Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

Lecture 3

Data Science and Computation PhD + Master in Bioinformatics

University of Bologna

Communications

Written exam of AML-Basic

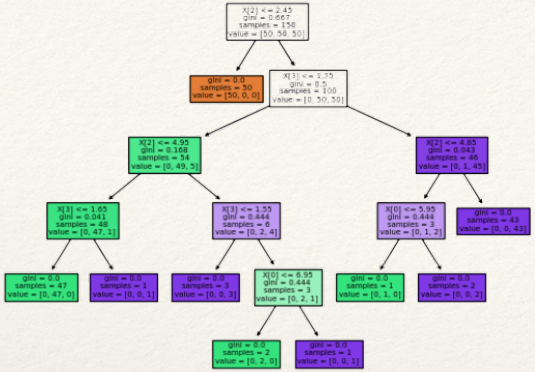
(**Confirmed**) date for the AML-Basic written exam before Adv ends

- **22 May 2023, start at 14:00ish**
- → it is required to sign-up on AlmaEsami (deadline: 19/5)

	Attività formative	Descrizione	Data e ora	Place	Tipo	Stato	Iscr.		
	88407 - Applied Machine Learning (Cds. 8020) 93279 - Applied Machine Learning (Cds. 8020) 93282 - Applied Machine Learning - Advanced (Cds. 8020) ...	Written Exam of Applied Machine Learning (BASIC) - "special date"	22/05/2023 14:00	Dipartimento DIFA - Via Irnerio 46 - Lab. Informatico	Scritto	Iscrizioni in corso	0	Iscritti Modifica Duplica	Cancella

More details at the lecture

- e.g. how will the exam be?
- e.g. can/should I bring my own laptop?
- e.g. can I do my exam remotely?



Decision Trees

Decision Trees (**DT**)

Decision Trees are quite versatile ML algos

- they can perform both *classification* and *regression* tasks (even multi-output tasks)
- they are powerful enough to be capable of fitting complex datasets
- DTs are also the fundamental components of **Random Forests**, which are among the most powerful ML algos available today

What we cover about DTs

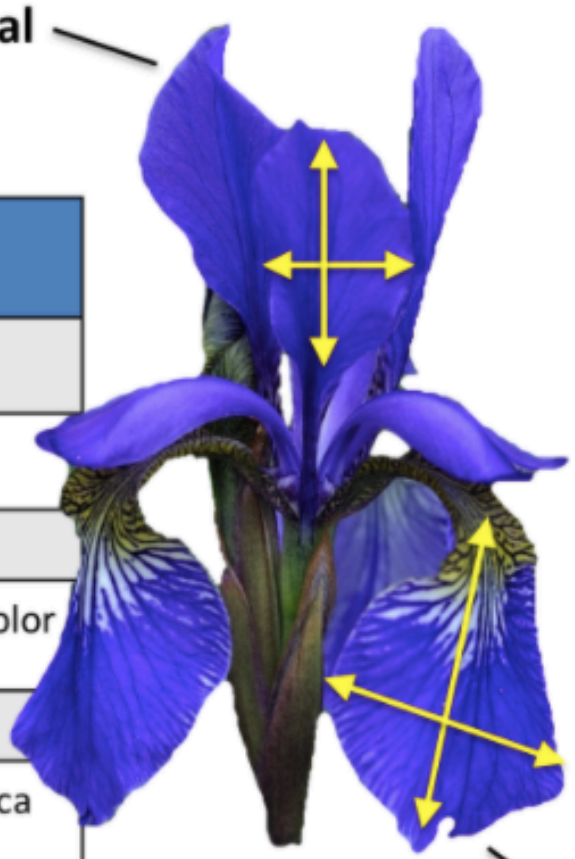
Briefly:

- how to train, visualise, and make predictions with Decision Trees
- CART training algorithm used by **sklearn**
- how to apply this to both classification and regression problems
- pros and cons of Decision Trees

Example: the **IRIS** dataset

Samples
(instances, observations)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica



Features
(attributes, measurements, dimensions)

Class labels
(targets)

The IRIS dataset: use of a DT for **classification**

Sklearn on the iris flowers dataset

→ `DecisionTree`**`Classifier`**

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

The IRIS dataset: use of a DT for **classification**

Sklearn on the iris flowers dataset

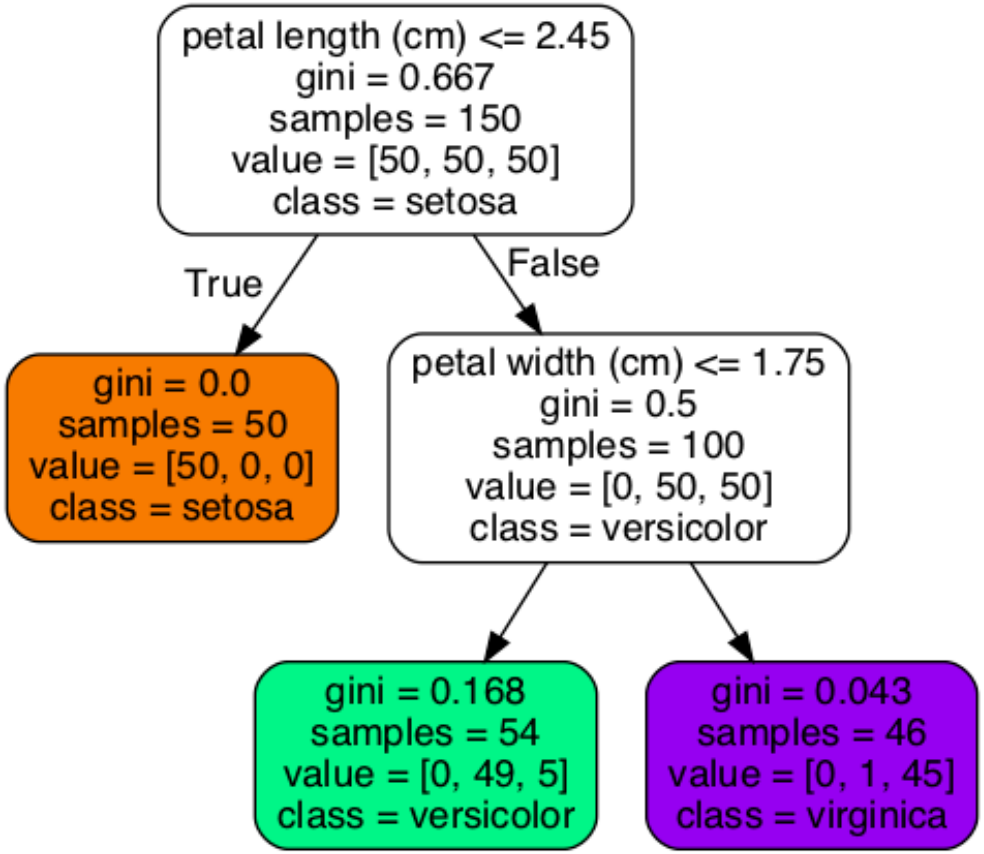
→ `DecisionTree`**`Classifier`**

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

creates this:



The IRIS dataset: use of a DT for **classification**

Sklearn on the iris flowers dataset

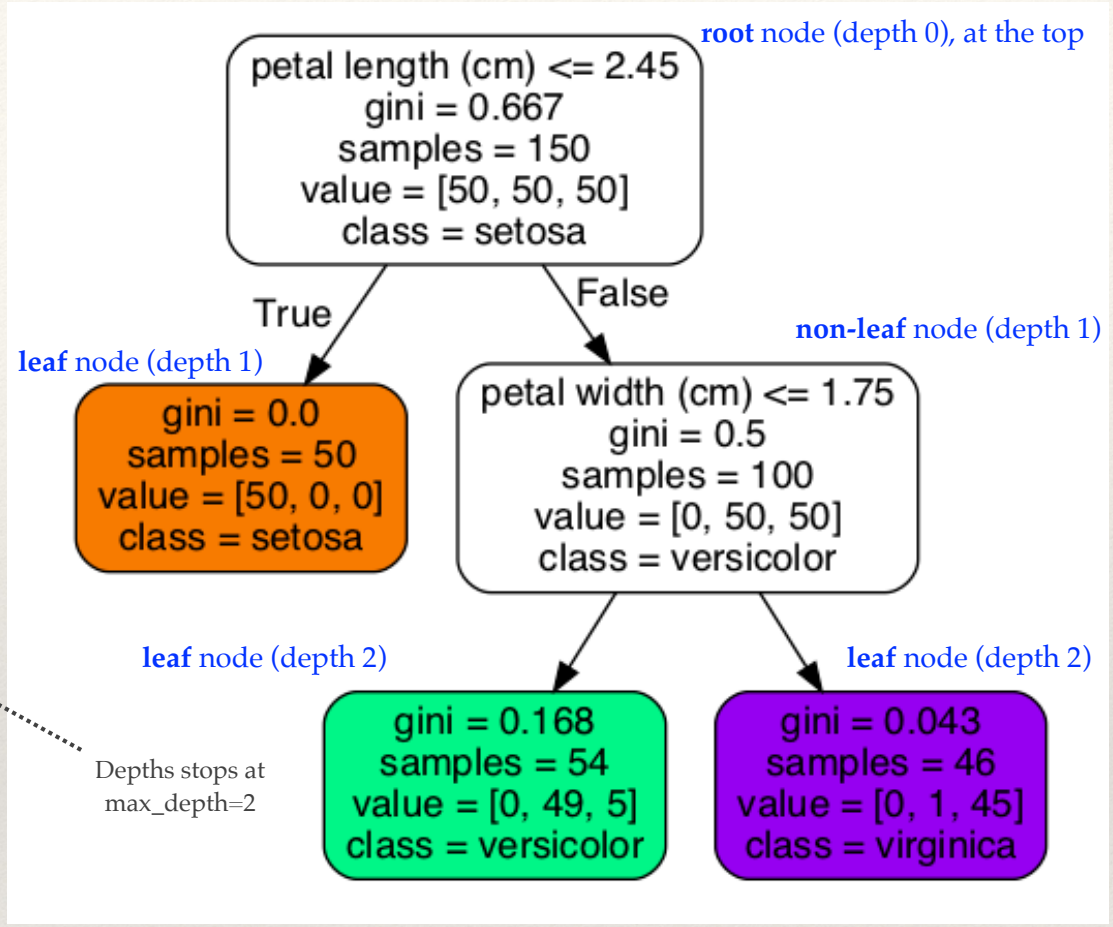
→ `DecisionTree`**`Classifier`**

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

creates this:



Look at the boxes:

- their position (depth)
- their color (leaf or not)

