Ensemble Methods

Plan

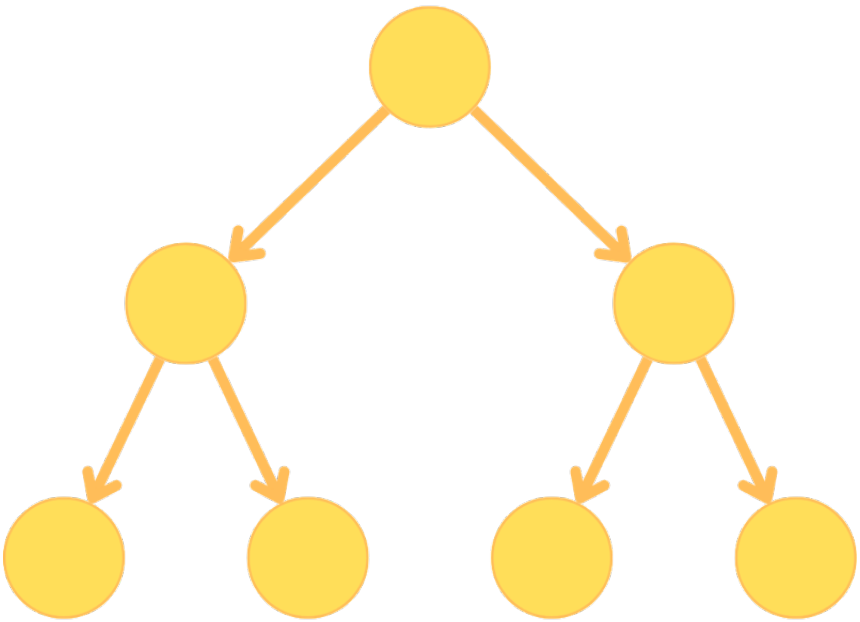
1. Decision Trees
2. Bagging (e.g. `RandomForest`)
3. Boosting (e.g. `GradientBoosting`)
4. Stacking

1. Decision Trees

Decision Trees are hierarchical supervised learning algorithms

They primarily help us with:

- classification and Regression
- non-linear modeling
- breaking down data through binary decisions



1.1 🖥️ `DecisionTreeClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeCla

In [ ]:
```
from sklearn.datasets import load_iris

iris = load_iris()
data = pd.DataFrame(data= np.c_[iris['data'], iris['target']],
                    columns= iris['feature_names'] + ['target'])
data.drop(columns=['sepal length (cm)', 'sepal width (cm)'], inplace=True)
data
```

Out[ ]:

	petal length (cm)	petal width (cm)	target
0	1.4	0.2	0.0
1	1.4	0.2	0.0
2	1.3	0.2	0.0
3	1.5	0.2	0.0
4	1.4	0.2	0.0
...	...	...	...
145	5.2	2.3	2.0
146	5.0	1.9	2.0
147	5.2	2.0	2.0
148	5.4	2.3	2.0
149	5.1	1.8	2.0

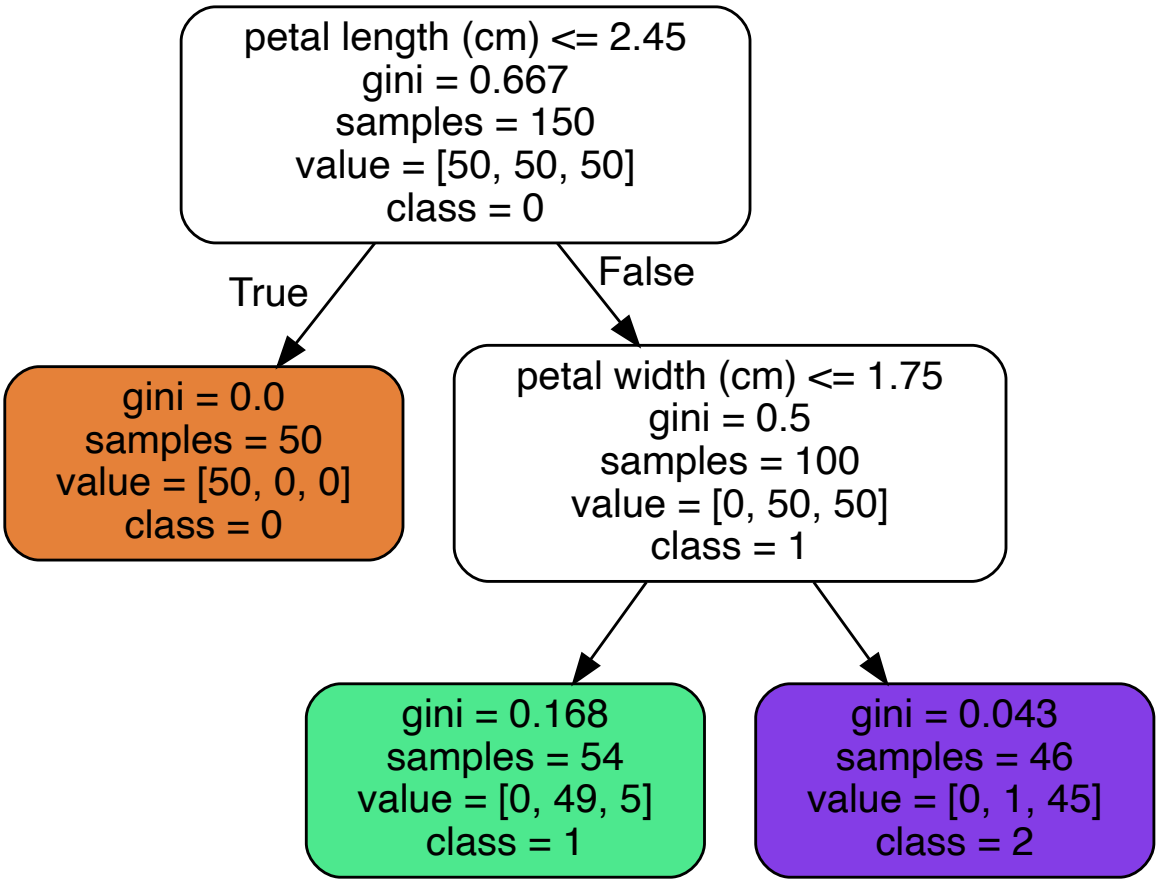
150 rows × 3 columns

In [ ]:
```
X = data.drop(columns=['target']).values
y = data.target.values
```

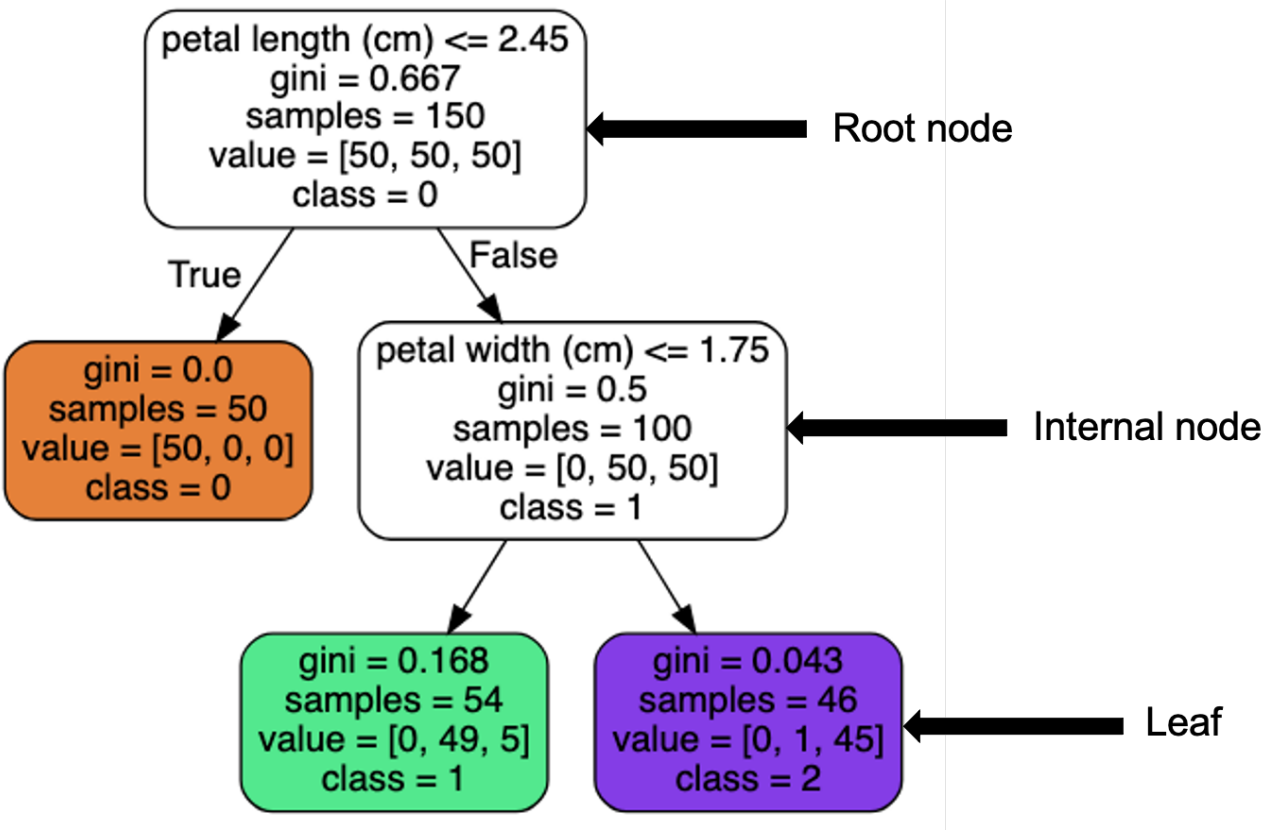
In [ ]:
```
# Instantiate and train model
from sklearn.tree import DecisionTreeClassifier
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=2)
tree_clf.fit(X,y)
```

Out[ ]:
```
DecisionTreeClassifier(max_depth=2, random_state=2)
```

In [ ]:
```
import graphviz

# Export model graph
from sklearn.tree import export_graphviz
export_graphviz(tree_clf, out_file="iris_tree.dot",
                feature_names=data.drop(columns=['target']).columns,
                class_names=['0','1','2'],
                rounded=True, filled=True)

# Import model graph
with open("iris_tree.dot") as f:
    dot_graph = f.read()
    display(graphviz.Source(dot_graph))
```



Jargon



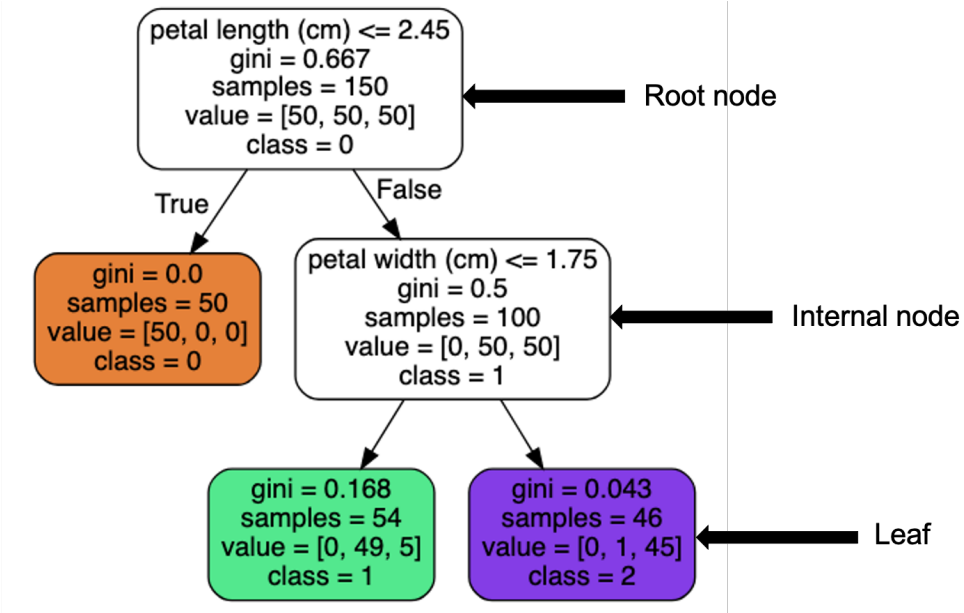
Gini Index

The Gini index measures the ability of each feature to **separate** the data

It calculates the **impurity** of each node, between [0, 1]; the lower the better

$$\text{Gini}(\text{node}) = 1 - \sum p_i^2$$

p_i
being the ratio between the observations in a class
i
and the total number of observations remaining at a given node



In [ ]:
```
# Calculate gini of root node
1 - (50/150)**2 - (50/150)**2 - (50/150)**2
```

Out[ ]:
```
0.6666666666666665
```

In [ ]:
```
# Calcultate gini green leaf
1 - 0**2 - (49/54)**2 - (5/54)**2
```

Out[ ]:
```
0.1680384087791495
```

How Do We "Grow" a Tree?

