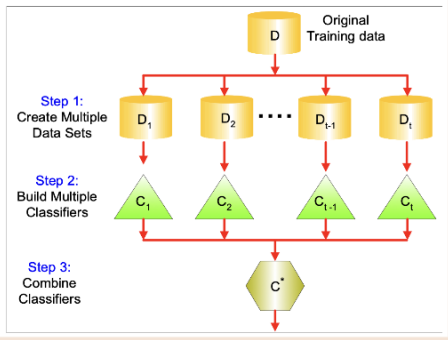
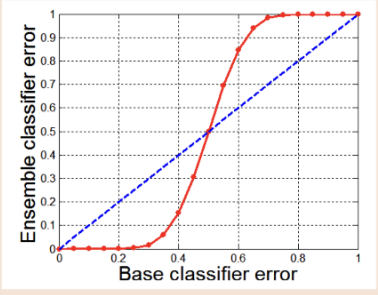
✲ Ensemble Methods

- Generate a set of classifiers from training data → Predict class label of previously unseen cases by aggregating predictions made by multiple classifiers
- For classification → "Majority Vote" , For Regression → "Averaging"
- Disadvantage: usually produces output that is very hard to analyze.





- Why it works? : Suppose 25 (independent) classifier each has error rate $\varepsilon = 0.35$

$$P(X \geq 13 \mid p=\varepsilon, n=25) = \sum_{i=13}^{25} \binom{25}{i} \varepsilon^i (1-\varepsilon)^{25-i} = 0.06$$ (Prob. of ensemble makes a wrong prediction)

• Bagging:

- Given a set D, generate k training dataset Di using bootstrap
- Compute k models (Mi) using Di's
- Each classifier Mi computes its prediction for unknown sample x
- The bagged classifier (M*) return the class predicted by the majority of models
- When class values are -1 and 1 → output of ensemble is; $M^*(x) = \text{sign}\left(\sum_{i=1}^{k} M_i(x)\right)$
- Models may be weighted based on their performance
- For regression; average models' output
- Pseudocode:

```
def BaggingFit(X,y,base_model,no_models):
    m,n = X.shape

    models = [ None ] * no_models

    for i in range(no_models):

    # generate the array of bootstrapped examples
    ind = np.floor( m * np.random.rand(m) ).astype(int)

    # create a copy of the base model
    model = sklearn.base.clone(base_model)

    # fit the classifier
    model.fit(X[ind,:], y[ind])

    # memorize the model for later prediction
    models[i] = model

    # return the array of models
    return models
```

```
def BaggingPredict(models,X,use_average=False):
    no_models = len(models)
    m = X.shape[0]
    predict = np.zeros( (m, no_models) )
    for i in range(no_models):
        predict[:,i] = models[i].predict(X)

    return predict

    if use_average:
        predict = np.mean(predict, axis=1)
    else:
        # Make overall prediction by majority vote
        predict = np.mean(predict, axis=1) > 0 # if +1 vs -1

    return predict
```

- Bagging reduces variance by voting/averaging
- Usually the more classifier, the better
- It can help a lot if data are noisy
- Stable / Unstable Classifiers:

A classifier is unstable when small changes in training set can cause large changes to the model (Decision/Regression trees, linear regression, neural networks are unstable model examples)

If the learning algorithm unstable bagging always improves performance → Bagging stable classifier is a bad idea!

Random Forests:

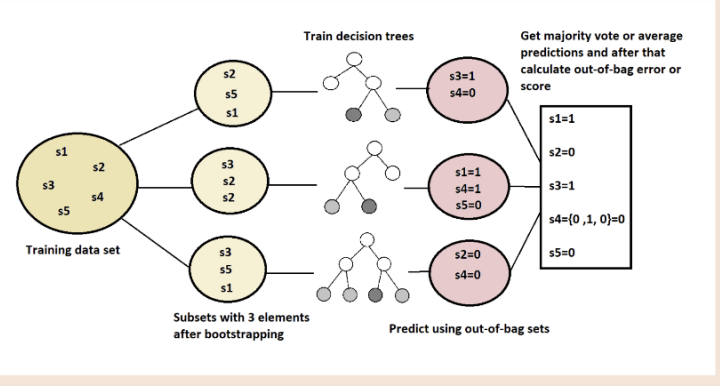
- Random Forests are a combination of tree predictors.
- Each tree depends on values of a random vector sampled independently and with the same distribution for all trees in the forest
- Generalization error of a forest of tree classifiers depend on → strength of individual trees (How good they are in average?) / correlation between them (How much group-think?)
- Using random selection of features to split each node yields error rates more robust to noise.
- Pseudocode;

