UNIVERSIDAD DE CHILE

Minería de Datos

Welcome to the Machine Learning class

Valentin Barriere

Universidad de Chile – DCC

CC5205, Fall 2025

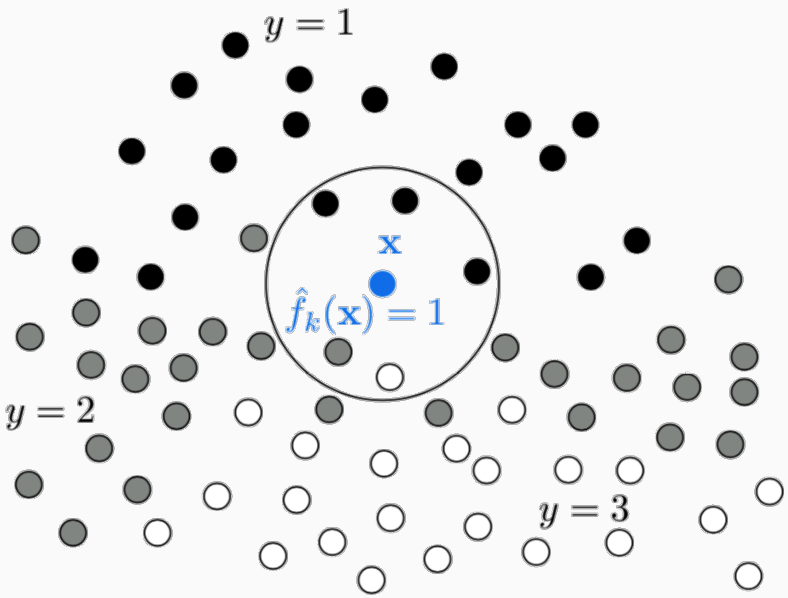
Supervised Learning – Classifiers

K-Nearest Neighbors

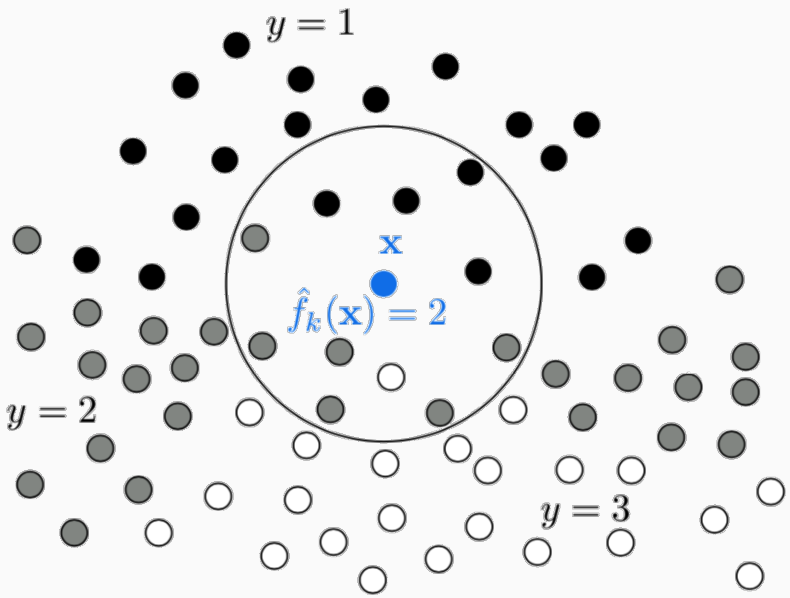
Outline : K-Nearest Neighbors

K-Nearest Neighbors

We look at the closest data points in the feature space to classify our new instance:

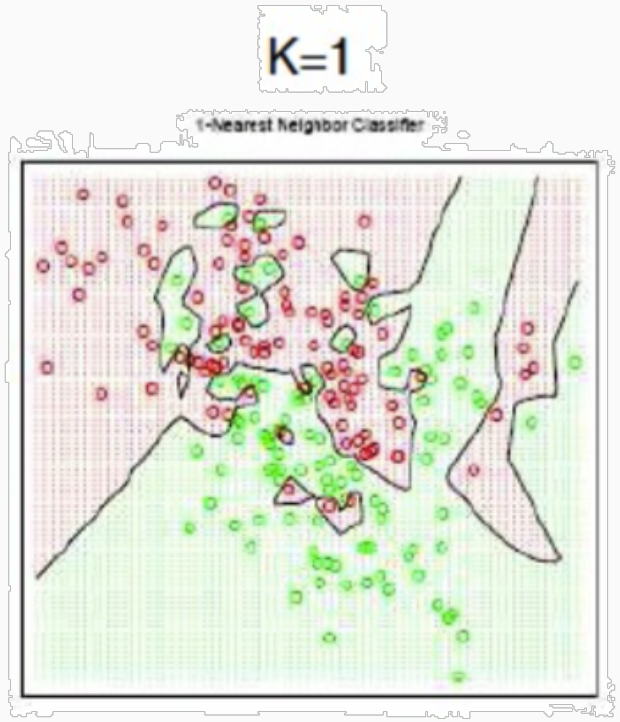


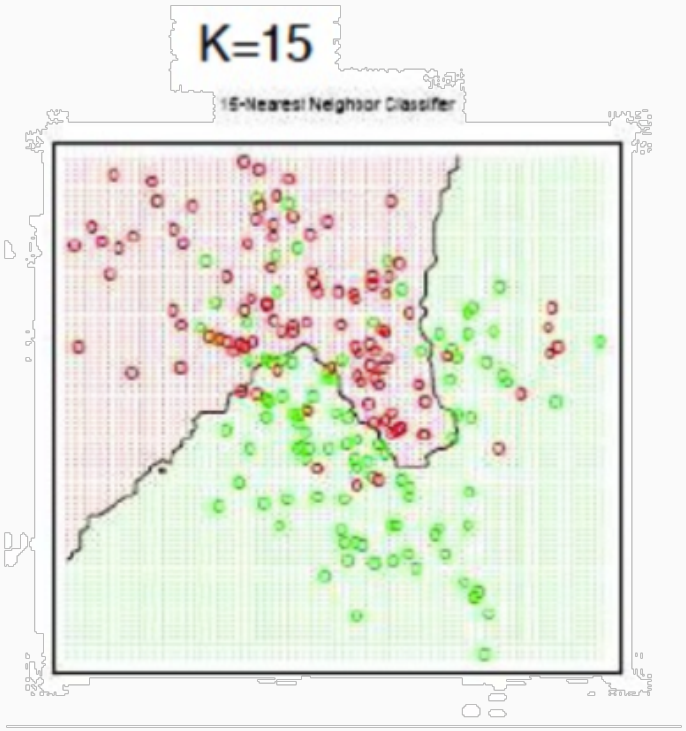
(a) $k = 5$



(b) $k = 11$

K-Nearest Neighbors: Importance of K





The number of neighbors is a model hyperparameter

- K too small: the function f is too sensitive to the data — high variance
- K too large: the function f becomes overly insensitive to the data — high bias

K-Nearest Neighbors: Classification

K-NN (K-Nearest Neighbors)

Binary classification case:

$$h_{KNN}(\mathbf{X}) = \arg \max_{y \in \{-1,1\}} \frac{N_y^K(\mathbf{X})}{K}$$

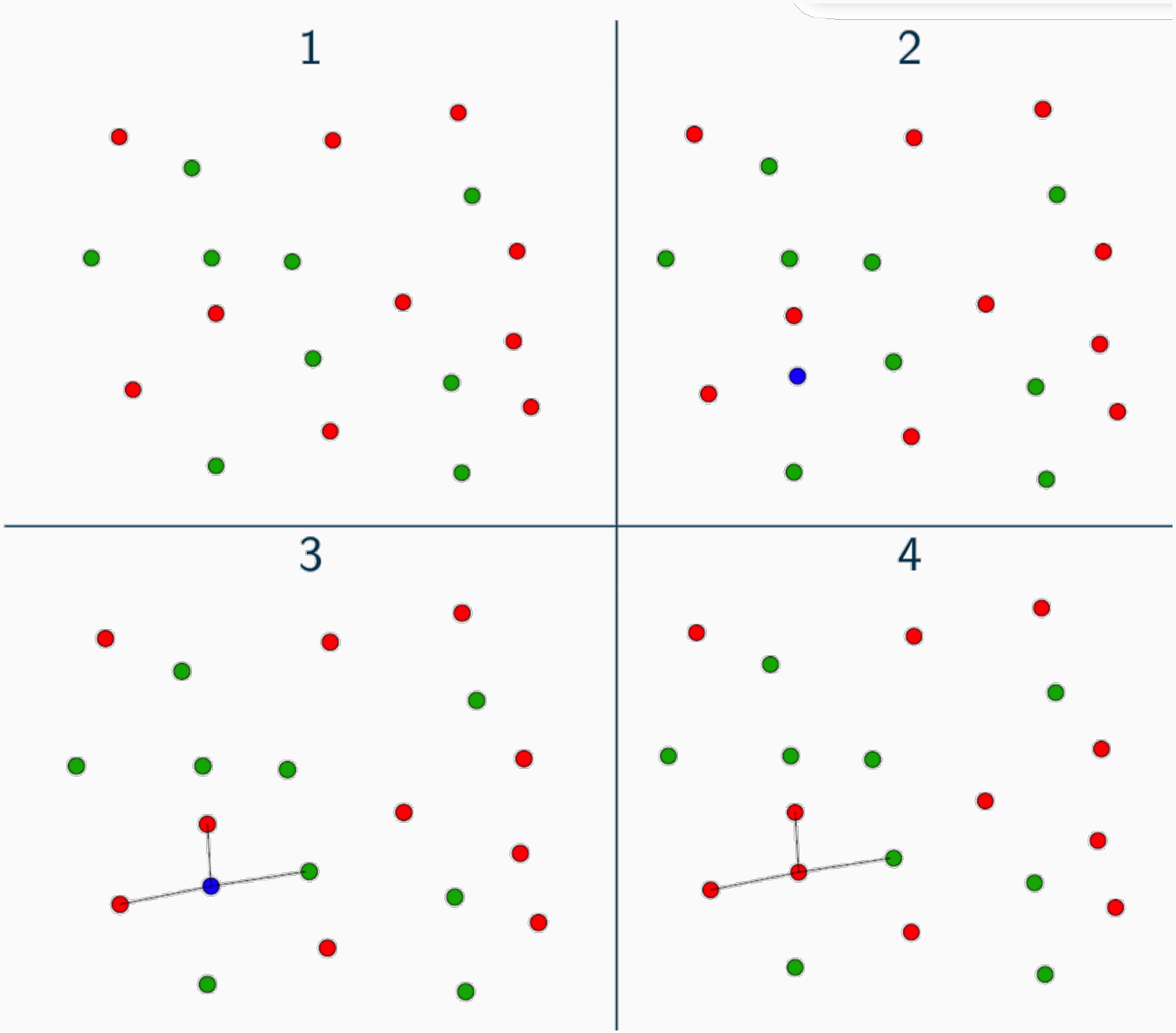
where:

- Let K be a strictly positive integer
- Let d be a metric defined on $\mathcal{X} \times \mathcal{X}$
- $S = \{(\mathbf{X}_i, y_i), i = 1, ..., n\}$
- For a given $\mathbf{X}$, define σ as the permutation of indices in $\{1, ..., n\}$ such that:

$$\mathrm{d}(\mathbf{X}, \mathbf{X}_{\sigma(1)}) \leq \mathrm{d}(\mathbf{X}, \mathbf{X}_{\sigma(2)}) ... \leq \mathrm{d}(\mathbf{X}, \mathbf{X}_{\sigma(n)})$$

- $S_{\mathbf{X}}^K = \{\mathbf{X}_{\sigma(1)}, ..., \mathbf{X}_{\sigma(K)}\}$: first K neighbors of $\mathbf{X}$
- $N_y^K(\mathbf{X}) = |\{\mathbf{X}_i \in S_{\mathbf{X}}^K, y_i = y\}|$

K-Nearest Neighbors: Algorithm



K-Nearest Neighbors: Distance Metrics

Metric used to compute distances:

$$d(x, y) = \sqrt[p]{\sum |x - y|^p}$$

Common metrics

- Manhattan: $p = 1$
- Euclidean: $p = 2$
- Minkowski: other p
- Chebyshev: $p = \infty$
- Mahalanobis: $d(x, y) = \sqrt{(x - y)^T \Sigma^{-1} (x - y)}$

(see `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.DistanceMetric.html`)

Attributes must be normalized because distances are computed across dimensions, and some attributes might dominate others.

Data Transformations: Distribution-Based

Standardization

The estimator may behave poorly if individual features are not roughly normally distributed: Gaussian with **zero mean and unit variance**: $\frac{x-\mu_x}{\sigma_x}$

Scaling features to a range

Usually between 0 and 1. This helps mitigate very small standard deviations and preserves zero entries in sparse data: $\frac{x-\min_x}{\max_x-\min_x}$

Normalization

Normalization adjusts each sample to have a **unit norm**.

More info here

K-NN: Additional Notes

K-NN classifiers are **lazy learners**:

- They do not build explicit models, offering flexibility by avoiding global model assumptions
- In contrast to **eager learners** like decision trees or rule-based classifiers
- Independent of the number of classes
- Classification is more costly (memory and computation)

K-NN: Dimensionalidad

With High dimensionality: K-NN suffers from Curse of Dimensionality

- **Data Sparsity**: With more features, the data becomes increasingly sparse, making it harder to find similar instances.
- **Noise Accumulation**: Noisy features can dominate the distance calculations, leading to poor predictions.
- **Overfitting**: Models may overfit to the noise in the data rather than the underlying patterns.

Consequences of the Curse of Dimensionality:

- **Decreased Accuracy**: As the number of features increases, the KNN algorithm's accuracy may decrease due to the increased difficulty in finding meaningful neighbors.
- **Increased Computational Cost**: Calculating distances in HD spaces becomes computationally expensive, leading to slower prediction times.
- **Data Requirements**: The algorithm requires a large amount of data to achieve reliable results, which can be challenging to obtain, especially in high-dimensional spaces.

