Decision Tree and Boosting Approaches for Predicting Bank Marketing Subscriptions

I. Introduction

Data is often non-linear, complex, and non-independent. When this happens, we cannot simply use linear regression or Naive Bayes, which assume linearity and independence. Decision trees allow us to model non-linear, complex relationships and can handle both numeric and categorical features.

The bank marketing dataset represents the marketing campaigns of a Portuguese banking institution. The dataset contains features on demographic and campaign features such as age, job type, education, marital status, previous campaign outcome, and call duration. I use this dataset to build a predictive model using a decision tree to determine whether a customer subscribes to a term deposit. Specifically, I will use a classification tree. Understanding which customers are likely to subscribe helps the bank target its marketing efforts more effectively.

II. Data Preparation

First, I check for null values. There are no null values, but several variables contain

```
Null value count:
 age               0
job                0
marital            0
education          0
default            0
housing            0
loan               0
contact            0
month              0
day_of_week        0
duration           0
campaign           0
pdays              0
previous           0
poutcome           0
emp.var.rate       0
cons.price.idx     0
cons.conf.idx      0
euribor3m          0
nr.employed        0
y                  0
```

```
'unknown' value count:
age                  0
job                330
marital             80
education         1731
default           8597
housing            990
loan               990
contact              0
month                0
day_of_week          0
duration             0
campaign             0
pdays                0
previous             0
poutcome             0
emp.var.rate         0
cons.price.idx       0
cons.conf.idx        0
euribor3m            0
nr.employed          0
y                    0
```

```
Percentage of 'unknown' values:
default      20.87
education     4.20
housing       2.40
loan          2.40
job           0.80
marital       0.19
```

"unknown" entries. The variables 'education', 'housing', 'loan', 'job', and 'marital' all contain

unknown values, but these represent a small proportion of the total values. Since the 'unknown' values may be informative in itself and the proportion is small, I will keep the unknown values for these variables. I will, however, drop the 'default' variable since it contains a much larger percentage of unknown values. In addition, this attribute highly affects the output target of whether an applicant will subscribe to a term deposit since the duration is not known before the call is made.

Next, I convert categorical variables into numerical form because some boosting algorithms require numeric inputs to perform splits. I one-hot encode the variables 'job', 'marital', 'housing', 'loan', 'contact', and 'poutcome', dropping the first variable to reduce the number of one-hot encoding variables created.

Since there is an inherent order within the 'education' variable, I label encode it according to the number of years of schooling: 'illiterate' to 0, 'basic.4y' to 1, 'basic.6y' to 2, 'basic.9y' to 3, 'high.school' to 4, 'professional.course' to 5, and 'university.degree' to 6. I map 'unknown' to the middle-value 3 because this represents a neutral level of education and prevents the model from treating ‘unknown’ as an extreme category. There is also a natural order in the variables 'month' and 'day_of_week', so I label encode them as well.

For ‘pdays’, most values are '999', meaning the client was never contacted. I do not want the model to treat these numbers as the client was contacted 999 days ago. Rather, I want the model to interpret this as the customer was never contacted. Thus, I create a new binary variable 'prev_contacted', where 0 represents the client was not contacted (999 values) and 1 represents the client was contacted. I then drop the 'pdays' variable.

The rest of the variables, 'age', 'campaign', 'pdays', 'previous', 'emp.var.rate', 'cons.price.idx', 'cons.conf.indx', 'euribor3m', and 'nr.employed' are all numeric, so I keep them as is. I do not need to standardize these variables because tree-based methods split based on order, not scale.

In addition, I create another variable called 'campaign_effort' to represent the intensity of the campaign outreach relative to the prior contact frequency. This shows us how hard the bank tried to reach a client before and during the campaign. It is derived from the 'campaign' and 'previous' variables. Finally, I convert the target variable 'y', indicating whether a customer subscribed to a term deposit, into binary form (0=customer did not subscribe, 1=customer subscribed).

Before I build the decision tree, I define the target variable as 'y', indicating a customer's subscription status, and the predictor variables as all of the other variables. The dataset is then split into 70% training, 15% validation, and 15% testing sets.