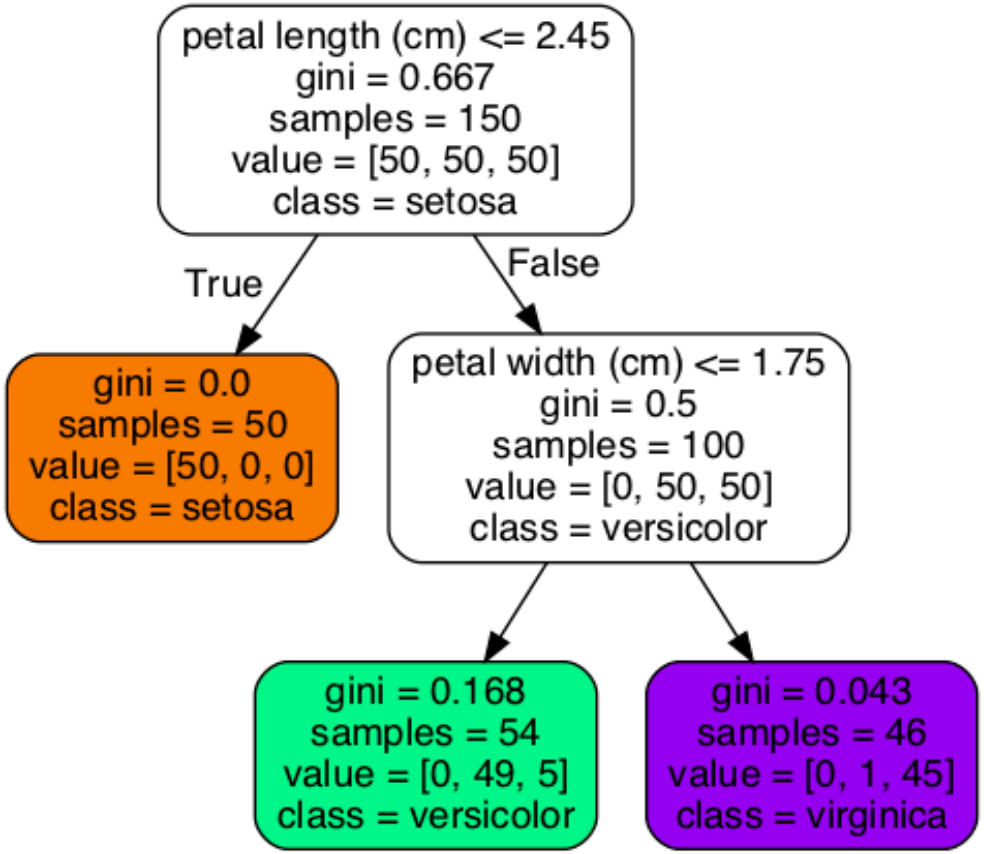
The IRIS dataset: use of a DT for **classification**

Sklearn on the iris flowers dataset
→ `DecisionTreeClassifier`

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```



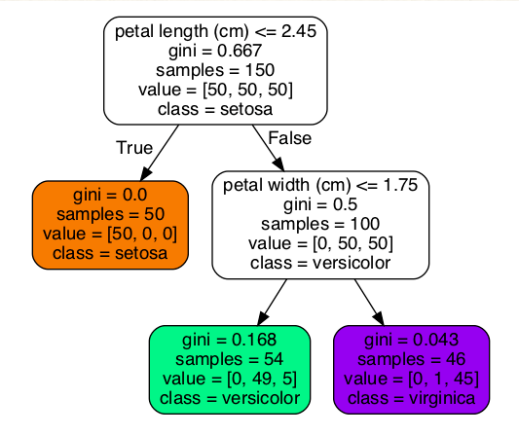
Look at the content of the boxes

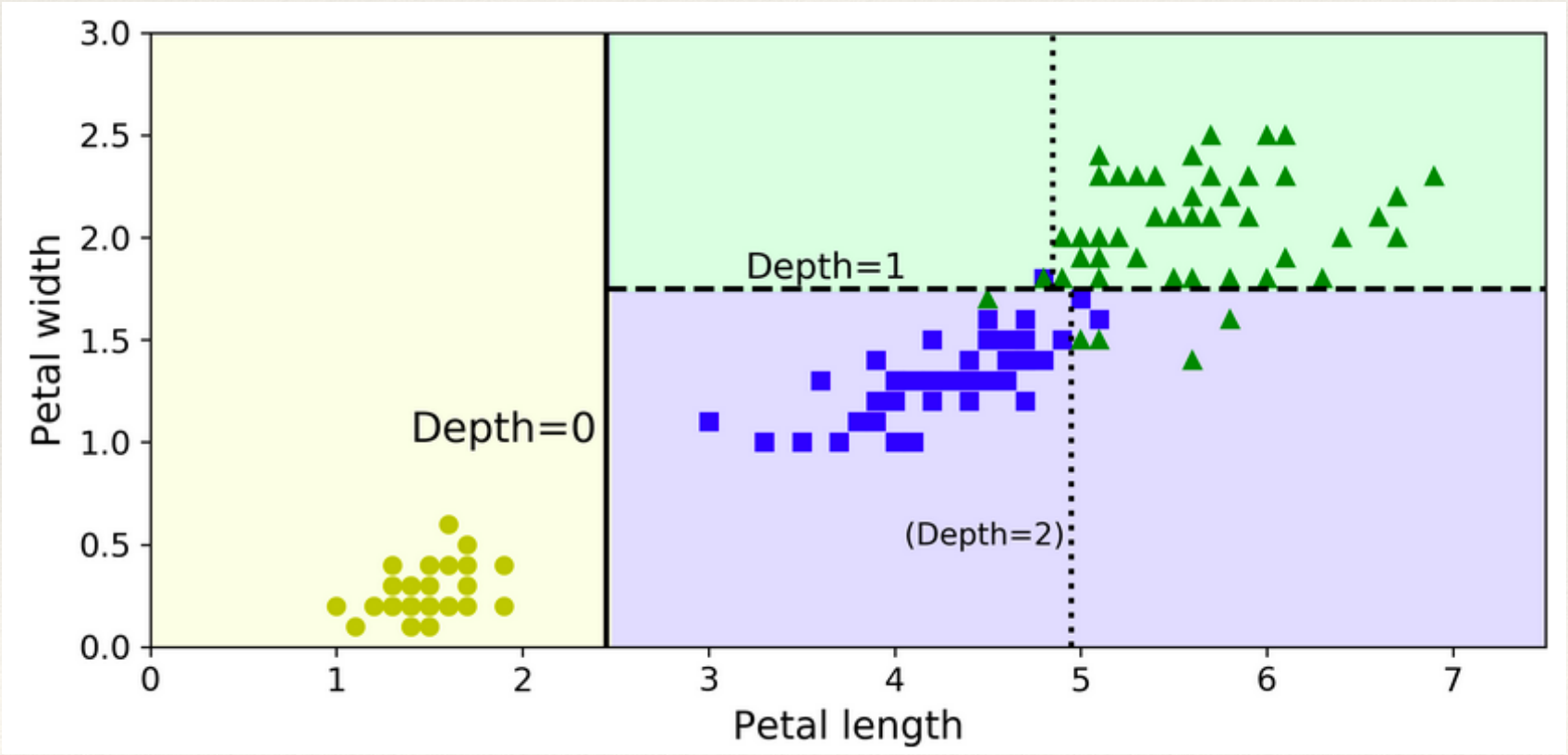
node's **samples** attribute = counts how many training instances it applies to
node's **value** attribute = how many training instances of each class this node applies to
node's **gini** attribute = measures its impurity (pure node - gini=0 - if all training instances it applies to belong to the same class)

Among the qualities of DTs is that they require **very little data preparation**

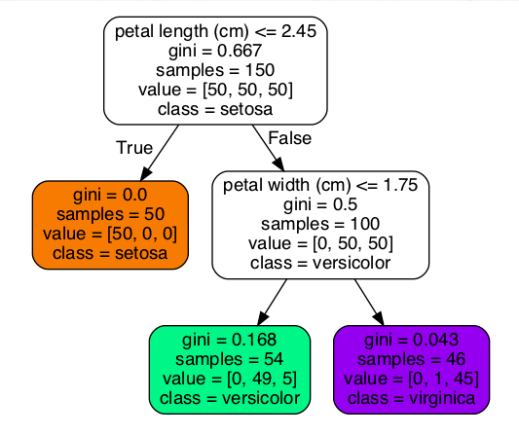
- in particular, they don't require feature scaling or similar, at all

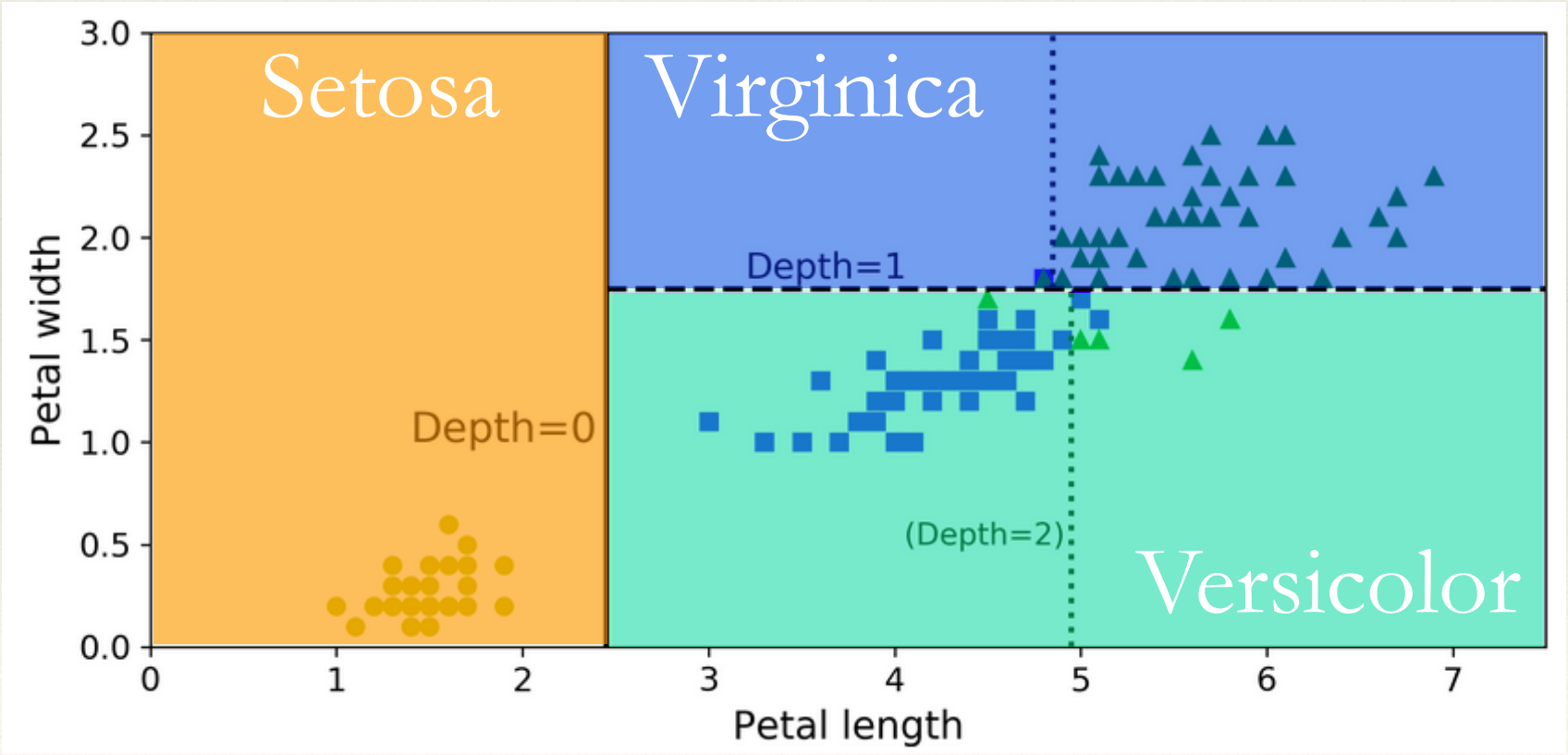
The IRIS dataset: use of a DT for **classification**