Naive Bayes

Outline : Naive Bayes

Naive Bayes

The conditional probability of event A given event B is:

$$P(A|B) = \frac{P(A, B)}{P(B)}$$

The Bayes' theorem for events A and B is:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Naive Bayes

Naive Bayes

- Classical algorithm modeling $\mathbb{P}\{\mathbf{X}|Y\}$
- Strong independence assumption between features: $\mathbb{P}\{\mathbf{X}|Y\} = \prod_{i=1}^{d} \mathbb{P}\{X^{(i)}|Y\}$

- Example with c classes, $o = (w_1, ..., w_N)$ observations of N words:

$$\hat{c} = \underset{c}{\operatorname{argmax}}\, P(c|o) = \underset{c}{\operatorname{argmax}}\, \frac{P(o|c)P(c)}{P(o)}$$

- Assuming $P(o)$ is constant:

$$\hat{c} = \underset{c}{\operatorname{argmax}}\, P(o|c)P(c)$$

- **Naive** assumption: features are independent:

$$P(o|c) = P(w_1, ..., w_N|c) = \prod_{i=1..N} P(w_i|c)$$

$P(w_i|c) = \frac{\text{Count}(w_i \in C)}{\text{Count}(w_i)}$: counting feature/class co-occurrences

Naive Bayes

The algorithm uses the following formula to compute the posterior probability of a class C given a set of observations o:

$$P(c|o) = \frac{P(o|c) \cdot P(c)}{P(o)}$$

Where:

- $P(c|o)$ is the **posterior probability** of class c given features o
- $P(o|c)$ is the **likelihood** of the features given class c
- $P(c)$ is the **prior probability** of class c
- $P(o)$ is the evidence (probability of the features)

Laplace smoothing helps handle cases where $P(w_i|c) = 0$.

Naive Bayes: Continuous Features

Different Naive Bayes classifiers vary in their assumptions about the distribution of $P(w_i|c)$.

For continuous features, the Gaussian Naive Bayes assumes a normal distribution:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

Parameters σ_y and μ_y are estimated using maximum likelihood (i.e., from training data per class).

Naive Bayes: Summary

- Known as a decent classifier, but a **poor probability estimator**
- Robust to isolated noisy points
- Robust to irrelevant features (affect all classes equally)
- Particularly effective for categorical or discrete features
- Provides good baseline performance even with small training data
- Sensitive to highly correlated features due to independence assumption

Random Forests

Outline : Random Forests

Outline : Decision Trees

K-Nearest Neighbors

Naive Bayes

Random Forests

Decision Trees

Ensemble Methods

Random Forests

API : Scikit-learn

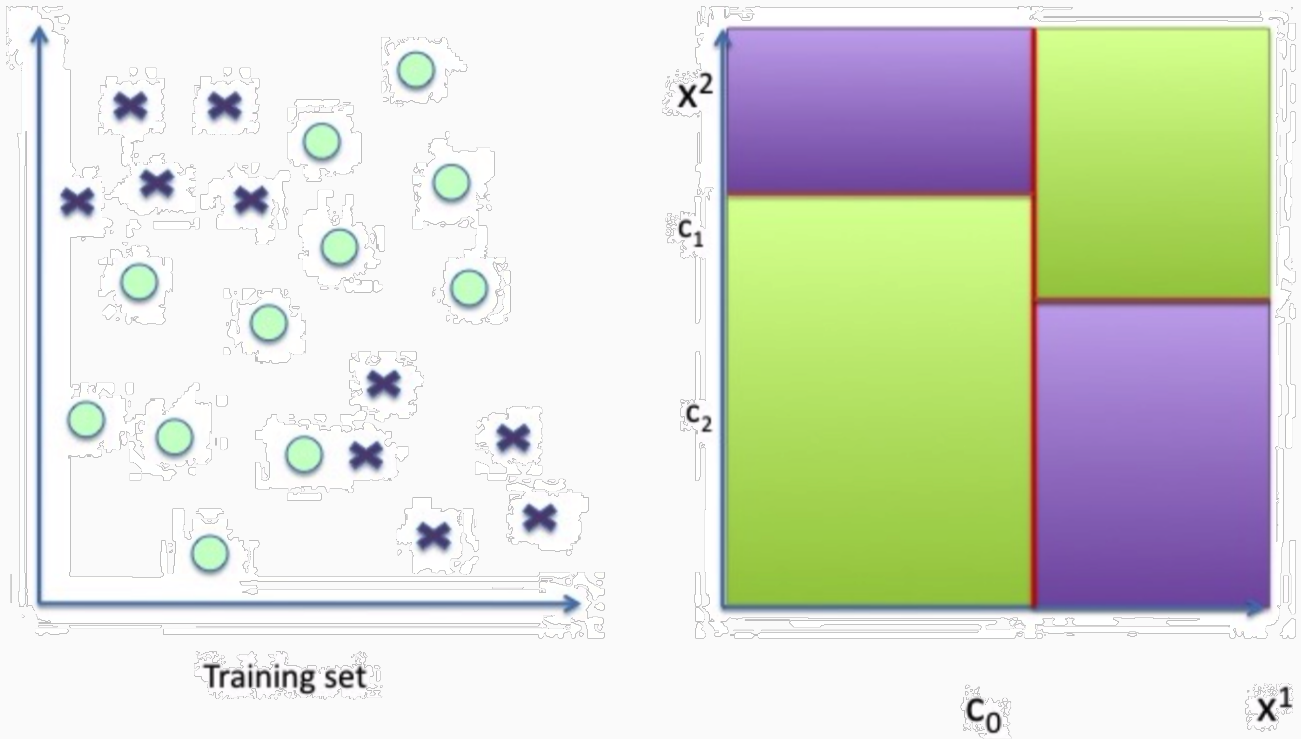
Decision Trees

Three Principles

Find thresholds in different features to discriminate between classes:

P1 Using multiple linear separators

P2 Using hyperplanes $x^j = c_k$ to maintain interpretability

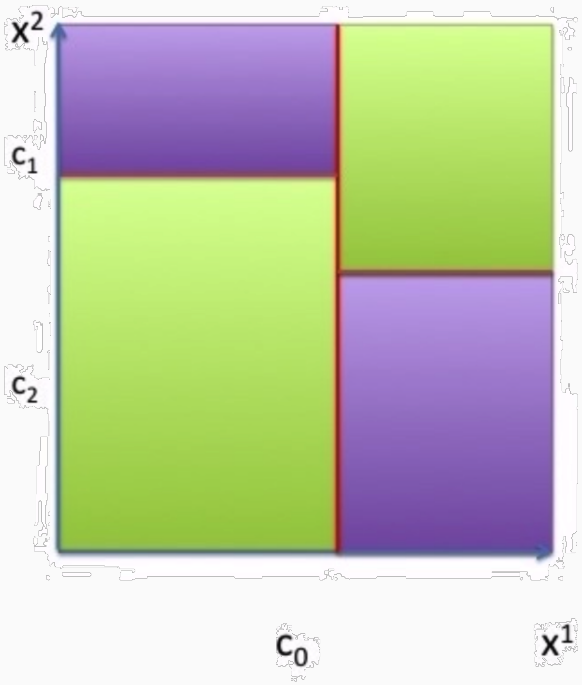


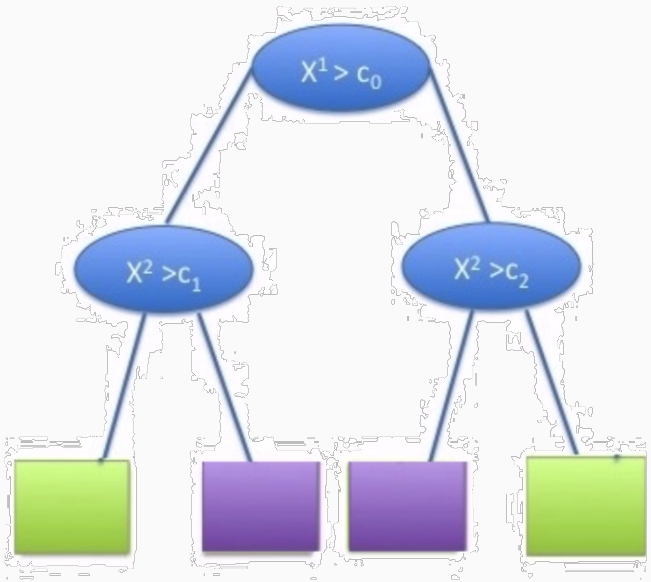
Decision Trees

Three Principles

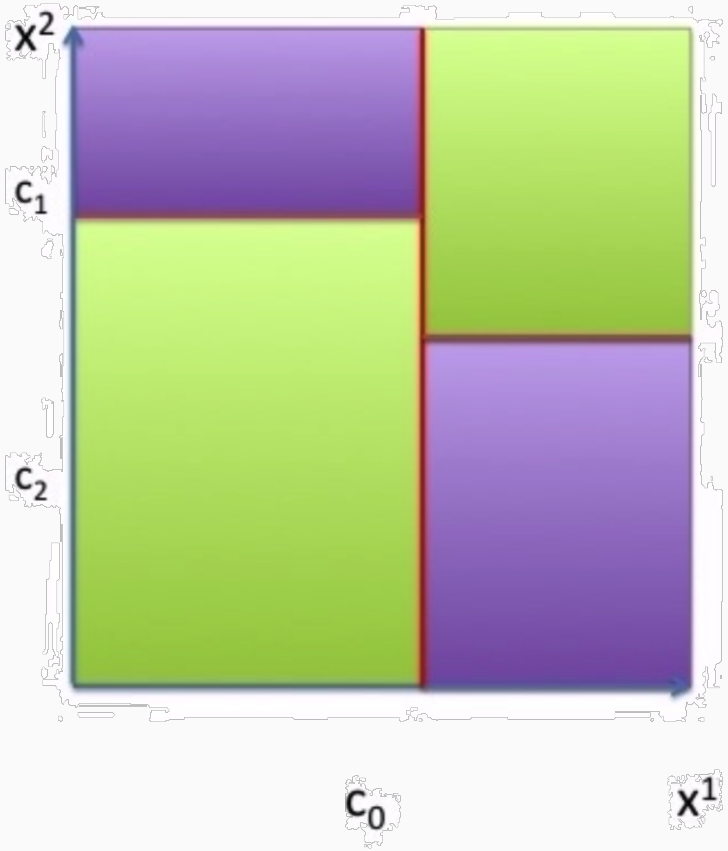
Partition the feature space non-linearly:

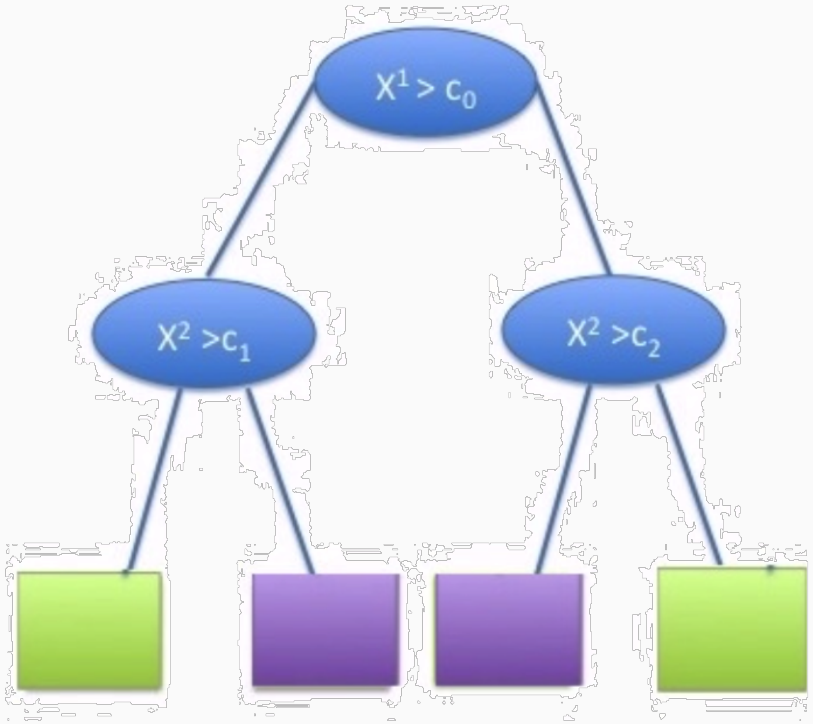
P3 The learned function is represented by a tree structure: each node is associated with a hyperplane $x^j = \theta_j$, each leaf with a class





Decision Trees





At the end of the learning phase, it is clear which features influence the final decision: If $x^{k_{j_1}} < c_{k_{j_1}}$ and $x^{k_{j_2}} > c_{k_{j_2}}$ and ..., then **X** belongs to class k (with k classes and j possible paths).

