# Decision Tree

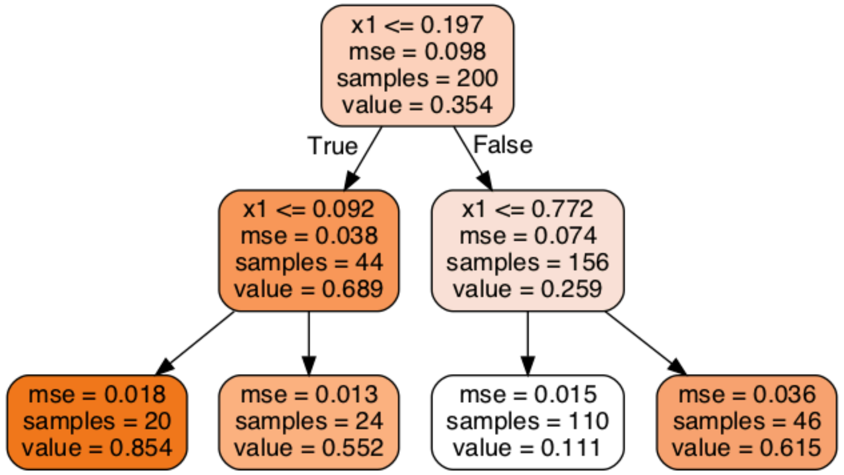
- Decision Trees make very few assumptions about the training data (versatile but wild)
- The tree structure could adapt itself to fit each and every point in the training data – **overfitting**
- At least restrict the maximum depth of the Decision Tree (**max_depth**)
    - **min_samples_split**: the minimum number of samples a node must have before it can be split
    - **min_samples_leaf**: the minimum number of samples a leaf node must have
    - **min_weight_fraction_leaf**: same as **min_samples_leaf** but expressed as a fraction of the total number of weighted instances
    - **max_leaf_nodes**: the maximum number of leaf nodes
    - **max_features**: the maximum number of features that are evaluated for splitting at each node
- Increasing **min_*** hyperparameters or reducing **max_*** hyperparameters will regularize the model
- Cross-validation or grid search

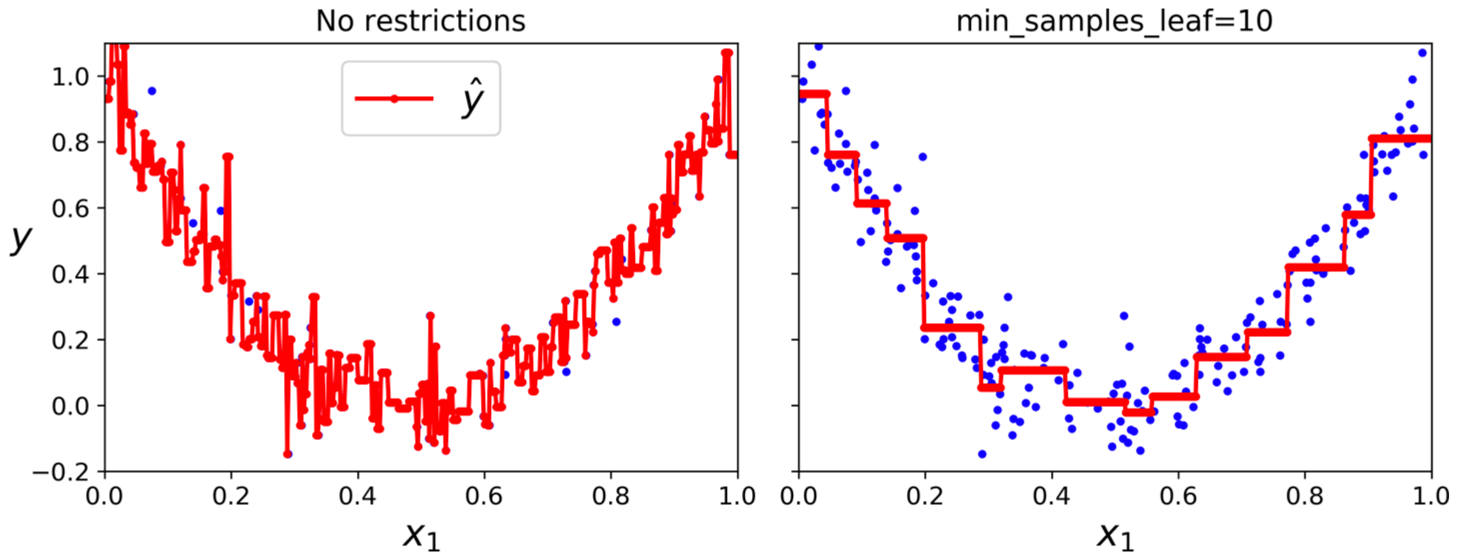
https://www.dezyre.com/recipes/optimize-hyper-parameters-of-decisiontree-model-using-grid-search-in-python

