Lecture 16
Forecasting with ML: Random Forest, Gradient Boosting & ANN

Machine Learning: Introduction

• **Machine learning** algorithms (**ML**) are an area of statistics that is used to extract patterns ("learn") from available data, including **non-linear** and/or **hidden** relationships.

• The available data is usually divided into **training data**, used to extract patterns, and **validation data**, used to confirm or validate the pattern.

• Based on these patterns, ML methods build models in order to make predictions. For us, the goal is to get a flexible estimator $m(x_0)$:

$$m(x_0) = E[Y|X = x_0]$$

• Very general definition. We think of ML methods as methods that capture complicated non-linear relations and interactions.

Machine Learning: Trees & Random Forest

• **Random forest models (RF)** belongs to the class of **decision-tree** models; RF are composed of many decision trees. These decision trees are formed by segmenting the space generated by the explanatory variables ("**predictors**") into a number of simple regions.

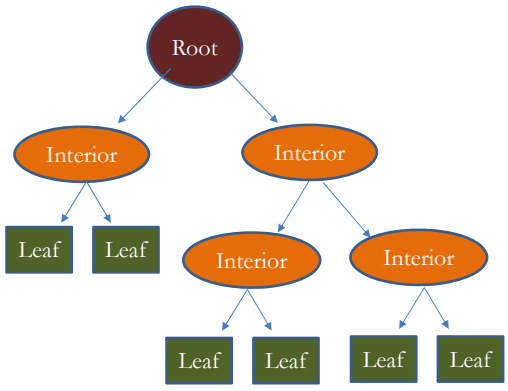
• The set of splitting rules used to segment the predictor space can be represented by an upside tree. Thus, we call this method a **decision tree**.

• A decision tree makes a prediction for a given observation, usually by the mean of the data in the region to which it belongs.

• RF models make a prediction based on many decision trees. These trees are formed by randomly selecting **predictors** and subsamples.

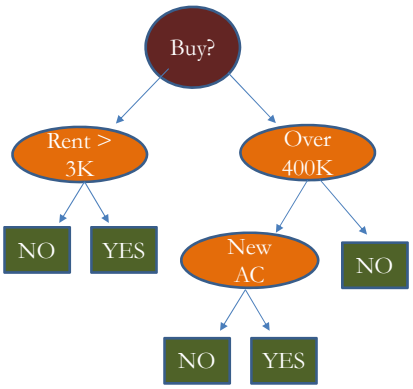
Decision Trees: Upside Tree

• Decision Trees are made up of a **root** node (at the top), followed by **interior** nodes (or **decision** nodes, since the tree creates **branches** based by answering Yes/No questions), and, ending in **leaf** (**final**) nodes. At this final leaf node, we make a prediction. For example,



Decision Trees: Upside Tree and Nodes

- A **branch** represents an outcome of a test (Buy?, Rent?)
- An **interior node** tests an attribute of the decision. At each node, an attribute is chosen to split the data (Over/Under 400K?)
- A **leaf** represents a category (class) or a value (numerical prediction)



Decision Trees: Depth and Leaves

- In a regression tree, **depth** is the maximum number of recursive splits along a root-to-leaf path. As we add depth to a tree, we add complexity, which reflect interactions between variables and potential non-linear behavior.

- When depth = 1, we have **stumps** -i.e., just one split- determining two leaves. If we have depth = d, we have 2^d leaves.

- Each additional level introduces higher-order interactions, which is good **locally** –i.e., for in-sample estimation. But, at the same time, the effective sample size per leaf decreases, which affects the average.

- As we will see later, the tree depth is a very important parameter: It determines the complexity in tree-based methods.

Decision Trees: Leaves and Predictions

• In a decision tree, an input (observation) is entered at the top (**root**) and as it traverses down the tree the data gets bucketed into smaller and smaller sets. The last bucket is the **leaf**. That is, a regression tree partitions the covariate space, Λ, into L rectangles ("leaves"):

$$\Lambda = \cup_{l=1}^{L} R_l$$

• The leaves are the basis of predictions. Usually, the prediction is the average of all the observations that reach a particular leaf. (It is possible to use medians or some aggregate.)

$$\hat{m}(x_0) = \sum_{l=1}^{L} \bar{y}_l * I(x_{0i} \in R_l)]$$

where $\bar{y}_l$ is the sample mean of Y in region R_l.

Decision Trees: Leaves and Predictions

Example: We want to predict IBM monthly excess returns, using as predictors the 3 Fama-French factors. The training data corresponds to the period **1973:Feb – 2024:Dec**.

All observations with Mkt_RF < **2.7%**, go to the left (**Yes**); the rest to the right (**No**). On the left, we further split by looking at observations with SMB returns < **0.5%**. Then, we compute the average return in each final node (**leaf**).

We generate a tree with 4 **branches** and 3 **leaves**. All observations go through the tree. We average the value of IBM excess returns in each **leaf**:

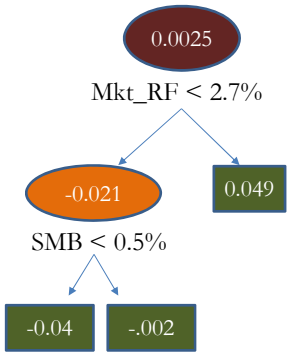
Leaf 1 (Mkt_RF < **2.7%** & SMB < **0.5%**): **-4.0%**.

Leaf 2 (Mkt_RF < **2.7%** & SMB > **0.5%**): **-0.2%**.

Leaf 3 (Mkt_RF > **2.7%**): **4.9%**.

Decision Trees: Leaves and Predictions

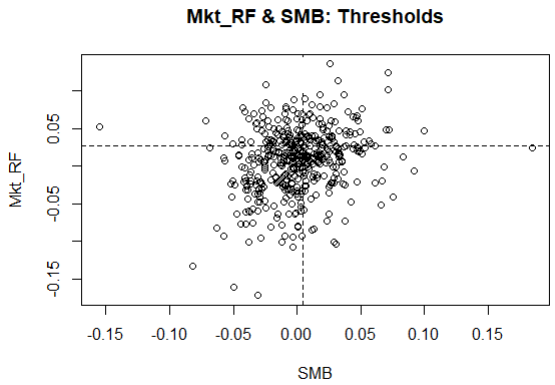
Example (continuation): The average value of IBM excess returns in each **leaf** is reported in the green squares.



Now, we have a predictive model. For example, if the market is less than **2.7%** & SMB is less than **0.5%**, we predict IBM excess returns to be **-4%**.

Decision Trees: Leaves and Predictions

Example (continuation): We have generated regions (leaves), based on Mkt_RF & SMB to split the IBM excess returns. Each regions delivers a prediction.



Decision Trees: Recursive Partitioning

• The basic algorithm that is used to fit decision trees to data is **recursive partitioning** of the input space in the **training** data set.

• This algorithm grows a tree, one decision node at a time. The order in which these nodes are added is **unique**. Once a node enters a tree it remains there (in its order) forever, with the decision tree "split points" for each node **staying the same as the tree grows**.

• We start at the root node. We search every distinct value of every input variable to find the predictor and split value that partitions the data into two regions (R_1 & R_2) such that the overall sums of squares error (RSS) are minimized:

$$\min_{c_1, c_2} \sum_{i \in R_1} (y_i - c_1)^2 + \sum_{i \in R_2} (y_i - c_2)^2$$

where c_1 and c_2 are the means in each region.

Decision Trees: Growing the Tree

• In the previous IBM example, we found R_1 as the region with **Mkt_RF < 2.7%** (& c_1 = **-0.021**), while R_2 as the region with **Mkt_RF > 2.7%** (& c_1 = **0.049**) .

• Once we found the best split for the root node, we repeat the splitting process on each of the two regions. We continue this process until some stopping criterion is reached.

• It is possible that we keep partitioning the data into too "many branches," ending with a *lengthy* tree, with too many leaves. We call these lengthy trees "**complex**." As the tree grows -i.e., become more complex-, we increase the variance in the predictions.

• These lengthy trees tend to **overfit** to specific attributes of the sample data. A slight change in the sample can vary predictions a lot.

Decision Trees: Pruning

• Complex trees tend to **overfit**: They have poor performance in the testing data ("a slight change in the data").

• Suppose we grow the tree to its maximum size, with T_0 nodes. It is common to reduce the tree by **pruning**. That is, eliminating nodes, to simplify the tree. This will help to stabilize estimation –i.e., decrease variance-, but it will introduce a little bit of bias.

• We prune in some **optimal** way. Usually, we reduce the tree by minimizing some RSS and limiting the tree.

• Two approaches to prune a full tree:

(1) **Pre-pruning approach**.

(2) **Post-pruning approach**.

Decision Trees: Pre-Pruning

• In the **pre-pruning approach**, we stop the algorithm before it can generate a fully grown tree that perfectly fits the training data set.

• Two popular ways to limit the growth of the tree:

(1) require a minimum number of cases in all final nodes

(2) impose a maximum number of decision nodes in building the tree.

• Unfortunately, it is sometimes difficult to choose the right threshold for early termination. Too high of a threshold will result in underfitted models, while a threshold that is set too low may not be sufficient to overcome the model overfitting problem.

Note: We say we "**tune**" the parameters in (1) or (2), when we adjust them to limit the growth of the tree in some optimal way.

Decision Trees: Post-Pruning

• In the **post-pruning approach**, the tree is initially fully grown, with T_0 nodes. Then, the tree-pruning step trims the fully grown tree in reverse order in order to obtain an "optimal" trimmed tree.

• Since there is a unique sequence in which the tree was build, it can be "pruned back" in **reverse sequence**. Now, we have a sequence of subtrees, each subtree has one less decision node than its predecessor.

• We want to select the optimal sub-tree. We can estimate an MSE (or RMSE) for each sub-tree, using ***K*-fold cross-validation**. (It may take a while to do this if we have a big number of sub-trees.)

Note: In this case, the tuning parameter is the number of decisions (nodes) in the tree.

Decision Trees: Variance-Bias Trade-off

• We **prune** by eliminating nodes that do not add much value to the tree. Pruning introduces bias, but helps to stabilize estimation and prediction –i.e., it reduces variance.

