Notes on Machine Learning

Qiao Zhou[*]

August 11, 2024

Contents

[*]Haas School of Business, University of California, Berkeley. Email address: qiao_zhou@mfe.berkeley.edu.

1 Introduction

1.1 How much ML is needed?

Unless you are hired for ML Engineering or research scientist, most data scientists are hired to solve business problems, but blindly throw complicated neural networks on top of dirty data[3].

1.2 Human experience vs Power of computation

Is more inputs from sensible human knowledge always good? Not always according to Rich Sutton: *'The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation.'* [**?**].

1.3 Book Recommendations

1. James, G., Witten, D., Hastie, T., Tibshirani, R. (2021). An Introduction to Statistical Learning: With Applications in R. United States: Springer US.

1.4 Learning Resource

- Course on Tree-based model from DataCamp. **Several sections of this document cite contents from this course**.

1.5 Credit and Rights

This document is used for personal learning only. In writing this notes, I referred to many external publications, online resources, courses, books. I acknowledge that not all references are properly cited, and please reach out to me if you believe your copy rights are being violated and I'll add proper citation or remove them from this document and exclude them from future reference without explicit author consent.

2 Primers

2.1 Linear Algebra

2.1.1 Eigenvalues and Eigenvectors

For some nxn matrix A, x is an eigenvector of A if $Ax = \lambda x$, where λ is a scalar called eigenvalue. A can represent a linear transformation and, when applied to vector x, results in another vector called eigenvector, which has the same direction as x and with magnitude scaled (merely elongated or shrunk) by a factor of λ known as eigenvalue.

The decomposition of a square matrix A into its eigenvectors and eigenvalues is called an *eigendecomposition.*

$$A = QAQ^{-1}$$

where Q is the square n $\times$ n matrix whose ith column is the eigenvector q_i of A, and Λ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{ii} = \lambda_i$.

2.2 Singular Value Decomposition (SVD)

2.3 Model Evaluation

Model evaluation is the process of evaluating how well a model performs on the test set after it's being trained on the training set. The objective of model fitting is to use $\hat{f}$ to approximate the true underlying process f.

2.3.1 Overfitting and Underfitting

Overfitting[1]. When a model overfits the training set, its predictive power on unseen datasets is pretty low. This is illustrated by the predictions of the decision tree regressor shown here in red. The model clearly memorized the noise present in the training set. Such model achieves a low training set error and a high test set error.

Underfitting[1]. When a model underfits the data, like the regression tree whose predictions are shown here in red, the training set error is roughly equal to the test set error. However, both errors are relatively high. Now the trained model isn't flexible enough to capture the complex dependency between features and labels. In analogy, it's like teaching calculus to a 3-year old. The child does not have the required mental abstraction level that enables him to understand calculus.

2.3.2 Generalization Error, Bias, and Variance

Generalization error of $\hat{f}$ tells us whether $\hat{f}$ can generalize well on unseen data. It can be decomposed into three components[1]:

$$GE = \textbf{bias}^2 + \textbf{variance} + \textbf{irreducible error}$$

, where irreducible error is error contribution from noise.

Assuming a target variable y that we want to predict. Given prediction $\hat{y}$, we can decompose the error $\epsilon = y - \hat{y}$ as

1. **Bias** how close between target y and prediction $\hat{y}$, with smaller being better. High bias model leads to underfitting.
2. **Variance** the extent to which model prediction error changes based on training inputs, with smaller being better. High variance model leads to overfitting.

2.3.3 Model Complexity

Complexity of a model sets the flexibility of $\hat{f}$. For example, increasing maximum tree depth increases the model complexity[1].

Occam's razor, applied to ML, suggests that simpler models all else equal are better than more complicated one, as they tend to generalize better.

A model can be overfitted when it picks up too much noise or spurious patterns using the training data. Simpler models are less likely to overfit. Conversely, underfitting can happen when a model is not picking up enough of the true relationship underlying the data. Overfitting happens often in reality and one common approach to mitigate it is through regularization.

Bias-Variance Trade-off. There is a trade-off between bias and variance. Though low variance and low bias is desirable, usually a model cannot achieve both at the same time. For a model predicting housing price, a linear regression model based on a set of features may have low variance but high bias. That means the linear prediction could often be off the market value, but the variance in these predictions are low. This can be due to under-fitting. Conversely, a neural net model could fit target in sample with low bias, its prediction out of sample could vary widely depending on input features.

