

Customer Churn Classification

RAN (GREEN) GUO

Department of Mathematics, University of Los Angeles

AOS C111: Introduction to Machine Learning for Physical Sciences

Dr. Alexander Lozinski

December 4, 2025

1 Introduction

Classification is one of the most common tasks in supervised Machine Learning (ML), with applications spanning from spam detection to image recognition to disease diagnosis. A diverse set of models has been developed to address classification problems, including logistic regression, support vector machines, and decision trees.

In the field of business, ML classification techniques are particularly valuable for predicting customer behavior. For example, anticipating churn - the rate at which customers stop using a company's product or service - enables companies to identify at-risk customers. This insight allows the company to evaluate how well their products or services are received and to intervene with retention strategies that ultimately drive growth and long-term loyalty.

2 Data

This paper utilizes the Telco Customer Churn (12.1.x) sample data module from IBM Cognos Analytics. The dataset tracks 7,043 instances of a fictional California-based Telecommunications company's customer churn. The Churn Label column is binary and indicates whether a customer has stayed (Churn Label = No, Churn Value = 0) or left (Churn Label = Yes, Churn Value = 1) the company within one month of Q3 of an arbitrary year. The dataset also provides a customer's geographic, financial, and social information, as well as the types of service the customer selected. There are a total of five Excel spreadsheets that make up this dataset, which are titled demographics, location, population, services, and status.

2.1 Cleaning and Preprocessing

To prepare the data, I used the `pandas` library to load all five Excel spreadsheets. Through the unique Customer-ID, demographics, location, services, and status were merged into one master dataframe. I created a dictionary that maps each unique Zip code to its corresponding total population and then used the dictionary to add the population value to every Zip code in the master dataframe.

Next, I used the `.drop` method to get rid of redundant features, post-churn explanatory variables, and identifiers that are high-cardinality.

Lastly, I converted all features with strings into floats. For binary features, I created

a dictionary that maps Yes \rightarrow 1, No \rightarrow 0, Male \rightarrow 1, and Female \rightarrow 0. For multi-categorical features, I used `.get_dummies`, which, through one-hot encoding, returns a DataFrame where each unique category in the original data is converted into a separate column with only 0 or 1.

The resulting master dataframe contains 48 numerical features:

- | | |
|---------------------------------------|-----------------------------------|
| 1. Gender | 25. Streaming Music |
| 2. Age | 26. Unlimited Data |
| 3. Under 30 | 27. Paperless Billing |
| 4. Senior Citizen | 28. Monthly Charge |
| 5. Married | 29. Total Charges |
| 6. Dependents | 30. Total Refunds |
| 7. Number of Dependents | 31. Total Extra Data Charges |
| 8. Zip Code | 32. Total Long Distance Charges |
| 9. Latitude | 33. Total Revenue |
| 10. Longitude | 34. Satisfaction Score |
| 11. Referred a Friend | 35. Churn Value |
| 12. Number of Referrals | 36. Churn Score |
| 13. Tenure in Months | 37. CLTV |
| 14. Phone Service | 38. Population |
| 15. Avg Monthly Long Distance Charges | 39. Offer (B) |
| 16. Multiple Lines | 40. Offer (C) |
| 17. Internet Service | 41. Offer (D) |
| 18. Avg Monthly GB Download | 42. Offer (E) |
| 19. Online Security | 43. Internet Type (DSL) |
| 20. Online Backup | 44. Internet Type (Fiber Optic) |
| 21. Device Protection Plan | 45. Contract (One Year) |
| 22. Premium Tech Support | 46. Contract (Two Year) |
| 23. Streaming TV | 47. Payment Method (Credit Card) |
| 24. Streaming Movies | 48. Payment Method (Mailed Check) |

