Data-driven Decision Making

Machine Learning: more fundamental algorithms

João Pedro Pedroso

2024/2025

Last classes:

- Introduction do machine learning
- Fundamental algorithms

Today

- Building Blocks of a Learning Algorithm:
  - loss function;
  - optimization criterion based on the loss function
  - optimization method to find a solution
  - Basic practice

Anatomy of a Learning Algorithm

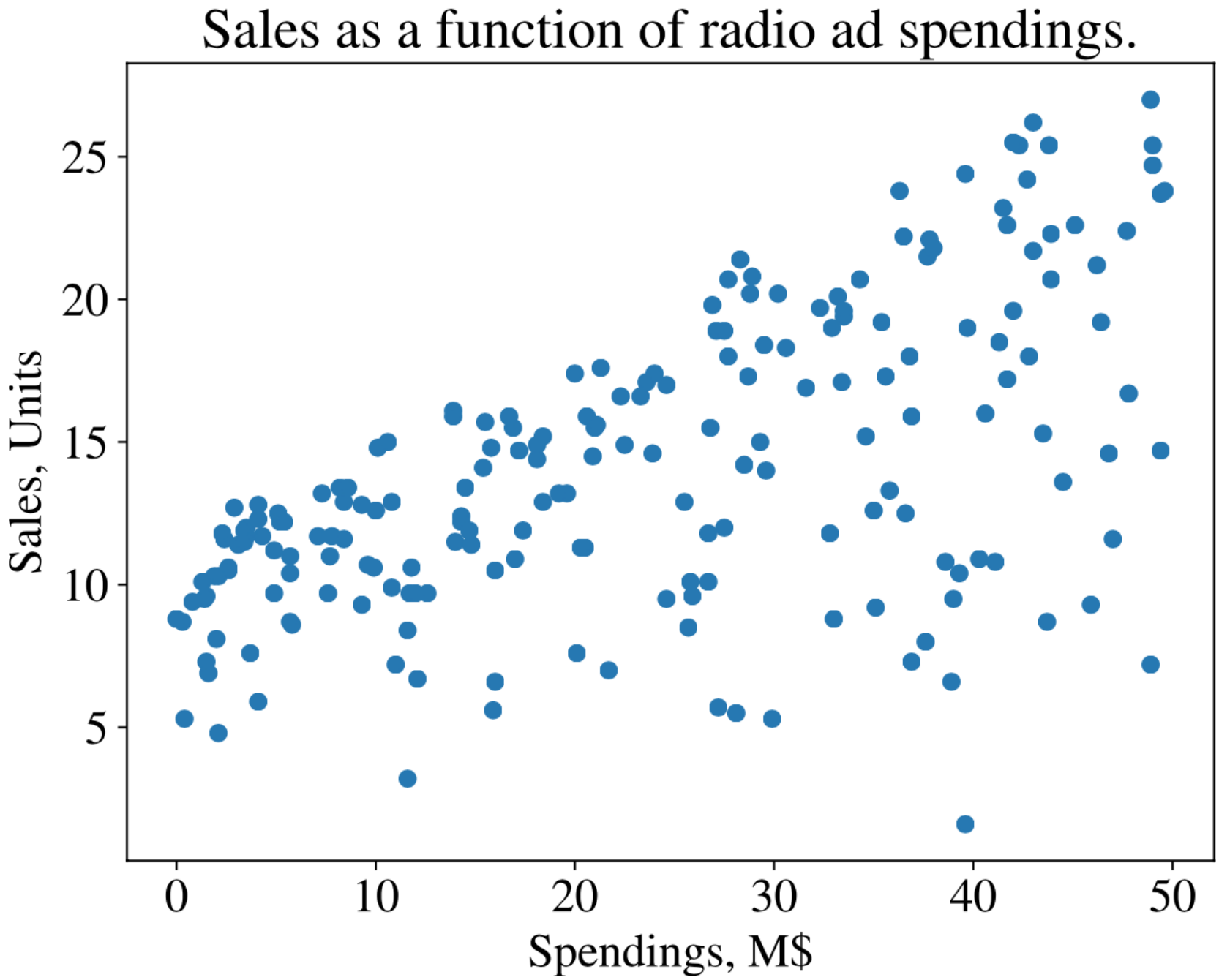
Anatomy of a Learning Algorithm

- Building Blocks of a Learning Algorithm:
  1. loss function;
  2. optimization criterion based on the loss function
     - cost function
  3. optimization method to find a solution
- Some algorithms: designed to explicitly optimize a specific criterion
  - linear and logistic regressions, SVM
- Others optimize the criterion implicitly
  - decision tree learning and kNN:
    - among the oldest machine learning algorithms
  - were invented experimentally based on intuition
  - specific global optimization criteria were developed later to explain why those algorithms work
- Frequently used optimization algorithms: gradient descent
  - iterative algorithm for finding local minimum of a function
  - start at some random point
  - take steps proportional to the negative of the gradient at the current point

Gradient Descent

- illustration to find solution of linear regression
- example based on dataset with one feature
  - two parameters: w, b
- dataset:
  - spendings of various companies on radio advertising each year
  - their annual Sales in terms of units sold
- regression model: predict units sold based on how much a company spends on radio advertising
  - data and programs available in book's support page

Company	Spendings, M$	Sales, Units
1	37.8	22.1
2	39.3	10.4
3	45.9	9.3
4	41.3	18.5
...	...	...


Sales as a function of radio ad spendings.
Sales, Units
25
20
15
10
5
0
10
20
30
40
50
Spendings, M$

Gradient descent: *loss function*

- Loss function

$$l \stackrel{\text{def}}{=} \frac{1}{N}\sum_{i=1}^{N}(y_i - (wx_i + b))^2$$

- Partial derivative for each parameter:
  - indicate direction of function growth

$$\frac{\partial l}{\partial w} = \frac{1}{N}\sum_{i=1}^{N} -2x_i(y_i - (wx_i + b))$$

$$\frac{\partial l}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} -2(y_i - (wx_i + b))$$

Gradient descent: *epochs*

- Epoch: using the training set entirely to update each parameter:
- Initilizing $w \leftarrow 0, b \leftarrow 0$, then in an epoch update:

$$w \leftarrow w - \alpha \frac{\partial l}{\partial w}$$
$$b \leftarrow b - \alpha \frac{\partial l}{\partial b}$$

- $\alpha \rightarrow$ learning rate
- at next epoch:
  - recalculate partial derivatives
  - updata again w, b
- continue until convergence
  - w, b don't change much
  - then, stop

Python code: updating w, b

```
def update_w_and_b(spendings, sales, w, b, alpha):
    dl_dw = 0.0
    dl_db = 0.0
    N = len(spendings)
    for i in range(N):
        dl_dw += -2*spendings[i]*(sales[i] - (w*spendings[i] + b))
        dl_db += -2*(sales[i] - (w*spendings[i] + b))
    # update w and b
    w = w - (1/float(N))*dl_dw*alpha
    b = b - (1/float(N))*dl_db*alpha
    return w, b
```

Python code: looping over epochs

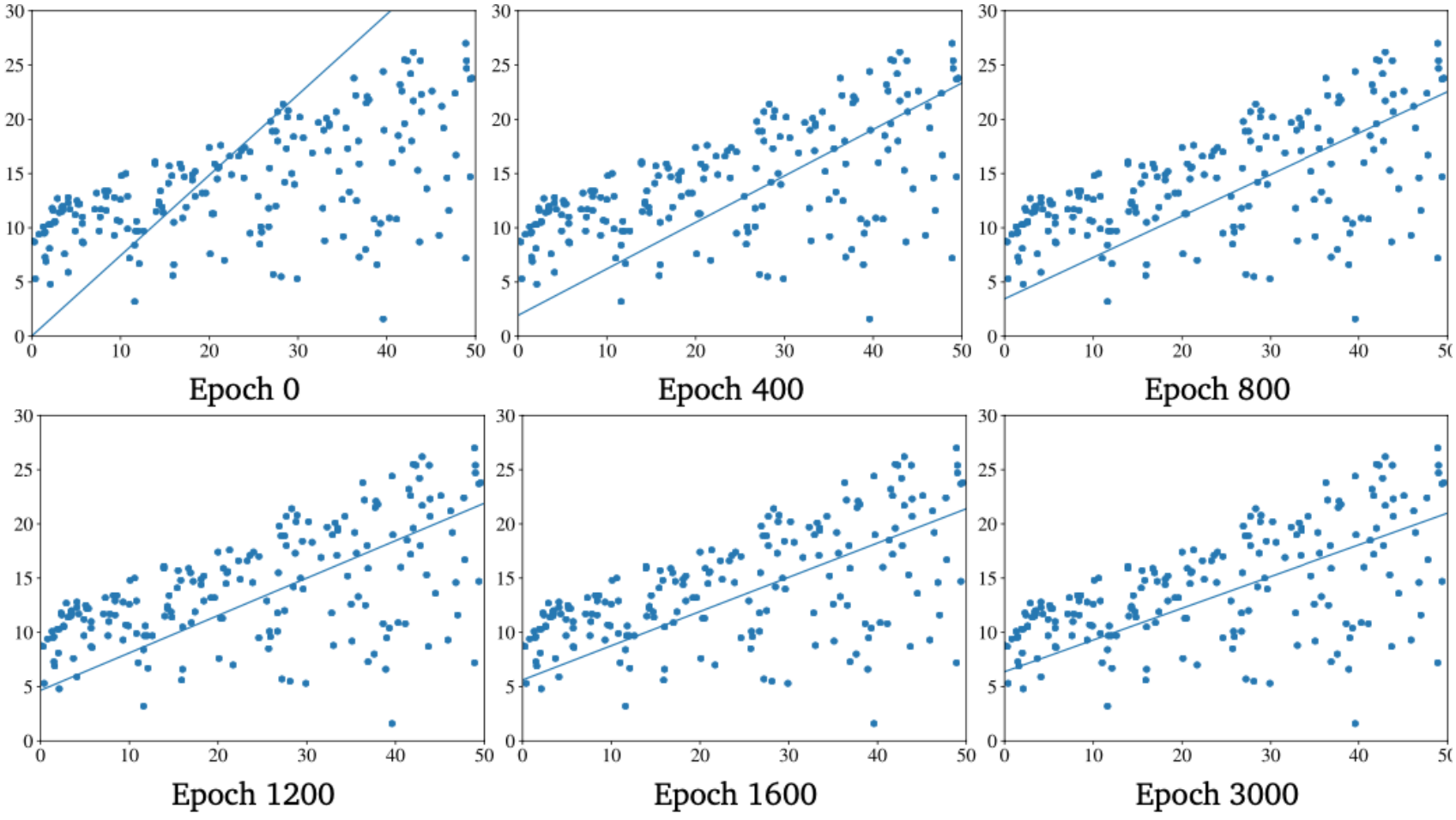
```
def avg_loss(spendings, sales, w, b):
    N = len(spendings)
    total_error = 0.0
    for i in range(N):
         total_error += (sales[i] - (w*spendings[i] + b))**2
    return total_error / float(N)


def train(spendings, sales, w, b, alpha, epochs):
    for e in range(epochs):
        w, b = update_w_and_b(spendings, sales, w, b, alpha)
        # log the progress
        if e % 400 == 0:
            print("epoch:", e, "loss: ", avg_loss(spendings, sales, w, b))
    return w, b
```

