SyriaTel Customer Churn Prediction - Phase 3 Project

Business Problem

Objective: Build a classifier to predict whether SyriaTel customers will "soon" stop doing business with the company.

Problem Type: Binary Classification

- Target variable: Customer churn (True/False)
- Audience: SyriaTel business stakeholders interested in reducing revenue loss

Business Context:

- Customer acquisition costs are high in telecommunications
- Retaining existing customers is more cost-effective
- Early identification of at-risk customers enables proactive retention

Data Loading and Exploration

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score,
                             roc_auc_score, confusion_matrix, classification_report)

df = pd.read_csv('bigml_data.csv')

df.head()
```

	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	...	total eve calls	total eve charge	total night minutes	total night calls
0	KS	128	415	382-4657	no	yes	25	265.1	110	45.07	...	99	16.78	244.7	91
1	OH	107	415	371-7191	no	yes	26	161.6	123	27.47	...	103	16.62	254.4	103
2	NJ	137	415	358-1921	no	no	0	243.4	114	41.38	...	110	10.30	162.6	104
3	OH	84	408	375-9999	yes	no	0	299.4	71	50.90	...	88	5.26	196.9	89
4	OK	75	415	330-6626	yes	no	0	166.7	113	28.34	...	122	12.61	186.9	121

5 rows × 21 columns

```
print("Target Variable Analysis:")
print(f"Churn distribution: {df['churn'].value_counts().to_dict()}")


df['churn'] = df['churn'].astype(str).map({'True': 1, 'False': 0})
churn_rate = df['churn'].mean()
print(f"Churn rate: {churn_rate:.1%} ({df['churn'].sum()} churned out of {len(df)} total)")

# Check for missing values
print(f"\nMissing values per column:")
missing_values = df.isnull().sum()
print(missing_values[missing_values > 0] if missing_values.sum() > 0 else "No missing values")
```

```
Target Variable Analysis:
Churn distribution: {False: 2850, True: 483}
Churn rate: 14.5% (483 churned out of 3333 total)

Missing values per column:
No missing values
```

Data Preprocessing

```
# Data preprocessing
# cleanning of the  data unnecessary columns
df = df.drop(columns=['phone number'], errors='ignore')

# Encode categorical variables
categorical_columns = ['state', 'international plan', 'voice mail plan']
label_encoders = {}

for col in categorical_columns:
    if col in df.columns:
        le = LabelEncoder()
        df[col] = le.fit_transform(df[col])
        label_encoders[col] = le
        print(f"Encoded {col}: {len(le.classes_)} unique values")

# features and target
X = df.drop('churn', axis=1)
y = df['churn']

X.shape[0]
X.shape[1]
list(X.columns)
```

```
Encoded state: 51 unique values
Encoded international plan: 2 unique values
Encoded voice mail plan: 2 unique values
['state',
 'account length',
 'area code',
 'international plan',
 'voice mail plan',
 'number vmail messages',
 'total day minutes',
 'total day calls',
 'total day charge',
 'total eve minutes',
 'total eve calls',
 'total eve charge',
 'total night minutes',
 'total night calls',
 'total night charge',
 'total intl minutes',
 'total intl calls',
 'total intl charge',
 'customer service calls']
```

```
# Train-test split with stratification (important for classification)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

print(f"Data Split:")
print(f"Training set: {len(X_train)} samples ({y_train.mean():.1%} churn rate)")
print(f"Test set: {len(X_test)} samples ({y_test.mean():.1%} churn rate)")
print(f"Stratification maintained class balance ✓")
```

```
Data Split:
Training set: 2666 samples (14.5% churn rate)
Test set: 667 samples (14.5% churn rate)
Stratification maintained class balance ✓
```

Iterative Modeling Approach

Following Phase 3 requirements, I will build multiple models iteratively:

1. **Baseline Model**: Simple, interpretable logistic regression
2. **Tuned Model**: Hyperparameter-optimized version of baseline
3. **Alternative Model**: Different algorithm for comparison

Each iteration includes justification for the approach and evaluation on both training and testing data.

Helper Functions for Model Evaluation

```
def evaluate_classification_model(name, model, X_train, X_test, y_train, y_test, use_scaling=False):
    if use_scaling:
        scaler = StandardScaler()
        X_train_eval = scaler.fit_transform(X_train)
        X_test_eval = scaler.transform(X_test)
    else:
```

```
        X_train_eval, X_test_eval = X_train, X_test

    # Train model
    model.fit(X_train_eval, y_train)

    # Training predictions (to check for overfitting)
    y_train_pred = model.predict(X_train_eval)
    y_train_proba = model.predict_proba(X_train_eval)[:, 1]

