Soo Min Kwon

Brief Review of Machine Learning Algorithms

Preamble

This review guide serves as a refresher for several widely used machine learning algorithms in practice. If you are not aware of some of these algorithms, then you may (or may not) have a hard time grasping the complete idea from just this review guide. Fortunately, there are many resources out there to do a complete thorough review, such as Medium and StatQuest.

1 Decision Trees for Classification

Decision trees are pretty straightforward in that we create a “tree” that we can traverse to make decisions. The following is an illustration of a decision tree:

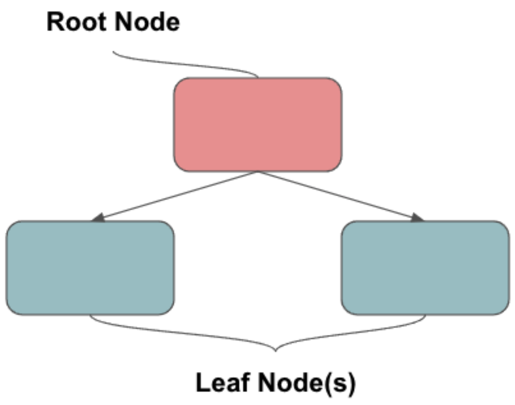

Figure 1: Illustration of a decision tree

Like most graphical trees, decision trees have a “root node” that is placed at the top, internal nodes that are placed in between, and “leaf nodes” that do not have any more arrows pointing away from them. Decision trees are flexible in the data we work with – we can use binary data, categorical data, rank data, and numerical data. Now the main questions we need to ask are how we can **choose the root node**, and how we can learn to **split each node**. In order to answer these questions, we need to introduce the **Gini index**.

1.1 Gini Index

The reason we need the Gini index is because we need some quantifiable way to determine **which feature (or variable) is best at predicting our dependent variable**. We want to put the feature that best predicts the dependent variable as our root node. We need to compute the Gini index of each feature and choose the feature with the lowest Gini index, which is defined as the following:

$$\text{Gini Index} = 1 - (\text{Probability of Yes})^2 - (\text{Probability of No})^2. \tag{1}$$

The Gini index tells us how “good” each feature is at predicting the dependent variable. Note that if one feature can perfectly classify the dependent variable, then the Gini index value would be 0. Likewise, if one feature cannot classify the dependent variable at all, then its Gini index would be 1. We can compute the Gini index for each feature and each leaf, and choose the Gini index with the lowest weighted average.

1.2 Dealing with Numerical Data

We can also easily compute the Gini index for numerical data. For variables with numerical values, we can first sort the data in increasing order and compute the Gini index by using the average numerical value between two data samples as a threshold. We compute these values for every two pair of data samples, and choose the threshold that results in the smallest Gini index value.

1.3 Pros and Cons of Decision Trees

The following are some advantages of decision trees:

- **Interpretability:** Decision trees are fairly intuitive and easy to explain.
- **Handles Missing Data:** Decision trees can easily handle missing data, which means that they require less pre-processing of the data.

The following are some disadvantages:

- **Overfitting:** Decision trees are infamous for overfitting – they can learn the training data very well but not generalize well to any other data samples.
- **High Complexity:** Decision trees often take a long time to train, as we need to compute the Gini index (or with some other quantifiable method) for each feature.

Note that one of the disadvantages of using decision trees overfitting. We can somewhat handle overfitting of decision trees by **pruning**. We can “pre-prune” a decision tree by limiting the number of samples needed to split a node, as well as limit the maximum depth that a tree can grow into. These limitations prohibit the tree from fitting the training data to high precision. We can also “post-prune” a decision tree by allowing the tree to grow the maximum depth, and then reduce its number of branches (and hence nodes) by using **cost complexity pruning**.