2.2 Visualization

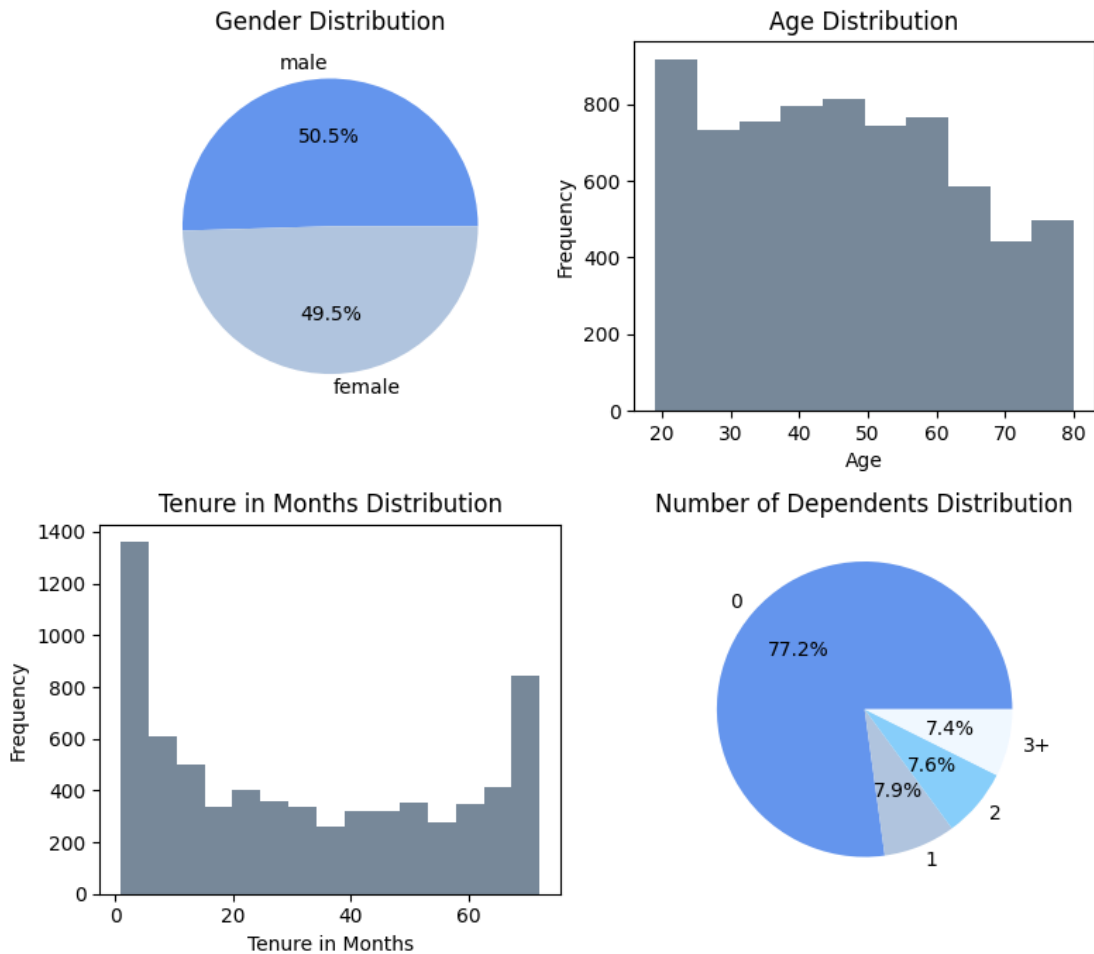


Figure 1: A 4-panel plot showing customer demographics and service characteristics. Distributions of gender (top left), age (top right), monthly tenure (bottom left), and number of dependents (bottom right).

While it is not practical to visualize every feature in graph form, the panel above highlights that the dataset represents adult customers with diverse backgrounds and varying levels of attachment to the company. The target variable in this study is Churn Value, which reflects whether a customer has stayed (Churn Value = 0) or left (Churn Value = 1). Complementing this, Churn Score provides values generated by IBM's SPSS Modeler. On a scale from 0 to 100, the higher values indicates a greater likelihood of customer departure. The histograms below display the distribution of customers who churned versus those who remained, grouped across 25 bins of Churn Score.

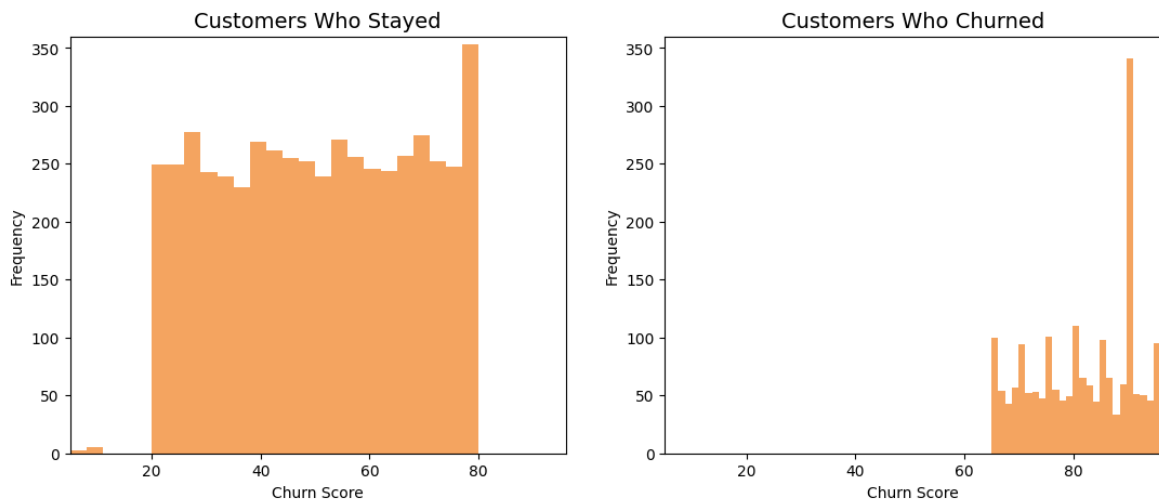


Figure 2: Histograms of churn scores for customers who stayed versus those who churned. All customers with a score above 80 ended up leaving, while all customers with a score below 65 stayed.