    #  predictions
    y_test_pred = model.predict(X_test_eval)
    y_test_proba = model.predict_proba(X_test_eval)[:, 1]


    metrics = {
        'train_accuracy': accuracy_score(y_train, y_train_pred),
        'test_accuracy': accuracy_score(y_test, y_test_pred),
        'train_auc': roc_auc_score(y_train, y_train_proba),
        'test_auc': roc_auc_score(y_test, y_test_proba),
        'precision': precision_score(y_test, y_test_pred),
        'recall': recall_score(y_test, y_test_pred),
        'f1_score': f1_score(y_test, y_test_pred),
        'confusion_matrix': confusion_matrix(y_test, y_test_pred)
    }

    return metrics, model

def print_model_evaluation(name, metrics):
    print(f"\n{name} - Classification Results:")
    print("-" * 50)

    # Primary classification metrics
    print(f"Training AUC:   {metrics['train_auc']:.4f}")
    print(f"Testing AUC:    {metrics['test_auc']:.4f}")
    print(f"Accuracy:       {metrics['test_accuracy']:.4f}")
    print(f"Precision:      {metrics['precision']:.4f}")
    print(f"Recall:         {metrics['recall']:.4f}")
    print(f"F1-Score:       {metrics['f1_score']:.4f}")

    # Check for overfitting
    auc_diff = metrics['train_auc'] - metrics['test_auc']
    if auc_diff > 0.05:
        print(f"Potential overfitting (AUC difference: {auc_diff:.4f})")

    # Business interpretation of confusion matrix
    cm = metrics['confusion_matrix']
    tn, fp, fn, tp = cm.ravel()
    print(f"\nBusiness Impact:")
    print(f"- True Positives:  {tp} (churns correctly identified)")
    print(f"- False Negatives: {fn} (churns missed - revenue lost)")
    print(f"- False Positives: {fp} (false alarms - wasted retention costs)")
    print(f"- True Negatives:  {tn} (loyal customers correctly identified)")
```

Model 1: Baseline Logistic Regression

```
print("MODEL 1: BASELINE LOGISTIC REGRESSION")


print("Justification: Simple, interpretable model good for binary classification")
print("This serves as our baseline to compare against")


lr_baseline = LogisticRegression(random_state=42, max_iter=1000)
metrics_baseline, model_baseline = evaluate_classification_model(
    "Baseline Logistic Regression", lr_baseline,
    X_train, X_test, y_train, y_test, use_scaling=True
)


print_model_evaluation("Baseline Logistic Regression", metrics_baseline)


results = {'Baseline_LR': metrics_baseline}
```

```
MODEL 1: BASELINE LOGISTIC REGRESSION
Justification: Simple, interpretable model good for binary classification
This serves as our baseline to compare against

Baseline Logistic Regression - Classification Results:
```

```
--------------------------------------------------
Training AUC:   0.8252
Testing AUC:    0.8166
Accuracy:       0.8591
Precision:      0.5349
Recall:         0.2371
F1-Score:       0.3286

Business Impact:
- True Positives:  23 (churns correctly identified)
- False Negatives: 74 (churns missed - revenue lost)
- False Positives: 20 (false alarms - wasted retention costs)
- True Negatives:  550 (loyal customers correctly identified)
```

Baseline Model Analysis: The baseline logistic regression provides a conservative approach with good precision but low recall. This means it's accurate when it predicts churn, but misses many actual churning customers

Model 2: Hyperparameter-Tuned Logistic Regression

```
print("\n" + "=" * 60)
print("MODEL 2: TUNED LOGISTIC REGRESSION")
print("=" * 60)
print("Justification: Improve baseline through systematic hyperparameter optimization")
print("Testing different regularization and class balancing strategies")

# Define hyperparameter grid
param_grid = {
    'C': [0.1, 1.0, 10.0],  # Regularization strength
    'class_weight': [None, 'balanced'],  # Handle class imbalance
    'solver': ['liblinear', 'lbfgs']  # Optimization algorithms
}

# Prepare scaled data for grid search
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Perform grid search with cross-validation
lr_grid = GridSearchCV(
    LogisticRegression(random_state=42, max_iter=1000),
    param_grid, cv=5, scoring='roc_auc', n_jobs=-1
)
lr_grid.fit(X_train_scaled, y_train)

print(f"Best hyperparameters found: {lr_grid.best_params_}")
print(f"Cross-validation AUC: {lr_grid.best_score_:.4f}")
print(f"Hyperparameter tuning completed using 5-fold cross-validation")
```

```
============================================================
MODEL 2: TUNED LOGISTIC REGRESSION
============================================================
Justification: Improve baseline through systematic hyperparameter optimization
Testing different regularization and class balancing strategies
Best hyperparameters found: {'C': 0.1, 'class_weight': 'balanced', 'solver': 'liblinear'}
Cross-validation AUC: 0.8179
Hyperparameter tuning completed using 5-fold cross-validation
```

```
# Evaluate tuned model on test data
y_test_pred_tuned = lr_grid.predict(X_test_scaled)
y_test_proba_tuned = lr_grid.predict_proba(X_test_scaled)[:, 1]
y_train_pred_tuned = lr_grid.predict(X_train_scaled)
y_train_proba_tuned = lr_grid.predict_proba(X_train_scaled)[:, 1]