Python code: make predictions

```
def predict(x, w, b):
    return w*x + b
```

Can be called with

```
w, b = train(x, y, 0.0, 0.0, 0.001, 15000)
x_new = 23.0
y_new = predict(x_new, w, b)
print(y_new)
```


Epoch 0
Epoch 400
Epoch 800
Epoch 1200
Epoch 1600
Epoch 3000

Remarks

- Gradient descent:
  - sensitive to the choice of learning rate α
  - slow for large datasets
- Many improved variants:
  - Minibatch stochastic gradient descent
    - speeds up computation by approximating the gradient using smaller batches of the training data
  - Adagrad
    - version of SGD that scales α for each parameter according to the history of gradients
    - α reduced for very large gradients and vice-versa
  - Momentum
    - accelerates SGD by orienting the gradient descent in the relevant direction and reducing oscillations
  - specific versions for training neural neural network
    - *backpropagation*
- Gradient descent and variants are not machine learning algorithms
  - just solvers of minimization problems
  - can be used when function to minimize has a gradient

Using Machine Learning

- Usually, machine learning users do not implement machine learning algorithms themselves
  - use libraries, most of which are open source
  - most frequently used in practice: `scikit-learn`
  - many algorithms available
  - can be user in a coherent way

```
from sklearn.linear_model import LinearRegression
# we could also try: from sklearn import neighbors

def train(x, y):
    model = LinearRegression().fit(x,y)
    # or: model = neighbors.KNeighborsRegressor(n_neighbors=5).fit(x,y)
return model

model = train(x,y)
x_new = 23.0
y_new = model.predict(x_new)
print(y_new)
```

Learning Algorithms' Particularities

What differentiates one learning algorithm from another?

- Some algorithms can accept categorical features
  - *e.g.*, decision tree learning
  - others expect numerical values for all features
- All algorithms implemented in `scikit-learn` expect numerical features
  - next class: how to convert categorical into numerical features

Classification models

- Some algorithms allow providing weightings for each class
  - *e.g.*, SVM
  - influence how the decision boundary is drawn
    - high $\rightarrow$ try to not make errors in predicting training examples of this class
  - important if
    - instances of some class are in the minority in training data
    - but we want to avoid misclassifying examples of that class
- Some only output the class
  - *e.g.*, SVM, kNN
- Other can also return the score between 0 and 1
  - *e.g.*, logistic regression, decision trees
  - interpreted as *how confident the model is about the prediction*
    - probability that the input example belongs to a certain class