• We have a classic **variance-bias trade-off** situation. A big tree pays a lot of attention to the training data.

- **Low bias**, by capturing a lot of patterns.
- **Increases variance**, since it will capture a lot of noise.

• Typical solution: In the RSS minimization problem add a complexity penalty, similar to what we did with LASSO and Ridge regressions. For example, penalize the number of nodes in a tree.

Decision Trees: Variance-Bias Trade-off

• Suppose we $\hat{y}_{R_m}$ is the predicted response associated with leaf R_m. Then, we build the tree by minimizing a RSS with a penalty:

$$\min_{\hat{y}_{R_m},\lambda} \{ \sum_{m=1}^{t} \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2 + \lambda |T| \}$$

where T is the number of nodes and λ is the penalty parameter. If $\lambda = 0$, we grow the full tree. The larger λ, the smaller the tree.

• After some reformulation, the above minimization problem can be written as a Lagrangean, that has a LASSO-like form:

$$\min_{T} L = \{ \sum_i (y_i - \hat{y}_{R_m})^2 + \mu |T| \} \quad (\mu: \text{Lagrange multiplier})$$

• Usually, we use **cross-validation** to identify the optimal λ (or μ) and, therefore, the optimal subtree.

Decision Trees: Variance-Bias Trade-off

• We build the tree by minimizing a RSS with a penalty:

$$\min_{\hat{y}_{R_m},\lambda} \{ \sum_{m=1}^{T} \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2 + \lambda |T| \}$$

where T is the number of nodes and λ is the penalty parameter. If $\lambda = 0$, we grow the full tree. The larger λ, the smaller the tree.

• After some reformulation, the above minimization problem can be written as a Lagrangean, that has a LASSO-like form:

$$\min_{T} L = \{ \sum_i (y_i - \hat{y}_{R_m})^2 + \mu |T| \} \quad (\mu: \text{Lagrange multiplier})$$

• Usually, we use **cross-validation** to identify the optimal λ (or μ) and, therefore, the optimal subtree.

Decision Trees: Variance-Bias Trade-off

• In addition to λ, it is common to impose other (pre-pruning) restrictions; the most common ones:

- Requiring a minimum number of data points to look for a split, usually 20-30.

- Requiring a maximum number of internal nodes between the root and the leaves, usually 20-30.

R Note: The package *rpart* estimates regression trees, using K-fold CV to estimate λ; setting $K = 10$ is a good starting point. The rpart default imposes 20 minimum number of data points before a split (*minsplit*) and 30 maximum internal nodes (*maxdepth*).

Decision Trees: Typical Algorithm

• Steps:

Step 1: Grow Large Tree. Use recursive splitting to grow a large tree, with T nodes. We stop the recursive splitting when each terminal node has fewer than some minimum number of observations.

Step 2: Prune. Get a sequence of best subtrees, as a function of λ.

Step 3: CV. Use ***K*-fold cross-validation** to pick λ.

• Once we identify the optimal λ, from Step 2, we recover the optimal pruned subtree.

Decision Trees: Summary

- A tree split is a data-driven process based on interactions of covariates and thresholds. Equivalent to a high-dimensional set of indicator functions.
- No functional form is assumed. It accommodates non-linearities.
- Easy to interpret and determine important driving variables
- Fast predictions: Just averages of observations in each leaf.
- Handles well missing data
- Like all non-parametric methods, a single tree has a low bias and high variance. It is very sensitive to sample noise.
- Very poor predictive performance.

Regression Trees: IBM Example

Example: We build a Decision tree for IBM excess returns using the 3 F-F factors as predictors. Steps:

1) Start at the root node: We start with 436 observations.

2) First, we split on Market: All observation with **Mkt_RF < 0.02695** go to the "**Yes**" (left) branch (288); the rest (138) go to the "**No**" (right) branch. We compute the average excess return on IBM in each branch: **-.0213** (Yes) & **0.0488** (No).

4) Continue on the Yes branch: We further split on Market: All observation with **Mkt_RF < -0.04905** go to the "**Yes**" branch (44); the rest (144) go to the No branch. The average excess return for each branch are: **-0.07** (Yes) & **-0.0124** (No).

⋮

3) Continue on the No branch (from Step 2): All observations with **Mkt_RF < 0.0687 go** to **Yes** branch (119); the rest to **No** side.

Regression Trees: IBM – Code in R

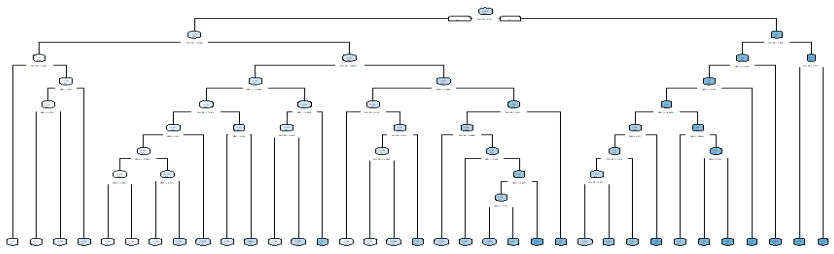
Example (continuation): We use R package rpart.

```
ibm_data <- data.frame(ibm_x, Mkt_RF, SMB, HML)      # Data frame
library(rsample)                                     # data splitting
library(dplyr)                                       # data manipulation
library(rpart)                                       # regression trees
library(rpart.plot)                                  # plotting regression
set.seed(123)

ibm_split <- initial_split(ibm_data, prop = .7)      #split data in training/validation (test)
ibm_train <- training(ibm_split)                     # Training (estimation) sample
ibm_test  <- testing(ibm_split)                      # Validation (test) sample

# Rpart with no complexity penalty (cp = 0) & 10-fold X-validation (xval = 0)
m_ibm_ft <- rpart(  formula = ibm_x ~ .,
            data    = ibm_train,
            method  = "anova",
            control = list(cp = 0, xval = 10))
```

Regression Trees: IBM – Full Tree Plot

Example (continuation):

```
> pred_ibm_ft <- predict(m_ibm_ft, newdata = ibm_test)
> RMSE(pred = pred_ibm_ft, obs = ibm_test$ibm_x)
[1] 0.06952041

> rpart.plot(m_ibm_ft)        # Full tree, with no complexity penalty imposed (cp=0)
```

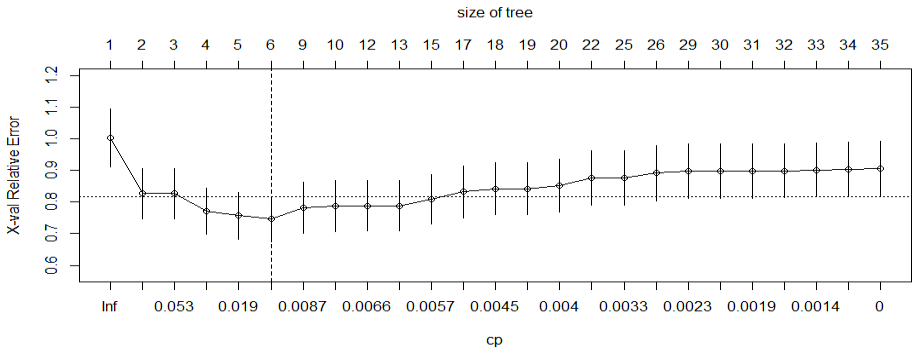


Note: The full tree has 34 internal nodes and 35 leaves.

Regression Trees: IBM – Full Tree CV & Size

Example (continuation):

```
> plotcp(m_ibm_ft)
> abline(v = 6, lty = "dashed")
```



Note: After 6 internal nodes, the CV error does not improve. At each size, the plot includes 1 **CV error SE** in both directions; Breiman et al (1984) suggests using the smallest tree within one CV SE.

Regression Trees: IBM – Complexity Penalty

Example (continuation): Now, we apply a complexity penalty to prune the tree.

```
# rpart applies complexity penalty
m_ibm <- rpart(  formula = ibm_x ~ .,
                 data    = ibm_train,
                 method  = "anova")
rpart.plot(m_ibm)
plotcp(m_ibm)
m_ibm
printcp(m_ibm)                  # Print complexity parameters
```

Regression Trees: IBM – Complexity Penalty

Example (continuation): We use R package rpart.

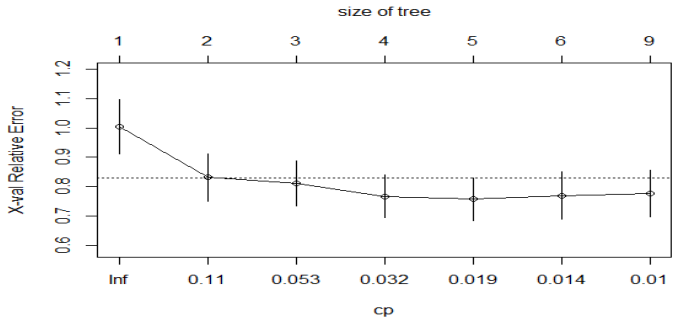
```
> printcp(m_ibm)
Regression tree:
rpart(formula = ibm_x ~ ., data = ibm_train, method = "anova")

Variables actually used in tree construction:
[1] HML    Mkt_RF SMB

Root node error: 2.1835/436 = 0.005008

n= 436

        CP nsplit rel error  xerror     xstd
1 0.219881      0   1.00000 1.00547 0.092778
2 0.057276      1   0.78012 0.82161 0.081079
3 0.048319      2   0.72284 0.80936 0.079805
4 0.020649      3   0.67452 0.76668 0.074995
5 0.017695      4   0.65388 0.74769 0.072682
6 0.010782      5   0.63618 0.75766 0.074625
```

Regression Trees: IBM – Pruned Tree CV & Size

Example (continuation):

```
> pred_ibm_tree <- predict(m_ibm, newdata = ibm_test)
> RMSE(pred = pred_ibm_tree, obs = ibm_test$ibm_x)
[1] 0.06401828
```

```
> plotcp(m_ibm)
```

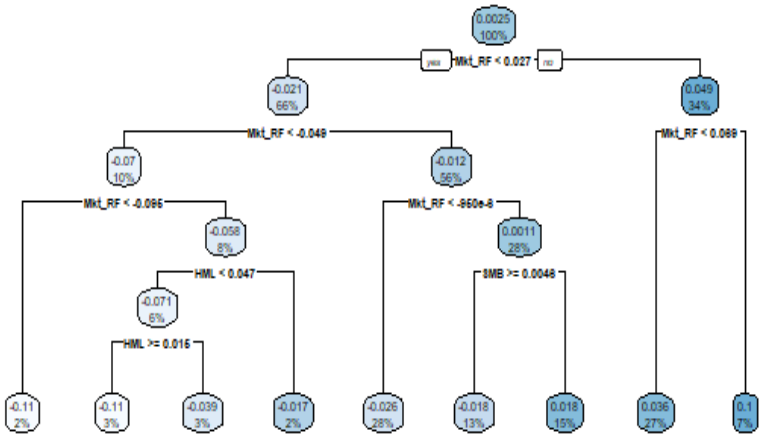


Note: After 5 internal nodes, the CV error does not improve.