2.3 Preparation for Modeling

To prepare the dataset for modeling, I defined the target variable y and the feature set X by restructuring the master dataframe into two separate dataframes. Using the `sklearn.model_selection` module, I then partitioned the data into a training set (80%) and a testing set (20%).

```
from sklearn.model_selection import train_test_split

y = df["Churn Value"]
X = df.drop(columns=["Churn Value", "Churn Score"])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42,
                                                    stratify=y)
```

3 Modeling

The goal of this study is to develop a robust and accurate model capable of identifying patterns across numerous categorical features in order to predict a binary outcome from pre-labeled data. Given the nature of the task, supervised classification methods are the most appropriate approach. This section outlines the modeling pipeline, including the use of **scikit-learn** APIs, hyperparameter tuning, and model evaluation across three major classes of algorithms:

1. Logistic Regression
2. Decision Trees
3. Ensemble Methods (Random Forests, Gradient Boosting, and AdaBoost)

3.1 Logistic Regression

The **LogisticRegression** model estimates the probability of a binary outcome using the sigmoid function. This probability can be converted into a class prediction by applying a threshold, typically set at 0.5. For the binary case of customer churn, I used **solver** = 'liblinear'. The probability of the positive class is given by

$$P(y_i = 1 \mid X_i) = \frac{1}{1 + \exp(-X_i w - w_0)}.$$

The model parameters are learned by minimizing the following weighted log-loss function:

$$\frac{1}{S} \sum_{i=1}^n s_i \left[-y_i \log(\hat{p}(X_i)) - (1 - y_i) \log(1 - \hat{p}(X_i)) \right] + \frac{r(w)}{SC},$$

where s_i denotes the assigned weight for sample i , and S is the total weight across all samples. The regularization term

$$r(w) = \frac{1}{2} \|w\|_2^2 = \frac{1}{2} w^T w$$

is applied when **penalty** = l_2 . This helps improve numerical stability and reduces the risk of underfitting or overfitting.

Additionally, since Logistic Regression is sensitive to the magnitude of features, I used **ColumnTransformer** + **Pipeline** + **StandardScaler** to ensure every feature column has values on the same scale.

```

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

transformer = Pipeline(steps=[("scaler", StandardScaler())])
preprocessor = ColumnTransformer(transformers=[("num", transformer, X.columns)])

```

During the model fitting and prediction, I added a for-loop to tune the hyperparameter C , which controls the strength of regularization. This iteration process clearly shows how different levels of regularization affects model performance, allowing me to identify the setting that produced the best balance between bias and variance.

```

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

C_value = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
scores = []

for c in C_value:
    logreg_pipeline = Pipeline(steps=[
        ("preprocessor", preprocessor),
        ("classifier", LogisticRegression(
            C=c, penalty="l2", solver="liblinear", max_iter=1000))]

    logreg_pipeline.fit(X_train, y_train)

    y_pred = logreg_pipeline.predict(X_test)
    scores.append(accuracy_score(y_test, y_pred))

```

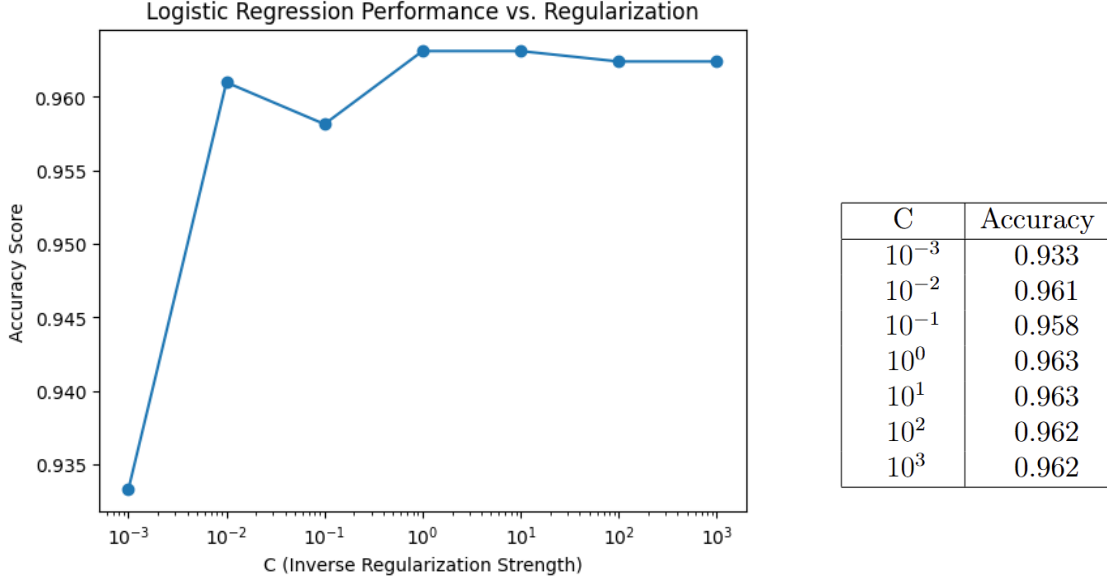


Figure 3: The accuracy of the Logistic Regression model based on different regularization strengths. Smaller values specify stronger regularization, or simpler model.

3.2 Decision Trees

Decision trees classify data by recursively splitting it into regions based on the feature and threshold that provide the greatest reduction in impurity. The `DecisionTreeClassifier` in scikit-learn implements the CART (Classification and Regression Trees) algorithm, which builds binary trees by selecting splits that maximize information gain at each node. In this study, I set `criterion = 'gini'`, meaning impurity is measured as

$$H(Q_m) = \sum_k p_{mk}(1 - p_{mk})$$

where p_{mk} represents the proportion of class k samples in node m .