III. Baseline Model: Decision Tree

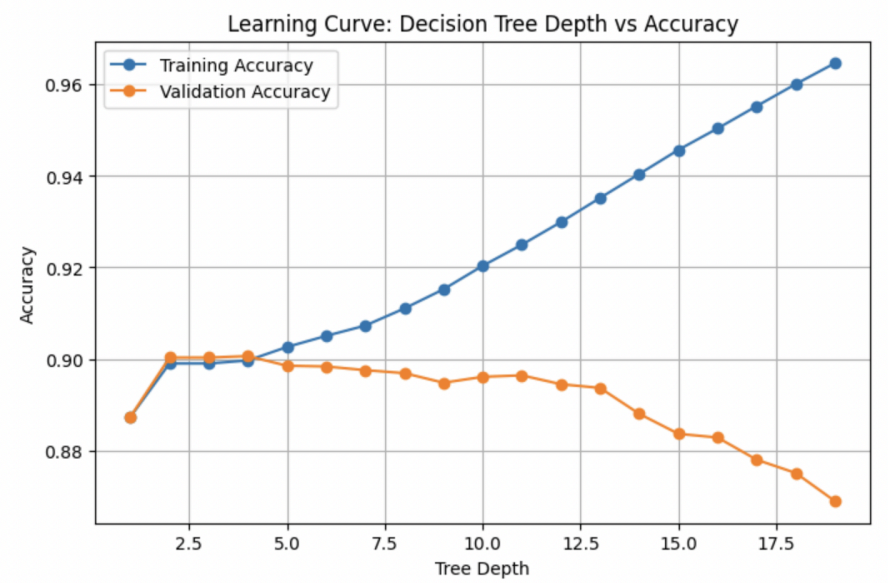
I create a baseline decision tree using the Gini impurity criterion, which measures how mixed the classes are within each node. To prevent overfitting, I tune the tree's complexity by adjusting its maximum depth, minimum number of samples required to split a node, and pruning parameters. These hyperparameters control how deeply the tree grows and help prevent it from memorizing the training data rather than generalizing to unseen examples.

I use the F1 score in addition to validation accuracy to determine the best set of parameters. Because the dataset contains far more non-subscribers, accuracy may be dominated by the majority class. The F1 score balances both precision and recall, providing a better measure of overall model performance on imbalanced data.

Based on cross-validation across parameter combinations, the best-performing parameters were max_depth = 5, min_samples_split = 2, and ccp_alpha = 0.0. In context, this means the decision tree was allowed to grow to a maximum depth of five levels, each internal node requires at least two samples to attempt a split, and no pruning is applied.

With these parameters, the model achieves a training accuracy of 0.903, validation accuracy of 0.899, and test accuracy of 0.902. These values indicate that the model is performing well. The training and test accuracy values are nearly identical, suggesting the model is not overfitting on the training data. If the training accuracy were much higher than the test accuracy, that would indicate overfitting. On the other hand, if both were much lower, the model would be underfitting. The overall high and consistent accuracies show us that the chosen parameters allow the model to balance fit and generalization effectively.

The learning curve showing accuracy versus tree depth confirms this. As seen in the graph, both training and validation accuracy start at similar levels. As the depth of the tree

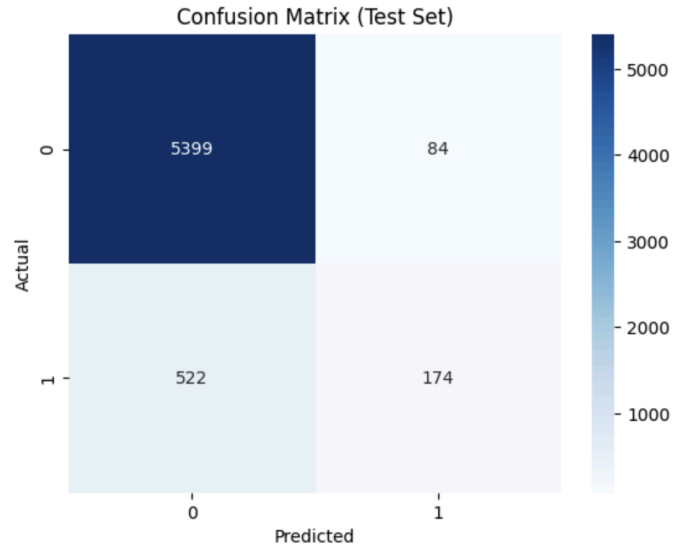


increases, however, training accuracy continues to rise while validation accuracy declines. This makes sense, as the larger the tree is, the more the tree memorizes the training data, leading to a higher training accuracy score. As the model memorizes the training data, the model's generalization ability decreases, as seen by the validation accuracy.

IV. Boosting Methods

Boosting can improve our decision tree model. Boosting involves training models sequentially, where each new model focuses on correcting the errors of the previous model. I apply both Gradient Boosting, XGBoost, and LightGBM methods to compare the performance to the baseline decision tree.

I already created a decision tree with the parameters max_depth = 5, min_samples_split = 2, and ccp_alpha = 0.0. With these parameters, the training accuracy is 0.903, validation accuracy is 0.899, and test accuracy is 0.902.