One's goal is to find the model complexity that achieves the lowest generalization error. Since this error is the sum of three terms with the irreducible error being constant, you need to find a balance between bias and variance because as one increases the other decreases. This is known as the bias-variance trade-off[1].

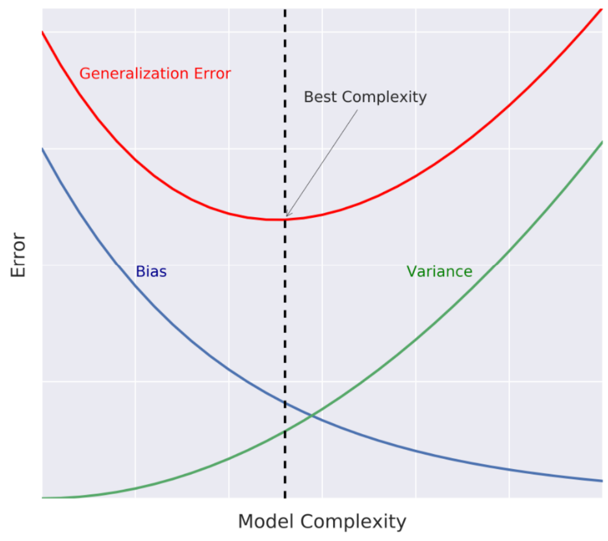


Figure 1: Bias-Variance Tradeoff

2.3.4 Cross-Validation

How can one estimate generalization error so we can pick the right model complexity? One approach is to do train and test split, fit the model on the training set, and then use evaluate the error of $\hat{f}$ on the unseen test set. Generalization error can be roughly approximated by the test set error. However, there is an issue. Test set should usually be kept unseen until we finalize the model $\hat{f}$. To estimate generalization error without using test set, one can use a technique called **Cross-Validation** (CV).

A popular cross-validation approach is *k-fold cross-validation*:

1. Randomly shuffle data into k equal size buckets (folds).
2. For each fold $i \in [1, 2, ..., k]$, train the model using all data except fold i, and evaluate the validation error using block i.
3. Average the k validation errors to get an estimate of true error.

Below figure illustrates the process of 10-K CV[1]:

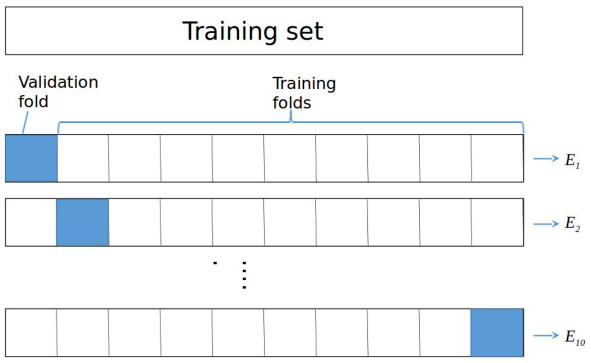


Figure 2: K-Fold CV (K=10)

A special case of CV is *leave-one-out* cross-validation. Special treatment is needed for dealing with timeseries data given one cannot use future data for training.

2.3.5 Model Diagnostics

CV can be used to diagnose variance and bias problems[1].

Diagnosis of High Variance Problem. If f suffers from high-variance, then one should expect CV error of f to be larger than training set error of f. If model suffers from high variance, then one can:

- decrease model complexity (e.g., reduce max tree depth, increase min sample per leaf).
- increase training sample size

Diagnosis of High Bias Problem. If f suffers from high-bias, then one should expect CV error of $f \approx$ training set error of $f >>$ desired error. If model suffers from high bias, then one can:

- increase model complexity (e.g., increase max tree depth, reduce min sample per leaf).
- increase dimension of the features

2.3.6 Regularization

Regularization is a *shrinkage method* aims to reduce model complexity in a way that significantly reduce model variance while only slightly increasing model bias. The most widely used regularization are L1 and L2. When regularization term is introduced, a penalty term is added to the objective function in a way that it shrinks feature coefficients to reduce overfitting.

1. L1 regularization is called *lasso*. L1 penalty adds absolute value of coefficients into objective function. It serves as a feature selection tool, since many feature coefficients are shrunken to zero. It leads to a more sparse model.
2. L2 regularization is call *ridge*. L2 penalty adds squared magnitudes of coefficients into objective function. Different from L1, it tends to shrink all coefficnets towards zero.
3. L1 and L2 regularization can be linearly combined, called *elastic net*.