Continuous or Real-Valued Variables

Depending on the variable type x^j, the split t is defined differently:

- If x^j is continuous:

$$\text{sign}(x^j - c_{k_j})$$

- If x^j is categorical with two values $v_j^1, ..., v_j^2$:

$$\mathbb{I}(x^j = v_j^1)$$

Decision Trees: Example

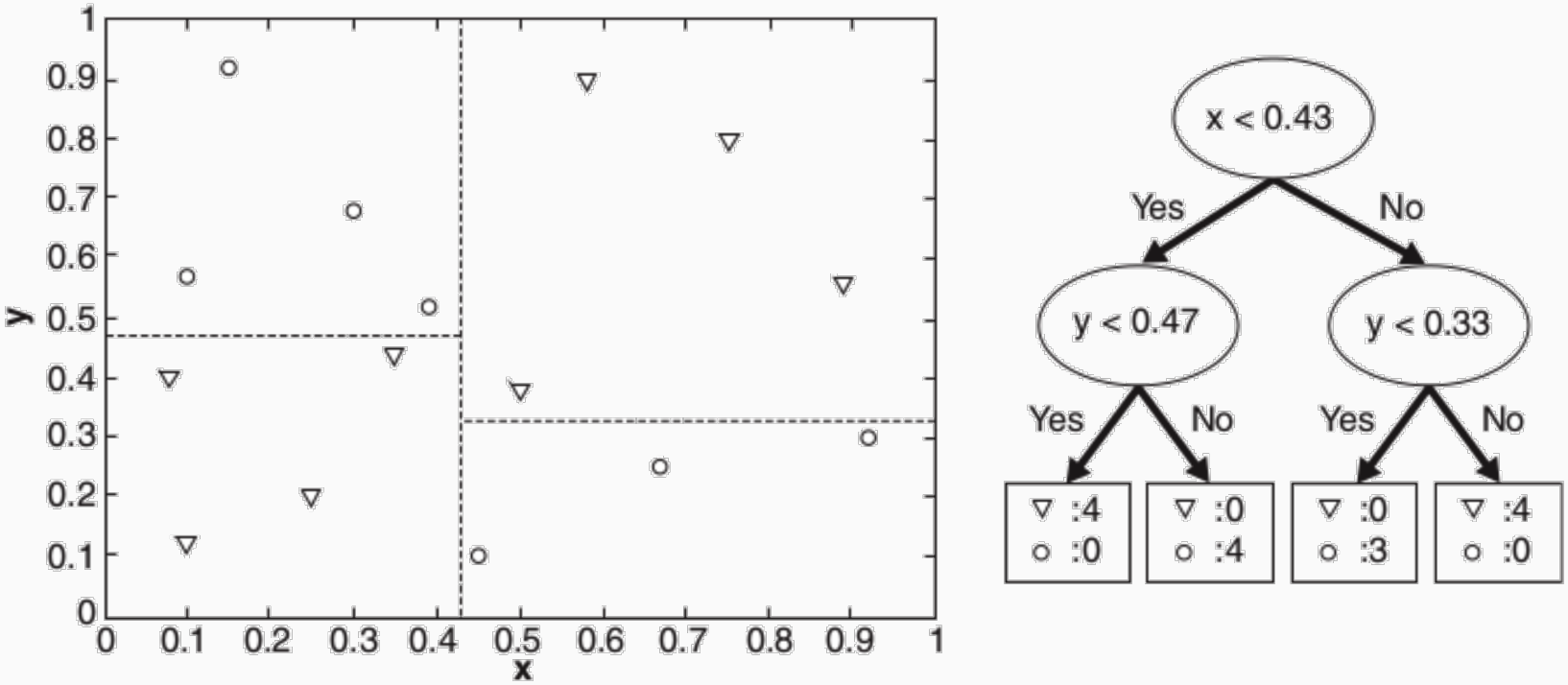


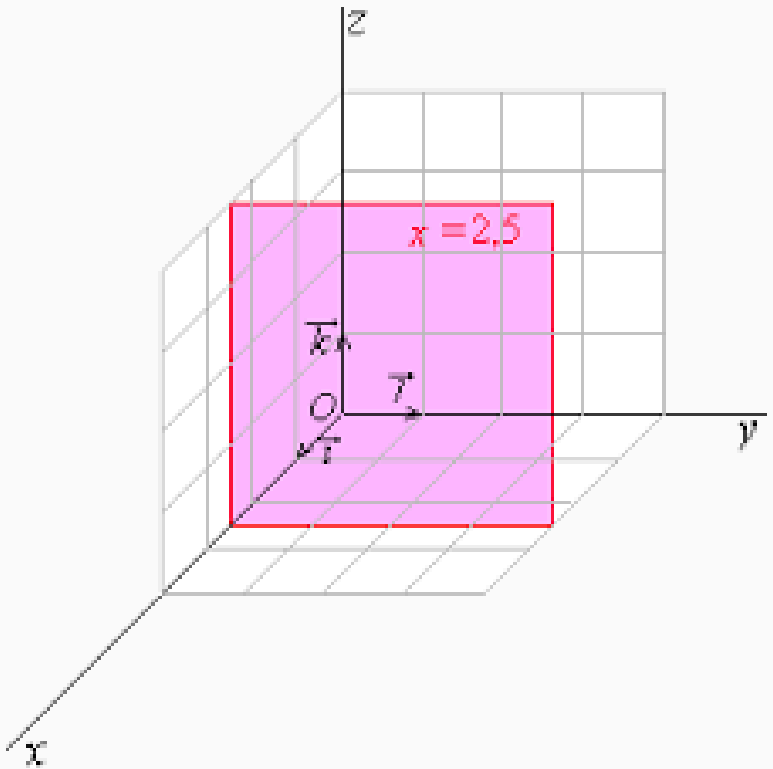
Figure 3.20. Example of a decision tree and its decision boundaries for a two-dimensional data set.

Partition: Indicator Function Class

Definition

$$\forall \mathbf{X} \in \mathbb{R}^p, h_{tree}(\mathbf{X}) = \sum_{l=1}^{L} \alpha_l \mathbb{I}_{\mathcal{C}_l}(\mathbf{X})$$

Where $\mathcal{C}_1 \cup ... \cup \mathcal{C}_L$ is a partition of $\mathbb{R}^p$, and each $\mathcal{C}_l$ is defined by a subset of hyperplanes orthogonal to a basis vector



Partition: Majority Class Vote

- Let $\mathbf{X} \in \mathbb{R}^p$, then $I(\mathbf{X}) \in \{1, ..., L\}$ such that $\mathbf{X} \in \mathcal{C}_{I(\mathbf{X})}$
- Classification is done by majority vote in $\mathcal{C}_{I(\mathbf{X})}$:

$$h_n(\mathbf{X}) = \alpha_{I(\mathbf{X})} = \arg\max_k \sum_{X^{(i)} \in \mathcal{C}_{I(\mathbf{X})}} \mathbb{I}(y_i = k)$$

Goal

Learn the partition $\mathcal{C}_1 \cup ... \cup \mathcal{C}_L$

Decision Trees: Algorithm

For a binary tree:

1. Let $\mathcal{S}$ be the training set
2. Build the root node
3. Find the best split $t(\mathbf{X})$ to apply to the current training set $\mathcal{S}$ such that the local loss $L(t, \mathcal{S})$ be minimal
4. Split $\mathcal{S}$ into $\mathcal{S}_g$ and $\mathcal{S}_d$ using the split
5. Build a right node and a left node
6. Measure the stopping criterion on the right side and if checked, then the right node becomes a leaf, otherwise go to 3 with $\mathcal{S}_d$
7. Same for the left node with $\mathcal{S}_g$

Decision Trees: Local Loss and Impurity Criterion I

Let $\mathcal{S}$ be the training set and $t_{j,\tau}$ a split:

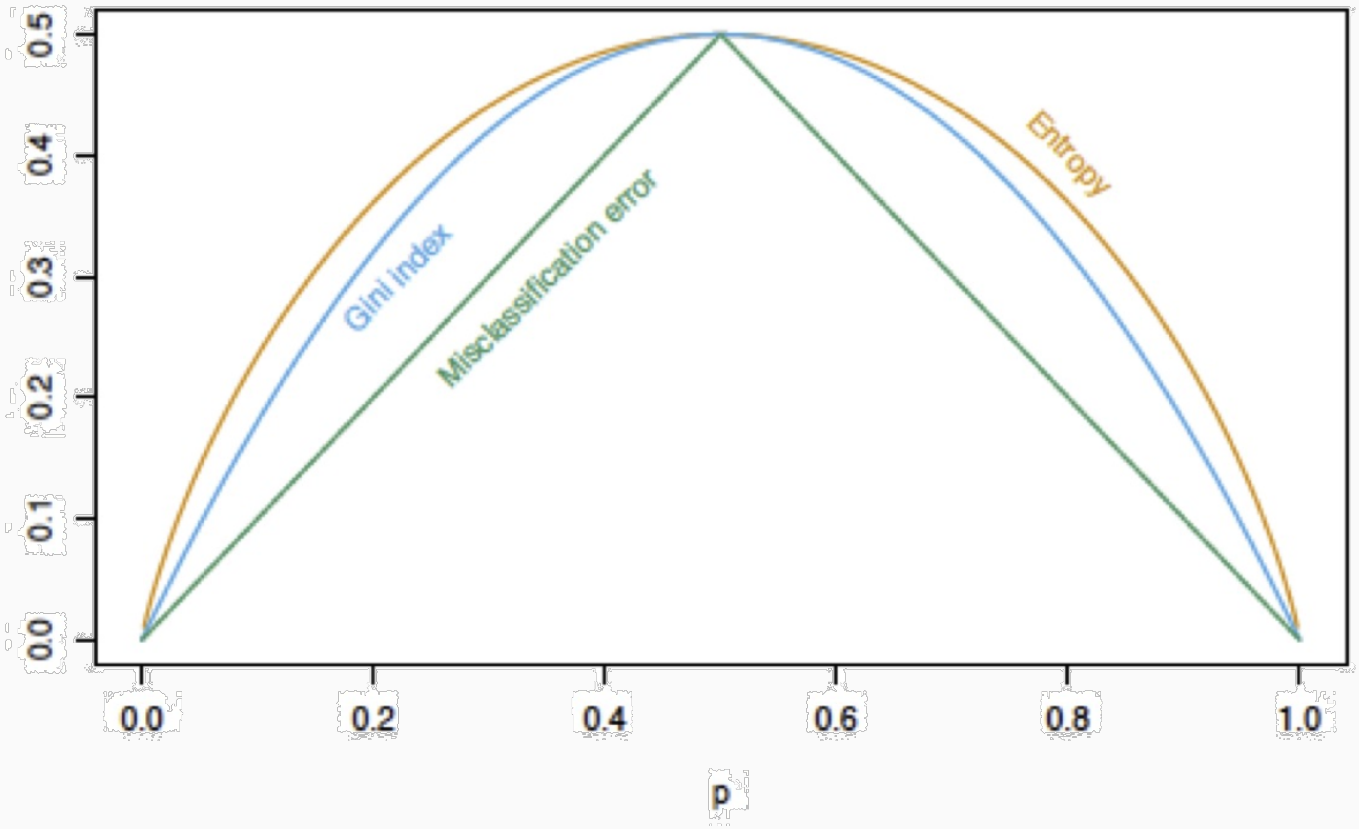
- $\mathcal{D}(\mathcal{S}, j, \tau) = \{(\mathbf{X}, Y) \in \mathcal{S}, t_{j,\tau} > 0\}$
- $\mathcal{I}(\mathcal{S}, j, \tau) = \{(\mathbf{X}, Y) \in \mathcal{S}, t_{j,\tau} \leq 0\}$

The τ_i can be chosen uniformly or by histogram. Among parameters $(j, \tau) \in \{1, ..., d\} \times \{\tau_1, ..., \tau_C\}$, we find the one minimizing:

$$L(t_{j,\tau}, \mathcal{S}) = \frac{|\mathcal{D}(\mathcal{S}, j, \tau)|}{n} H(\mathcal{D}(\mathcal{S}, j, \tau)) + \frac{|\mathcal{I}(\mathcal{S}, j, \tau)|}{n} H(\mathcal{I}(\mathcal{S}, j, \tau))$$

H measures the impurity of a subset $\mathcal{S}$. We want the least homogeneous distribution in terms of class.

Decision Trees: Local Loss and Impurity Criterion II



Possible impurity measures

- Cross-Entropy: $H(\mathcal{S}) = -\sum_{c=1}^{C} p_c(\mathcal{S}) \log(p_c(\mathcal{S}))$
- Gini Index: $H(\mathcal{S}) = \sum_{c=1}^{C} p_c(\mathcal{S})(1 - p_c(\mathcal{S}))$
- Majority class error: $H(\mathcal{S}) = 1 - p_{\text{major class}(\mathcal{S})}$

Entropy is maximum at $p = 0.5$, which is the most uncertain for binary classification.

Example: Information Gain

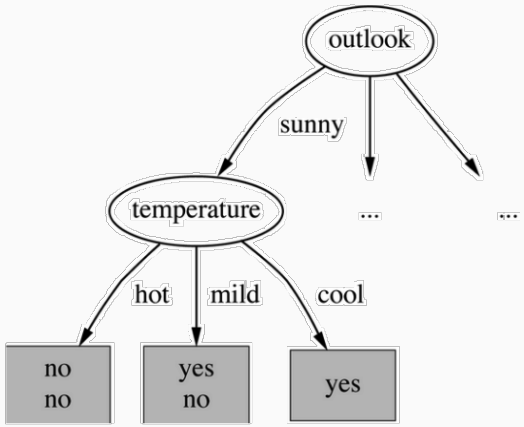
Information Gain

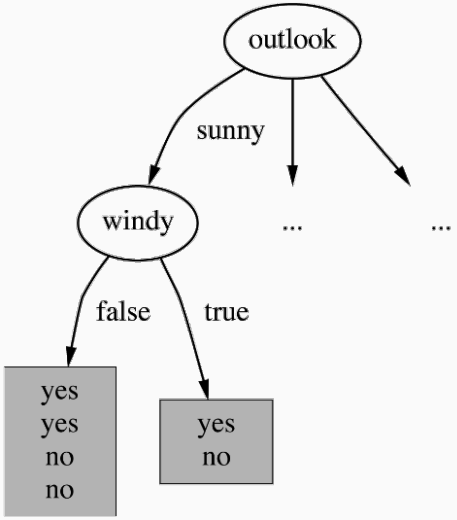
Using H, we can compute the information gain in bits — the difference in entropy before and after the split.

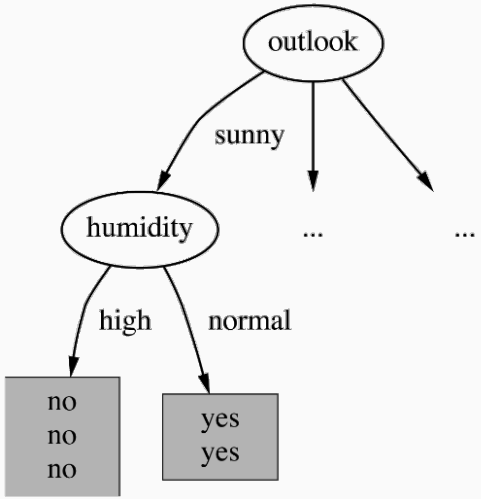
This optimization helps build the smallest possible tree. H gives the value in bits representing the *information gain*:

- **Heuristic**: choose the attribute that creates the purest child nodes
- **Popular purity criterion**: information gain, increases as average purity increases
- **Strategy**: pick the attribute that maximizes information gain

Example: Information Gain







gain(*Temperature*) = 0.571 bits
gain(*Humidity*) = 0.971 bits
gain(*Windy*) = 0.020 bits

Humidity allows a **good class separation**, meaning **low entropy**, and a **high information gain**.