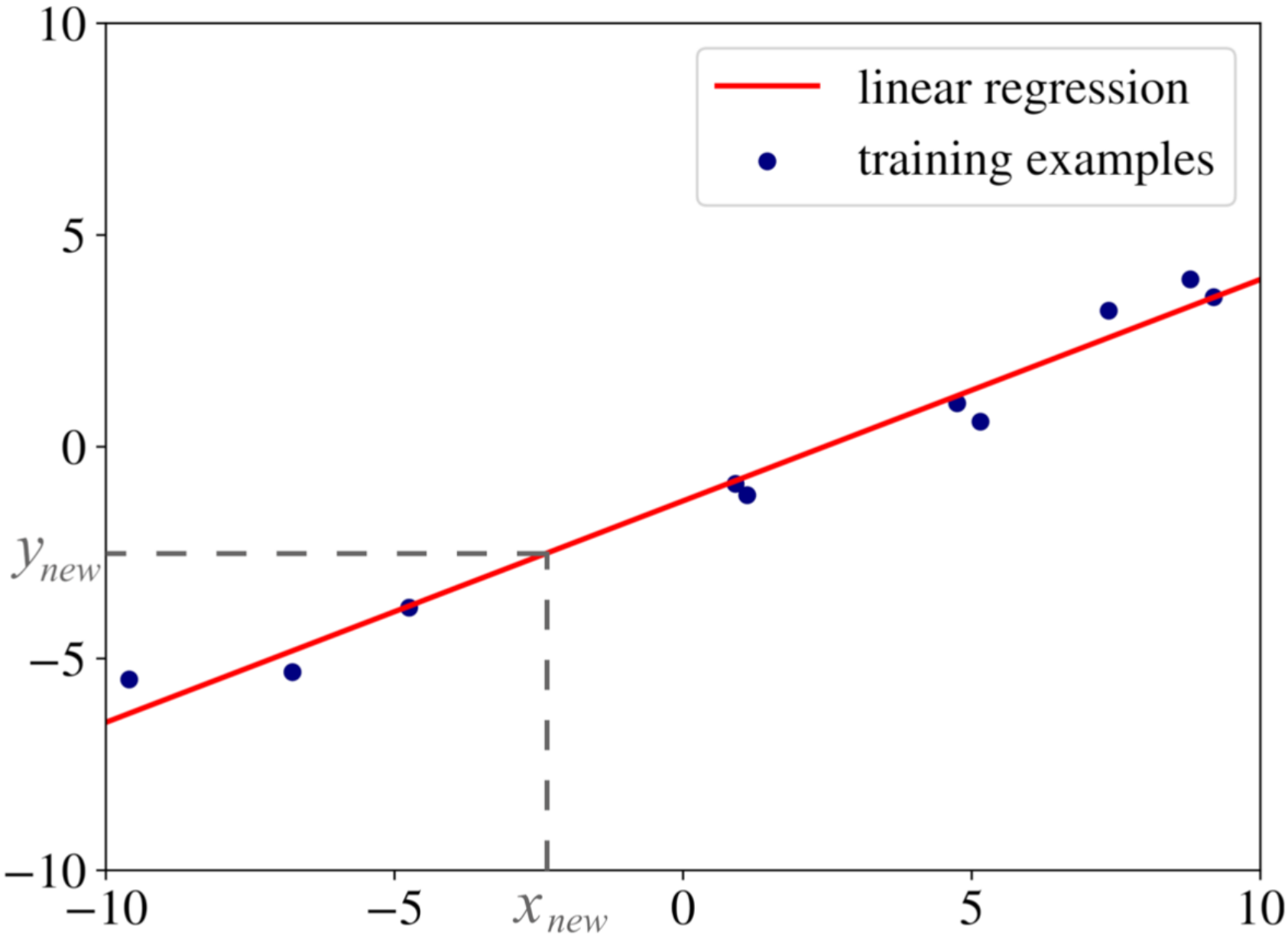
Supervised learning

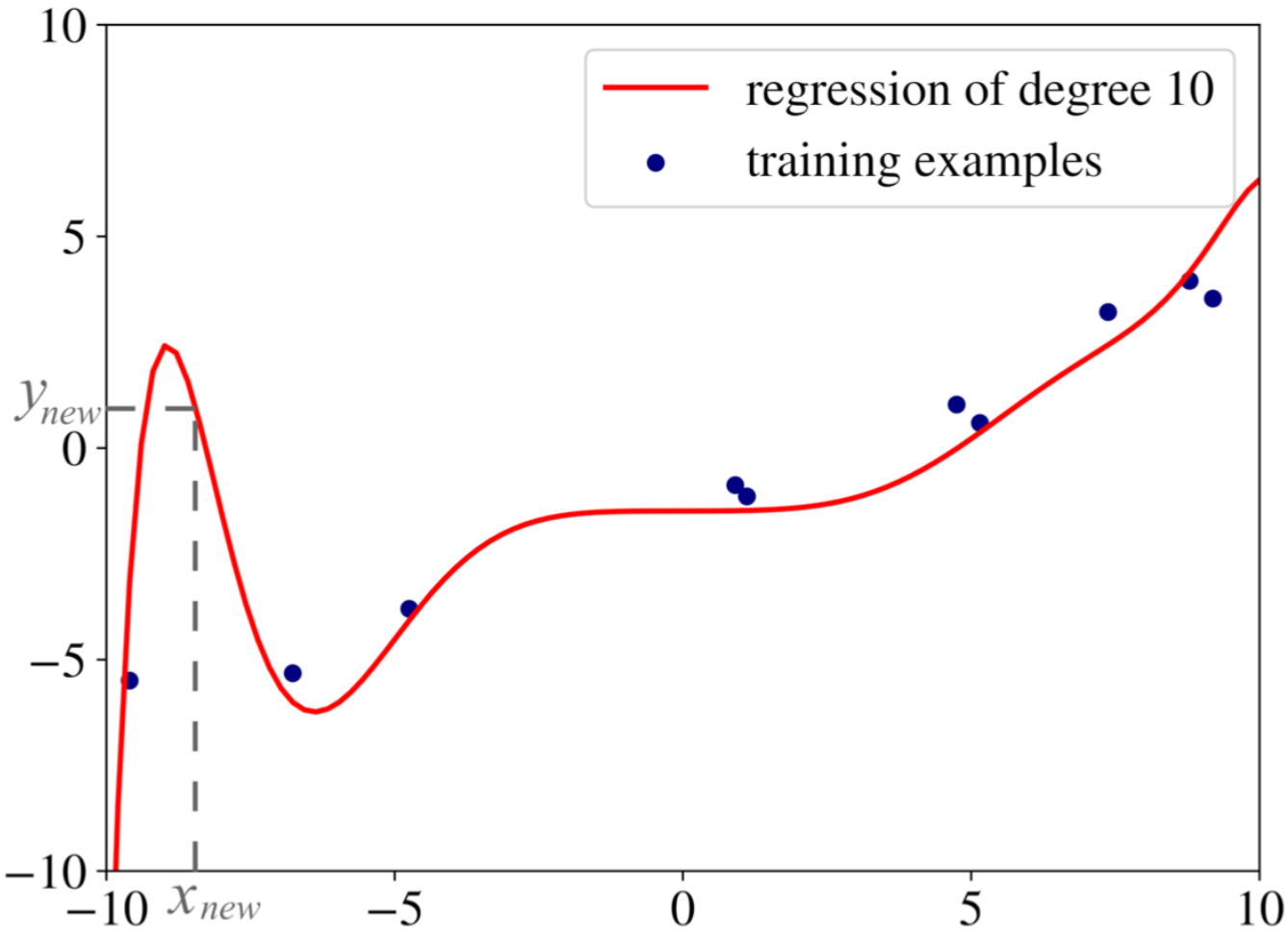
- Some build the model using the whole dataset at once
  - *e.g.*, decision tree learning, logistic regression, SVM
  - if we have got additional examples $\rightarrow$ rebuild model from scratch
- Other algorithms can be trained iteratively, one batch at a time
  - *e.g.*, Naïve Bayes, multilayer perceptron
  - once new training examples are available, we can update model using only the new data
- Some algorithms can be used for both classification and regression
  - *e.g.*, decision tree learning, SVM, kNN
- Other can only solve one type of problem
  - either classification or regression, but not both

Using libraries

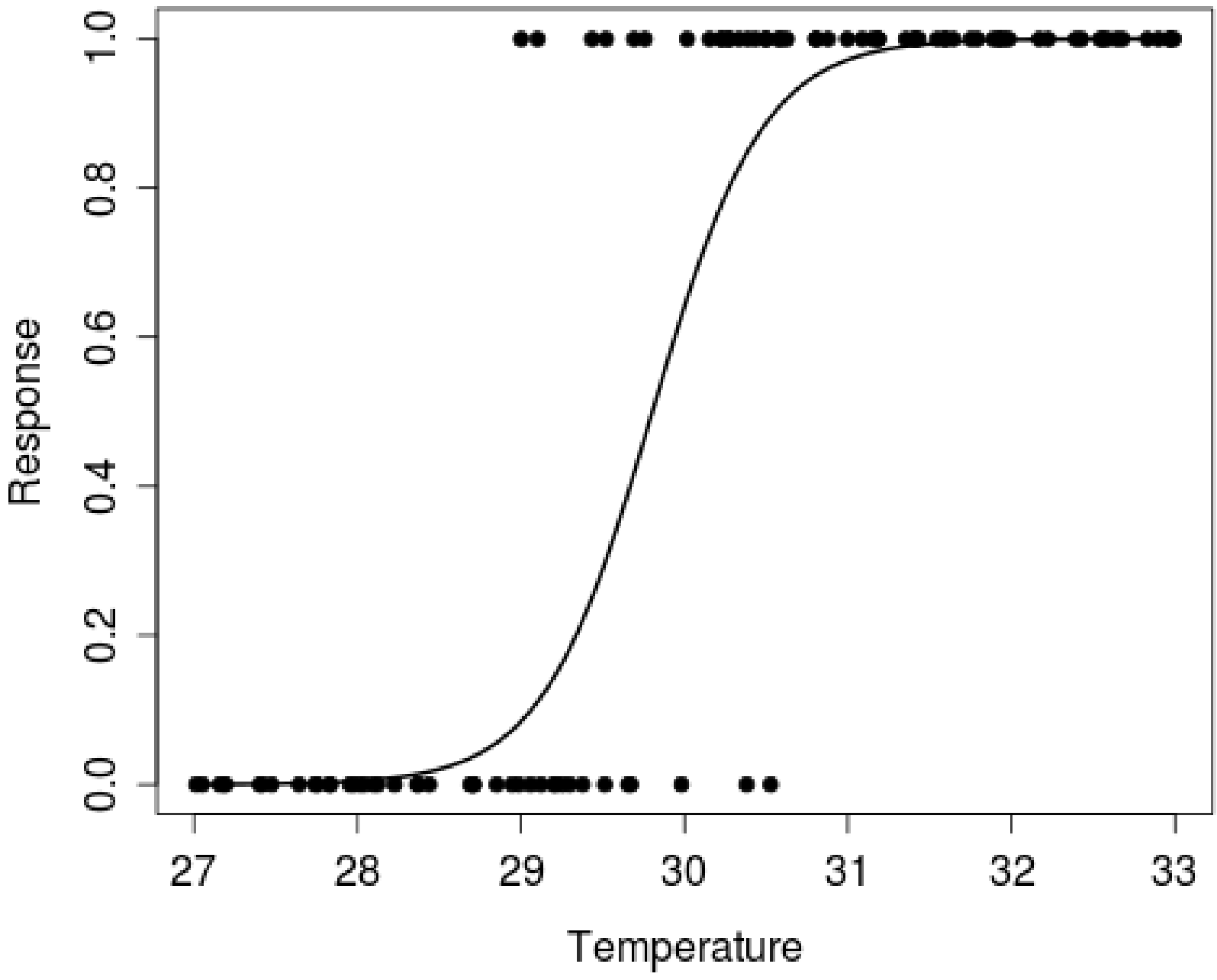
- Usually, libraries provides documentation, explaining
  - what kind of problem each algorithm solves
  - what input values are allowed
  - what kind of output the model produces
  - also, information on hyperparameters