2.3.7 Interpretability

There is often a trade-off between model performance and interpretability. There can be different ways to interpret different models.

1. Linear models: regression weights can be visualized and used to measure impact in the final prediction.
2. Random forest: feature importance can be used.
3. More generally, *SHAP* (Shapley Additive exPlanation) is model agnostic, which uses Shapley values to denote the average marginal contribution of a feature over all possible combinaiton of inputs.
4. Another model agnostic approach is *LIME* (Local Interpretable Model-agnostic Explanations), using sparse linear models built around various predictions to understand how a model performs around a local vicinity.

2.4 Model Evaluation Metrics

2.5 ROC-AUC

AUC stands for 'Area under the Receiver Operating Characteristic (ROC) Curve.' The AUC ROC curve is basically a way of measuring the performance of an ML model. AUC measures the ability of a binary classifier to distinguish between classes and is used as a summary of the ROC curve.

The ROC AUC score of a binary classifier can be determined using the roc_auc_score() function from sklearn.metrics.

3 Linear Models

3.1 Linear Regression

3.2 Logistic Regression

A classification tree divides the feature space into rectangular regions. In contrast, a linear model such as logistic regression produces only a single linear decision boundary dividing the feature space into two decision regions.

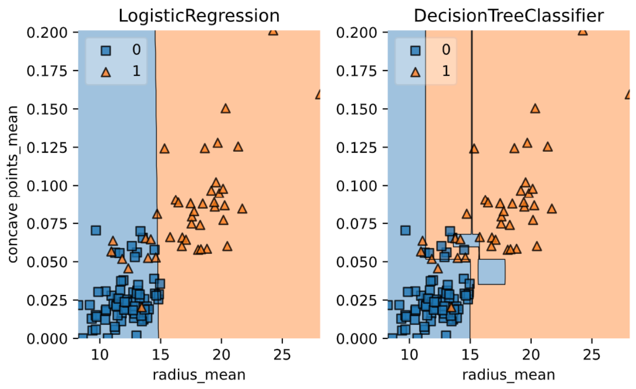


Figure 3: Decision boundary: Logistic vs DT

4 Tree-based Models

Tree-based models such as random forests can handle both classification and regression tasks with all kinds of input features with minimal processing needed.

4.1 Decision Trees

Also known as CART (classification and regression trees).

4.1.1 Classification Tree

Training. How is a decision tree being trained? DT is a model that can be represented in a treelike form determined by binary splits made in the feature space (a classification tree divides the feature space into rectangular regions) and resulting in various leaf nodes, each with a different prediction. It is trained in a greedy and recursive manner, starting at a root node and subsequently proceeding through a series of binary splits that lead to minimal error in the classification of observations.

To produce the purest leaf possible, at each node, a tree asks a question involving one feature f and a split-point sp. How does it know what feature and split-point to use? It considers that each node contains *information*, and splits the nodes to maximize the *information gain* (IG).

$$IG(f, sp) = I(parent) - (\frac{N_{left}}{N} \times I(left) + \frac{N_{right}}{N} \times I(right))$$

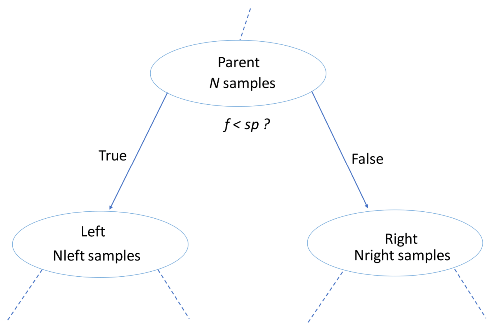


Figure 4: Information Gain

The information gain can be commonly measured by Gini index (Impurity) or *Entropy*. Entropy of a random variable Y quantifies the uncertainty in Y, for a discrete variable Y with K states,

$$H(Y) = \sum_{k=1}^{K} P(y = k)log(P(y = k))$$

When a variable has high entropy, it has high uncertainty and its distribution is closer to a uniform one. In the context of decision tree's node split, the reduction in uncertainty (or information gain) based on feature X can be measured as

$$IG(Y, X) = H(Y) - H(Y|X)$$

The general training process accesses all features in consideration and chooses the feature that maximize this information gain, then recursively repeats the process on the two resulting branches. The training results may depend on the information criterion (e.g., in sklearn criterion can be entropy or gini for DecisionTreeClassifier).