The tree's structure is defined through the following steps:

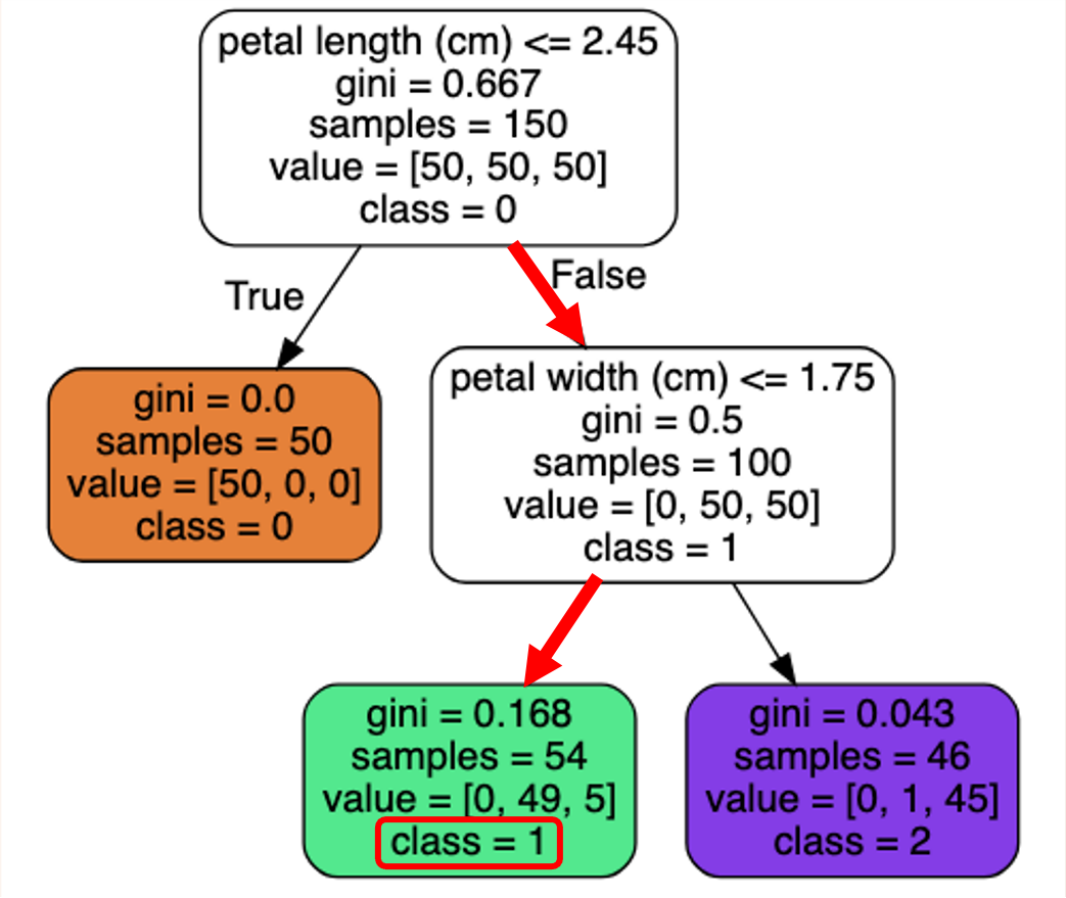
1. Start at root node, which contains your entire dataset
2. Try various combinations of **(feature, threshold)** combos; each would split your dataset into 2 child nodes
3. For each combination, compute the **weighted average Gini index** of both child nodes (weighted by number of instances)
4. Select the **(feature, threshold)** combo yielding the **lowest** index (i.e. the "purest child nodes")
5. Split the dataset in two using this rule
6. Repeat step 2 for both subsets
7. Stop when no feature improves node impurity (at what risk?)

[Source: CART algorithm (Aurelien Geron) (https://wagon-public-datasets.s3.amazonaws.com/data-science-images/lectures/machine-learning/tree_growth_aurelien_geron.png)](https://wagon-public-datasets.s3.amazonaws.com/data-science-images/lectures/machine-learning/tree_growth_aurelien_geron.png)

Predicting

- A new point is passed through the tree from top to bottom until it reaches a leaf
- The most represented class in that leaf is the predicted class for a given data point

Example: consider the following data point: `X_new = [4(length), 1(width)]`



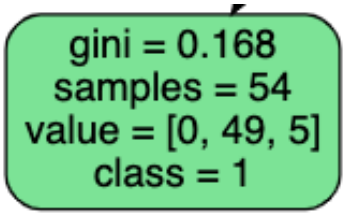
In [ ]:
```
# Let's predict the class of a new flower
print(tree_clf.predict([[4,1]]))
```

```
[1.]
```

In [ ]:
```
# Predict proba
print(tree_clf.predict_proba([[4,1]]))
```

```
[[0.         0.90740741 0.09259259]]
```

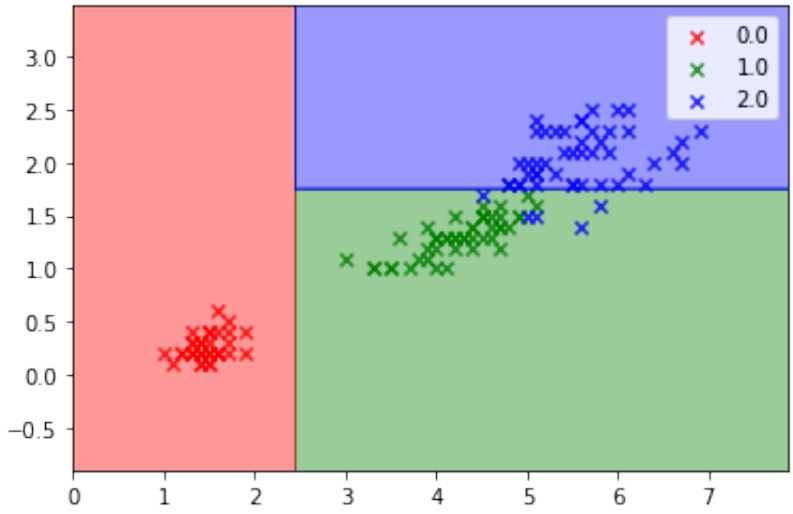
❗ *90% is not a "probability" per se, it corresponds to the ratio of classes in the node*



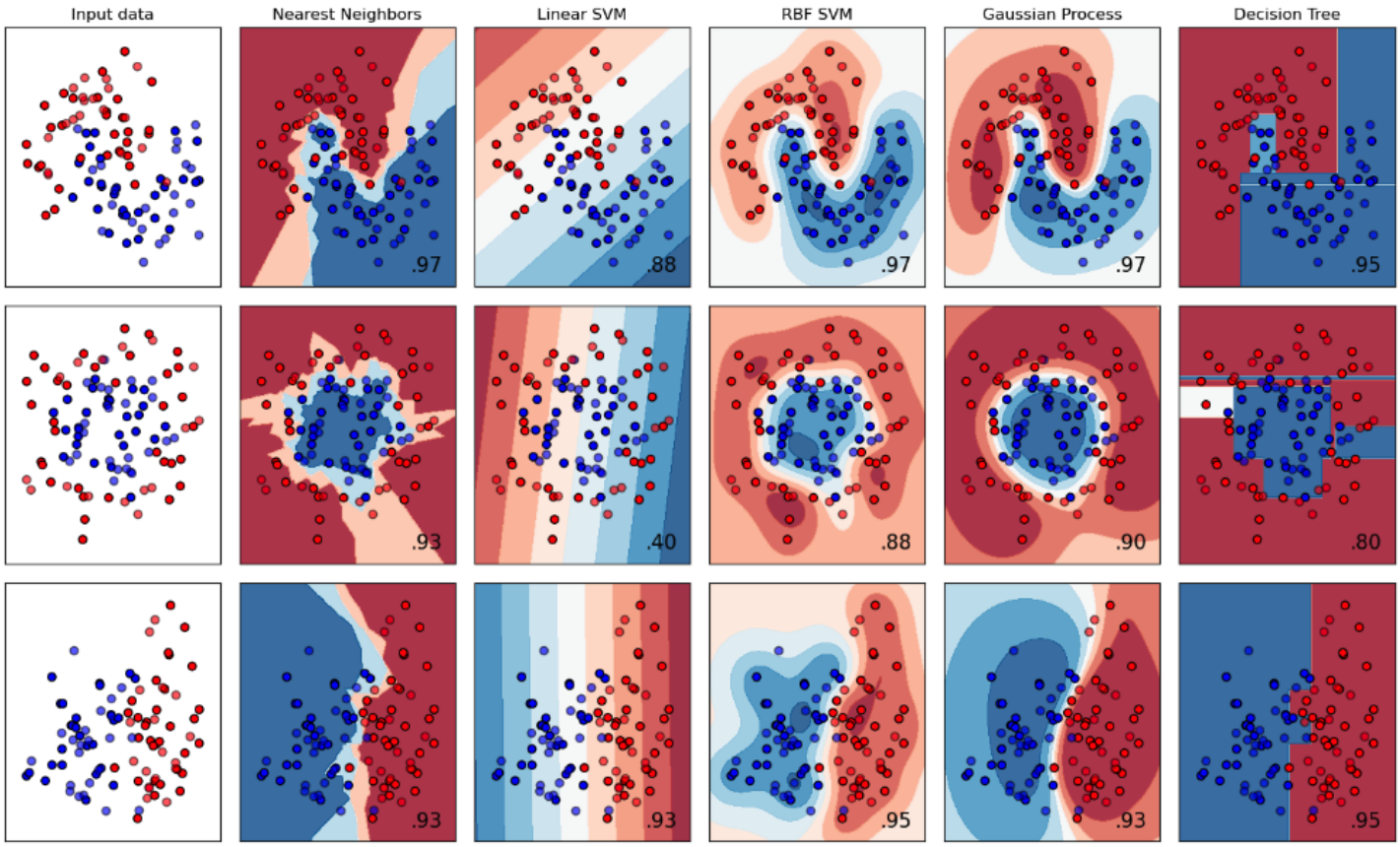
Trees are not *calibrated* classifiers (as opposed to logistic regressors)

Think about decision trees as "orthogonal" classifiers

In [ ]:
```
# Hand-made function to plot all predictions
plot_decision_regions(X, y, classifier=tree_clf)
```

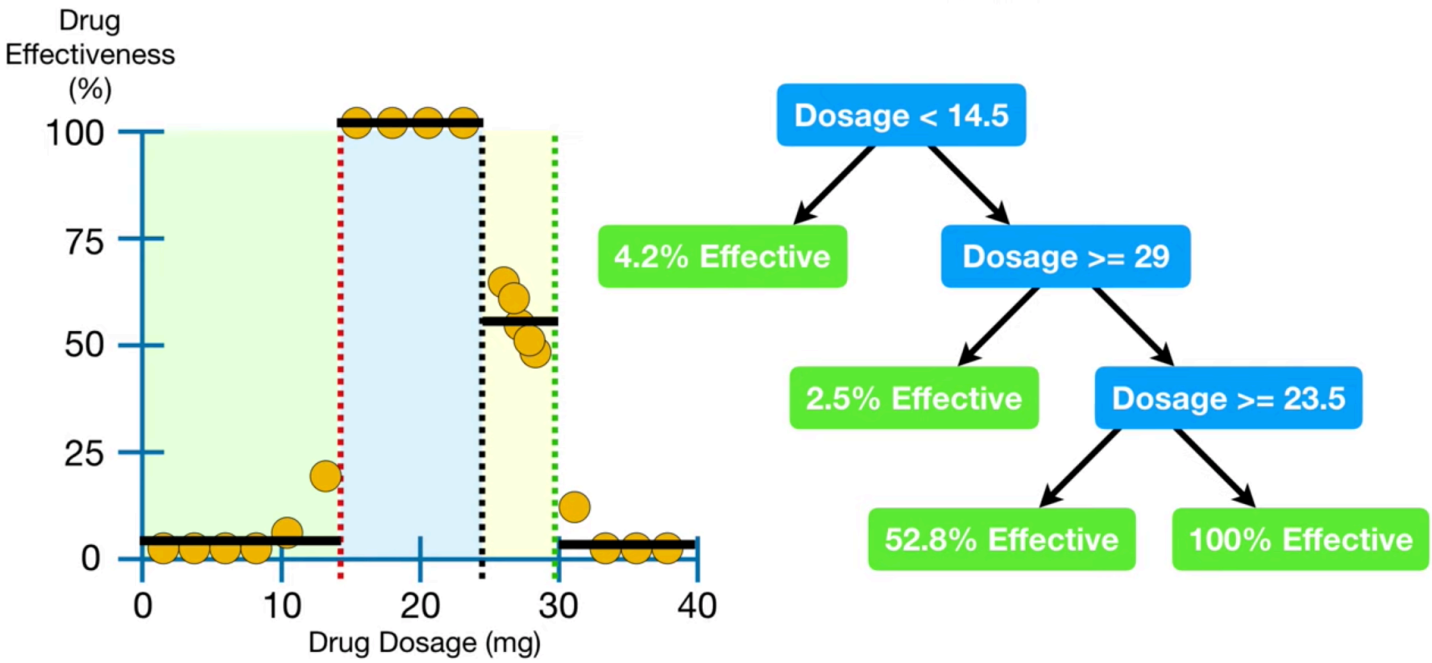


👇 scikit-learn classifier comparison (source (https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html))