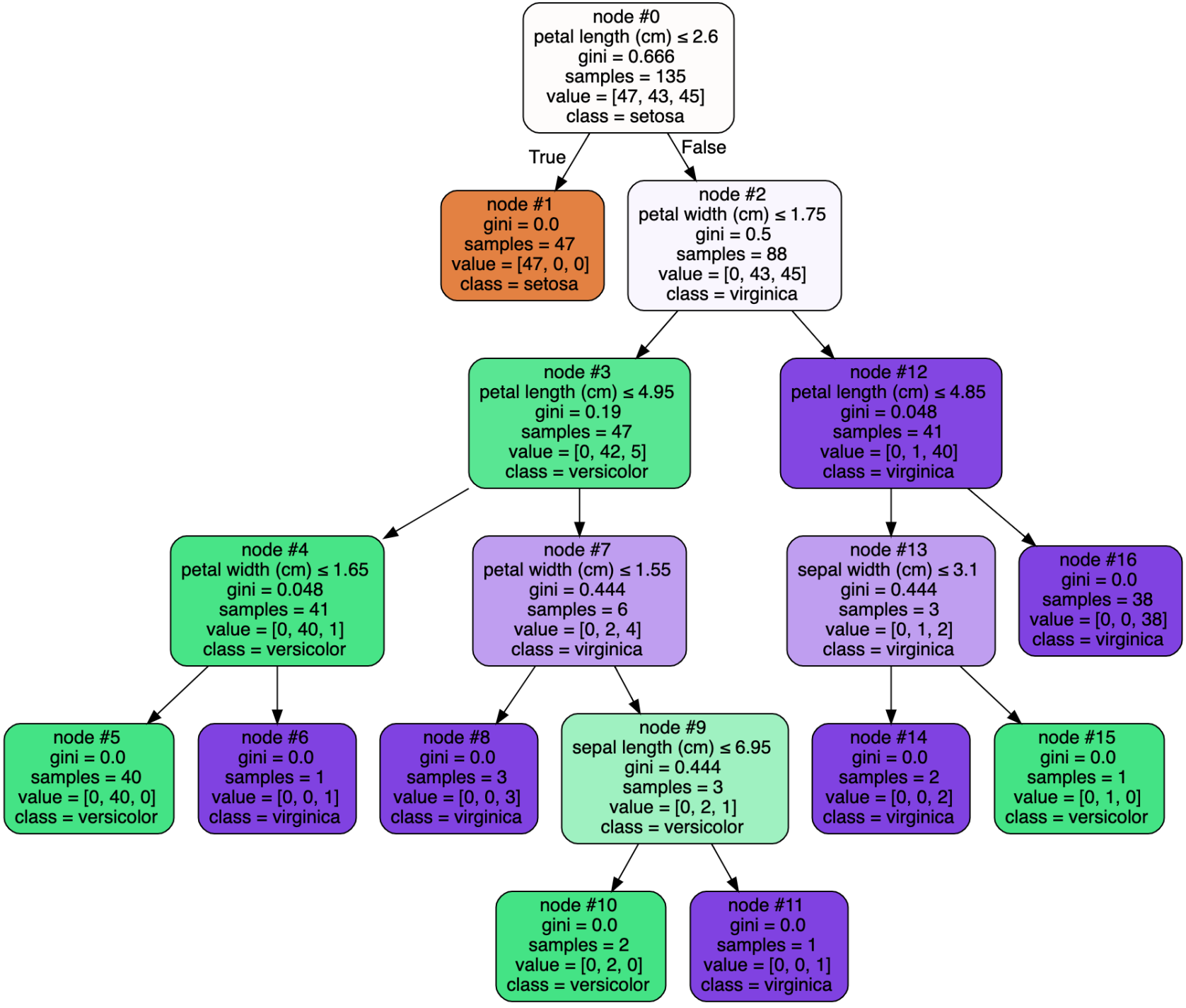
Summary

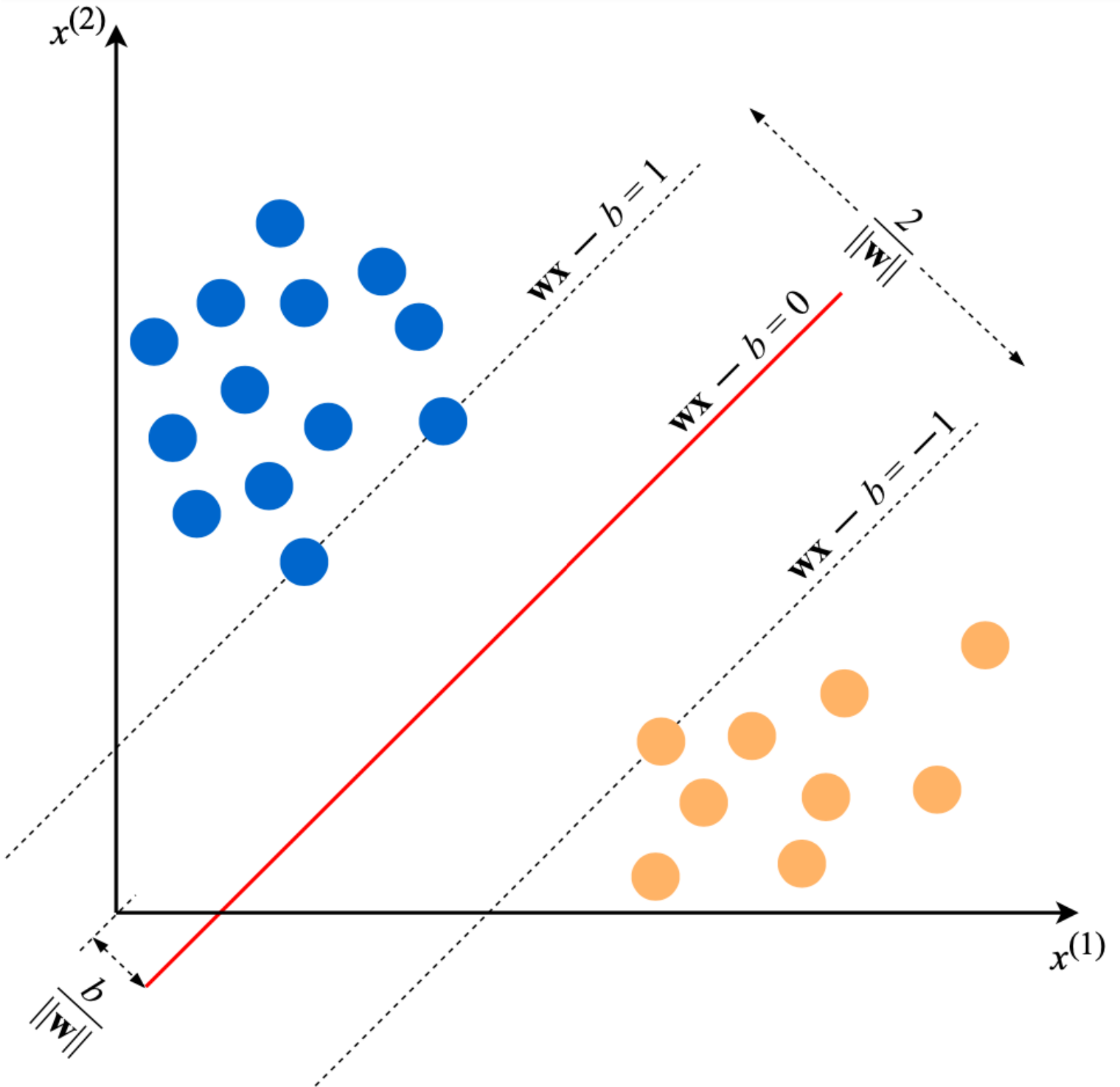

linear regression
training examples
10
5
0
−5
−10
−10
−5
0
5
10
y_{new}
x_{new}

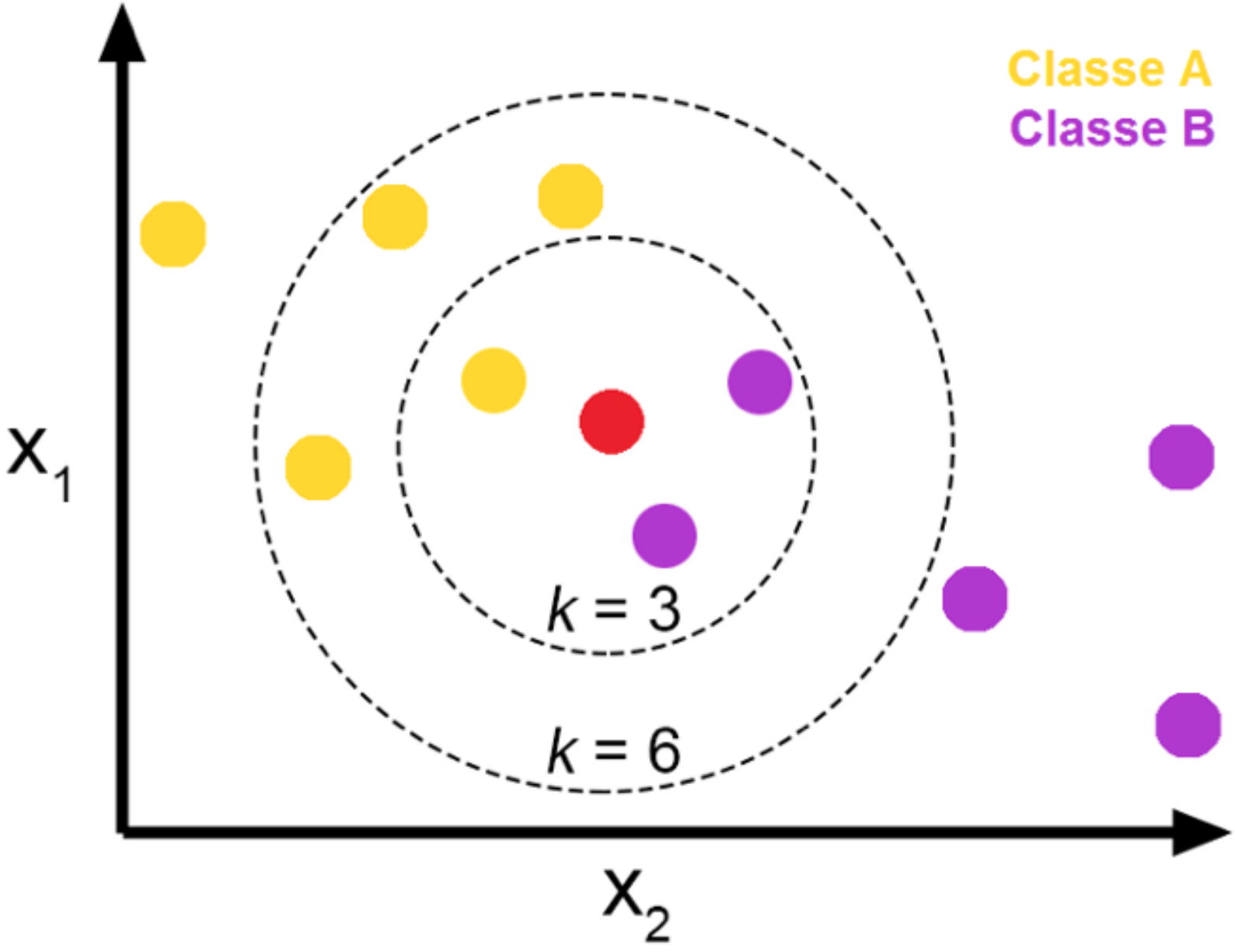


- *more powerful models* $\Rightarrow$ *higher risk of overfitting*


Response
0.0
0.2
0.4
0.6
0.8
1.0
27
28
29
30
31
32
33
Temperature


node #0
petal length (cm) ≤ 2.6
gini = 0.666
samples = 135
value = [47, 43, 45]
class = setosa
True
False
node #1
gini = 0.0
samples = 47
value = [47, 0, 0]
class = setosa
node #2
petal width (cm) ≤ 1.75
gini = 0.5
samples = 88
value = [0, 43, 45]
class = virginica
node #3
petal length (cm) ≤ 4.95
gini = 0.19
samples = 47
value = [0, 42, 5]
class = versicolor
node #12
petal length (cm) ≤ 4.85
gini = 0.048
samples = 41
value = [0, 1, 40]
class = virginica
node #4
petal width (cm) ≤ 1.65
gini = 0.048
samples = 41
value = [0, 40, 1]
class = versicolor
node #7
petal width (cm) ≤ 1.55
gini = 0.444
samples = 6
value = [0, 2, 4]
class = virginica
node #13
sepal width (cm) ≤ 3.1
gini = 0.444
samples = 3
value = [0, 1, 2]
class = virginica
node #16
gini = 0.0
samples = 38
value = [0, 0, 38]
class = virginica
node #5
gini = 0.0
samples = 40
value = [0, 40, 0]
class = versicolor
node #6
gini = 0.0
samples = 1
value = [0, 0, 1]
class = virginica
node #8
gini = 0.0
samples = 3
value = [0, 0, 3]
class = virginica
node #9
sepal length (cm) ≤ 6.95
gini = 0.444
samples = 3
value = [0, 2, 1]
class = versicolor
node #14
gini = 0.0
samples = 2
value = [0, 0, 2]
class = virginica
node #15
gini = 0.0
samples = 1
value = [0, 1, 0]
class = versicolor
node #10
gini = 0.0
samples = 2
value = [0, 2, 0]
class = versicolor
node #11
gini = 0.0
samples = 1
value = [0, 0, 1]
class = virginica