In summary, when an unconstrained tree is trained,

- Nodes are grown recursively
- At each node, split the data based on f and split-point sp such that information gain $IG(node)$ is maximized.
- If $IG(node) = 0$, a leaf is found

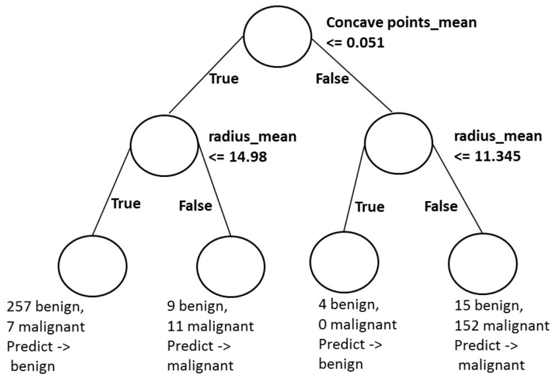


Figure 5: Decision Tree for Breast Cancer dataset

4.1.2 Regression Tree

Training. DT can also be used to give continuous output (label). When a regression tree is trained, the impurity of the node is measured using mean-squared-error (MSE) of the targets in that node. The regression tree tries to find the splits that produce leafs where in each leaf the target values are on average, the closest possible to the mean-value of the labels in that particular leaf.

$$I(node) = MSE(node) = \frac{1}{N_{node}} \sum_{i \in node} (y_i - \hat{y}_{node})^2$$

Where

$$\hat{y}_{node} = \frac{1}{N_{node}} \sum_{i \in node} y_i$$

Prediction. A a new instance traverses the tree and reaches a certain leaf, the prediction for the target variable is computed as the average of target variables in that leaf.

$$\hat{y}_{leaf} = \frac{1}{N_{leaf}} \sum_{i \in leaf} y_i$$

Below we show the decision boundary of regression tree and linear regression. As one can see, it can fit the training sample better. That said, it can also easily overfit. This can be mitigated by techniques such as regularization and ensembling.

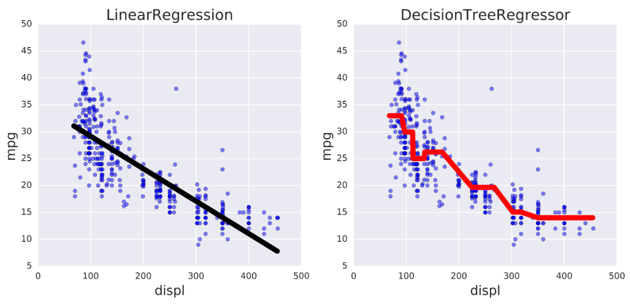


Figure 6: Linear Regression vs Regression Tree

4.2 Ensemble Learning

In ensemble learning, we train different models using the same data. Each model will have their predictions, and a meta model will then aggregate all the individual model predictions. A strong ensemble model can be learned if each individual predictor have complementary predictability. The meta model can be more robust and less prone to errors. The below chart illustrates an ensemble learning.

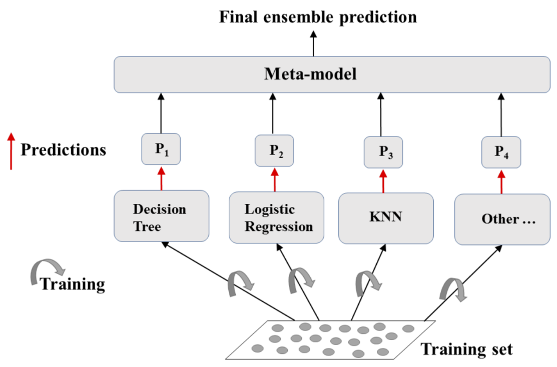


Figure 7: Illustrative Example of Ensemble Learning (Voting Classifier)

4.2.1 Bootstrapping and Bootstrap Aggregation (Bagging)

Bagging stands for Bootstrap Aggregation. It is a technique that can be used to reduce variance of a model.

In a VotingClassifier, we train a list of different estimators (algorithms) using the same data. In contrast, in bagging, we train the same algorithm using different subset of the data.

In **bootstrapping**, we sample observations from a dataset repeatedly with replacement and estimate population measure by averaging estimates from multiple smaller models, each trained with smaller data sample. Bootstrapping can also be useful in dealing with *class imbalance* by over-sampling rare classes.

