1. AdaBoost

Implementing AdaBoost for Predicting Online Shoppers' Purchase Intentions

- Team: 404 Not Found
- Team Members: Diksha Krishnan, Xinyu Zhou, Shen Yu, Dongyan Sun
- Date: December 9, 2024

2.1. An overview of the algorithm and describe its advantages and disadvantages.

- AdaBoost, short for Adaptive Boosting, is a powerful ensemble learning technique designed to combine multiple weak learners into a single, strong classifier in a sequential and adaptive manner. Developed by Yoav Freund and Robert Schapire, AdaBoost operates by iteratively training weak models, typically decision stumps (one-level decision trees), on a weighted version of the dataset, where the weights are adjusted dynamically after each iteration. Initially, all data points are given equal weights, but as the training progresses, higher weights are assigned to instances that were misclassified by previous learners, compelling subsequent models to focus more on these challenging examples. This adaptive weighting mechanism allows AdaBoost to progressively build a robust predictive model where each weak learner compensates for the shortcomings of its predecessors. The algorithm assigns importance to each learner based on its accuracy, combining their outputs in a weighted manner to form the final strong classifier. This makes AdaBoost particularly effective for classification tasks, as it transforms a series of weak models, each performing slightly better than random guessing, into a highly accurate predictive system. AdaBoost is widely recognized for its ability to generalize well across diverse datasets, making it a popular choice for applications such as facial recognition, medical diagnostics, fraud detection, and natural language processing. Its adaptability and focus on difficult examples give it an edge over many standalone algorithms, offering improved accuracy and better handling of complex datasets.
- AdaBoost offers significant advantages but also comes with certain disadvantages. Its simplicity and flexibility allow it to be implemented easily with various weak learners, and it works effectively for both classification and regression problems. By focusing on misclassified samples, AdaBoost transforms weak learners that perform only slightly better than random guessing into a strong ensemble model. AdaBoost is also robust to

overfitting when properly tuned and can handle non-linear problems by combining weak learners to form complex decision boundaries.

- One of its disadvantages is that AdaBoost is sensitive to outliers because misclassified samples are given higher weights, which can lead to poor performance if the misclassified samples are noisy or outliers. The algorithm requires careful tuning of hyperparameters, such as the number of iterations and learning rate, to achieve optimal performance without overfitting or underfitting. The effectiveness of AdaBoost also depends on the quality of the weak learners. If the weak learners are too complex, the model may overfit the training data. Additionally, AdaBoost can be computationally intensive due to its iterative nature, particularly when applied to large datasets.

2.2. Representation: describe how the feature values are converted into a single number prediction.

In the case of AdaBoost, the weak learner is the Desicion Stumps. AdaBoost creates a weighted combination of these stumps, resulting in a more complex decision boundary that better separates the data.

Each weak learner is assigned a weight w_m, which reflects its accuracy. The weight w_m is calculated as:

$$w_m = \frac{1}{2}\ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$$

where ϵ_m is the weighted error rate of the weak learner. A smaller error leads to a larger weight, indicating a better learner.

The final prediction $H(x)$ of the AdaBoost model is a weighted sum of the predictions from all weak learners:

$$H(x) = \text{sign}\left(\sum_{m=1}^{M} w_m h_m(x)\right)$$

M: Total number of weak learners.

$h_m(x)$: Prediction of the m-th weak learner (usually $+1$ or -1 for binary classification).

w_m: Weight of the m-th weak learner.

sign: Ensures the final output is a classification decision (e.g., $+1$ or -1).

2.3. Loss: describe the metric used to measure the difference between the model's prediction and the target variable.

AdaBoost minimizes an exponential loss function which ensures that large errors (misclassified points) contribute more significantly to the loss, encouraging the model to focus on correcting them. In this loss function, H represents the model's prediction, and y_i and x_i are the true label and the feature vector of the i-th data point, respectively. The exponential term emphasizes the errors made by the model, meaning that misclassified points (where $y_i H(x_i)$ is negative) will contribute significantly more to the overall loss. By focusing on these larger errors, AdaBoost adjusts its model iteratively, giving more weight to the misclassified points in subsequent rounds. This process leads to improved performance on harder-to-classify examples, with the model refining its predictions to minimize these larger errors.

$$\mathcal{L}(H) = \sum_{i=1}^{m} \exp(-y_i H(x_i))$$

2.4. Optimizer: describe the numerical algorithm used to find the model parameters that minimize the loss given a training set.

