Regression

Goal

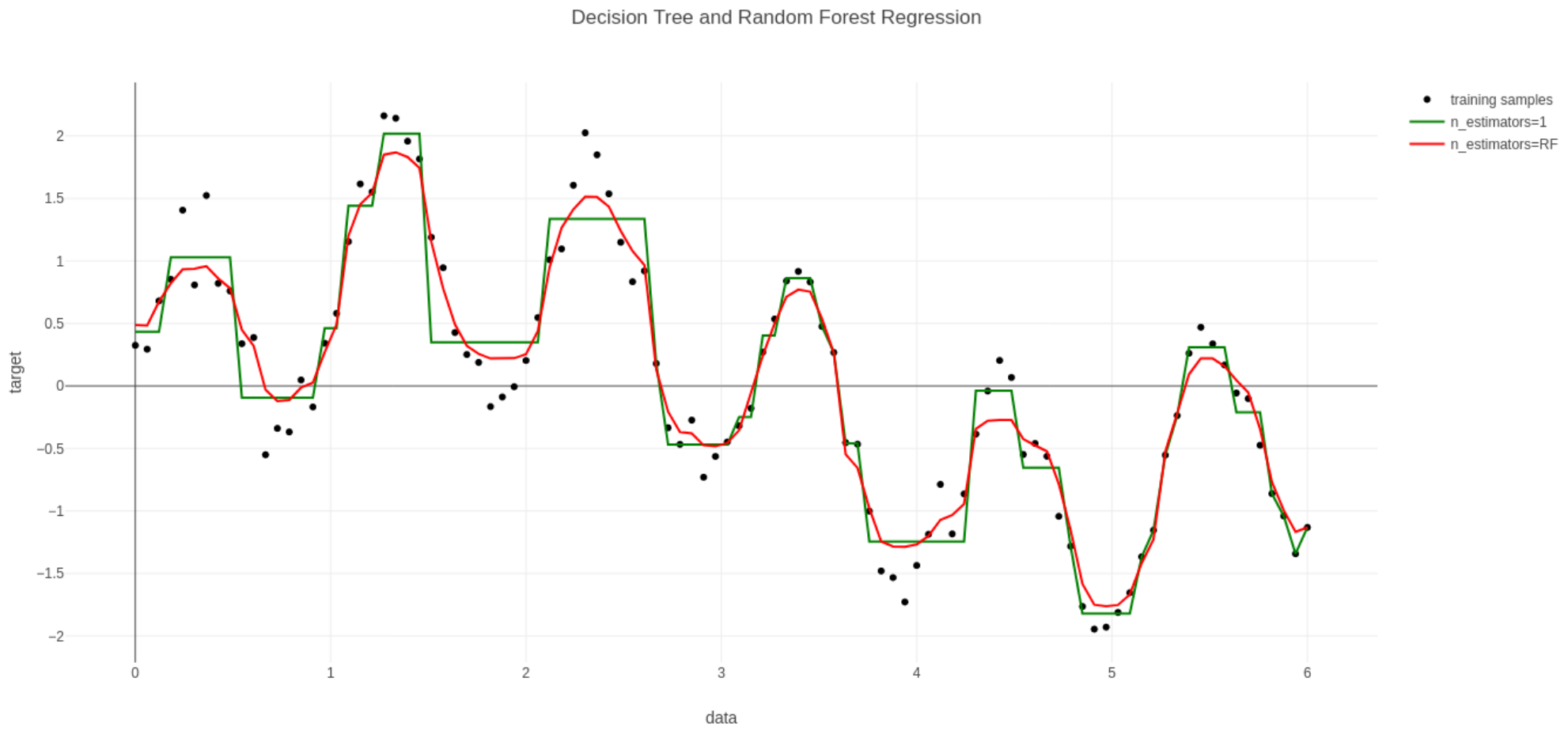
What do we have now

- classifier completed
- design open to new impurity measures

Next step is to do random forest *regression* :

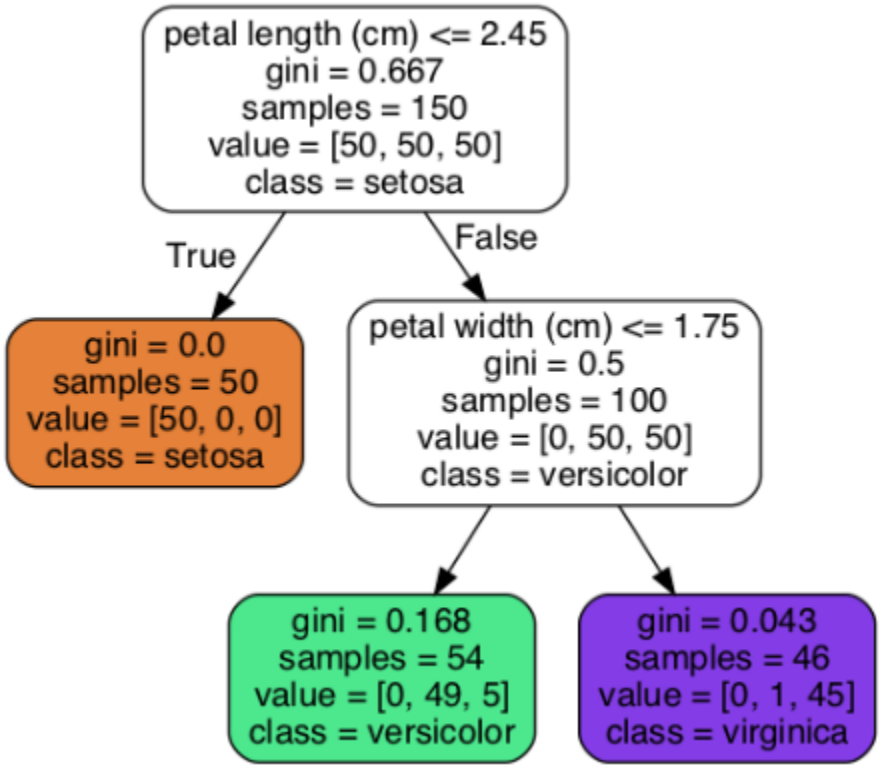
One of the nice properties of random forests is that they also are well suited to regression and we only need to perform a few small changes to convert our classifier into a regressor. This implies to identify what a random forest classifier and a regressor have in common and move it somewhere to avoid, of course, redundancy (the "don’t repeat yourself" principle).