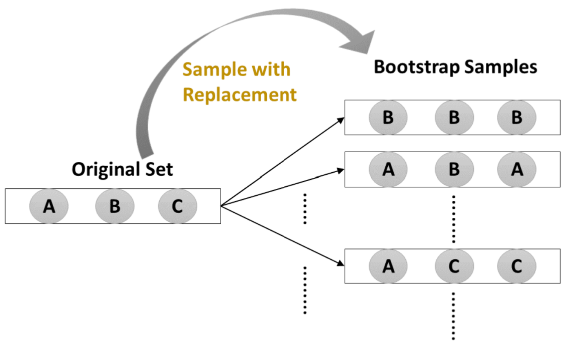


Figure 8: Bootstrapping

Bootstrapping is used in ensemble learning, when averaging estimates from many smaller (weaker) models into a main (stronger) model. This aggregation process is also *bagging*. Ensemble models like random forests utilize it.

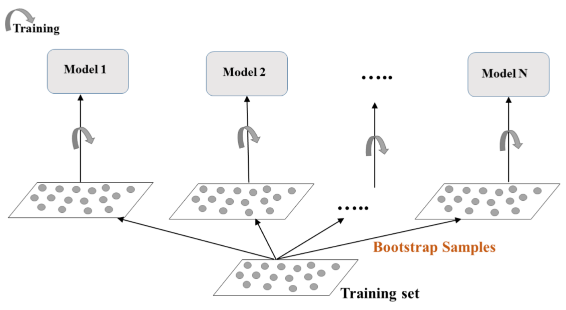


Figure 9: Bagging: Training

In prediction, each model predicts an outcome and the meta model aggregates them. In classification (BaggingClassifier), the meta model aggregates the results through majority voting. In regression, the aggregation is done through averaging[1].

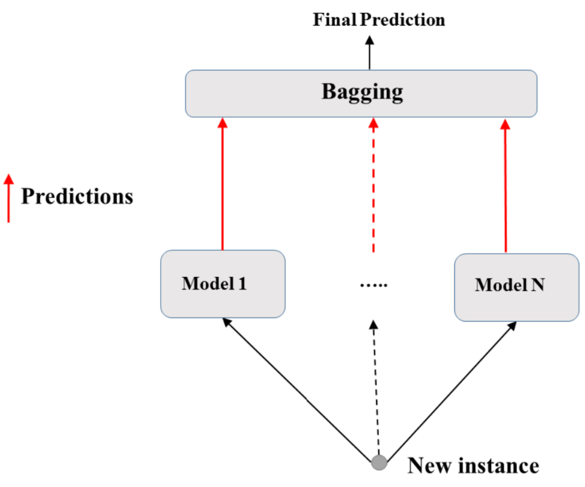


Figure 10: Bagging: Prediction

When sampling, there can be samples that are sampled multiple times, while some samples may be not sampled at all. These unseen samples were not used during the training process, and are known as out-of-bag (OOB) instances. Below chart shows the workflow of OOB evaluation[1]:

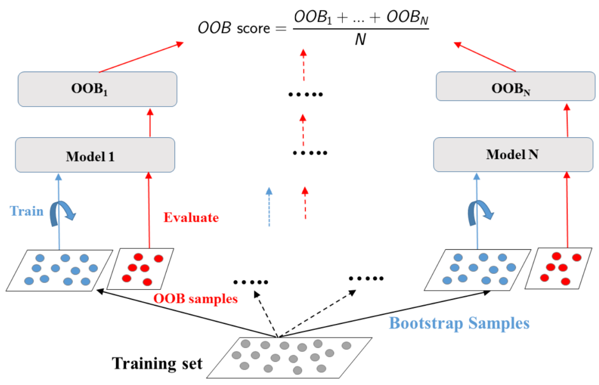


Figure 11: OOB Evaluation

OOB accuracy can be close to test-set accuracy. OOB-evaluation can be an efficient technique to obtain a performance estimate of a bagged-ensemble on unseen data without performing cross-validation.

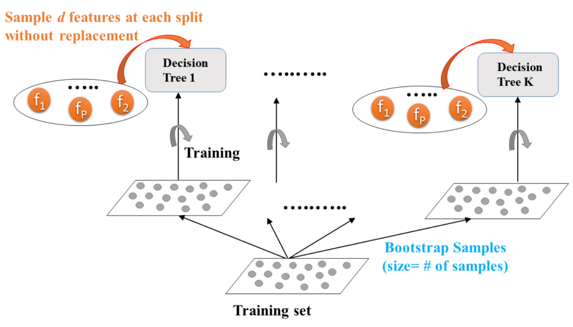
4.3 Random Forests

Training. Random Forests is a ensemble model that uses Decision Tree as the base estimator. An individual decision tree is prone to overfitting, as a leaf can be created for each observations. Random forest, as an ensemble method, usually achieve better results than individual decision tree due to two features.