The decision stump, as the weak learner, is a key component of the AdaBoost algorithm. It is the base model that is iteratively trained and combined to construct the final AdaBoost hypothesis. The adaptive weighting and boosting of these simple decision stumps is what allows AdaBoost to transform a collection of weak learners into a robust, accurate classifier.

Let S be a training set and assume that at each iteration of AdaBoost, the weak learner returns a hypothesis for which $\epsilon_t \leq \frac{1}{2} - \gamma$. Then, the training error of the output hypothesis of AdaBoost is at most:

$$L_S(h_s) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{I}[h_s(x_i) \neq y_i] \leq \exp(-2\gamma^2 T)$$

Proof:
For each *t*, denote $f_t = \sum_{p \leq t} w_p h_p$. Therefore, the output of AdaBoost is f_T. In addition, denote

$$Z_t = \frac{1}{m} \sum_{i=1}^{m} e^{-y_i f_t(x_i)}$$

Note that for any hypothesis, we have $\mathbb{I}[h(x) \neq y] \leq e^{-yh(x)}$. Therefore $L_S(f_T) \leq Z_T$, so it suffices to show that $Z_T \leq e^{-2\gamma^2 T}$. To upper bound Z_T, we rewrite it as:

$$Z_T = \frac{Z_T}{Z_0} \cdot \frac{Z_T}{Z_{T-1}} \cdot \frac{Z_{T-1}}{Z_{T-2}} \cdot \cdots \cdot \frac{Z_2}{Z_1} \cdot \frac{Z_1}{Z_0}$$

where we used the fact that $Z_0 = 1$ because $f_0 = 0$. Therefore, it suffices to show that for every round t,

$$\frac{Z_{t+1}}{Z_t} \leq e^{-2\gamma^2}$$

To do so, first note that using a simple inductive argument, for all t and i,

$$D_i^{(t+1)} = \frac{e^{-y_i f_t(x_i)}}{\sum_{j=1}^m e^{-y_j f_t(x_j)}}$$

Hence,

$$\frac{Z_{t+1}}{Z_t} = \frac{\sum_{i=1}^m e^{-y_i f_{t+1}(x_i)}}{\sum_{j=1}^m e^{-y_j f_t(x_j)}} = \frac{\sum_{i=1}^m e^{-y_i f_t(x_i)} e^{-y_i w_{t+1} h_{t+1}(x_i)}}{\sum_{j=1}^m e^{-y_j f_t(x_j)}} = e^{-w_t} \left(\sum_{i:y_i h_{t+1}(x_i)=1} D \right.$$

This simplifies to:

$$e^{-w_t} \left(1 - \epsilon_t + e^{w_t} \epsilon_{t+1}\right)$$

By our assumption, $\epsilon_{t+1} \leq \frac{1}{2} - \gamma$. Since the function $g(a) = a(1-a)$ is monotonically increasing in $[0, 1/2]$, we obtain that:

$$2\sqrt{\epsilon_t(1-\epsilon_{t+1})} \leq 2\sqrt{\left(\frac{1}{2} - \gamma\right)\left(\frac{1}{2} + \gamma\right)} = \sqrt{1 - 4\gamma^2}$$

The proof shows that AdaBoost's training error is exponentially bounded by $\exp(-2\gamma^2 T)$, where γ is the margin of the weak learners and T is the number of iterations. The key idea is to bound the normalization factor Z_T, which ensures that AdaBoost minimizes error during each iteration. By expressing Z_T in terms of previous normalization factors and using an inductive argument, the proof demonstrates that Z_{t+1}/Z_t decreases exponentially at each step. This reduction is driven by the weak learner's error rate, ϵ_t, and the argument uses a bound on ϵ_t to show that the error of AdaBoost decreases over time. Thus, AdaBoost's output hypothesis improves with each iteration, leading to an exponentially decaying training error.

2.5. Pseudo Code to explain how numerical algorithms work.

Input:

$S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, where $y_i \in \{-1, +1\}$,

number of iterations T,

weak learner WL

Initialize: Sample weights $D_i^{(1)} = \frac{1}{m}, \forall i = 1, \ldots, m$.

for $t = 1$ **to** T :

Invoke weak learner $h_t = WL(D^{(t)}, S)$

Compute error rate: $\epsilon_t = \sum_{i=1}^m D_i^{(t)} 1[h_t(x_i) \neq y_i]$.