Testing Classification Report:

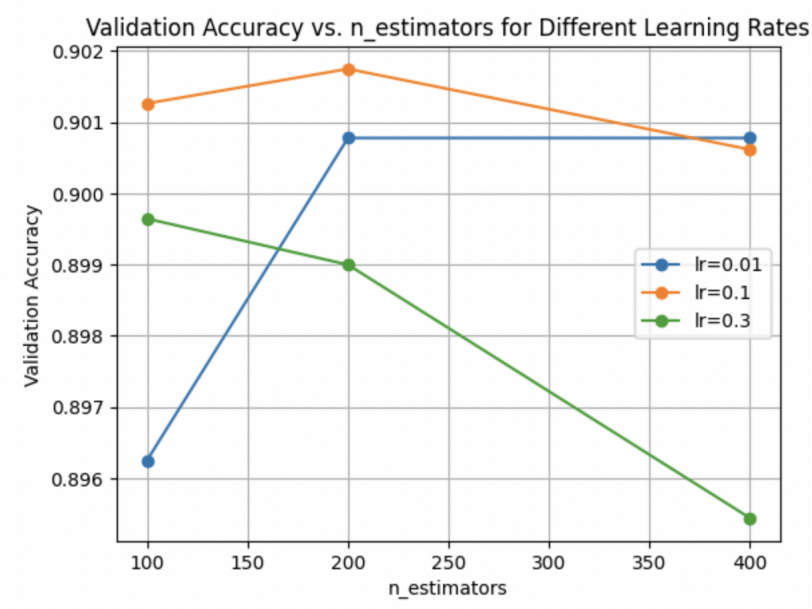
	precision	recall	f1-score	support
0	0.91	0.98	0.95	5483
1	0.64	0.25	0.36	696
accuracy			0.90	6179
macro avg	0.77	0.62	0.65	6179
weighted avg	0.88	0.90	0.88	6179

According to the testing classification report, for non-subscribers, the precision is 0.91, recall is 0.98, and f1-score is 0.95. This indicates that the model performs very well for the majority class by correctly identifying nearly all of the non-subscribers. However, the model is less effective for predicting actual subscribers. Although 64% of the predicted subscribers were correct, the recall shows that the model only correctly identifies a subscriber 25% of the time.

This behavior is also reflected in the confusion matrix, where the majority of predictions were correctly predicted as non-subscribers (true negatives). However, many subscribers were also predicted to be non-subscribers (false positives).

Gradient Boosting builds the model step by step, fitting each tree to the residual errors of the model so far. I use Gradient Boosting and experiment with different values of learning_rate, n_estimators, max_depth, and subsample. To make it easier to interpret each individual parameter, I will compare how different values for learning_rate and n_estimators affect the model and then how different values for max_depth and subsample affect the model.

Looking at how different values for learning_rate and n_estimators affect the model tells us how fast and how long the model learns, and how this affects the model's performance. As

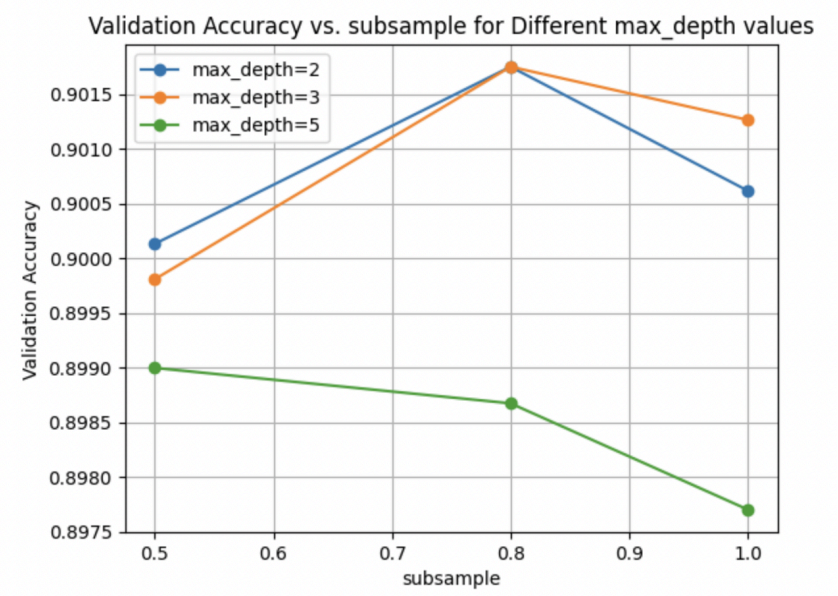


seen in the graph, the validation accuracy is the highest when the learning rate is 0.1, suggesting that this rate provides the best balance between learning speed and generalization for this dataset. The validation accuracy is also the highest when around 200 trees are added sequentially, suggesting that around 200 estimators is sufficient to reach optimal generalization without overfitting.