1. **Bagging**. Each sub-tree is trained on different bootstrap sample of the sample size as the training data. By averaging decisions of many smaller trees, out-of-sample variance is significantly reduced.

2. A **random subset of features are selected with-replacement during each split**, which prevents important in-sample features being always present at the top of each individual trees. In scikit, the default is to keep $d = \sqrt{N}$ features.

The following figure[1] illustrates the training process of random forests:



Sample d features at each split without replacement
Decision Tree 1
f_1
f_p
f_2
Training
Decision Tree K
Bootstrap Samples (size= # of samples)
Training set

Figure 12: Illustration of training process of Random Forests

Prediction. For classification, the final prediction is made by majority voting. The corresponding scikit-learn class is RandomForestClassifier. For regression, the final prediction is the average of all the labels predicted by the base estimators. The corresponding scikit-learn class is RandomForestRegressor. In general, Random Forests achieves a lower variance than individual trees[1].

Feature Importance. How much a tree uses a particular feature to reduce its impurity.

What are other pros and cons of random forests?

1. Interpretability. Feature importance can be used.

2. Fast training. Training of each sub-tree can be done in parallel.

4.4 Boosting

Boosting is an ensemble method where lots of weak predictors are trained, and each predictor tries to learn from (correct) the errors from its predecessors. These weak learners will be combined to form a strong learner. Each weak learner is weak by itself, but still better than a random guesser. In tree-based models, a weak learner can be a simple decision tree (e.g., a decision tree with depth of one, a.k.a. a decision stump). Two popular boosting methods are Adaboost and Gradient Boosting.

4.5 AdaBoost

One type of boosting model is **AdaBoost**, which stands for **Ada**ptive **Boost**ing.

Training. In AdaBoost, each predictor pays more attention to samples wrongly predicted by its predecessors, by weighting these samples more in the training phase. The training process is illustrated in the chart below[1]. The coefficient assigned to each predictor, α, is a function on the predictor's training error. This coefficient α_1 is then used to determine weight for the subsequent predictor W_2, which is a weight vector that assigns more weights to incorrectly predicted samples. These weighted samples will be used to train the predictor 2, hence it will pay more attention to better fit the previously incorrectly predicted instances. These process repeats sequentially until the final predictor forms its prediction.

One hyper-parameter used in this process is the learning rate, which is used to shrink the $\alpha = \eta \cdot \alpha$. A smaller learning rate should be compensated with more number of predictors.

Prediction. For classification, the prediction is determined by weighted majority voting. For regression, the prediction is based on weighted average.

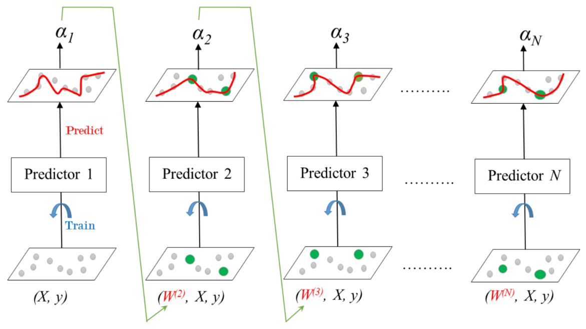


Figure 13: AdaBoost Training

4.6 Gradient Boosting

Similar to AdaBoost, in Gradient Boosting (GB), each predictor in the ensemble sequentially correct the error of the previous predictor. Different to AdaBoost, the training weight is not changed. Instead, each predictor uses the residual from its predecessor as training label. One type of GB model is **Gradient Boosted Tree** (GBT), which uses CART as the base learner.

Training. GBT is trained sequentially. The training process is illustrated in the chart below[1]. Each subsequent predictor uses the residual from its predecessor as the training target. This process is repeated until all trees are trained. One parameter used in the training is the shrinkage η, similar to learning rate used in AdaBoost.