The maximum allowable depth is $n_m < \min_{samples}$. Although deeper trees can capture complex patterns, they often overfit, meaning that they achieve high accuracy on the training set but fail to generalize to unseen data. To illustrate this trade-off, I trained trees with `max_depth` ranging from 1 to 20 and plotted the resulting training and testing errors.

```

from sklearn.tree import DecisionTreeClassifier

train_errors = []
test_errors = []
depths = range(1, 21)

for d in depths:
    dt = DecisionTreeClassifier(max_depth=d, criterion="gini", random_state=42)
    dt.fit(X_train, y_train)

    y_train_pred = dt.predict(X_train)
    y_test_pred = dt.predict(X_test)

    train_errors.append(1 - accuracy_score(y_train, y_train_pred))
    test_errors.append(1 - accuracy_score(y_test, y_test_pred))

```

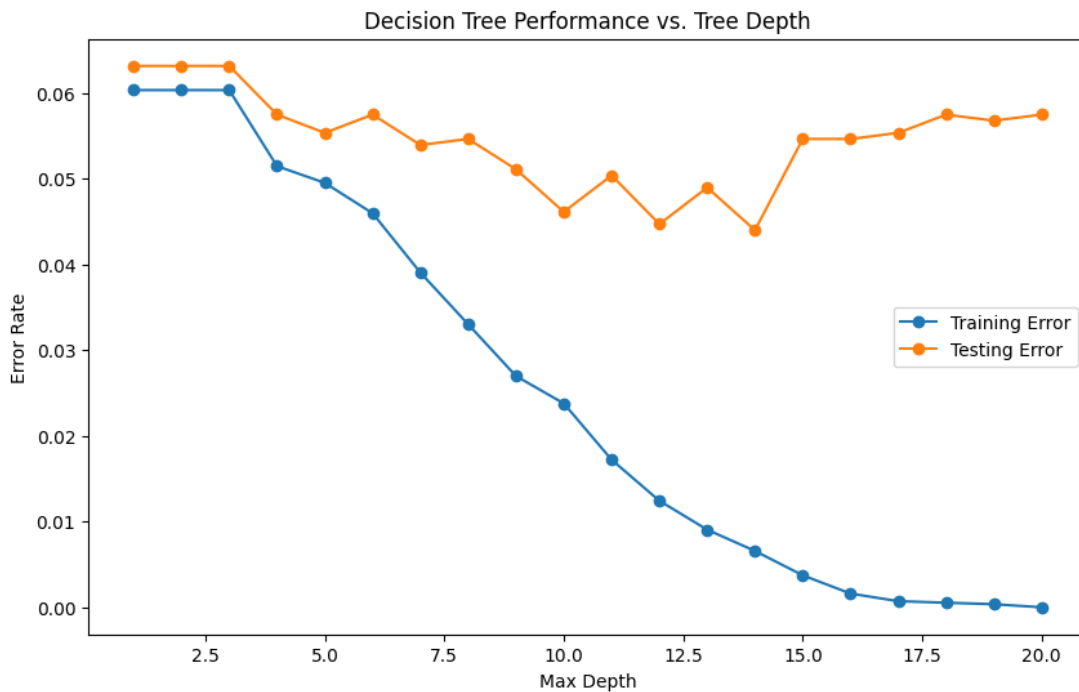


Figure 4: Training and testing error rates for decision trees with varying maximum depths (1–20). Deep trees can reach perfect accuracy during training but still perform poorly during testing due to overfitting.

In contrast to the black box problem of neural networks, decision trees are easy to visualize and interpret. The following is a figure of the model when `max_depth = 3`.

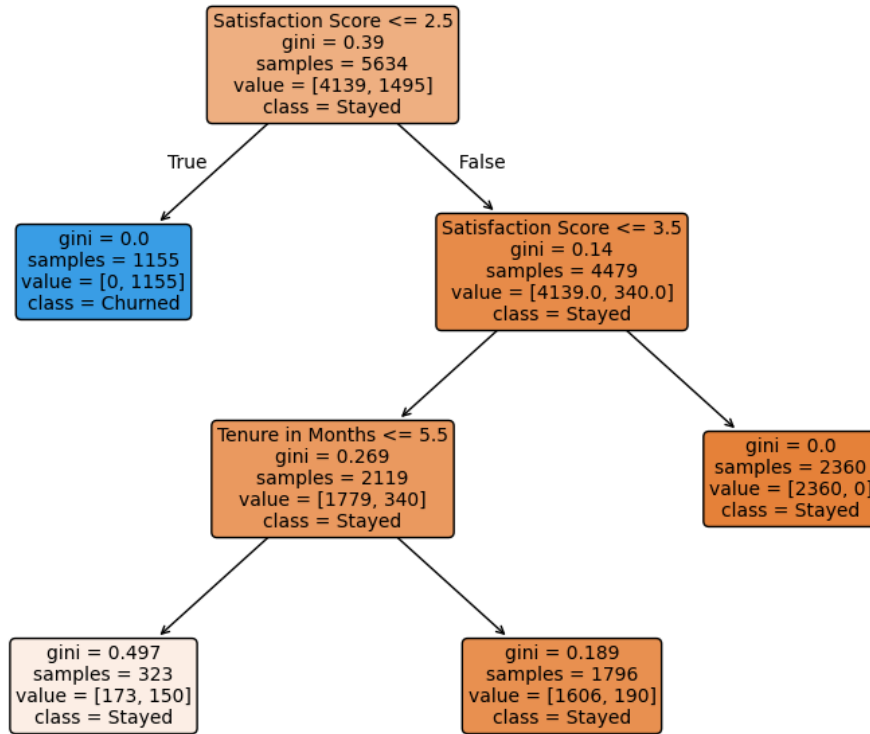


Figure 5: Decision Tree visualization with a maximum depth of 3.

