**Gradient Boosting Classification Algorithms**

**Gradient Boosting Classification:**

<https://colab.research.google.com/drive/13p46IFhg3h6BIdjxUcfXPco13jIOCV6I?usp=sharing>

<https://colab.research.google.com/drive/15G44KBuSgHs7hdSIuhwh-LR7Qur4xRyD?usp=sharing>

<https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-2-classification-d3ed8f56541e/>

<https://www.digitalocean.com/community/tutorials/gradient-boosting-for-classification>

[**What is Gradient Boosting?**](https://www.digitalocean.com/community/tutorials/gradient-boosting-for-classification#what-is-gradient-boosting)

Let’s start by briefly reviewing **ensemble learning**. Like the name suggests, ensemble learning involves building a strong model by using a collection (or “ensemble”) of “weaker” models. Gradient boosting falls under the category of boosting methods, which iteratively learn from each of the weak learners to build a strong model. It can optimize:

* Regression
* Classification
* Ranking

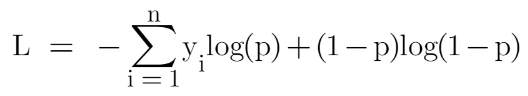
Gradient Boosting has three main components:

* **Loss Function** - The role of the loss function is to estimate how good the model is at making predictions with the given data. This could vary depending on the problem at hand. For example, if we’re trying to predict the weight of a person depending on some input variables (a regression problem), then the loss function would be something that helps us find the difference between the predicted weights and the observed weights. On the other hand, if we’re trying to categorize whether a person will like a certain movie based on their personality, we’ll require a loss function that helps us understand how accurate our model is at classifying people who did or didn’t like certain movies.
* **Weak Learner**—A weak learner classifies our data but does so poorly, perhaps no better than random guessing. In other words, it has a high error rate. These are typically decision trees (also called decision stumps because they are less complicated than typical decision trees).
* **Additive Model** - This is the iterative and sequential approach of adding the trees (weak learners) one step at a time. After each iteration, we need to be closer to our final model. In other words, each iteration should reduce the value of our loss function.

A gradient-boosting classifier works when the target column is binary. You apply all the steps explained in the gradient-boosting regressor here, the only difference is changing the loss function. Earlier, we used Mean squared error when the target column was continuous, but we will use log-likelihood as our loss function this time.

Let’s see how this loss function works.

The loss function for the classification problem is given below:



Our first step in the Gradient Boosting Algorithm in Machine Learning was to initialize the model with some constant value, there we used the average of the target column but here we’ll use log(odds) to get that constant value. The question arises: Why log(odds)?

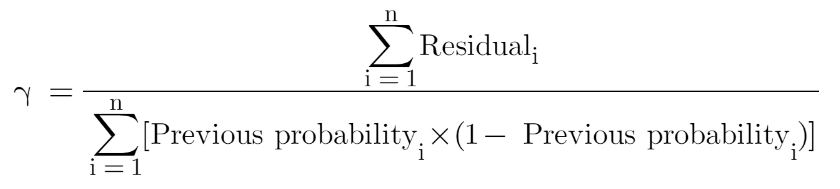
When we differentiate this loss function, we will get a function of log(odds), and then we need to find a value of log(odds) for which the loss function is minimum.

Our first step in the Gradient Boosting Algorithm in Machine Learning was to initialize the model with some constant value, there we used the average of the target column but here we’ll use log(odds) to get that constant value. The question arises: Why log(odds)?

When we differentiate this loss function, we will get a function of log(odds), and then we need to find a value of log(odds) for which the loss function is minimum.

After finding the residuals, we can build a decision tree with all independent variables and target variables as “Residuals”.