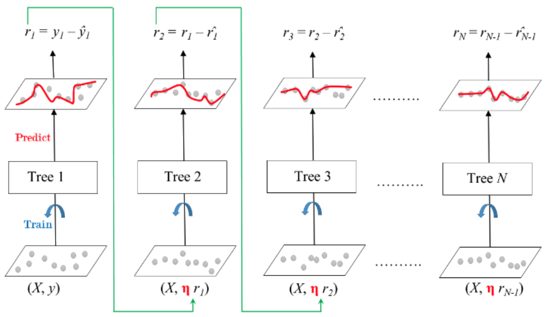


Figure 14: Gradient Boosted Tree for Regression: Training

Prediction. For regression, the prediction is a weighted average of the predictions:

$$\hat{y} = y_1 + \eta \cdot r_1 + \eta \cdot r_2 + ... + \eta \cdot r_N$$

Python Implementation To implement GB regressor in Python:

```
# Import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingRegressor

# Instantiate gb
gb = GradientBoostingRegressor(max_depth=4,
            n_estimators=200,
            random_state=2)

# Fit gb to the training set
gb.fit(X_train, y_train)

# Predict test set labels
y_pred = gb.predict(X_test)

# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error

# Compute MSE
mse_test = mean_squared_error(y_test, y_pred)

# Compute RMSE
rmse_test = mse_test ** 0.5

# Print RMSE
print('Test set RMSE of gb: {:.3f}'.format(rmse_test))
```

4.7 Stochastic Gradient Boosting

One weakness of GB is it involves an exhaustive search procedure. Each base leaner (e.g. CART) is trained to find the best split points and features. And this may lead to each CARTs using the same split points and same features. SGB can be used to mitigate this weakness. In SGB, each tree is trained on a random subset of training data (sampled without replacement), and for each predictor, a random subset of features are used (sampled without replacement).

Training. The training process is illustrated in the chart below[1].

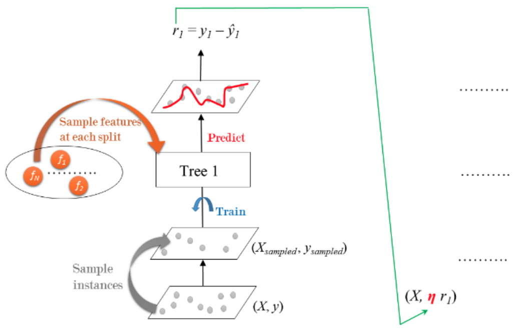


Figure 15: SGB: Training

Python Implementation.

```
# Import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingRegressor

# Instantiate sgbr
sgbr = GradientBoostingRegressor(max_depth=4,
            subsample=0.9,
            max_features=0.75,
            n_estimators=200,
            random_state=2)

# Fit sgbr to the training set
sgbr.fit(X_train, y_train)

# Predict test set labels
y_pred = sgbr.predict(X_test)

# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE

# Compute test set MSE
mse_test = MSE(y_pred, y_test)
```

```
# Compute test set RMSE
rmse_test = mse_test ** 0.5

# Print rmse_test
print('Test set RMSE of sgbr: {:.3f}'.format(rmse_test))
```

4.8 Hyperparameters Tuning

There are two types of parameters[1]:

1. **Parameters**: learned from data. For example, in case of CART, split-point of a node, split-feature of a node
2. **Hyperparameters**: not learned from data, set prior to training. For example, in case of CART, max depth, min samples per leaf, spliting criterion

Hyperparameters Tuning is the process to search for the optimal hyperparameters for a learning model. Such hyperparameters, once found, will result in an optimal model that yields an optimal score based on some evaluation metric. In sklearn, the default scores used are accuracy (for classification) and R^2 (for regression).

There are many approaches to hyperparameter tuning, including grid search, random search, etc.

4.8.1 Grid search cross validation

In grid search based hyperparameter tuning, we'll manually set a grid of discrete hyperparameter values. We'll also set a metric for scoring model performance. We'll then search exhaustively through the grid. For each set of hyperparameters, evaluate the CV score. The optimal set of hyperparameters are those using which the model is able to achieve the best CV score[1].