Regression Trees: IBM – Pruned Tree Plot

Example (continuation):

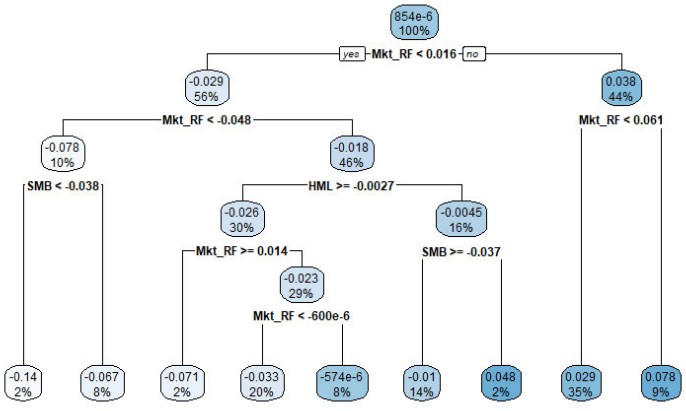
> rpart.plot(m_ibm) #Pruned tree, after complexity penalty imposed



Note: The final tree has 7 internal nodes and 9 leaves.

Regression Trees: IBM – Pruned Tree Plot

Example (continuation): If we set seed (223), we get a different pruned tree.



Note: The final tree has 7 internal nodes and 9 leaves.

Regression Trees: IBM 2 – Change Training Set

Example (continuation):

```
# Split data keeping time series structure (training sample = 70% of initial observations)
s_size <- floor(0.70 * nrow(ibm_data))
split <- split(ibm_data, rep(1:2, each = s_size))
names(split) <- c("train_ts", "test_ts")
ibm_train_ts <- split$`train_ts`
ibm_test_ts <- split$`test_ts`

# Rpart applies complexity penalty
m_ibm_ts <- rpart(
            formula = ibm_x ~ .,
            data    = ibm_train_ts,
            method  = "anova")
printcp(m_ibm_ts)
```

Regression Trees: IBM 2 – Complex Penalty

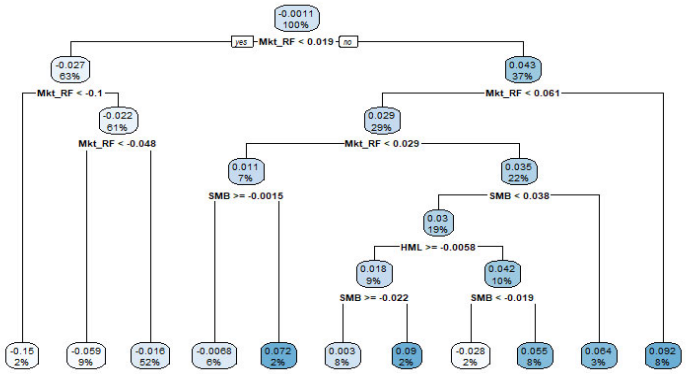
Example (continuation): We use R package rpart.

```
> printcp(m_ibm_ts)

Regression tree:
rpart(formula = ibm_x ~ ., data = ibm_train_ts, method = "anova")

Variables actually used in tree construction:
[1] HML    Mkt_RF SMB

Root node error: 2.4586/436 = 0.005639

n= 436

        CP nsplit rel error  xerror     xstd
1 0.198975      0   1.00000 1.00528 0.090922
2 0.060690      1   0.80103 0.84544 0.078786
3 0.044032      2   0.74034 0.81751 0.074818
4 0.024731      3   0.69630 0.78636 0.074896
5 0.010566      4   0.67157 0.74872 0.070412
6 0.010000     10   0.60818 0.77078 0.072119
```

Regression Trees: IBM 2 – Pruned Tree Plot

Example (continuation):

> rpart.plot(m_ibm_ts) #Pruned tree, after cp imposed



Note: The final tree has 9 internal nodes and 11 leaves.

Regression Trees: IBM 2 – Pruned Tree CV

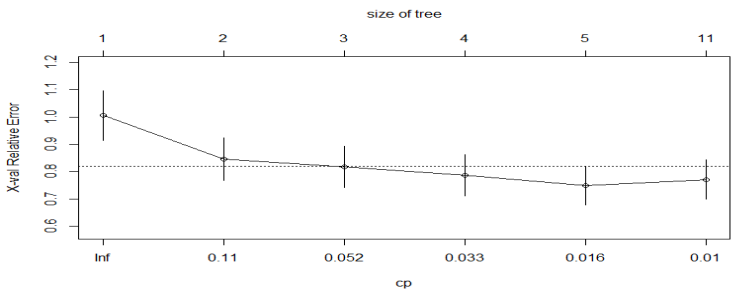
Example (continuation):

> pred_ibm_tree_ts <- predict(m_ibm, newdata = ibm_test_ts)

> RMSE(pred = pred_ibm_tree_ts, obs = ibm_test_ts$ibm_x)

[1] 0.04740236

> plotcp(m_ibm_ts)



Note: After 5 internal nodes, the CV error does not improve.

Decision Trees: Ensemble Methods

• Single tree models show high variance. Small perturbations of the sample can lead to very different fitted functions. Pruning helps, but there are methods than can significantly improve the performance of trees. **Ensemble methods** are one of these methods.

• Main principle behind ensemble methods: A group of "**weak learners**" (predictors show low correlation with dependent variable) can come together to form a "**strong learner**" (predictors show higher correlation with dependent variable) .

• Two popular ensemble method are:

- **Bagging** (**B**oostrap **agg**regat**ing**), proposed by Breiman (1996).
- **Random Forest** (**RF**), proposed by Breiman (2001).

Bagging Trees

• **Bagging** (**B**oostrap **agg**regat**ing**) is a variance-reduction technique, designed to reduced estimation instability. It combines and averages many versions of the same estimator (single trees).

• Averaging across multiple trees reduces the variability of any one tree and reduces overfitting, which improves predictive performance.

• Bagging steps:

1. Create B bootstrap samples from the training data.

2. For each bootstrap sample grow a single, unpruned regression tree. We have B slightly different trees, but with the same distribution.

3. Average individual predictions from each tree to create an overall average predicted value.

Bagging Trees: Bootstrapping

• Bagging summary: From the B bootstrap samples from the training data, we compute B single tree estimators: $\hat{m}^{(b)}(x)$. Then, we average all trees:

$$\hat{m}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{m}^{(b)}(x)$$

As $B \to \infty$, $\hat{m}_{bag}(x) \to \mathrm{E}[\hat{m}(x)]$ under the bootstrap distribution.

• Bagging smooths an estimator with respect to the DGP, not the covariate space. Equivalently:

$$\hat{m}_{bag}(x) \approx E^*[\hat{m}(x)]$$

where E^* is the bootstrap expectation conditional on the original sample. This is **Monte Carlo integration** over sampling noise.

Bagging Trees: Bias-Variance

• Let: $\hat{m}(x) = m(x) + b(x) + \varepsilon(x)$,

where

$b(x) = bias$

$\varepsilon(x) = sampling\ error$

• Bagging changes variance but only modestly affects bias:

$$\begin{aligned}\mathrm{Var}\left[\hat{m}_{bag}(x)\right] = \{&\mathrm{var}\left[\hat{m}^{(1)}\right] + \mathrm{var}\left[\hat{m}^{(2)}\right] + \cdots + \mathrm{var}\left[\hat{m}^{(B)}\right]\}/B^2 \\ &+ 2\{\mathrm{cov}\left[\hat{m}^{(1)}, \hat{m}^{(2)}\right] + \mathrm{cov}\left[\hat{m}^{(1)}, \hat{m}^{(3)}\right] + \cdots + \mathrm{cov}\left[\hat{m}^{(1)}, \hat{m}^{(B)}\right] \\ &+ \mathrm{cov}\left[\hat{m}^{(2)}, \hat{m}^{(3)}\right] + \cdots + \mathrm{cov}\left[\hat{m}^{(2)}, \hat{m}^{(B)}\right] + \ \ldots\ \ldots + \mathrm{cov}\left[\hat{m}^{(B-1)}, \hat{m}^{(B)}\right]\}/B^2\end{aligned}$$

$$\begin{aligned}\Longrightarrow \mathrm{Var}\left[\hat{m}_{bag}(x)\right] &\approx \frac{1}{B}\,\mathrm{var}\left[\hat{m}(x)\right] + 2\frac{B(B-1)}{B^2 * 2}\,\mathrm{cov}\left[\hat{m}^{(b)}, \hat{m}^{(b')}\right] \\ &\approx \frac{1}{B}\,\sigma^2 + \frac{B-1}{B}\,\rho\sigma^2\end{aligned}$$

Bagging: Bias-Variance & Tree Correlation

• If the trees were *i.i.d.*, the ensemble variance would be given by $\frac{\sigma^2}{B}$, But, we do not have independence; we have **tree correlation**. Why? Suppose we have a very strong predictor, then, all B trees will split along similar branches.

