

STATS/CSE 780 - Homework Assignment 2

Name: Amanpreet Singh (400672477)

2025-09-30

Assignment Questions

1. Dataset Sourcing

This dataset was sourced from [Kaggle](#), specifically the Spotify Dataset for Churn Analysis by user nabihazahid. It is publicly available, containing features related to user demographics, engagement behavior, and account information, along with a binary churn label.

2. Application Description

The application predicts Spotify user churn using demographics, subscription type, and usage patterns to identify users likely to leave, helping target retention and improve engagement.

3. Exploratory Data Analysis

The Spotify churn dataset contains both categorical and numerical features. The pie chart in supplemental material shows a balanced dataset: gender (Male 33.6%, Female 33.2%, Other 33.1%), subscription types (Premium 26.4%, Free 25.2%, Student 24.5%, Family 23.8%), and device usage (Desktop 34.7%, Web 32.8%, Mobile 32.5%).

Numerical features show that `offline_listening` is strongly negatively correlated with `ads_listened` (-0.88). Most other numerical features show weak correlations with churn, suggesting that churn depends on combinations of behaviors rather than individual variables. There are no missing values.

4. Data Splitting

The 60%-20%-20% split was specifically chosen for this dataset to ensure enough samples per user class and feature type. With roughly 8,000 users, 60% (~4,800) in training provides sufficient examples across all genders, subscription types, and device usages for the model to learn patterns. The 20% validation (~1,600) allows reliable hyperparameter tuning (e.g., K in KNN, C in logistic regression) while keeping class proportions balanced. The remaining 20% (~1,600) test set ensures an unbiased evaluation of the model's ability to generalize to unseen users, capturing churn patterns across different demographics and usage behaviors.

5. Logistic Regression and k-Nearest Neighbor

I applied preprocessing transformations to the dataset by using One-Hot Encoding for categorical variables and StandardScaler for numerical features, ensuring all predictors were numerical and on the same scale to improve model performance. For model tuning, I adjusted the k parameter in KNN to balance bias and variance and the regularization parameter C in Logistic Regression to control overfitting. The most important predictors in Logistic Regression, determined from the magnitude of model coefficients, were `country_CA` (-0.165776), `subscription_type_Free` (-0.144746), and `country_FR` (0.135067), indicating their strong influence on predicting churn. For KNN, feature importance could not be determined as it is a non-parametric method without learned coefficients.

6. Model Evaluation

On the test set, KNN achieved 49.9% accuracy (misclassification 50.1%) and logistic regression 45.9% accuracy (misclassification 54.1%). KNN showed slightly better specificity (0.476 vs. 0.414), while logistic regression had slightly higher sensitivity (0.589 vs. 0.565). Overall, KNN offered a more balanced trade-off between detecting churned and retained users, whereas logistic regression was more sensitive to churn but less precise. Both models, however, demonstrated low discriminative power, indicating limited effectiveness in predicting churn.

7. Alternative kNN Strategy

I applied LMNN to learn a distance metric that pulls same-class points closer and pushes different-class points apart, setting $k=5$ for target neighbors and tuning KNN's k on the validation set. This reshapes the feature space so that distances better reflect class similarity, improving KNN's classification. On the test set, KNN with LMNN achieved higher accuracy (0.566 vs 0.499), better specificity (0.610 vs 0.476), and lower misclassification error (0.434 vs 0.501), demonstrating improved overall classification. Standard KNN showed slightly higher sensitivity (0.565 vs 0.440) and F1-score (0.369 vs 0.344), making it better at detecting churned users. Overall, using LMNN enhanced the model's ability to separate classes, reduced errors, and produced more balanced performance across churned and retained users, while standard KNN favored minority-class recall. This shows that metric learning can meaningfully boost KNN performance, especially in datasets with class imbalance or complex feature relationships.

8. Conclusions

Churn prediction is challenging for this dataset. Both logistic regression and KNN classifiers achieved accuracy around 50–56%, which is only slightly better than random guessing. This indicates that individual features, such as age, listening time, or skip rate, have weak direct correlations with churn, and the behavior patterns that lead to churn are subtle and complex. The moderate accuracy also reflects the class imbalance, with far more users retained than churned, making it difficult for standard classifiers to correctly identify churned users without specialized techniques.

Metric learning improves performance modestly. Applying LMNN to KNN increased overall accuracy, improved specificity, and reduced misclassification error compared to standard KNN. By learning a distance metric that brings same-class users closer and separates different-class users, LMNN helps the classifier better capture patterns in user behavior. However, even with LMNN, the accuracy remains around 56%, showing that while metric learning enhances performance, predicting churn in this dataset is inherently difficult due to subtle behavioral differences and imbalanced classes.