Let $w_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$.

$$\text{Update sample weights: } D_i^{(t+1)} = \frac{D_i^{(t)} \exp(-w_t y_i h_t(x_i))}{Z_t},$$

$$\text{where } Z_t = \sum_{i=1}^{m} D_i^{(t)} \exp(-w_t y_i h_t(x_i)), \text{ for all } i = 1, \ldots, m$$

$$\text{Output: Final hypothesis: } h_s(x) = \text{sign}\left(\sum_{t=1}^{T} w_t h_t(x)\right).$$

2.6. References and Citations

- Shalev-Shwartz, S. and Ben-David, S. (2014) Understanding Machine Learning: From Theory to Algorithms. Cambridge: Cambridge University Press.
- In pages 105-112 of the book, the mathematics and numerical algorithms behind AdaBoost, along with its application in face recognition, are discussed. Learned from both the book and real practice, AdaBoost relies heavily on numerical techniques such as exponentiation, logarithms, and weight normalization to ensure the algorithm's stability and efficiency during training. One key technique is exponentiation for weight updates, where AdaBoost increases the weights of misclassified samples exponentially based on the weak classifier's error. This allows the algorithm to focus more on challenging samples in subsequent rounds. However, exponentiation can lead to very large or small values, which could cause numerical instability. To prevent issues like underflow or overflow, AdaBoost uses efficient computation of exponential functions to maintain stability across a range of values. Another important numerical technique is the logarithmic computation of alpha, which adjusts the weight given to each weak classifier based on its performance. Weak classifiers with low error rates receive higher alpha values, contributing more to the final prediction. Using logarithms to compute alpha ensures that the values remain within a manageable range, preventing excessively large or small values that could destabilize training. Finally, normalization of weights is applied after each update to prevent the weights from growing too large. This ensures that the total weight remains constant, allowing AdaBoost to continue focusing on misclassified samples without causing numerical instability. These techniques are essential for AdaBoost's reliable performance, especially on complex real-world datasets.