Regression 1d

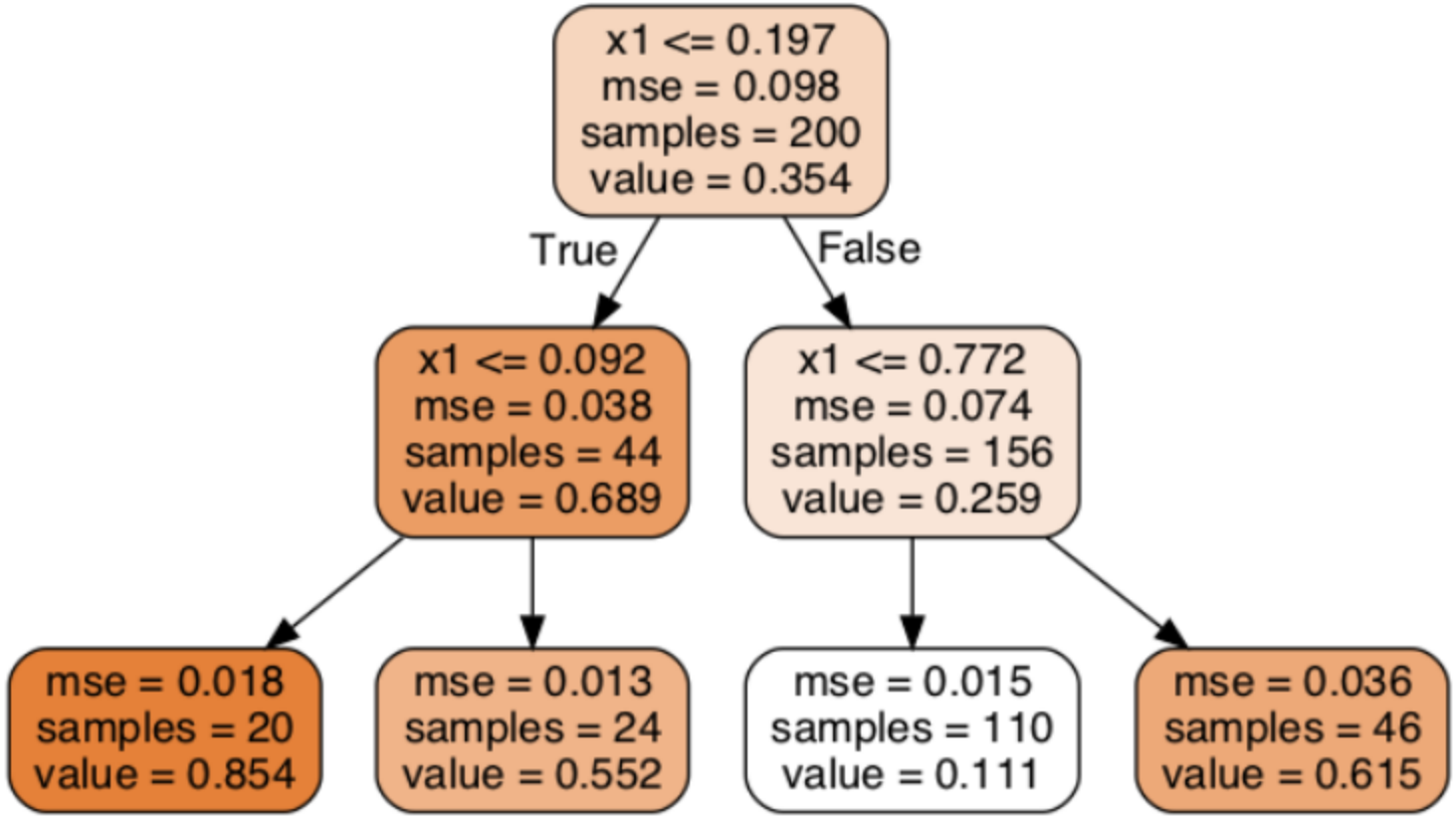


But decision trees and random forests not constrained by number of dimensions.