3.3 Ensemble Methods

Ensemble methods combine the outputs of multiple learners, each producing slightly different results, to achieve stronger performance than any single classifier alone. In this section, I experimented with three algorithms from `sklearn.ensemble`:

1. **RandomForestClassifier** constructs an average prediction of decision trees, each trained on a bootstrap sample of the data. Bootstrapping means that each tree is trained on a random sample of the training set, drawn with replacement and equal in size to the original dataset. The decision trees is measured by the Gini impurity as described in 3.2.
2. **GradientBoostingClassifier** builds trees sequentially, where each new tree corrects the errors of the previous ones. The algorithm uses gradient descent to minimize a loss function. In the binary case, this is the log-function introduced in 3.1.

3. `AdaBoostClassifier` begins with a base classifier trained on the dataset, then iteratively fits additional classifiers while reweighting misclassified samples so that subsequent models focus more on difficult cases. Here, the base estimator is a `DecisionTreeClassifier` with stumping (`max_depth = 1`).

To begin with, I set the hyperparameter `n_estimators = 100`, the same across all three classifiers. I set `learning_rate = 0.1`, the same across the latter two.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score

rf = RandomForestClassifier(n_estimators=100, random_state=42)
gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1,
                               random_state=42)
ab = AdaBoostClassifier(n_estimators=100, learning_rate=0.1, random_state=42)

ensemble_models = {"Random Forest": rf,
                   "Gradient Boosting": gb,
                   "AdaBoost": ab }

scores = {}
for name, model in ensemble_models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    scores[name] = accuracy_score(y_test, y_pred)

for name, score in scores.items():
    print(f"{name}: Accuracy = {score:.3f}")
```

```
Random Forest: Accuracy = 0.956
Gradient Boosting: Accuracy = 0.960
AdaBoost: Accuracy = 0.937
```

Next, I trained the `RandomForestClassifier` model iteratively while increasing `n_estimators` from 1 to 100 in small intervals. This process is shown by the following code and graph.

```
n_estimators_range = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20,
                      30, 40, 50, 70, 100]
rf_scores = []

for n in n_estimators_range:
    rf = RandomForestClassifier(n_estimators=n, random_state=42)
    rf.fit(X_train, y_train)
    rf_scores.append(accuracy_score(y_test, rf.predict(X_test)))
```

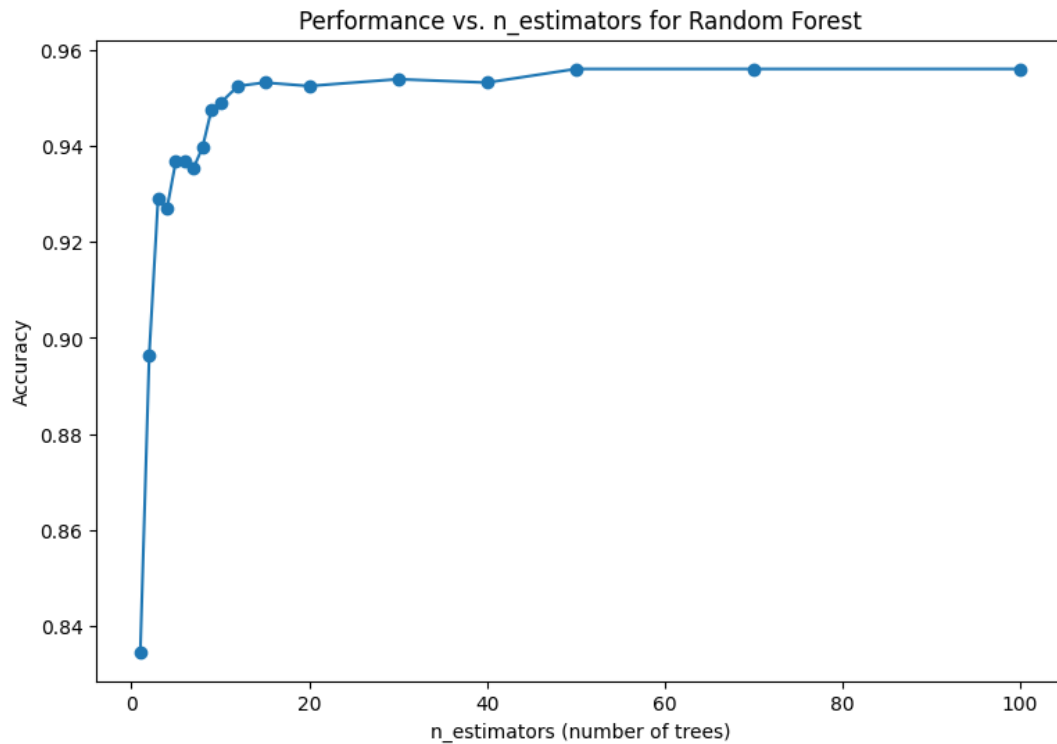


Figure 6: Accuracy of the Random Forest classifier as the number of trees increases. The plot shows that performance improves rapidly with the addition of more estimators to a certain point, leveling off when `n_estimators` ≥ 30 .

For the latter two algorithms, I compared their performance by varying `learning_rate` from 0.01 to 1.0 in small intervals. For `GradientBoostingClassifier`, `n_estimators` was set to the default of 100; for `AdaBoost`, the default of 50. There is a difference because Gradient Boosting typically requires more boosting stages to gradually reduce residual errors, whereas AdaBoost places higher emphasis on reweighting misclassified samples, allowing fewer iterations to achieve meaningful improvements. The comparison is illustrated in the following code and graph.

