Gradient Boosting Algorithm - Part 2. Classification

Algorithm explained with an example, math, and code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import scipy.stats as stats
from sklearn.tree import DecisionTreeRegressor

import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots

sns.set()
```

Algorithm with an Example

Gradient boosting is one of the variants of ensemble methods where you create multiple weak models (they are often decision trees) and combine them to get better performance as a whole. In this section, we are building a gradient boosting classification model using very simple example data to intuitively understand how it works.

The picture below shows the example data. It has the binary class y (0 and 1) and two features (x_1 and x_2).

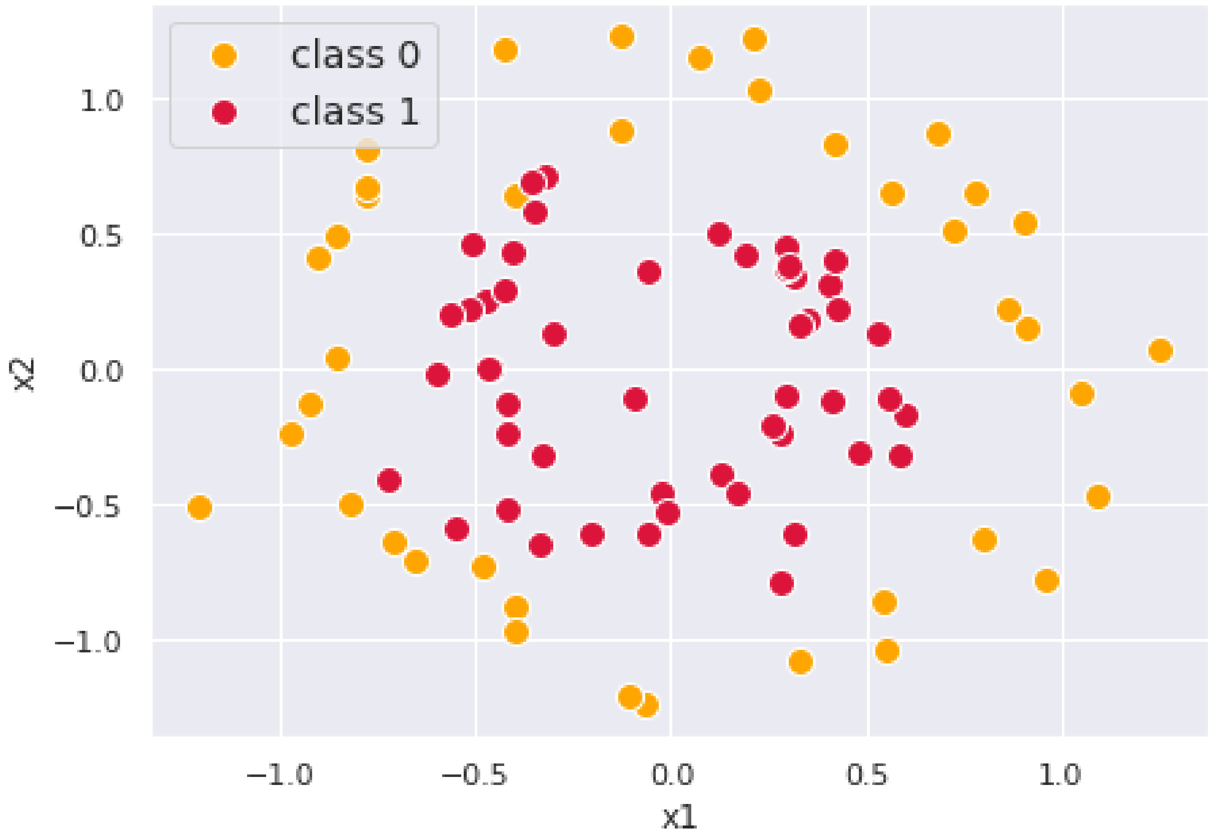
```
import sklearn.datasets as datasets
from matplotlib.colors import ListedColormap

data = datasets.make_circles(n_samples=100, factor=0.5, noise=0.15,
random_state=0)
x, y = data[0], data[1]

# make it imbalance
idx = np.sort(np.append(np.where(y != 0)[0], np.where(y == 0)[0][:-
10]))
x, y = x[idx], y[idx]

plt.figure(figsize=(7, 5))
plt.scatter(x[y==0, 0], x[y==0, 1], c='orange', edgecolors='w', s=100,
label='class 0')
```