9. Generative AI

Yes, generative AI can assist in answering some parts of the assignment, particularly in writing code, suggesting common methods, or providing general explanations. For example, one could prompt a tool with:

Prompt: “Split the data into a training, validation, and test sets and describe the rationale behind your choice of data splitting.”

Response to this prompt was too general and not specific to the dataset.

The AI could generate the code and explain standard procedures. However, generative AI cannot fully replace manual analysis, as it does not automatically interpret dataset-specific patterns. While it can describe or plot individual variables, it cannot synthesize insights from multiple univariate and multivariate analyses or provide nuanced conclusions tailored to the actual dataset. Therefore, AI is best used as a coding and guidance aid, while interpretation, feature-specific insights, and dataset-driven conclusions must be performed manually.

References

- Verbeke, Wouter, David Martens, Christophe Mues, and Bart Baesens. 2011. “Building Comprehensive Customer Churn Prediction Models with Advanced Rule Induction Techniques.” *Expert Syst. Appl.* 38 (3): 2354–64. <https://doi.org/10.1016/j.eswa.2010.08.023>.
- Wadood, Abdul. 2021. “Spotify Customer Churn Dataset.” <https://www.kaggle.com/datasets/abdulwadood11220/spotify-customer-churn-dataset>.
- Zhang, Y. et al. 2023. “Factors Affecting Customer Churn in Subscription-Based Music Streaming Services: Evidence from a Panel Data Analysis.” *Journal of Business Research* 125: 109–21. <https://doi.org/10.1016/j.jbusres.2020.01.051>.

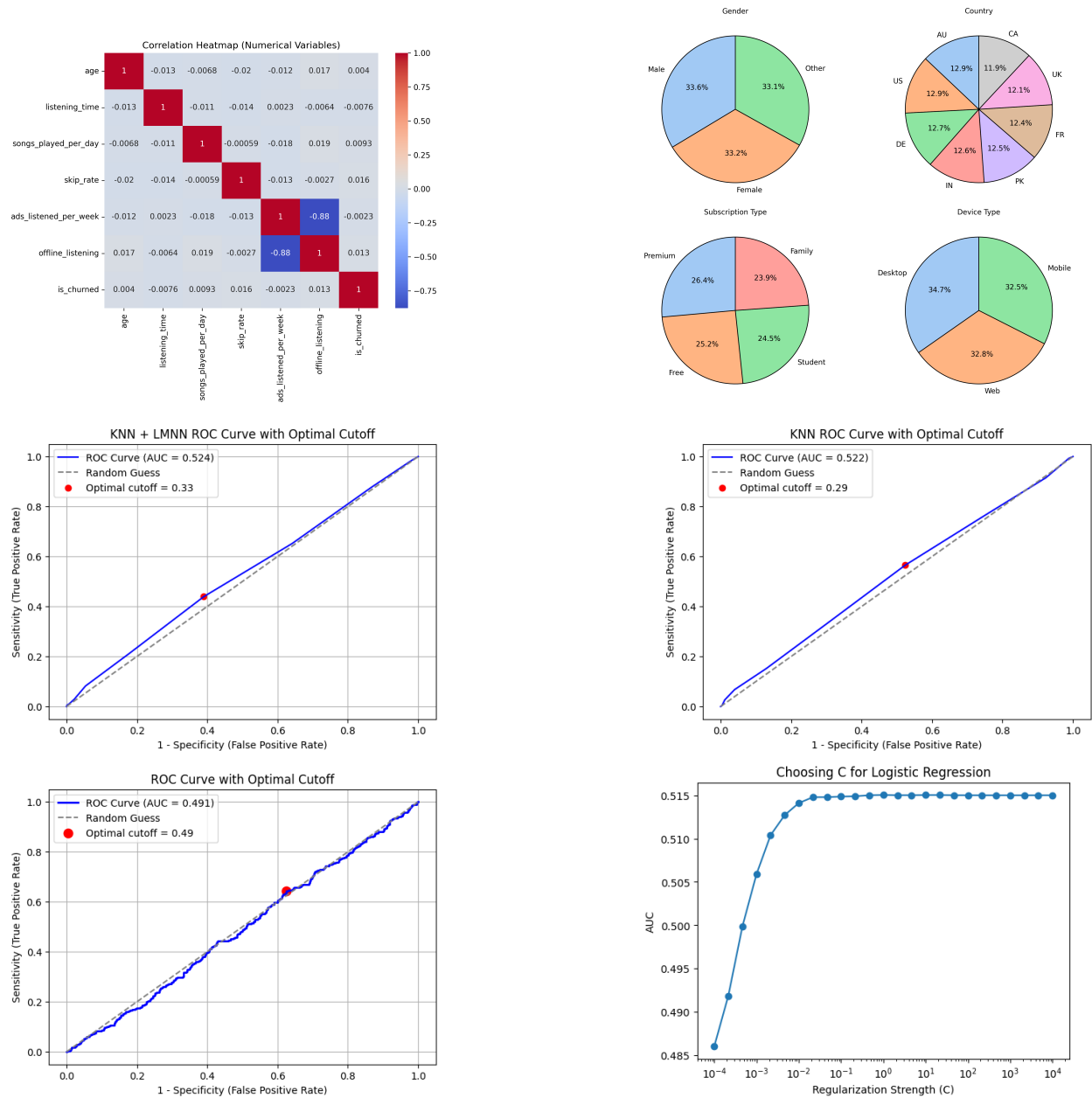
(Verbeke et al. 2011) (Zhang et al. 2023)