```
learning_rates = [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

gb_scores, ab_scores = [], []

for lr in learning_rates:
    gb = GradientBoostingClassifier(learning_rate=lr, random_state=42)
    gb.fit(X_train, y_train)
    gb_scores.append(accuracy_score(y_test, gb.predict(X_test)))

    ab = AdaBoostClassifier(learning_rate=lr, random_state=42)
    ab.fit(X_train, y_train)
    ab_scores.append(accuracy_score(y_test, ab.predict(X_test)))
```

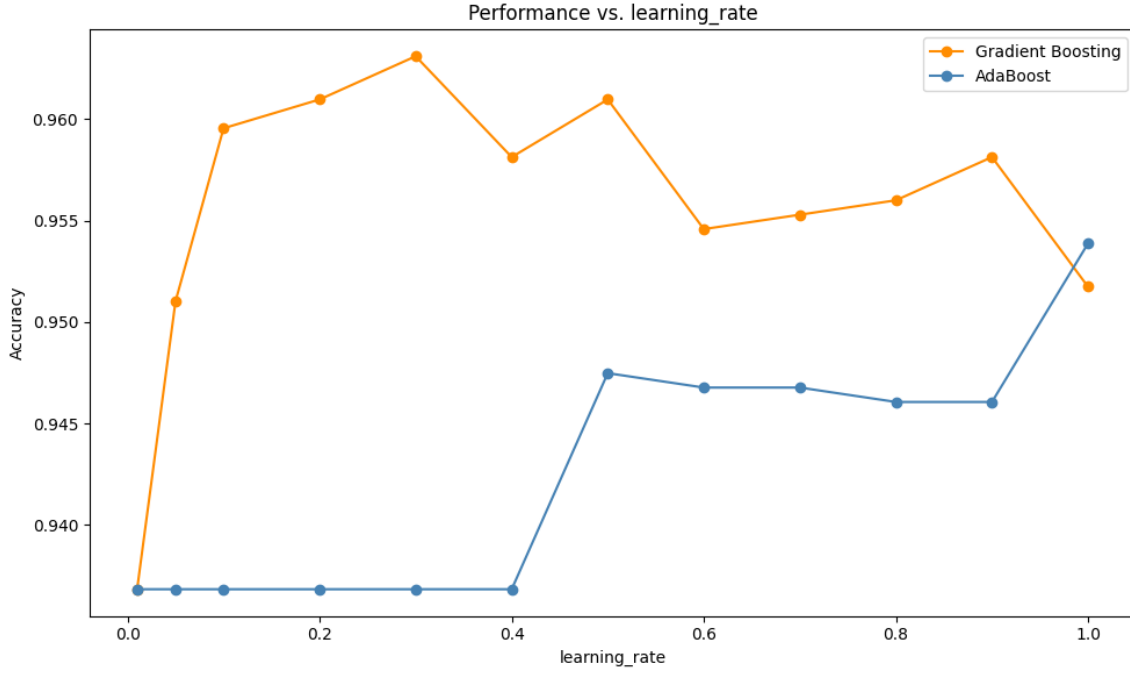


Figure 7: Accuracy of Gradient Boosting and AdaBoost classifiers across varying learning rates. Learning rate controls the contribution of each new tree to the overall model; smaller values slow learning but improve generalization, while larger values speed learning but risk overfitting.

4 Results

In the previous section, I conducted hyperparameters tuning for each of the three major classes of algorithms. By plotting performance curves, I was able to select values that balance accuracy and generalization, yielding the most effective results for each model.

1. **LogisticRegression**: from Figure 3, the model's accuracy curve is the highest at $1.0 \leq C \leq 10$. Thus, I set $C = 5.0$ as a representative value within this range.
2. **DecisionTreeClassifier**: in Figure 4, although error decreases considerably as `max_depth` increases, I want a value where the training and testing errors are most similar. To avoid overfitting, I set `max_depth = 3`.

3. Ensemble methods:

- (a) **RandomForestClassifier**: from Figure 6, the accuracy curve starts to level off at `n_estimators` ≈ 30 . I set `n_estimators` = 50, where the performance stabilizes at a slightly higher level.
- (b) **GradientBoostingClassifier**: From Figure 7, the Gradient Boosting curve reaches its peak when `learning_rate` = 0.3 with `n_estimators` = 100, reflecting the algorithm’s need for more boosting stages to gradually reduce residuals.
- (c) **AdaBoostClassifier**: In contrast, AdaBoost achieves its best performance at `learning_rate` = 1.0 with `n_estimators` = 50, consistent with its design of requiring fewer iterations than Gradient Boosting.

In addition to evaluating accuracy, I also calculated the F1 score, which is a harmonic mean of precision and recall. Both metrics have a best of 1 and a worst of 0.

Table 1: Accuracy and F1 scores for different classifiers.

	Accuracy	F1 Score
Logistic Regression	0.963	0.929