Decision tree for classification

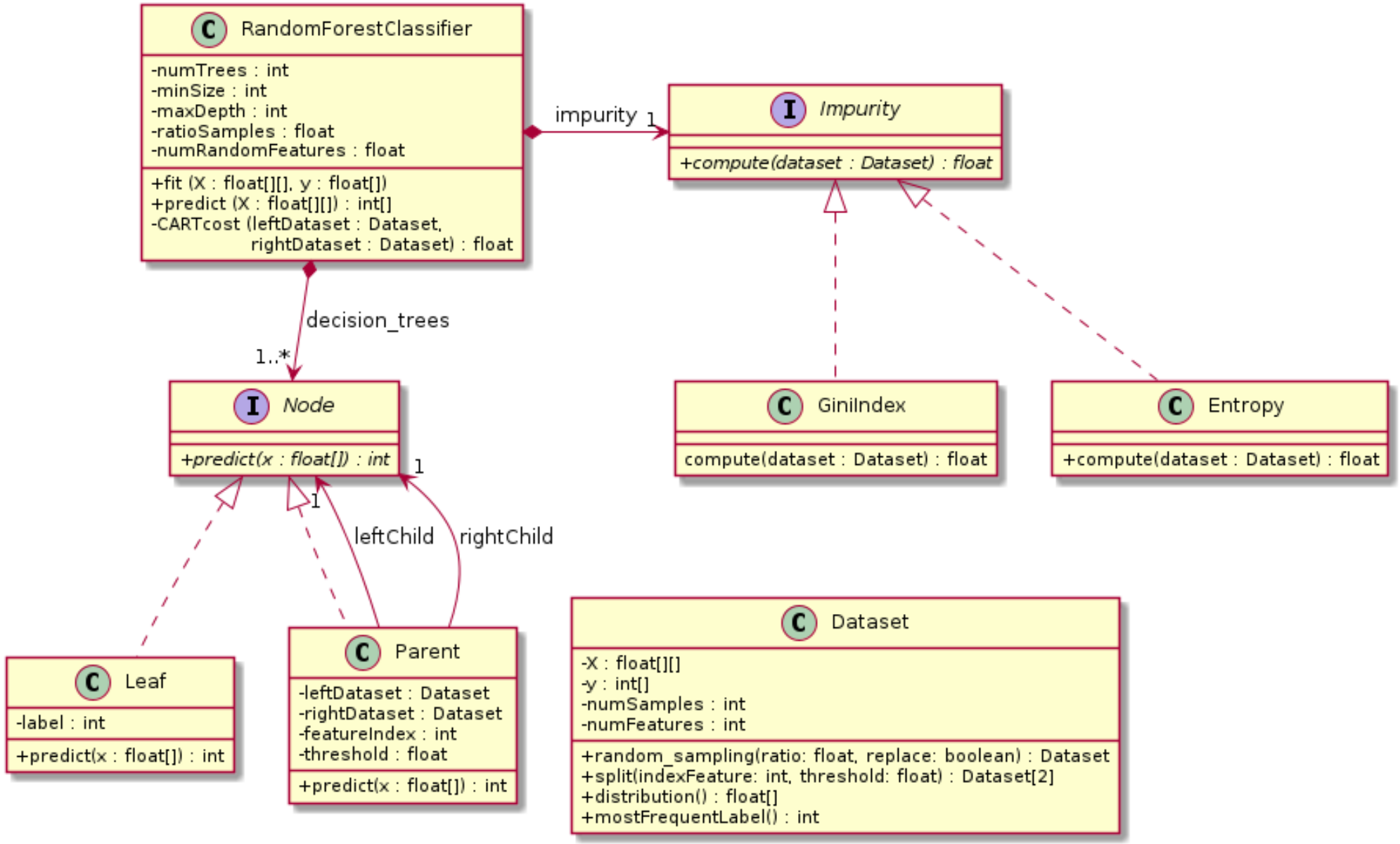


Decision tree for regression



same kind of rules, different impurity measure and value

Design for classification



What's different in regression

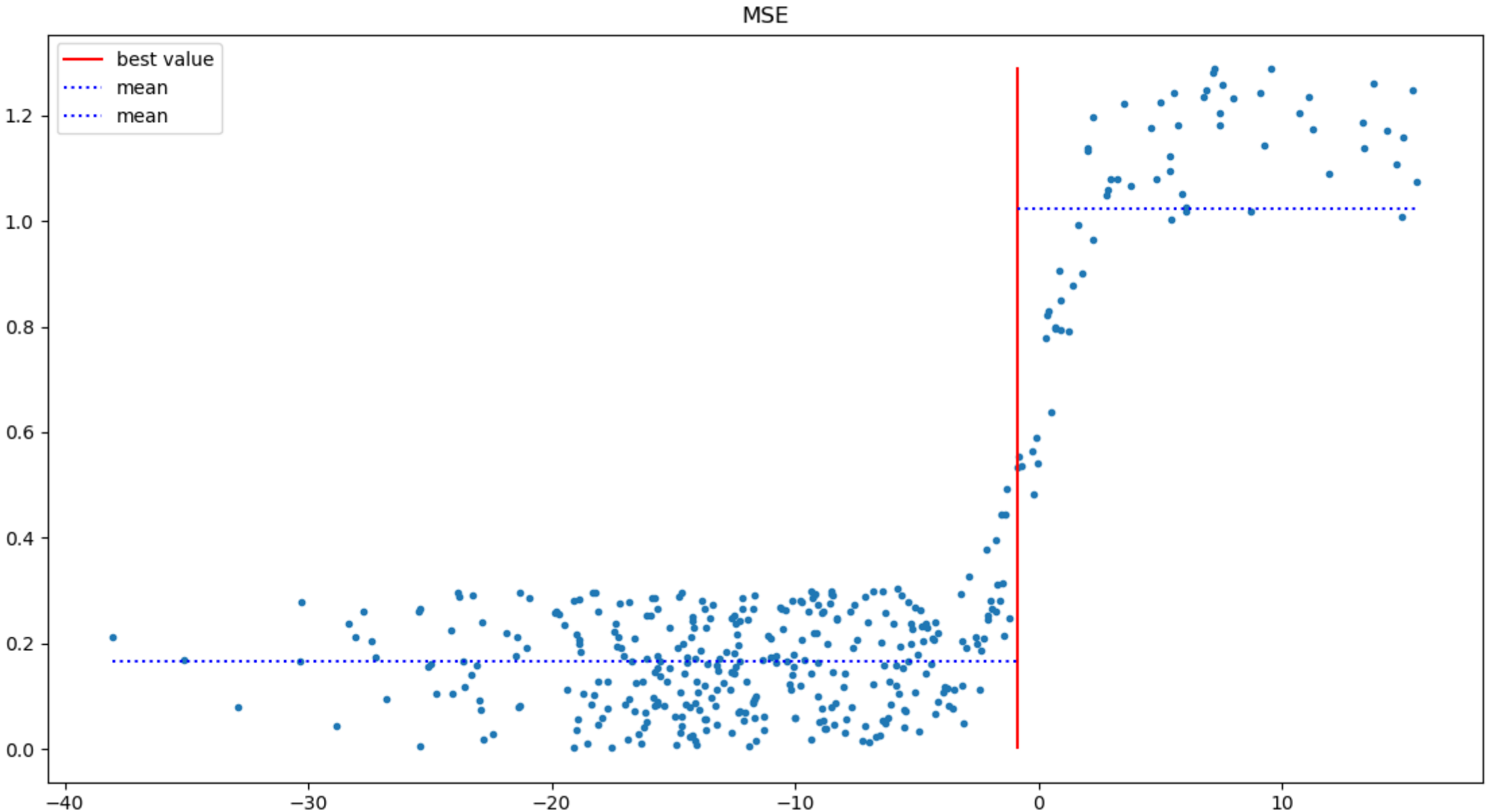
1. Gini, Entropy $\rightarrow$ SSE = sum of square errors
2. `predict()` of class `Leaf` most frequent class in $y \rightarrow$ mean y
3. majority voting of trees predictions $\rightarrow$ average

1. Sum of square errors

```
best_k = None, best_v = Inf, min_cost = Inf
for all features k:
    for all different v in X[:,k]:
        cost = J(k,v) # CART_cost()
        if cost < min_cost:
            best_k = k, best_v = v, min_cost = cost
:
:
idx_left = X[:,k] < v # array of booleans
y_left = y[idx_left] # select coordinates where True
y_right = y[np.logical_not(idx_left)]
```

$$J(k,v) = \frac{m_{\text{left}}}{m}\text{SSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{SSE}_{\text{right}}$$

$$\text{SSE}_{\text{left}} = \sum_{i \in \text{left}} (y_i - \bar{y}_{\text{left}})^2 \quad , \quad \bar{y}_{\text{left}} = \frac{1}{m_{\text{left}}} \sum_{i \in \text{left}} y_i$$


MSE
best value
mean
mean
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−40
−30
−20
−10
0
10

RF classifier : `Gini` , `Entropy`

```
rf = RandomForestClassifier(..., GiniIndex(), ...)

class RandomForestClassifier():
    def __init__(..., impurity_measure, ):
        :
        self.impurity_measure = impurity_measure
    :
    def _CART_cost(...)
        :
        score = self.impurity_measure.compute(dataset)