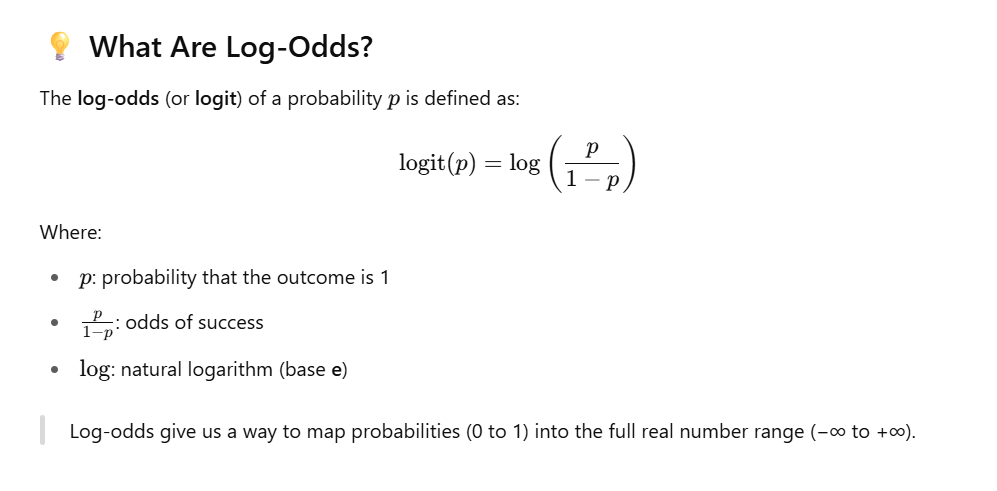
When we have our first decision tree, we find the final output of the leaves because there might be a case where a leaf gets more than 1 residuals, so we need to calculate the final output value. The math behind this step is out of the scope of this article, so I will mention the direct formula to calculate the output of a leaf:

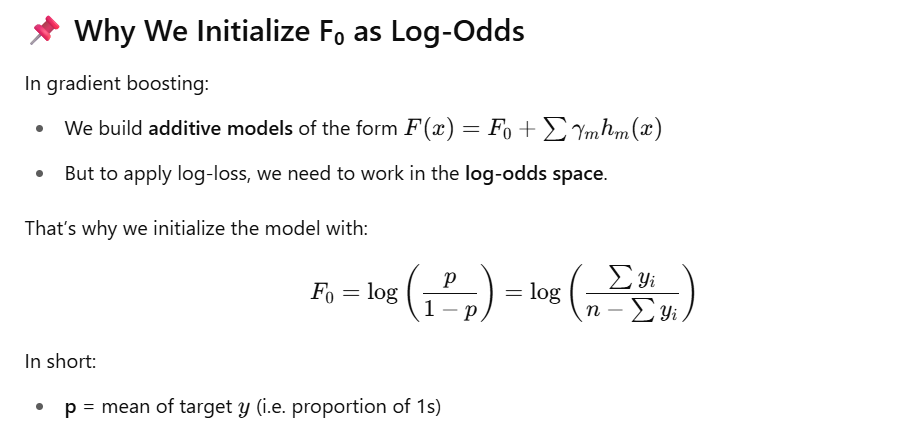


Finally, we are ready to get new predictions by adding our base model with the new tree we made on residuals.

**Why Log(Odds) in Gradient Boosting Classification?**

Gradient Boosting for binary classification is typically trained to **minimize the log-loss**, and under the hood it **models the log-odds** (also called **logit**) rather than probabilities directly.