Decision Trees: Stopping Criteria

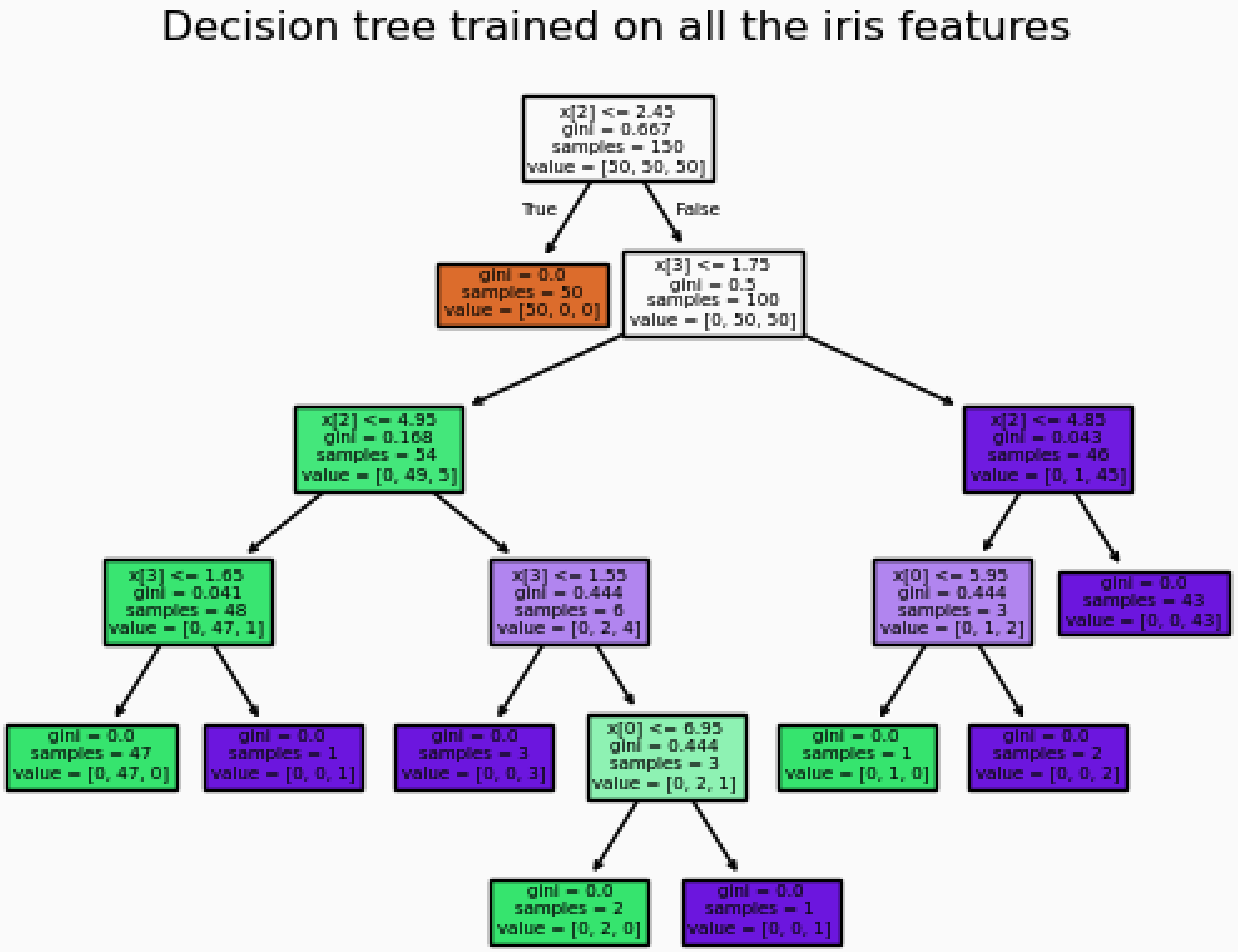
Stop growing the tree when any of the following is met:

- Maximum depth
- Maximum number of leaves
- Minimum number of samples in a leaf

Note: If the minimal number of training data in a leaf is equal to 1, it is possible that the tree grows until a perfect classification of the training examples is achieved: overfitting !

Decision Trees: Scikit-learn

Decision trees can be visualized using `sklearn.tree.plot_tree`



Decision Trees: Model Selection

To tune decision tree hyperparameters:

- **Cross-validation** on: max depth, max leaves
- **Pruning**: on a validation set, remove branches that don't improve performance

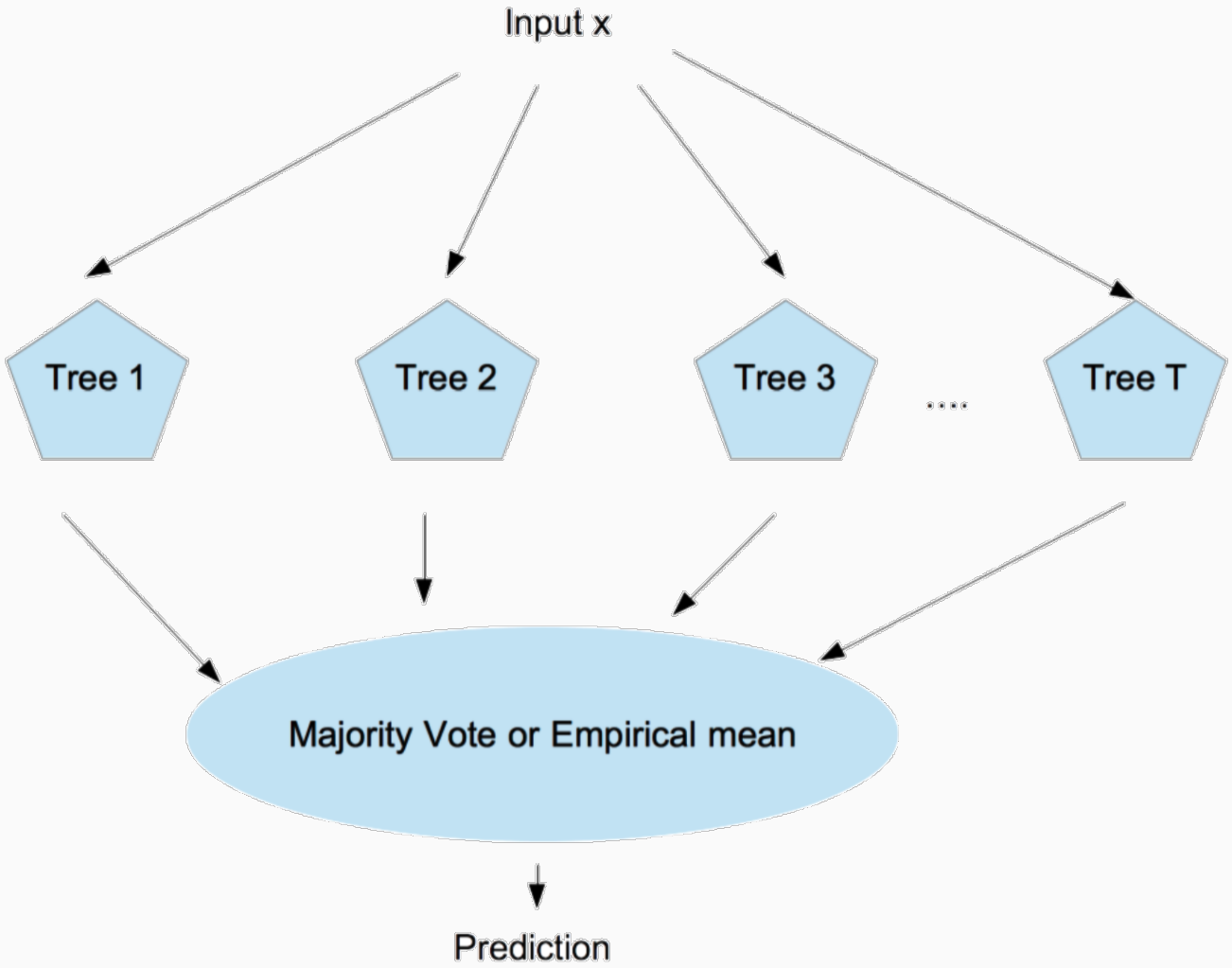
Outline : Ensemble Methods

Ensemble Methods

Machine Learning not so "automatic": too many hyperparameters to tune!

Ensemble Methods

Machine Learning not so "automatic": too many hyperparameters to tune! Ensemble learning improves upon a single predictor by building an **ensemble of weak predictors** (simple model with no hyperparameter)



Diversity of the base predictors

$$MSE(f_{ens}) = \mathbb{E}[(y - f_{ens}(x))^2] = \frac{1}{T^2}\mathbb{E}\left[\left(\sum_t \epsilon_t(x)\right)^2\right]$$

Now, if ϵ_t are mutually independent with zero mean, then we have:

$$MSE(f_{ens}) = \frac{1}{T^2}\mathbb{E}\left[\sum_t \epsilon_t(x)^2\right]$$

Diversity of the base predictors

$$MSE(f_{ens}) = \mathbb{E}[(y - f_{ens}(x))^2] = \frac{1}{T^2}\mathbb{E}\left[\left(\sum_t \epsilon_t(x)\right)^2\right]$$

Now, if ϵ_t are mutually independent with zero mean, then we have:

$$MSE(f_{ens}) = \frac{1}{T^2}\mathbb{E}\left[\sum_t \epsilon_t(x)^2\right]$$

$\Rightarrow$ The more diverse are the classifiers, the more we reduce the mean square error

Diversity of the base predictors

$$MSE(f_{ens}) = \mathbb{E}[(y - f_{ens}(x))^2] = \frac{1}{T^2}\mathbb{E}\left[\left(\sum_t \epsilon_t(x)\right)^2\right]$$

Now, if ϵ_t are mutually independent with zero mean, then we have:

$$MSE(f_{ens}) = \frac{1}{T^2}\mathbb{E}\left[\sum_t \epsilon_t(x)^2\right]$$

$\Rightarrow$ The more diverse are the classifiers, the more we reduce the mean square error

Encourage the diversity of base predictors by:

- using bootstrap samples (Bagging and Random forests)
- using randomized predictors (ex: Random forests)
- using weighted version of the current sample (Boosting)
- with weights dependent on the previous predictor (adaptive sampling)

An Example of Bagging

A first algorithm example:

- Construct T training sets $\mathcal{S}_1, ..., \mathcal{S}_T$.
- Learn a model $f_t \in \mathcal{F}$ for each of these training sets $\mathcal{S}_t, t = 1...T$.
- Calculate the mean of the models $f_{ens} = \frac{1}{T} \sum_{t=1}^{T} f_t(\mathbf{X})$

When is it useful?

The approximation error is due to bias (model with respect to the task), variance (model with respect to the partition of the data set) and noise (depends on which descriptors the model can see). Making a multi-model approximation helps to reduce variance, which is useful for models with high variance such as decision trees or neural networks (i.e., less stable models).

Bagging (Bootstrap Aggregating) I

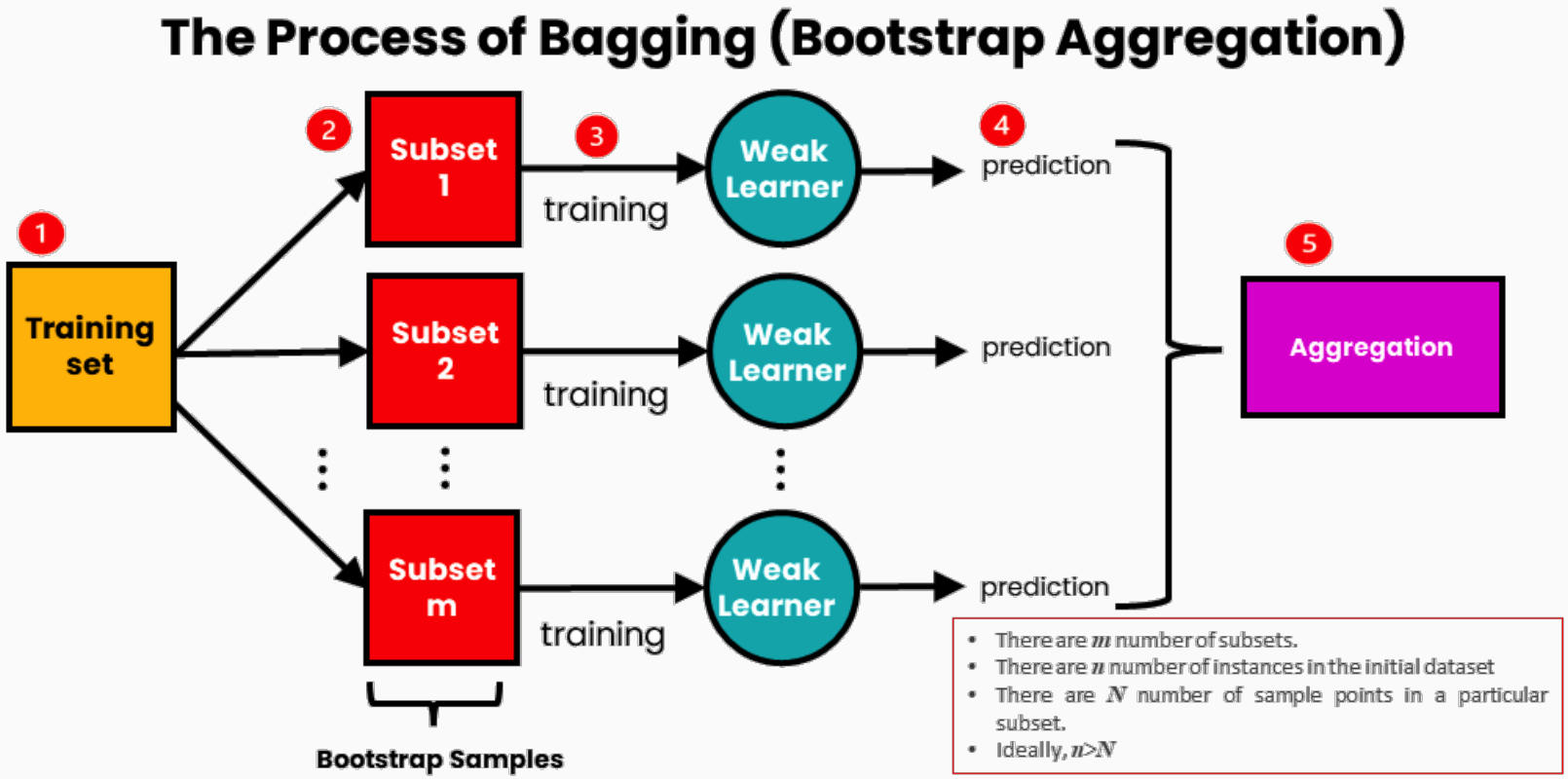
Definition

Ensemble learning technique designed to improve the accuracy and stability of machine learning algorithms.