2 Decision Trees for Regression

In the previous section, we discussed how we can use decision trees for classification. In the same way, we can use decision trees for regression, which means that we can use them to **predict continuous values**. Recall that for classification, we used the Gini index to quantify how “good” a feature was in predicting the dependent variable. For regression, instead of the Gini index, we use the **sum of squared errors**, which is just the square of the difference between a point and a boundary. So how do we choose this boundary? Well first, we need to pick a root node by thresholding each data sample. Then, the boundary is chosen by taking **the average of each data sample in that feature**. We choose the boundary that yields the smallest sum of squared errors, and predict that boundary value as our continuous, dependent variable. Just as we did in decision trees for classification, we need to compute the sum of squared errors for each feature, and choose the feature with the lowest value to be our root node. Then we traverse down the tree, and prune our trees in the same way we did for our decision trees. An advantage for regression trees over linear regression is that regression trees can handle **non-linear data**.

3 Random Forests

Random forests are a powerful tool used widely in data science for classification and regression. This algorithm is built upon creating many decision trees, hence its name. Recall that decision trees are often prone to **overfitting**. Random forests combine the simplicity of decision trees with flexibility to increase their performance. At a high level, one can think of random forests as an ensemble method, where we create a bunch of decision trees and take the majority vote as the outcome.

3.1 Bootstrapping

For random forests, we need to first create a **bootstrapped dataset**. We create a bootstrapped dataset that is the same size as the original dataset by **randomly selecting** samples from the original dataset. Note that we can choose the same data sample more than once for our bootstrapped dataset. The purpose of the bootstrapped dataset is to reduce the variance of a decision tree. In random forests, we create a bootstrapped dataset for each decision tree that we build.

3.2 Building the Decision Trees

Upon creating a bootstrapped dataset, the next step is to build a decision tree. Recall that for a decision tree, we need to find the independent variable that is most associated to predicting the dependent variable using the Gini index for our root node. We find this independent variable by computing the Gini index against **all** other independent variables. However, for decision trees in random forests, we only consider a **random subset** of independent variables for each node. This gives us variability in our trees, as we need to create many other trees.

3.3 Bagging

Upon bootstrapping and building approximately M decision trees, the next thing we need to do is actually classify the new data samples. As mentioned in the high-level explanation, we classify our data samples by obtaining a decision from each decision tree, and then by doing majority voting. Bootstrapping the data and using the aggregate to make a decision is called “bagging”.

3.4 Evaluating the Random Forest Model

In order to evaluate our random forest model, we need to use the “out-of-bag” dataset. This dataset is the portion of the data samples **that did not make it in our bootstrapped dataset.** We run these out-of-bag data samples against all of the decision trees that were trained without these data samples and compute the output. The portion of of out-of-bag samples that were incorrect classified is called the **out-of-bag error**. Putting everything together, the following is a brief outline of the random forest algorithm:

1. For each tree, sample with replacement a bootstrapped dataset.
2. With the bootstrapped dataset, create a classification or regression tree by randomly choosing T number of features.
3. Evaluate the model using the out-of-bag dataset.
4. Optimize the model by using cross validation for the number of features used, T. In practice, we generally start with $T \approx \sqrt{N}$, where N is the number of features in the dataset.

With these said, the following are the **hyperparameters for a random forest model:**

- Number of trees in the random forest
- Maximum depth for each decision tree (typically 8 to 32)
- Minimum samples for split (used to control overfitting)
- Maximum features considered for looking for the best split
- Minimum impurity for split

4 Gradient Boosting

4.1 Gradient Boosting for Regression

We first explain how we can use the Gradient Boosting algorithm for regression (i.e. predict continuous values). The main idea behind gradient boosting for regression is that we start with an initial guess, which is the average of the dependent variable, and build decision trees to refine the initial guess. We terminate the algorithm when a newly built decision tree does not improve our estimate by some small threshold. Briefly, the algorithm does the following:

1. Initialize guess by taking the average of all of the data samples' dependent variable.
2. Compute the residual between the initial estimate and the true value.
3. Build a decision tree with a finite depth (i.e. we do not let the tree build to full depth) to **predict the residuals**. Note that if multiple data samples predict the same leaf, then we output the **average of the residuals**.
4. Refine residuals by running training data through decision tree and computing

$$\text{New Guess} = \text{Initial Guess} + \alpha * (\text{Predicted Residual}) \tag{2}$$
$$\text{New Residual} = \text{True Value} - \text{New Guess} \tag{3}$$

5. Refine these residuals by building another tree, and predicting the **new residuals**. Continue this process until convergence.