Algorithm 22.5: Random Forest Algorithm

RANDOMFOREST$(\mathbf{D}, K, p, \eta, \pi)$:

foreach $\mathbf{x}_i \in \mathbf{D}$ **do**
  $v_j(\mathbf{x}_i) \leftarrow 0$, for all $j = 1, 2, \ldots, k$
for $t \in [1, K]$ **do**
  $\mathbf{D}_t \leftarrow$ sample of size n with replacement from $\mathbf{D}$
  $M_t \leftarrow$ DECISIONTREE $(\mathbf{D}_t, \eta, \pi, p)$
  foreach $(\mathbf{x}_i, y_i) \in \mathbf{D} \setminus \mathbf{D}_t$ **do** // out-of-bag votes
    $\hat{y}_i \leftarrow M_t(\mathbf{x}_i)$
    if $\hat{y}_i = c_j$ **then** $v_j(\mathbf{x}_i) = v_j(\mathbf{x}_i) + 1$
$\epsilon_{oob} = \frac{1}{n} \cdot \sum_{i=1}^{n} I\left(y_i \neq \arg\max_{c_j} \{v_j(\mathbf{x}_i) | (\mathbf{x}_i, y_i) \in \mathbf{D}\}\right)$ // OOB error
return $\{M_1, M_2, \ldots, M_K\}$

- Out of Bag Samples: For each observation (x_i, y_i) construct its random forest predictor by averaging only those trees corresponding to bootstrap samples in which the observation did not appear.
  - ! Almost identical with n-fold cross validation
  - ! Once the OOB error stabilizes, training can be terminated



- Properties of Random Forest:
  - » Easy to use, only 2 parameters
  - » Very high accuracy
  - » No overfitting if selecting large # of trees
  - » Insensitive to the choice of split %.
  - » Returns an estimate of variable importance

Boosting:

- Weights are assigned to each training example
- Series of k classifiers iteratively learned
- After a classifier M_i is learned, weights are updated.
- Next classifier M_{i+1} will focus more on misclassified (by M_i) training data
- Final M^* is a weighted sum of all classifiers' output

» Adaboost

- Compute strong classifier as a combination of weak classifiers

$$H(x) = \text{sign}\left(\sum_{i=1}^{N} \alpha_i h_i(x)\right) \quad \alpha_i = \ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$$

(α_i: weight; ε_i: estimated error; $h_i(x)$: output of i^{th} weak classifier)

- Ex: Our data consist of 10 points with +1/-1 label. At the beginning, they have same weight (1/10)

w	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100
x	0	1	2	3	4	5	6	7	8	9
y	+1	+1	+1	-1	-1	-1	-1	-1	+1	+1
	+1	+1	+1	h_1	-1	-1	-1	-1	-1	-1

	-1	-1	-1	-1	-1	-1	-1	-1	h_2 +1	+1
w	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.25	0.25
x	0	1	2	3	4	5	6	7	8	9
y	+1	+1	+1	-1	-1	-1	-1	-1	+1	+1

We generate the training data using the weights and compute weak classifier h_1 → error $\varepsilon_1 = 2/10$ and $\alpha_1 = 1.386$

Weights are updated by multiplying them with $e^{-\alpha_1}$ if it is correctly classified, e^{α_1} if it is incorrectly classified. → then we normalize them

We continue with the same processes (compute h_2, ε_2, α_2 → compute new weights)

$\varepsilon_2 = 0.625 \times 3 = 0.1875$ and $\alpha_2 = 1.466$

↙ error is increased, stop!