# Decision Tree Regression

**Decision Tree Regressor**: It's used to solve **regression** problems. It predicts continuous  outputs.

```python
from sklearn.tree import DecisionTreeRegressor

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





**Bias vs Variance**

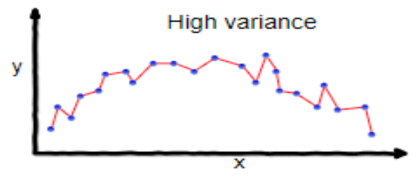
So the expected squared error at a point x is

$$Err(x) = E\left[(Y - \hat{f}\,(x))^2\right]$$

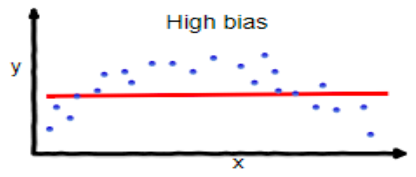
The Err(x) can be further decomposed as

$$Err(x) = \left(E[\hat{f}\,(x)] - f(x)\right)^2 + E\left[\left(\hat{f}\,(x) - E[\hat{f}\,(x)]\right)^2\right] + \sigma_e^2$$

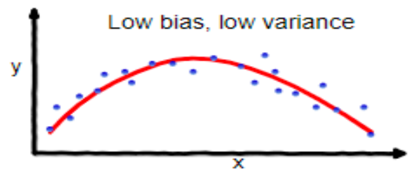
$$Err(x) = \mathrm{Bias}^2 + \mathrm{Variance} + \mathrm{Irreducible\ Error}$$



**overfitting**



**underfitting**



**Good balance**

## Model Ensembles

- Rather than creating a single model they generate a set of models and then make predictions by aggregating the outputs of these models.
- A prediction model that is composed of a set of models is called a **model ensemble**.
- In order for this approach to work the models that are in the ensemble must be different from each other.

## There are two standard approaches to creating ensembles:

- Boosting
  - Decrease Bias
  - **Sequential Learners**, where different models are generated sequentially, and the mistakes of previous models are learned by their successors. This aims at exploiting the dependency between models by giving the mislabeled examples higher weights.

- Bagging
  - Decrease Variance
  - **Parallel Learners**, where base models are generated in parallel. This exploits the independence between models by averaging out the mistakes.

## Boosting

- Weak learner (predictive model) :
    - learners which does only slightly better than random guessing
- Strong learner:
    - achieving high accuracy/precision
- Boosting:
    - Just as humans learn from their mistakes and try not to repeat them further in life, the Boosting algorithm tries to build a strong learner from the mistakes of several weaker models.

- Boosting works by iteratively creating models and adding them to the ensemble.
- The iteration stops when a predefined number of models have been added.
- When we use **boosting** each new model added to the ensemble is biased to pay more attention to instances that previous models miss-classified.