```
plt.scatter(x[y==1, 0], x[y==1, 1], c='crimson', edgecolors='w',
s=100, label='class 1')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(fontsize=14)
plt.show()
```



```
c_scale = [[0, "rgb(247, 168, 84)"], [1, "rgb(209, 0, 0)"]]

def create_scatter(x, y, plot_type):

    return go.Scatter3d(
                x=x[:, 0], y=x[:, 1], z=y,
                mode="markers",
                marker=dict(
                    size=6,
                    color=y,
                    colorscale=c_scale if plot_type == "pred" else
"Blugrn",
                    line=dict(width=4, color="White"),
                ),
            )
```

```
def format_plot(fig):
  return fig.update_layout(
      scene=dict(
          xaxis_title="x1",
          yaxis_title="x2",
          zaxis_title="y"
      ),
      height=400,
      width=600,
      margin=dict(l=10, r=10, t=40, b=20),
  )

fig = go.Figure()
fig.add_trace(create_scatter(x, y, "pred"))
fig = format_plot(fig)
fig.show()
```

Our goal is to build a gradient boosting model that classifies those two classes. The first step is making a uniform prediction on a probability of class 1 (we will call it p) for all the data points. The most reasonable value for the uniform prediction might be the proportion of class 1 which is just a mean of y.

$$p = P(y=1) = \acute{y}$$

Here is a 3D representation of the data and the initial prediction. At this moment, the prediction is just a plane which has the uniform value $p = mean(y)$ on the y axis all the time.

```
# creating mesh data to visualize prediction planes
x1_min, x1_max = x[:, 0].min() - 0.5, x[:, 0].max() + 0.5
x2_min, x2_max = x[:, 1].min() - 0.5, x[:, 1].max() + 0.5

h = 0.02  # step size in the mesh
x1_mesh, x2_mesh = np.meshgrid(np.arange(x1_min, x1_max, h),
np.arange(x2_min, x2_max, h))

def create_surface(x1, x2, y, plot_type):

    return go.Surface(x=x1, y=x2, z=y,
                      showscale=False,
                      opacity=0.5,
                      colorscale="Peach" if plot_type == "pred" else
"Tealgrn",
                      )

p = y.mean()

fig = go.Figure()
fig.add_trace(create_scatter(x, y, "pred"))
fig.add_trace(create_surface(x1_mesh, x2_mesh, np.full(x1_mesh.shape,
p), "pred"))
```

```
fig = format_plot(fig)
fig.show()
```

In our data, the mean of y is 0.56. As it is bigger than 0.5, everything is classified into class 1 with this initial prediction. Some of you might feel that this uniform value prediction does not make sense, but don't worry. We will improve our prediction as we add more weak models to it.

To improve our prediction quality, we might want to focus on the residuals (i.e. prediction error) from our initial prediction as that is what we want to minimize. The residuals are defined as $r_i = y_i - p$ (i represents the index of each data point). In the figure below, the residuals are shown as the brown lines that are the perpendicular lines from each data point to the prediction plane.

```
def create_residual_lines(x, y, pred):

    #create the coordinate list for the lines
    x_list, y_list, z_list = [], [], []
    for i in range(len(x)):
        x_list.extend([x[i, 0], x[i, 0], None])
        y_list.extend([x[i, 1], x[i, 1], None])
        z_list.extend([y[i], pred[i], None])

    return go.Scatter3d(x=x_list,
                        y=y_list,
                        z=z_list,
                        mode='lines',
                        line=dict(
                            color='brown',
                            width=5
                        ))

fig = go.Figure()
fig.add_trace(create_scatter(x, y, "pred"))
fig.add_trace(create_surface(x1_mesh, x2_mesh, np.full(x1_mesh.shape,
p), "pred"))
fig.add_trace(create_residual_lines(x, y, np.full(y.shape, p)))
fig = format_plot(fig)
fig.update_layout(showlegend=False)
fig.show()
```

To minimize these residuals, we are building a regression tree model with both x_1 and x_2 as its features and the residuals r as its target. If we can build a tree that finds some patterns between x and r, we can reduce the residuals by utilizing that created tree. To simplify the demonstration, we are building very simple trees each of that only has one split and two terminal nodes which is called "stump". Please note that gradient boosting trees usually have a little deeper trees such as ones with 8 to 32 terminal nodes. Here we are creating the first tree predicting the residuals with two different values $r = \{0.1, -0.6\}$.