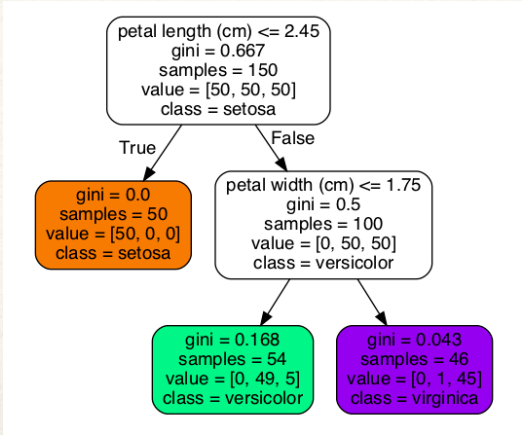


The IRIS dataset: use of a DT for **classification**

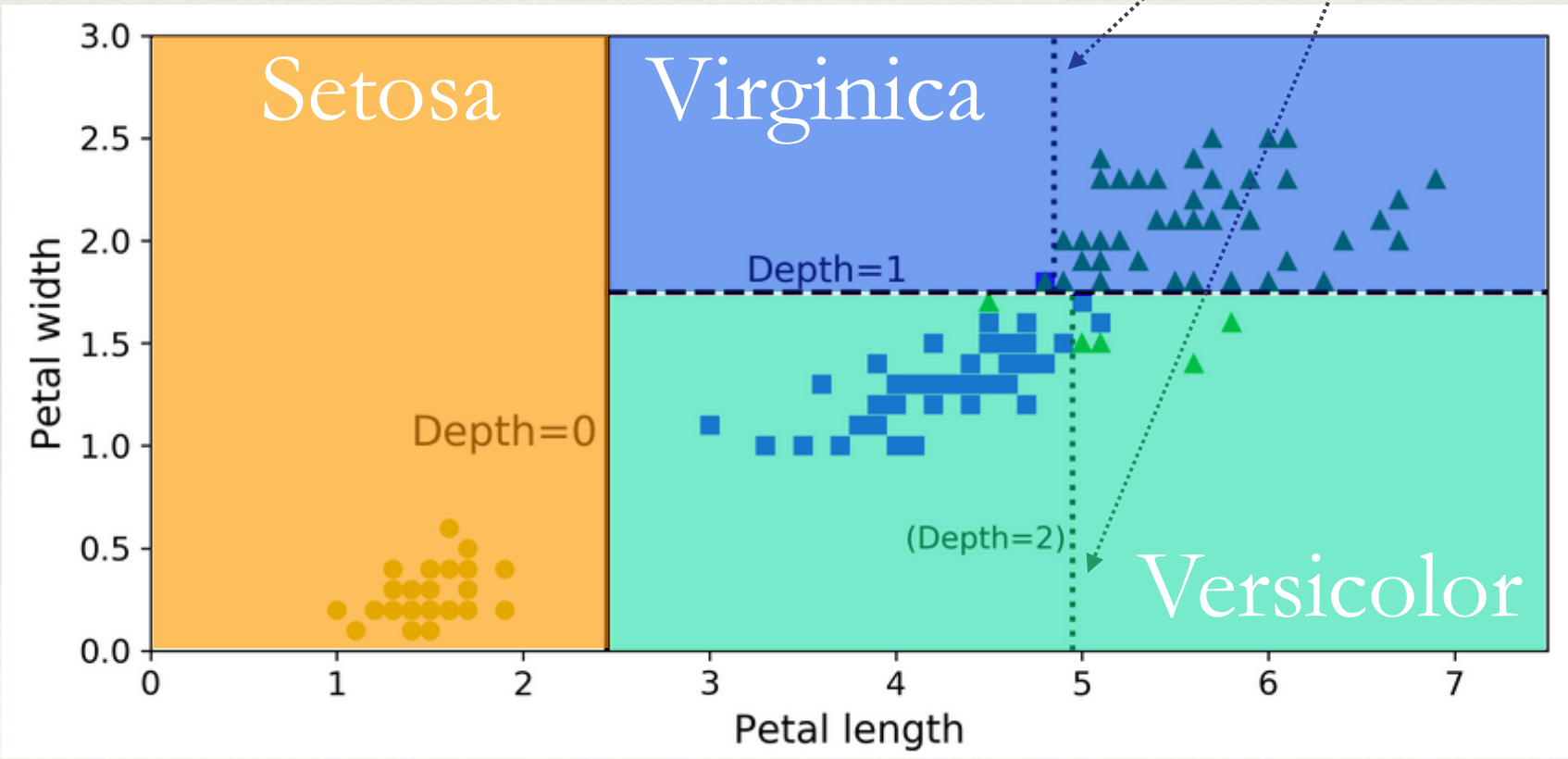




The IRIS dataset: use of a DT for **classification**



max_depth=3 would have added "depth 3" decision boundaries



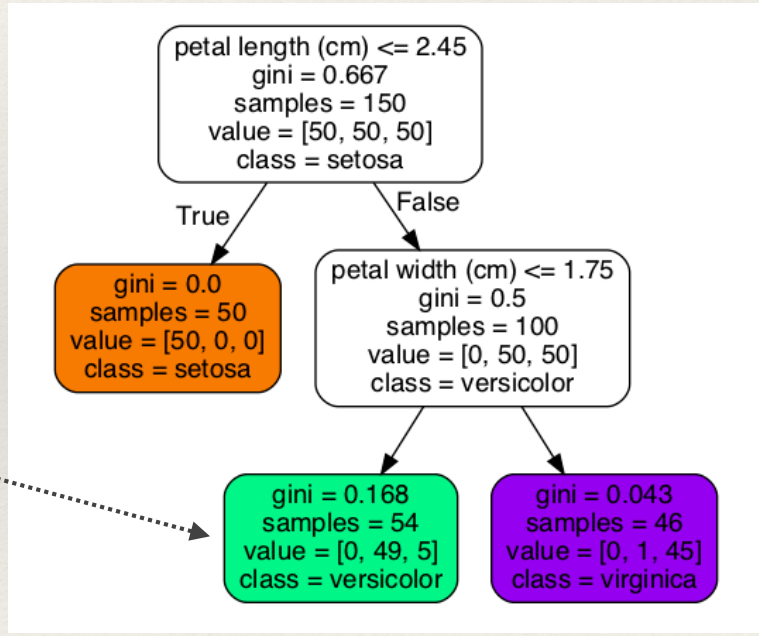
DT: estimating class probabilities

A DT can estimate the probability that an example (or instance) belongs to a particular class **k**

- first, it traverses the Tree to find the leaf node for this instance
- second, it returns the ratio of training instances of class k in this node

Example:

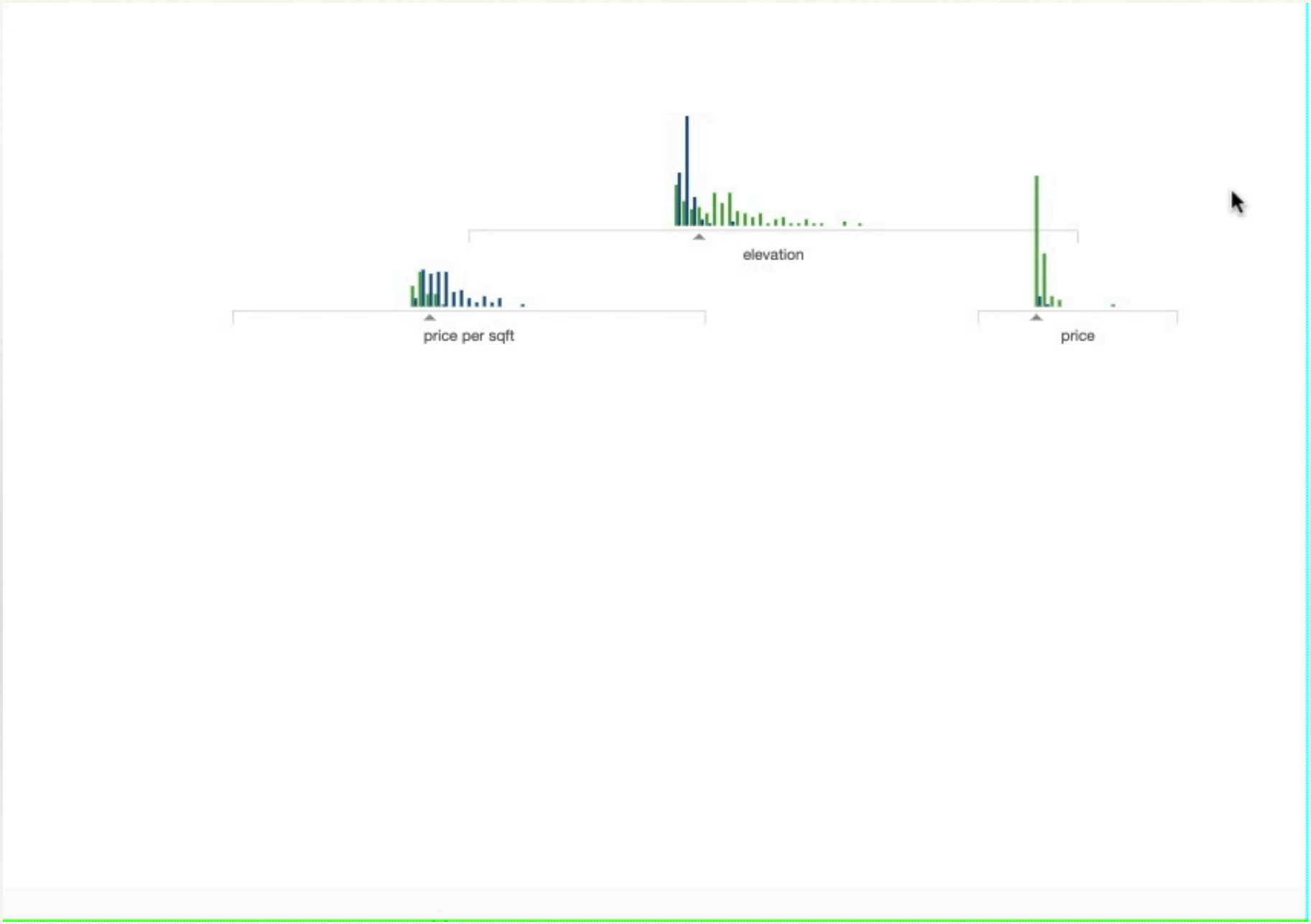
- a iris flower with 5cm long and 1.5cm wide petals corresponds to the green, bottom left, depth 2 node
- DT outputs the following probabilities:
  - 0/54 → 0% for **Iris-Setosa**
  - 49/54 → 90.7% for **Iris-Versicolor**
  - 5/54 → 9.3% for **Iris-Virginica**
- prediction outputs **Iris-Versicolor**




```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[0.        , 0.90740741, 0.09259259]])
```

White box vs Black box

*How is your **intuition** of the characteristics of this algo?*


elevation
price per sqft
price

White box vs Black box

*How is your **intuition** of the characteristics of this algo?*

Decision Trees fairly intuitive and decisions easy to interpret

- → **"white box" models**
- you may easily apply "manually" (non-ML) the same (and known) classification rules (a.k.a selection cuts) that the DT applied, and get the same results

Random Forests or Neural Networks give great(er?) predictions, you still can check calculations but often harder to explain why a prediction was made

- e.g. still interpretable, but less explainable..
- → **"black box" models**
  - e.g. a cat recognised in a picture from.. the ears? the tail? at which %?

Classification And Regression Tree (**CART**)

Sklearn uses the **CART** algo to train DTs

- it produces only binary trees (non-leaf nodes always have 2 children, i.e. questions only have yes/no answers)
- Other algos (such as **ID3**) can produce DTs with nodes that have >2 children

The idea behind the CART algo is quite simple

- it split the training set in 2 subsets using a single feature **k** and a threshold $\mathbf{t_k}$
- it chooses **k** and $\mathbf{t_k}$ by searching for the pair that produces the purest subsets (weighted by their size)
  - The cost function that the CART algo tries to minimize for classification is:

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where $\begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$

- then, it splits the 2 sub-sets in 2 again using the same logic, and so on, recursively, until indicated by the max_depth hyperparameter (or if it cannot find a split that will reduce impurity)
  - a few other hyperparameters (described in a moment) control additional stopping conditions (min_samples_split, min_samples_leaf, min_weight_fraction_leaf, and max_leaf_nodes).