```

RF regressor : `SumSquareError`

```
class Impurity(ABC):
    @abstractmethod
    def compute(self, dataset):
        pass

class GiniIndex(Impurity):
    def compute(self, dataset):
        probs = dataset.distribution()
        return 1.0 - np.sum(probs ** 2)

class SumSquareError(Impurity):
    #TODO
```

And now

```
rf = RandomForestRegressor(..., SumSquareError(), ...)
```

Thanks to the *Strategy* design pattern

If you have done like

```
rf = RandomForestClassifier(..., 'gini', ...)

class RandomForestClassifier():
    def __init__(..., name_impurity, ):
        :
        if name_impurity=='gini':
            self.impurity_measure = Gini()
        elif: ... # other impurities for regression
```

then you can introduce a `_make_impurity` method:

```
class RandomForestClassifier():
    def __init__(..., name_impurity, ...):
        :
        self.impurity_measure = self._make_impurity(name_impurity)

    def _make_impurity(self, name):
        if name=='gini':
            return Gini()
        elif ... # other impurities for classification
```

```
class RandomForestRegressor():
    def __init__(..., name_impurity, ...):
        :
        self.impurity_measure = self._make_impurity(name_impurity)

    def _make_impurity(self, name):
        if name=='sum square error':
            return SumSquareError()
        elif ... # other impurities for regression
```

In any case, do not repeat code in the RF classifier and RF regressor *constructors*, **make an abstract class with a common constructor and use `super`**