The model's performance varies across different combinations of learning_rate and n_estimators. Higher learning rates (0.1, 0.3) perform best when fewer trees are added sequentially (lower n_estimators), since each tree makes larger corrective updates to the residuals. Lower learning rates (0.01, 0.1) require more trees (higher n_estimators) to achieve

similar performance, as each tree contributes smaller corrections. Overall, this reflects the typical trade-off in gradient boosting: a smaller learning rate requires more boosting iterations to reduce bias, while a larger learning rate needs fewer trees but risks higher variance and overfitting. In addition, using too few trees added sequentially (low n_estimators) can lead to underfitting due to high bias, while too many trees added sequentially (high n_estimators) can reduce bias but increase variance and overfitting.

Analyzing different values for max_depth and subsample affects the model tells us the overfitting control of the model. As seen in the graph, validation accuracy is highest when the



tree depth is smaller, suggesting that deeper trees tend to overfit the training data. In terms of the subsample parameter, using only a portion of the data for each boosting iteration helps improve generalization. Models trained with a moderate subsample (around 0.8) achieve the best validation accuracy, particularly for smaller tree depths. When the subsample approaches 1.0, the model uses all data for every iteration, which reduces randomness and increases overfitting risk.

This behavior reflects how both smaller trees and partial subsampling act as forms of regularization in Gradient Boosting. Shallower trees limit the model's complexity (reducing

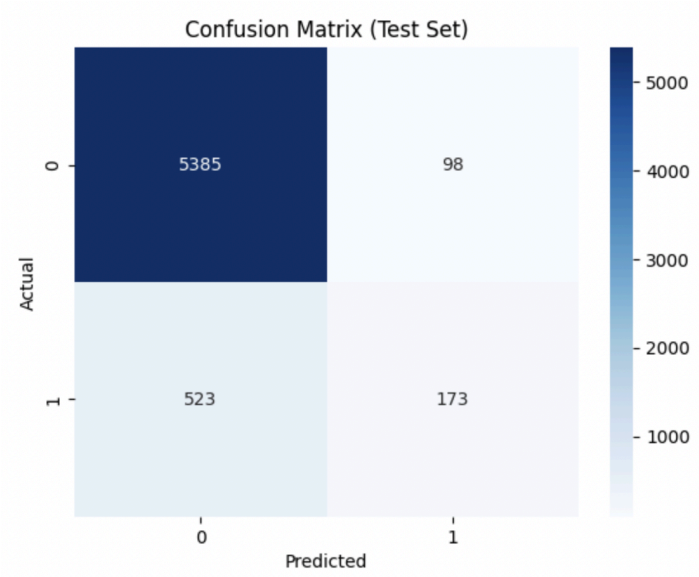
variance), while subsampling introduces randomness. Together, they help achieve a better bias–variance balance and improve validation accuracy.

I now compare which combination of all four parameters produces the best validation accuracy, using the best parameters found in the previous two analyses. This makes testing each combination take less time since I can narrow it down to 2-3 options per parameter. I want

	learning_rate	n_estimators	max_depth	subsample	train_acc	val_acc	val_recall	val_f1
19	0.30	100	3	1.0	0.910860	0.902234	0.262931	0.377320
20	0.30	200	2	0.8	0.908536	0.898511	0.248563	0.355601
21	0.30	200	2	1.0	0.907322	0.897540	0.232759	0.338558
22	0.30	200	3	0.8	0.917311	0.898996	0.277299	0.382178
23	0.30	200	3	1.0	0.916930	0.899644	0.270115	0.377510

the combination with the highest validation accuracy as well as f1. The highest values can be seen in row 19. Row 19 has learning_rate = 0.3, n_estimator=100, max_depth=3, and subsample=1.0.

With these parameters, the training accuracy is 0.911, validation accuracy is 0.902, and testing accuracy is 0.899. The training accuracy is very similar to the testing accuracy, suggesting the model is not overfitting. This is also shown by looking at the validation accuracy score, which is fairly high showing good generalization. Overall, the model is performing well with accuracy scores all in the low 0.90s.