- This is done by incrementally adapting the dataset used to train the models. To do this we use a **weighted dataset**

## Weighted Dataset

- Each instance has an associated weight $\mathbf{w}_i \geq 0$,
- Initially set to $\frac{1}{n}$ where $n$ is the number of instances in the dataset.
- After each model is added to the ensemble it is tested on the training data and the weights of the instances the model gets correct are decreased and the weights of the instances the model gets incorrect are increased.
- These weights are used as a distribution over which the dataset is sampled to created a replicated training set, where the replication of an instance is proportional to its weight.

During each training iteration the algorithm:

1. Induces a model and calculates the total error, $\epsilon$, by summing the weights of the training instances for which the predictions made by the model are incorrect.
2. Increases the weights for the instances misclassified using:
$$\mathbf{w}[i] \leftarrow \mathbf{w}[i] \times \left(\frac{1}{2 \times \epsilon}\right)$$
3. Decreases the weights for the instances correctly classified:
$$\mathbf{w}[i] \leftarrow \mathbf{w}[i] \times \left(\frac{1}{2 \times (1 - \epsilon)}\right)$$
4. Calculate a **confidence factor**, $\alpha$, for the model such that $\alpha$ increases as $\epsilon$ decreases:
$$\alpha = \frac{1}{2} \times log_e\left(\frac{1 - \epsilon}{\epsilon}\right)$$

# Types of Boosting Algorithms

- **AdaBoost (Adaptive Boosting)**
  - The first boosting technique that aims at combining multiple weak classifiers to build one strong classifier.

- **Gradient Tree Boosting**
  - A loss function to be optimized.
  - A weak learner to make predictions.
  - An additive model to add weak learners to minimize the loss function.

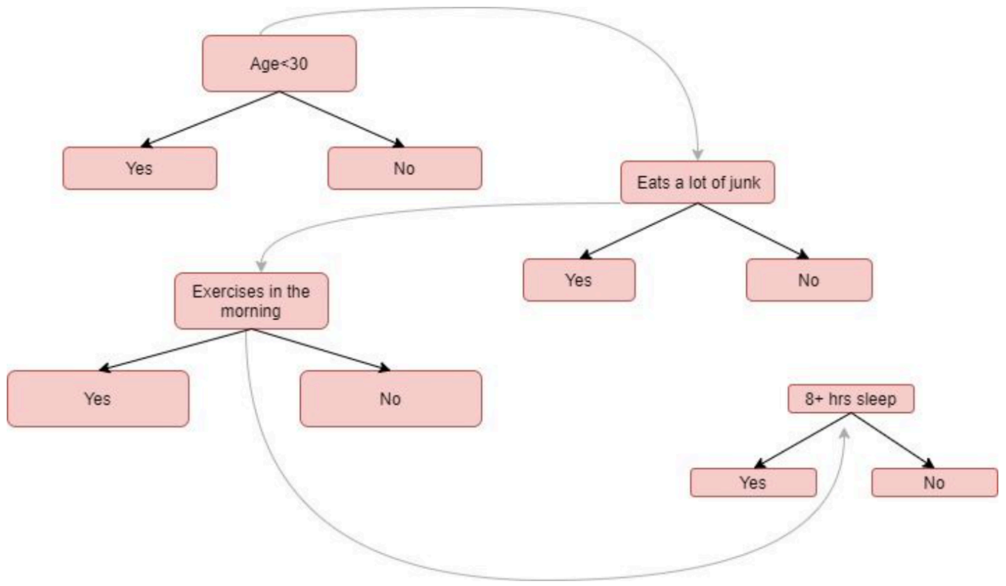
- **XGBoost -(Tianqi Chen)**
  - XGBoost stands for eXtreme Gradient Boosting.
  - It is an implementation of gradient boosted decision trees designed for efficiency of compute time and memory resources and model performance with distributed computing.

https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/

https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/

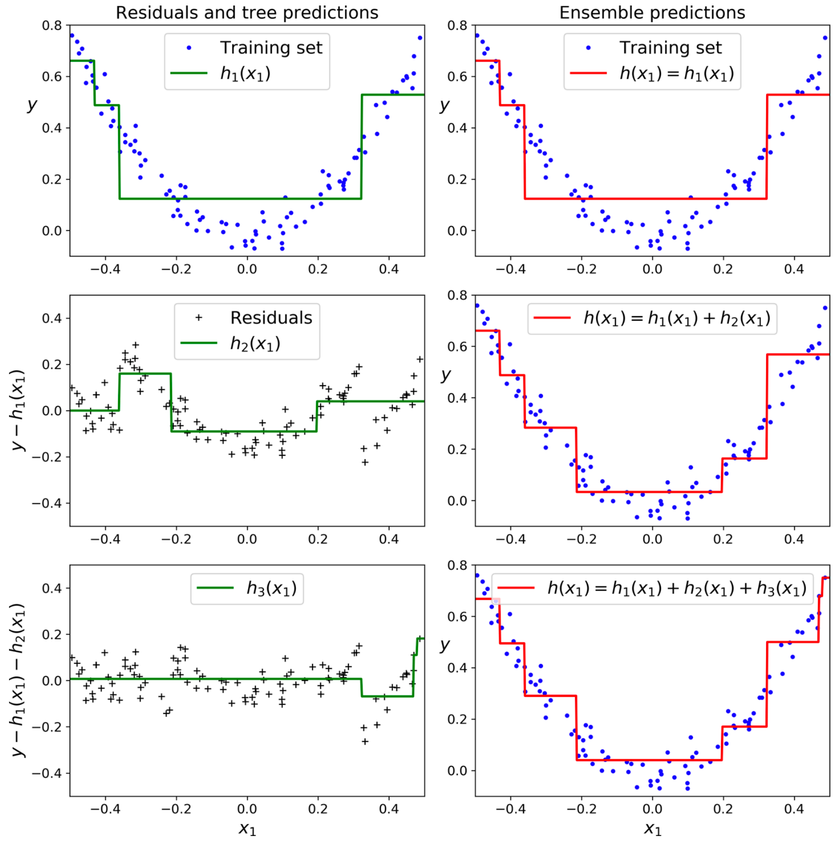
# Boosting: State-of-the-art

## AdaBoost



