to decrease in bias + variance - compared to bagging
  - In practice, AdaBoost known for high variance → tend to overfit training data
- AdaBoost - uses the complete training dataset to train weak learners → training examples reweighed after each iteration to learn from mistakes



The concept of AdaBoost to improve weak learners

AdaBoost steps:

1. Set weight vector, w, to uniform weights, $\sum_i w_i = 1$

2. For j in m boosting rounds:
    a. Train weighted weak learner, $C_j = train(X, y, w)$
    b. Predict class labels, $\hat{y} = predict(C_j, X)$
    c. Compute weighted error rate, $\epsilon = w \cdot (\hat{y} \neq y)$
    d. Compute coefficient, $\alpha_j = 0.5 log(\frac{1-\epsilon}{\epsilon})$
    e. Update weights, $w := w \times exp(-\alpha_j \times \hat{y} \times y)$
    f. Normalise weights to sum to 1, $w := w / \sum_i w_i$
3. Compute final prediction, $\hat{y} = \left(\sum_{j=1}^{m} (\alpha_j \times predict(C_j, X) > 0 \right)$

⭐ **Considered bad practice to select a model based on the repeated usage of the test dataset.** **→ Generalisation performance may be overoptimistic**

⭐
- **Ensemble learning increases computational complexity compared to individual classifiers**
    - **In practice, ask yourself, is it worth the modest improvement in predictive performance for increased computational cost**

Boosting vs. Stacking

	Boosting	**Stacking**
Training Order	Sequential training	Parallel training
Core Goal	Reduce bias by repeatedly correcting mistakes	Exploit diversity - learn how to mix different strong models
New Model Fitting	Emphasises misclassified/large error samples	Each base model sees same data, meta-model trained on predictions
Combination Method (final predictions)	Weighted sum / large-error samples	Meta-model output

Parallelisation	Limited (Iteration dependant)	Base learners easily parallelisable
Sensitivity	Can overfit/noise-sensitive if not regularised	Can overfit if meta-model is too flexible → relies on good CV setup
Typical Use	Gradient Boosting Trees, XG Boost, AdaBoost	Competition/production 'blender' → Combine Trees, Linear Models, SVMs, Neural Nets, etc…

Gradient Boosting - Training Ensemble Based on Loss Gradients

⭐ **Gradient boosting is important → Forms basis of popular ml algo like XGBoost → Well known for kaggle competitions**

- Another variant of boosting concept → Training weaker learners
- Gradient boost fits decisions trees in an iterative fashion using prediction errors
    - Deeper than decision tree stumps and typically a maximum depth of 3 to 6 (max 8-64 leaf nodes)
    - Does not use prediction errors for assigning sample weights, instead it goes directly to form target variable fro fitting the next tree
    - Uses same global learning rate for each tree

Outline of Gradient Boost Algorithm

⭐ **Gradient boost is a general-purpose supervised learning method**

- Here we'll look gradient boost in a classification model (binary classification example)
- Builds series of trees, where each tree is fit on error (difference between label and predicted value)

- Each round, tree ensemble improves, which nudges the tree more in the right direction with small updates
- Updates based on loss gradient, that's how it got the gradient 'boosting' name
- **Step-by-step (general algorithm):**
  1. Initialise model to return constant prediction value. Decision tree root node (decision tree with single leaf node)

$$F_0(x) = arg_{\hat{y}}min \sum_{i=1}^{n} L(y_i, \hat{y})$$

- $\hat{y}$ = value returned by decision tree
- L = loss function
- n = n training examples

  2. For each tree $m = 11, \cdots, M$, M is user-specified total number of trees
     a. Compute difference between predicted value $F(x_i) = \hat{y}_i$ and class label y_i, sometimes called `Pseudo-Response` or `Pseudo-Residual`

     Formally → Write Pseudo-residual as negative of loss function with respect to predicted values:

$$r_{im} = \left[\frac{\partial(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$$

     - $F(x)$ = prediction of previous tree $F_{m-1}(x)$
     - **If it's the first round then the constant value from single node tree instead of prediction of the previous tree**

     c. Fit a tree to pesudo-residual r_{im}, R_{jm} to denote $j = 1, \cdots, J_m$ lead nodes of resulting tree in iteration m

     d. For each lead node R_{jm} → compute output value:

$$\gamma_{jm} = arg_{\gamma}min \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

- γ_{jm} = computed by

minimising
loss function

⭐ R_{jm} can contain more than one training example, hence summation

e. Update model by adding output value γ_m to previous tree:

$$F_m(x) = F_{m-1}(x) + \eta\gamma_m$$

Instead of adding full predicted values of current tree γ_m to previous tree F_{m-1}, scale γ_m by learning rate n (small step → typically between 0.01 - 1)

Gradient Boost for Classification

- Single training example, logistic loss:

$$L_i = -y_i log p_i + (1 - y_i) log(1 - p_i)$$

- $log(odds)$:

$$\hat{y} = log(odds) = log(\frac{p}{1-p})$$

- Use $log(odds)$ to rewrite equation:

$$L_i = log(1 + e^{\hat{y}_i}) - y_i\hat{y}_i$$

- Partial derivative of loss function with respect to $log(odds)$, $\hat{y}$:

$$\frac{\partial L_i}{\partial \hat{y}_i} = \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} - y_i = p_i - y_i$$

After this is complete, the next step is to now add the gradient boost steps:

1. Create root node, which will minimise logistic loss. Loss minimised if root node return $log(odds)$, $\hat{y}$
2. For each tree $m = 1, \cdots, M$, M is user-specified total number of trees

a. Convert $log(odds)$ into probability using logistic function (we used this in logistic regression):

$$p = \frac{1}{1+e^{-y}}$$

Then we compute pseudo-residual, which is the negative partial derivative of loss with respect to $log(odds)$

- This is the difference between class label and the predicted probability

b. Fit new tree to pseudo-residuals

c. For each R_{jm}, compute value γ_{jm}, which minimises loss function

$$\gamma_{jm} = arg_\gamma min \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$
$$= log(1 + e^{\hat{y_i}+\gamma}) - y_i(y_i \hat{+} \gamma)$$

This results in:

$$\gamma_{jm} = \frac{\Sigma_i y_i - p_i}{\Sigma_i p_i (i - p_i)}$$

- This is only for R_{jm}, and not full training set

d. Update model by adding gamma value from `2c` with learning rate η:

$$F_m(x) = F_{m-1}(x) + \eta\gamma_m$$

⭐ **After boosting algorithm is complete → predict class labels by thresholding probability values of final mode, $F_m(x) = 0.5$, like you would do in logistic regression**

- **However, boosting will produce non-linear decision boundaries unlike logistics regression**

XGBoost

- Gradient boosting is s sequential process, which is slow to train (inefficient time wise)
- The alternative is **XGBoost**
- XGBoost (Extreme Gradient Boost) → Proposed several tricks + approximations → speeds up the training process → used a lot in kaggle competitions