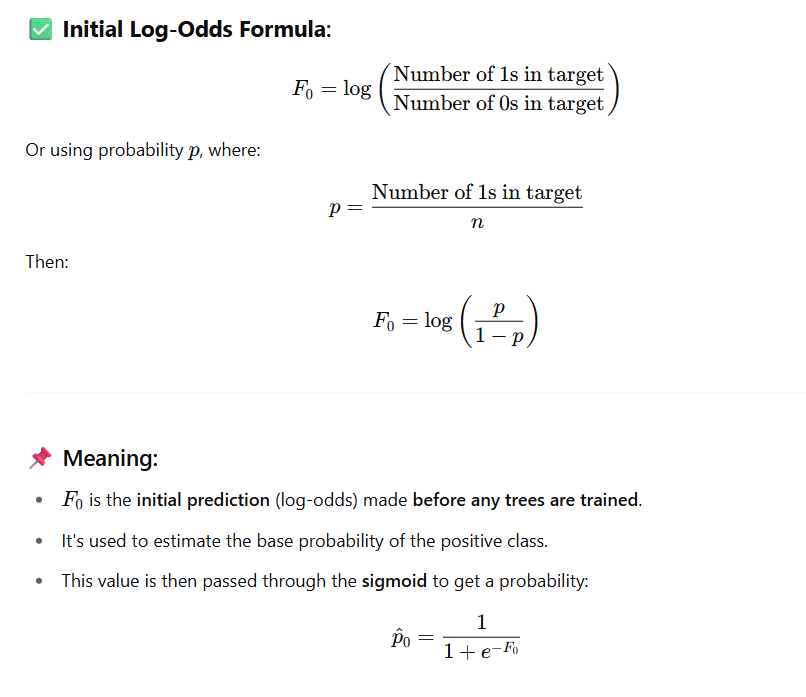
**All steps from scratch:**

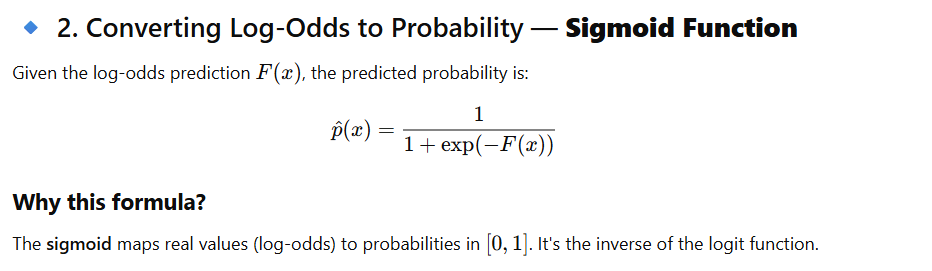
**Gradient Boosting for Binary Classification (with Log Loss)**

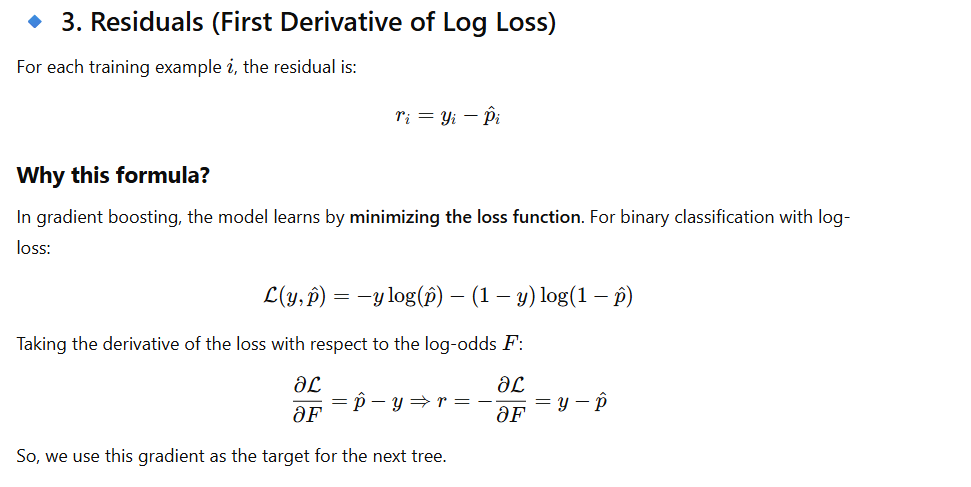
Gradient Boosting builds an **ensemble of decision trees**, where each tree fits the **residuals** (gradients) of the previous model. In binary classification with **log loss**, this process is optimized using derivatives of the log-loss function.