```python
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

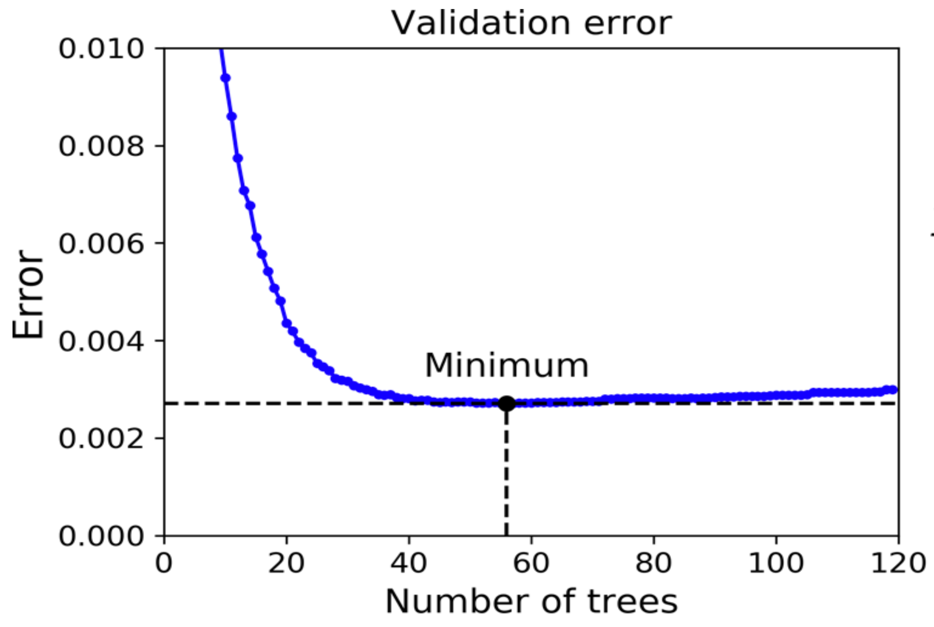
## Gradient Boost

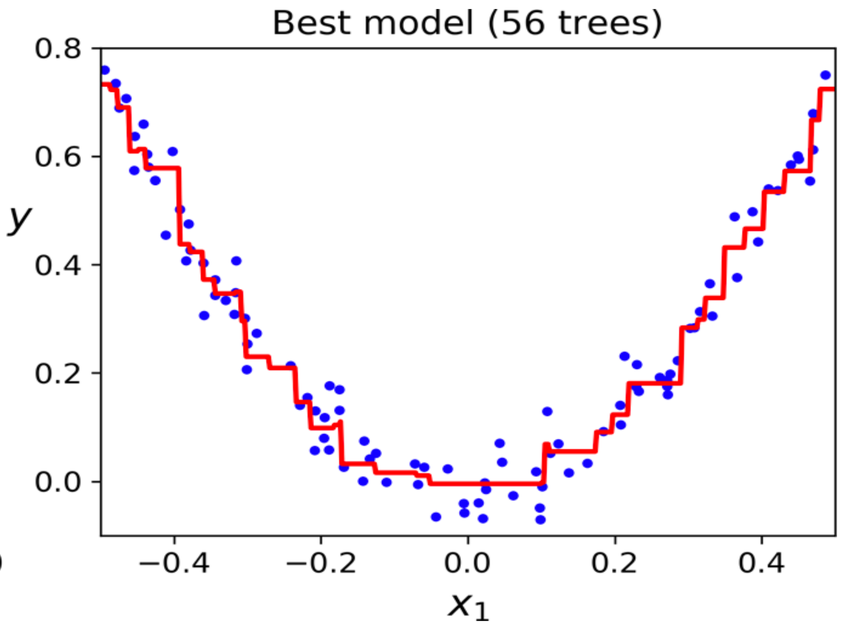


```python
import xgboost

xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