→ $\varepsilon_3 = 0.038 \times 5 = 0.19$ and $\alpha_3 = 1.45$

w	0.167	0.167	0.167	0.038	0.038	0.038	0.038	0.038	0.154	0.154
x	0	1	2	3	4	5	6	7	8	9
y	+1	+1	+1	-1	-1	-1	-1	-1	+1	+1
	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
										h_3

» The final model computed as: $H(x) = \text{sign}(\alpha_1 h_1(x) + \alpha_2 h_2(x) + \alpha_3 h_3(x))$

• Pseudocode:

Algorithm 22.6: Adaptive Boosting Algorithm: AdaBoost

ADABOOST(K, D):

$\mathbf{w}^0 \leftarrow \left(\frac{1}{n}\right) \cdot \mathbf{1} \in \mathbb{R}^n$
$t \leftarrow 1$
while $t \leq K$ **do**
  $\mathbf{D}_t \leftarrow$ weighted resampling with replacement from $\mathbf{D}$ using $\mathbf{w}^{t-1}$
  $M_t \leftarrow$ train classifier on $\mathbf{D}_t$
  $\epsilon_t \leftarrow \sum_{i=1}^{n} w_i^{t-1} \cdot I\left(M_t(\mathbf{x}_i) \neq y_i\right)$ // weighted error rate on D
  if $\epsilon_t = 0$ **then break**
  else if $\epsilon_t < 0.5$ **then**
    $\alpha_t = \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$ // classifier weight
    for $i \in [1, n]$ **do**
      // update point weights
      $w_i^t = \begin{cases} w_i^{t-1} & \text{if } M_t(\mathbf{x}_i) = y_i \\ w_i^{t-1}\left(\frac{1-\epsilon_t}{\epsilon_t}\right) & \text{if } M_t(\mathbf{x}_i) \neq y_i \end{cases}$
    $\mathbf{w}^t = \frac{\mathbf{w}^t}{\mathbf{1}^T\mathbf{w}^t}$ // normalize weights
    $t \leftarrow t + 1$
return $\{M_1, M_2, \ldots, M_K\}$

» More on Boosting

- It is iterative: new models are influenced by performance of previously built ones.
- Several variants: Boosting by sampling, weights are used to sample data for training
  Boosting by weighting, weights are used by the learning algorithm.
- Base classifiers should not to be too complex, their error should not become too large quickly
  (In practice boosting sometimes overfit)