3. Model

In [1]:
```python
# Add docstrings to each method and function and explain what they do
import numpy as np
import pandas as pd

class AdaBoost:
    def __init__(self, n_estimators=50):
        """
        Initialize the AdaBoost model.
        Args:
            n_estimators (int): The number of weak classifiers to use
        Attributes:
            alphas (list): Weights of each weak classifier.
```

```
            models (list): Parameters of the weak classifiers (decisio
        """
        self.n_estimators = n_estimators
        self.alphas = []  # Store the weights of the weak learners
        self.models = []  # Store weak classifiers (decision stumps)

    def _build_stump(self, X, y, weights):
        """
        Build a decision stump, which is a weak classifier based on a
        Args:
            X (numpy.ndarray): Feature matrix of shape (n_samples, n_f
            y (numpy.ndarray): Target labels of shape (n_samples,).
            weights (numpy.ndarray): Weights for each sample of shape

        Returns:
            stump (dict): Parameters of the best decision stump.
            min_error (float): Minimum classification error achieved b
        """
        n_samples, n_features = X.shape
        min_error = float("inf")  # Initialize with a high error
        stump = {}  # To store the best stump parameters

        for feature in range(n_features):
            # Iterate over all unique thresholds for the feature
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                for polarity in [-1, 1]:
                    # Predict using the current threshold and polarity
                    predictions = np.ones(n_samples)
                    if polarity == 1:
                        predictions[X[:, feature] <= threshold] = -1
                    else:
                        predictions[X[:, feature] > threshold] = -1

                    # Calculate weighted error
                    error = np.sum(weights[predictions != y])
                    if error < min_error:
                        # Update the best stump parameters if error is
                        min_error = error
                        stump['feature'] = feature
                        stump['threshold'] = threshold
                        stump['polarity'] = polarity
                        stump['predictions'] = predictions

        return stump, min_error

    def train(self, X, y):
        """
        Train the AdaBoost classifier using the training data.
        Args:
            X (numpy.ndarray): Feature matrix of shape (n_samples, n_f
            y (numpy.ndarray): Target labels of shape (n_samples,).

        Modifies:
            self.models: Appends the parameters of each weak learner.
            self.alphas: Appends the weight of each weak learner.
        """
        n_samples, _ = X.shape
        if n_samples <= 1:
            return
        weights = np.ones(n_samples) / n_samples  # Initialize uniform

        for _ in range(self.n_estimators):
            # Build the best weak classifier
            stump, error = self._build_stump(X, y, weights)
            # Compute the alpha (importance) of the stump
            alpha = 0.5 * np.log((1 - error) / (error + 1e-10))
```

```
            stump['alpha'] = alpha

            # Update sample weights
            weights *= np.exp(-alpha * y * stump['predictions'])
            weights /= np.sum(weights)  # Normalize weights

            # Store the stump and its weight
            self.models.append(stump)
            self.alphas.append(alpha)

    def predict(self, X):
        """
        Predict the class labels for the given data using the trained
        Args:
            X (numpy.ndarray): Feature matrix of shape (n_samples, n_f

        Returns:
            numpy.ndarray: Predicted labels of shape (n_samples,), val
        """
        n_samples = X.shape[0]
        final_prediction = np.zeros(n_samples)  # Accumulate weighted

        for model in self.models:
            # Extract parameters of the weak classifier
            feature = model['feature']
            threshold = model['threshold']
            polarity = model['polarity']
            alpha = model['alpha']

            # Make predictions using the weak classifier
            predictions = np.ones(n_samples)
            if polarity == 1:
                predictions[X[:, feature] <= threshold] = -1
            else:
                predictions[X[:, feature] > threshold] = -1

            # Add weighted predictions to final prediction