4.2 Gradient Boosting for Classification

Similar to Gradient Boosting for regression, gradient boosting for classification does the following:

1. Recall that the log odds is computed as

$$\text{Log Odds} = \log\left(\frac{p}{1-p}\right), \tag{4}$$

or textually as the ratio between an event happening to an event not happening. We initialize our guess by computing **the log odds ratio of our binary responses**.

2. Compute the residuals between the initial log odds and the true probabilities. Note that in the first step, since we have binary responses, a "Yes" would be a probability of 1, and a "No" would be a probability of 0.
3. Build a decision tree where we limit the number of leaves we allow in the tree (i.e. do not let it grow to full depth) to predict the residuals.
4. Recall that in gradient boosting for regression trees, if multiple data samples went to the same leaf as an output, we outputted the **average of the residuals**. However, for gradient boosting for classification, we compute the following:

$$\text{Output} = \frac{\sum_i \text{Residuals}_i}{\sum_i \text{Previous Probability}_i \times (1 - \text{Previous Probability}_i)} \tag{5}$$

5. Upon computing these output values, we compute

$$\text{New Guess} = \text{Initial Guess} + \alpha * (\text{Output}) \tag{6}$$

6. This "New Guess" value refines our estimate of the log odds. We need to convert these log odds back into a probability by using the sigmoid function,

$$\sigma(x) = \frac{\exp(x)}{1 + \exp(x)} \tag{7}$$

7. Note that now, we will have a separate predicted probability for each data sample. We refine these predicted probabilities by building a new tree, and outputting **the updated residual values**. We continually refine these probabilities by building more trees and updating the residual values, until the probabilities do not change by a small threshold.

5 Extreme Gradient Boosting (XGBoost)

5.1 XGBoost for Regression

Recall that in Gradient Boosting for Regression, we iteratively refined our initial guess by building decision trees that predicted the residuals. The nodes in these decision trees were built "normally", in the sense that we used the Gini index (or Entropy) to split the nodes. Similarly, in XGBoost for Regression, we also iteratively build decision trees to refine our estimates of the residuals (and hence the output), but we do not use the Gini index to build our trees. Instead, for our residuals, we compute a Similarity Score defined as the following:

$$\text{Similarity Scores} = \frac{\text{Squared Sum of Residuals}}{\text{Number of Residuals} + \lambda}, \tag{8}$$

where λ is a regularization parameter. Upon computing this similarity score for the root node, we need to compute the similarity score to decide **how we split our root node into leaves**. Upon splitting, we compute the similarity score again and **use these similarity scores to compute a value called Gain:**

$$\text{Gain} = \text{Left Similarity} + \text{Right Similarity} - \text{Root Similarity}. \tag{9}$$

We compute this Gain value for other thresholds and choose the threshold for the root node (and hence split) that gives us the most "gain".

5.2 Pruning Decision Trees in XGBoost

Now that we have a small sense of how we build decision trees in XGBoost, we need to know how we can **prune the trees to handle overfitting.** For XGBoost, we prune the trees based on the Gain score. We subtract the Gain score by another parameter γ, and if this difference is negative, then we would prune that leaf away. We would only prune the root node away if the difference for every branch including the root node is negative. **Note** that even if λ, the regularization parameter in the similarity score, is 0, there is still a possibility for pruning. Hence, it is easier to prune trees with a value of $\lambda > 0$.

5.3 Output of the Decision Tree

Even though we predict the residuals in the decision trees for XGBoost, the residuals are not the outputs. The **outputs of the tree is a value based on the residuals**, which is the following:

$$\text{Output} = \frac{\text{Sum of Residuals}}{\text{Number of Residuals} + \lambda}. \tag{10}$$

Then, our new output is

$$\text{New Output} = \text{Initial Guess} + \alpha * (\text{Output}), \tag{11}$$

which we use to refine our residuals and build more trees. The algorithm is briefly outlined as follows:

1. Set initial regression guess to 0.5.

2. Build M decision trees by first computing the similarity scores to decide on the threshold for the root nodes.

3. Using the similarity scores, compute the Gain score in order to decide how to split each node. We will split the node if the Gain is greater than the previous root's Gain value.