$x^{(2)}$
$\mathbf{wx} - b = 1$
$\frac{2}{\|\mathbf{w}\|}$
$\mathbf{wx} - b = 0$
$\mathbf{wx} - b = -1$
$x^{(1)}$
$\frac{b}{\|\mathbf{w}\|}$


Classe A
Classe B
x_1
x_2
$k = 3$
$k = 6$

Basic Practice

Basic Practice

- We have seen some issues that we need to consider in machine learning
  - feature engineering
  - overfitting
  - hyperparameter tuning
- These and other aspects must be considered before choosing and fitting a model

Feature Engineering

Feature Engineering

- Prior to selecting and training a model, we need to gather data
  - build a dataset
- In supervised learning: labeled examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$
  - each $\mathbf{x}_i$: *feature vector*
    - dimension $j = 1, \ldots, D$
    - contains a value describing the example
    - each feature is denoted as $x^{(j)}$
- Feature engineering
  - problem of transforming raw data into a dataset
  - usually a labor-intensive process that demands domain knowledge
  - everything measurable can be used as a feature
    - we must select/create informative features
    - aim: allow learning algorithm to build a good model
    - model with low bias $\rightarrow$ predicts the training data well

One-Hot Encoding

- Some learning algorithms only work with numerical feature vectors
- When some feature is categorical $\rightarrow$ transform it into several binary features
  - *e.g.*: like "colors" feature with three possible values

$$\begin{aligned} \text{"red"} &\rightarrow [1,0,0] \\ \text{"yellow"} &\rightarrow [0,1,0] \\ \text{"green"} &\rightarrow [0,0,1] \end{aligned}$$

  $\rightarrow$ increases dimensionality
- We should not transform in numerical values, *e.g.*1,2,3
  - this would imply an order between values in this category
  - order would have implications in the model
  - algorithm would try to find order where is does not exist
    - as opposed to, *e.g.*, `poor, fair, good, excellent`
    - could be assigned values $\{1,2,3,4\}$

Binning

- when we have a numerical feature that we want to convert into a categorical one
- process of converting a continuous feature into multiple binary features
  - called bins or buckets
  - typically based on value range
  - less frequent in practice
- *e.g.*, *age*: in binning we create "additional features", like in one-hot encoding
  - bin 1: 0 and 5 years-old $\rightarrow [1, 0, 0, \ldots]$
  - bin 2: 6 to 10 years-old $\rightarrow [0, 1, 0, \ldots]$
  - $\ldots$
- in some cases, a carefully designed binning can help algorithm to learn better
  - give a "hint" that when the value of a feature falls within a specific range, the exact value of the feature doesn't matter

Normalization

- process of converting an actual range of values which a numerical feature can take, into a standard range of values
  - typically: interval $[-1, 1]$ or $[0, 1]$
- *e.g.*, for interval $[0, 1]$:

$$\bar{x}^{(j)} = \frac{x^{(j)} - \min^{(j)}}{\max^{(j)} - \min^{(j)}}$$

  - $\min^{(j)}/\max^{(j)} \rightarrow$ minimum/maximum value of feature j in dataset
- motivations:
  - consider, *e.g.*, $x^{(1)} \in [0, 1000], x^{(2)} \in [0, 0.0001]$
  - in gradient descent, partial derivative w.r.t. larger feature will dominate update
  - also important to limit numeric rounding errors

Standardization (or z-score normalization)

- feature values are rescaled so that they have the properties of a standard normal distribution
  - mean $\mu = 0$, standard deviation $\sigma = 1$
  - computed over all examples in the dataset
- standardized values:

$$\hat{x}^{(j)} = \frac{x^{(j)} - \mu^{(j)}}{\sigma^{(j)}}$$

  - $\mu^{(j)}/\sigma^{(j)} \rightarrow$ mean/standard deviation of feature j in dataset
- standardization vs normalization: what the book says
  - unsupervised learning algorithms, in practice, more often benefit from standardization than from normalization
  - standardization also preferred if values of the feature are distributed close to a normal distribution
  - standardization is preferred for a feature if it can have extremely high/low values (outliers)
    - normalization "squeezes" most values into a very small range
  - all other cases: normalization is preferable

Dealing with Missing Features

- In some datasets, values of some features are missing
- Often when dataset involves human intervention
  - some values not filled/not measured
- Dealing with missing values for a feature:
  - remove examples with missing features from the dataset
    - possible when dataset is big enough to sacrifice some training examples
  - use learning algorithm that can deal with missing feature values
    - depends on the library/implementation of the algorithm);
  - use a *data imputation technique*

Data Imputation Techniques

1. replace missing value by average value of the feature in the dataset
   - often, median is better
2. replace missing value with a value outside the normal range
   - algorithm will learn what is best to do when the feature has a value significantly different from regular values
3. advanced technique: compute missing value as target in regression problem
   - let j be the feature with a missing value
   - using all remaining features, and all examples except those with missing $x_i^{(j)}$
   - build regression problem with target $x^{(j)}$
   - with it, predict target on $[x_i^{(1)}, \ldots, x_i^{(j-1)}, x_i^{(j+1)}, \ldots, x_i^{(D)}]$
4. for large dataset with just a few features with missing values:
   - 4.1 increase the dimensionality of feature vectors
   - 4.2 add binary indicator feature for each feature with missing values
   - 4.3 set that feature equal to 1 on examples where original is present, 0 otherwise
   - 4.4 missing value then can be replaced by any number (*e.g.*, zero)

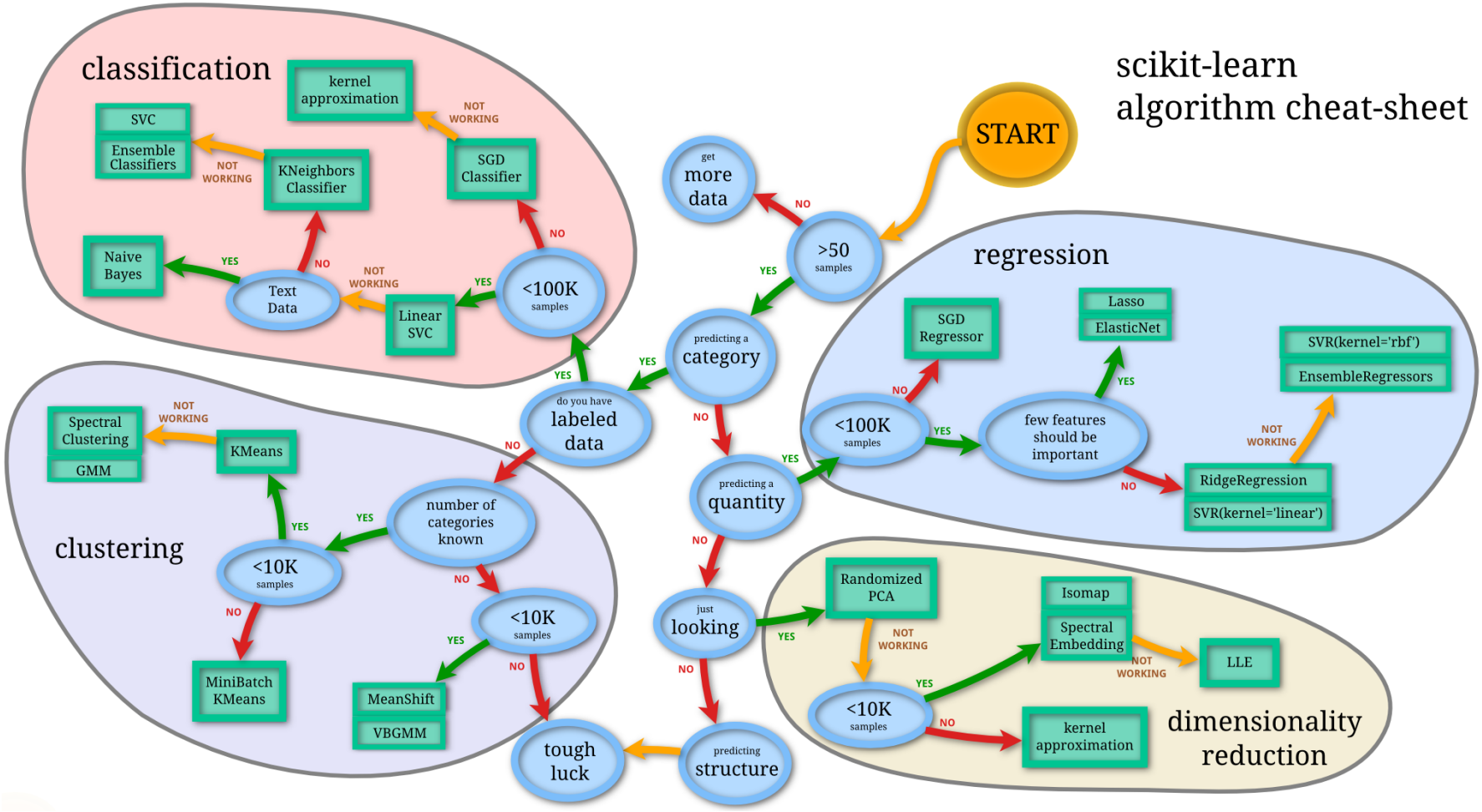
Data Imputation Techniques

- At prediction time:
  - if example to predict is not complete $\rightarrow$ use the same data imputation technique to fill the missing features
- Usually we cannot tell a priori which data imputation technique will work the best
  - try several techniques, build several models and select the one that works the best

Learning Algorithm Selection

Learning Algorithm Selection

- How do we select a machine learning algorithm?
  - if we have much time $\rightarrow$ try all available
  - probably not practical
- We may select a few models and choose one by testing it on the validation set
  - next topic
- Consider the algorithm selection diagram of scikit-learn
- Next, some points to take into account



https://scikit-learn.org/stable/tutorial/machine_learning_map

Explainability

Does the model have to be explainable to a non-technical audience?