Decision Tree	0.945	0.887
Random Forest	0.956	0.911
Gradient Boosting	0.963	0.928
AdaBoost	0.954	0.909

Lastly, I plotted an ROC (Receiver Operating Characteristic) curve to illustrate how well each model distinguishes between positive and negative classes across varying thresholds. If the Area Under the Curve (AUC) is high, (i.e., close to 1.0), the model has a stronger predictive capability. The graph below depicts ROC curves that rise sharply toward the top-left corner for all five classification models.

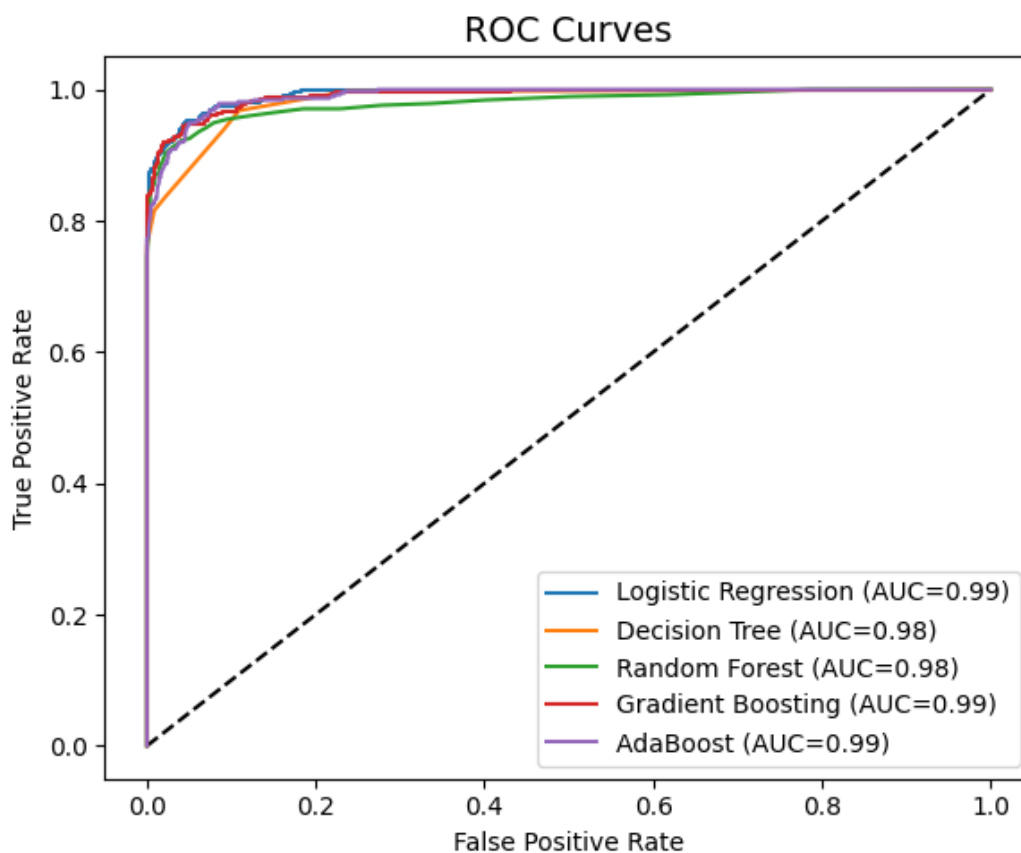


Figure 8: ROC curves for the different classifiers. The diagonal dashed line represents random guessing, while curves closer to the top-left corner indicate stronger discriminatory power.

5 Discussion

Across all methods, satisfaction score consistently ranked as the most influential variable, meaning it acts as the primary determinant of customer churn. This suggests that improving satisfaction metrics should be a priority in retention strategies for businesses.

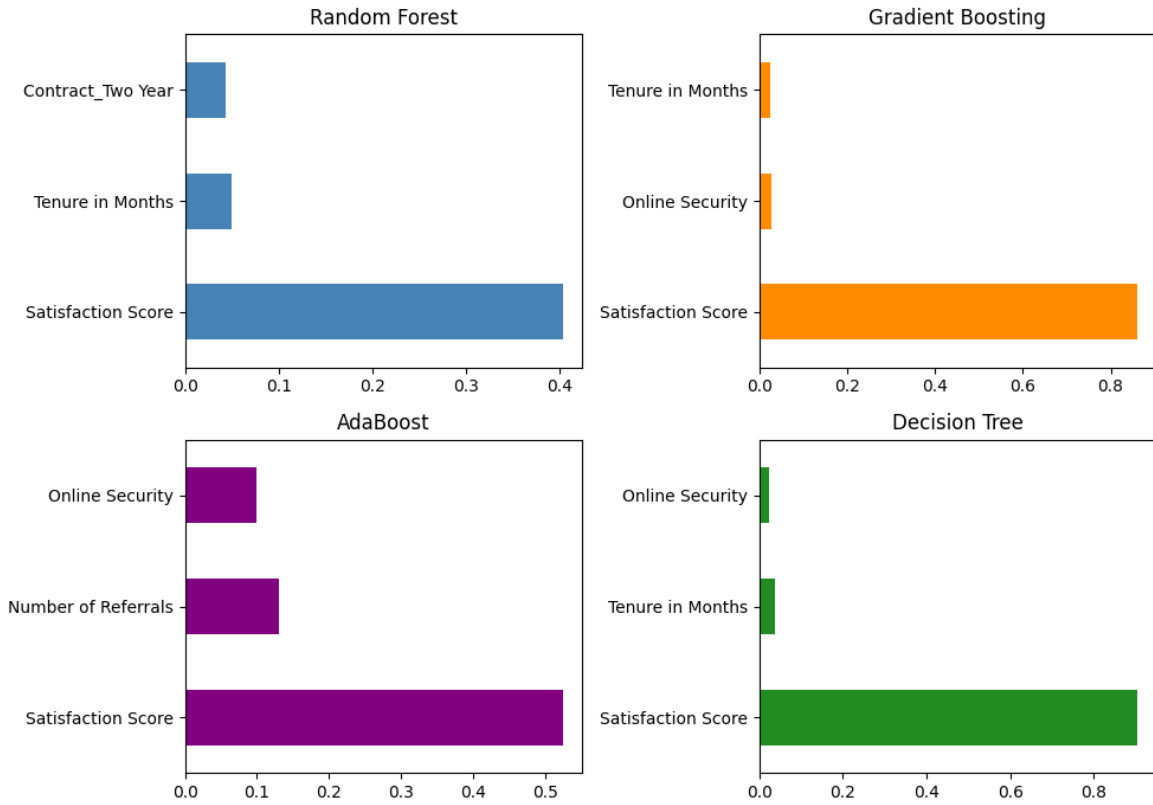


Figure 9: Top three feature importances identified by four tree-based classifiers

Next, I decided to benchmark my model against IBM's SPSS Modeler. I first converted the continuous churn score into a binary classification. Referencing Figure 2, I noticed that all customers with a score above 80 ended up leaving, while all customers with a score below 65 stayed. Thus, I set the following rule: a score of ≥ 80 indicates churn and a score of < 80 indicates retention. Using `accuracy_score`, I created the following comparison of the accuracy between my model and IBM's SPSS Modeler.

Table 2: My Logistic Regression Classifier vs IBM's SPSS Modeler

	Accuracy
Logistic Regression	0.963
SPSS Modeler	0.879

The results show that the Logistic Regression model outperformed the SPSS predictor. While the SPSS score provides a useful baseline through a fixed threshold, the Logistic Regression model leverages statistical learning and tuned hyperparameters to capture more nuanced relationships in the data. This improvement demonstrates stronger

predictive reliability and highlights the advantage of machine learning approaches over simpler models.

Despite my models' high performance, there are some limitations to consider:

1. While the learning rate is a critical hyperparameter, it is difficult to illustrate in isolation since the optimizer dynamically adjusts how each stage contributes to error reduction.