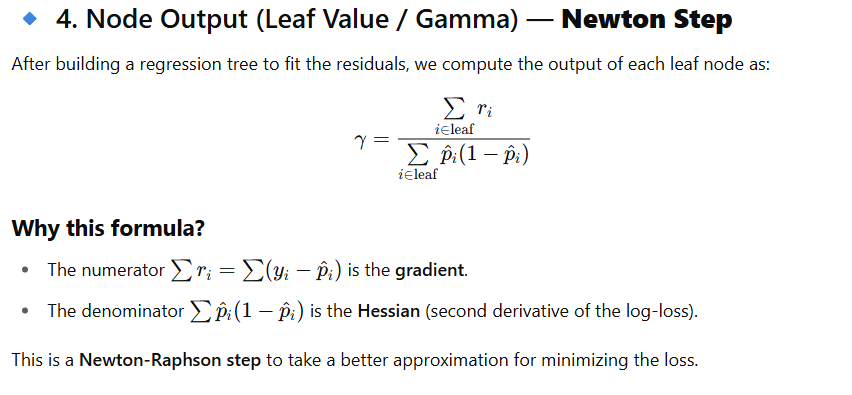
**🔹 1. Initial Prediction — Log-Odds**

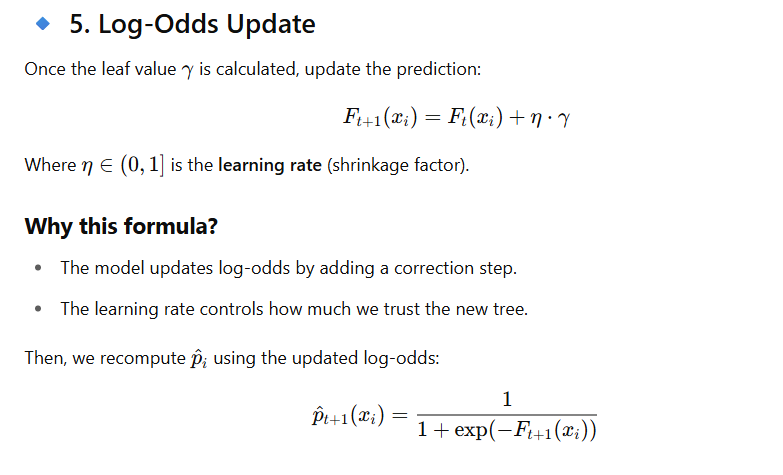
**Formula**

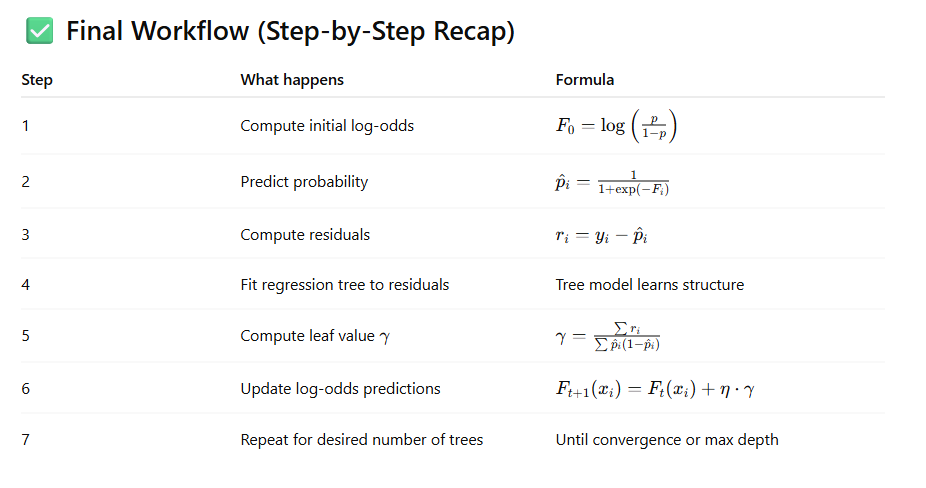












import pandas as pd

import numpy as np

from math import log, exp

# Sample binary classification data with 3 features and target (0 or 1)

data = pd.DataFrame({

'Feature1': [2, 1, 3, 4, 2, 1],

'Feature2': [1, 2, 1, 3, 2, 1],

'Feature3': [0, 1, 0, 1, 0, 1],

'Target': [0, 1, 0, 1, 0, 1]

})

# Step 1: Initial Log-Odds F0

num\_pos = data['Target'].sum()

num\_neg = len(data) - num\_pos

p = num\_pos / len(data)

F0 = log(p / (1 - p))

data['F0\_log\_odds'] = F0

# Step 2: Initial probability

data['P0'] = 1 / (1 + np.exp(-data['F0\_log\_odds']))

# Step 3: Initial residual

data['Residual1'] = data['Target'] - data['P0']

# Step 4: Assume we "build a tree" — manually partition into two leaves by Feature1 <= 2

data['Leaf1'] = data['Feature1'].apply(lambda x: 0 if x <= 2 else 1)