Note: CART is a greedy algo (searches for an optimum split at the top level already, and repeats, no check whether the split will eventually lead to the lowest possible impurity several levels down). OK for good solution, no guarantees for the optimal one - which is a problem that scales exponentially with m. With DT and CART, do trade for a "reasonably good" solution.. and quickly…

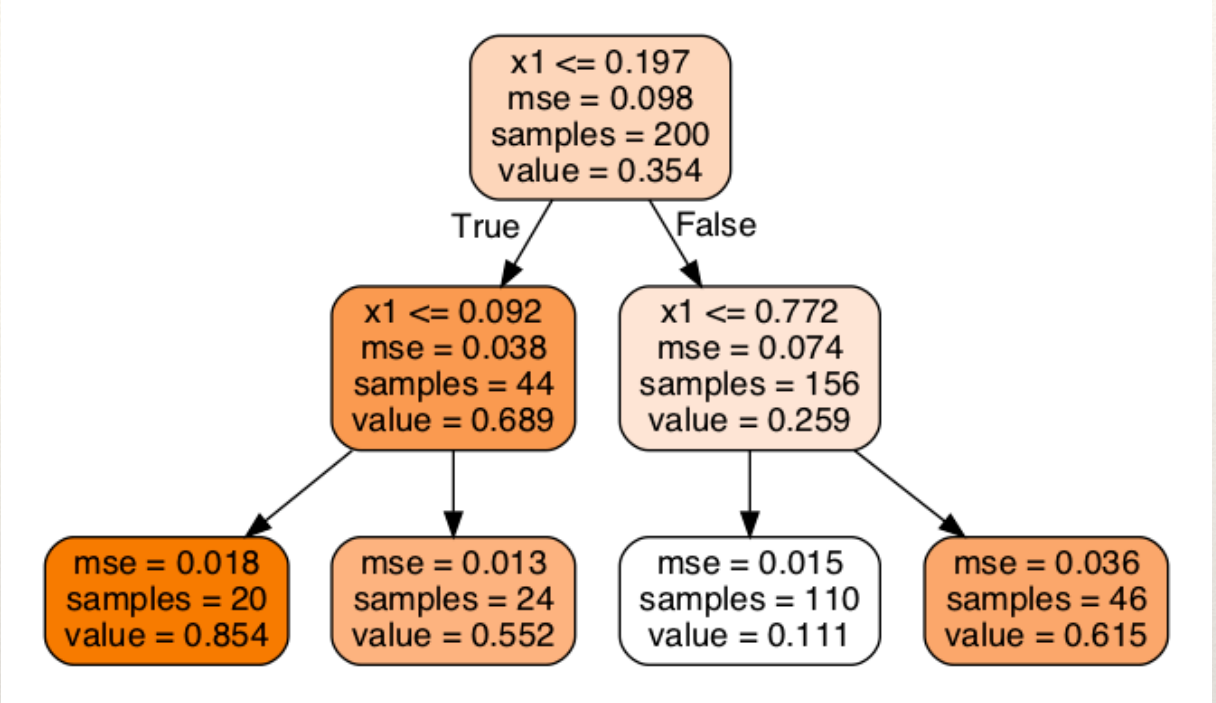
Another dataset: use of a DT for **regression**

Sklearn on a noise quadratic dataset
→ `DecisionTree`**`Regressor`**

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

creates this:



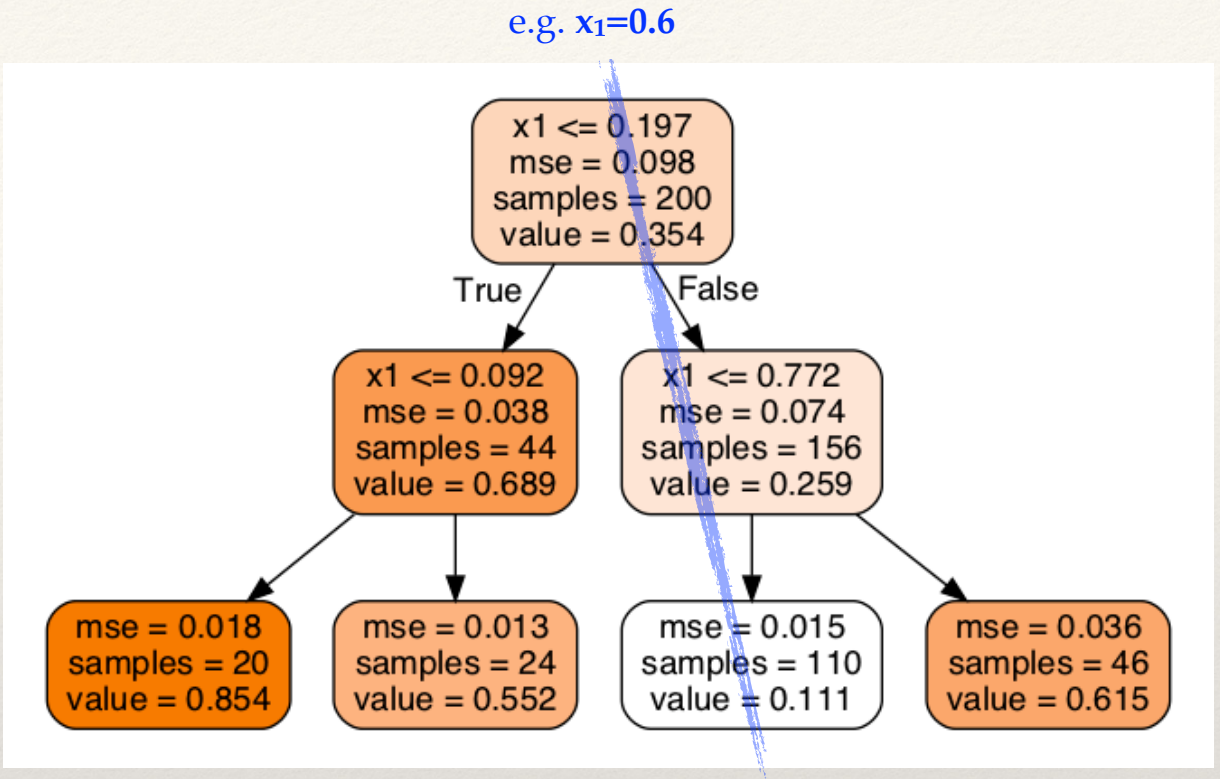
Another dataset: use of a DT for **regression**

Sklearn on a noise quadratic dataset
→ `DecisionTreeRegressor`

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

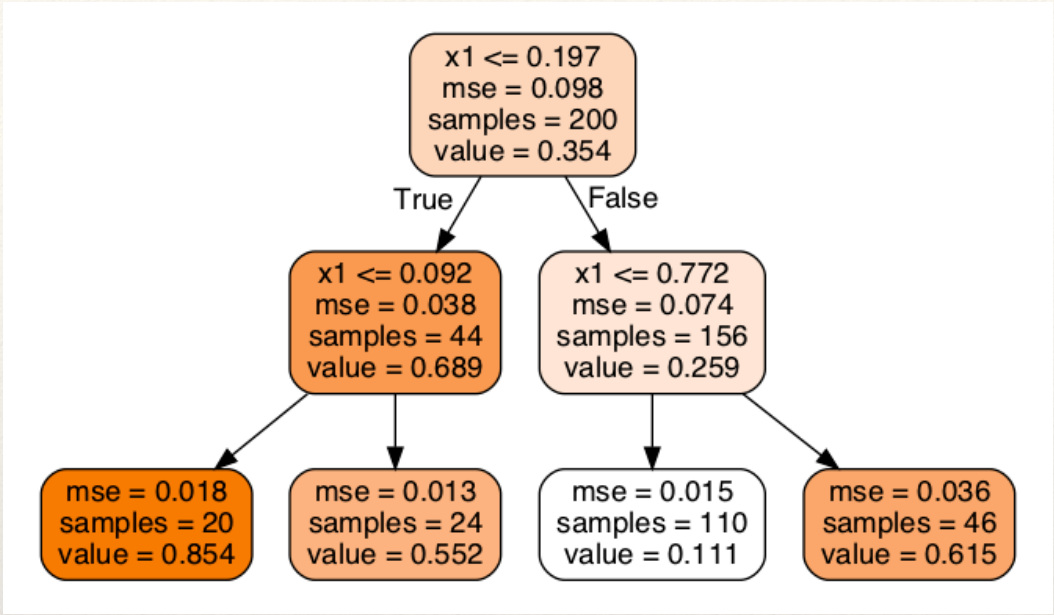
creates this:



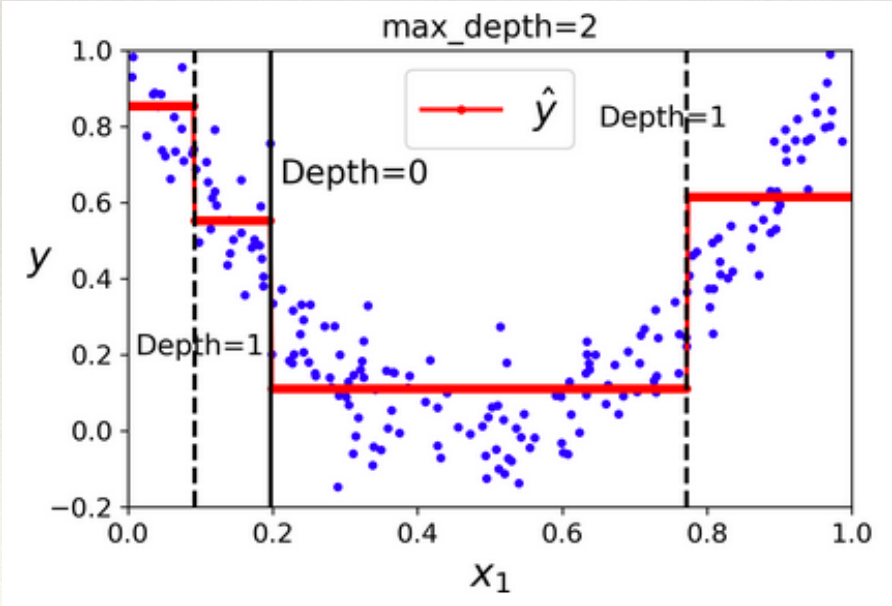
It does not predict a **class**, but a **value**

- example: I want a value prediction for a new instance with x_1=0.6, I transverse the tree starting from the top, and reach the leaf node that predicts value=0.111 - which is the average target value of the 110/200 training instances associated to this leaf node, and this prediction results in a Mean Squared Error (MSE) equal to 0.015 over these 110 instances.