2. majority voting $\rightarrow$ average of predicted values

```
class RandomForestClassifier():

    def predict(self, X):
        y_pred = []
        for x in X:
            predictions = [tree.predict(x) \
                           for tree in self.decision_trees]
            y_pred.append(self._combine_predictions(predictions))
        return np.array(y_pred)

    def _combine_predictions(self, predictions):
        return np.argmax(np.bincounts(predictions))
        # most frequent class = majority voting
        # also max(set(predictions), key=predictions.count)
```

```
class RandomForestRegressor():

    def predict(self, X):
        y_pred = []
        for x in X:
            predictions = [tree.predict(x) \
                           for tree in self.decision_trees]
            y_pred.append(self._combine_predictions(predictions))
        return np.array(y_pred)

    def _combine_predictions(self, predictions):
        return np.mean(predictions)
        # average of predicted values for an x
```

`predict` is the same, but of course *Don't repeat yourself* : move `predict` to an abstract super class

3. `predict()` of class `Leaf`

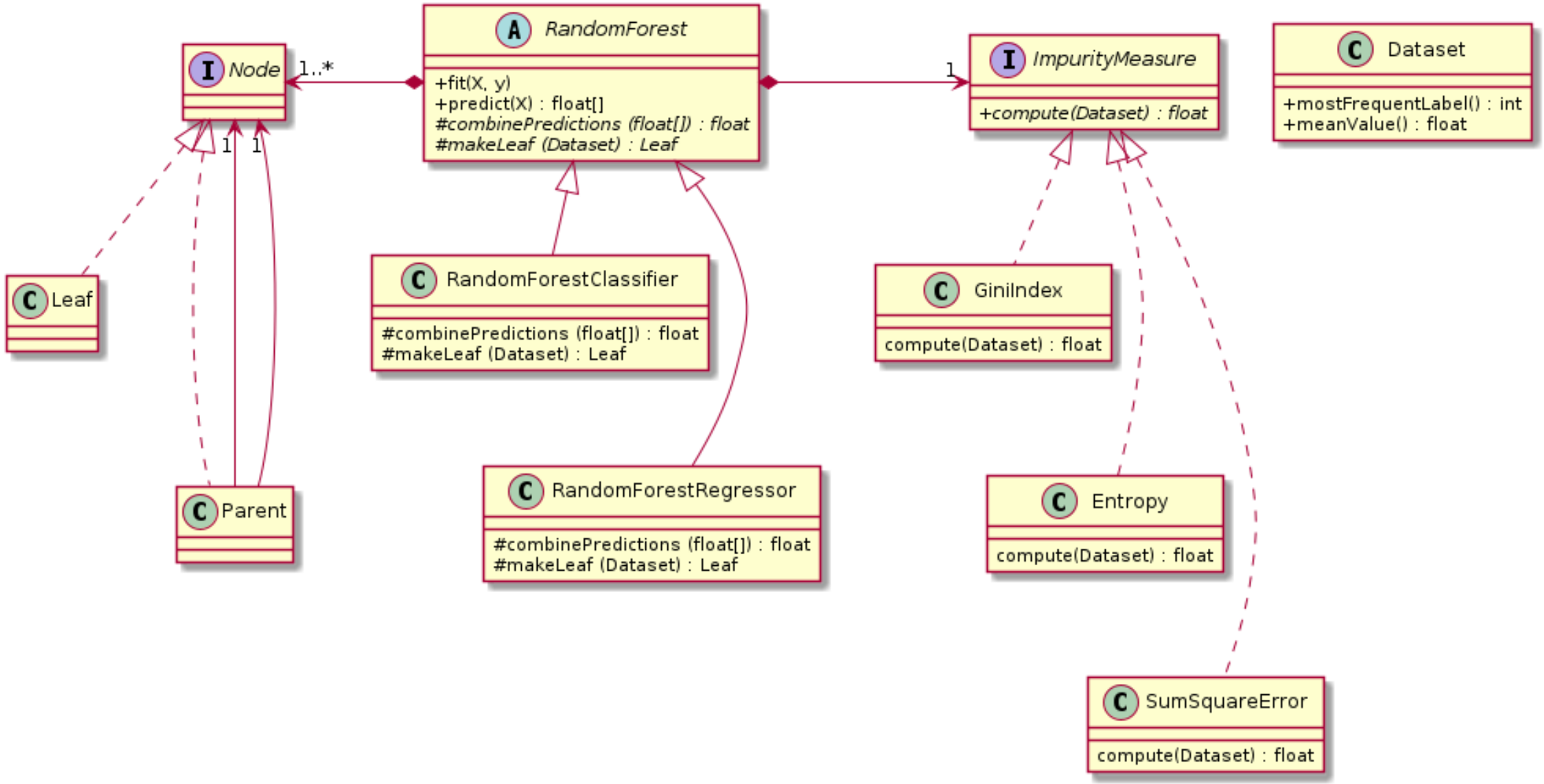
For a RF classifier, the value in a leaf is the most frequent class in the dataset y input to its constructor :

```
class RandomForestClassifier():

    def _make_leaf(self, dataset):
        return Leaf(dataset.most_frequent_label())
        # which is np.argmax(np.bincounts(y))

    def _make_node(...)
        :
        node = self._make_leaf(dataset)
```

For a RF regressor instead, it's the mean of the y values

```
class RandomForestRegressor():

    def _make_leaf(self, dataset):
        return Leaf(dataset.mean_value())

    def _make_node(...)
        :
        node = self._make_leaf(dataset)
```