```
def train_and_update(x, y, Fm, x1_mesh, x2_mesh, Fm_mesh,
learing_rate=0.1, print_tree=True):

    p = np.exp(Fm) / (1 + np.exp(Fm))
    r = y - p
    tree = DecisionTreeRegressor(max_depth=1, random_state=0)
    tree.fit(x, r)
    ids = tree.apply(x)

    if print_tree:
      print_tree_structure(tree)

    x_mesh = np.c_[x1_mesh.ravel(), x2_mesh.ravel()]
    r_pred_mesh = tree.predict(x_mesh).reshape(x1_mesh.shape)

    for j in np.unique(ids):
        fltr = ids == j
        num = r[fltr].sum()
        den = (p[fltr]*(1-p[fltr])).sum()
        gamma = num / den
        Fm[fltr] += learing_rate * gamma

        # update prediction value in the tree
        tree.tree_.value[j, 0, 0] = gamma

    gamma_update_mesh = tree.predict(x_mesh).reshape(x1_mesh.shape)
    Fm_mesh += learing_rate * gamma_update_mesh

    p_mesh = np.exp(Fm_mesh) / (1 + np.exp(Fm_mesh))

    return tree, r, Fm, r_pred_mesh, Fm_mesh

# this function print out tree structures. adapted from
https://scikit-learn.org/stable/auto_examples/tree/plot_unveil_tree_st
ructure.html
def print_tree_structure(clf):

    n_nodes = clf.tree_.node_count
    children_left = clf.tree_.children_left
    children_right = clf.tree_.children_right
    feature = clf.tree_.feature
    threshold = clf.tree_.threshold

    node_depth = np.zeros(shape=n_nodes, dtype=np.int64)
    is_leaves = np.zeros(shape=n_nodes, dtype=bool)
    stack = [(0, 0)]  # start with the root node id (0) and its depth
(0)
    while len(stack) > 0:
        # `pop` ensures each node is only visited once
        node_id, depth = stack.pop()
        node_depth[node_id] = depth

        # If the left and right child of a node is not the same we
have a split
        # node
        is_split_node = children_left[node_id] !=
children_right[node_id]
        # If a split node, append left and right children and depth to
`stack`
        # so we can loop through them
        if is_split_node:
            stack.append((children_left[node_id], depth + 1))
            stack.append((children_right[node_id], depth + 1))
        else:
            is_leaves[node_id] = True

    print('-'*80)
    print(
        "The binary tree structure has {n} nodes and has "
        "the following tree structure:\n".format(n=n_nodes)
    )
    for i in range(n_nodes):
        if is_leaves[i]:
            print(
                "{space}node={node} is a leaf node.".format(
                    space=node_depth[i] * "\t", node=i
                )
            )
            print(node_depth[i] * '\t', f"prediction:
{clf.tree_.value[i, 0, 0]:.1f}")
        else:
            print(
                "{space}node={node} is a split node: "
                "go to node {left} if X[:, {feature}] <= {threshold} "
                "else to node {right}.".format(
                    space=node_depth[i] * "\t",
                    node=i,
                    left=children_left[i],
                    feature=feature[i],
                    threshold=threshold[i],
                    right=children_right[i],
                )
            )
    print('-'*80)

F0 = np.log(p/(1-p))
F0 = np.full(len(y), F0)
F0_mesh = np.full(x1_mesh.shape, F0[0])
learing_rate = 0.9
tree, r, Fm, r_pred_mesh, Fm_mesh = train_and_update(x, y, F0,
x1_mesh, x2_mesh, F0_mesh, learing_rate=learing_rate)

fig = go.Figure()
fig.add_trace(create_scatter(x, r, "res"))
fig.add_trace(create_surface(x1_mesh, x2_mesh, r_pred_mesh, "res"))
fig = format_plot(fig)
fig.update_layout(showlegend=False)
fig.show()