2. I did not experiment with alternative impurity measures (e.g., Gini vs. entropy) or regularization penalties, which could have influenced model stability and generalization.
3. The dataset was limited to California customers, so the model may not generalize well to populations in other regions with different behavioral patterns.

6 Conclusion

In this report, multiple machine learning models were developed and evaluated to predict customer churn using categorical variables. Logistic Regression, Decision Trees, Random Forest, Gradient Boosting, and AdaBoost were trained and assessed with metrics such as accuracy, F1-score, and ROC curves. I implemented manual hyperparameter tuning to illustrate improvement in model stability. Finally, I analyzed feature importance, identifying customer satisfaction score as the most influential predictor of churn across all algorithms.

Among the models, Logistic Regression worked particularly well in this binary setting, achieving the highest accuracy (96.3 %) and F1 score (92.9 %). In contrast, the Decision Tree model performed less well due to its greedy nature and tendency to overfit training data. All classifiers outperformed IBM's SPSS Modeler, reinforcing the strength of state-of-the-art Machine Learning methods.

Future work could include testing alternative impurity and penalty terms, introducing data samples of customers outside of California, and replacing the satisfaction score with raw behavioral and transactional variables. Since satisfaction is itself an arbitrarily computed metric, relying on direct inputs such as usage frequency, support interactions, or payment history may yield a model that can be deployed during earlier stages of prediction.

References

1. IBM, Telecommunications Industry Sample Data (DAT00148), IBM Sample Data Repository
2. Scikit-learn developers, Decision Trees — scikit-learn 1.7.2 documentation. [Online]. Available: <https://scikit-learn.org/stable/modules/tree.html>
3. Scikit-learn developers, Ensembles: Gradient boosting, random forests, bagging, voting, stacking — scikit-learn 1.7.2 documentation. [Online]. Available: <https://scikit-learn.org/stable/modules/ensemble.html>
4. Scikit-learn developers, Multiclass Receiver Operating Characteristic (ROC) — scikit-learn 1.7.2 documentation. [Online]. Available: https://scikit-learn.org/stable/modules/model_evaluation.html