- very accurate algorithms are often "black boxes"
- very few errors, but difficult to understand/explain
- simple models are less accurate, but easy to explain
  $\rightarrow$ *e.g.*, linear regression, kNN, decision trees

In-memory vs out-of-memory

Can the dataset be fully loaded into computer's RAM?

- If yes $\rightarrow$ wide variety of algorithms
- Otherwise: incremental learning algorithms are preferable
  - models that can be improved by adding more data gradually

Number of features and examples

How large is the the dataset?

- Size:
  - how many examples?
  - how many features?
- Some algorithms can handle a huge number of examples and features
  - neural networks, gradient boosting
- Others are limited
  - kNN (all data must be kept in memory)
  - SVM (underlying optimization method)

Categorical vs. numerical features

- Is our data composed of categorical only, or numerical only features, or a mix of both?
  - *e.g.*, algorithms in scikit-learn can handle only numerical features
    - $\rightarrow$ convert categorical features into numerical ones

Non-linearity of the data

- Are linear models enough?
  - SVM with the linear kernel, logistic or linear regression can be good choices
- Otherwise, consider kernel SVMs, deep neural networks or ensemble algorithms

Training speed

- How much time is allowed to build a model?
  - neural networks $\rightarrow$ slow to train
    - deep learning $\rightarrow$ usually require GPUs
  - simple algorithms are much faster
    - like logistic and linear regression or decision trees
- Libraries may differ on implementation efficiency
- Some algorithms are suitable for multiple CPU cores
  - building time can be significantly reduced on a machine with dozens of cores
  - *e.g.*random forests

Prediction speed

- How fast does the model have to be when generating predictions?
  - SVMs, linear and logistic regression $\rightarrow$ extremely fast at prediction
  - kNN, ensemble algorithms, very deep neural networks are slower

Dataset partition

Dataset partition

In any supervised learning project, we need to work with three distinct sets:

1. training set
2. validation set
3. test set

Steps for preparing them:

- shuffle the examples
- split the dataset into these three subsets
- training set: usually the biggest, used to build the model
- validation and test sets
  - roughly the same sizes, much smaller than training set
  - learning algorithm cannot use these examples to build the model
  - also called *holdout sets*

Dataset partition

Reason to have three sets:

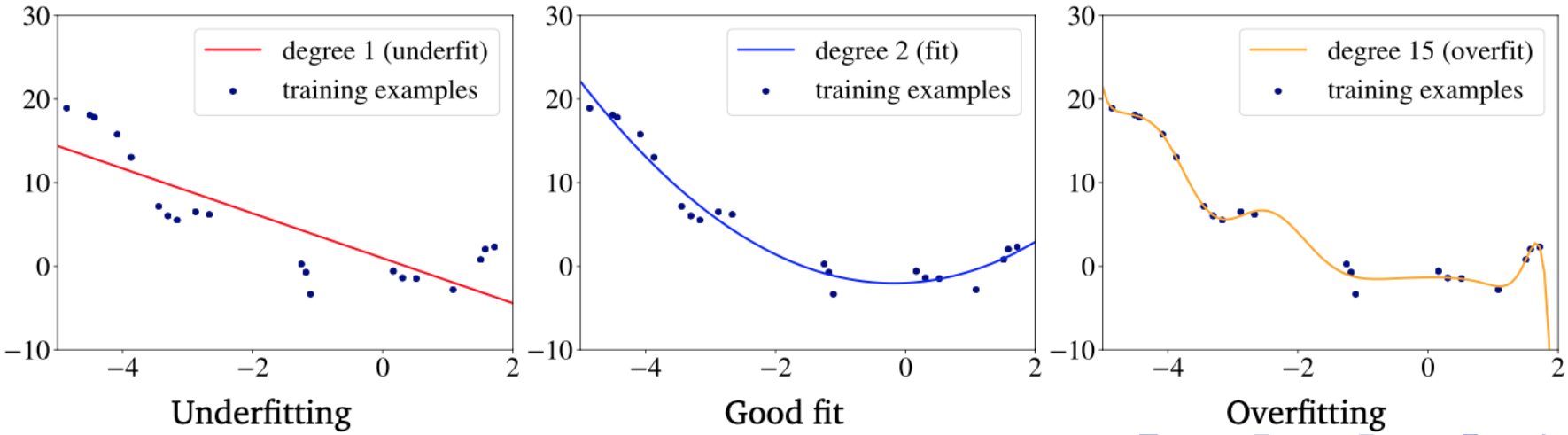
- when we build a model, model is adjusted to training set
  - *training error* is optimized
- but ultimate goal is to use it in new, unseen data
  - *generalization error* $\rightarrow$ we cannot optimize it
  - a proxy: error obtained with data not used for training
    - validation set $\rightarrow$ used to select among models/hyperparameters
    - test set $\rightarrow$ used only for assessing quality of the final model

Dataset: partition sizes

- There's no optimal proportion to split the dataset
- Scarce data: rule of thumb:
  - 70% for training
  - 15%/15% for validation/testing
  - use *crossvalidation*
- Abundant data:
  - 95% for training
  - 2.5%/2.5% for validation/testing

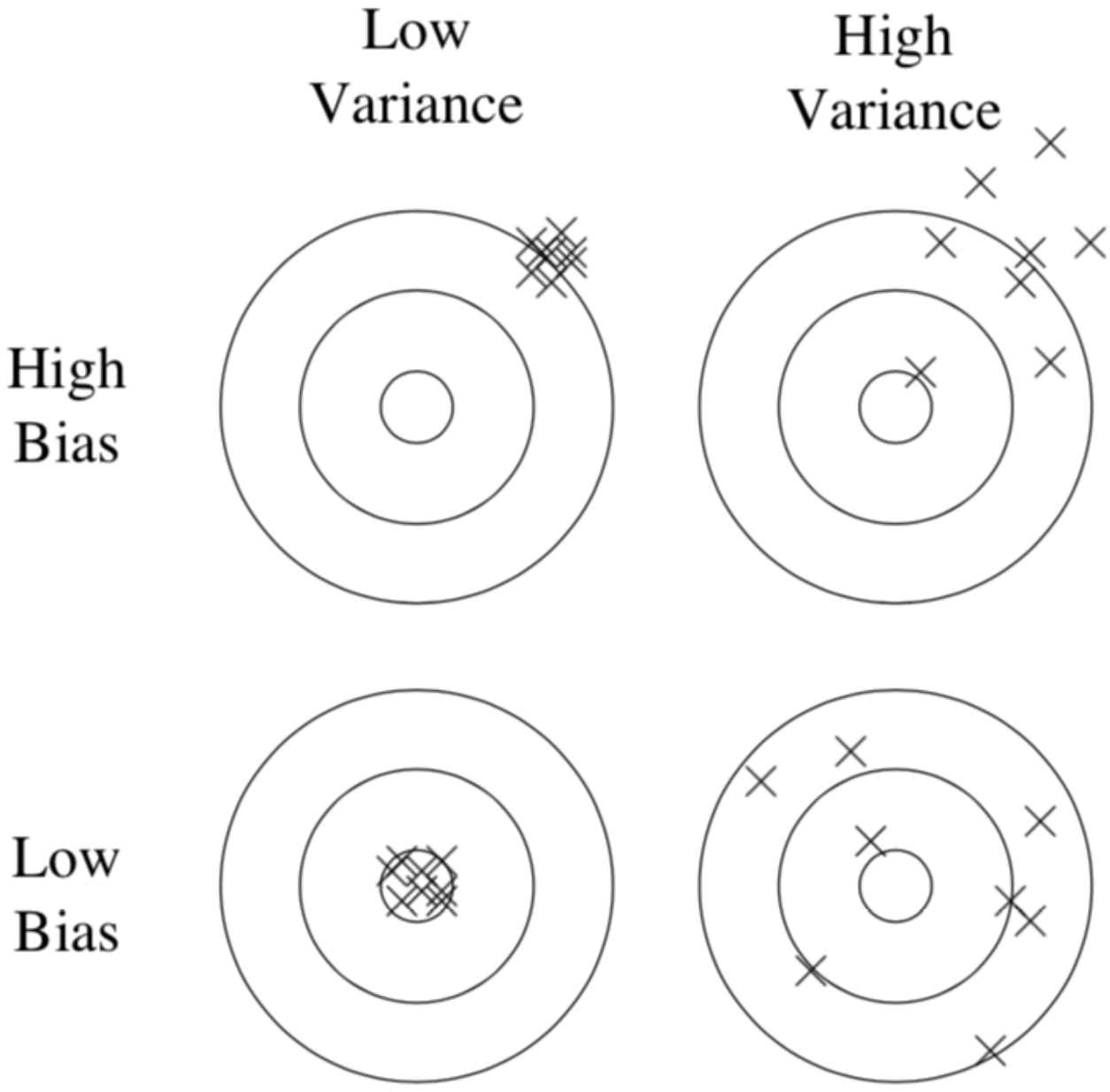
Underfitting and Overfitting

- If model predicts well the labels of the training data: low bias
- If model makes many mistakes on the training data:
  - high bias
  - model *underfits* $\rightarrow$ not able to predict well the labels of the data used for training
    1. model is too simple for the data
    2. features are not informative enough
- Overfitting: model predicts very well training data, but poorly on different data
  1. model is too complex for the data
     - *e.g.*, very deep decision tree/neural network
  2. too many features, too few training examples