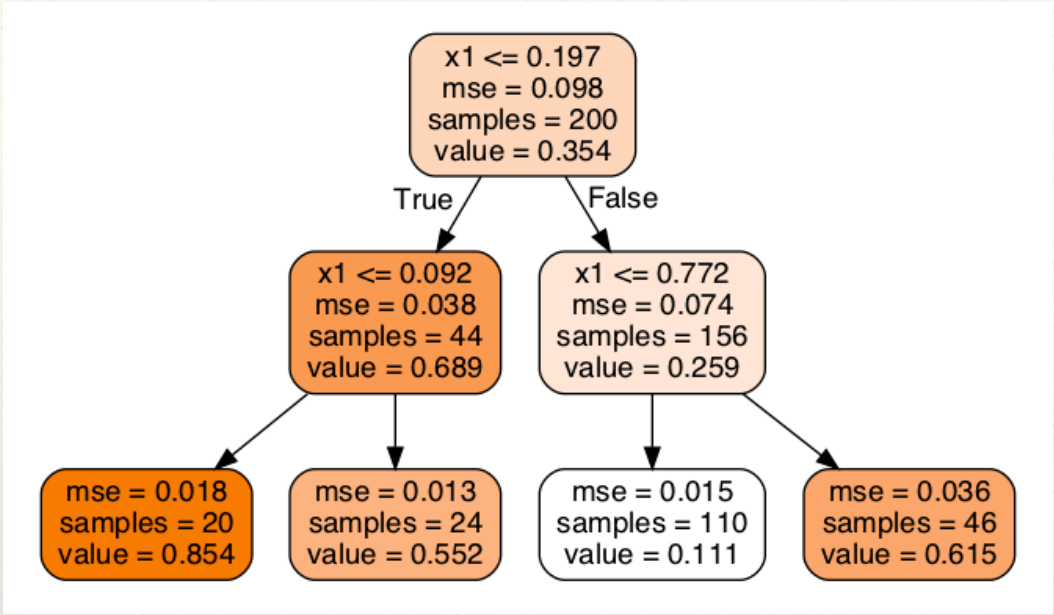
Another dataset: use of a DT for **regression**



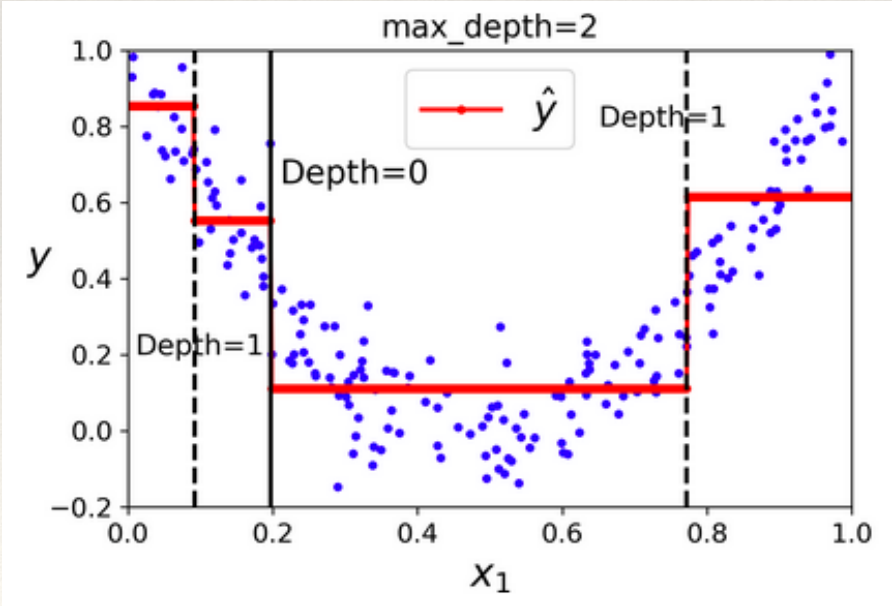
Note: the **predicted value** for each region is always the average target value of the instances in that region.



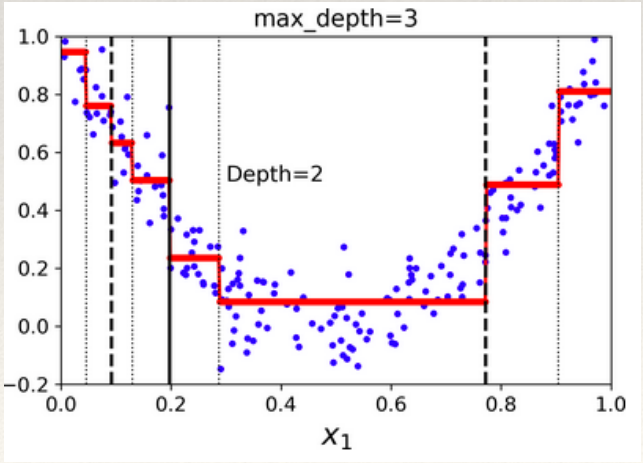
Another dataset: use of a DT for **regression**



Note: the **predicted value** for each region is always the average target value of the instances in that region.



If you change to max_depth=3:



Example of a DT for **regression**

The CART algo for regression (classification) tries to split the training set in a way that minimises impurity (MSE)

- The cost function that the CART algo tries to minimise for classification was:

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where $\begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$

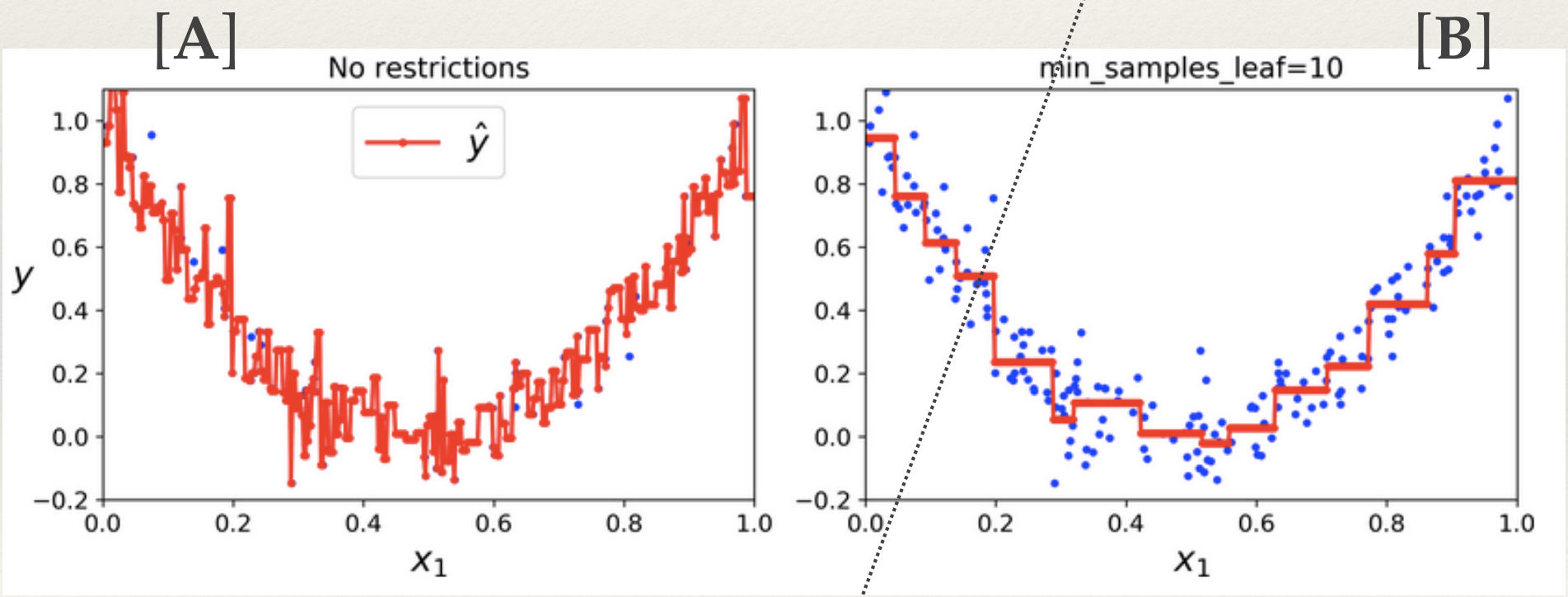
- … while for regression is:

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}} \quad \text{where} \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

DTs prone to overfitting

Note that - just like for classification tasks - DTs are prone to **overfitting** also when dealing with regression tasks.

- Using the default hyper-parameters, i.e. without any regularization, you get the predictions of type [A] below. Just setting min_samples_leaf=10 in sklearn results in a much more reasonable model as in [B] below



min_samples_leaf: minimum nb of examples required to be a leaf node

More

DTs prone to overfitting

from sklearn documentation:

max_leaf_nodes : *int, default=None*

Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_samples_leaf : *int or float, default=1*

The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

min_samples_split : *int or float, default=2*

The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

E.g. DecisionTreeClassifier(max_leaf_node=N) specifies N leaves at maximum, so - unless the tree builder has some other reasons to stop - it will hit the max. Suppose I run it with N=10, it might happen that some (e.g. 4) of the N leaves would have a very small amount of samples (e.g. ≲3) compared to the others 6 leaves (e.g. ≳50), this being a possible sign of over-fitting.

One could "prune" the tree a-posteriori (after training). Or, another viable option is to specify either min_samples_leaf (or min_samples_split) to better guide the training: it will likely get rid of the problematic leaves

- e.g. set the value 0.03 if you want at least 3% of examples

DT instabilities and limitations [1/2]

In summary, DTs have a lot of pros:

- simple to understand and **interpret**
- **easy** to use, **versatile**, **powerful**

DT instabilities and limitations [1/2]

In summary, DTs have a lot of pros:

- simple to understand and **interpret**
- **easy** to use, **versatile**, **powerful**

They also have cons:

- Explore and use right configs to prevent overfitting
- In general, main issue with DTs is that **they are very sensitive to small variations in the training data**
- Example: DTs are inclined towards orthogonal decision boundaries (all splits are perpendicular to an axis): this makes them sensitive to training set rotation
  - e.g. rotate a data points distribution by 45° and an easy split becomes an unnecessarily convoluted decision boundary. Both DTs fit the training set just fine, but one should not be surprised that the latter model may not generalise well.
    - one way to limit this problem is to use PCA, which often results in a better orientation of the training data..

DT instabilities and limitations [2/2]

Additionally, remember that since the training algo used by sklearn is stochastic, one may get different models even on the same training data

- unless one sets the random_state hyper-parameter

Random Forests can limit this instability by averaging predictions over many trees

- so a DT might be your *starting line* and not your *finish line*..

Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

Hands-on:
DT

Data Science and Computation PhD + Master in Bioinformatics
University of Bologna

Hands-on: DT

DT on the IRIS dataset

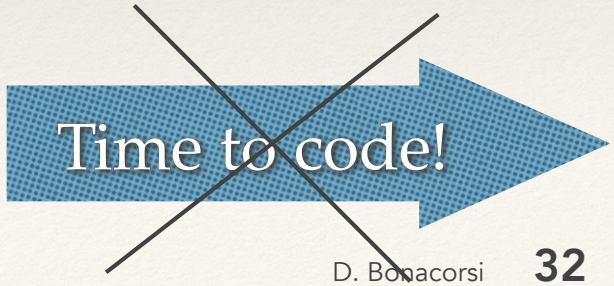
[credits to: A. Geron, "Hands-On Machine Learning With Scikit-Learn and Tensorflow"]

Coding material

AML2223Adv_DT.ipynb

→ gym on the notebook

No more ARROW and supporting slides..
time to hands-on with less guidance from me!



Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

Hands-on: MNIST2A

Data Science and Computation PhD + Master in Bioinformatics
University of Bologna

Hands-on: MNIST2A

Classification with Keras: a simple NN

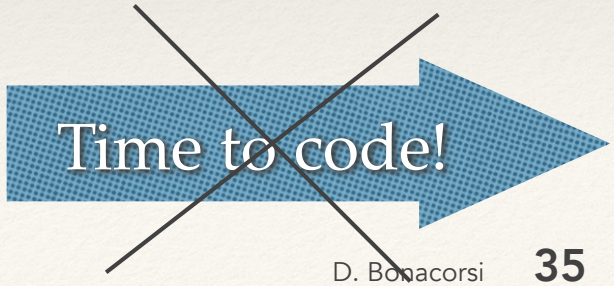
[credits to: A. Geron, "Hands-On Machine Learning With Scikit-Learn and Tensorflow"]

Coding material

AML2223Adv_HandsOn_MNIST2A.ipynb

→ go to the "**MNIST2A**" notebook

No more ARROW and supporting slides..
time to hands-on with less guidance from me!