            final_prediction += alpha * predictions

        return np.sign(final_prediction)  # Return the final ensemble

    @staticmethod
    def f1_score(y_true, y_pred):
        """
        Compute the F1 Score of the predictions.
        Args:
            y_true (numpy.ndarray): True labels of shape (n_samples,).
            y_pred (numpy.ndarray): Predicted labels of shape (n_sampl

        Returns:
            float: F1 Score.
        """
        tp = np.sum((y_true == 1) & (y_pred == 1))
        fp = np.sum((y_true == -1) & (y_pred == 1))
        fn = np.sum((y_true == 1) & (y_pred == -1))

        precision = tp / (tp + fp + 1e-10)
        recall = tp / (tp + fn + 1e-10)
        return 2 * (precision * recall) / (precision + recall + 1e-10)

    @staticmethod
    def accuracy(y_true, y_pred):
        """
        Compute the accuracy of the predictions.
        Args:
            y_true (numpy.ndarray): True labels of shape (n_samples,).
            y_pred (numpy.ndarray): Predicted labels of shape (n_sampl
```

```
        Returns:
            float: Accuracy.
        """
        return np.mean(y_true == y_pred)
```

3. Model - Main

In [2]:

```
def preprocess_data(file_path):
    """
    Read and preprocess the dataset from a CSV file.
    Args:
        file_path (str): Path to the CSV file.

    Returns:
        X (numpy.ndarray): Preprocessed feature matrix.
        y (numpy.ndarray): Target labels converted to {1, -1}.
    """
    df = pd.read_csv(file_path)

    # Encode categorical features using one-hot encoding
    categorical_features = ['OperatingSystems','Month', 'Browser', 'Re
    df = pd.get_dummies(df, columns=categorical_features, drop_first=T

    # Combine all numerical features for standard scaling
    numerical_features = [
        'Administrative', 'Informational', 'Administrative_Duration',
        'Informational_Duration', 'ProductRelated', 'ProductRelated_Du
        'ExitRates', 'PageValues', 'SpecialDay'
    ]

    # Standard scale numerical features
    for col in numerical_features:
        df[col] = (df[col] - df[col].mean()) / (df[col].std() + 1e-10)

    # Separate features (X) and labels (y)
    X = df.drop(columns=['Revenue']).values  # Feature matrix
    y = df['Revenue'].values                 # Target labels
    y = np.where(y == 1, 1, -1)              # Convert labels to {1, -

    return X, y



def stratified_split(X, y):
    """
    Split the dataset into training, validation, and test sets while m
    Args:
        X (numpy.ndarray): Feature matrix of shape (n_samples, n_featu
        y (numpy.ndarray): Target labels of shape (n_samples,).

    Returns:
        Tuple[numpy.ndarray]: (X_train, y_train, X_val, y_val, X_test,
    """
    n_samples = len(y)
    neg_indices = np.where(y == -1)[0]
    pos_indices = np.where(y == 1)[0]

    # Shuffle indices
    np.random.shuffle(neg_indices)
    np.random.shuffle(pos_indices)

    # Split indices by class
    neg_train_end = int(0.6 * len(neg_indices))
    neg_val_end = int(0.8 * len(neg_indices))
```

```
    pos_train_end = int(0.6 * len(pos_indices))
    pos_val_end = int(0.8 * len(pos_indices))

    train_indices = np.concatenate([neg_indices[:neg_train_end], pos_i
    val_indices = np.concatenate([neg_indices[neg_train_end:neg_val_en
    test_indices = np.concatenate([neg_indices[neg_val_end:], pos_indi

    np.random.shuffle(train_indices)
    np.random.shuffle(val_indices)
    np.random.shuffle(test_indices)

    return X[train_indices], y[train_indices], X[val_indices], y[val_i


def main():
    # Load and preprocess the dataset
    file_path = '/Users/emmasun/Desktop/2060/project/online_shoppers_i
    X, y = preprocess_data(file_path)

    # Perform stratified split
    X_train, y_train, X_val, y_val, X_test, y_test = stratified_split(

    # Train the AdaBoost model
    model = AdaBoost(n_estimators=50)
    model.train(X_train, y_train)

    # Evaluate the model
    for split_name, (X_split, y_split) in zip(
        ['Training', 'Validation', 'Test'],
        [(X_train, y_train), (X_val, y_val), (X_test, y_test)]
    ):
        predictions = model.predict(X_split)
        acc = model.accuracy(y_split, predictions)
        f1 = model.f1_score(y_split, predictions)
        print(f"{split_name} Accuracy: {acc:.4f}, F1 Score: {f1:.4f}")


if __name__ == "__main__":
    main()
```

```
Training Accuracy: 0.8981, F1 Score: 0.6396
Validation Accuracy: 0.8865, F1 Score: 0.6078
Test Accuracy: 0.8950, F1 Score: 0.6357
```