It involves the following steps:

1. **Data Sampling**: Creating multiple subsets of the training dataset using bootstrap sampling (random sampling with replacement; $\approx$ 30% smaller than S): Construct T training sets $\mathcal{S}_1, ..., \mathcal{S}_T$
2. **Model Training**: raining a separate model on each subset of the data: Learn a model $f_t \in \mathcal{F}$ for each of these training sets $\mathcal{S}_t, t = 1...T$
3. **Aggregation**: Combining the predictions from all individual models (averaged for regression or majority voting for classification) to produce the final output: Calculate the mean of the models $f_{ens} = \frac{1}{T} \sum_{t=1}^{T} f_t(\mathbf{X})$

Bagging (Bootstrap Aggregating) II



It helps to:

- **Reduces Variance**: By averaging multiple predictions, bagging reduces the variance of the model and helps prevent overfitting.
- **Improves Accuracy**: Combining multiple models usually leads to better performance than individual models

Boosting I

Definition

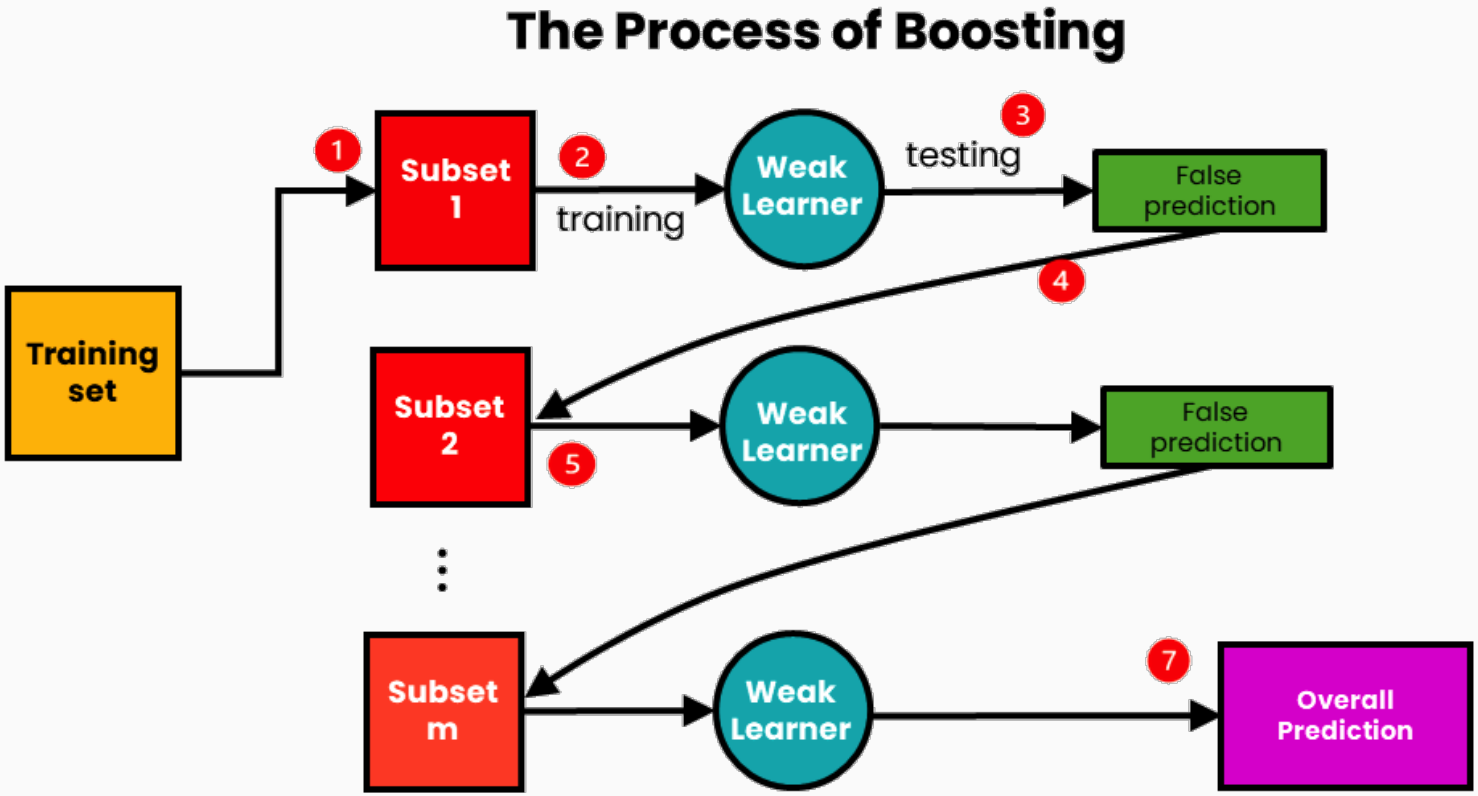
Ensemble learning technique that focuses on creating a strong model by combining several weak models.

It involves the following steps:

1. **Sequential Training**: Training models sequentially, each one trying to correct the errors made by the previous models.
2. **Weight Adjustment**: Each instance in the training set is weighted. Initially, all instances have equal weights. After each model is trained, the weights of misclassified instances are increased so that the next model focuses more on difficult cases.
3. **Model Combination**: Combining the predictions from all models to produce the final output, typically by weighted voting or weighted averaging.

More info about boosting and bagging here and there.

Boosting II

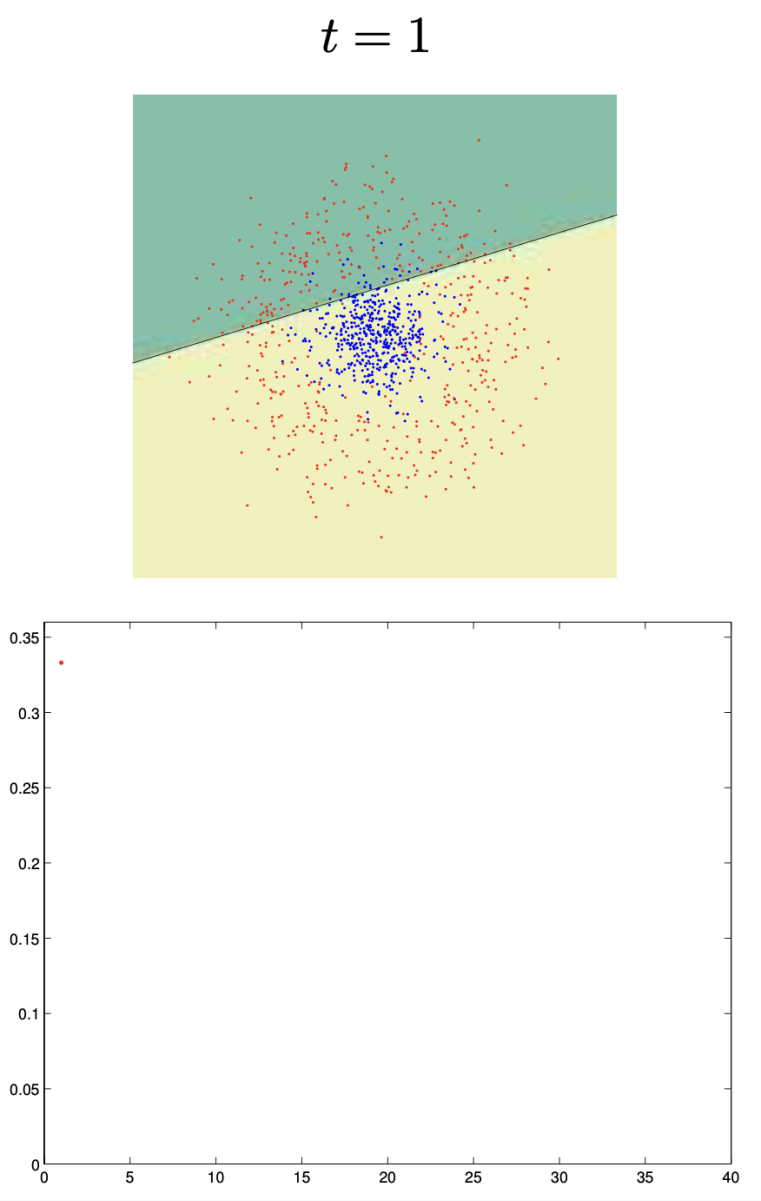


It helps to:

- **Reduces Bias**: By focusing on hard-to-classify instances, boosting reduces bias and improves the overall model accuracy.
- **Produces Strong Predictors**: Combining weak learners leads to a strong predictive model.

An Example of Boosting

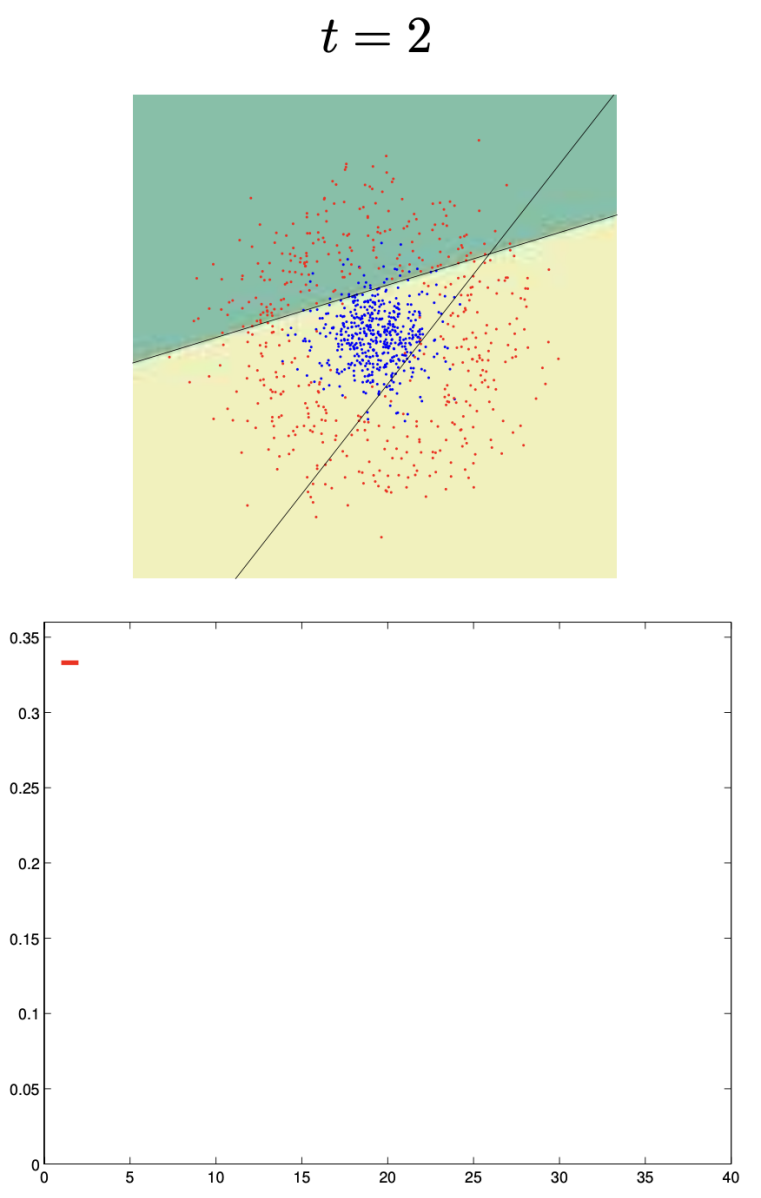
- Binary Classifier:
  $F_1(x) = sign(H_1(x))$



Example from this Adaboost class

An Example of Boosting

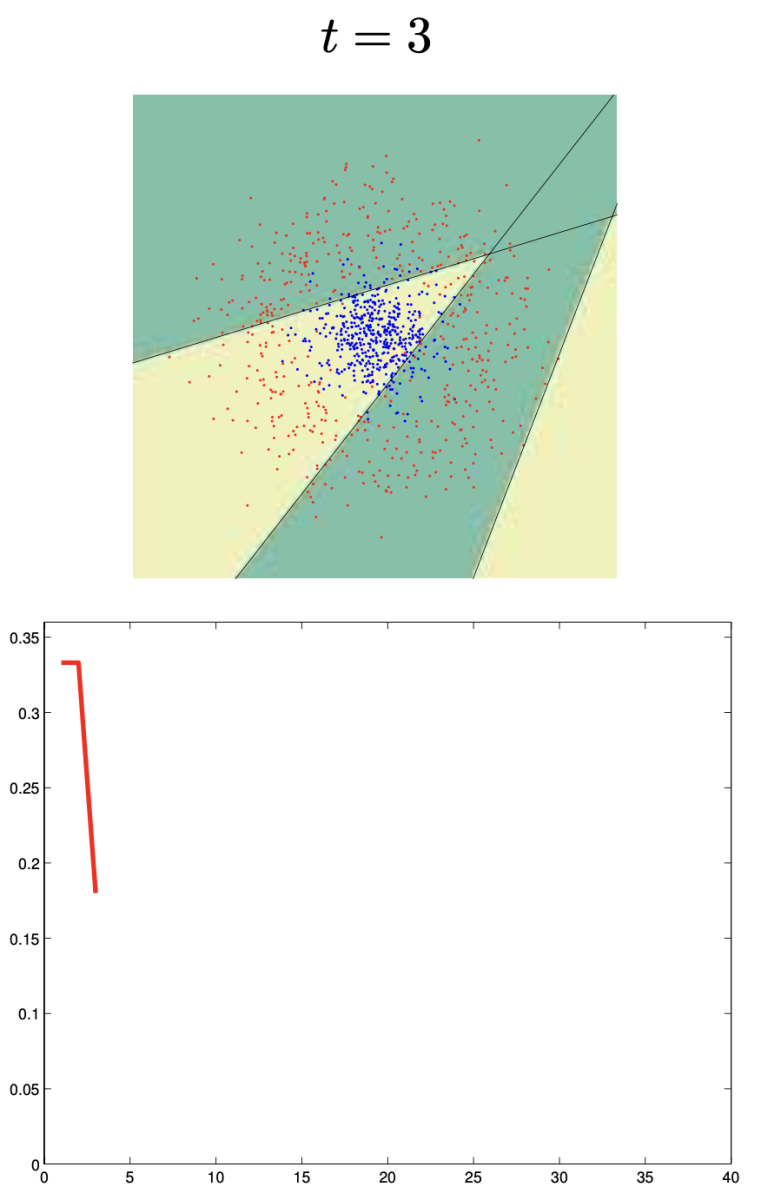
- Binary Classifier:
  $F_1(x) = sign(H_1(x))$
- $H_2(x) = h_1(x) + h_2(x)$
- Binary Classifier:
  $F_2(x) = sign(H_2(x))$