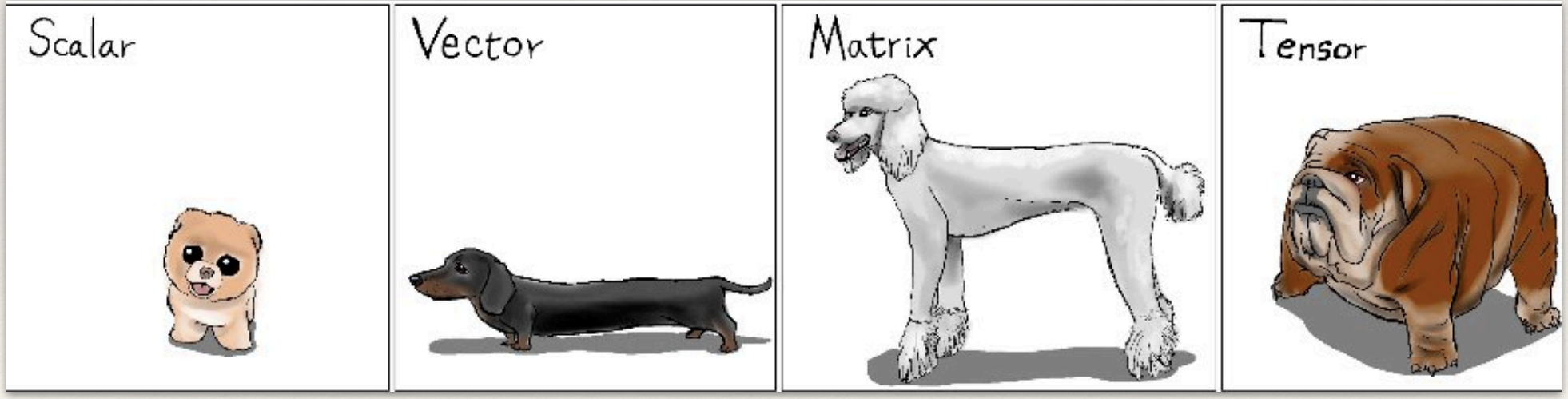
NNs' building blocks

Tensors and tensor operations

Tools.. on the data!

Let's "match" what we introduced about the **tools** we will use, with some notes on the actual **data structures** we will use for our datasets.

Data tensors



Real-world example of data tensors

Let's make data tensors more concrete, with some examples of what you may encounter in real life (or: "you might need to force your dataset into").

The data you'll manipulate will almost always fall into one of the following categories:

Vector data: **2D tensors** of shape (samples, features)

Timeseries data (or "sequence data"): **3D tensors** of shape (samples, timesteps, features)

Images : **4D tensors** of shape (samples, height, width, channels) or (samples, channels, height, width)

Videos: **5D tensors** of shape (samples, frames, height, width, channels) or (samples, frames, channels, height, width)

Let's discuss each.

Vector data

The most common case. In such a dataset:

- every single data point can be encoded as a vector
- → a batch of data will be encoded as a **2D vector data tensor** (i.e. an array of vectors)
- the first axis is the **samples axis** and the second axis is the **features axis**

Vector data

E.g.:

A dataset of people, where we consider e.g. each person's age, ZIP code, and income

- Each person can be characterised as a vector of 3 values, and thus an entire dataset of 100k people can be stored in a 2D tensor of shape (100'000, 3).

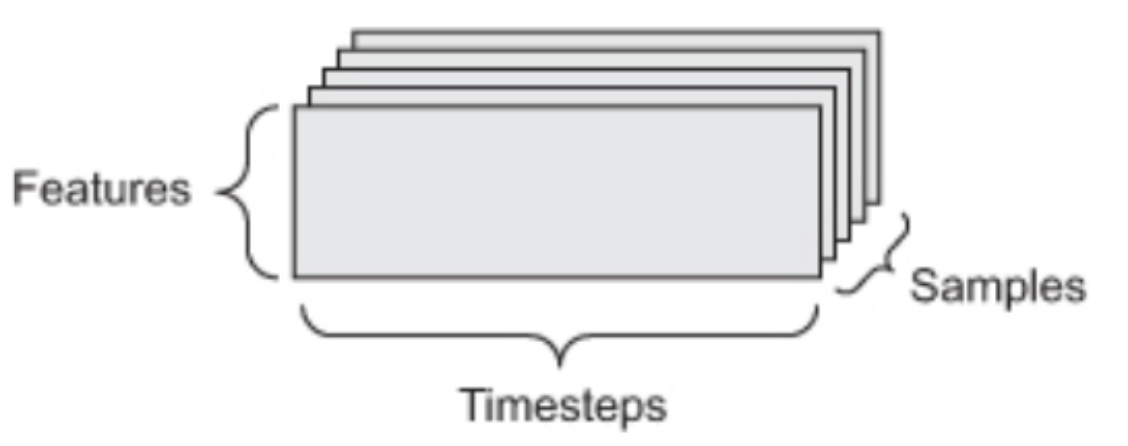
A dataset of text documents, where we represent each document by the counts of how many times each word appears in it (out of a dictionary of e.g. 20k common words)

- each document can be encoded as a vector of 20k values (one count per word in the dictionary), and thus an entire dataset of 500 documents can be stored in a tensor of shape (500, 20'000)

Time-series (or sequence data)

Whenever time matters in your data (or the notion of sequence order), it makes sense to store it → a **3D time-series (or sequence) data tensor** with an explicit time axis

- every sample can be encoded as a sequence of vectors (i.e. a 2D tensor)
- → a batch of data will be encoded as a **3D tensor** which has thus an explicit **time axis**
- the time axis is always the second axis (axis of index 1), by convention



Time-series (or sequence data)

E.g.:

A dataset of stock prices. Every minute, we store e.g. the current price of the stock, the highest price in the past minute, and the lowest price in the past minute.

- → every minute's information is encoded as a (3D) vector → an entire day of trading (390 minutes) is encoded as a 2D tensor of shape (390, 3) → 252 trading days' worth of data can be stored in a 3D tensor of shape (252, 390, 3). Here, each sample would be one day's worth of data

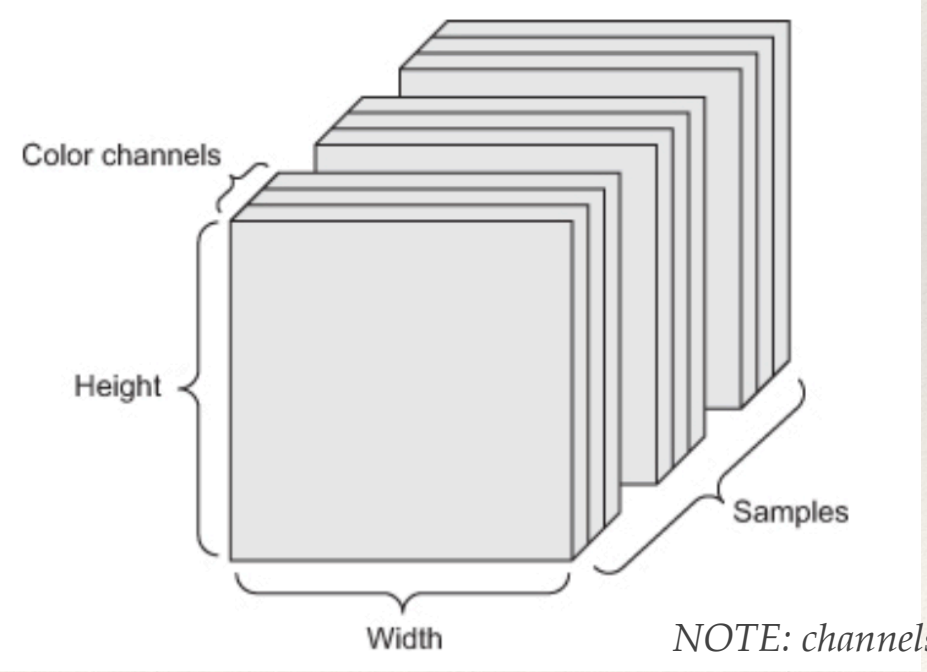
A dataset of tweets. We encode each tweet as a sequence of (now) 280 characters out of an alphabet of 128 unique characters

- → each character can be "one-hot-encoded" as a binary vector of size 128 (an all-0s vector except for a 1 entry at the index corresponding to the character) → each tweet can be encoded as a 2D tensor of shape (280, 128), and a dataset of 1M tweets can be stored in a tensor of shape (1000000, 280, 128)

Image data

Images typically have 3 dimensions: height, width, and color depth. This is for each sample, so → **4D image data tensors**

- grayscale images (e.g. MNIST) have only a single color channel and could thus be stored in 3D tensors, by convention image tensors are always 4D, with a one-dimensional color channel for grayscale images
- → a batch of 128 grayscale images of size 256×256 could thus be stored in a tensor of shape (128, 1, 256, 256), while a batch of 128 color images could be stored in a tensor of shape (128, 1, 256, 256)
  - NOTE: channels-last (here) vs channels-first conventions → see next slide



NOTE: channels-first convention

Image data

Careful about the conventions, for image data tensors.

There are 2 conventions for their shapes:

- the channels-last convention (e.g. used by TensorFlow)
  - it places the color-depth axis at the end: (samples, height, width, color_depth)
- the channels-first convention (e.g. used by Theano)
  - it places the color depth axis right after the batch axis: (samples, color_depth, height, width).
- And the Keras framework? Needless to say, it provides support for both formats.

Video data

A video can be interpreted as a sequence of frames, each frame being a color image → a vector of 4D image data tensors → **5D video data tensors**

- Video data is one of the few types of real-world data for which you'll need 5D tensors..
- → each frame can be stored in a 3D tensor (height, width, color_depth)
- → a sequence of frames can be stored in a 4D tensor (frames, height, width, color_depth)
- → a batch of different videos can be stored in a 5D tensor of shape (samples, frames, height, width, color_depth)

Video data

E.g.

A dataset of YouTube videos. A 60-second, 144×256 YT video clip (w/ colour) sampled at 4 frames per second would have 240 frames. A batch of 4 such video clips would be stored in a tensor of shape (4, 240, 144, 256, 3).

- total: 106,168,320 values (!)
- assume the *dtype* of the tensor was float32 → each value would be stored in 32 bits → the tensor would be 405 MB.
  - This builds us quite heavy storage needs for videos, in theory. In practice: videos you encounter in real life are much lighter, because they aren't stored in float32, they're typically compressed by a large factor (e.g. in the MPEG format)

Tensor operations

Coding material

AML2223Adv_Tensors.ipynb

→ we will proceed on slides though

The gears inside NNs: tensor operations

Any computer program does what it does, if reduced to its core actions, via a combination of a relatively small set of binary operations on binary inputs

- i.e. take bits and perform combinations of ANDs, ORs, NORs, ..

Much along the same lines, all "*transformations*" that a deep NN can learn can be reduced to a handful of **tensor operations** applied to tensors of numerical data

- i.e. take tensors of various ranks, and do additions, multiplications, ..

Inside a layer instance: basic tensor ops

```
my_network = keras.models.Sequential([
  keras.layers.Dense(512, activation='relu'),
  keras.layers.Dense(10, activation='softmax')
])
```

```
output = relu(dot(W, input) + b)
```

Take our first NN (see notebook). We built the NN by stacking Dense layers on top of each other. Take one (see above):

- this layer can be interpreted as a **function**: it takes as "input" a tensor and returns as "output" a new **representation** of the input tensor

Specifically, the function does:

- a dot product, an addition and then applies a function relu(x) (which is basically max(x,0))

(NOTE: of course all this is linear algebra.. here: using Numpy snippets instead)

Element-wise operations

If we naively implement relu(x) and the addition in Python using Numpy, we may just use for loops:

```
def naive_relu(x):
    assert len(x.shape) == 2

    x = x.copy()

    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

```
def naive_add(x, y):
    assert len(x.shape) == 2

    assert x.shape == y.shape
    x = x.copy()

    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

The relu operation and the addition are *element-wise* operations

- ops applied independently to each entry in the tensors being considered
- highly adequate to massively parallel implementations (i.e. *vectorized* implementations)

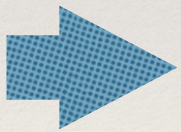
Same principle applies for element-wise multiplication, subtraction, ..