# Step 5: Compute gamma (leaf value) per node

def compute\_gamma(group):

p\_i = group['P0']

r\_i = group['Residual1']

numerator = r\_i.sum()

denominator = (p\_i \* (1 - p\_i)).sum()

return numerator / denominator if denominator != 0 else 0

leaf\_gammas1 = data.groupby('Leaf1').apply(compute\_gamma).to\_dict()

data['Gamma1'] = data['Leaf1'].map(leaf\_gammas1)

# Step 6: Update log odds

learning\_rate = 0.1

data['F1'] = data['F0\_log\_odds'] + learning\_rate \* data['Gamma1']

data['P1'] = 1 / (1 + np.exp(-data['F1']))

data['Residual2'] = data['Target'] - data['P1']

# Display after first tree

round1\_result = data[[

'Feature1', 'Feature2', 'Feature3', 'Target',

'F0\_log\_odds', 'P0', 'Residual1', 'Leaf1', 'Gamma1', 'F1', 'P1', 'Residual2'

]].copy()

round1\_result.columns = [

'Feature1', 'Feature2', 'Feature3', 'Target',

'F0 (log-odds)', 'P0 (prob)', 'Residual1', 'Leaf1', 'Gamma1',

'F1 (log-odds)', 'P1 (prob)', 'Residual2'

]

round1\_result.round(4)

import numpy as np

import pandas as pd

from sklearn.tree import DecisionTreeRegressor

# --------------------------

# Sample binary dataset

# --------------------------

df = pd.DataFrame({

'Feature1': [2, 1, 3, 4, 2, 1],

'Feature2': [1, 2, 1, 3, 2, 1],

'Feature3': [0, 1, 0, 1, 0, 1],

'Target': [0, 1, 0, 1, 0, 1]

})

# --------------------------

# Initial Log-Odds F0

# --------------------------

n\_pos = df['Target'].sum()

n\_neg = len(df) - n\_pos

p = n\_pos / len(df)

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

df['F0'] = F0

# Learning rate

lr = 0.1

# For tracking results

results = []

# Initial prediction and residual

df['F'] = F0 # Log-odds

df['P'] = 1 / (1 + np.exp(-df['F'])) # Initial probability

df['Residual'] = df['Target'] - df['P']

# --------------------------

# Boosting rounds

# --------------------------

for round\_num in range(1, 4):

# Fit tree on residuals

features = ['Feature1', 'Feature2', 'Feature3']

tree = DecisionTreeRegressor(max\_depth=1)

tree.fit(df[features], df['Residual'])

# Get predictions from the tree (leaf outputs)

df[f'Leaf{round\_num}'] = tree.apply(df[features])

# Compute gamma (leaf output) manually for each leaf

gammas = {}

for leaf in df[f'Leaf{round\_num}'].unique():

idx = df[f'Leaf{round\_num}'] == leaf

residuals = df.loc[idx, 'Residual']

probs = df.loc[idx, 'P']

gamma = residuals.sum() / (probs \* (1 - probs)).sum()

gammas[leaf] = gamma

# Map gamma and update log-odds F

df[f'Gamma{round\_num}'] = df[f'Leaf{round\_num}'].map(gammas)

df[f'F{round\_num}'] = df['F'] + lr \* df[f'Gamma{round\_num}']

df['F'] = df[f'F{round\_num}'] # update for next round

# Update probability and residuals

df['P'] = 1 / (1 + np.exp(-df['F']))

df['Residual'] = df['Target'] - df['P']

# Save results

results.append(df[[

'Feature1', 'Feature2', 'Feature3', 'Target',

f'F{round\_num}', 'P', 'Residual',

f'Leaf{round\_num}', f'Gamma{round\_num}'

]].copy().rename(columns={

f'F{round\_num}': 'F (log-odds)',

'P': 'P (probability)',

'Residual': 'Residual',

f'Leaf{round\_num}': 'Leaf ID',

f'Gamma{round\_num}': 'Leaf Output (γ)'

}))

# --------------------------

# Display results

# --------------------------

for i, res in enumerate(results, 1):

print(f"\n=== Round {i} ===")

print(res.to\_string(index=False, float\_format="%.4f"))