```python
xgb_reg.fit(X_train, y_train,
            eval_set=[(X_val, y_val)], early_stopping_rounds=2)
y_pred = xgb_reg.predict(X_val)
```

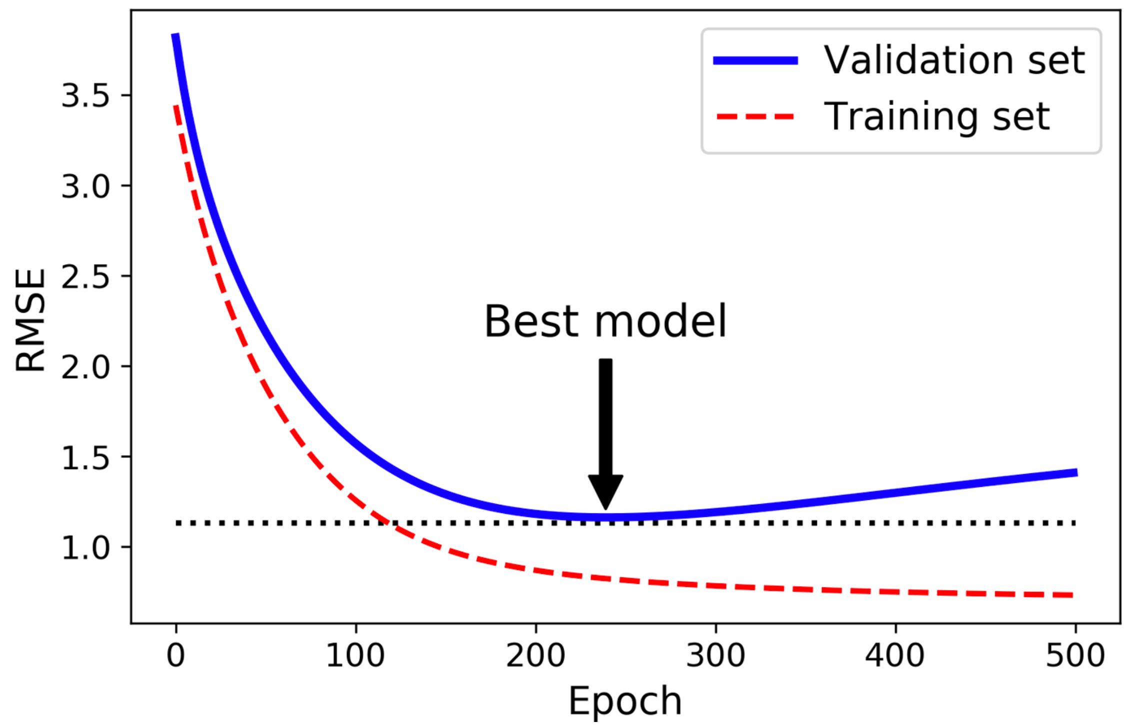
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```





## Early Stopping

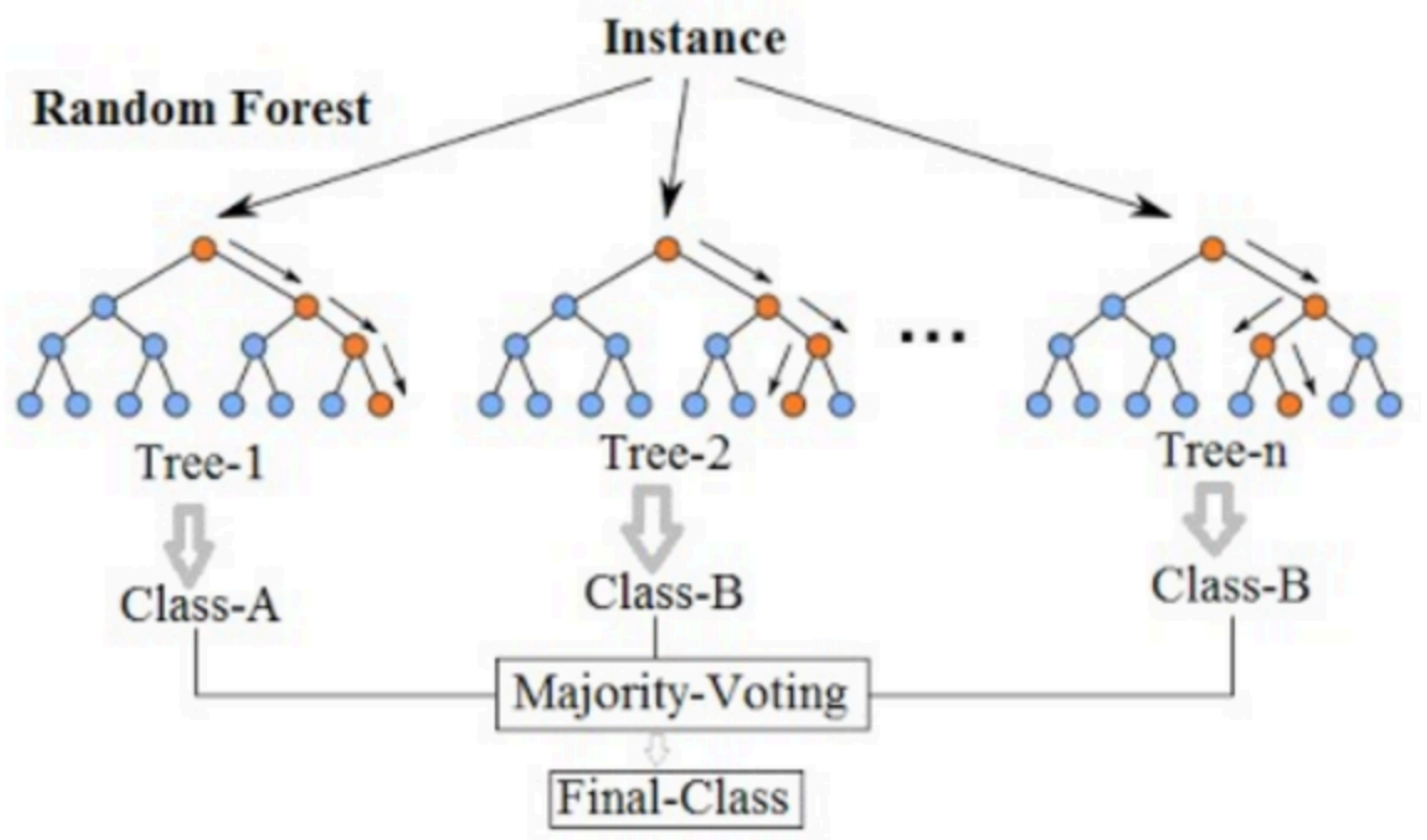
- Stop training as soon validation error hits a target
- Simple and effective
- Commonly used in DL



## Bagging

- When we use **bagging** (or **bootstrap aggregating**) each model in the ensemble is trained on a random sample of the dataset known as **bootstrap samples**.
- Each random sample is the same size as the dataset and **sampling with replacement** is used.