----------------------------------------------------------------------
----------
The binary tree structure has 3 nodes and has the following tree
structure:

node=0 is a split node: go to node 1 if X[:, 0] <= 0.6397739946842194
else to node 2.
     node=1 is a leaf node.
      prediction: 0.1
     node=2 is a leaf node.
      prediction: -0.6
----------------------------------------------------------------------
----------
```

You might now think we want to add these predicted values to our initial prediction p to reduce its residuals if you already read the post talking about the regression algorithm, but things are slightly different with the classification. The values (we call it γ gamma) that we are adding to our initial prediction is computed in the following formula:

$$\frac{\sum_{x_i \in R_j} \left(y_i - p\right)}{\sum_{x_i \in R_j} p\left(1 - p\right)}$$

$\Sigma x_i \in R_j$ means we are aggregating the values in the sigma Σ on all the sample x_is that belong to the terminal node R_j. j represents the index of each terminal node. You might notice that the numerator of the fraction is the sum of the residuals in the terminal node j. We will go through all the calculations that give us this formula, but let's just use it to calculate γ for now. Below is the computed values of γ_1 and γ_2.

$$\gamma_1 = \frac{\sum_{x_i \in R_1} \left(y_i - 0.56\right)}{\sum_{x_i \in R_1} 0.56\left(1 - 0.56\right)} = 0.3$$

$$\gamma_2 = \frac{\sum_{x_i \in R_2} \left(y_i - 0.56\right)}{\sum_{x_i \in R_2} 0.56\left(1 - 0.56\right)} = -2.2$$

```
print_tree_structure(tree)
```

```
----------------------------------------------------------------------
----------
The binary tree structure has 3 nodes and has the following tree
structure:

node=0 is a split node: go to node 1 if X[:, 0] <= 0.6397739946842194
else to node 2.
     node=1 is a leaf node.
      prediction: 0.3
     node=2 is a leaf node.
      prediction: -2.2
----------------------------------------------------------------------
----------
```

This γ is not simply added to our initial prediction p. Instead, we are converting p into log-odds (we will call this log-odds converted value $F(x)$), then adding γ to it. For those who are not familiar with log-odds, it is defined below. You might have seen it used in logistic regression.

$$log(odds) = log\left(\frac{p}{1-p}\right)$$

One more tweak on the prediction update is that γ is scaled down by **learning rate** ν, which ranges between 0 and 1, and then added to the log-odds-converted prediction$F(x)$. This helps the model not to overfit the training data.

$$F_1(x) = F_0(x) + \nu \cdot \gamma$$

In this example, we use a relatively big learning rate $\nu = 0.9$ to make the optimization process easier to understand, but it is usually supposed to be a much smaller value such as 0.1. By substituting actual values for the variables in the right side of the above equation, we get our updated prediction $F_1(x)$.

$$F_1(x) = \left\{ \begin{array}{ll} log\left(\frac{0.56}{1-0.56}\right) + 0.9 \cdot 0.3 = 0.5 & \text{if } x \le 0.64 \\ log\left(\frac{0.56}{1-0.56}\right) - 0.9 \cdot 2.2 = -1.7 & \text{otherwise} \end{array} \right)$$

If we convert log-odds $F(x)$ back into the predicted probability $p(x)$ (we will cover how we can convert it in the next section), it looks like a stair-like object below.

```python
def create_prev_surface(x1, x2, y, plot_type):

    return go.Surface(x=x1, y=x2, z=y,
                      showscale=False,
                      opacity=0.5,
                      colorscale="Purples",
                      surfacecolor=np.ones(x1.shape),
                      )
```

```
p_mesh = np.exp(Fm_mesh) / (1 + np.exp(Fm_mesh))

fig = go.Figure()
fig.add_trace(create_scatter(x, y, "pred"))
fig.add_trace(create_prev_surface(x1_mesh, x2_mesh,
np.full(x1_mesh.shape, p), "pred"))
fig.add_trace(create_surface(x1_mesh, x2_mesh, p_mesh, "pred"))
fig = format_plot(fig)
fig.update_layout(showlegend=False)
fig.show()
```

The purple-colored plane is the initial prediction p_0 and it is updated to the red and yellow plane p_1. Now, the updated residuals r looks like this:

```
p = np.exp(Fm) / (1 + np.exp(Fm))

fig = go.Figure()
fig.add_trace(create_scatter(x, y, "pred"))
fig.add_trace(create_surface(x1_mesh, x2_mesh, p_mesh, "pred"))
fig.add_trace(create_residual_lines(x, y, p))
fig = format_plot(fig)
fig.update_layout(title="Updated Residuals", showlegend=False)
fig.show()
```

In the next step, we are creating a regression tree again using the same x_1 and x_2 as the features and the updated residuals r as its target. Here is the created tree:

```
p_mesh_prev = p_mesh
tree, r, Fm, r_pred_mesh, Fm_mesh = train_and_update(x, y, Fm,
x1_mesh, x2_mesh, Fm_mesh, learing_rate=learing_rate)

fig = go.Figure()
fig.add_trace(create_scatter(x, r, "res"))
fig.add_trace(create_surface(x1_mesh, x2_mesh, r_pred_mesh, "res"))
fig = format_plot(fig)
fig.update_layout(showlegend=False)
fig.show()
```

```
----------------------------------------------------------------------
----------
The binary tree structure has 3 nodes and has the following tree
structure:

node=0 is a split node: go to node 1 if X[:, 0] <= -0.6295882761478424
else to node 2.
     node=1 is a leaf node.
      prediction: -0.5
     node=2 is a leaf node.
      prediction: 0.1
```

```
-----------------------------------------------------------------------
----------
```

We apply the same formula to compute γ. The calculated γ along with the updated prediction F_2(x) are as follows.

$$F_2(x)=\left\{\begin{array}{ll} F_1(x)-v\cdot 2.3=0.5-0.9\cdot 2.3=-1.6 & \text{if } x_1\le -0.63 \\ F_1(x)+v\cdot 0.4=0.5+0.9\cdot 0.4=0.9 & \text{else if } -0.63<x_1\le 0.64 \\ F_1(x)+v\cdot 0.4=-1.7+0.9\cdot 0.4=-1.3 & \text{otherwise} \end{array}\right)$$

Again, if we convert log-odds $F_2(x)$ back into the predicted probability $p_2(x)$, it looks like something below.

```python
p_mesh = np.exp(Fm_mesh) / (1 + np.exp(Fm_mesh))

fig = go.Figure()
fig.add_trace(create_scatter(x, y, "pred"))

fig.add_trace(create_prev_surface(x1_mesh, x2_mesh, p_mesh_prev,
"pred"))
fig.add_trace(create_surface(x1_mesh, x2_mesh, p_mesh, "pred"))
p = np.exp(Fm) / (1 + np.exp(Fm))
# fig.add_trace(create_residual_lines(x, y, pred))
fig = format_plot(fig)
fig.update_layout(showlegend=False)
fig.show()
```

We iterate these steps until the model prediction stops improving. The figures below show the optimization process from 0 to 4 iterations.

```python
def create_subplots(iter_num):

    return make_subplots(
        rows=1, cols=2,
        specs=[[{"is_3d": True}, {"is_3d": True}]],
        horizontal_spacing=0,
        subplot_titles=(f"Residuals of iteration {iter_num}",
f"Predictions of iteration {iter_num}"),
    )

def format_subplots(fig):
    fig.update_layout(
        scene=dict(
            xaxis_title="x1",
            yaxis_title="x2",
            zaxis_title="y"
        ),
        scene2=dict(
            xaxis_title="x1",
            yaxis_title="x2",
            zaxis_title="y"
        ),
        height=400,
        width=1200,
        margin=dict(l=10, r=10, t=20, b=20),
    )
    fig.update_layout(showlegend=False)
    return fig

p = y.mean()
F0 = np.log(p/(1-p))

learing_rate = 0.9
Fm_mesh = np.full(x1_mesh.shape, F0)
Fm = np.full(len(y), F0)
n_estimators = 4
p = y.mean()

fig = create_subplots(0)
fig.append_trace(create_scatter(x, y, "pred"), row=1, col=2)
fig.append_trace(create_surface(x1_mesh, x2_mesh,
np.full(x1_mesh.shape, p), "pred"), row=1, col=2)
fig = format_subplots(fig)
fig.show()

for i in range(n_estimators):
    tree, r, Fm, r_pred_mesh, Fm_mesh = train_and_update(x, y, Fm,
x1_mesh, x2_mesh, Fm_mesh,

learing_rate=learing_rate, print_tree=False)
    p_mesh = np.exp(Fm_mesh) / (1 + np.exp(Fm_mesh))

    fig = create_subplots(i+1)
    fig.append_trace(create_scatter(x, r, "res"), row=1, col=1)
    fig.append_trace(create_surface(x1_mesh, x2_mesh, r_pred_mesh,
"res"), row=1, col=1)
    fig.append_trace(create_scatter(x, y, "pred"), row=1, col=2)
    fig.append_trace(create_surface(x1_mesh, x2_mesh, p_mesh, "pred"),
row=1, col=2)

    fig = format_subplots(fig)
    fig.show()
```

You can see the combined prediction $p(x)$ (red and yellow plane) is getting closer to our target y as we add more trees into the combined model. This is how gradient boosting works to predict complex targets by combining multiple weak models.

Code

In this section, we are translating the maths we just reviewed into a viable python code to help us understand the algorithm further. We are using `DecisionTreeRegressor` from scikit-learn to build trees which helps us just focus on the gradient boosting algorithm itself instead of the tree algorithm. We are imitating scikit-learn style implementation where you train the model with `fit` method and make predictions with `predict` method.