• Thus, each tree is identically distributed *(i.d.)*; not *i.i.d.* Tree correlation affects the variance reduction of the average:

$$\text{Var}\left[\hat{m}_{bag}(x)\right] \approx \frac{1}{B}\,\sigma^2 + \frac{B-1}{B}\,\rho\sigma^2$$

• As B increases, the first term is the only one disappearing. Bagging decreases variance, but there is a limit to the variance reduction: $\rho\sigma^2$.

Bagging: Bias-Variance & Tree Correlation

$$\text{Var}\left[\hat{m}_{bag}(x)\right] \approx \frac{1}{B}\,\sigma^2 + \frac{B-1}{B}\,\rho\sigma^2$$

• Then, bagging works well when:

- Base estimator has high variance.
- Bootstrap estimators are not perfectly correlated.

Bagging: Econometric Interpretations

• Bagging can be viewed as a **smoothing operator**: It smooths the estimator over sampling variation, discrete split decisions, and model selection noise

• This is analogous in spirit (but not form) to kernel smoothing, series regularization, and shrinkage estimators.

• Think of bagging as a **non-parametric regularization method**. It stabilizes the estimation without imposing a functional form. It can be compared to:

- LASSO: Regularization shrinks coefficients.
- Ridge: Regularization penalizes the curvature
- Bagging: Sampling-space averaging.

Bagging: Econometric Interpretations

• Bagged estimators are algorithmic objects, not parameter estimates. There is no structural interpretation, no theoretical inference.

• Bagging does not fix inconsistency. If the base learner is inconsistent (consistent), the bagged estimator remains inconsistent (consistent).

• Bagging primarily improves finite-sample risk, not asymptotic rates. This is an important result: Bagging improves prediction but does not introduce identification.

• Summary: Bagging is a variance-reduction technique that improves the finite-sample performance of unstable nonparametric estimators by averaging them over bootstrap perturbations of the data.

Bagging: OOB

• One benefit of bagging is that, on average, a bootstrap sample will contain close to 2/3 of the training data. This leaves about 1/3 of the data out of the bootstrapped sample.

• We call this the **out-of-bag** (**OOB**) sample. We can use the OOB observations for cross-validation of the model.

• Usually, the more trees the better. As we add more trees, we average over more high variance single trees. Thus, we tend to observe big reductions in variance early on in the process.

• Eventually, the reduction in variance stabilizes, which we take as a sign that we have reached an appropriate number of trees to create a stable model. In general, we need less than 60 trees.

Bagging Trees: Pros & Cons

• **Pros**

- Since full trees are used, it is possible to capture complex relations between predictors and dependent variable.
- The average produces a lower variance in the final prediction.
- The bootstrap allows to use a new metric to evaluate the performance of the model: OOB error.

• **Cons**

- Depending on the number of trees used, it can still overfit or underfit (OOB error can be used to select the number of trees.)
- Lack of interpretation.
- Trees tend to be highly correlated.

Bagging: Summary

• Bagging uses an ensemble of trees, each trained on a bootstrap sample of the training set. Then, we average the predictions.

• This average reduces overfitting (variance) and, thus, the accuracy of a prediction over a single tree. But, tree correlation limits bagging benefits.

• Interpretation is not so simple. The logic of the simple decision tree gets lost with the average.

Bagging Trees: Application

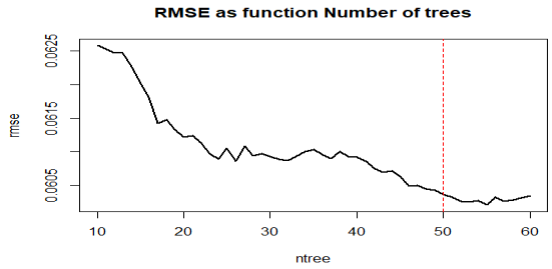
Example: We use bagging to improve the predictions for IBM excess returns using the 3 F-F factors as predictors. We use R package ipred, function bagging.

```
library(ipred)
ntree <- 10:60
rmse <- vector(mode = "numeric", length = length(ntree))      # store OOB RMSEs

for (i in seq_along(ntree)) {
set.seed(123)                                  # set seed to be able to reproduce results

 # perform bagged model (set coob = TRUE to use OOB sample to estimate error.)
  model_ibm_bag <- bagging(
    formula = ibm_x ~ .,
    data    = ibm_train,
    coob    = TRUE,
    nbagg   = ntree[i] )
rmse[i] <- model$err            }            # get OOB error
```

Bagging Trees: Application – RMSE

Example (continuation): We use R package ipred, function bagging.

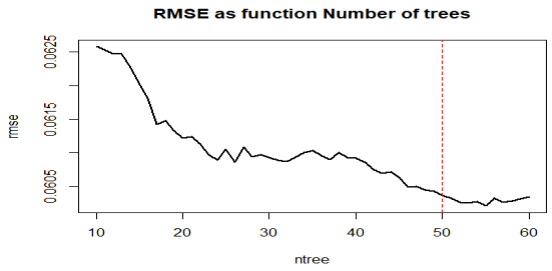
```
> model_ibm_bag

Bagging regression trees with 60 bootstrap replications

Call: bagging.data.frame(formula = ibm_x ~ ., data = ibm_train, coob = TRUE,
    nbagg = ntree[i])

Out-of-bag estimate of root mean squared error:  0.0603
```



Bagging Trees: Application – Predictions

Example (continuation): We use R package ipred, function bagging.

```
plot(ntree, rmse, type = 'l', lwd = 2)
abline(v = 25, col = "red", lty = "dashed")
```




```
> pred_bag <- predict(model_ibm_bag, newdata = ibm_test)        # predictions
> RMSE(pred = pred_bag, obs = ibm_test$ibm_x)
[1] 0.06284899
```

Random Forest Models

• One way to reduce the problem of correlated trees, and, thus, the limit in variance reduction, is to consider only a subset of the predictors at each split.

• If we select randomly the subsets of predictors, at each node, the correlation among trees, ρ, should drastically decrease. This is what **Random Forest** (**RF**) **models** do, as proposed by Breiman (2001).

• The random forest is another **ensemble approach**:

$$\hat{m}_{RF}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{m}^{(b)}(x)$$

where $\hat{m}^{(b)}$ is a regression tree built under randomness. We have two sources of randomness: the usual bootstrap sampling (Bagging) and random selection of predictors.

Random Forest Models: Reduce ρ

• As in bagging, we build B decision trees on bootstrapped training samples each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of k predictors.

Note: If $m = k$, then, this is bagging.

• The prediction of the random forest is made by taking the average of the predictions of the randomly generated (random sample & random predictors) single trees.

Remark: Random forests work because they minimize covariance across trees while keeping bias small.

RF Models: Algorithm

• Steps:

1. Draw a bootstrap sample Z^* of size N from the training data.

2. Grow a random-forest tree with Z^*, by recursively repeating the following steps for each node of the tree, until a minimum node size, say, l, is reached.

2.a Randomly select m variables from the k predictors ($m < k$).

2.b Split tree using the best variable/split-point among the m.

2.c Repeat Steps **2.a** & **2.b** until some size requirement is met.

3. Repeat Steps **1** & **2** B times. Do not prune!

4. Output the ensemble of trees.

Note: We can think of m and l as tuning parameters. It is common to set $m = k/3$ and $l = 5$.

RF Models: Econometric Interpretation

• A RF is a nonparametric, non-smooth estimator of the conditional mean function that trades interpretability for robust, low-variance prediction, achieved by averaging many de-correlated regression trees.

• Consistency. Under weak conditions (tree grows with N, each leaf must contain enough observations, sufficient randomization),

$$\text{as } N \longrightarrow \infty, \quad \widehat{m}_{RF}(x) \longrightarrow m(x)$$

• Random forests are consistent because they behave like adaptive nearest-neighbor estimators once trees become sufficiently fine.

• $\widehat{m}_{RF}(x)$ can be viewed as:

- An adaptive series estimator
- A Kernel estimator with data-driven bandwidths and metrics.

RF Models: Econometric Interpretation

• A crucial theoretical insight is that RF can be written as:

$$\hat{m}_{RF}(x) = \sum_{i=1}^{N} w_{N,i}(x)\, y_i$$

where

$w_{N,i}(x) \geq 0$

$\sum_{i=1}^{N} w_{N,i}(x) = 1$

$w_{N,i}$ depend on how often observation i falls in the same leaf as x.

• This representation links RF to other nonparametric estimators: kernel and series estimators, & k-NN methods

• Not easy to apply CLT since RF is non-smooth, tree structure changes with N, and bias is non-negligible relative to variance. Under many restrictions (honestly), it is possible to get asymptotic normality.

RF Models: Honesty

• A tree is **honest** if the data used to determine the partition, Λ_{split}, are independent of the data used to estimate leaf outcomes, Λ_{est}:

$$\Lambda = \Lambda_{split} \cup \Lambda_{est}$$

• Conditional on the partition, leaf means are unbiased and classical asymptotics become available.

• We can think of an honest trees as local estimators. Let $R_m(x)$ be the leaf (partition) containing x. Then, an honest tree estimator is:

$$\hat{m}_{HT}(x) = \frac{1}{\Lambda_{est} \cap R_m(x)} \sum_{i \in \Lambda_{est} \cap R_m(x)} y_i$$

The partition is fixed from the estimation perspective. This is a **piecewise-constant series estimator**, with data-independent basis functions (conditional on the split sample).

RF Models: Asymptotic Normality

• Honesty alone does not ensure consistency, but it simplifies the conditions. With honesty, bias comes only from leaf size/partition granularity, not from endogenous split selection.

• Consistency still requires that as N grows, the leaf diameter shrinks and the leaf sample size increases.

• Honesty ensures: $\mathrm{E}[\hat{m}_{HT}(x)|\, R_m(x)] = E[Y|X \in R_m(x)]$
This conditional unbiasedness is critical for asymptotic theory.