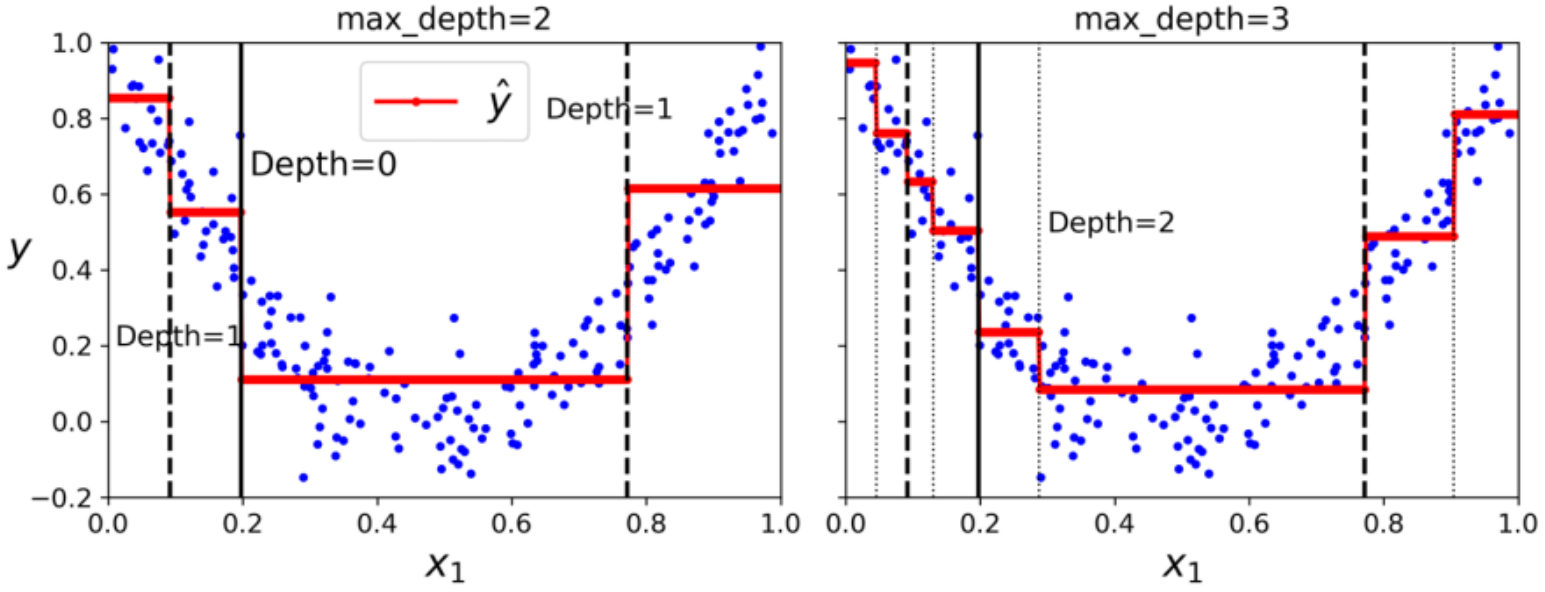
Again, `_make_leaf` is different but `_make_node` is the same, *do not repeat yourself*


Node
1..*
RandomForest
+fit(X, y)
+predict(X) : float[]
#combinePredictions (float[]) : float
#makeLeaf (Dataset) : Leaf
1
ImpurityMeasure
+compute(Dataset) : float
Dataset
+mostFrequentLabel() : int
+meanValue() : float
Leaf
1
1
RandomForestClassifier
#combinePredictions (float[]) : float
#makeLeaf (Dataset) : Leaf
GiniIndex
compute(Dataset) : float
Parent
RandomForestRegressor
#combinePredictions (float[]) : float
#makeLeaf (Dataset) : Leaf
Entropy
compute(Dataset) : float
SumSquareError
compute(Dataset) : float

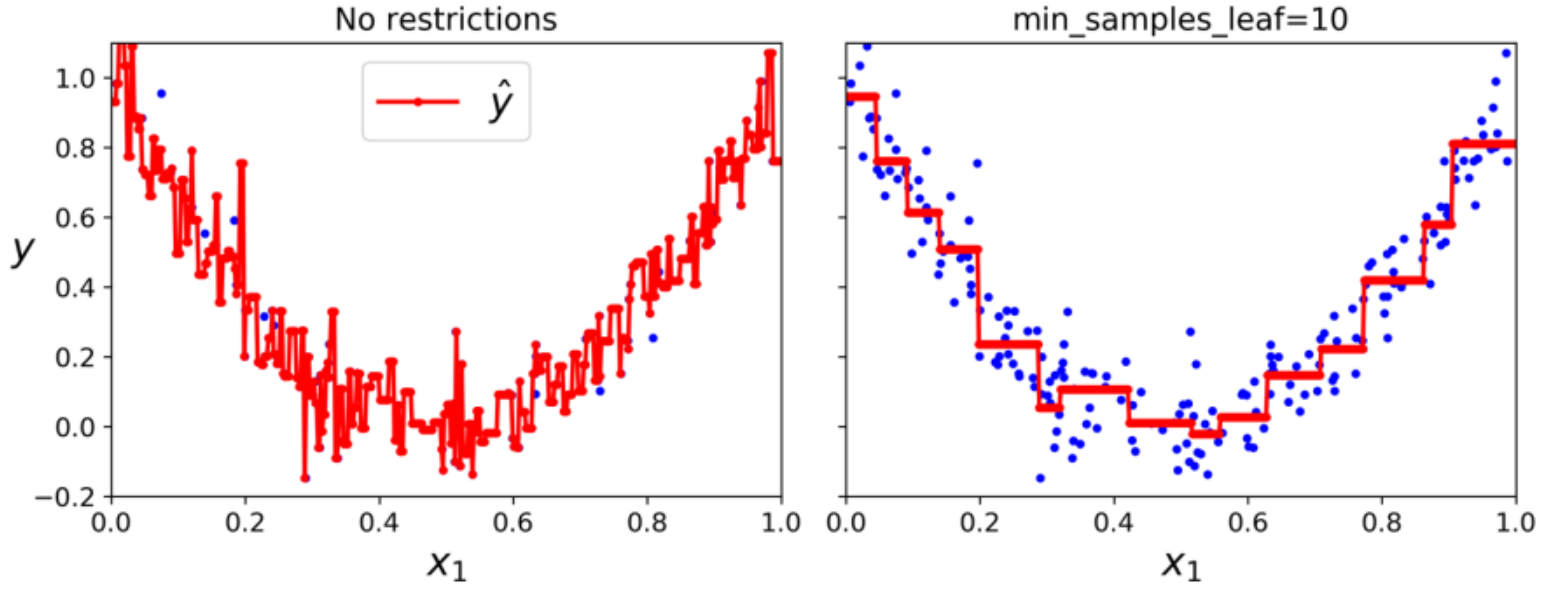
Note:

- `RandomForest` *has* a constructor: what is common in constructors of `RandomForestClassifier` and `RandomForestRegressor`
- maybe also a third method `# makeImpurity()` if impurity passed as string, not object
- *composite* + *strategy* + another design pattern we've studied, which one ?
- in `RandomForest` `predict(X) : float[]` for classification also

Regularization by hyperparameters : still the same



`max_depth`



`min_size`

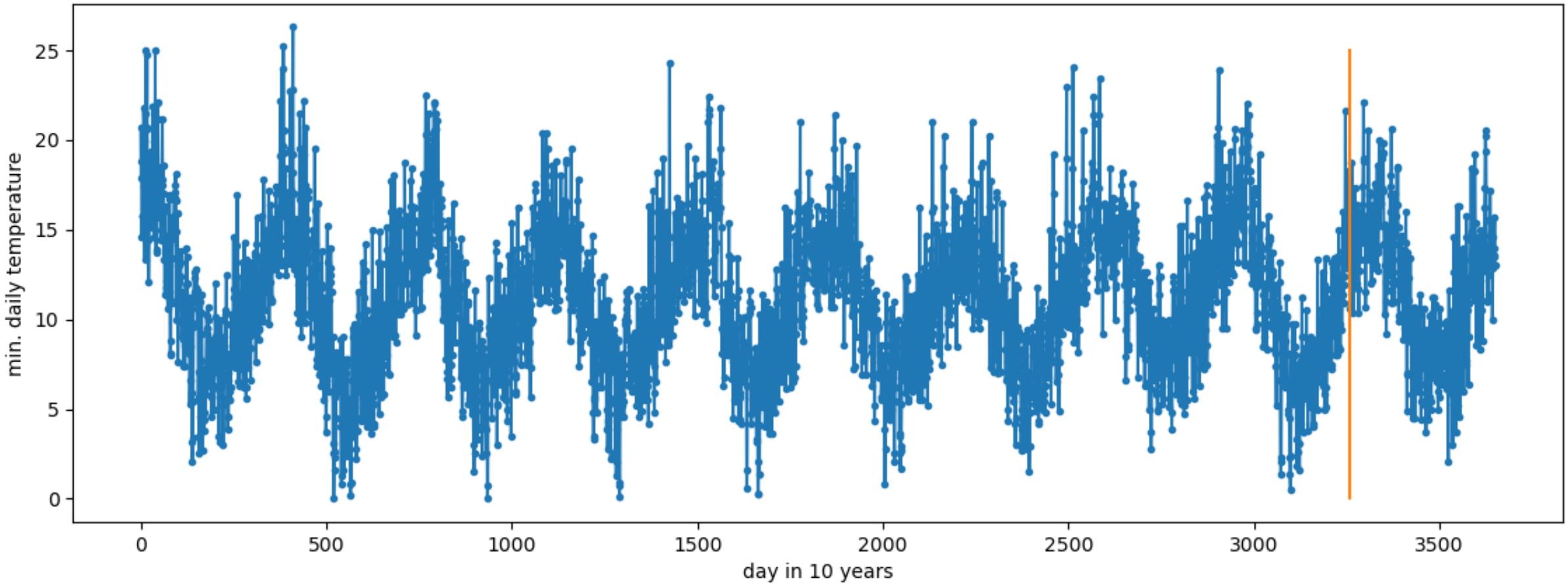
Test

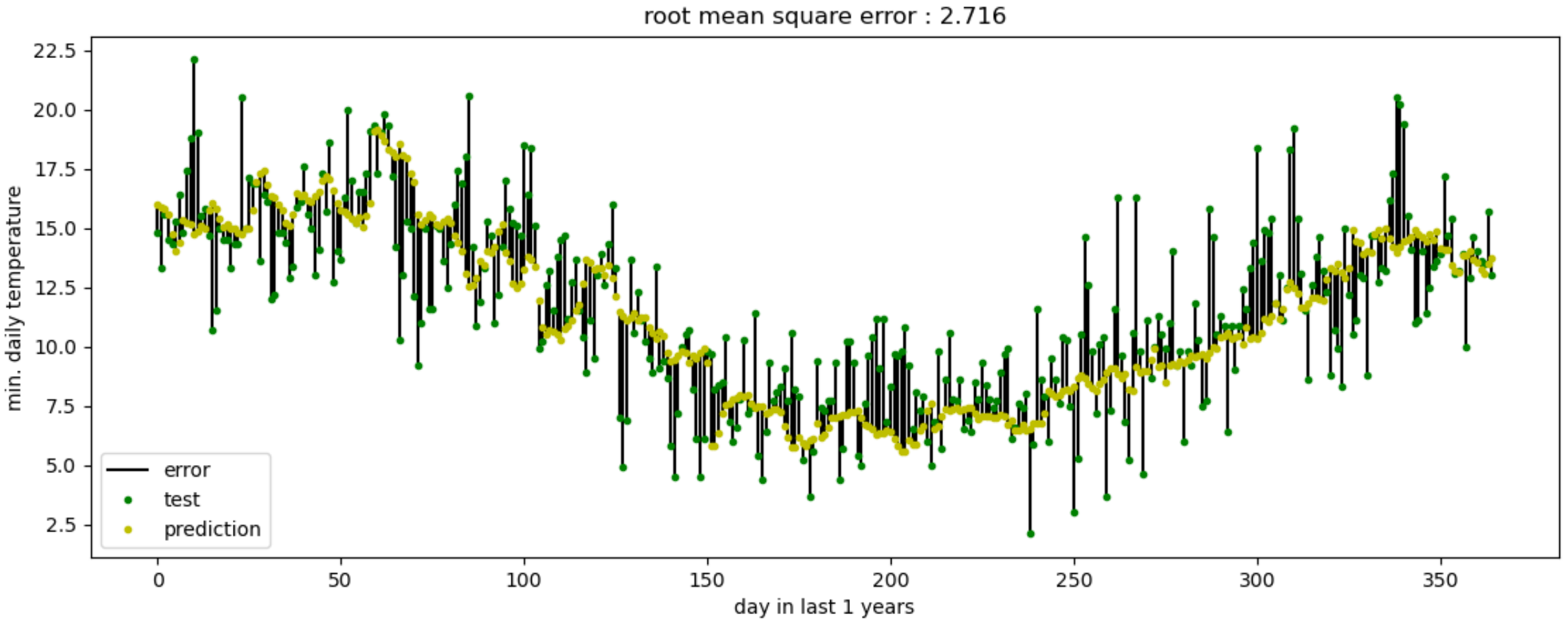
```
import pandas as pd

def load_daily_min_temperatures():
    df = pd.read_csv('https://raw.githubusercontent.com/jbrownlee/'
                     'Datasets/master/daily-min-temperatures.csv')
    # Minimum Daily Temperatures Dataset over 10 years (1981-1990)
    # in Melbourne, Australia. The units are in degrees Celsius.
    # These are the features to regress:
    day = pd.DatetimeIndex(df.Date).day.to_numpy() # 1...31
    month = pd.DatetimeIndex(df.Date).month.to_numpy() # 1...12
    year = pd.DatetimeIndex(df.Date).year.to_numpy() # 1981...1999
    X = np.vstack([day, month, year]).T # np array of 3 columns
    y = df.Temp.to_numpy()
    return X, y
```

Download the .csv file to not depend on Internet connection.

```
def test_regression(last_years_test=1):
    X, y = load_daily_min_temperatures()
    plt.figure()
    plt.plot(range(len(day)), y,'.-')