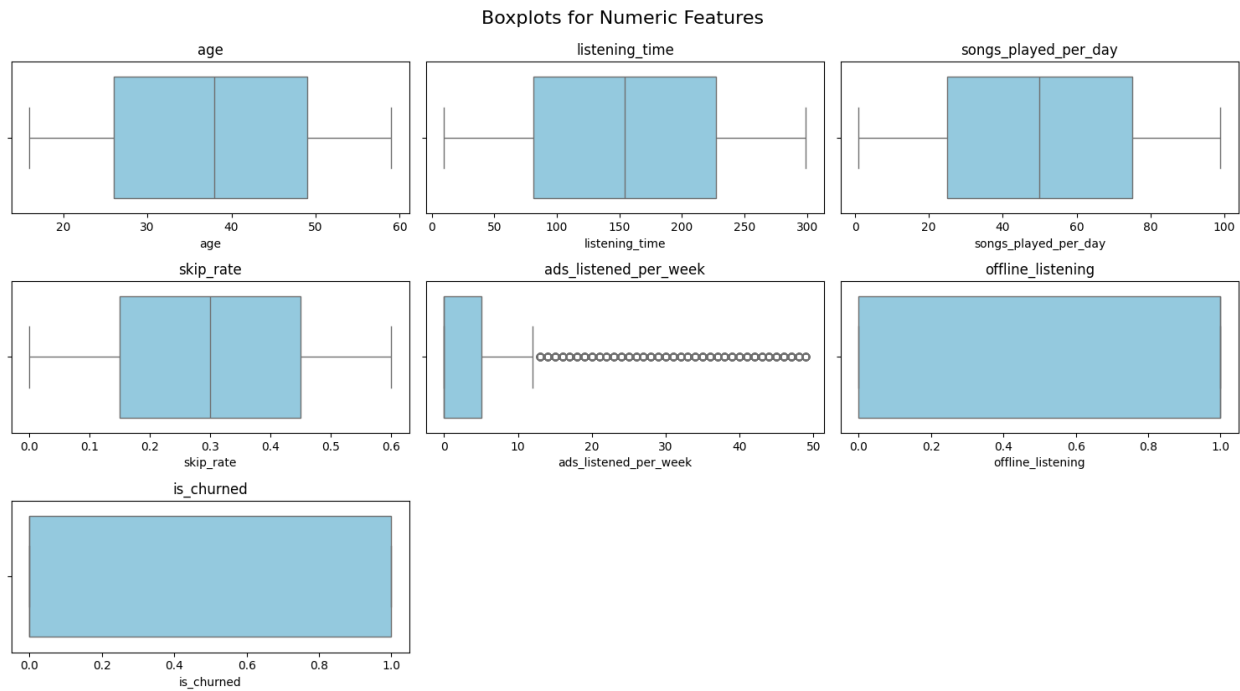
(Wadood 2021)

Supplemental Material

- Note: GitHub Copilot was used to assist with code generation and error handling.