4. Check Model

4.1. Unit Test

In [2]:

```
# Helper function for generating synthetic data
def generate_synthetic_data(n_samples, n_features, imbalance_ratio=0.5
    """
    Generate synthetic binary classification data.
    Args:
        n_samples (int): Number of samples.
        n_features (int): Number of features.
        imbalance_ratio (float): Proportion of positive samples.
    Returns:
        X (np.ndarray): Feature matrix.
        y (np.ndarray): Target labels.
    """
    X = np.random.rand(n_samples, n_features)
    y = np.random.choice([1, -1], size=n_samples, p=[imbalance_ratio,
    return X, y
```

```python
# Unit test for `_build_stump` method
def test_build_stump():
    """
    Test `_build_stump` method.
    Goals:
    - Validate correct selection of feature, threshold, and polarity.
    - Handle edge cases like tied feature values, single sample, and e
    """
    model = AdaBoost()

    # Case 1: Tied feature values
    # Goal: Ensure stump can handle identical feature values correctly
    X = np.array([[1], [1], [1], [1]])
    y = np.array([1, -1, 1, -1])
    weights = np.ones(4) / 4
    stump, error = model._build_stump(X, y, weights)
    assert stump["threshold"] == 1, "Threshold mismatch for tied featu
    assert error == 0.5, f"Error mismatch: {error}"

    # Case 2: Single sample
    # Goal: Verify behavior when only one sample is present.
    X = np.array([[2]])
    y = np.array([1])
    weights = np.array([1])
    stump, error = model._build_stump(X, y, weights)
    assert stump["feature"] == 0, "Feature mismatch for single sample'
    assert stump["threshold"] == 2, "Threshold mismatch for single sam
    assert error == 0, "Error mismatch for single sample"

    # Case 3: Extreme weights
    # Goal: Test if the stump handles very small and very large weight
    X = np.array([[1], [2], [3], [4]])
    y = np.array([1, -1, 1, -1])
    weights = np.array([1e-10, 1e-10, 1e-10, 1.0])
    stump, error = model._build_stump(X, y, weights)
    assert error < 1, f"Error out of range: {error}"

    print("test_build_stump passed!")

# Unit test for `predict` method
def test_predict():
    """
    Test the `predict` method.
    Goals:
    - Verify correctness and robustness across various edge cases.
    """
    # Case 1: Near decision boundaries
    # Goal: Test predictions for samples near decision boundaries.
    X_train = np.array([[1], [2], [3], [4]])
    y_train = np.array([1, 1, -1, -1])
    X_test = np.array([[1.5], [3.5]])
    model = AdaBoost(n_estimators=5)
    model.train(X_train, y_train)
    predictions = model.predict(X_test)
    assert predictions.shape == (X_test.shape[0],), "Prediction shape
    assert np.all(np.isin(predictions, [-1, 1])), "Invalid prediction

    # Case 2: Extreme values
    # Goal: Test predictions for unseen extreme feature values.
    X_test = np.array([[100], [-100]])
    predictions = model.predict(X_test)
    assert predictions.shape == (X_test.shape[0],), "Prediction shape

    # Case 3: Empty dataset
    # Goal: Verify behavior with an empty dataset.
    X_test = np.empty((0, 1))
    predictions = model.predict(X_test)
```

```
    assert predictions.shape == (0,), "Prediction shape mismatch for e

    # Case 4: Multi-dimensional features
    # Goal: Test predictions for multi-dimensional input features.
    X_train = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
    y_train = np.array([1, 1, -1, -1])
    X_test = np.array([[2, 3], [6, 7]])
    model = AdaBoost(n_estimators=5)
    model.train(X_train, y_train)
    predictions = model.predict(X_test)
    assert predictions.shape == (X_test.shape[0],), "Prediction shape

    print("test_predict passed!")

# Unit test for evaluation metrics
def test_evaluation_metrics():
    """
    Test `accuracy` and `f1_score` methods.
    Goals:
    - Verify correctness for perfect and partially incorrect predictio
    """
    model = AdaBoost()

    y_true = np.array([1, 1, -1, -1])
    y_pred_perfect = np.array([1, 1, -1, -1])
    y_pred_partial = np.array([1, -1, -1, 1])

    # Perfect predictions
    assert abs(model.accuracy(y_true, y_pred_perfect) - 1.0) < 1e-6, '
    assert abs(model.f1_score(y_true, y_pred_perfect) - 1.0) < 1e-6, '

    # Partially incorrect predictions
    assert abs(model.accuracy(y_true, y_pred_partial) - 0.5) < 1e-6, '
    assert abs(model.f1_score(y_true, y_pred_partial) - 0.5) < 1e-6, '

    print("test_evaluation_metrics passed!")

# Unit test for large datasets
def test_large_datasets():
    """
    Test model scalability and performance on large datasets.
    Goals:
    - Validate training and prediction with large sample sizes.
    """
    X, y = generate_synthetic_data(5000, 5)
    model = AdaBoost(n_estimators=20)
    model.train(X, y)
    predictions = model.predict(X)
    assert predictions.shape == (X.shape[0],), "Prediction shape misma
    print("test_large_datasets passed!")

# Run all unit tests
def run_all_tests():
    test_build_stump()
    test_predict()
    test_evaluation_metrics()
    test_large_datasets()
    print("All tests passed!")

# Execute the tests
run_all_tests()
```

```
test_build_stump passed!
test_predict passed!
test_evaluation_metrics passed!
test_large_datasets passed!
All tests passed!
```

4.2. Previous work where AdaBoost is applied on a public dataset.

- Swetha, T., R, R., Sajitha, T., B, V., Sravani, J. and Praveen, B. (2024) 'Forecasting Online Shoppers Purchase Intentions with Cat Boost Classifier', 2024 International Conference on Distributed Computing and Optimization Techniques (ICDCOT), Bengaluru, India, 2024, pp. 1-6. doi: 10.1109/ICDCOT61034.2024.10515309.

- In the previous study, a system was proposed to predict online shoppers' intentions (whether they will buy or not) by analyzing various classification algorithms, including Random Forest, Decision Tree, Support Vector Machine, Gradient Boosting, AdaBoost, LightGBM, and CatBoost. The experiments were conducted using an online shoppers' intention dataset. Hyperparameter tuning was performed by first selecting an appropriate assessment metric, such as accuracy or ROC-AUC, to evaluate the model's performance. Practitioners then chose a search method, such as grid search or random search, to effectively explore the hyperparameter space. The search space for each hyperparameter was defined by a range of values or distributions. Cross-validation was employed to assess the model's performance across different hyperparameter configurations while preventing overfitting. The optimal hyperparameter values were determined by splitting the dataset into training and validation sets and iteratively training the model with different configurations. The final performance of the tuned model was evaluated on a separate test set to ensure generalization to unseen data. Experimental results showed that AdaBoost, with hyperparameter tuning, achieved an accuracy of 89.14%, while CatBoost, also with hyperparameter tuning, delivered the highest accuracy of 98.73%.