- Consequently, every bootstrap sample will be missing some of the instances from the dataset so each bootstrap sample will be different and this means that models trained on different bootstrap samples will also be different

# Random Forest Simplified



- Which approach should we use? Bagging is simpler to implement and parallelize than boosting and, so, may be better with respect to ease of use and training time.
- Empirical results indicate:
  - boosted decision tree ensembles were the best performing model of those tested for datasets containing up to 4,000 descriptive features.
  - random forest ensembles (based on bagging) performed better for datasets containing more that 4,000 features.

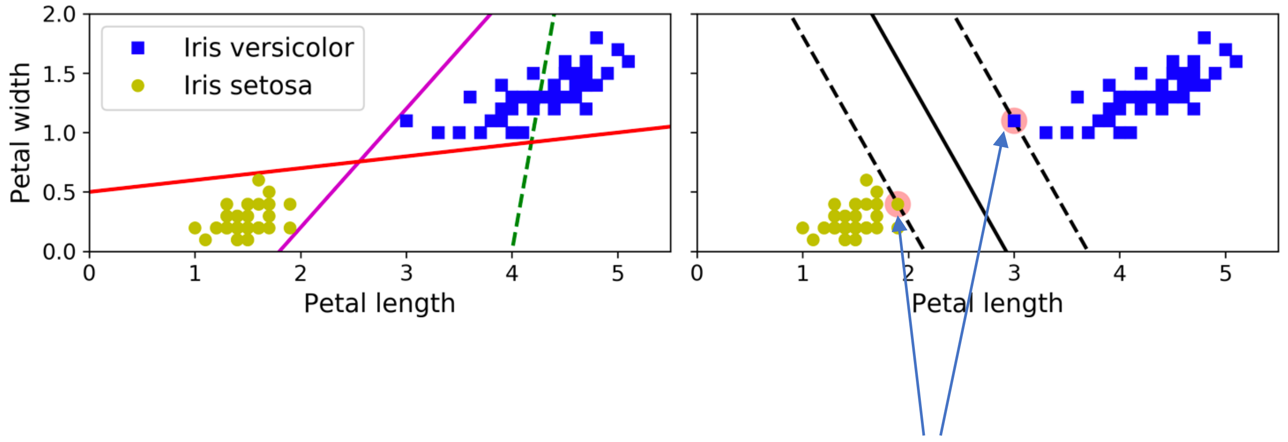
## Support Vector Machine (SVM)

## SVM

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data, the algorithm outputs an optimal hyperplane which categorizes new examples.

# SVM: Classifier

- SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes.
- This is called *large margin classification*.
- Only need consider support vectors for training
- find a good balance between keeping the street as large as possible and limiting the *margin violations* (i.e., instances that end up in the middle of the street or even on the wrong side)

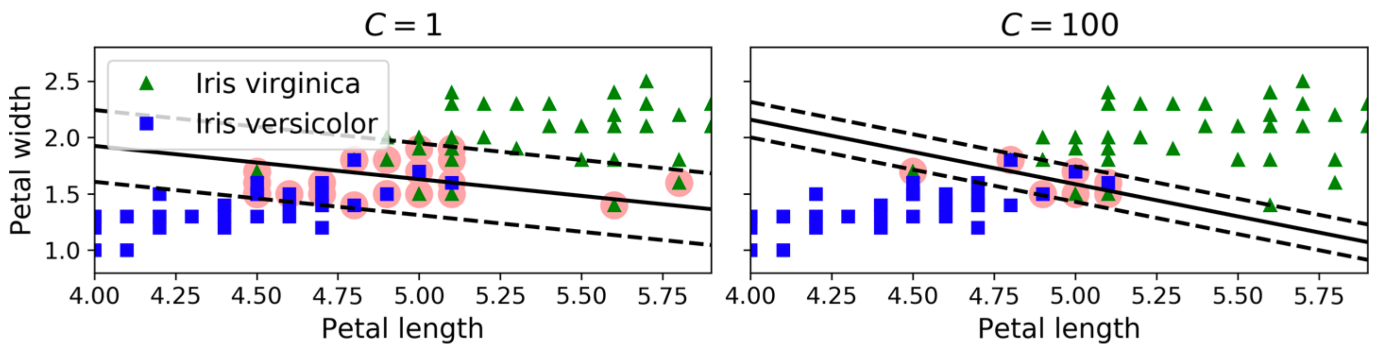


Support Vectors: points on the edges

# SVM in Scikit Learn