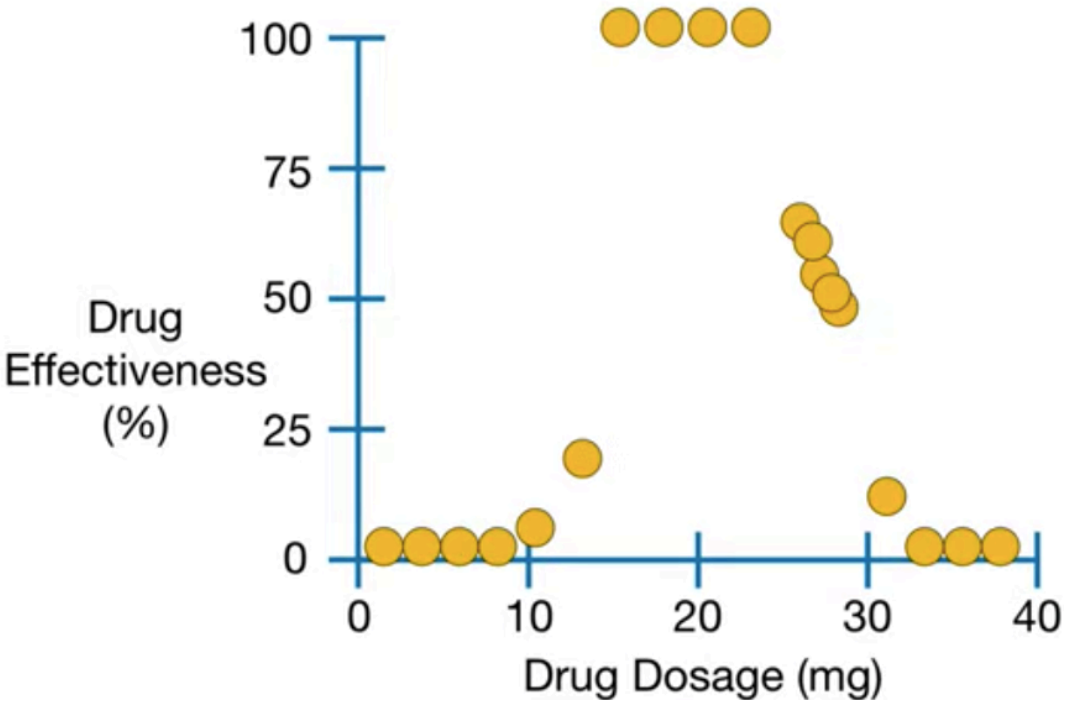


1.2 `DecisionTreeRegressor` (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRe

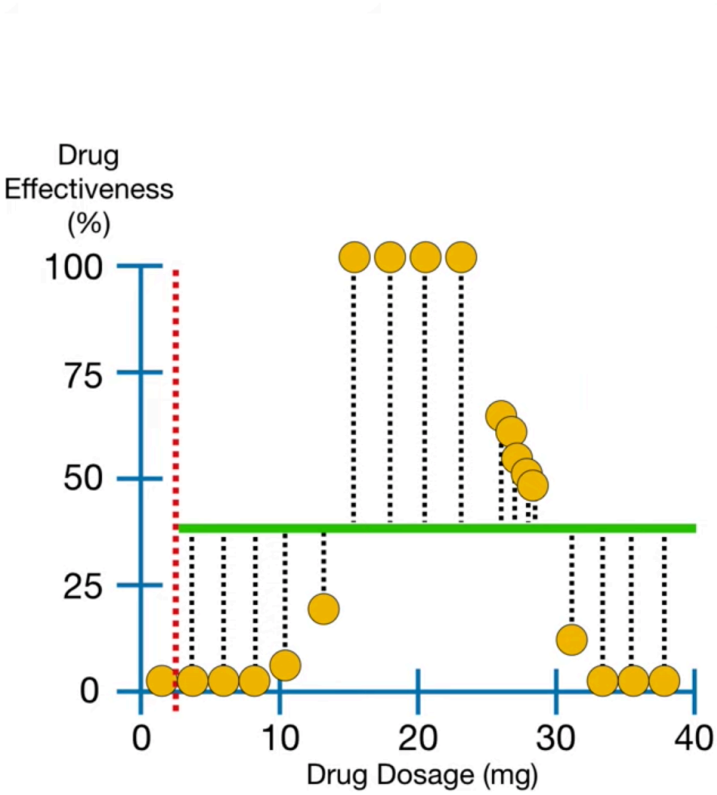
The goal of regression trees is to predict a continuous value, they are "grown" differently than classification trees

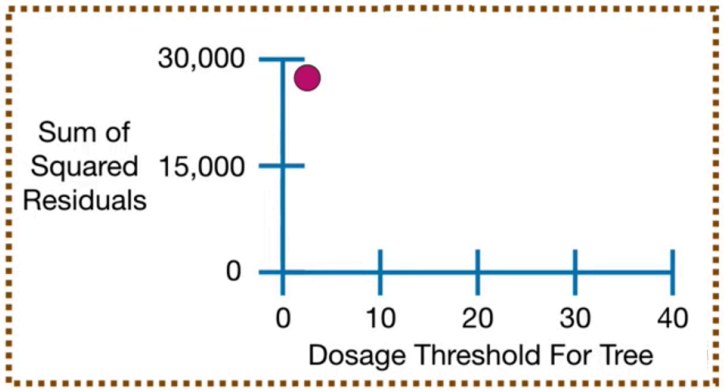


Growing the Regression Tree

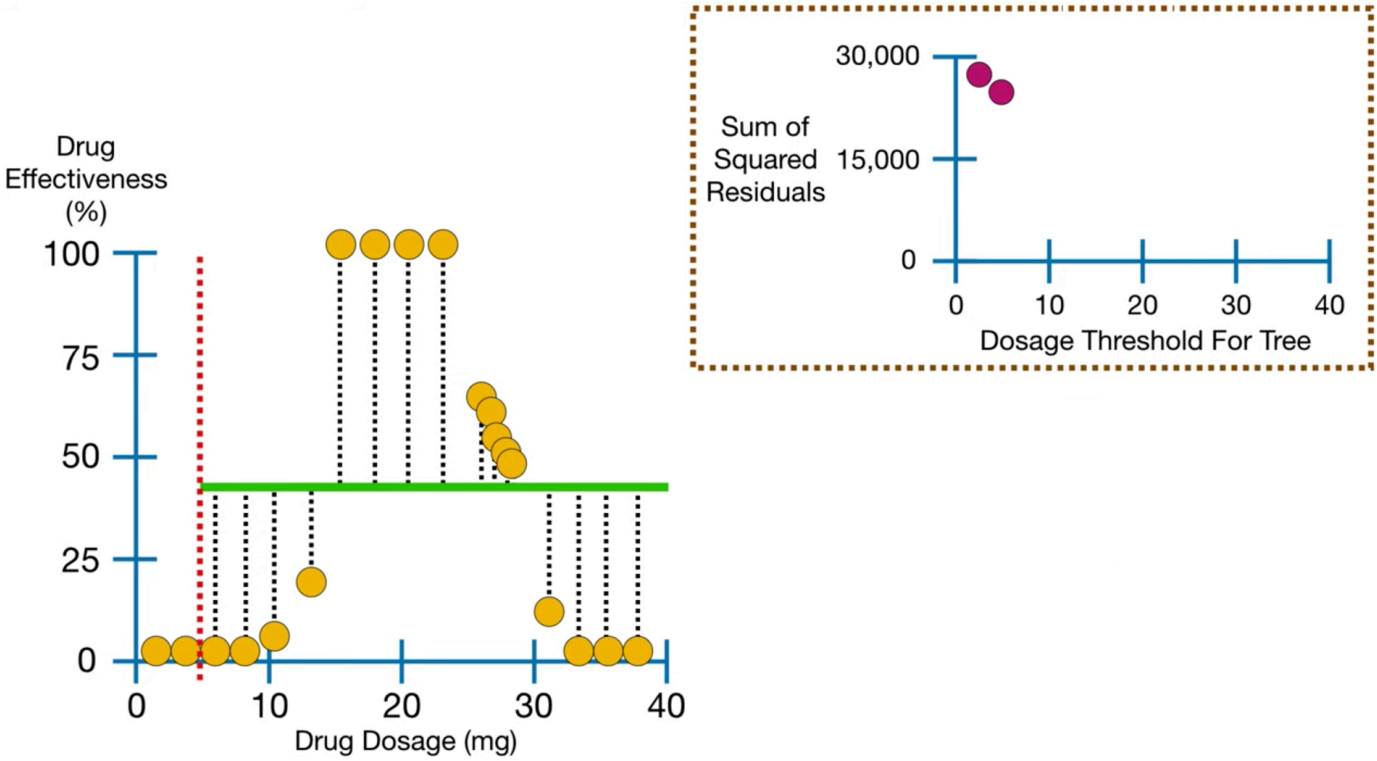

100
75
50
25
0
Drug
Effectiveness
(%)
0
10
20
30
40
Drug Dosage (mg)

- Select a threshold for drug dosage
- On both sides, compute the **sum of squared residuals** (SSR) between the average of each side and the ground truth
- Report the weighted average SSR of both sides, weighted by the number of data points ⤵