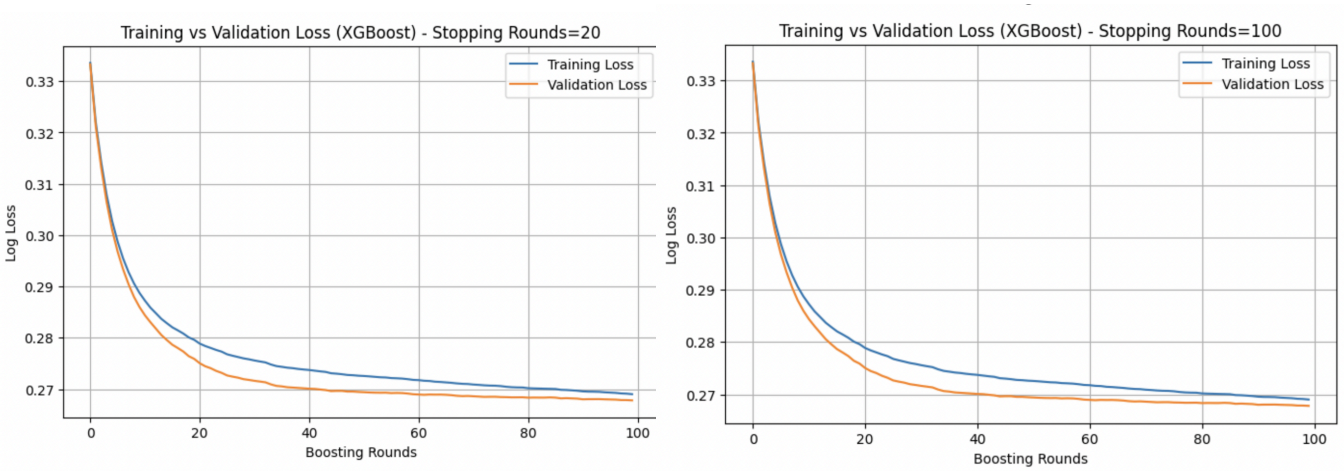
Testing Classification Report:

	precision	recall	f1-score	support
0	0.91	0.98	0.95	5483
1	0.64	0.25	0.36	696
accuracy			0.90	6179
macro avg	0.77	0.62	0.65	6179
weighted avg	0.88	0.90	0.88	6179

According to the classification report, y values of '0', the precision is 0.91, recall is 0.98, and F1 score is 0.95. This indicates that the model performs very well for the majority class by correctly identifying nearly all of the non-subscribers. However, the model is less effective for predicting actual subscribers (y = 1). Although 64% of the predicted subscribers were correct, the recall shows that the model only correctly identifies a subscriber 25% of the time.

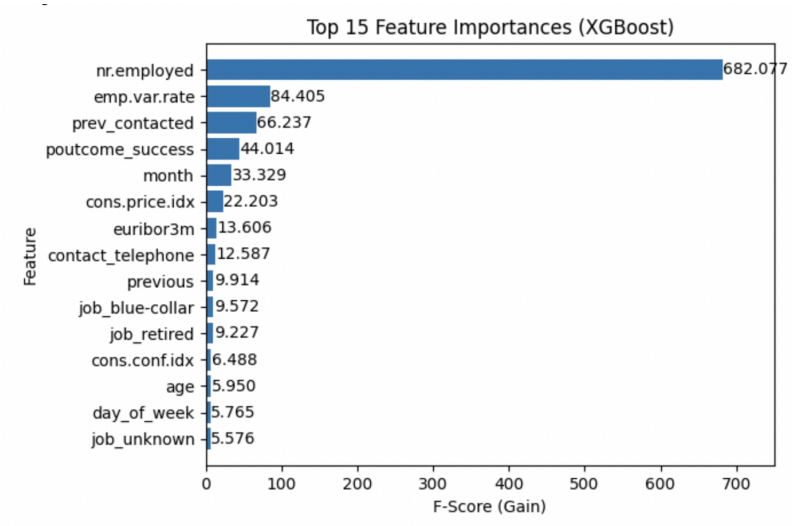
This behavior is also reflected in the confusion matrix, where the majority of predictions were correctly predicted as non-subscribers (true negatives). However, many subscribers were also predicted to be non-subscribers (false positives).

I now implement XGBoost, which enhances Gradient Boosting with regularization and faster computation. I use the same parameter values (learning_rate, n_estimators, max_depth, and subsample) as used in Gradient Boosting. However, I will experiment with two different early_stopping_rounds settings: 20 and 50.



In both training versus validation loss graphs, the validation loss does not spike as the training loss continues to drop, so there is no clear sign of overfitting in either case. Both curves decrease and gradually converge, indicating that the model generalizes well. In both graphs, the validation curve flattens around 20-30 boosting rounds. Based on this pattern, I select the early_stopping_rounds value of 20, as it effectively halts training once validation performance stabilizes while maintaining efficiency.