Example from this Adaboost class

An Example of Boosting

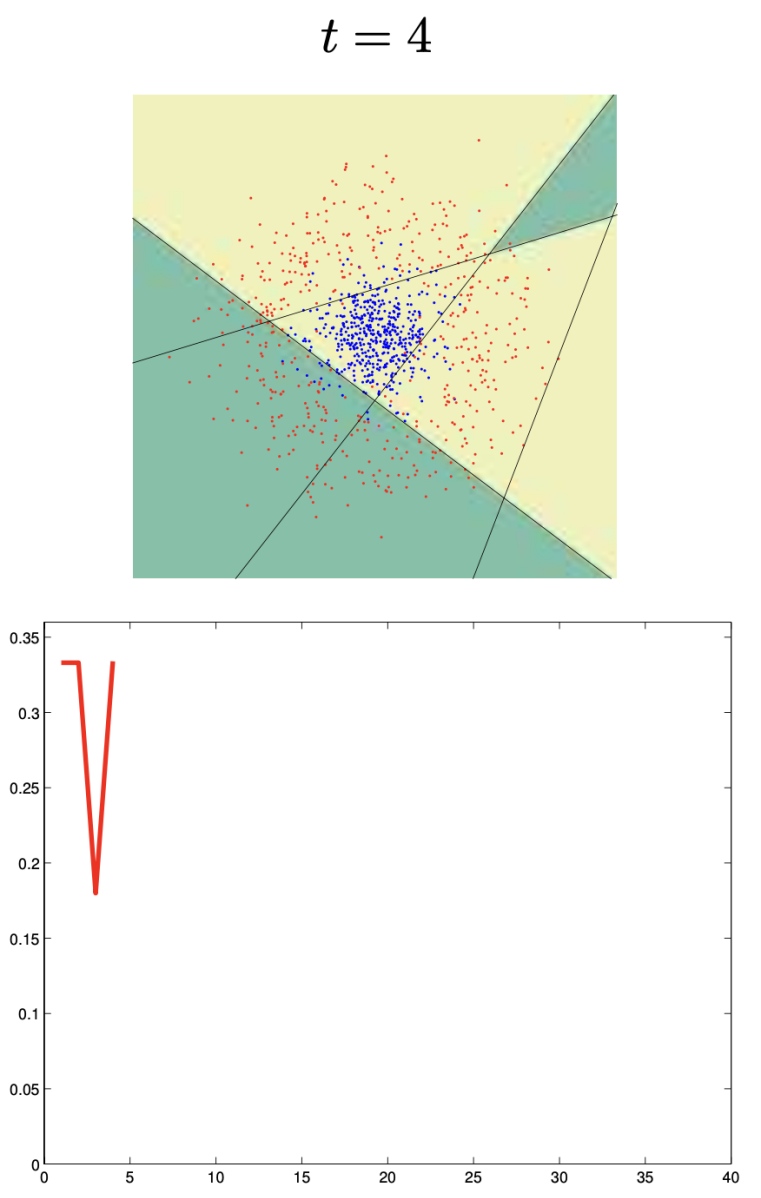
- Binary Classifier:
  $F_1(x) = sign(H_1(x))$
- $H_2(x) = h_1(x) + h_2(x)$
- Binary Classifier:
  $F_2(x) = sign(H_2(x))$
- ...



Example from this Adaboost class

An Example of Boosting

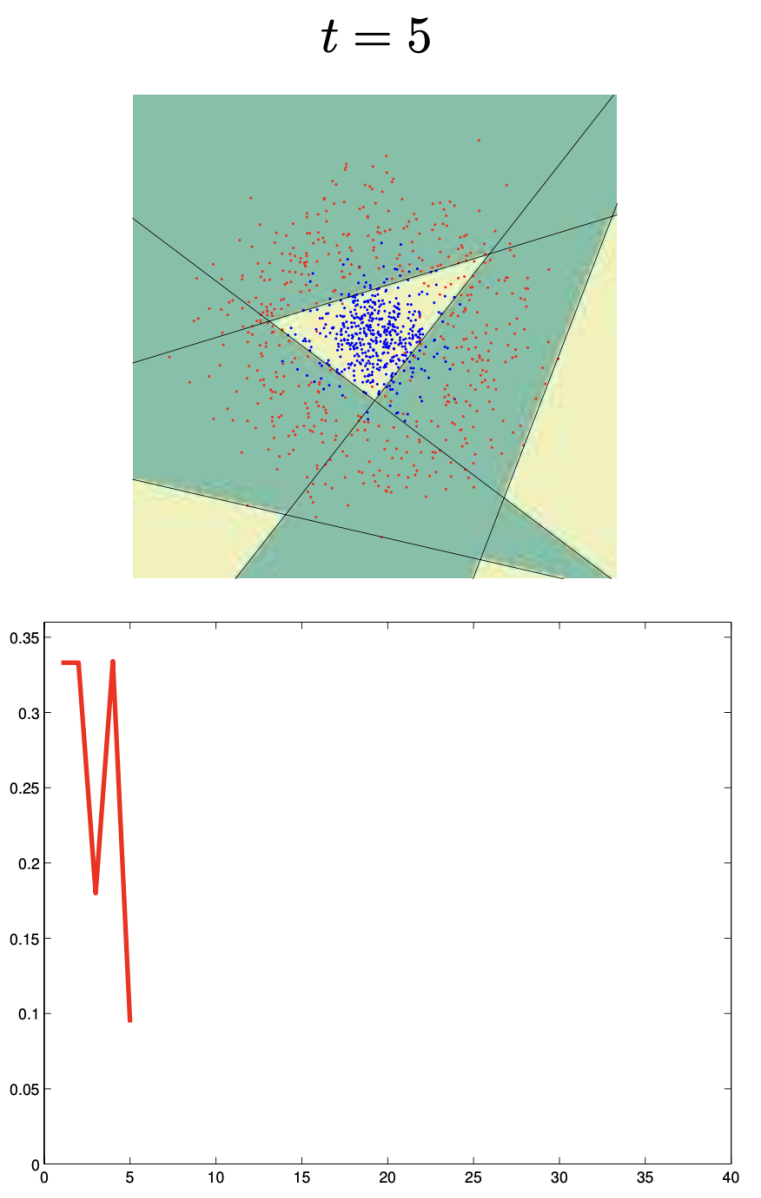
- Binary Classifier:
  $F_1(x) = sign(H_1(x))$
- $H_2(x) = h_1(x) + h_2(x)$
- Binary Classifier:
  $F_2(x) = sign(H_2(x))$
- ...



Example from this Adaboost class

An Example of Boosting

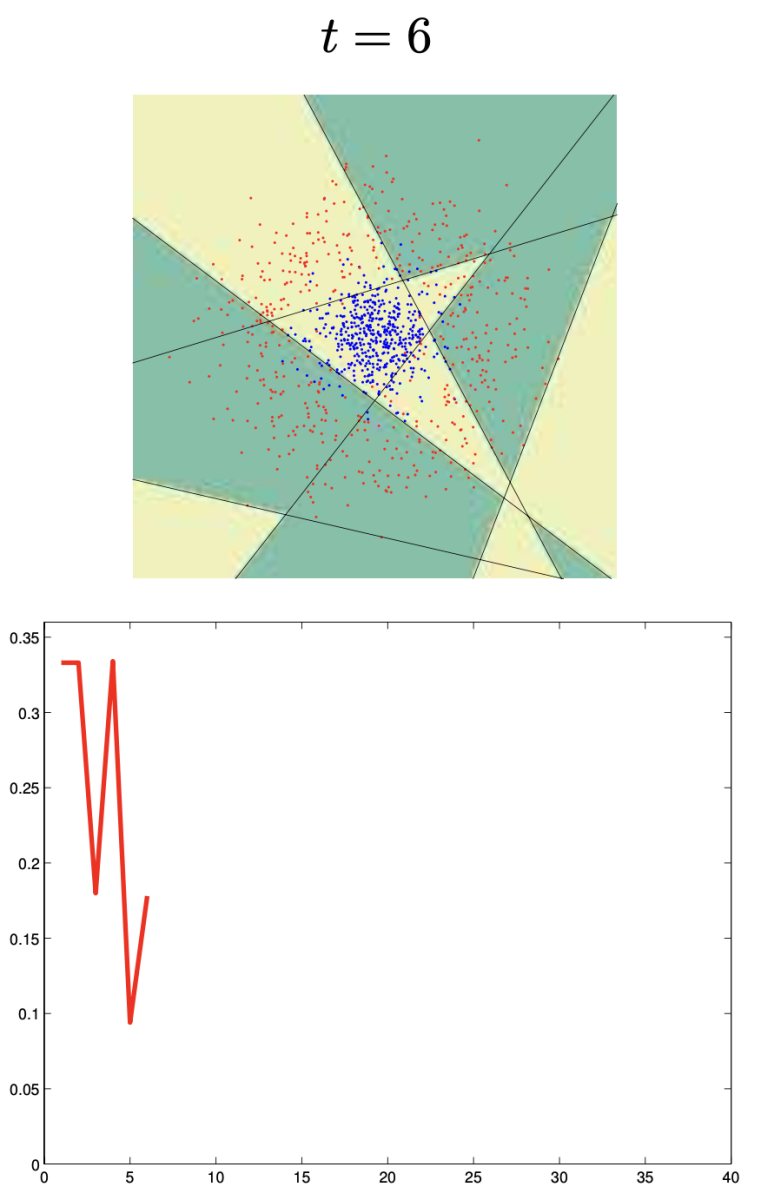
- Binary Classifier:
  $F_1(x) = sign(H_1(x))$
- $H_2(x) = h_1(x) + h_2(x)$
- Binary Classifier:
  $F_2(x) = sign(H_2(x))$
- ...



Example from this Adaboost class

An Example of Boosting

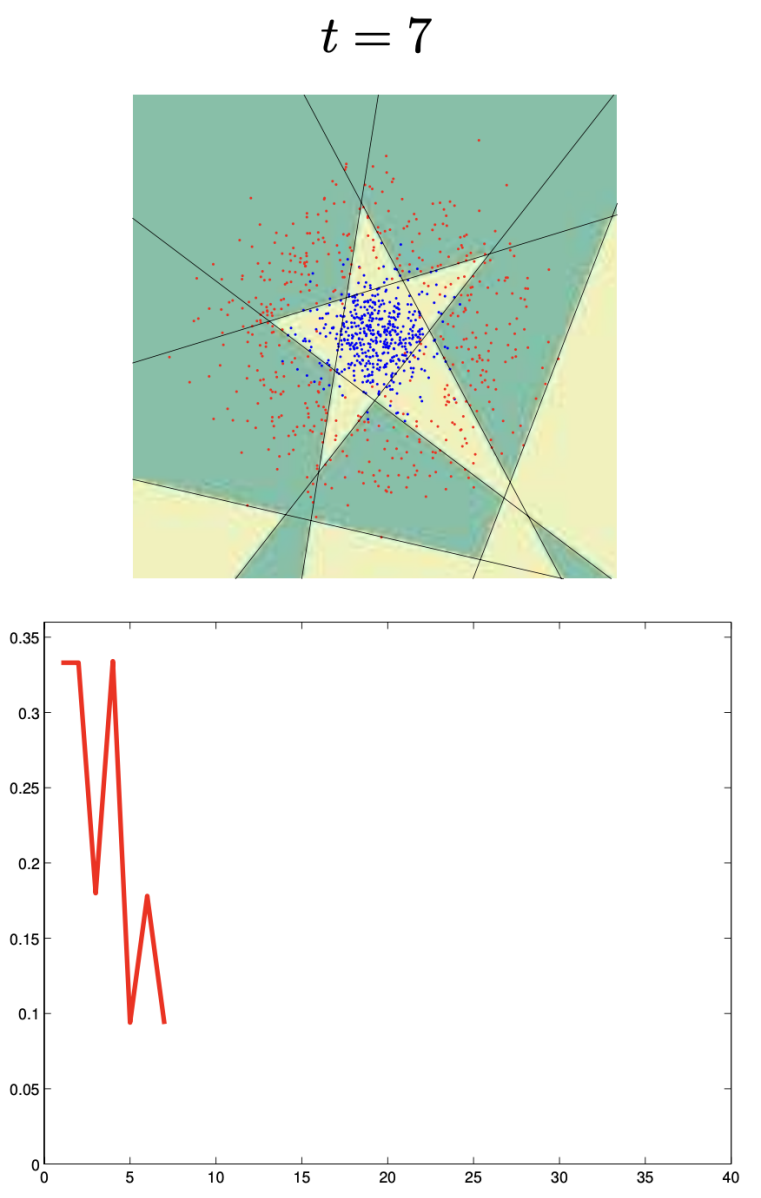
- Binary Classifier:
  $F_1(x) = sign(H_1(x))$
- $H_2(x) = h_1(x) + h_2(x)$
- Binary Classifier:
  $F_2(x) = sign(H_2(x))$
- ...



Example from this Adaboost class

An Example of Boosting

- Binary Classifier:
  $F_1(x) = sign(H_1(x))$
- $H_2(x) = h_1(x) + h_2(x)$
- Binary Classifier:
  $F_2(x) = sign(H_2(x))$
- ...