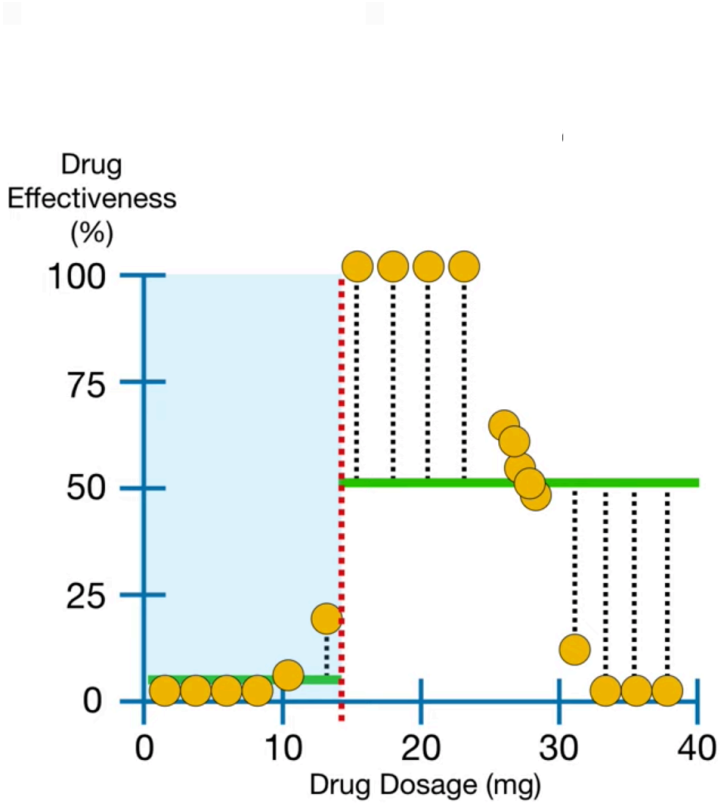


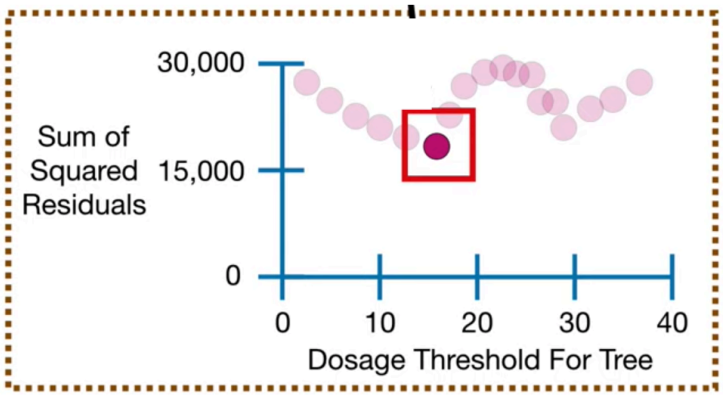


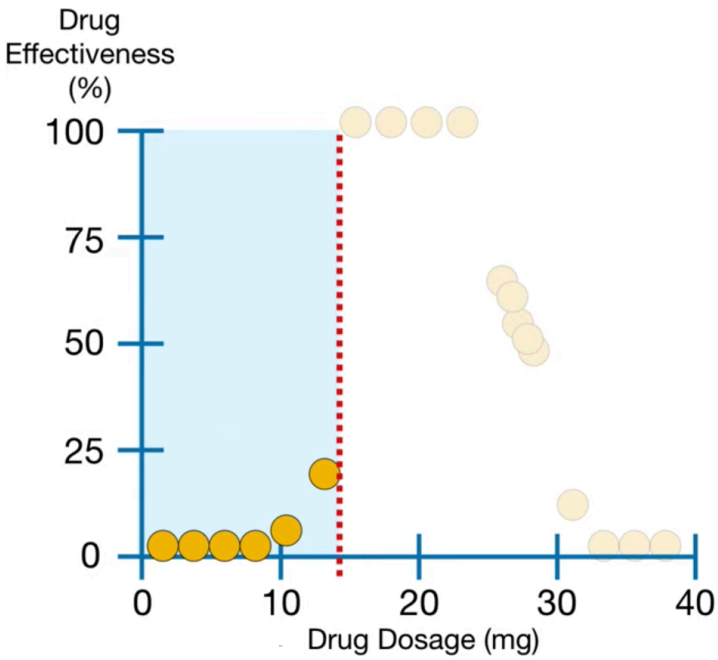
- Move the threshold to the right, and compute the weighted average SSR again ↘

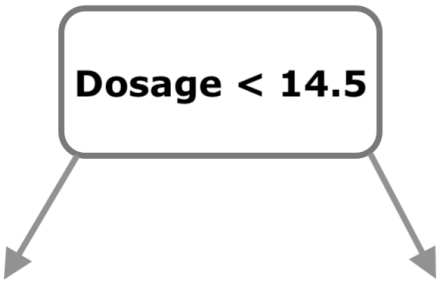


The threshold that minimizes the residuals becomes the **root node** of the tree





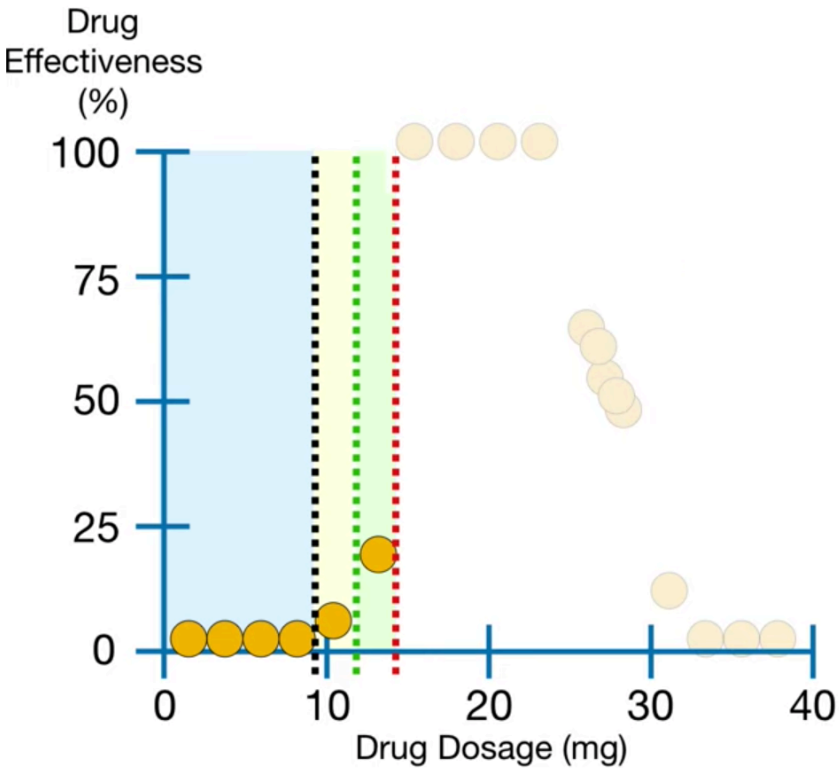




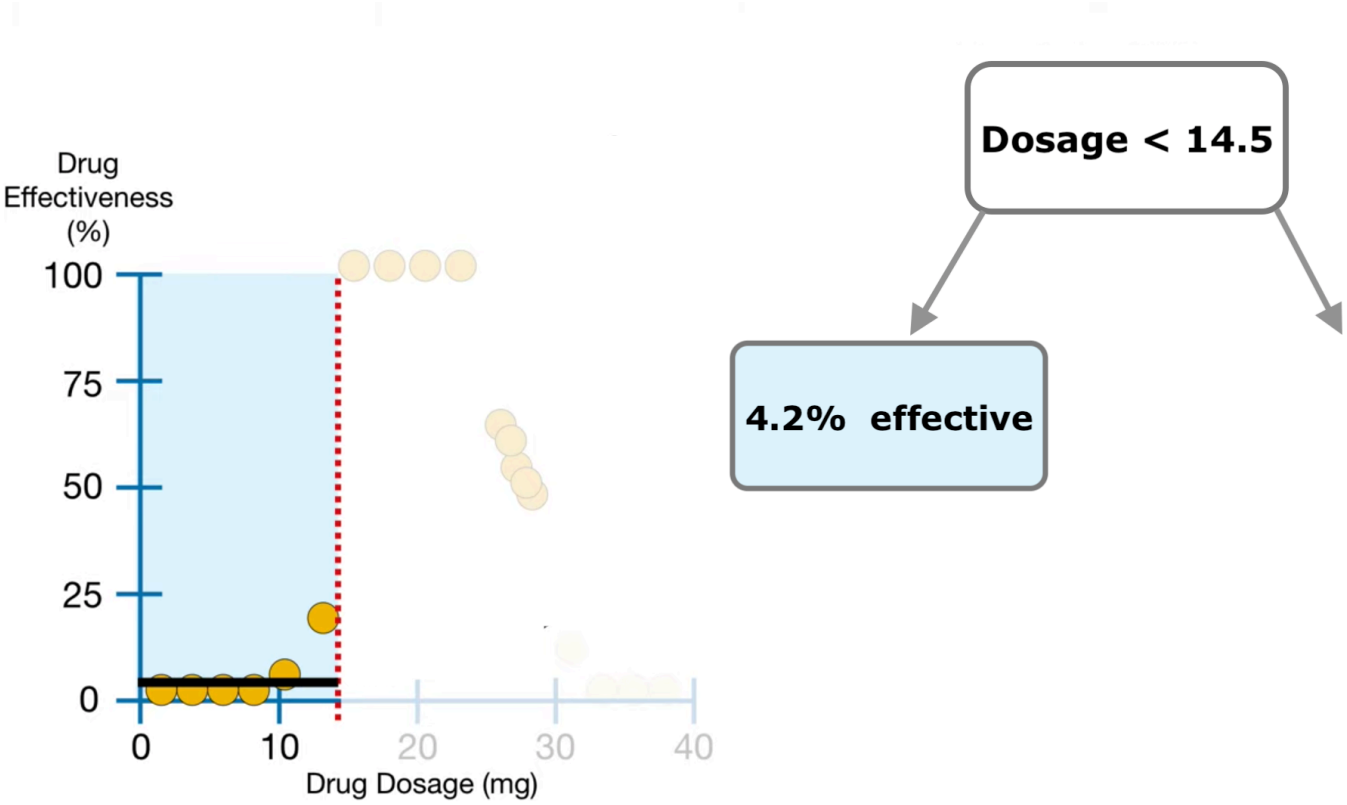
We could further split the points below the dosage of 14.5, but we probably shouldn't

Why?

⚠️ We would be **overfitting**!



Instead, stop splitting and use the average value of all points within the group to **generalize**

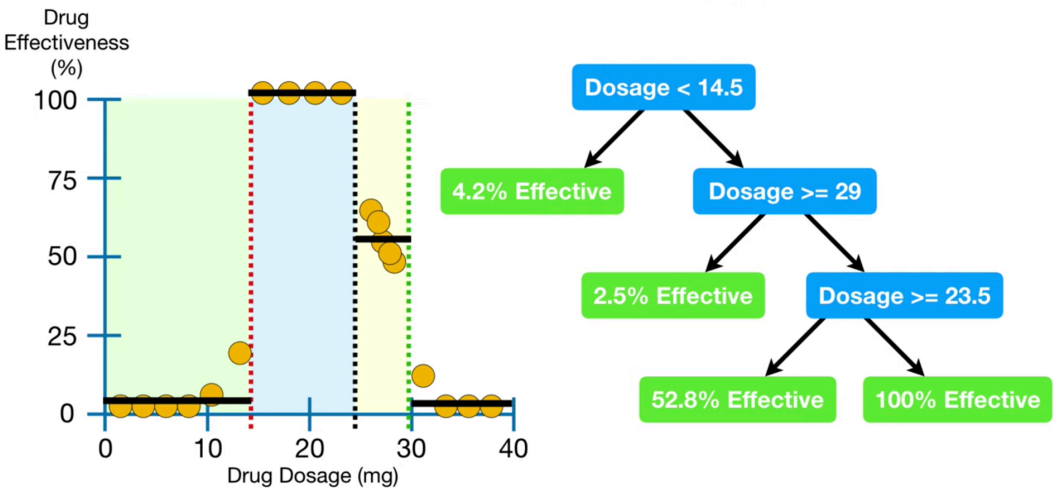


⚠️ Controlling Overfitting

- **Decision trees must be tuned!**
- Default parameters will almost certainly overfit
  - Control split with `min_samples_split`
  - Control leaves with `min_samples_leaf`
  - Control tree depth with `max_depth`

`min_samples_split`

- Specifies the minimum number of samples required to split an internal node
- In the example, it is set to `7`



`min_samples_leaf`

- Minimum number of samples for node to be a leaf (the smaller the number, the more it overfits)

`max_depth`

- Maximum depth of the tree (the larger the number, the more it overfits)

💻 Variance Illustrated

Regression

The `flats` dataset is available here (https://wagon-public-datasets.s3.amazonaws.com/flats.csv)

In [ ]:
```
import pandas as pd
data = pd.read_csv('https://wagon-public-datasets.s3.amazonaws.com/flats.csv')
data.head(3)
```

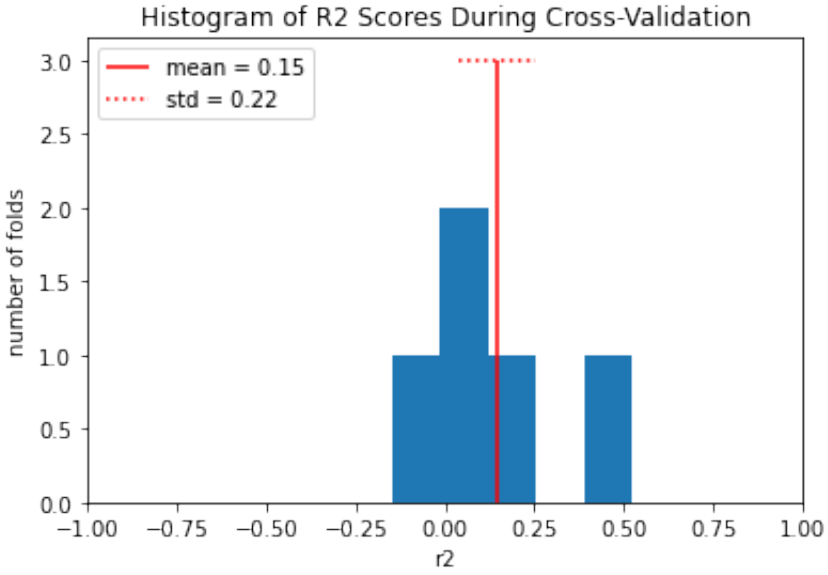
Out[ ]:

	price	bedrooms	surface	floors
0	274.0	3	1830	2.0
1	500.0	4	2120	1.0
2	320.0	3	1260	1.0