• CLT: Under honest trees plus regularity conditions (subsampling, regular leaf growth, mild smoothness of $\hat{m}(x)$), we can show:

$$\sqrt{N}\,(\hat{m}_{RF}(x) - \hat{m}(x)) \xrightarrow{d} N(0, \sigma^2(x))$$

RF Models: Pros & Cons

• **Pros:**

- In most applications, very good performance, with very little tuning.
- Works well at capturing non-linearities & interactions.
- Built-in validation set; it can use all the data for estimation (training).
- Can handle samples with missing data and outliers.
- We can compute an **importance index** for each predictor variable. But, importance is just predictive, not structural (no causality!).
- Fast, we can compute all trees at the same time (parallel processing).

• **Cons:**

- It can be very slow on large data sets.
- **Boosting** often produces better results.
- Not easy to interpret.

RF Models: Applications

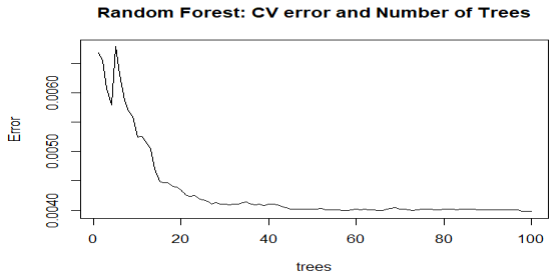
Example: We use RF to improve the predictions for IBM excess returns using the 3 F-F factors as predictors. We use R package *randomForest*.

```
library(randomForest)
rf_ibm <- randomForest(ibm_x ~ ., data = ibm_train, ntree = 100)
> rf_ibm

Call:
 randomForest(formula = ibm_x ~ ., data = ibm_train, ntree = 100)
               Type of random forest: regression
                     Number of trees: 100
No. of variables tried at each split: 1

          Mean of squared residuals: 0.003980812
                    % Var explained: 20.51
```

RF Models: Applications – RMSE

Example (continuation):

```
> plot(rf_ibm, main="Random Forest: CV error and Number of Trees")
```




```
n_rf <- which.min(rf_ibm$mse)                    # number of trees with lowest MSE
> n_rf
[1] 99
> pred_rf <- predict(rf_ibm , newdata = ibm_test)
> RMSE(pred = pred_rf, obs = ibm_test$ibm_x)
[1] 0.06298878
```

RF Models: Applications – Tuning

Example (continuation):

In the randomForest function, we can use tuneRF to tune m, which by default is equal to $k/3$. We start with some m value (mtry), which is increased by stepFactor until some predetermined OOB error is achieved.

```
ibm_preds <- setdiff(names(ibm_train), ibm_x)
set.seed(123)

rf_ibm2 <- tuneRF(
  x          = ibm_train[ibm_preds],
  y          = ibm_train$ibm_x,
  ntreeTry   = 500,
  mtryStart  = 2,              # intial value for m
  stepFactor = 1.5,            # m increase factor in the loop
  improve    = 0.01,           # predetermined error of OOB error to loop
  trace      = FALSE           # not showing real-time progress
)
```

RF Models: Remarks

- When a new input is entered into the system, it is run down all of the trees. The prediction will be either a simple average or weighted average of all of the terminal nodes that are reached.

- With a large number of predictors, the eligible predictor set will be quite different from node to node.

- The greater the inter-tree correlation, the lower the benefit of the RF model. Uncorrelated trees are very important.

- As m goes down, both inter-tree correlation and the strength of individual trees go down. Thus, some way of getting an optimal value for m should be used.

RF Models: Time Series

• A RF has no intrinsic notion of time. Per se, they do not learn dynamic transitions, state evolutions, and/or temporal ordering.

• Time has to be explicitly setup in x_t and/or differences, trends, seasonal factors included should specifically included.

• Training sample cannot be randomly selected. Exchangeability is not a valid assumption. Use forward-chaining (expanding rolling window).

Training procedure follows **recursive forecasting** method:
- Train on data at $t = 1, 2, \dots, T_0$
- Validate on data at $t = T_0, +1, T_0+, \dots, T$
- Continue forward.

RF Models: Time Series

• Since order of sample matters, standard bootstrap does not work. Replace it with blocks or gaps.

• Simplest case: One-step ahead forecast:

$$\hat{y}_{t+1} = \hat{m}_{RF}\left(y_t, y_{t-1}, \dots, y_{t-p}\right)$$

• Multi-step (recursive) forecasting shares same problems as recursive AR models, but with worse error accumulation, due to non-linearities.

• Consistency is achieved under the usual assumptions (stationarity, mixing, correct identification of lag structure). Same conditions used for kernel and series estimators with time series data.

RF Models: Time Series & Honesty

• Honesty is harder but more important in time series: Future outcomes must never influence past predictions; splits exploiting future information create look-ahead bias (**leakage**).

• Practical advice: Use for splits & estimation only past data (rolling or expanding windows also work).

• Honest trees + temporal ordering is analogous to **strict exogeneity** in dynamic panel models.

• Summary: RF is a useful powerful short-horizon forecaster. But, it does not model dynamics directly and has non-standard asymptotics.

Boosting Trees: Introduction

• With the variance-bias trade-off in mind, Bagging and RF focused on variance reduction using complex, low bias, trees. **Boosting** methods, instead, focus on decreasing the bias of simple, low-variance, trees.

• **Boosting** is a general ensemble ML method that builds a series of sequential decision trees, each aimed at correcting the error of the previous one.

• There are several boosting methods. The most popular ones are:

- **Adaptive Boosting** (AdaBoost) uses very small trees, actually stumps (a root with two branches) – see Freund & Schapire (1999).
- **Gradient Boosting** (**GB**) uses deeper trees – see Friedman (2001).

Boosting Trees: Introduction

• Key intuition of Combination of Forecast: We take an ensemble of simple forecasting models and additively combine them into a single, more complex forecasting model.

• Each tree might be a "weak learner" (a poor fit for the data), but a linear combination of all trees can be "strong learner."

• Boosting creates a sequence of trees, each new tree is built to minimize the residual errors– the differences between actual and predicted values– of the previous tree.

• The final prediction aggregates the outcomes from all trees.

Boosting Trees: GB Models

• For regression tasks, Gradient Boosting builds sequential trees. Based on the errors of each tree, we use the predictors to build sequential trees based on the errors. After each (errors-based) tree, we update the prediction of the model.

• A lot of trees will be built. Then, to minimize the effect of each, we can assign a small weight, ω, called **learning rate** or **shrinkage factor**, to adjustments proposed by the new tree. The learning rate is usually a number between 0.01 and 0.3; in R the default rates are 0.1.

• As the number of trees, M, increases, the trees focus on harder-to-predict patterns. Thus, as M increases, the bias decreases & variance increases. The learning rate prevents overfitting by limiting each tree's contribution.

GB Models: Algorithm

• Algorithm idea:

1. From the initial tree, using training data $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$, we get prediction, $\hat{y}_{0,i}$. Then, we compute residuals $e_{0,i} = y_i - \hat{y}_{0,i}$.

2. We add a second tree, built on the $e_{0,i}$'s. Now, the training data is $\{(\boldsymbol{x}_1, e_{0,1}), \ldots, (\boldsymbol{x}_N, e_{0,N})\}$. This second tree produces predictions of residuals, $\hat{e}_{0,i}$, which we use to update the tree's predictions:

$$\hat{y}_{1,i} = \hat{y}_{0,i} + \omega * \hat{e}_{0,i}.$$

3. From **2**, we compute new residuals, $e_{1,i} = y_i - \hat{y}_{1,i}$, which we use to fit a new tree. We use the predictions, $\hat{e}_{1,i}$, to update the tree:

$$\hat{y}_{2,i} = \hat{y}_{1,i} + \omega * \hat{e}_{1,i}$$

4. Continue until some rule stops the loop, for example, a maximum number of trees, usually 100, is fit.

GB Models: Gradient Descent

• With each new tree, the residuals are reduced. For example, after we build tree j, $f_J(\boldsymbol{x}_j)$, the residual is reduced by:

$$e_{j,i} - \omega\, f_J(\boldsymbol{x}_j)$$

• The learning rate, ω, can be tuned. Q: How can we select it?

• In **gradient descent** methods, we use the gradient –i.e., a vector of first derivatives–, multiplied by a step-size, to iteratively update parameters, to minimize a loss function, for example, MSE:

$$MSE = \frac{1}{N} * \sum_i^N (y_i - \hat{y}_i)^2$$

taking first derivatives with respect to the predictions, $\hat{y}_i$:

$$\nabla MSE = \left[\frac{\delta MSE}{\delta \hat{y}_1}, \ldots., \frac{\delta MSE}{\delta \hat{y}_N}\right]$$

GB Models: Gradient Descent

• Then, the gradient:

$$\nabla MSE = \left[\frac{\delta MSE}{\delta \hat{y}_1}, \dots, \frac{\delta MSE}{\delta \hat{y}_N}\right]$$
$$= -2 * [(y_1 - \hat{y}_1), \dots, (y_N - \hat{y}_N)]$$
$$= -2 * [e_1, \dots, e_N]$$

• Now, we update the predictions, using the step-size parameter, ω:

$$\hat{y}_i = \hat{y}_i + \omega * e_i \qquad \text{for } i = 1, 2, \dots, N.$$

• In our context, minimizing MSE with respect to the predictors $\hat{y}_i$'s is not very interesting. We get a sequence of $\{\hat{y}_i^1, \dots, \hat{y}_i^i, \dots\}$, which does not produce a model. The predictions in the sequences do not depend on the predictors.

GB Models: Gradient Descent

• Then, we modify the update step: Instead of using the gradient –i.e., the residuals –, we use an *approximation* of the gradient that depends on the predictors:

$$\hat{y}_i = \hat{y}_i + \omega * \hat{e}_i(\boldsymbol{x}_i), \qquad \text{for } i = 1, 2, \dots, N.$$

• Then, we think of GB as a form of gradient descent.

Technical Note: GB is descending in a space of models (functions) relating $\boldsymbol{x}_i$ to y_i.

• We know a lot about gradient descent methods. We can apply what we know to GB, for example, how to select ω.