    plt.xlabel('day in 10 years'), plt.ylabel('min. daily temperature')
    idx = last_years_test*365
    Xtrain = X[:-idx,:] # first years
    Xtest = X[-idx:]
    ytrain = y[:-idx] # last years
    ytest = y[-idx:]
    #
    # TODO: your regression code here
    # for instance, max_depth=10, min_size=5,
    # ratio_samples=0.5, num_trees=50,
    # num_random_features=2
```

```
plt.figure()
x = range(idx)
for t, y1, y2 in zip(x, ytest, ypred):
    plt.plot([t, t], [y1, y2], 'k-')
plt.plot([x[0], x[0]],[ytest[0], ypred[0]], 'k-', label='error')
plt.plot(x, ytest, 'g.', label='test')
plt.plot(x, ypred, 'y.', label='prediction')
plt.xlabel('day in last {} years'.format(last_years_test))
plt.ylabel('min. daily temperature')
plt.legend()

errors = ytest - ypred
rmse = np.sqrt(np.mean(errors**2))
plt.title('root mean square error : {:.3f}'.format(rmse))
plt.show()
```


min. daily temperature
day in 10 years
0
5
10
15
20
25
0
500
1000
1500
2000
2500
3000
3500


root mean square error : 2.716
min. daily temperature
day in last 1 years
error
test
prediction
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0
50
100
150
200
250
300
350

How to assess the regression error

RMSE = root of mean square error, $n =$ number of test samples

$$\mathrm{RMSE}(y_{\mathrm{true}}, y_{\mathrm{pred}}) = \sqrt{\frac{\sum_{i=1}^{n} (y_{\mathrm{pred},i} - y_{\mathrm{true},i})^2}{n}}$$