In [ ]:
```
X = data[['bedrooms','surface','floors']]
y = data['price']
```

In [ ]:
```
def plot_histogram_cv_results(cv_results):
    # Calculating the std and the mean
    std = cv_results['test_score'].std()
    mean = cv_results['test_score'].mean()

    # Getting the number of folds
    n_cv = len(cv_results['test_score'])

    # Building plot
    plt.hist(cv_results['test_score'], bins=n_cv)

    # Creating red lines
    plt.vlines(mean, 0, 3, color='red', label=f'mean = {mean:.2f}')
    plt.hlines(
        3, mean - 1/2 * std, mean + 1/2 * std,
        color='red', label=f'std = {std:.2f}', ls='dotted'
    )

    # Setting the title
    plt.title('Histogram of R2 Scores During Cross-Validation')

    # Setting the labels and xlim
    plt.xlim((-1, 1))

    plt.xlabel('r2')
    plt.ylabel('number of folds')

    # Showing the legend
    plt.legend(loc='upper left')

    plt.show()
```

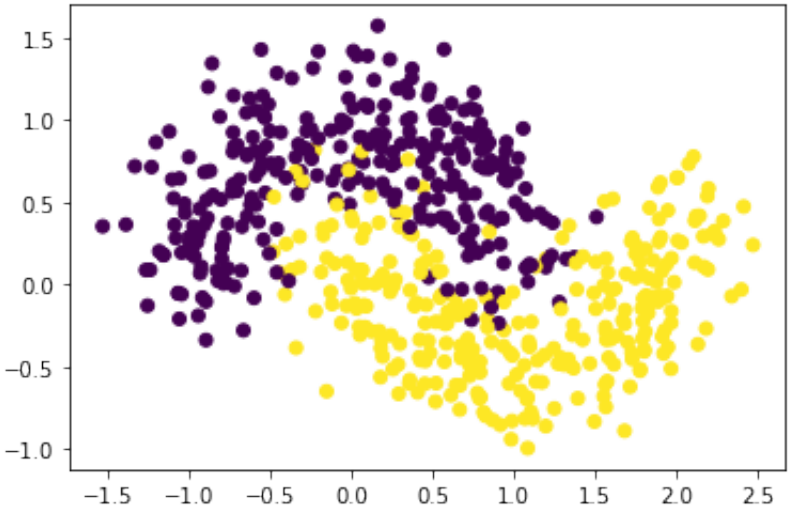
In [ ]:
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_validate

tree = DecisionTreeRegressor()

cv_results = cross_validate(tree, X, y, scoring = "r2", cv=5)

# Custom method
plot_histogram_cv_results(cv_results)
```



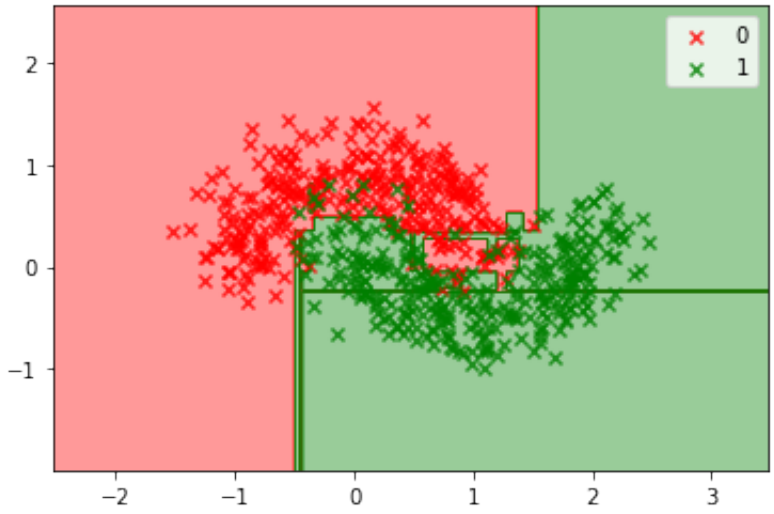
Classification

In [ ]:
```
from sklearn.datasets import make_moons

n=600
X_moon,y_moon = make_moons(n_samples=n, noise=0.25, random_state=0)

plt.scatter(X_moon[:,0], X_moon[:,1], c=y_moon)
```

Out[ ]: <matplotlib.collections.PathCollection at 0x7f5ce842ac10>



In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
from ipywidgets import interact

#@interact(max_depth=10)
def plot_classifier(max_depth):
    clf = DecisionTreeClassifier(max_depth=max_depth)
    clf.fit(X_moon, y_moon)
    plot_decision_regions(X_moon, y_moon, classifier=clf)

plot_classifier(max_depth=10)
```



Pros and Cons of Decision Trees

👍 Advantages

- No scaling necessary
- Resistant to outliers
- Intuitive and easy to interpret
- Allows for feature selection (see Gini-based `feature_importance_` (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeCla
- Non-linear modeling

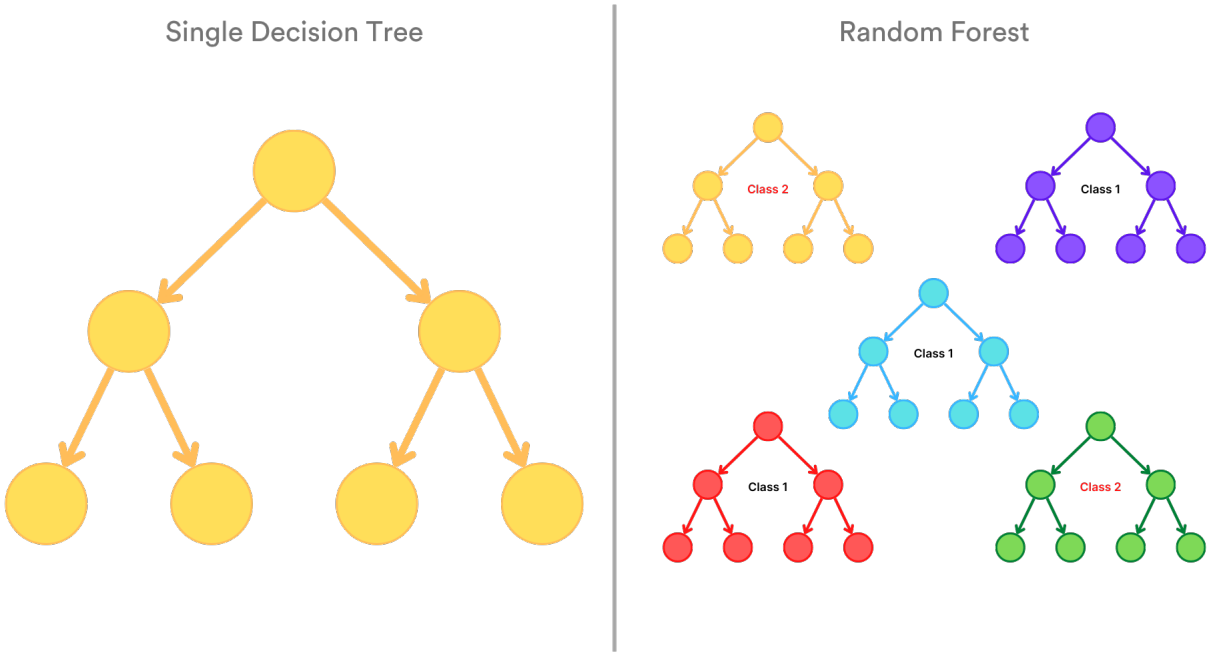
👎 Disadvantages

- High variance (i.e. a small change in the data causes a big change in the tree's structure)
- Long training time if grown up to (large) max depth, as $O(N_{obs} * m_{feat} * depth)$
- Splits data "orthogonally" to feature directions
  - 💡 use PCA upfront to "orient" data

2. Bagging (i.e Bootstrap Aggregating)

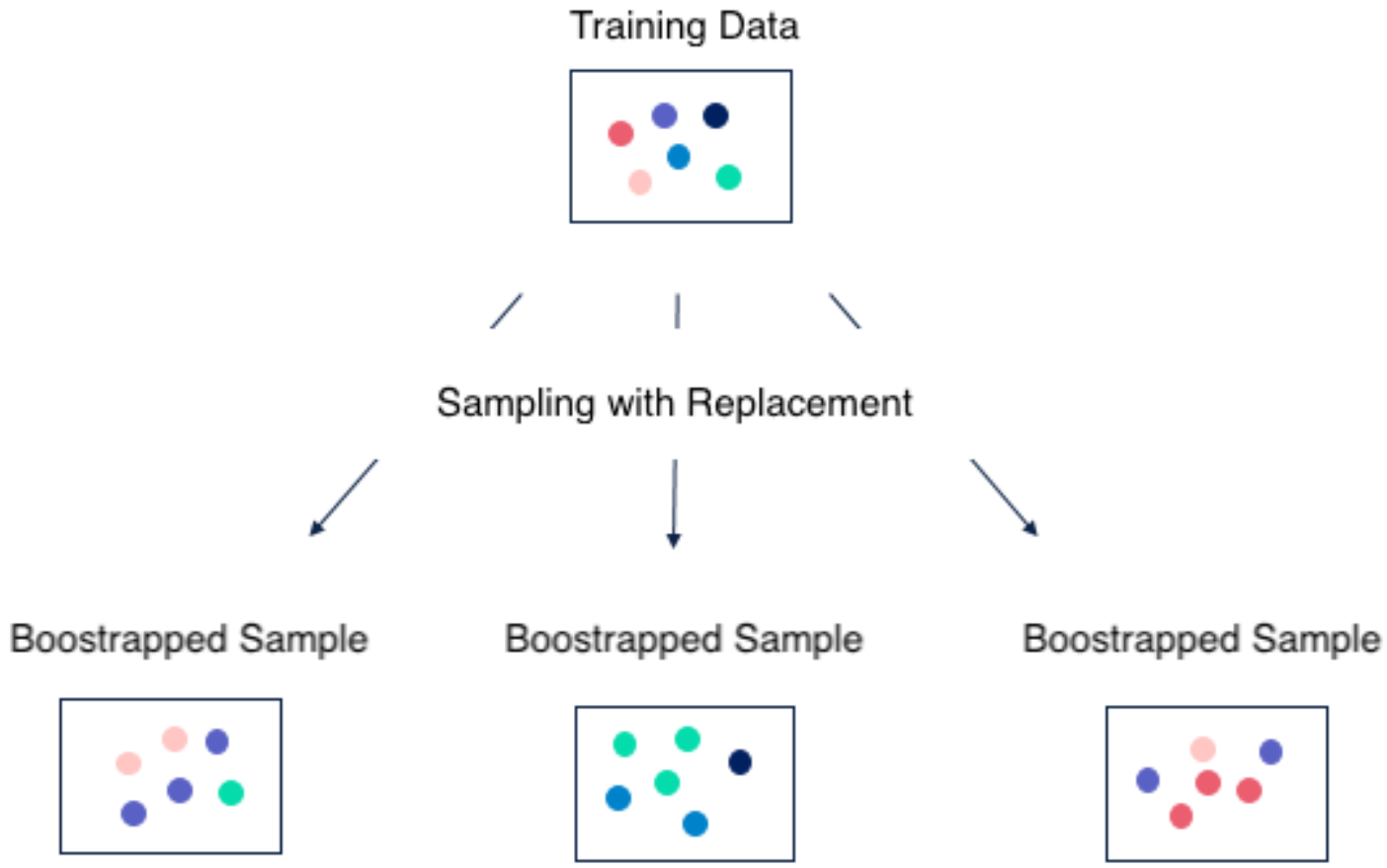
Bootstrap aggregating, also known as Bagging, is the aggregation of multiple versions of a model

- It is a **parallel** ensemble method
- The aim of bagging is to **reduce variance**
- Each version of the model is called a **weak learner**
- Weak learners are trained on **boostrapped** samples of the dataset