- In our work, we further explore the impact of different numbers of estimators on the performance of the AdaBoost model. Specifically, we test 9 different values for n_estimators — [10, 50, 100, 150, 200, 250, 300, 350, 400] — and observe similar results. Our work achieved accuracy of 90.56% compared to 89.14%, F1 score of 66.18% compared to 63.26%, precision of 74.27% compared to 69.34%, and recall of 59.69% compared to 58.17%. This investigation helps validate the robustness of the AdaBoost model and its performance across varying configurations.

In [3]:
```
from sklearn.metrics import precision_score, recall_score

def main():
    # Load and preprocess the dataset
    file_path = '/Users/emmasun/Desktop/2060/project/online_shoppers_i
    X, y = preprocess_data(file_path)

    # Perform stratified split
    X_train, y_train, X_val, y_val, X_test, y_test = stratified_split(

    # Define the list of n_estimators to try
    n_estimators_list = [10, 50, 100, 150, 200, 250, 300, 350, 400]

    # Iterate over different values of n_estimators
    for n_estimators in n_estimators_list:
```

```
        # Initialize and train the AdaBoost model
        model = AdaBoost(n_estimators=n_estimators)
        model.train(X_train, y_train)

        # Evaluate the model on the test set only
        predictions = model.predict(X_test)

        # Compute the accuracy, F1 score, precision, and recall
        acc = model.accuracy(y_test, predictions)
        f1 = model.f1_score(y_test, predictions)
        precision = precision_score(y_test, predictions)
        recall = recall_score(y_test, predictions)

        # Print the evaluation results for the test set
        print(f"Test Set Evaluation for {n_estimators} Estimators:")
        print(f"Accuracy: {acc:.4f}, F1 Score: {f1:.4f}, Precision: {p
        print("-" * 50)

if __name__ == "__main__":
    main()
```

```
Test Set Evaluation for 10 Estimators:
Accuracy: 0.8877, F1 Score: 0.6137, Precision: 0.6567, Recall: 0.5759
--------------------------------------------------
Test Set Evaluation for 50 Estimators:
Accuracy: 0.8897, F1 Score: 0.6047, Precision: 0.6797, Recall: 0.5445
--------------------------------------------------
Test Set Evaluation for 100 Estimators:
Accuracy: 0.8881, F1 Score: 0.5988, Precision: 0.6732, Recall: 0.5393
--------------------------------------------------
Test Set Evaluation for 150 Estimators:
Accuracy: 0.8877, F1 Score: 0.5908, Precision: 0.6780, Recall: 0.5236
--------------------------------------------------
Test Set Evaluation for 200 Estimators:
Accuracy: 0.8873, F1 Score: 0.5888, Precision: 0.6769, Recall: 0.5209
--------------------------------------------------
Test Set Evaluation for 250 Estimators:
Accuracy: 0.8869, F1 Score: 0.5854, Precision: 0.6770, Recall: 0.5157
--------------------------------------------------
Test Set Evaluation for 300 Estimators:
Accuracy: 0.8857, F1 Score: 0.5816, Precision: 0.6712, Recall: 0.5131
--------------------------------------------------
Test Set Evaluation for 350 Estimators:
Accuracy: 0.8837, F1 Score: 0.5723, Precision: 0.6644, Recall: 0.5026
--------------------------------------------------
Test Set Evaluation for 400 Estimators:
Accuracy: 0.8857, F1 Score: 0.5816, Precision: 0.6712, Recall: 0.5131
--------------------------------------------------
```

5. Github repo

- https://github.com/delio05/AdaBoost

6. References

- Shalev-Shwartz, S. and Ben-David, S. (2014) Understanding Machine Learning: From Theory to Algorithms. Cambridge: Cambridge University Press.

- Swetha, T., R, R., Sajitha, T., B, V., Sravani, J. and Praveen, B. (2024) 'Forecasting Online Shoppers Purchase Intentions with Cat Boost Classifier',

2024 International Conference on Distributed Computing and Optimization Techniques (ICDCOT), Bengaluru, India, 2024, pp. 1-6. doi: 10.1109/ICDCOT61034.2024.10515309.

- Sakar, C. & Kastro, Y. (2018). Online Shoppers Purchasing Intention Dataset [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C5F88Q.

7. Contributions

- Thank you to Diksha Krishnan, Xinyu Zhou, Shen Yu, and Dongyan Sun for their equal contributions to the final project.