Overfitting and *high variance*

- In statistics, overfitting is called *high variance*
  - variance $\rightarrow$ errors of the model due to sensitivity to small fluctuations in the training set
  - if training data was sampled differently $\rightarrow$ learning would result in very different model
- Models that overfit perform poorly on the test data
  - *large generalization error*


Low
Variance
High
Variance
High
Bias
Low
Bias

Overfitting and size of the training set

- Even a simple model can overfit the data:
  - if data is high-dimensional (many features)
  - if number of training examples is low
- *e.g.*, linear models in high dimensions:
  - assign non-zero values to most parameters $w^{(j)}$
  - determine complex relationships between all available features to predict labels of training examples perfectly
    - usually, very sensitive to small perturbations in data
    - if $N \approx D \rightarrow$ training error ≈ 0
- Solutions:
  1. Try a simpler/less powerful model
     - one with less parameters
  2. Reduce the dimensionality
     - select fewer features
     - use a dimensionality reduction technique
  3. Add more training data
  4. Regularize the model

Regularization

Regularization

- force the learning algorithm to build a less complex model
  - slightly higher bias
  - significantly lower variance
- *bias-variance trade-off*
- to create a regularized model:
  - modify the objective function by adding a penalizing term whose value is higher when the model is more complex
- illustration: *L1 and L2 regularization*

Regularization

Let us consider *linear regression*

- objective:

$$\min_{\mathbf{w},b} \frac{1}{N} \sum_{i=1}^{N} (f_{\mathbf{w},b}(\mathbf{x}_i) - y_i)^2$$

- L1-regularized objective:

$$\min_{\mathbf{w},b} \left[C|\mathbf{w}| + \frac{1}{N} \sum_{i=1}^{N} (f_{\mathbf{w},b}(\mathbf{x}_i) - y_i)^2 \right]$$

$$\text{where} \quad |\mathbf{w}| \stackrel{\text{def}}{=\!=} \sum_{j=1}^{D} |w^{(j)}|$$

$C \rightarrow$ hyperparameter controlling *importance of regularization*

- L2-regularized objective: use regularization term $C||\mathbf{w}||^2$

$$||\mathbf{w}||^2 \stackrel{\text{def}}{=\!=} \sum_{j=1}^{D} (w^{(j)})^2$$

Effects of regularization

- L1 regularization (also known as *lasso*) produces a sparse model
  - most of its parameters are zero, for large enough C
    - (in case of linear models, most of w(j))
  - so L1 performs feature selection
    - choose features that are essential for prediction
  - can be useful in case you want to increase model explainability
- L2 regularization (also known as *ridge regularization*)
  - usually models have better performance on holdout data
  - differentiable $\rightarrow$ allow using gradient descent
- elastic net regularization: combine L1 and L2 regularization
  - *e.g.*, regularization term $(C_1|\mathbf{w}| + C_2||\mathbf{w}||^2)$

Model Performance Assessment

- Question: how good is a model created by a learning algorithm?
  - use the test set to assess it
  - examples not seen before
    - if our model performs well on them, it "generalizes well"
- Assessing model performance

Assessing model performance: regression

Compare model to *mean model*

- model which always predicts the average of labels in training data
- regression model should be better. . .
- check performances on training and test data; *e.g.*, mean squared error
  - if the MSE on test data is substantially higher than on training data $\rightarrow$ overfitting
    - consider regularization or a better hyperparameter tuning

Assessing model performance: classification

Most widely used metrics:

- Confusion Matrix
- Precision/Recall
- Accuracy
- Cost-Sensitive Accuracy
- Area under the ROC Curve (AUC)

Illustrations with *binary classification*

Confusion Matrix

	Predicted	
Actual	True	False
True	true positives (TP)	false negatives (FN)
False	false positives (FP)	true negatives (TN)

Example: spam detection

	Predicted	
Actual	spam	not spam
spam	TP=23	FN=1
not spam	FP=12	TN=556

- correctly classified: TP + TN
- mistakes: FP + FN

Precision/Recall

- precision: *e.g.*, proportion of correctly classified examples among those that were classified as positive

$$\text{precision} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- recall: *e.g.*, proportion of correctly classified examples among those that actually are positive

$$\text{recall} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Precision/Recall

- depending on the problem, it may be important to have good precision or good recall
  - consider, eg, detecting cancer
- often, we have to choose between high precision or high recall
- controlling this trade-off in learning algorithms:
  - assign higher weighting to examples of a specific class
  - tune hyperparameters to maximize precision or recall on the validation set
  - varying the decision threshold for algorithms that return probabilities
    - *e.g.*, logistic regression: we can decide that the *prediction will be positive* only if the probability returned by the model is higher than 0.9.
- for using in multiclass classification:
  - first select class for which we to assess these metrics
  - then consider:
    - examples of this class as positives
    - examples of the remaining classes as negatives

Accuracy

- accuracy: proportion of correctly classified examples

$$\text{accuracy} \stackrel{\text{def}}{=} \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- useful metric when errors in predicting all classes are equally important
- default in most learning algorithms for classification

Cost-Sensitive Accuracy

- cost-sensitive accuracy: useful metric when different classes have different importance
  - first assign a cost (a positive number) to both types of mistakes (FP and FN)
  - compute TP, TN, FP, FN as usual
  - multiply the counts for FP and FN by the corresponding cost
  - use these values for calculating the accuracy

Area under the ROC Curve (AUC)

- ROC curve: "receiver operating characteristic"
  - term comes from radar engineering
  - use a combination of
    - *true positive rate* $\rightarrow$ proportion of positive examples predicted correctly (*recall*)

$$\text{TPR} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP} + \text{FN}}$$

    - *false positive rate* $\rightarrow$ proportion of negative examples predicted incorrectly

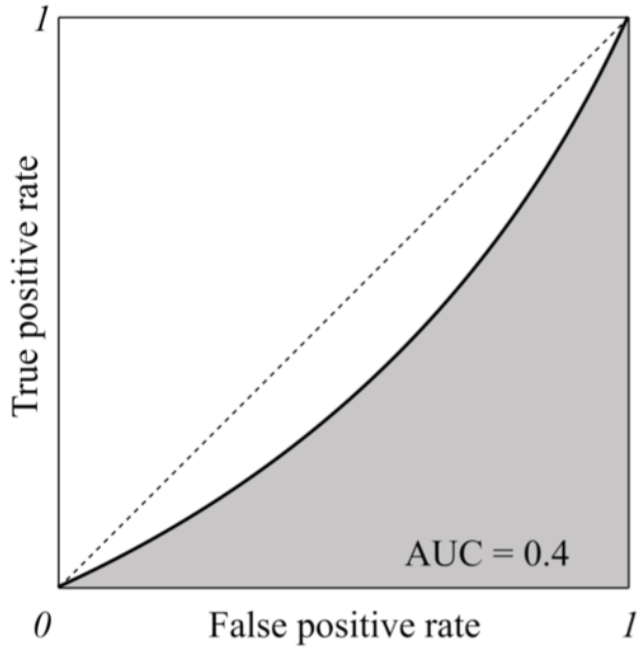
$$\text{FPR} \stackrel{\text{def}}{=} \frac{\text{FP}}{\text{FP} + \text{TN}}$$

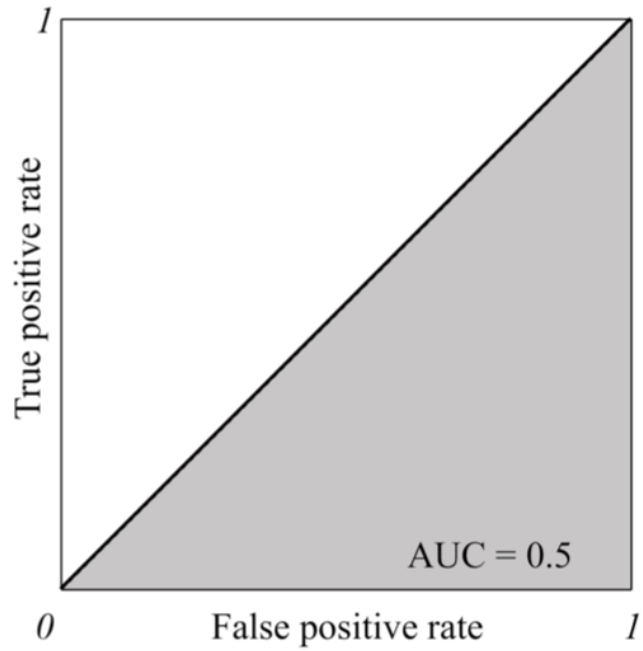
  - can only be used with classifiers that return prediction's probability/confidence score
    - logistic regression, neural networks, decision trees