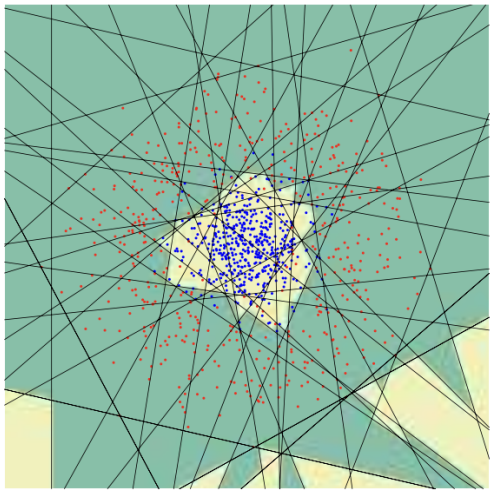


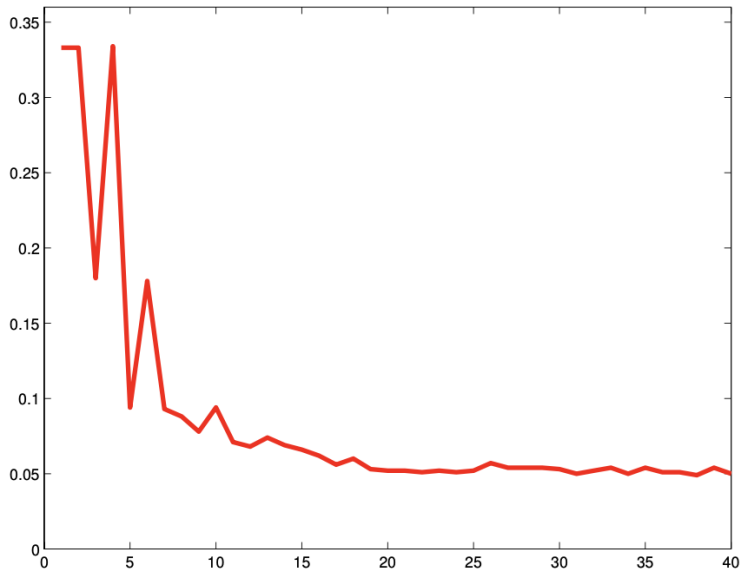
Example from this Adaboost class

An Example of Boosting

- Binary Classifier:
  $F_1(x) = sign(H_1(x))$
- $H_2(x) = h_1(x) + h_2(x)$
- Binary Classifier:
  $F_2(x) = sign(H_2(x))$
- ...
- Binary Classifier:
  $F_n(x) = sign(H_{n-1}(x) + h_n(x))$

$t = 40$





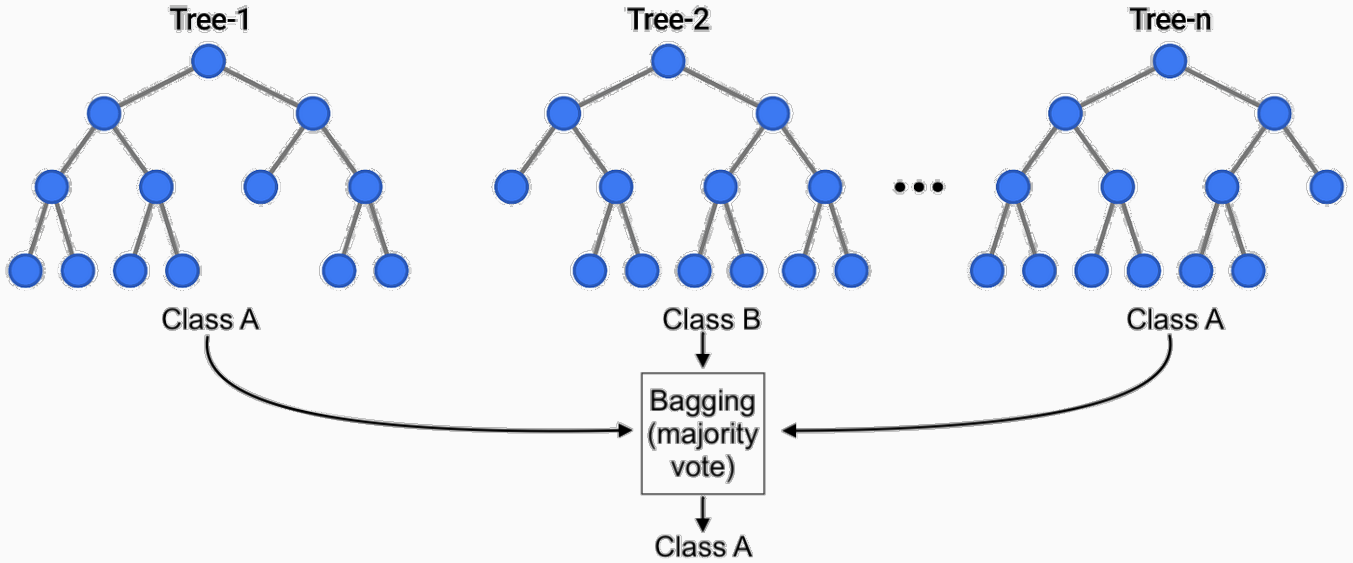
Example from this Adaboost class

Outline : Random Forests

Random Forests

Algorithm

- F total features, training set $\mathcal{S}_{train}$
- For $t = 1 \ldots T$:
  - $\mathcal{S}_{train}^{(t)} \leftarrow$ sample n_t points from $\mathcal{S}_{train}$ with replacement (bootstrap)
  - $h_{tree}^{(t)} \leftarrow$ train a random decision tree on $\mathcal{S}_{train}^{(t)}$
- Final prediction: $h^T = \frac{1}{T} \sum_t h_{tree}^{(t)}$

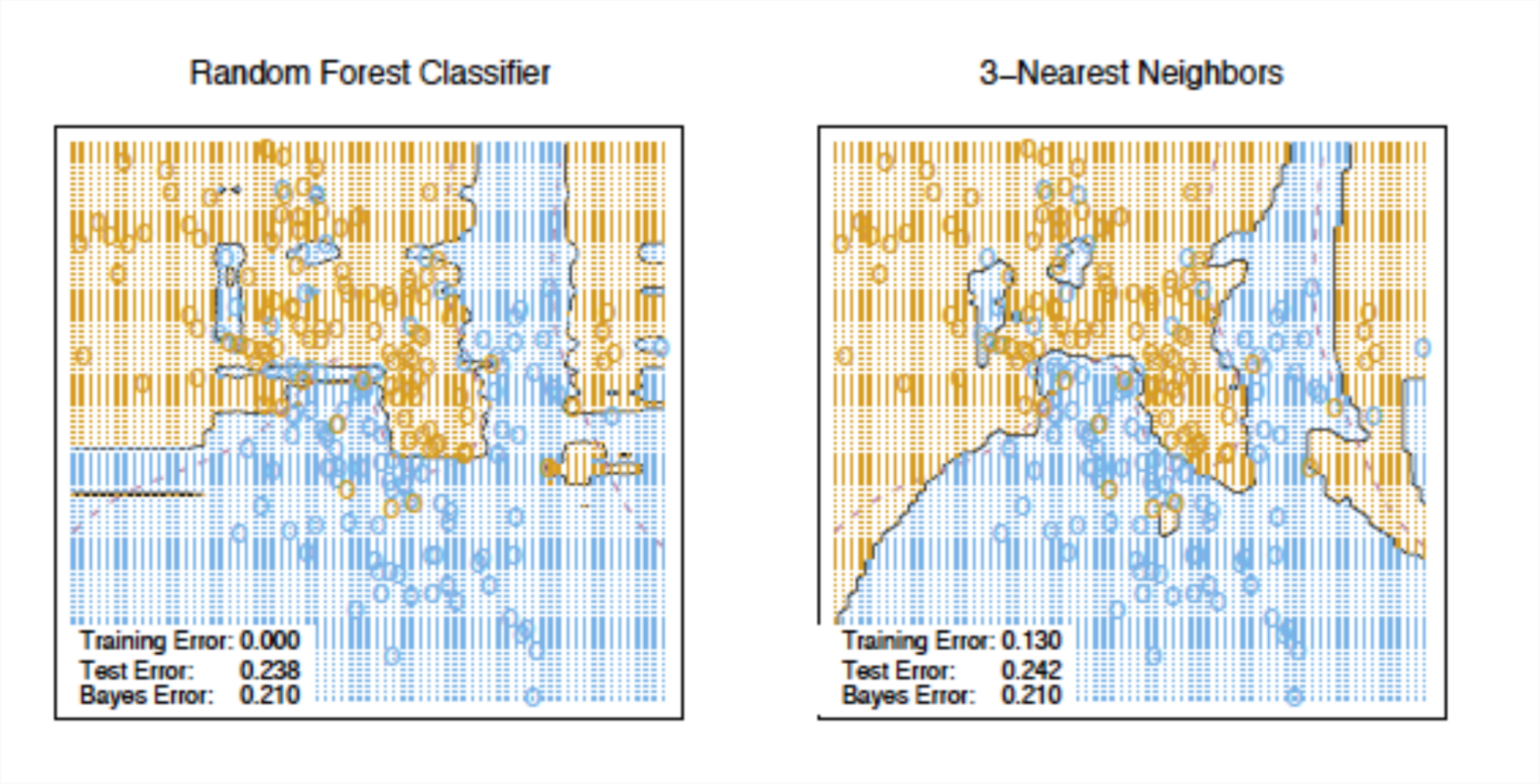


Random Forests: Learning a Tree

Learning a single random decision tree:

- For each node split:
  - $R_k(F)$: randomly (without replacement) select k features from the F total features, with $k \ll F$
  - Find the best split over these k features
- Do not prune the tree

Random Forests: Visualization



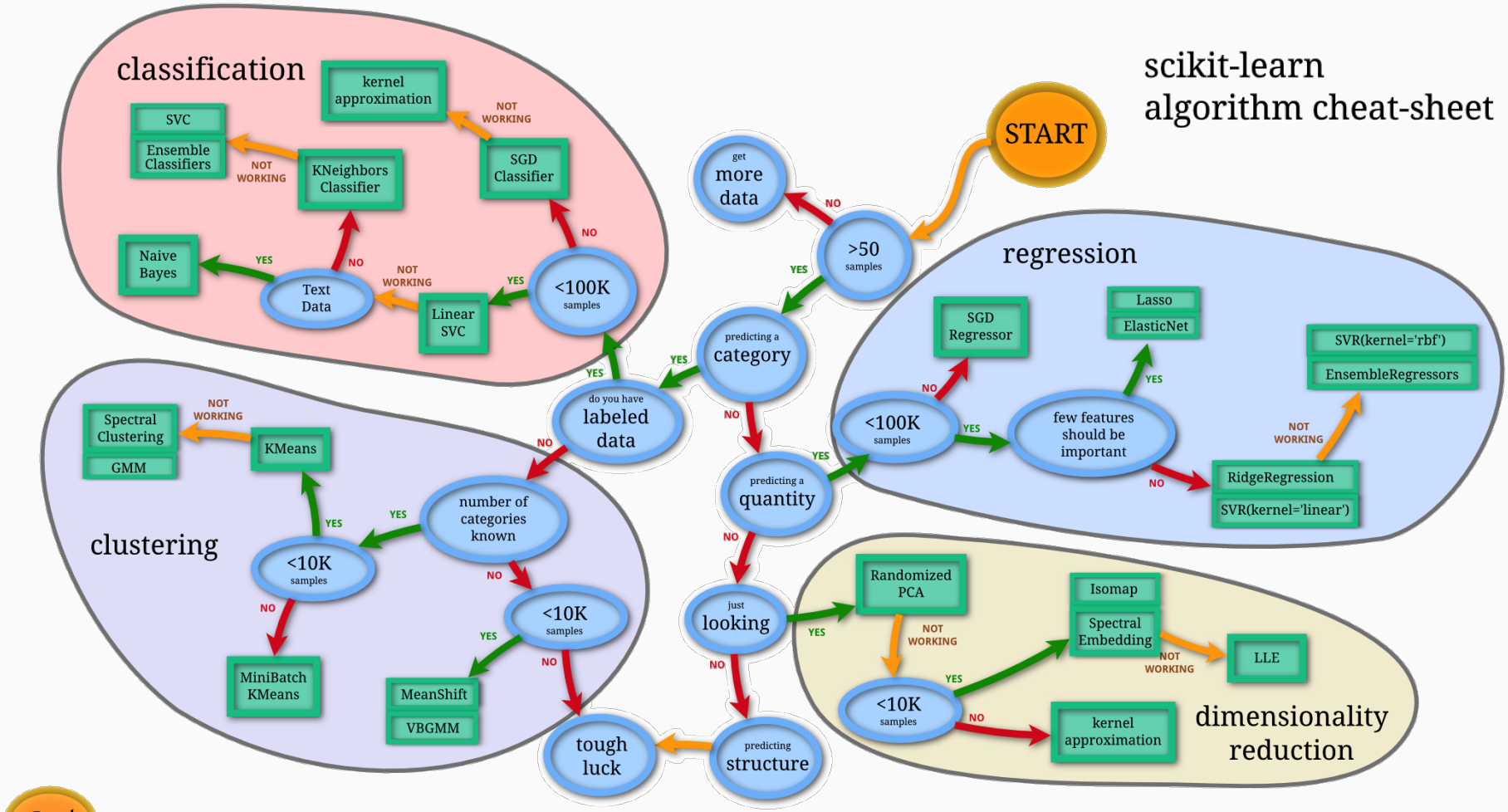
Summary

Algo	Type	Tolerance number features	Parametrization	Memory size	Minimal required quantity	Com m	Overfitting Tendency	Difficulty	Time for Learning	Time for predicting
Linear Regression	R	Weak	Weak	Small	Small	++	Low	Weak	Weak	Weak
Logistic Regression	C	Weak	Simple	Small	Small	++	Low	Weak	Weak	Weak
Decision Tree	R & C	Strong	Simple / intuitive	Large	Small	+++	Very high	Weak	Weak	Weak
Random Forest	R & C	Strong	Simple / intuitive	Very Large	Large	++	Average	Average	Costly	Costly
Boosting	R & C	Strong	Simple / intuitive	Very Large	Large	+	Average	Average	Costly	Weak
Naive Bayes	C	Weak	No params.	Small	Small	++	Low	Weak	Weak	Weak
SVM	C	Very strong	Not intuitive	Small	Large	--	Average	High	Costly	Weak
Neural Network (NN)	C	Very strong **	Not intuitive	Inter	Large	---	Average	Very high	Costly	Weak
Deep Neural Network	C	Very strong **	Not intuitive	Very Large	Very Large	---	High	Very high	Very costly	Weak
K-Means	CL*	Strong	Simple / Intuitive		Small	+		High	Weak	
One class SVM	A	Very strong	Not intuitive	Weak	Large	--	Average	High	Costly	Weak

A summary of main ML models

Another comparison table

Summary: Scikit-learn Cheatsheet



Regression Trees

Only change the impurity function H to suit regression problems.