4. Prune the decision trees by subtracting the Gain value by some value γ.
5. Compute the output value and continue making trees until maximum number of trees is reached or does not change residual values by some small threshold.

5.4 XGBoost for Classification

Similar to XGBoost for Regression, XGBoost for classification is similar to that of Gradient Boosting for classification. The difference is that while we also predict the residuals to compute the Gain score to split the nodes, **XGBoost has a threshold for the minimum number of residuals for each leaf**. This minimum number is determined by calculating a value called **Cover**. This cover value is computed as the following:

$$\text{Cover} = \sum_i \text{Previous Probability}_i \times (1 - \text{Previous Probability}_i) \tag{12}$$

Cover is a hyperparameter that we have to choose to build our decision trees in XGBoost for classification.

5.5 Similarity Score for Classification

Recall that we need to use the similarity score to compute the Gain score to know how to split our decision trees. The similarity score for XGBoost for classification is the following:

$$\text{Similarity Score} = \frac{\sum_i (\text{Residuals}_i)^2}{\sum_i \text{Previous Probability}_i \times (1 - \text{Previous Probability}_i) + \lambda}, \tag{13}$$

where λ is the regularization parameter. Here, if $\lambda > 0$, then it reduces the amount that a single observation adds to a new prediction. Thus, λ reduces the prediction's sensitivity to isolated observations. The algorithm can be summarized as the following:

1. Initialize prediction probability to 0.5.
2. Compute the residuals and put it at the top of the decision tree. Compute the similarity score to decide how to cluster the residuals (and hence split them) in our decision tree.
3. Decide on the threshold for the decision tree by using the similarity score and compute the Gain score. Split the decision tree again by comparing the Gain scores.
4. Compute the output score defined as

$$\text{Output} = \frac{\sum_i \text{Residuals}_i}{\sum_i \text{Previous Probability}_i \times (1 - \text{Previous Probability}_i) + \lambda} \tag{14}$$

5. Similar to gradient boosting, update the probabilities by updating the log odds, and then using the sigmoid function.

5.6 Optimizing the XGBoost Model

The way that in which the XGBoost model builds its decision trees is **greedy**, since it needs to check every possible threshold to compute the similarity and Gain scores. This is where we introduce the **approximate greedy algorithm**. The approximation portion is that we do not have to check every possible threshold for every feature of our dataset, and rather check **quantiles** as thresholds for each variable.

6 Light Gradient Boosting Machine (LightGBM)

The "light" term in LightGBM comes from the fact that LightGBM models are faster to train than XGBoost models. The main difference between these two algorithms is that LightGBM **grows its trees vertically**, while other algorithms grow their trees horizontally. The LightGBM also **bundles features together**, which speeds up the training process. The following are some of the most important parameters associated with training a LightGBM model:

- Maximum Depth: the maximum depth of a tree used to handle overfitting
- Minimum Data in a Leaf: the minimum number of data samples needed in order to consider splitting the leaf
- Feature Fraction: the fraction of the parameters to be used to training and building decision trees
- Bagging Fraction: the fraction of the data to be used for training, generally used to speed up training and prevent overfitting
- Minimum Gain in Split: the minimum number of Gain to split a node
- Number of Leaves: the maximum number of leaves in a tree

7 Logistic Regression

Recall that in linear regression, the objective is to fit a linear model of the form

$$\mathbf{y} = \mathbf{W}^\top \mathbf{x} + b. \tag{15}$$

In logistic regression, instead of predicting continuous values $\mathbf{y}$, the objective is to predict the log odds of the form

$$\log\left(\frac{\mathbb{P}(y=1)}{1-\mathbb{P}(y=1)}\right) = \mathbf{W}^\top \mathbf{x} + b. \tag{16}$$