GB Models: Time Series

• Like all regular trees, GB has no notion of time, time enters through covariates/features, for example, with seasonal effects, regime dummies or lagged covariates: $\{\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \dots, \boldsymbol{x}_{t-p}\}$.

• GB for time series is a high-dimensional non-linear AR model:

$$y_t = f(y_{t-1}, y_{t-2}, \dots., \boldsymbol{x}_{t-1}, \dots, \boldsymbol{D}_t) + \varepsilon_t.$$

• GB still minimizes a loss function,

$$MSE = \frac{1}{T} * \sum_i^T (y_i - \widehat{m}_{GB}(\boldsymbol{x}_t))^2$$

• Consistency relies on stationarity, mixing conditions, & model complexity controls (depth, learning rate, early stopping). Similar to the conditions for nonparametric regression under dependence.

GB Models: Time Series

• As mentioned before, as the number of trees, M, increases, the bias decreases & variance increases. With dependence, variance inflates much faster than in *i.i.d.* data.

• Then, a small ω is critical: It controls overfitting to serial noise.

• Tree depth reveals the order of interactions

- Depth 1–2: Usually, we have an additive AR structure
- Larger depth: Non-linear regime interactions

• Asymptotics: Under stationarity, strong mixing, fixed depth and $M \to \infty$, $\omega \to 0$, GB behaves like a **sieve estimator** of $E[y_t|\mathcal{F}_{t-1}]$, where $\mathcal{F}_{t-1}$ is the sigma-field of the information up to time $t-1$.

GB Models: Asymptotics & Sieve Estimators

• Review: A sieve estimator approximates an infinite-dimensional object (for us, a conditional mean) by estimating it within a sequence of finite-dimensional spaces whose dimension grows with T.

Formally, let the true object be $f_o \in \mathcal{F}$ (infinite-dimensional).

Define a sequence of sieve spaces, $\mathcal{F}_{K_n} \subset \mathcal{F}, \dim(\mathcal{F}_{K_T}) = K_T \to \infty$

.The estimator is $\hat{f}_T = \arg\min_{f \subset \mathcal{F}_{K_T}} \frac{1}{T} * \sum_i^T L(y_i, f(\boldsymbol{x}_i))$

• **Consistency**: Under standard conditions (identification, compactness, regularity of loss function). But, we require,

$$\frac{K_T}{T} \to 0$$

That is, the sieve must expand (to remove bias), but slowly enough to let the data estimate its parameters: "Sieve does not grow **too fast**."

GB Models: Asymptotics & Sieve Estimators

• **Asymptotic Normality**: The entire function estimator $\hat{f}_T$ is not asymptotically normal. But, for smooth, low (finite)-dimensional functionals, under mild regularity conditions, we get asymptotic normality.

• In time-series settings, dependence slows rates but does not change the qualitative conclusions.

• Pointwise inference for $\hat{f}_o(x_o)$ gets complicated, since asymptotic normality may fail, even with a bias correction. C.I. are non-Gaussian.

Remark: Formal theory is incomplete, but similar to non-parametrics with dependent observations.

GB Models: Algorithm – Remarks

• The name is derived from the **gradient descent** methods, the backbone of minimization algorithms.

• In our case, the gradient used to update the tree is the residual and the step-size is the learning rate.

• GB predictions are very sensitive to the choice of two (hyper-) parameters: learning rate & number of trees. A lot of tuning is usually required. A small (large) learning rate usually required a large (small) number of trees to minimize the loss function (*RSS*).

• In general, shorter trees are preferred. It is common to restrict the depth in a trees to 4-6.

Rule of thumb: Use shallow trees + many iterations + early stopping

GB Models: Algorithm – Remarks

• A large number of trees can lead to overfitting. Setting a maximum number of trees is good practice.

• Following the Bagging idea, there is a variation of GB: **Stochastic Boosting**, where for each new tree we use a subsample of the training data (with or without replacement. Check R's package *gbm*).

• GBM tends to do better than RF, but, it needs more tuning. It is not as good **out-of-the-box**.

• Tremendously popular in ML competitions: credit fraud detection, sales forecasting, ads clicking rates, search rankings of webpages, etc.

GB Models: Application

Example: We use GB to improve the predictions for IBM excess returns using the 3 F-F factors as predictors. We use R package *gbm*.

```
set.seed(123)                                   # for reproducibility
ibm_gbm <- gbm(
  formula = ibm_x ~ .,
  data = ibm_train,
  distribution = "gaussian",                    # SSE loss function
  n.trees = 500,
  shrinkage = 0.1,                              # learning rate
  interaction.depth = 3,
  n.minobsinnode = 10,
  cv.folds = 10)
> best <- which.min(ibm_gbm$cv.error)           # Picks "optimal" n.trees (best MSE)
> sqrt(ibm_gbm$cv.error[best])                  # get MSE and compute RMSE
[1] 0.06006266
```

Artificial Neural Networks: Introduction

• A **neural network** (**NN**) is a computational model inspired by the brain that maps inputs to outputs using layers of interconnected units called neurons. Each neuron computes a weighted sum of its inputs, applies a nonlinear activation, and passes the result forward.

• An **artificial neural network** (**ANN**) is a flexible model that approximates an unknown, potentially non-linear, function:

$$y = m(\boldsymbol{x}) + \varepsilon$$

using linear transformations and non-linear activation functions.

• From an econometric perspective, ANNs are best understood as **non-linear sieve estimators** or **high-dimensional series regression** with data-driven basis functions.

Artificial Neural Networks: Introduction

• A standard **feed-forward** (information & processing moves forward, with no feedback) ANN defines the function:

$$\hat{y} = m(\boldsymbol{x}, \boldsymbol{\omega}).$$

where the parameters $\boldsymbol{\omega}$ are estimated by minimizing a loss function, typically MSE-like, via gradient-based optimization.

Note: The constant in the vector $\boldsymbol{\omega}$ is referred as "bias", the rest of the elements in $\boldsymbol{\omega}$ are referred as weights.

• Many types of NNs: Multilayer perceprton (MLP), Recurrent NN (RNN), Convolutional NN (CNN), Long/Short-term memory NN (LSTM), Generative adversarial network (GAN), etc.

ANN: Layers

• The ANN architecture can be very simple with a single processing neuron, a single "**hidden layer**," or include several hidden layers.

• A single neuron computes:

$$h = \sigma\,(\boldsymbol{\omega}'\boldsymbol{x} + \boldsymbol{b}),$$

where $\boldsymbol{x}$ is a vector of inputs (covariates), $\boldsymbol{\omega}$ is a vector of weights, $\boldsymbol{b}$ is the "**bias**," and σ is a non-linear function, usually referred as **activation function** (for example, a sigmoid function). In a linear model, we have $\boldsymbol{h = \omega'x + b}$.

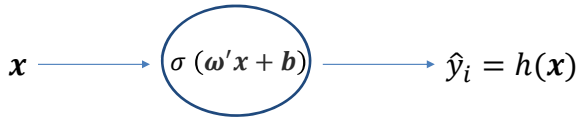
• A network with multiple L layers computes:

$$h^{(l)} = \sigma\left(\boldsymbol{W}^{(l-1)}h^{(l-1)} + b^{(l)}\right) \qquad l = 1, 2, \ldots, L$$

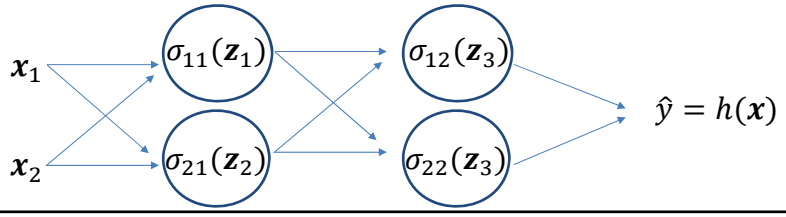
which is a nested function: Each layer learns from the previous layer. A lot of non-linear interactions.

ANN: Layers – Illustration

• A single neuron i receives information (data, exogenous input), transform it and produces an output: $\hat{y}_i = h(\boldsymbol{x})$.

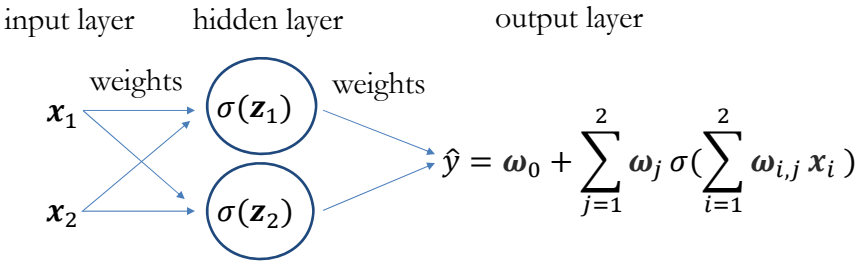


• Multi-layer network receives data in many nodes, transform the data in the hidden layers, and combines the transformations of data to produces an output: $\hat{y} = h(\boldsymbol{x})$.



ANN: Simple Example

Example: We have two vectors of inputs ($\boldsymbol{x}_1$ & $\boldsymbol{x}_2$) and two neurons. The neuron nodes combine the data ($\boldsymbol{z}_1$ & $\boldsymbol{z}_2$). The output is a combination of the nodes' outputs: $\hat{y} = h\big(\sigma(\boldsymbol{z}_1), \sigma(\boldsymbol{z}_2)\big)$.



where $\boldsymbol{z}_1 = \boldsymbol{\omega}_1'\boldsymbol{x}$ & $\boldsymbol{z}_2 = \boldsymbol{\omega}_2'\boldsymbol{x}$, where $\boldsymbol{\omega}_i$ is a vector of weights.

Note: This is a "shallow" NN, with only one hidden layer (one layer between input and output).

ANN: Activation Functions

• An **activation function** is a non-linear map applied element-by-element to linear indices, transforming a linear combination of inputs into a non-linear feature.

• Without activation functions, the entire network collapses to a single linear regression, regardless of depth.