Python Implementation. We'll show an example to tune hyperparameters for a classification tree by performing grid search using a 5-fold CV, using ROC AUC score as the metric. Grid search is an exhausive process, so it may take a lot of time to train the model.

```
# Define params_dt
params_dt = {'max_depth': [2, 3, 4],
'min_samples_leaf': [0.12, 0.14, 0.16, 0.18]}

# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Instantiate grid_dt
grid_dt = GridSearchCV(estimator=dt,
                              param_grid=params_dt,
                              scoring='roc_auc',
                              cv=5,
                              n_jobs=-1)
# This is time consuming (X_train is 80% of the full data)
grid_dt.fit(X_train, y_train)

# Import roc_auc_score from sklearn.metrics
from sklearn.metrics import roc_auc_score

# Extract the best estimator
best_model = grid_dt.best_estimator_

# Predict the test set probabilities of the positive class
y_pred_proba = best_model.predict_proba(X_test)[:,1]

# Compute test_roc_auc
test_roc_auc = roc_auc_score(y_test, y_pred_proba)

# Print test_roc_auc
print('Test set ROC AUC score: {:.3f}'.format(test_roc_auc))
```

4.8.2 Hyperparameter Tuning for Random Forest

In general, hyperparameter tuning is expensive. It is computationally expensive, and sometimes leads to only marginal improvement [1].

Python Implementation. We'll manually set the grid of hyperparameters that will be used to tune rf's hyperparameters and find the optimal regressor. For this purpose, you will be constructing a grid of hyperparameters and tune the number of estimators, the maximum number of features used when splitting each node and the minimum number of samples (or fraction) per leaf. This example is taken from [1].

```
# Define the dictionary 'params_rf'
params_rf = {'n_estimators': [100, 350, 500],
     'max_features': [ 'log2', 'auto', 'sqrt'],
     'min_samples_leaf': [2, 10, 30]
}

# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Instantiate grid_rf
grid_rf = GridSearchCV(estimator=rf,
                              param_grid=params_rf,
                              scoring='neg_mean_squared_error',
                           cv=3,
                           verbose=1,
                           n_jobs=-1)

grid_rf.fit(X_train, y_train)

# Import mean_squared_error from sklearn.metrics as MSE
from sklearn.metrics import mean_squared_error as MSE

# Extract the best estimator
best_model = grid_rf.best_estimator_

# Predict test set labels
y_pred = best_model.predict(X_test)

# Compute rmse_test
rmse_test = MSE(y_pred, y_test) ** 0.5

# Print rmse_test
print('Test RMSE of best model: {:.3f}'.format(rmse_test))
```

5 Unsupervised Learning

5.1 PCA

How to use PCA to analyze statistical factors driving portfolio returns?

6 Commonly Asked questions

6.1 Conceptual

6.1.1 What is ROC-AUC score?

6.1.2 How does PCA work?

What are the common pitfalls of using PCA?

6.1.3 What's the difference between random forest and XGBoost?

7 ML in Asset Management

7.1 The WorldQuant Approach[2]

Scaling up the number of alphas into what Igor begain to call an 'alpha factory'. No regularity lasts forever, so there's a constant need for new alphas to identify new marker signals derived from new data sets and new insights. He developed what he calls 'tenets of prediction'. He found that the accuracy of models tended to rise logarithmically with the number of algorithmic models. 100 times more models produce in theory 10 times better predictions.

Every model or formula attepts to predict returns based on a very specific piece of information or a very specific way of looking at a piece of information. WQ has several hundred throusand alpha built on relatively simple and publicly available price and volume information. If combined predictions are uncorrelated, then their power increases with the square root of the number of predictions. If they are correlated, the increase is less steep, typically following a logarithmic improvement for each new alpha. If we combine an exponential number of logarithmic improvements, we get a linear improvement with the logarithm of alpha quantity. Quantity is necessary to win the 'prediction war' in the markets, in the immune system, and in any predictive system. Getting data is effectively an arms race.

Algorithms may contain internal biases in approach that their developers never recognize. A bias can create inadvertent correlations in seemingly diverse strategies, spawning a corwded trade.

Idea Arbitrage: Arbitrage is a fundamental aspect of trading, and the engine driving the market toward accurate pricing- price discovery is itself a process of prediction. THe essence of arbitrage is the power of competition to produce a winner. The process pf idea arbitrage is: ideate (develop ideas), arbitrate (the verbal form of arbitrage), and predict. COmbine with ensemble.

8 Word2vec

Word2vec can be trained using Neuron networks.

8.1 Loss Function

Sigmoid/Logistic function:

$$\sigma(x) = \frac{1}{1 + exp(-x)}$$

References

[1] Datacamp course: Machine learning with tree-based models in python.

[2] How ai is making prediction more precise — and what that means for risk and human behavior.

[3] Kevin Huo Nick Singh. Ace the data science interview: 201 real interview questions asked by faang, tech startups, wall street. 2022.