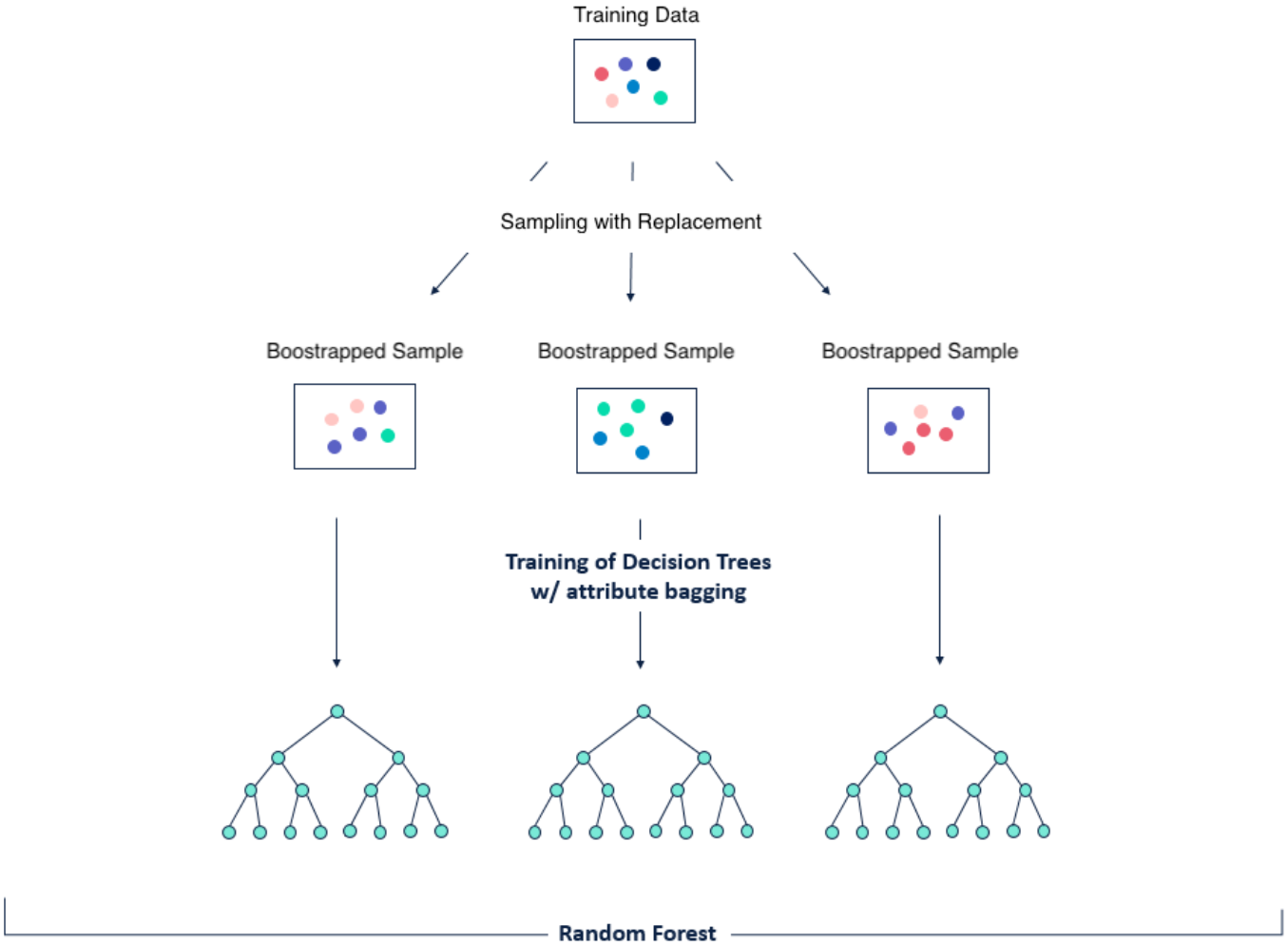
Bootstrapping (or Generating Bootstrapped Samples)

- The samples are created by randomly drawing data points, with replacement
- Features can also be randomly filtered to increase bagging diversity

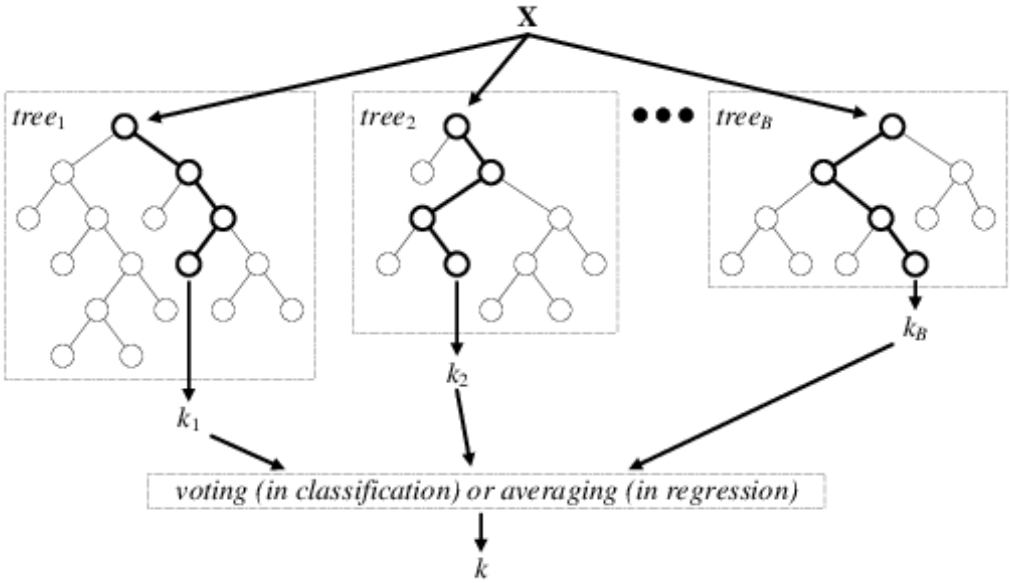


Random Forests = Bagged Trees

Random Forests are a bagged ensemble of Decision Trees

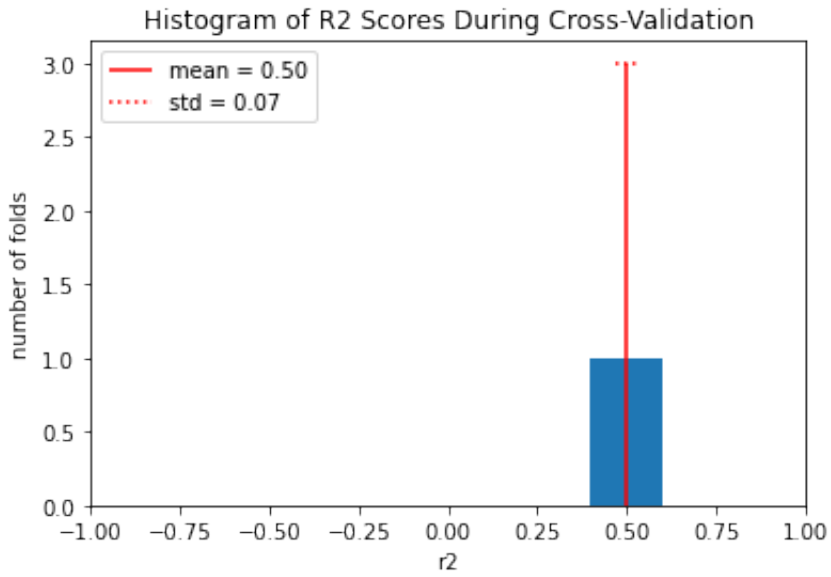


Predictions are **averaged** in regression tasks, and **voted** in classification tasks

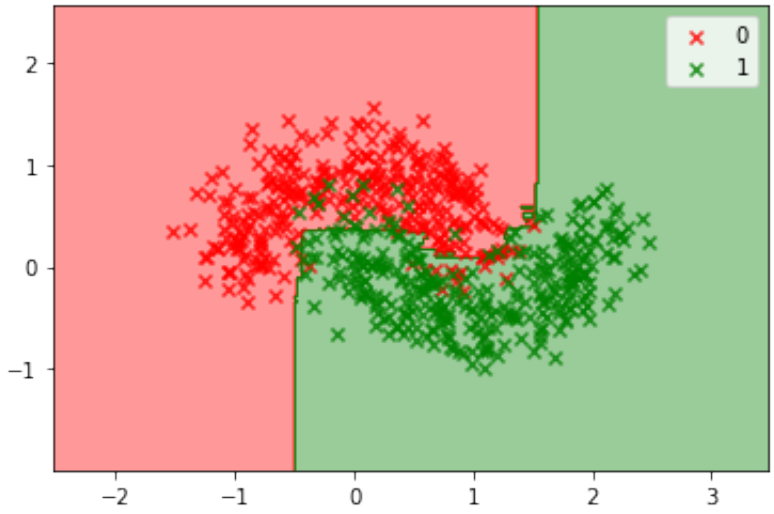


💻 **Sklearn [RandomForestRegressor (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor) and [Classifier (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.)**

In [ ]:
```
from sklearn.ensemble import RandomForestRegressor

forest = RandomForestRegressor(n_estimators=100)

cv_results = cross_validate(forest, X, y, scoring = "r2", cv=5)

plot_histogram_cv_results(cv_results)
```



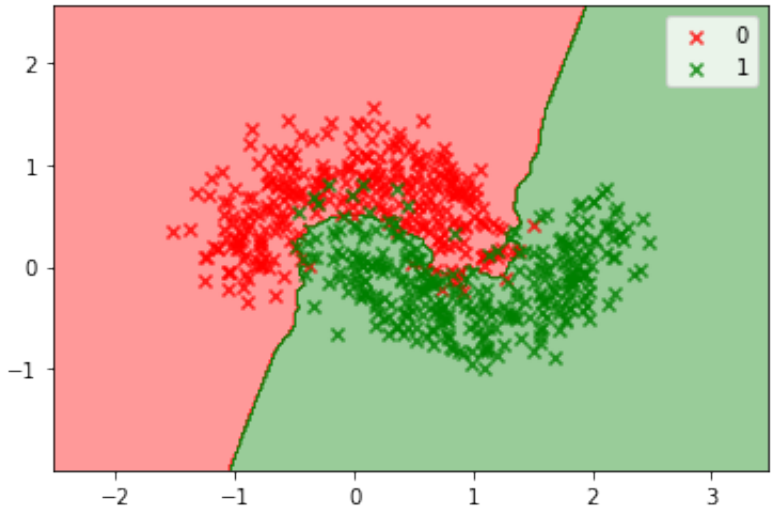
In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

#@interact(max_depth=5)
def plot_classifier(max_depth):
    cls = RandomForestClassifier(max_depth=max_depth)
    cls.fit(X_moon, y_moon)
    plot_decision_regions(X_moon, y_moon, classifier=cls)

plot_classifier(max_depth=5)
```



💻 Bagging any Algorithm (not just Trees)!

Bagging can be implemented on any algorithm using either the `BaggingRegressor` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingRegressor.html) or the `BaggingClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html)

In [ ]:
```
from sklearn.ensemble import BaggingClassifier, BaggingRegressor
from sklearn.neighbors import KNeighborsClassifier

weak_learner = KNeighborsClassifier(n_neighbors=3)
bagged_model = BaggingClassifier(weak_learner, n_estimators=40)

bagged_model.fit(X_moon, y_moon)
plot_decision_regions(X_moon, y_moon, classifier=bagged_model)
```



Out-of-Bag Samples

Sample not "drawn" by the bagging can be used to give a pseudo "test" score

In [ ]:
```
from sklearn.linear_model import LinearRegression

linear_model = LinearRegression()

bagged_model = BaggingRegressor(linear_model,
                                n_estimators=50,
                                oob_score=True)

bagged_model.fit(X, y).oob_score_
```

Out[ ]:
```
0.5072479697254035
```

💡 Notice

```
RandomForestClassifier(n_estimators=100)
BaggingClassifier(DecisionTreeClassifier(), n_estimators=100)  # similar (
but slightly less optimized)
```