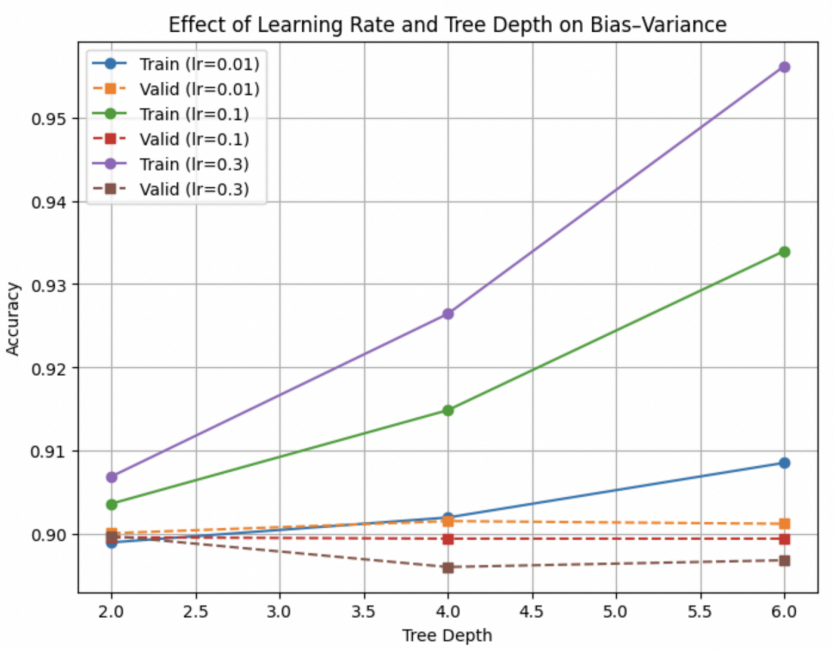
In order to get a better understanding on which variables have the strongest influence on the XGBoost model, I create a feature importance graph. The graph shows that 'nr.employed'



has the strongest influence on the model's predictions by far. This indicates that the overall employment levels play a crucial role in determining whether a custom subscribes to a term

deposit. Although much less influential in comparison, 'emp.var.rate', 'prev_contacted', and 'poutcome_success' also greatly contribute to the model's decisions. The feature importance graph allows us to better understand our predictions, and why our model behaves a certain way. If the model predicts that a customer subscribed, I can now understand that the decision was mostly influenced by the overall employment level, as well as the employment rate, whether the customer was recently contacted, and whether the outcome of the previous marketing campaign was a success. The feature importance gives us meaning behind the results.

Next, I look at how learning rate and tree depth affect the bias-variance bias. In order to do this, I test 3 different parameters for both learning_rate and max_depth. As seen in the



	learning_rate	max_depth	train_acc	val_acc	val_f1
0	0.01	2	0.899032	0.900129	0.283391
1	0.01	4	0.902015	0.901586	0.304348
2	0.01	6	0.908571	0.901263	0.342672
3	0.10	2	0.903680	0.899644	0.337607
4	0.10	4	0.914883	0.899482	0.365679
5	0.10	6	0.933960	0.899482	0.388177
6	0.30	2	0.906940	0.899806	0.365128
7	0.30	4	0.926433	0.896083	0.377907
8	0.30	6	0.956124	0.896892	0.405229

graph, as tree depth increases, the training accuracy spikes up, yet the validation accuracy stays around 0.90.

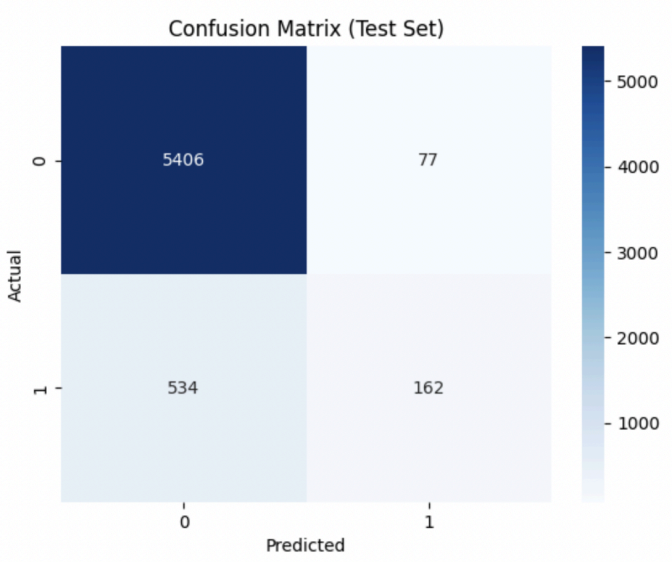
The model with a learning_rate of 0.3 is specifically prone to overfitting, as the training accuracy rapidly spikes up while the validation accuracy stays somewhat flat. This happens when the model begins to memorize the training data. In other words, the variance is high but the bias is low.