```python
class CustomGradientBoostingClassifier:

    def __init__(self, learning_rate, n_estimators, max_depth=1):
        self.learning_rate = learning_rate
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.trees = []

    def fit(self, X, y):

        F0 = np.log(y.mean()/(1-y.mean()))  # log-odds values
        self.F0 = np.full(len(y), F0)  # converting to array with the
input length
        Fm = self.F0.copy()

        for _ in range(self.n_estimators):
            p = np.exp(Fm) / (1 + np.exp(Fm))  # converting back to
probabilities
            r = y - p  # residuals
            tree = DecisionTreeRegressor(max_depth=self.max_depth,
random_state=0)
            tree.fit(X, r)
            ids = tree.apply(x)  # getting the terminal node IDs

            # looping through the terminal nodes
            for j in np.unique(ids):
              fltr = ids == j

              # getting gamma using the formula (Σresiduals/Σp(1-p))
              num = r[fltr].sum()
              den = (p[fltr]*(1-p[fltr])).sum()
              gamma = num / den

              # updating the prediction
              Fm[fltr] += self.learning_rate * gamma

              # replacing the prediction value in the tree
              tree.tree_.value[j, 0, 0] = gamma

            self.trees.append(tree)

    def predict_proba(self, X):
        Fm = self.F0

        for i in range(self.n_estimators):
            Fm += self.learning_rate * self.trees[i].predict(X)

        return np.exp(Fm) / (1 + np.exp(Fm))  # converting back to
probabilities
```

Please note that all the trained trees are stored in `self.trees` list object and it is retrieved when we make predictions with `predict_proba` method. Next, we are checking if our `CustomGradientBoostingClassifier` performs as the same as `GradientBoostingClassifier` from scikit-learn by looking at their log-loss on our data.

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import log_loss

custom_gbm = CustomGradientBoostingClassifier(
    n_estimators=20,
    learning_rate=0.1,
    max_depth=1
)
custom_gbm.fit(x, y)
custom_gbm_log_loss = log_loss(y, custom_gbm.predict_proba(x))
print(f"Custom GBM Log-Loss:{custom_gbm_log_loss:.15f}")

sklearn_gbm = GradientBoostingClassifier(
    n_estimators=20,
    learning_rate=0.1,
    max_depth=1
)
sklearn_gbm.fit(x, y)
sklearn_gbm_log_loss = log_loss(y, sklearn_gbm.predict_proba(x))
print(f"Scikit-learn GBM Log-Loss:{sklearn_gbm_log_loss:.15f}")

Custom GBM Log-Loss:0.461943067988696
Scikit-learn GBM Log-Loss:0.461943067988696
```

As you can see in the output above, both models have exactly the same log-loss.

Recommended Resources

In this notebook, we have reviewed all the details of the gradient boosting classification algorithm. If you are also interested in the regression algorithm, please look at the Part 1 notebook.

There are also some other great resources if you want further details of the algorithm:

- **StatQuest, Gradient Boost Part3 and Part 4**

These are the YouTube videos explaining the gradient boosting classification algorithm with great visuals in a beginner-friendly way.

- **Terence Parr and Jeremy Howard, How to explain gradient boosting**

While this article focuses on gradient boosting regression instead of classification, it nicely explains every detail of the algorithm.

- **Jerome Friedman, Greedy Function Approximation: A Gradient Boosting Machine**

This is the original paper from Friedman. While it is a little hard to understand, it surely shows the flexibility of the algorithm where he shows a generalized algorithm that can deal with any type of problem having a differentiable loss function.

References

- Jerome Friedman, Greedy Function Approximation: A Gradient Boosting Machine
- Terence Parr and Jeremy Howard, How to explain gradient boosting
- Matt Bowers, How to Build a Gradient Boosting Machine from Scratch
- Wikipedia, Gradient boosting