Element-wise operations: behind the scenes

What's happening behind the scene?

- we use Numpy arrays → ops available as well-optimized built-in Numpy functions → they delegate the hard work to BLAS (Basic Linear Algebra Subprograms) implementations, that are low-level, highly parallel, efficient tensor-manipulation routines
  - typically implemented in Fortran or C

Try it out on the notebook.



Also for larger tensors..
It is pretty fast.

```
[25] x = np.array([[1, 1],
                   [1, 1]])
     y = np.array([[2, 2],
                   [2, 2]])

[28] z = x + y
     z

     array([[3, 3],
            [3, 3]])

[27] %%time
     z = x + y
     z

     CPU times: user 20 µs, sys: 3 µs, total: 23 µs
     Wall time: 27.7 µs
```

Element-wise operations: broadcasting

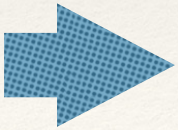
But we have "naive" implementations

- i.e. only supporting the addition of 2D tensors with identical shapes

Already in the Dense layer we have an addition of a 2D tensor with a vector. What happens?

Broadcasting. When possible, and if there's no ambiguity, the smaller tensor will be broadcasted to match the shape of the larger tensor, in 2 steps:

- axes (called *broadcast axes*) are added to the smaller tensor to match the ndim of the larger tensor
- the smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor

See on the notebook 

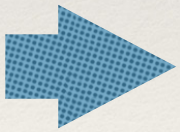
```
[ ]  def naive_add_matrix_and_vector(x, y):
         assert len(x.shape) == 2
         assert len(y.shape) == 1
         assert x.shape[1] == y.shape[0]
         x = x.copy()

         for i in range(x.shape[0]):
             for j in range(x.shape[1]):
                 x[i, j] += y[j]
         return x
```

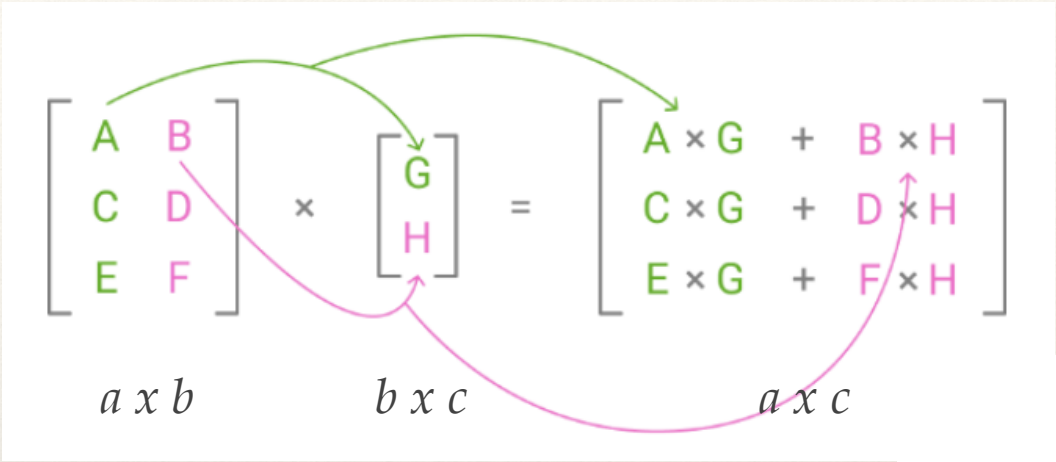
Dot products

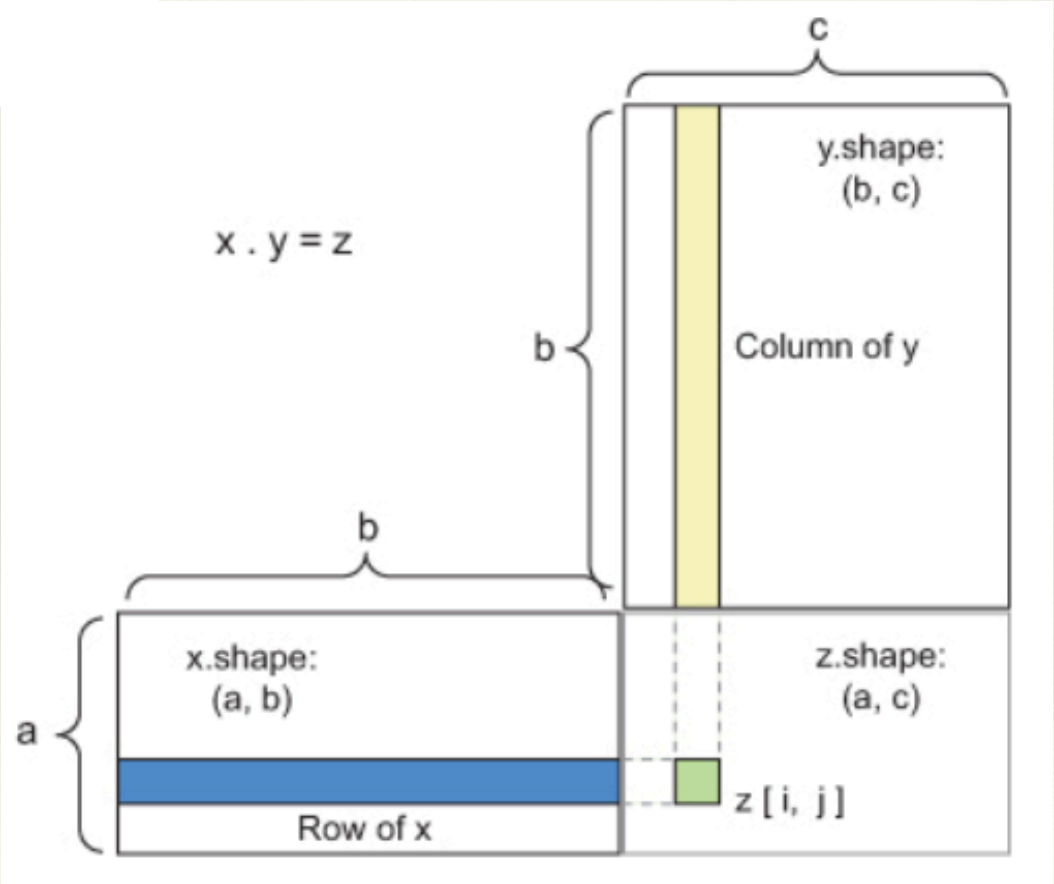
The **tensor product** (or **dot product**) is the most common and useful tensor op

- NOTE: not to be confused with an element-wise product!
  - in Numpy/Keras/TF/Theano: use the * operator
- contrary to element-wise operations, a dot product combines entries in the input tensors
  - in Numpy/Keras/Theano (not in TF): use the dot operator

See on the notebook. 

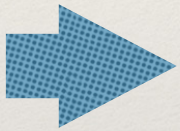
Dot products "visualised"





Visualising tensors as "boxes" of numbers, and aligning them

Tensor reshaping (incl. transposition)

See on the notebook. 

From a geometric interpretation of tensor ops..

The tensors' content can be interpreted as point coordinates in some geometric space → tensor operations have a geometric interpretation

- e.g. a vector A[1,2] can be interpreted as the coordinate of a point in a 2D space, and one can even draw a vector as if it was a position vector of a physical point w.r.t the origin of a coordinate system
- if you take another vector and make the sum, analytically or visually, you get the coordinate of a third point in the same geometrical space

This is general - elementary geometric operations such as affine transformations, rotations, scaling, .. can be expressed as tensor operations

- e.g. a rotation of a 2D vector by an angle θ can be achieved via a dot product with a 2×2 rotation matrix R=[u,v], where u and v are vector of the plane, namely u=[cos(θ), sin(θ)] and v=[-sin(θ), cos(θ)]

.. to a geometric interpretation of DL

NNs consist entirely of chains of tensor ops → all these tensor ops are just geometric transformations of the input data → we can interpret a NN as a (very complex) **geometric** transformation in a high-dimensional space, implemented via a long series of simple steps (operations)

Visual example:

- DL as the process of uncrumpling a complicated manifold of data (a paper ball done by sheets of different colours), i.e. figure out which set of many (deep!) transformations - need to be applied one after another (layers!) that may make the original paper sheets cleanly separable again

Gradient-based optimization

Trainable parameters

Let's restart from here:

```
my_network = keras.models.Sequential([
  keras.layers.Dense(512, activation='relu'),
  keras.layers.Dense(10, activation='softmax')
])
```

```
output = relu(dot(W, input) + b)
```

W and b are **trainable parameters** (or **weights**) of that layer, and they are **attributes** of the layer (namely, the **kernel** and **bias** attributes). They contain the information learned by the NN from exposure to training data

- from the Basic part of the course: the θ parameters

They are **tensors**. What's in them?

- initially, they are filled with small random values (**random initialization**)
- of course, applying a function to random stuff does not yield any useful representation - it is just a starting point. Weights need to be adjusted, based on some feedback. This gradual adjustment is called **training** (the "learning" in "machine learning")
  - see next

Training

The training happen in loops. Let's zoom into one loop:

1. Draw a **batch** of training samples x and corresponding targets y
2. Run the NN, i.e. run a **forward pass** on x to obtain predictions y_pred
3. Compute the loss of the NN on this batch (i.e. a measure of the mismatch between y_pred and y)
4. Update all weights of the network in a way that slightly reduces the loss on this batch

When you eventually end up with a NN that has a very low loss on its training data, and you call it done, you judge that the NN has "learned" to map its inputs to correct targets.

Let's revisit steps 1-4 on the basis of what happens behind the scene, and how complex it is.

Training loop evaluation

One training loop:

1. Draw a **batch** of training samples x and corresponding targets y
   - easy - just I/O code
2. Run the NN, i.e. run a **forward pass** on x to obtain predictions y_pred
3. Compute the loss of the NN on this batch (i.e. a measure of the mismatch between y_pred and y)
   - not as easy as 1, but both 2 and 3 are tensors ops - one can implement everything from scratch or use existing (optimized, sophisticated) libraries
4. Update all weights of the network in a way that slightly reduces the loss on this batch
   - this is tough. You need to update the NN's weights, decide if to increase or decrease, and by how much. How?

How to perform weights updates?

I can naively think of freezing all weights in the NN, except the one scalar coefficient being considered → then, try different values for that coefficient

- e.g. initial value 0.3. I do a first fwd pass on a batch of data → loss of the NN on that batch is 0.9. If I change the coefficient to 0.35 (0.25) and rerun the fwd pass, the loss increases (decreases) to 1.0 (0.8) → you update it e.g. by -0.05 to contribute to minimising the loss. And you need to repeat this for all coefficients in the NN
- terribly inefficient → 2 (expensive) fwd passes needed per each coefficient (and they are many)

A much better approach is to take advantage of the fact that all ops used in the NN are **differentiable**

- hence compute the **gradient** of the loss with regard to the NN's coefficients → you can then move the coefficients in the opposite direction from the gradient, thus decreasing the loss

Gradient as the derivative of a tensor operation

- i.e. the generalisation of the concept of derivatives to functions of multidimensional inputs (i.e. that take tensors as inputs)

Normal equation vs GD

We discussed it in the Basic part of the course in quite some detail

- we know already that an iterative method is highly desirable!