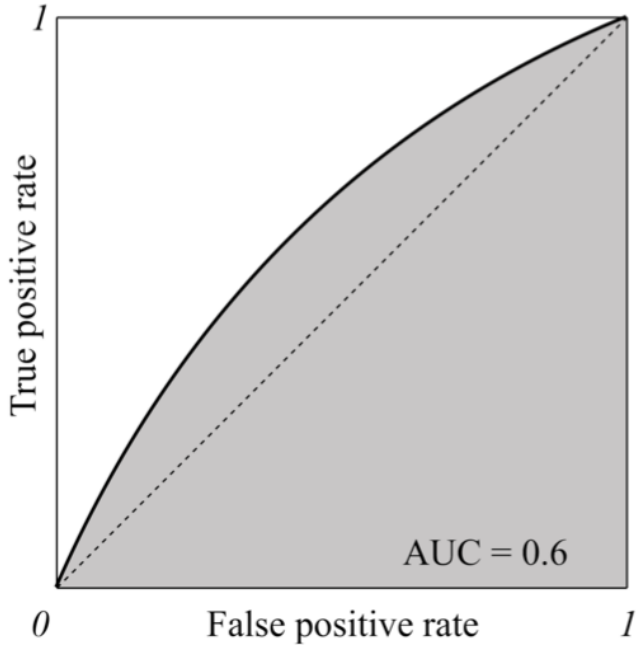
AUC

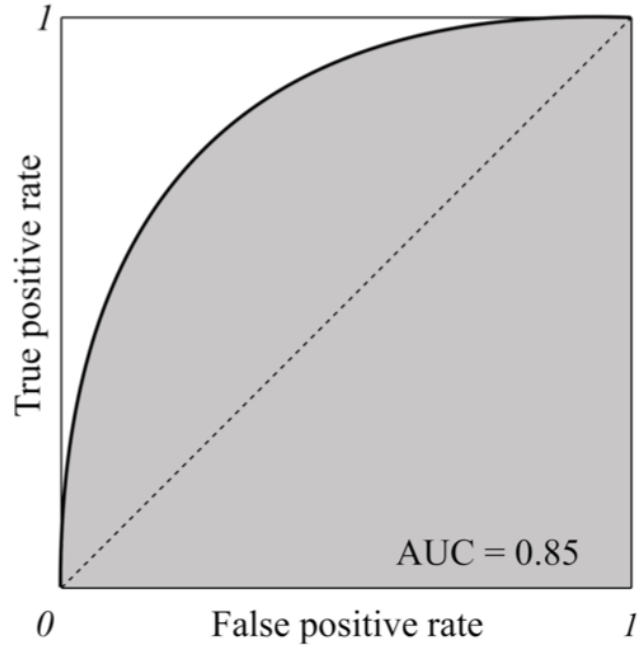
To draw a ROC curve:

- discretize the range of the confidence score
  - *e.g.*, $[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
- use each of these values as the prediction threshold
- predict the labels of examples in dataset using the model and this threshold
  - threshold $= 0 \rightarrow$ many false positives, no false negatives
  - threshold $= 1 \rightarrow$ many false negatives, no false positives


1
True positive rate
AUC = 0.4
0
False positive rate
1


1
True positive rate
AUC = 0.5
0
False positive rate
1


1
True positive rate
AUC = 0.6
0
False positive rate
1


1
True positive rate
AUC = 0.85
0
False positive rate
1

AUC

- the higher the AUC, the better the classifier is
- random classifier: AUC $= 0.5$
- perfect classifier: AUC $= 1$
- if model behaves well: we obtain a good classifier by selecting the value of the threshold that gives
  - TPR close to 1
  - keeping FPR near 0

Hyperparameter Tuning

- hyperparameters aren't optimized by the learning algorithm itself
- we have to "tune" them by experimentally finding the best combination of values
- grid search
  - decide a discrete set of values for each hyperparameter
  - try all possible combinations
  - asses performance on validation data
  - choose model with best performance
- then, we can assess that model using the test set

Hyperparameter Tuning: example

- suppose we want to train an SVM
- two hyperparameters to tune:
  - the penalty parameter C (a positive real number)
    - trick: use logarithmic scale
    - *e.g.*, $C \in [0.001, 0.01, 0.1, 1, 10, 100, 1000]$
  - kernel (*e.g.*, "linear" or "rbf")
  - then test combinations
    (0.001, “linear”)
    (0.01, “linear”)
    ...
    (0.001, “rbf”)
    ...
    (1000, “rbf”)
  - choose the one that gave the best performance, on the metric we chose
- problems: trying all combinations quickly becomes too time-consuming

Hyperparameter Tuning: alternatives

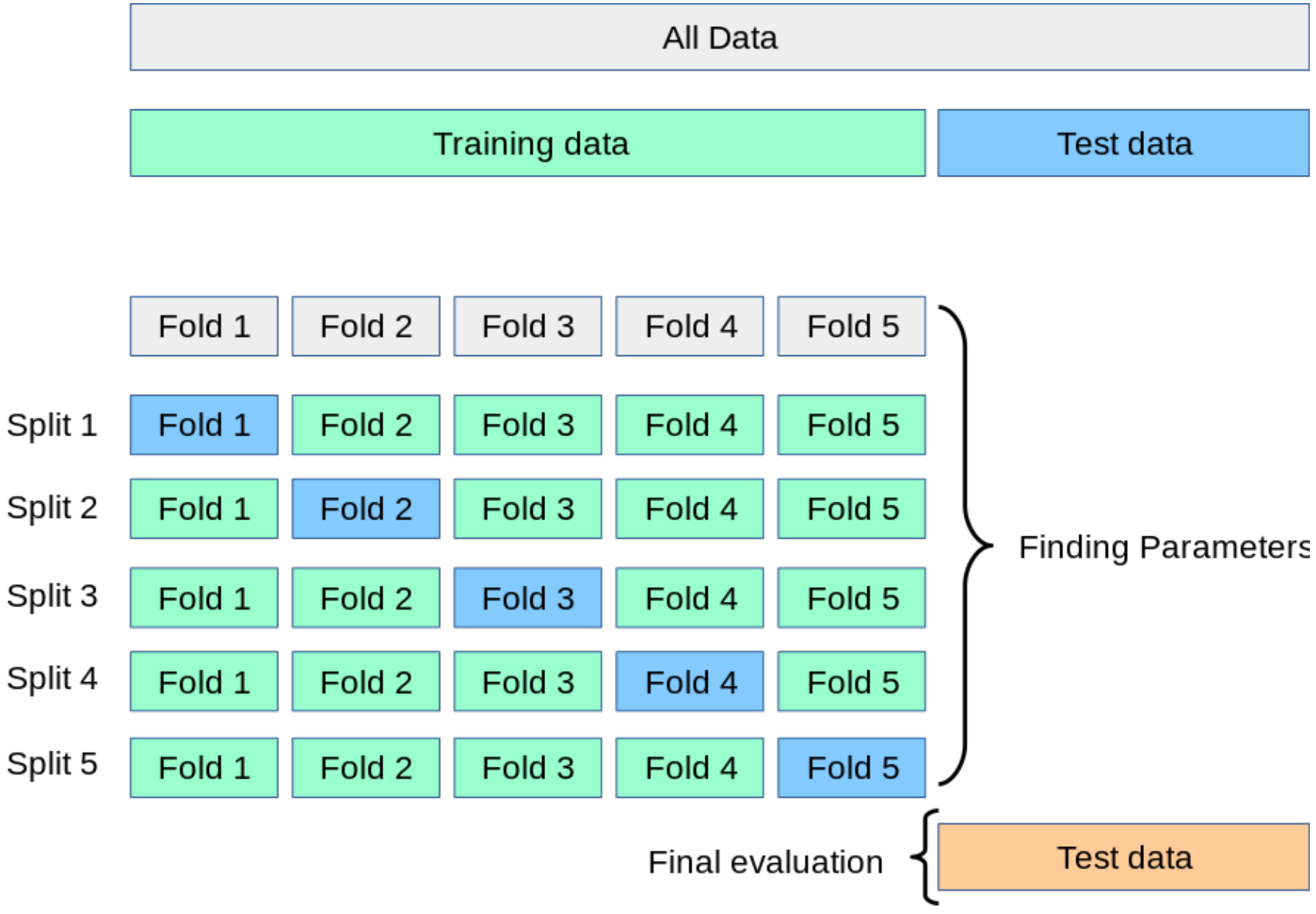
- Grid search
  - very time consuming
  - tests many combinations that won't work
- Random search
  - provide a statistical distribution for each hyperparameter
  - values are randomly sampled and and tested, until reaching the total number of tentatives we want to try
- Bayesian hyperparameter optimization
  - use past evaluation results to choose the next values to evaluate
  - idea: limit the number of tentatives by concentrating on values that have done well in the past

Cross-Validation

- used when the datasets are small
- not enough data to have validation set, for tuning hyperparameters
- idea:
  - split data into training and test set
  - use cross-validation on the training set to simulate a validation set
- we can use grid search with cross-validation to find the best hyperparameters
- then, use the entire training set to build the model with these best values of hyperparameters
- at the end, assess this model using the test set

Cross-Validation

1. fix the values of the hyperparameters we want to evaluate
2. split training set into several subsets of the same size
   - each subset is called a fold
   - *e.g.*, *five-fold cross-validation* is often used
3. asses each possible model (*i.e.*, each set of parameters) into all the folders, and average results
4. finally, you assess the model using the test set.


All Data
Training data
Test data
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Split 1
Split 2
Split 3
Split 4
Split 5
Finding Parameters
Final evaluation
Test data

Cross-Validation: suppose 5 folds:

- randomly split training data folds $\{F_1, F_2, \ldots, F_5\}$
- each F_k contains 20% of the training data
- then, train five models:
  - to train model f_1:
    - use all examples from folds F_2, F_3, F_4, F_5 as training set
    - use examples from F_1 as validation set
  - to train model f_2
    - use all examples from folds F_1, F_3, F_4, F_5 as training set
    - use examples from F_2 as validation set
  - ...
  - average the five values of the metric
    - use this value as for evaluating the model

Challenges

Challenge 1: decision trees

- Explore the way decision trees work using scikit-learn's page: `https://scikit-learn.org/stable/auto_examples/tree/plot_unveil_tree_structure.html`
  - Try different values for some of the hyperparameters; check how you would classify the last example in tha data if you set each of these hyperparameters to 3:
  - `max_leaf_nodes`
  - `max_depth`
  - `min_samples_split`

Challenge 2: kernel-based SVM

- Follow the SVM tutorial on scikit-learn's page:
  `https://scikit-learn.org/stable/modules/svm.html`
  - focus on examples of non-linear SVMs and their parameters
  - check which kernels are avalailble, and try a few of them
  - does the decision boundary change with the kernel?

Challenge 3: nearest neighbors

- Follow the k-nearest neighbors tutorial on scikit-learn's page: `https://scikit-learn.org/stable/modules/neighbors.html`
  - understand how it works both on classification and on regression
  - check the influence of parameter k; with increasing k, does the boundary become smoother or more irregular? what would you expect in terms of under/overfitting?