If we rearrange the equation above and solve for $\mathbb{P}(y=1)$, we get the following:

$$\begin{aligned}
\frac{\mathbb{P}(y=1)}{1-\mathbb{P}(y=1)} &= \exp\{\mathbf{W}^\top \mathbf{x} + b\} && (17)\\
\mathbb{P}(y=1) &= \exp\{\mathbf{W}^\top \mathbf{x} + b\} - \mathbb{P}(y=1)\exp\{\mathbf{W}^\top \mathbf{x} + b\} && (18)\\
\mathbb{P}(y=1) + \mathbb{P}(y=1)\exp\{\mathbf{W}^\top \mathbf{x} + b\} &= \exp\{\mathbf{W}^\top \mathbf{x} + b\} && (19)\\
\mathbb{P}(y=1) &= \frac{\exp\{\mathbf{W}^\top \mathbf{x} + b\}}{1+\exp\{\mathbf{W}^\top \mathbf{x} + b\}}. && (20)
\end{aligned}$$

Similarly, then solving for $\mathbb{P}(y=1) = 1 - \mathbb{P}(y=0)$ would give us

$$\mathbb{P}(y=0) = \frac{1}{1+\exp\{\mathbf{W}^\top \mathbf{x} + b\}}. \tag{21}$$

The objective is to find the parameters $\mathbf{W}$ and b such that we minimize the error of the probabilities of $\mathbb{P}(y=0)$ and $\mathbb{P}(y=1)$. The loss function to penalize both of these probabilities is

$$y_i \log(\mathbb{P}(y=1)) + (1-y_i)\log(1-\mathbb{P}(y=1)). \tag{22}$$

We can find the parameters by using Empirical Risk Minimization on this loss function.

8 Support Vector Machines

8.1 Linear SVM

Linear Support Vector Machines (SVM) on separable data is also called the maximum margin linear classifier. This name comes from the fact that the objective is to maximize the margin (or distance) between the classes of data and the boundary. With non-separable data, we "relax" the constraint from *all* of the data being correctly labeled to finding a hyperplane with a margin M such that the margin violations are as small as possible. Mathematically, we want to estimate $\mathbf{w}, b$ from

$$\mathbf{w}, b = \underset{\mathbf{w}, b}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y_i(\mathbf{w}^\top \mathbf{x}_i + b)\} + \frac{\lambda}{2} \|\mathbf{w}\|^2. \tag{23}$$

Note that this objective function uses the **hinge loss**, which is of the form

$$\ell(y, f(\mathbf{x})) = \max\{0, 1 - f(\mathbf{x})\}, \tag{24}$$

where $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}_i + b$ for linear SVM and $f(\mathbf{x}) = 0$ defines the decision boundary.

9 K-Means Clustering

The objective in k-means clustering is to form k clusters such that all of the samples in each cluster are "tightly" packed (i.e. the variation in each cluster is small). For example, if we use the ℓ_2 distance as our distance metric, we have the objective function

$$\mathcal{C}_1, \ldots, \mathcal{C}_k = \underset{\mathcal{C}_1, \ldots, \mathcal{C}_k}{\operatorname{argmin}} \sum_{i=1}^{k} \frac{1}{|\mathcal{C}_i|} \sum_{j, j' \in \mathcal{C}_i} \|\mathbf{x}_j - \mathbf{x}_{j'}\|_2^2. \tag{25}$$

However, it turns out this is actually a very challenging problem that requires an exhaustive search. Instead, we use Lloyd's approximation for the k-means algorithm, which results in the following:

1. Given: $\{\mathbf{x}_j\}_{j=1}^{n}$, k (the number of clusters), and $d(\cdot,\cdot)$ (distance metric)
2. Initialization: Randomly assign each $\mathbf{x}_j$ to one of k clusters
3. Loop until no change in any cluster occurs:
   (a) Compute centroid μ_i for $i = 1, \ldots, k$ of all clusters

   $$\mu_i = \frac{1}{|\mathcal{C}_i|} \sum_{j \in \mathcal{C}_i} \mathbf{x}_j \tag{26}$$

   (b) Go through all data samples $\mathbf{x}_j$ and put $\mathbf{x}_j$ in cluster i if

   $$\|\mathbf{x}_j - \mu_i\|_2 \leq \|\mathbf{x}_j - \mu_{i'}\|_2, \tag{27}$$

   for all $i \neq i'$.