The model with a learning_rate of 0.01 suffers from underfitting, as both the training and validation curves hover around the same area. This happens when the model is too conservative (high bias) and not capturing complex patterns (low variance), even as the depth increases.

The model with a learning_rate of 0.1 and a max_depth of 4 provides the best balance. This is the spot where the training and validation accuracies remain close, minimizing both bias and variance.

After training the model with learning_rate=0.1 and max_depth=4, I examine the performance metrics. The training accuracy is 0.906, the validation accuracy is 0.902, and the testing accuracy is 0.901. The close alignment among these values indicates strong generalization and suggests that the model is fitting the data well without overfitting.

According to the classification report, y values of '0', the precision is 0.91, recall is 0.99,



Testing Classification Report:

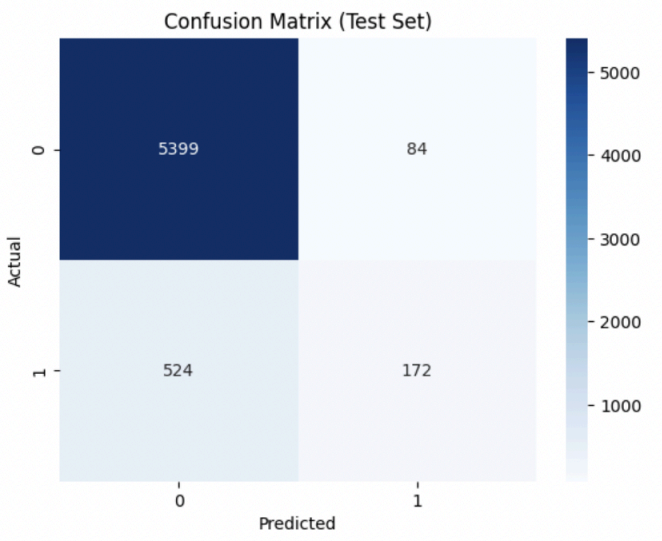
	precision	recall	f1-score	support
0	0.91	0.99	0.95	5483
1	0.68	0.23	0.35	696
accuracy			0.90	6179
macro avg	0.79	0.61	0.65	6179
weighted avg	0.88	0.90	0.88	6179

and F1 score is 0.95. This indicates that the model performs very well for the majority class by correctly identifying nearly all of the non-subscribers. However, the model is less effective for predicting actual subscribers (y = 1). Although 68% of the predicted subscribers were correct,

the recall shows that the model only correctly identifies a subscriber 23% of the time. This behavior is also reflected in the confusion matrix.

I now consider another boosting method: LightGBM. Unlike XGBoost, which grows trees depth-wise, LightGBM grows them leaf-wise. This can produce deeper trees that capture more complex patterns, but it also can create unbalanced trees, which increases the risk of overfitting if not properly regularized. However, the leaf-wise approach typically makes LightGBM faster and more memory-efficient than XGBoost, especially on large datasets.

To find the best performing model, I train the LightGBM model on different parameters for learning_rate, num_leaves, and n_estimators. The model achieves the highest validation accuracy and F1 score with learning_rate = 0.05, num_leaves = 127, and n_estimators = 400. Based on the training, validation, and testing accuracies, as well as the classification report and




```
Train Accuracy: 0.916
Validation Accuracy 0.903
Test Accuracy 0.902

Testing Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.98      0.95      5483
           1       0.67      0.25      0.36       696

    accuracy                           0.90      6179
   macro avg       0.79      0.62      0.65      6179
weighted avg       0.88      0.90      0.88      6179
```

the confusion matrix, the results are very similar to the Gradient Boosting model. Although the higher training accuracy compared to the testing accuracy suggests some slight overfitting, the overall performance of the model remains strong.

To compare the speeds of XGBoost and LightGBM, I record the training time for both methods. On average, XGBoost takes around 5 seconds, whereas LightGBM takes around 0.75

seconds. This confirms that LightGBM's leaf-wise splitting strategy trains faster while maintaining similar accuracy.