The strategy is simple:

- we modify the parameters little by little based on the current loss value on a random batch of data
- as we are dealing with a differentiable function, we can compute its gradient
- the gradient gives us an efficient way to implement the difficult step 4 as per our training loop → we just need to update the weights in the opposite direction from the gradient, and the loss will be a little less every time

Let's hence modify our initial training loop

- see next

SGD

The "new" training loop:

1. Draw a **batch** of training samples x and corresponding targets y
2. Run the NN, i.e. run a **forward pass** on x to obtain predictions y_pred
3. Compute the loss of the NN on this batch (i.e. a measure of the mismatch between y_pred and y)
4. Compute the gradient of the loss with regard to the network's parameters (a ***backward pass***).
5. Move the parameters a little in the opposite direction from the gradient thus reducing bit a the loss on the batch

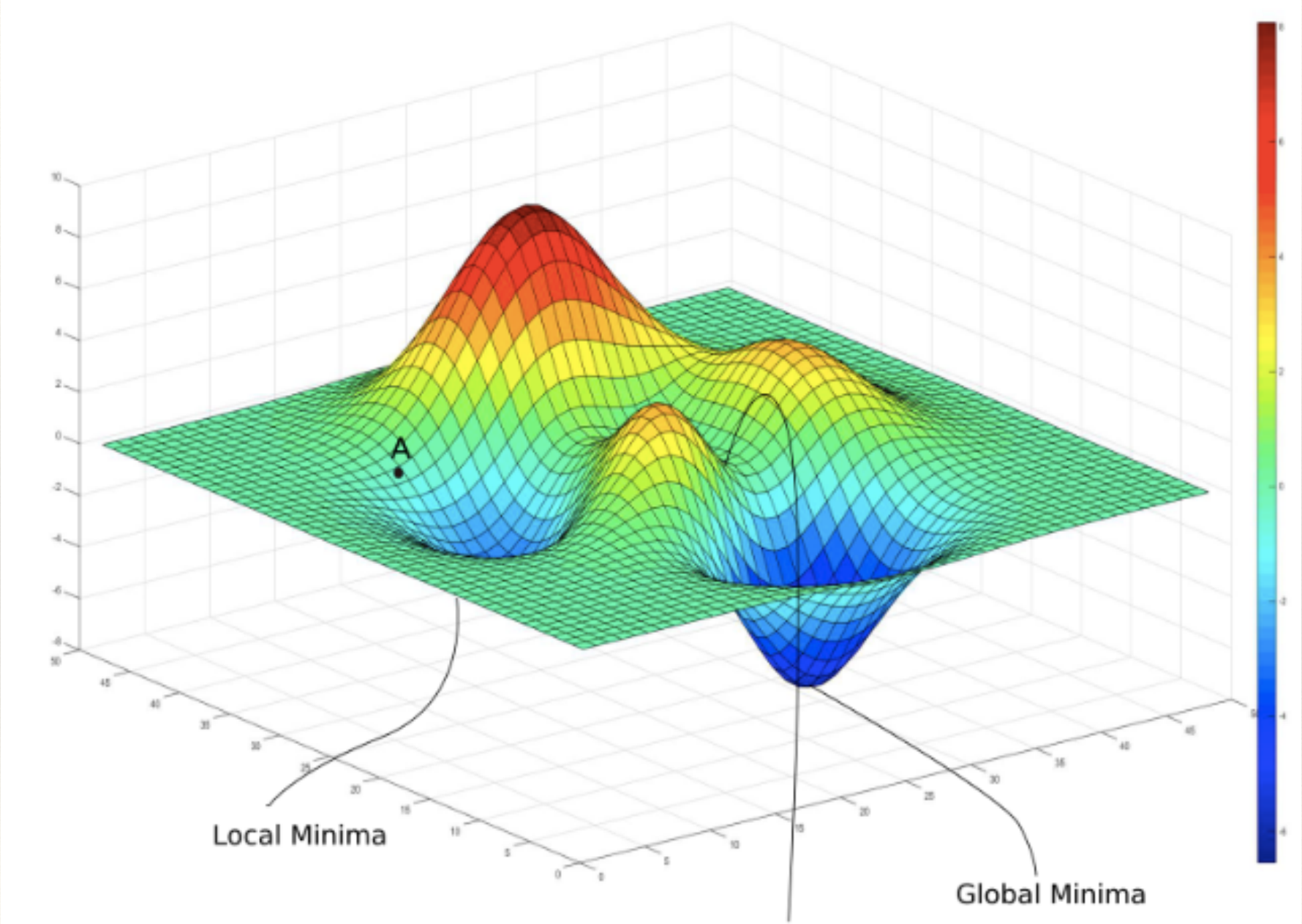
This is **mini-batch stochastic gradient descent** (**mini-batch SGD**)

- "stochastic" due to the random draw of samples to form each batch

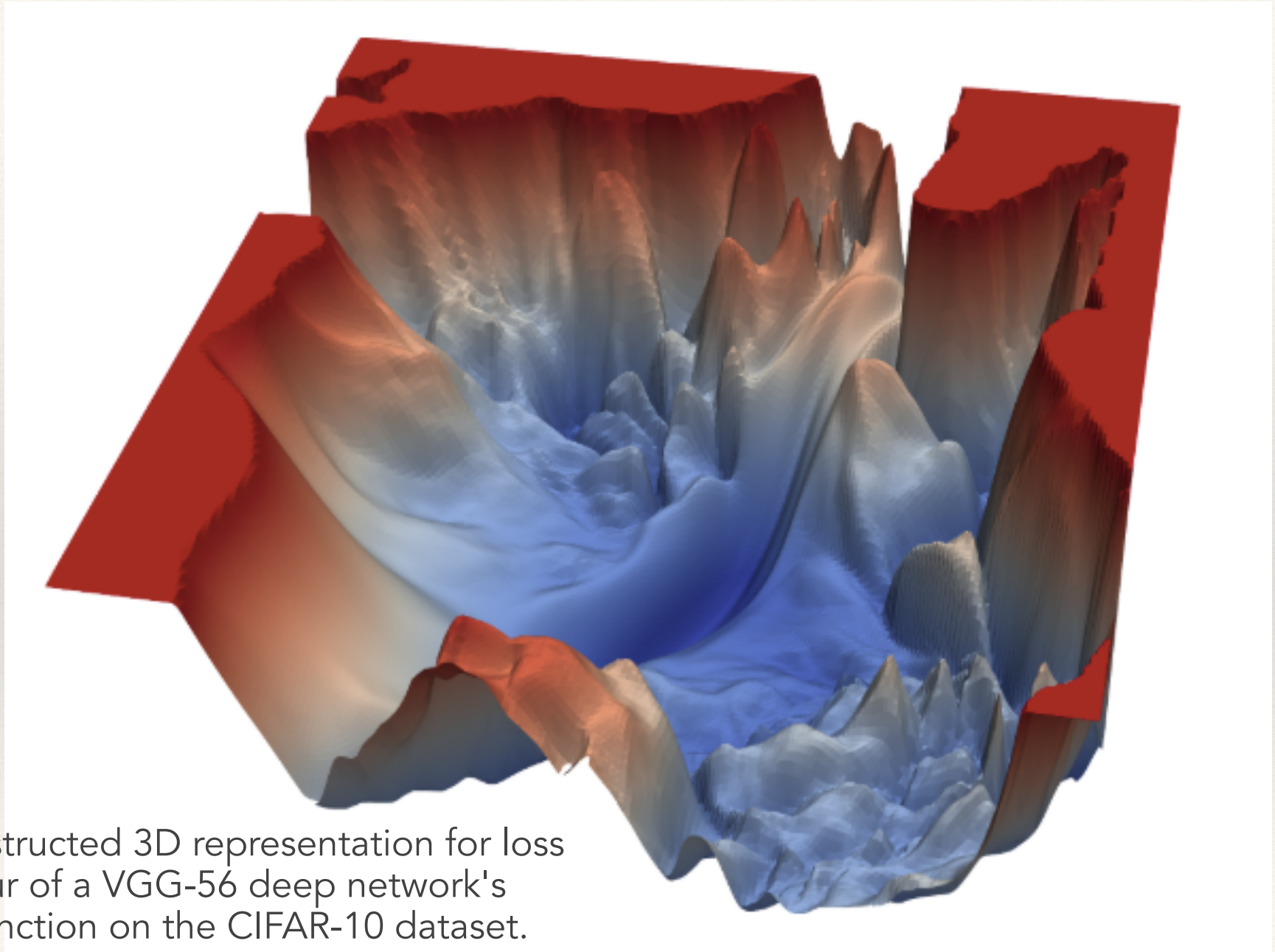
A good compromise among two extremes:

- "true SGD": no batch → perform training loop on every training instance
- "batch SGD": one single batch → perform training loop on all training set every time

Complexity of a loss curve? low here…



Complexity of a loss landscape? **high** here…



A constructed 3D representation for loss contour of a VGG-56 deep network's loss function on the CIFAR-10 dataset.

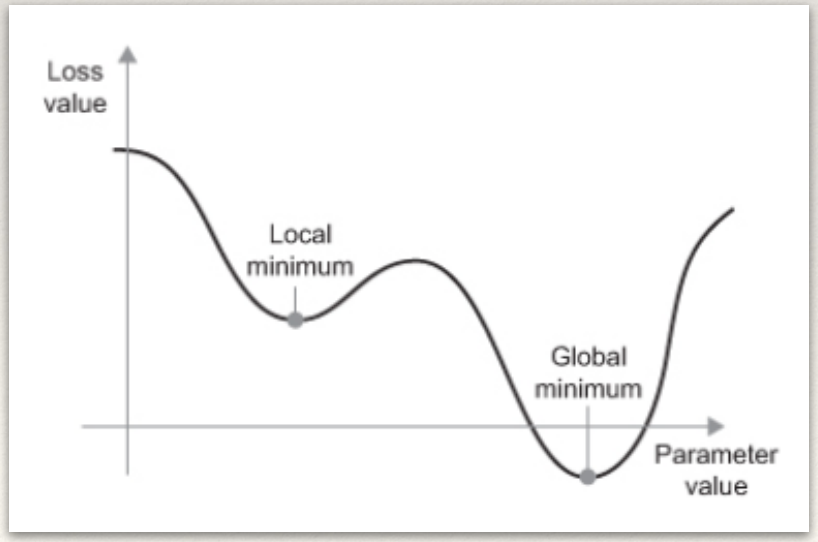
Optimisers

Multiple variants of SGD exist, and they are known as **optimisers** (or **optimisation methods**)

- e.g. **SGD with momentum**, as well as **Adagrad**, **RMSProp**, and others
- they differ by taking into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients

The concept of momentum (derives from physics) is interesting

- it addresses two SGD issues: convergence speed and local minima
- momentum is implemented by moving at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration)




```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum - learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

(naive implementation of SGD with momentum)

Chaining derivatives: the **backpropagation** algo

In the training loop discussed so far, we can casually think of a "NN function" that is differentiable → we can compute its derivatives

In practice, a NN function consists of *many* tensor operations chained together, each of which has a simple, known derivative

- e.g. f(w1, w2, w3) = a(w1, b(w2, c(w3))) is a NN function f composed by 3 tensor ops a, b and c acting on weight tensors w1, w2 and w2
- to derive f above, I would apply the chain rule (f(g(x)) = f'(g(x)) * g'(x))

Applying the chain rule to the computation of the gradient values of a NN gives rise to an algo called **backpropagation** (or **reverse-mode differentiation**).

- it starts with the final loss value and works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter had in the loss value.

A note

Nowadays, and in the foreseeable future, ML researchers will implement NNs in modern frameworks that are capable of ***symbolic differentiation*** (such as TensorFlow)

- given a chain of operations with a known derivative, they can compute a gradient *function* for the chain (by applying the chain rule) that maps network parameter values to gradient values
- when you have access to such a function, the backward pass is reduced to a call to this gradient function - no need to re-implement the backpropagation algo by hand

No deep dive on backprop in this Applied ML course, as most important is just a good intuition of how gradient-based optimization works.