Maximize variance reduction:

$$L(t_{j,\tau}, \mathcal{S}) = VAR_{emp}(\mathcal{S}) - \frac{|\mathcal{D}(\mathcal{S}, j, \tau)|}{n} VAR_{emp}(\mathcal{D}(\mathcal{S}, j, \tau)) - \frac{|\mathcal{I}(\mathcal{S}, j, \tau)|}{n} VAR_{emp}(\mathcal{I}(\mathcal{S}, j, \tau))$$

Where:

$$VAR_{emp}(\mathcal{S}) = \frac{1}{|\mathcal{S}|} \sum_{(x^i, y^i) \in \mathcal{S}} (y^i - \bar{y})^2$$

We attempt to get homogeneous outputs!

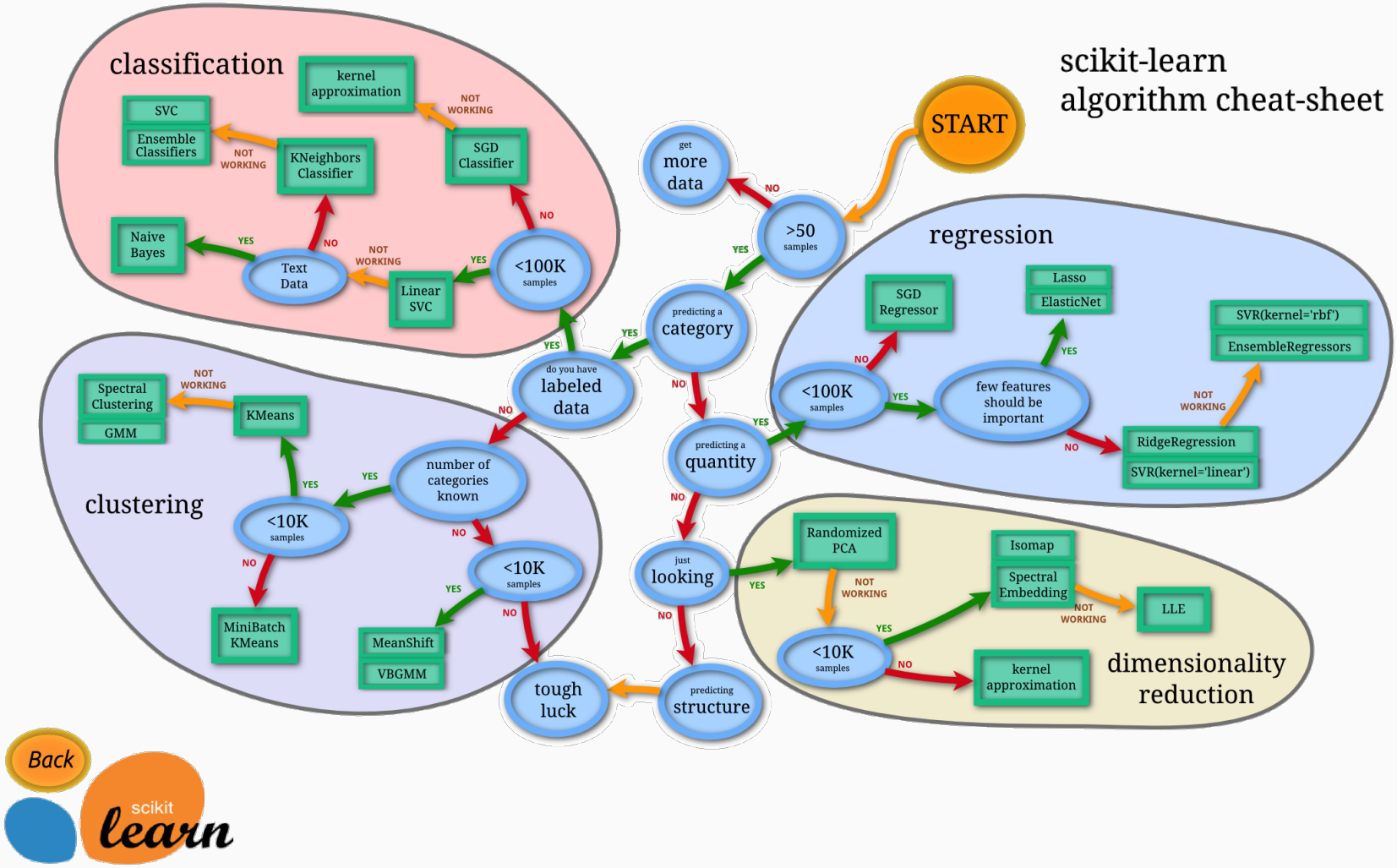
API : Scikit-learn

Outline : API : Scikit-learn

Scikit-learn: A Python Library

A very user-friendly Machine Learning library:
https://scikit-learn.org/

Scikit-learn: Possibility Map



Scikit-learn: Standardized Functions

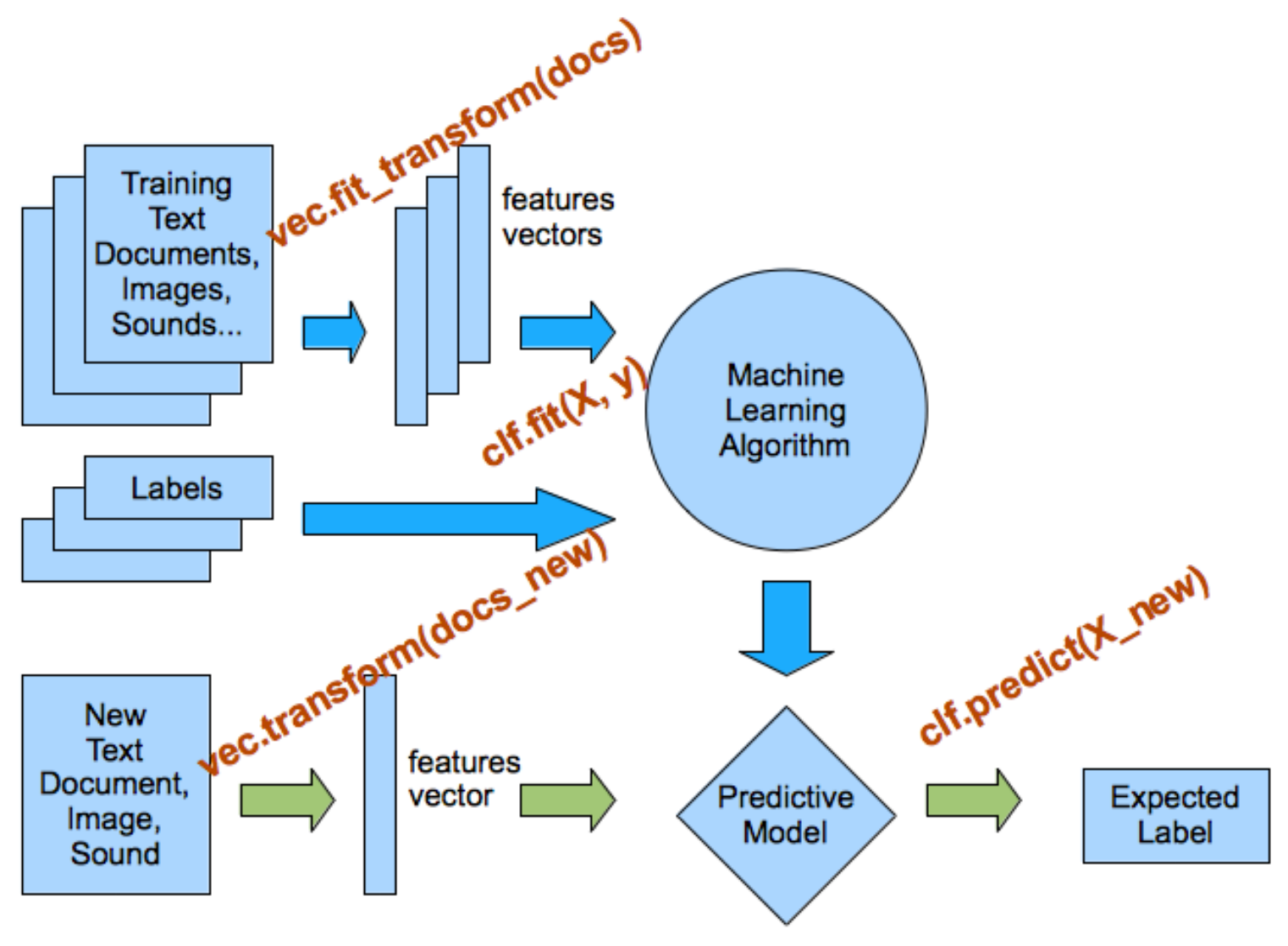
Standardized functions for fast learning and code clarity:

- Different models (or datasets) are implemented as classes (data types)
- Methods can be called on these objects for different stages of the algorithm: Feature extraction, Parameter learning, Prediction, etc.

```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> X, y = data.data, data.target
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     iris.data, iris.target, test_size=0.4, random_state=0)
>>> from sklearn.linear_model import SGDClassifier
>>> clf = SGDClassifier(max_iter=1000, tol=1e-3)
>>> clf.fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.92...
```

Scikit-learn: Standardized Functions

Standardized functions for fast learning and code clarity:



Scikit-learn: Datasets

Multiple available datasets :

- Toy datasets: Boston housing prices, Iris, Diabetes, Wine recognition, etc.
- Real-world datasets: Olivetti Faces (images), Forest cover types (geographic), RCV1 dataset (text), etc.

```
>>> from sklearn.datasets import load_wine
>>> data = load_wine()
>>> data.keys()
['target_names', 'data', 'target', ...
'DESCR', 'feature_names']
```

Scikit-learn: Feature Extraction

Pre-coded functions to easily vectorize features

- From dictionaries: `sklearn.feature_extraction`
- From images: `sklearn.feature_extraction.image`
- From text: `sklearn.feature_extraction.text`

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third.',
...     'Is this the first document?',
... ]
>>> X = vectorizer.fit_transform(corpus).toarray()
array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 2, 1, 0, 1],
       [1, 0, 0, 0, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 0, 0, 1, 0, 1]]...)
```

Scikit-learn: Feature Selection

Techniques to select features

- Remove low-variance features: `VarianceThreshold`
- Use models to select features: `SelectFromModel`
- Recursive Feature Elimination: `RFE`

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> lsvc = LinearSVC(C=0.01, penalty="l1", dual=False).fit(X, y)
>>> model = SelectFromModel(lsvc, prefit=True)
>>> X_new = model.transform(X)
>>> X_new.shape
(150, 3)
```

Scikit-learn: Data Preprocessing

Normalization, standardization necessary for better learning

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
scaler.transform(X_test)
```

Missing values

```
>>> from sklearn.impute import SimpleImputer
>>> imp = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
SimpleImputer(copy=True, fill_value=None, missing_values=nan, strateg
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[4.          2.        ]
 [6.          3.666...]
 [7.          6.        ]]
```

Scikit-learn: Cross-Validation

Functions for model validation and splitting into training/test sets:

- Simply using `train_test_split`, for K-Fold cross-validation `KFold`, for Leave-One-Out `LeaveOneOut`, preserving label proportions with `StratifiedKFold`
- Direct cross-validation with `cross_val_score`

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     iris.data, iris.target, test_size=0.4, random_state=0)

>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))
```

Scikit-learn: Supervised Learning Models

Numerous standardized supervised models for easy use:

- Logistic/Linear Regression, Linear/Quadratic Discriminant Analysis, SVM, K-NN, Stochastic Gradient Descent, Naive Bayes, Decision Trees, Random Forests, etc.
- Each of these models is implemented as a Python class with common methods: `fit`, `score`, `predict_proba`, `predict`, etc.

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegression
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegression(random_state=0, solver='lbfgs',
...                          multi_class='multinomial').fit(X, y)
>>> clf.predict(X[:2, :])
array([0, 0])
>>> clf.predict_proba(X[:2, :])
array([[9.8...e-01, 1.8...e-02, 1.4...e-08],
       [9.7...e-01, 2.8...e-02, ...e-08]])
>>> clf.score(X, y)
0.97...
```

Scikit-learn: Unsupervised Learning Models

Numerous standardized unsupervised models for clustering, component analysis, dimensionality reduction:

- Clustering: K-Means, PCA, Non-negative Matrix Factorization, Latent Dirichlet Allocation, t-SNE, ...
- These models are implemented as Python classes with common methods: `fit`, `fit_transform`, `predict`, etc.

```
>>> from sklearn.cluster import KMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [4, 2], [4, 4], [4, 0]])
>>> kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
>>> kmeans.labels_
array([0, 0, 0, 1, 1, 1], dtype=int32)
>>> kmeans.predict([[0, 0], [4, 4]])
array([0, 1], dtype=int32)
>>> kmeans.cluster_centers_
array([[1., 2.],
       [4., 2.]])
```

Scikit-learn: Metrics

Multiple available metrics for model evaluation:

- For classification: Recall, Precision, F1, AUC, ROC, Confusion Matrix, etc.
- For regression: R^2, Mean Squared Error, etc.
- For clustering: Completeness, V-measure, etc.

```
>>> import numpy as np
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
```

Scikit-learn: Hyperparameter Search

Functions to find optimal hyperparameters using cross-validation for different hyperparameters.

- Grid search: `GridSearchCV`
- Random search: `RandomizedSearchCV`
- These models are implemented as Python classes with common methods: `fit`, `fit_transform`, `predict`, etc.

```
>>> from sklearn import svm, datasets
>>> from sklearn.model_selection import GridSearchCV
>>> iris = datasets.load_iris()
>>> parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
>>> svc = svm.SVC(gamma="scale")
>>> clf = GridSearchCV(svc, parameters, cv=5)
>>> clf.fit(iris.data, iris.target)
```

Questions?

References i