```python
svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ])

svm_clf.fit(X, y)
```

```python
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

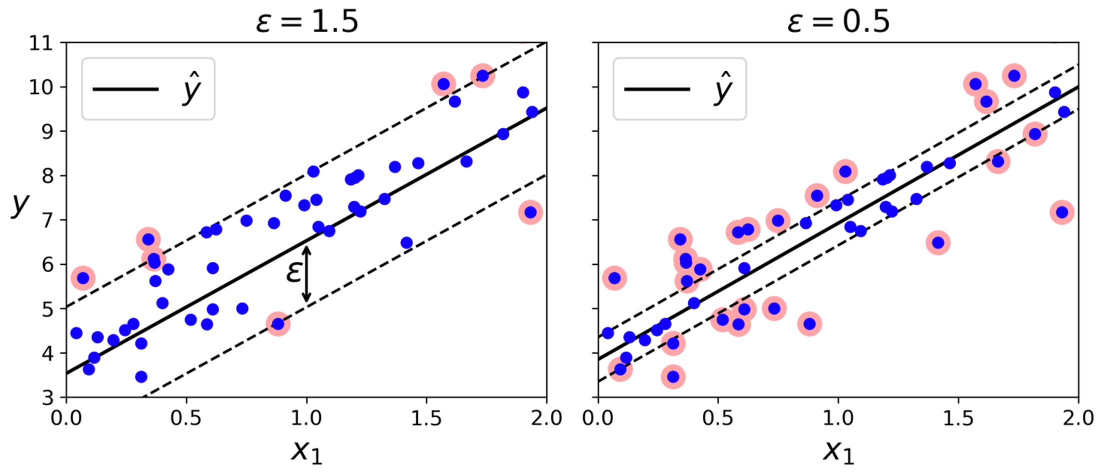


- **C** is the hyperparameter
- The higher the more overfitting

# SVR: SVM Regression

- SVM: move as many data *off* the street as possible

- SVR: include as many data *on* the street as possible

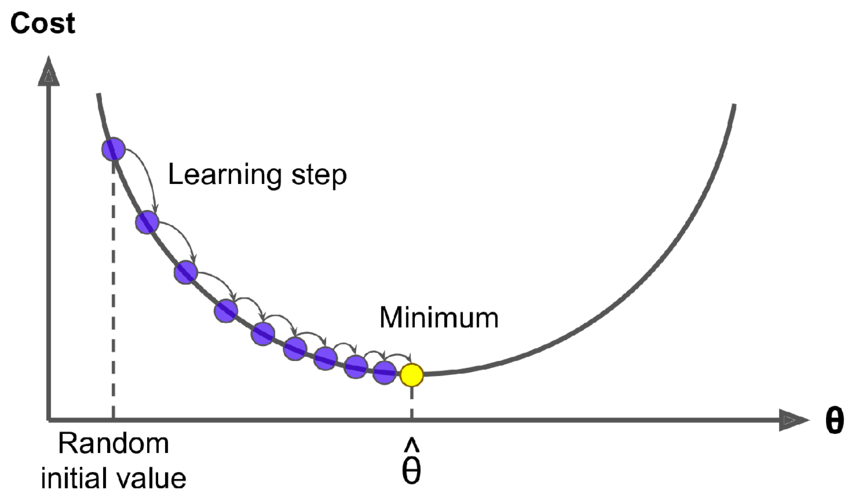


```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

## Optimization Algorithm: Gradient Descent

**Gradient descent** is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function.



# Regression Loss Function

MSE/Quadratic/L2 Loss

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Mean Absolute Error (MAE) /L1 Loss

$$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n}$$

Mean Bias Error

$$MBE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{n}$$

# Classification Loss Functions

## Hinge Loss

$$SVMLoss = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

the score of correct category should be greater than sum of scores of all incorrect categories by some safety margin (usually one)

Cross Entropy/(Multinomial) Log(istic)/ Loss/Negative Log Likelihood

$$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

- Most common setting for classification problems.
- Cross-entropy loss increases as the predicted probability diverges from the actual label.

Categorical Cross-Entropy/Softmax Loss

$$SoftmaxLoss = -\sum_{i=1}^{M} y_i \log(\hat{y}_i)$$

## Further Reading

- https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#
- https://www.analyticsvidhya.com/blog/2019/08/detailed-guide-7-loss-functions-machine-learning-python-code/
- https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a