» Gradient Tree Boosting

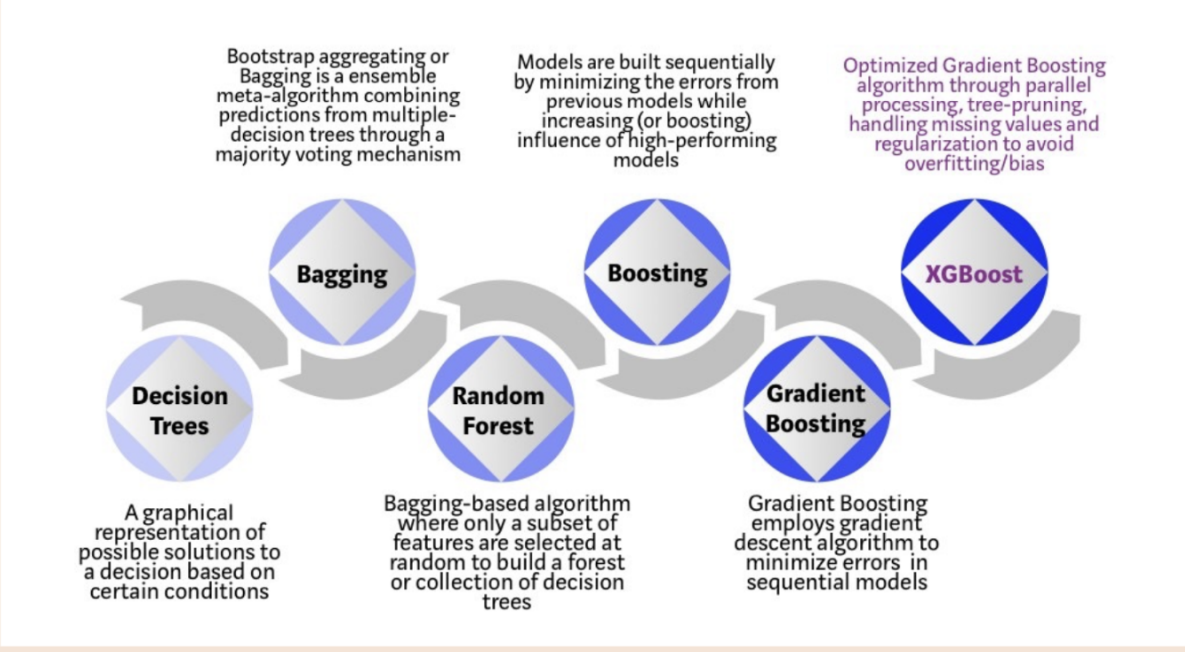
- Build a sequence of tree predictors by repeating three simple steps:
  1) Learn a basic predictor
  2) Compute gradient of a loss function → $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$ (loss function example)
  3) Compute a model to predict the residual
  4) Update the predictor with the new model ← $\hat{y}_i = y_i + \alpha \cdot \nabla MSE$
  5) Go to 2
- The predictor is increasingly accurate and increasingly complex.
- Pseudocode:

```
# number of models
n_boost = 50
# array of models
learner = [None]* n_boost

# start with the basic predictor (the mean)
mu = mean(y)
dY = y - mu

for k in range(n_boost):
    # fit the residual
    learner[k] = model.fit(X,dy)

    # set the learning rate (this should be adapted based on k, for example 1/log(k))
    alpha[k] = 1

    # update the residual
    dy = dy - alpha[k] * learner[k].predict(X)
```

```
# X_test and y_test are the test data
predict = zeros(len(y_test))+mu

for k in range(n_boost):
    predict = predict + alpha[k]*learner[k].predict(X_test)
```

⇒ Extreme Gradient Boosting (XGBoost)

- Efficient and scalable implementation of gradient boosting for classification and regression trees.
- It only deals with numerical variables
- Features: Can handle sparse data
  Approximate tree learning using quantile sketch
  x10 times faster on single machine
  More regularized model (control overfitting, better performance)



⇒ Light Gradient Boosting Machine (LightGBM)

- It grows trees leaf-wise and chooses the leaf that estimates will yield the largest decrease in loss
- Features: Faster training and higher efficiency, lower memory usage
  Better accuracy
  Support of parallel, distributed and GPU learning
  Capable of handling large-scale data

• Stacking:

- Instead of choosing one model to trust, stacking gives you a way to combine them
- Ensemble technique with two layers of classifiers
- First layer is composed of K base classifiers which are trained independently on the entire data D
  (Base classifiers should be complementary of each other, so they perform well on different subsets)
- Second layer comprises a combiner classifier C that is trained on the predicted classes from base classifiers
  (It automatically learns how to combine the outputs of the base classifiers to make final prediction)

Algorithm 22.7: Stacking Algorithm

STACKING($K, \mathbf{M}, C, \mathbf{D}$):

// Train base classifiers

for $t \in [1, K]$ **do**
  $M_t \leftarrow$ train tth base classifier on $\mathbf{D}$

// Train combiner model C on $\mathbf{Z}$

$\mathbf{Z} \leftarrow \emptyset$

foreach $(\mathbf{x}_i, y_i) \in \mathbf{D}$ **do**
  $\mathbf{z}_i \leftarrow \big(M_1(\mathbf{x}_i), M_2(\mathbf{x}_i), \ldots, M_K(\mathbf{x}_i)\big)^T$
  $\mathbf{Z} \leftarrow \mathbf{Z} \cup \{(\mathbf{z}_i, y_i)\}$

$C \leftarrow$ train combiner classifier on $\mathbf{Z}$

return $(C, M_1, M_2, \ldots, M_K)$