# Calculate comprehensive metrics
metrics_tuned = {
    'train_accuracy': accuracy_score(y_train, y_train_pred_tuned),
    'test_accuracy': accuracy_score(y_test, y_test_pred_tuned),
    'train_auc': roc_auc_score(y_train, y_train_proba_tuned),
    'test_auc': roc_auc_score(y_test, y_test_proba_tuned),
    'precision': precision_score(y_test, y_test_pred_tuned),
    'recall': recall_score(y_test, y_test_pred_tuned),
    'f1_score': f1_score(y_test, y_test_pred_tuned),
    'confusion_matrix': confusion_matrix(y_test, y_test_pred_tuned)
}

print_model_evaluation("Tuned Logistic Regression", metrics_tuned)
results['Tuned_LR'] = metrics_tuned
```

```
Tuned Logistic Regression - Classification Results:
--------------------------------------------------
Training AUC:   0.8280
Testing AUC:    0.8163
Accuracy:       0.7616
Precision:      0.3510
Recall:         0.7526
F1-Score:       0.4787

Business Impact:
- True Positives:  73 (churns correctly identified)
- False Negatives: 24 (churns missed - revenue lost)
- False Positives: 135 (false alarms - wasted retention costs)
- True Negatives:  435 (loyal customers correctly identified)
```

Tuned Model Analysis: Hyperparameter tuning with class balancing significantly improved recall (ability to catch churning customers) but reduced precision (more false alarms). This represents the classic precision-recall trade-off in classification problems.

Model 3: Decision Tree Classifier

```
print("\n" + "=" * 60)
print("MODEL 3: DECISION TREE CLASSIFIER")
print("=" * 60)
print("Justification: Alternative interpretable model that handles non-linear relationships")
print("Comparing tree-based vs linear approach for this classification problem")

# Create decision tree with controls to prevent overfitting
dt = DecisionTreeClassifier(
    random_state=42,
    max_depth=5,  # Limit depth to prevent overfitting
    class_weight='balanced'  # Handle class imbalance
)

metrics_dt, model_dt = evaluate_classification_model(
    "Decision Tree", dt,
    X_train, X_test, y_train, y_test, use_scaling=False
)

print_model_evaluation("Decision Tree", metrics_dt)
results['Decision_Tree'] = metrics_dt
```

```
============================================================
MODEL 3: DECISION TREE CLASSIFIER
============================================================
Justification: Alternative interpretable model that handles non-linear relationships
Comparing tree-based vs linear approach for this classification problem

Decision Tree - Classification Results:
--------------------------------------------------
Training AUC:   0.9306
Testing AUC:    0.8049
Accuracy:       0.9055
Precision:      0.6604
Recall:         0.7216
F1-Score:       0.6897
Potential overfitting (AUC difference: 0.1257)

Business Impact:
- True Positives:  70 (churns correctly identified)
- False Negatives: 27 (churns missed - revenue lost)
- False Positives: 36 (false alarms - wasted retention costs)
- True Negatives:  534 (loyal customers correctly identified)
```

Decision Tree Analysis: The decision tree shows signs of overfitting despite depth constraints (large gap between training and test AUC). However, it achieves good recall and provides a different perspective on the classification problem.

Model Comparison and Final Selection

```
print("MODEL COMPARISON")


# Create comparison table
comparison_data = []
for name, metrics in results.items():
    comparison_data.append({
```

```
        'Model': name.replace('_', ' '),
        'Test_AUC': metrics['test_auc'],
        'Precision': metrics['precision'],
        'Recall': metrics['recall'],
        'F1_Score': metrics['f1_score'],
        'Accuracy': metrics['test_accuracy'],
        'Train_AUC': metrics['train_auc'],
        'Overfitting': metrics['train_auc'] - metrics['test_auc']
    })

comparison_df = pd.DataFrame(comparison_data)
comparison_df = comparison_df.sort_values('Test_AUC', ascending=False)

print("Model Performance Summary  by Test AUC:")
print(comparison_df.round(4).to_string(index=False))

best_model = comparison_df.iloc[0]
print(f"\n FINAL MODEL SELECTED: {best_model['Model']}")
print(f"Selected based on highest test AUC score: {best_model['Test_AUC']:.4f}")
print(f"overfitting score: {best_model['Overfitting']:.4f}")
```

```
MODEL COMPARISON
Model Performance Summary  by Test AUC:
        Model  Test_AUC  Precision  Recall  F1_Score  Accuracy  Train_AUC  Overfitting
  Baseline LR    0.8166     0.5349  0.2371    0.3286    0.8591     0.8252       0.0086
     Tuned LR    0.8163     0.3510  0.7526    0.4787    0.7616     0.8280       0.0117
Decision Tree    0.8049     0.6604  0.7216    0.6897    0.9055     0.9306       0.1257

 FINAL MODEL SELECTED: Baseline LR
Selected based on highest test AUC score: 0.8166
overfitting score: 0.0086
```

Result

After systematically building and evaluating multiple models, the **Baseline Logistic Regression** is the optimal choice for this business problem.

81.7% AUC score 53.5% precision minimal overfitting