```
BaggingRegressor(RandomForestClassifier(), n_estimators=100)  # bad time!
10,000 trees to train!
```

Pros and Cons of Bagging

👍 Advantages:

- Reduces variance (overfitting)
- Can be applied to any model

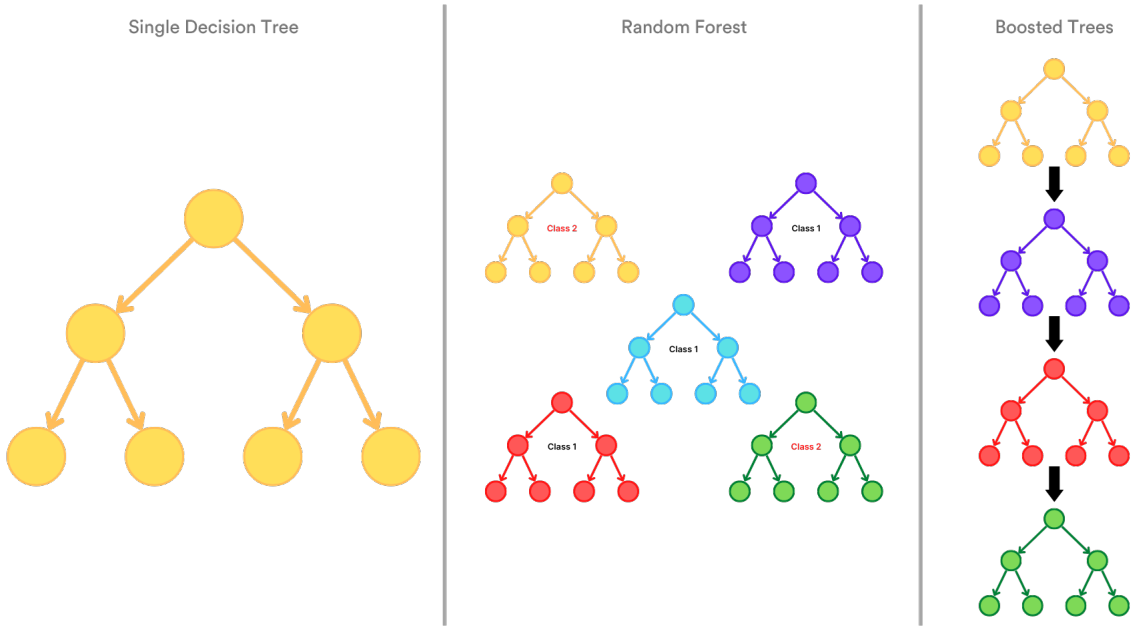
👎 Disadvantages

- Complex structure
- Long training time
- Disregards the performance of individual sub-models

3. Boosting

Boosting is a method designed to train weak learners that learn from their predecessor's mistakes

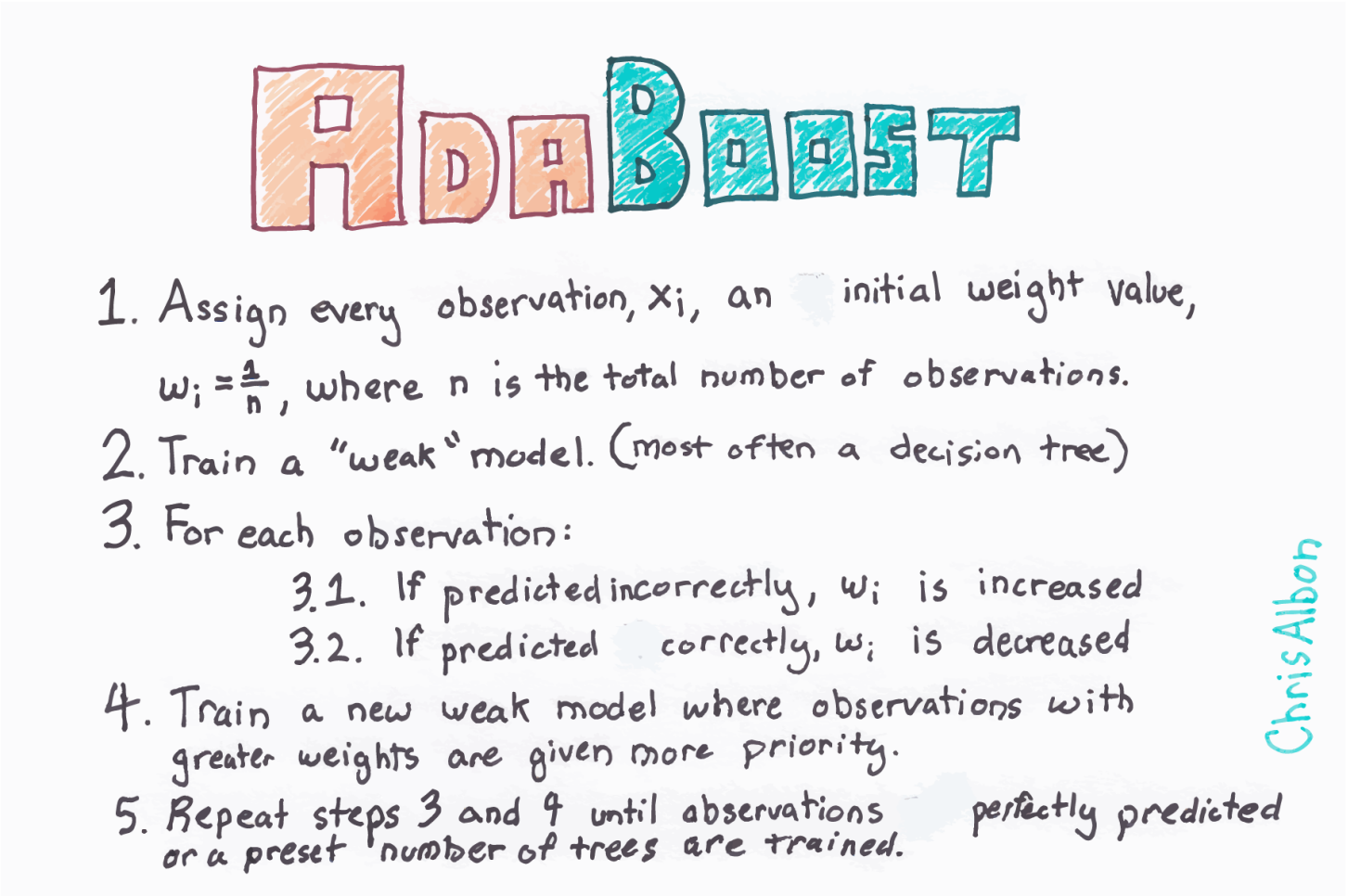
- It is a **sequential** ensemble method
- The aim of boosting is to **reduce bias**
- Focuses on the observations that are harder to predict
- The best weak learners are given more weight in the final vote



3.1 AdaBoost (Adaptative Boosting)

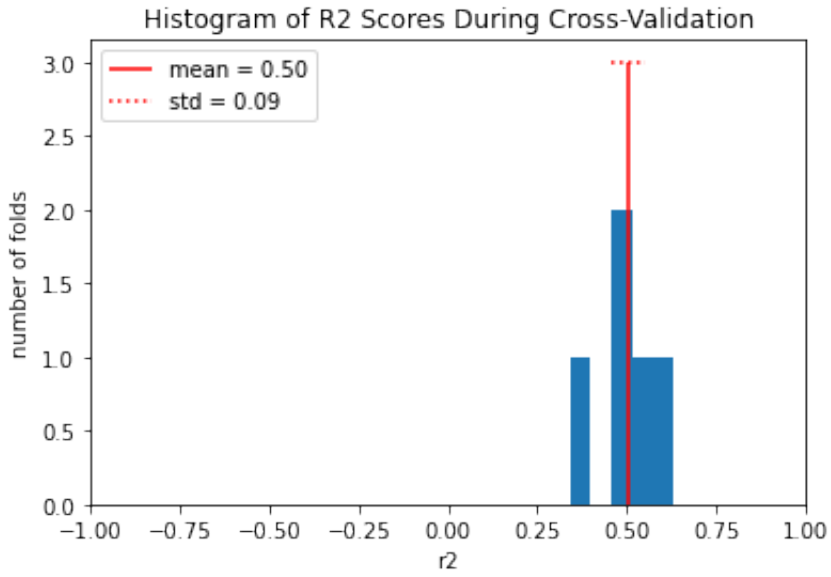
`AdaBoostRegressor` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html) and `AdaBoostClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html)

One implementation of boosting that works particularly well with trees

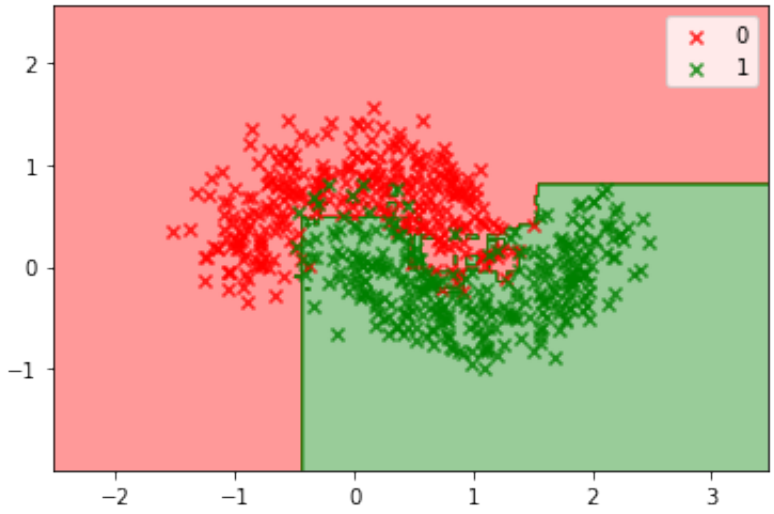


💻 AdaBoosted Trees in Sklearn

In [ ]:
```
from sklearn.ensemble import AdaBoostRegressor

adaboost = AdaBoostRegressor(
    DecisionTreeRegressor(max_depth=3),
    n_estimators=50)

cv_results = cross_validate(adaboost, X, y, scoring = "r2", cv=5)

plot_histogram_cv_results(cv_results)
```



In [ ]:
```
from sklearn.ensemble import AdaBoostClassifier

#@interact(n_estimators=[10, 30, 50,100], max_depth=3)
def plot_classifier(n_estimators, max_depth):
    model = AdaBoostClassifier(DecisionTreeClassifier(max_depth=max_de
pth),
                               n_estimators=n_estimators)
    model.fit(X_moon, y_moon)
    plot_decision_regions(X_moon, y_moon, classifier=model)

plot_classifier(n_estimators=50, max_depth=3)
```



3.2 Gradient Boosting 🔥

- Only implemented for trees
- Generally performs better than AdaBoost

Instead of updating the weights of observations that were misclassified, Gradient Boosting will

1. Recursively fit each weak learner
   $d_{\text{tree i}}$
   so as to predict the **residuals** of the previous one
2. **Add** all the predictions of all weak learners

$D(\mathbf{x}$

For classification, the principle is similar, but in the logit space (if `log-loss` is chosen as a loss function)

📚 scikit-learn user guide (https://scikit-learn.org/stable/modules/ensemble.html#gradient-boosting)
📚 Brilliantly Wrong visual blog post (https://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html)

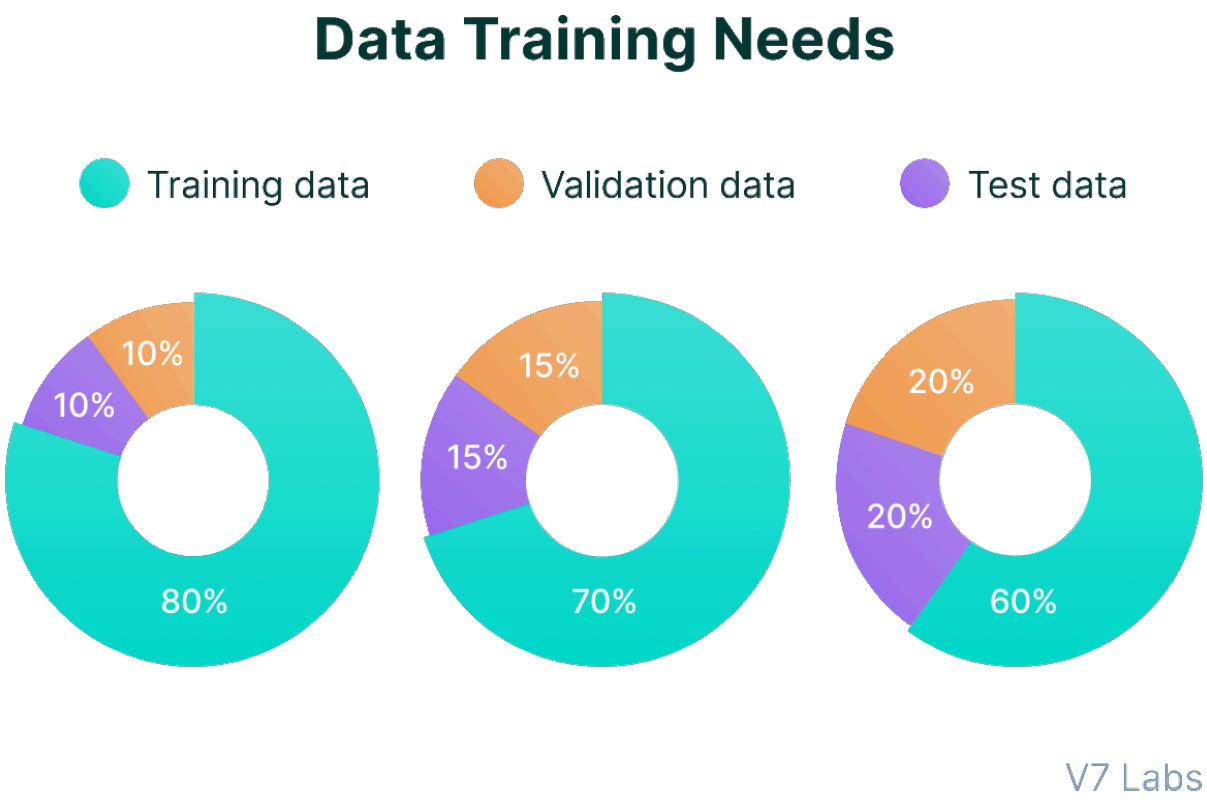
a) scikit-learn GradientBoostingRegressor (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegres and GradientBoostingClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifi

```
from sklearn.ensemble import GradientBoostingRegressor
model = GradientBoostingRegressor(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3
)
```

b) XGBOOST (https://xgboost.readthedocs.io/en/latest/) (Extreme Gradient Tree Boosting)

- Dedicated library, optimized for this task
- Nice features inspired by Deep Learning

First, let's split the dataset into train, test and validation sets



Splitting the Data

```
from sklearn.model_selection import train_test_split

# Split data into train, test and validation sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.3, random_state = 42  # TEST = 30%
)

# Use the same function above for the validation set
X_test, X_val, y_test, y_val = train_test_split(
    X_test, y_test, test_size = 0.5, random_state = 42  # TEST = 15%
)
```

Then, we plug the data into the `XGBRegressor` !

```
from xgboost import XGBRegressor

xgb_reg = XGBRegressor(max_depth=10, n_estimators=100, learning_rate=0.1)

xgb_reg.fit(X_train, y_train,
    # evaluate loss at each iteration
    eval_set=[(X_train, y_train), (X_val, y_val)],
    # stop iterating when eval loss increases 5 times in a row
    early_stopping_rounds=5
)

y_pred = xgb_reg.predict(X_val)
```

Or integrate it into `scikit-learn` as a Pipeline 🎉

```
from sklearn.pipeline import make_pipeline

pipe_xgb = make_pipeline(xgb_reg)
cv_results = cross_validate(pipe_xgb, X, y, cv=10, scoring='r2')
```

Pros and Cons of Boosting

👍 Advantages:

- Strong sub-models have more influence in final decision
- Reduces bias

👎 Disadvantages:

- Computationally expensive (sequential)
- Easily overfits
- Sensitive to outliers (too much time spent trying to correctly predict them)

Ensemble Methods Summary

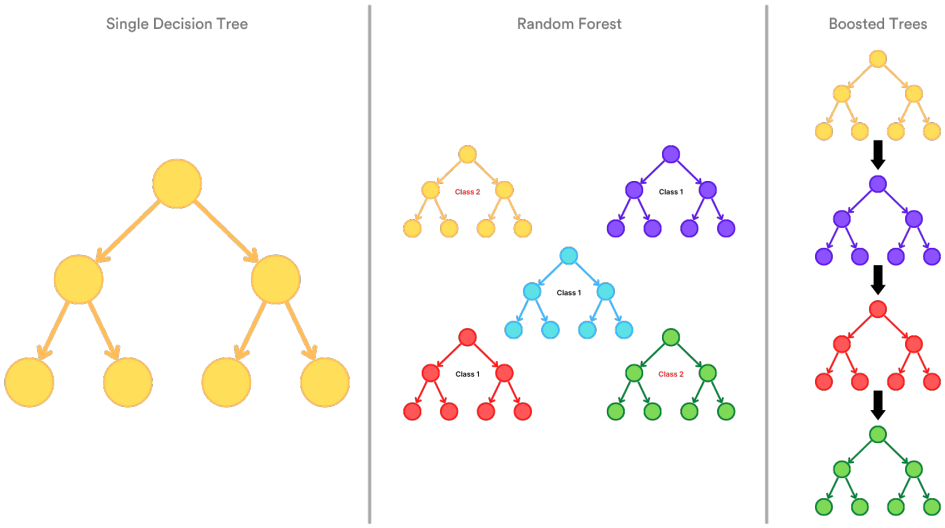
Ensemble learning combines several base algorithms, like Decision Trees
Ensemble methods can be broken down into two categories:

Parallel learners: models trained in parallel; predictions are aggregated

- `RandomForestRegressors`
- `BaggingRegressors`

Sequential learners: models trained sequentially so as to learn from predecessors' mistakes

- `AdaBoostRegressor`
- `GradientBoostRegressor`
- `XGBoostRegressor`



4. Stacking

Training **different estimators** and aggregating their predictions

- Different estimators (KNN, LogReg, etc.) capture **different structures in the data**
- Combining sometimes enhances the predictive power
- The results are aggregated either by voting (classification) or averaging (regression)

a) Simple aggregation

scikit-learn VotingClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html) and VotingRegressor (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingRegressor.html)

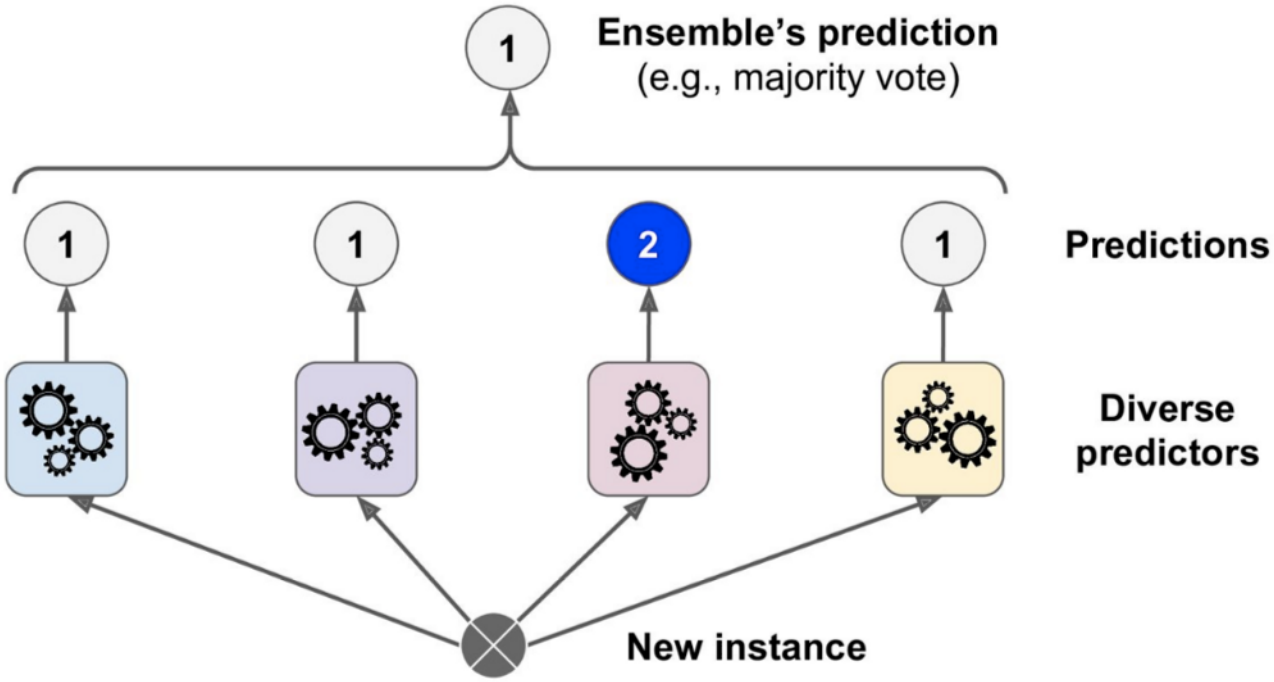
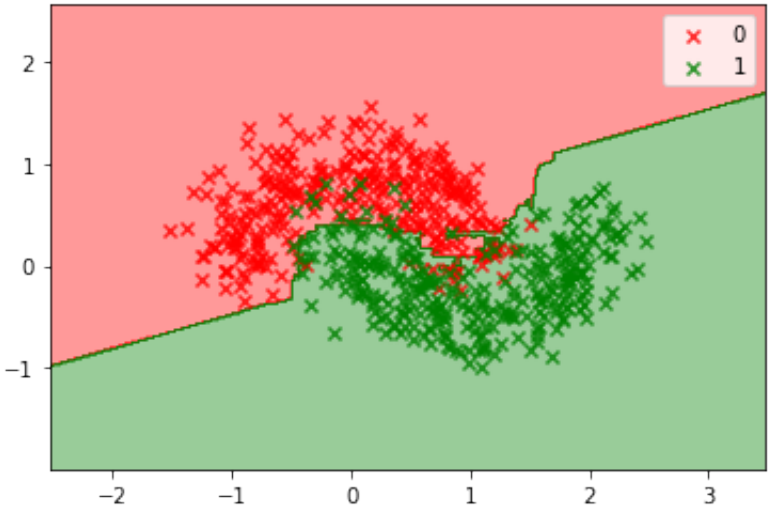


Figure 7-2. Hard voting classifier predictions

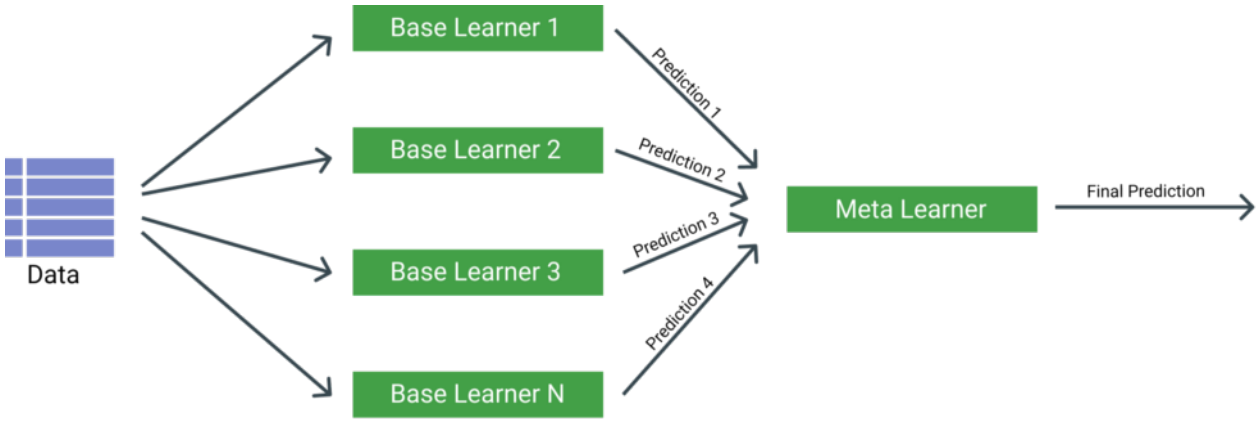
In [ ]:
```
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression

forest = RandomForestClassifier()
logreg = LogisticRegression()

ensemble = VotingClassifier(
    estimators = [("rf", forest),("lr", logreg)],
    voting = 'soft', # to use predict_proba of each classifier before
voting
    weights = [1,1] # to equally weight forest and logreg in the vote
)
ensemble.fit(X_moon, y_moon)
plot_decision_regions(X_moon, y_moon, classifier=ensemble)
```



🤔 How do you choose the best weights?

b) Multi-layer Stacking!

Train a **final estimator** on the predictions of the previous ones



scikit-learn StackingClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html) and StackingRegressor (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingRegressor.html)