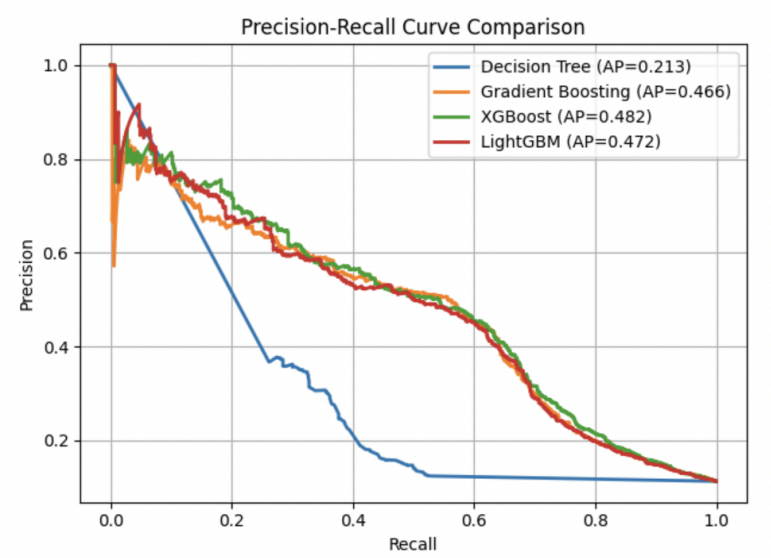
LightGBM is able to use categorical variables, allowing features to be passed as categorical data types without one-hot encoding. LightGBM assigns discrete bins and finds the best split for each category, which improves both speed and memory usage. In contrast, XGBoost requires numeric variables by one-hot or target encoding (although recent versions have added some categorical support). As for regularization, both methods apply L1 and L2 penalties to prevent overfitting, but LightGBM also provides additional controls such as min_child_samples, feature_fraction, and bagging_fraction to further constrain tree complexity.

V. Model Evaluation/Comparison

The baseline Decision Tree achieved a validation accuracy of 0.899 and a test accuracy of 0.902. Gradient Boosting achieved 0.902 and 0.899, XGBoost reached 0.902 0.901, and LightGBM achieved 0.903 and 0.902. All models have similar accuracies. In addition, all models show similar confusion matrices, dominated by true negatives, reflecting class imbalance.

Since the accuracies are similar, I look at a precision-recall curve to give us a further look into how each model performed. The PR curve is helpful when working with imbalanced datasets.

The baseline Decision Tree's PR curve drops rapidly as recall increases. This indicates that the model misses many true positives as the threshold is relaxed, which shows underfitting. This explains the Decision Tree's low AP score of 0.213. Gradient Boosting, XGBoost, and LightGBM all achieve higher PR curves. The boosting methods maintain precision at moderate recall levels, showing improved balance. There is some initial noise in both boosting models, but this can be normal for imbalanced data**.** Although Gradient Boosting, XGBoost, and LightGBM perform similarly, the XGBoost curve is slightly higher than the other curves, showing it makes



better predictions. These observations can be seen when looking at the AP values. Gradient Boosting has an AP value of 0.466, XGBoost has an AP value of 0.482, and LightGBM has an AP value of 0.472. This shows that the XGBoost model ranks positive cases (customers who subscribe to a term deposit) the highest and identifies the most true positives (true subscribers) while producing fewer false positives (incorrectly predicting non-subscribers as subscribers).

Boosting methods improve generalization by combining many weak learners into one strong predictive model. At each step, each new tree corrects the mistake of the previous tree, which gradually reduces bias. This explains why both Gradient Boosting and XGBoost models have higher AP values than the single Decision Tree, since they build an ensemble of shallow trees rather than relying on a single deep one that might overfit.

From the perspective of the bias–variance trade-off, deep trees have low bias but high variance, while shallow trees have high bias but low variance. Boosting balances this trade-off by sequentially combining many shallow trees that each contribute small corrections. By increasing the number of boosting iterations, adjusting the learning rate, or changing another parameter, the model gradually reduces bias while keeping variance under control.

Although boosting methods can improve a predictive model's performance, they take longer to compute since they train multiple trees sequentially. Gradient Boosting, in particular,

can be slow, while XGBoost improves efficiency through parallel processing and built-in regularization, and LightGBM improves efficiency even more by using a leaf-wise tree growth, reducing computation and memory usage. In contrast, a single decision tree is much faster to train and far easier to interpret. Since each split corresponds to a simple yes/no rule that can be directly visualized. With boosting, interpretability decreases because the final model uses hundreds of trees, making it difficult to trace individual decisions. However, feature importance plots help recover some interpretability by showing which variables most strongly influence the model's predictions.

VI. AI Tool Disclosure

I used chatGPT to help me generate some of the code for creating the graphs and finding the best parameter combination by iterating through different parameters. I wrote my own code for data cleaning and feature engineering steps, as well as the train/validation/test split and the creation of the decision tree models both without and with boosting. However, ChatGPT helped me generate starter code for the learning curves, the graphs comparing different model parameters, the feature importance graph, and the PR curve graph. ChatGPT also helped write code to record the time it took for models to train. I was able to create the confusion matrices and classifications reports myself this time around. Lastly, ChatGPT was helpful in suggesting which libraries and functions to import for specific tasks. The written report was done myself, although I occasionally used chatGPT to fact-check certain statements to confirm my understanding of the analysis.