• Suppose we stack linear layers only:

$$h^{(L)} = \boldsymbol{\omega}^{(L)}\boldsymbol{\omega}^{(L-1)} \dots \boldsymbol{\omega}^{(1)}\, \boldsymbol{x} \;\; h(\boldsymbol{x}) = \sigma\left(z^{(1)}\right)$$

This is still linear in $\boldsymbol{x}$.

• Activation functions introduce non-linearity, allowing ANNs to act as: non-linear sieve estimators, adaptive basis expansions, universal approximators of unknown functions.

ANN: Activation Functions – Core Ones

• Popular activation functions are:

- Logistic (Sigmoid): $\sigma(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic Tangent (tanh): $\sigma(z) = \tanh(z)$
- Rectified Linear Unit (ReLU): $\sigma(z) = \max\{0, z\}$
- Leaky ReLU: $\sigma(z) = \max\{\alpha z, z\}, \quad \alpha > 0$

• ReLu is very popular: A ReLU NN approximates functions as piecewise linear splines with data-driven knots. This connects with regression splines and series regressions.

• Leaky ReLu can avoid "dead nodes."

ANN: Activation Functions – Summary

- Activation functions introduce nonlinearities.
- They define the ANN's approximation space.
- ReLU ≈ adaptive spline regression.
- Output activations define the statistical model.
- Activations affect optimization, not identification.

Remark: Think of activation functions as endogenously learned transformations of covariates, not as biologically inspired components.

ANN: Approximation Theory View

- A key result, similar in spirit to the Kolmogorow-Arnold representation theorem, is the **Universal Approximation Theorem**:

A feed-forward network with a suitable $\sigma(z)$ and one hidden layer can approximate any continuous function on a compact set arbitrarily well, given enough neurons.

Note: Theorem holds for ReLU, sigmoid, tanh.

- From an economics perspective, ANN are comparable to series estimators (splines, polynomials), but **basis functions** are learned, not pre-specified.
- In practice, deeper networks may achieve the same approximation accuracy with exponentially fewer parameters, which can mitigate the curse-of-dimensionality.

ANN: Estimation

• ANNs are estimated by minimizing a loss function, $\mathcal{L}$:

$$\mathcal{L} = \min_{\boldsymbol{\omega}} \frac{1}{N} * \sum_i^N l(y_i, h\,(\boldsymbol{\omega}, \boldsymbol{x}))$$

for example, an MSE:

$$l(y_i, h\,(\boldsymbol{\omega}, \boldsymbol{x})) = (y_i - \hat{y}_i)^2$$

Note: Other loss functions: Quantile loss, log-loss (or cross-entropy).

• ANN training estimation is **M-estimation**. It is done numerically.

• The estimated parameters, the **weights**, $\boldsymbol{\omega}$, determine the importance of each connection between two neurons. Each connection between two neurons has a weight that indicates the influence of that neuron on the other neuron.

ANN: Estimation – SGD

• The minimization problem is usually solved via **stochastic gradient descent** (**SGD**).

• Since the objective function contains a huge number of terms (usually, many summations and combinations over training data points), gradient computation becomes prohibitively expensive.

• At each iteration, we do not compute the exact gradient $\nabla\mathcal{L}$, but an **unbiased**, computationally "cheap," estimator $\tilde{\nabla}\mathcal{L}$. That is,

$$\mathrm{E}\left[\tilde{\nabla}\mathcal{L}(\boldsymbol{\omega}_j)\right] = \nabla\mathcal{L}(\boldsymbol{\omega}_j).$$

• Then, the SGD algorithm at iteration j:

$$\boldsymbol{\omega}_{j+1} = \boldsymbol{\omega}_j + \lambda\,\tilde{\nabla}\mathcal{L}(\boldsymbol{\omega}_j)$$

ANN: Estimation – SGD

• Different ways to compute $\tilde{\nabla}\mathcal{L}$. A common choice is to replace the gradient of $\mathcal{L}$ with the average gradient of a uniformly randomly subset of training data (typically, sampled without replacement), called a **mini-batch**. This estimate is called as **mini-batch SGD**.

• The random mini-batch is usually easy to do by using a random re-arrangement of the dataset, and selecting the first B examples as the mini-batch (popular choices for B are 32 or 64).

• Note: The use of stochastic methods, as opposed to using an exact methods, has been observed empirically to lead to better solutions in ML settings (fitting well training data tends to "overfit"). Using SGD has been linked with a reduction in overfitting and increased success on fitting unseen data. This is likely due to the presence of noise.

ANN: Estimation – SGD & Backpropagation

• We use **backpropagation** to compute $\nabla\mathcal{L}$, which is an efficient algorithm for computing gradients via chain rule, exploiting the layered structured in NN.

Example: Consider a simple network:

$$z^{(1)} = \boldsymbol{\omega}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$$
$$h = \sigma\left(z^{(1)}\right)$$
$$z^{(2)} = \boldsymbol{\omega}^{(2)}h + \boldsymbol{b}^{(2)} \qquad \Rightarrow \hat{y} = z^{(2)}$$
$$\mathcal{L} = l(y, \hat{y})$$

Using Backward Pass, we compute: $\nabla\mathcal{L}(\boldsymbol{\omega}_j) = [\frac{\delta\mathcal{L}}{\delta\boldsymbol{\omega}^{(1)}}, \frac{\delta\mathcal{L}}{\delta\boldsymbol{\omega}^{(2)}}]$

Step 1: Output Layer Gradient:

$$\frac{\delta\mathcal{L}}{\delta\boldsymbol{\omega}^{(2)}} = \frac{\delta\mathcal{L}}{\delta z^{(2)}} * h'$$

ANN: Estimation – SGD & Backpropagation

Example (continuation):

Step 1: Output Layer Gradient:

$$\frac{\delta \mathcal{L}}{\delta \boldsymbol{\omega}^{(2)}} = \frac{\delta \mathcal{L}}{\delta z^{(2)}} * h'$$

Step 2: Hidden Layer Gradient:

$$\frac{\delta \mathcal{L}}{\delta h} = \frac{\delta \mathcal{L}}{\delta z^{(2)}} \boldsymbol{\omega}^{(2)\prime}$$

Then, by chain rule:

$$\frac{\delta \mathcal{L}}{\delta z^{(1)}} = \frac{\delta \mathcal{L}}{\delta h} \cdot \sigma'\left(z^{(1)}\right) \qquad (. = \text{elementwise multiplication})$$

Step 3: First-Layer Gradient

$$\frac{\delta \mathcal{L}}{\delta \boldsymbol{\omega}^{(1)}} = \frac{\delta \mathcal{L}}{\delta z^{(1)}} \boldsymbol{x}'$$

• This is just structured Jacobian-vector product calculations.

ANN: Estimation – SGD & Backpropagation

• Naively computing each partial derivative separately would cost exponential time.

• Q: Why is backpropagation computationally efficient:

- Traverses the network once forward.
- Then, once backward, updates parameters locally.
- Gradient computation has the same order of complexity as function evaluation.

• We have a non-convex problem: Many local minima. In general, SGD shows good empirical performance.

Remark: There is nothing intrinsically "neural" about backpropagation; it is pure calculus.

ANN: Regularization & Overfitting

• ANNs are heavily parameterized, in-sample (training) **overfitting** is common. Thus, regularization (restrictions) is crucial.

• Standard Techniques:
- **Weight decay** (L2 penalty), which produces ridge-like shrinkage.
- **L1 penalty**, which produces LASSO-like sparsity.
- **Early stopping**, which creates an implicit regularization.
- **Dropout**, which produces stochastic model averaging.

• Regularization controls the effective degrees of freedom.

• Overparameterized networks may still generalize due to implicit regularization from SGD.

ANN: Summary

• ANNs are functions.

• Formed by connected artificial neurons. (Nodes are neurons.)

• Input layers receive data; output layers produce the final result of network processing. In our case, output layers produce a prediction.

• Hidden layers do all the computations.

• UAT gives some theoretical background for method.

• Highly non-linear model, usually estimated by SGD, imposing restrictions.

• UAT gives some theoretical background for method.

ANN: In Practice

• Choice of loss function is driven by the problem at hand:
- Regression (continuous data): MSE (Non-linear LS).
- Binary Choice: Cross-entropy loss (Logit MLE).

• Optimization almost always done with SGD. Parameters chosen to minimize estimated out-of-sample risk, not an asymptotic MSE.

• Use a small, simple architecture. For example, ANN with 1-3 hidden layers (2 usually works well) and 10-200 neurons per layer.

• Many activation functions, but ReLU is the most common choice.

• Standardize all inputs. It tends to improve estimation. Always done.

ANN: In Practice

• Lots of attention given to bias/variance trade-off, stability across random seeds, and sensitivity to tuning

• Not a lot of attention is given to interpretation, asymptotics, or diagnostic testing.

• From an econometric perspective:
- ANNs are adaptive sieve estimators
- SGD + regularization implicitly controls complexity
- Cross-validation replaces asymptotic optimality

• Practitioners do ANNs the way applied economists do non-parametrics: Tune aggressively, validate carefully, and let theory justify afterward.

ANN: In Practice for Finance

• Q: How many layers, how many units (neurons) per layer?

- For monthly returns and a small k (predictors, say, 5 FF factors), 1 layer and 8-16 units is a good starting point. Very rare to see more than 32. This setup usually is enough to capture non-linearities. A typical rule of thumb for a single hidden layer:

$$\text{units} \sim \min\{(5\text{-}10) * k, \sqrt{N}, (20\text{-}50)\}$$

- If regime change is suspected, add a second hidden layer

• Each hidden layer unit adds parameter ($\approx 2^K$, K: # of units).

• In macro/asset pricing, N is usually small to moderate. The main risk is overfitting. We get into diminishing returns very quickly once the sieve dimension grows. Recall that asymptotically, $K_N/N \to 0$!

ANN: In Practice for Finance

• With time series –i.e., dependence-, the effective sample size gets reduced. Then, the true upper bound is K is lower. Common to think of the effective as $N/2$.

• In practice (and in papers):

1. Start small: 4–8 units
2. Increase gradually: 8 to 16, 16 to 32.
3. Stop when validation loss (MSE, MAE) stops improving and/or Weights become unstable.

Very rare to see benefits past one hidden layer with ~16 units for FF-style problems.

• Overfitting sign: Validation loss increases, while training loss falls.

ANN: Application

Example: We use ANN to improve the predictions for IBM excess returns using the 3 F-F factors as predictors. We use R package *keras3*.

```
ibm_data <- data.frame( ibm_x, Mkt_RF, SMB, HML)

## Split Sample
library(rsample)
set.seed(123)
split <- initial_time_split(ibm_data, prop = 0.7)
train <- training(split)
test  <- testing(split)

## ANNs require scaling.
x_mean <- colMeans(train[, -1])
x_sd   <- apply(train[, -1], 2, sd)
scale_x <- function(x) {  sweep(sweep(x, 2, x_mean), 2, x_sd, "/")}
X_train <- scale_x(as.matrix(train[, -1]))
X_test  <- scale_x(as.matrix(test[, -1]))
```

ANN: Application

Example (continuation):

```
y_train <- train$ibm_x
y_test  <- test$ibm_x

## Simple ANN Specification (FF3-Style)
# Depth = 1 hidden layer; Width = 8; Linear output → conditional mean
library(keras3)

model <- keras_model_sequential()      # Creates an empty sequential model

model <- layer_dense(model,
  units = 8,
  activation = "relu",
  input_shape = ncol(X_train))

model <- layer_dense(model,
  units = 1,
  activation = "linear")
```

ANN: Application

Example (continuation):

```
## Train Model
history <- fit(model,
   X_train, y_train,
   epochs = 200,
   batch_size = 32,
   validation_split = 0.2,
   verbose = 0)
## OOS Evaluation
pred <- as.numeric(predict(model, X_test))
oos_mse <- mean((y_test - as.numeric(pred))^2)
> oos_mse
[1] 0.002948608
pred_lm <- predict(fit_ff3, test)          # FF3 linear model.
> mean((y_test - pred_lm)^2)
[1] 0.002716747
```

ANN vs RF: Summary

- A RF approximates a function as a piecewise constant, defined on a data-adaptive partition of the covariate space aggregated across many randomized partitions.

  Econometric view: Close to k-NN estimators. Excellent at capturing kinks, thresholds, & interactions. Not good at extrapolating outside data support. Efficient with small data sets.

- An ANN approximates a function as a globally defined composition of linear maps and non-linear activations, often piecewise linear but globally smooth (ReLU).

  Econometric view: Close to non-linear series regression. Excellent at capturing smooth non-linearities & interactions. Very good at extrapolations. Needs large data sets.

ANN vs RF: In Practice

- Both are flexible nonparametric estimators, used primarily for prediction, but they are used **very differently** in practice.

- Random Forests Practitioners view RFs as "strong defaults," with low-tuning, that produce stable predictors, robust to bad scaling and weird distributions.

Practitioner's View: Let the algorithm discover interactions via trees.

- Neural Networks Practitioners view ANNs as high-capacity function approximators, that produce sensitive but extremely flexible powerful predictors, especially when data are large or structure is complex.

Practitioner's View: Learn a smooth mapping via gradient descent.

ANN vs RF: In Practice

- RFs are preferred when:
- Sample is small
- Covariates are mixed/discrete
- Robustness matters

- ANNs are preferred when:
- Sample is large
- Covariates are continuous/high-dimensional
- Smoothness is plausible.

- Practitioners say: "Try RF first; use ANN only if needed."

ANN: Time Series

• Similar to traditional tree models, traditional neural networks (like MLPs) treat observations as independent; the models lack memory. Then, for forecasting time series, new models are needed.

• **Long Short-Term Memory Networks** (**LSTM**) is a specialized **Recurrent Neural Network** (**RNN**) with a memory mechanism. We have **gates** (input, forget, output), designed to learn dependencies.

• LSTMs work very well for long-range temporal patterns. They are the most popular ANN model for time series forecasting.

Note: RNNs are designed for sequential data, with a hidden state, h_t, that evolves over time. Good idea, but they do not work well with time series.

LSTM: Time Series

• An LSTM is a type of Recurrent Neural Network (RNN) with a special internal structure called a memory cell.

• An LSTM processes a time series one time step at a time, while maintaining a "**memory state**" that decides: What to remember, what to forget, and what to output.

• This allows it to learn both short-term and long-term dependencies.

• Each time step t:

- Receives input $\boldsymbol{x_t}$ -e.g., lagged observations or exogenous variables.
- Keeps a cell state, c_t (long-term memory.)
- Produces a hidden state h_t (short-term memory).

LSTM: Cells & Gates

• Each LSTM cell has:

- **Cell state**, c_t, for long-term memory.
- **Hidden state**, h_t, for short-term output.
- Three gates controlling information flow: **Forget**, **Input** & **Output**.

• An LSTM has three gates that control information flow.

(1) **Forget Gate**: It decides what to discard from previous steps, f_t:

$$f_t = \sigma\left(\boldsymbol{x}_t, h_{t-1}, \boldsymbol{\theta}_f\right) \qquad f_t \in [0,1]$$

(2) **Input Gate**: It decides what new information to store, i_t, $\widetilde{c_t}$:

$$i_t = \sigma\left(\boldsymbol{x}_t, h_{t-1}, \boldsymbol{\theta}_i\right) \qquad i_t \in [0,1]$$

$$\widetilde{c_t} = g\left(\boldsymbol{x}_t, h_{t-1}, \boldsymbol{\theta}_c\right) \qquad \text{- candidate memory}$$

(3) Cell state update: $c_t = f_t \,.\, c_{t-1} + i_t \,.\, \widetilde{c_t}$.

(4) **Output Gate**: $\hat{y}_t = \boldsymbol{\omega}_y\, h_t + \boldsymbol{b}_y$ where $h_t = r(\boldsymbol{x}_t, h_{t-1}, c_t, \boldsymbol{\theta}_o)$

LSTM: Recursive Algorithm

• LSTM computations involve a recursive algorithm.

(0) **Data** : $\boldsymbol{x}_T = \{Y_T, Y_{T-1}, \ldots, Y_1\}$

We use a p lookback window. Thus,

$$\boldsymbol{x}_t = \{Y_{t-1}, Y_{t-2}, \ldots, Y_{t-p}\}$$

Set $h_t = 0$ & $c_t = 0$

for $t = p+1$ to T

(1) **Forget Gate**: It produces $f_t \in [0,1]$ (0 = forget completely, 1 = keep everything).

$$f_t = \sigma\left(\boldsymbol{\omega}_f \boldsymbol{x}_t + \boldsymbol{u}_f h_{t-1} + \boldsymbol{b}_f\right) \qquad \sigma = \text{logistic}$$

(2) **Input Gate**: It produces, $i_t \in [0,1]$ which weights proposed, $\widetilde{c_t}$

$$i_t = \sigma\left(\boldsymbol{\omega}_i \boldsymbol{x}_t + \boldsymbol{u}_i h_{t-1} + \boldsymbol{b}_i\right) \qquad \sigma = \text{logistic}$$

$$\widetilde{c_t} = g\left(\boldsymbol{\omega}_c \boldsymbol{x}_t + \boldsymbol{u}_c h_{t-1} + \boldsymbol{b}_c\right) \qquad g = \tanh$$

LSTM: Recursive Algorithm

- LSTM computations (continuation):

(3) **Cell state update**: It updates long-term memory:

$$c_t = f_t \,.\, c_{t-1} \;+\; i_t \,.\, \tilde{c}_t.$$

(4) **Output Gate**: It sets the part of the memory that becomes output, using $o_t \in [0,1]$ to weight $g(c_t)$.

$$o_t = \sigma\,(\boldsymbol{\omega}_o \boldsymbol{x}_t + \boldsymbol{u}_o h_{t-1} + \boldsymbol{b}_o) \qquad \sigma = \text{logistic}$$
$$h_t = o_t \,.\, g(c_t)$$

(5) **Forecast**:

$$\hat{y}_t = \boldsymbol{\omega}_y\, h_t + \boldsymbol{b}_y$$

- We minimize $\quad \mathcal{L} = \min_{\boldsymbol{\omega},\boldsymbol{u},\boldsymbol{b}} \frac{1}{T-p} * \sum_{t=p+1}^{T} (y_t - \hat{y}_t)^2$

LSTM: Input & Output Gates

Example: At t, we have $c_{t-1} = 10$. From current inputs, we get: $f_t = 0.7,\ i_t = 0.3, \tilde{c}_t = 4, o_t = 0.6$. We want to compute h_t.

Step 1 – **Forget Gate**

$f_t \,.\, c_{t-1} = 0.7 * 10 = 7.$ $\quad \Rightarrow$ We keep 70% of past memory

Step 2 – **Input Gate**

$i_t \,.\, \tilde{c}_t = 0.3 * 4 = 1.2.$ $\quad \Rightarrow$ Only 30% of new info into c_t

Step 3 – **Update Cell state**

$c_t = f_t \,.\, c_{t-1} \;+\; i_t \,.\, \tilde{c}_t = 7 + 1.2 = 8.2.$ $\quad \Rightarrow$ New memory

Step 4 – **Output Gate**

$$h_t = o_t \,.\, tanh(c_t) = 0.6 * tanh(8.2) \approx 0.6$$

Note: Step 4 decides how much of the stored memory, c_t, is revealed to the outside world, while the rest stays protected for future use.

LSTM: Comparison with Time Series

- In general, we prefer LSTM to ARIMA models when we have complex patterns in situations that long-term dependencies matter.

- We need a lot of historical data to estimate LSTM.

- Below, we compare both approaches:

Feature	ARIMA	LSTM
Linearity	Assumes linearity	Non-linear
Memory	Short (lags)	Long-term
Seasonality	Manual	Learned
Interpretability	High	Low
Data Needs	Small	Medium–large

LSTM: Pros and Cons

- **Pros**

- Handles well non-linearities
- Captures long memory patterns
- Works well with noisy data
- Admits covariates

- **Cons**

- Data-hungry
- Harder to interpret
- Sensitive to hyper-parameters
- Computationally expensive