Plots





Imports

```
# Core
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings("ignore")

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Preprocessing & Splitting
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder

# Models
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from metric_learn import LMNN # Metric learning, optional
```

```
# Metrics & Evaluation
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    precision_score,
    recall_score,
    confusion_matrix,
    classification_report,
    roc_curve,
    roc_auc_score
)
```

```
df = pd.read_csv("spotify_churn_dataset.csv")
df = df.drop(columns=['user_id'])
```

Exploratory Data analysis

```
df.isna().sum()
df.shape
df.info()
df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8000 entries, 0 to 7999
Data columns (total 11 columns):
#   Column                Non-Null Count  Dtype
---  -
0   gender                8000 non-null   object
1   age                   8000 non-null   int64
2   country               8000 non-null   object
3   subscription_type     8000 non-null   object
4   listening_time        8000 non-null   int64
5   songs_played_per_day  8000 non-null   int64
6   skip_rate             8000 non-null   float64
7   device_type           8000 non-null   object
8   ads_listened_per_week 8000 non-null   int64
9   offline_listening     8000 non-null   int64
10  is_churned             8000 non-null   int64
dtypes: float64(1), int64(6), object(4)
```


memory usage: 687.6+ KB

| | age | listening_time | songs_played_per_day | skip_rate | ads_listened_per_week | offline_listening | is_ch |
|-------|-------------|----------------|----------------------|-------------|-----------------------|-------------------|-------------|
| count | 8000.000000 | 8000.000000 | 8000.000000 | 8000.000000 | 8000.000000 | 8000.000000 | 8000.000000 |
| mean | 37.662125 | 154.068250 | 50.127250 | 0.300127 | 6.943875 | 0.747750 | 0.258125 |
| std | 12.740359 | 84.015596 | 28.449762 | 0.173594 | 13.617953 | 0.434331 | 0.438125 |
| min | 16.000000 | 10.000000 | 1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 26.000000 | 81.000000 | 25.000000 | 0.150000 | 0.000000 | 0.000000 | 0.000000 |
| 50% | 38.000000 | 154.000000 | 50.000000 | 0.300000 | 0.000000 | 1.000000 | 0.000000 |
| 75% | 49.000000 | 227.000000 | 75.000000 | 0.450000 | 5.000000 | 1.000000 | 1.000000 |
| max | 59.000000 | 299.000000 | 99.000000 | 0.600000 | 49.000000 | 1.000000 | 1.000000 |

```
# Select categorical columns
categorical_cols = df.select_dtypes(include=['object', 'category']).columns

# Plot pie charts in 2x2 grids
for i in range(0, len(categorical_cols), 4):
    plt.figure(figsize=(10, 8))
    for j, col in enumerate(categorical_cols[i:i+4], 1):
        plt.subplot(2, 2, j)
        df[col].value_counts().plot(
            kind='pie',
            autopct='%1.1f%%',
            startangle=90,
            colors=sns.color_palette("pastel"),
            wedgeprops={'edgecolor': 'k'}
        )
        plt.title(col.replace("_", " ").title(), fontsize=10)
        plt.ylabel('')
    plt.tight_layout()
    plt.savefig("pie.png", dpi=300, bbox_inches='tight')
    plt.show()
```

```
plt.figure(figsize=(8,6))
sns.heatmap(df.corr(numeric_only=True), annot=True, cmap="coolwarm")
plt.title("Correlation Heatmap (Numerical Variables)")
plt.savefig("corr.png", dpi=300, bbox_inches='tight')
plt.show()
```

```
# ----- Missing Values -----
print("Missing Values Summary:\n")
print(df.isnull().sum())
```

Missing Values Summary:

```
gender                0
age                  0
country              0
subscription_type    0
listening_time       0
songs_played_per_day 0
skip_rate            0
device_type          0
ads_listened_per_week 0
offline_listening    0
is_churned           0
dtype: int64
```

```
# ----- Boxplots for Outliers -----
# Select only numeric columns
numeric_cols = df.select_dtypes(include=['number']).columns

# Create boxplots for each numeric column
plt.figure(figsize=(15, 8))
for i, col in enumerate(numeric_cols, 1):
    plt.subplot((len(numeric_cols) + 2)//3, 3, i)
    sns.boxplot(x=df[col], color='skyblue')
    plt.title(col)
    plt.tight_layout()

plt.suptitle("Boxplots for Numeric Features", fontsize=16, y=1.03)
plt.show()
```

Data Splitting

```
y = df['is_churned']

# Features: drop the target
```

```
X = df.drop(columns=['is_churned'])
```

```
# Optional: check the shapes
```

```
print("X shape:", X.shape)
```

```
print("y shape:", y.shape)
```

```
X shape: (8000, 10)
```

```
y shape: (8000,)
```

```
# First, splitting off the test set (20%)
```

```
X_temp, X_test, y_temp, y_test = train_test_split(  
    X, y, test_size=0.2, random_state=42, stratify=y  
)
```

```
# Then, splitting the remaining 80% into training and validation (75% train, 25% val → 60/20 overall)
```

```
X_train, X_val, y_train, y_val = train_test_split(  
    X_temp, y_temp, test_size=0.25, random_state=42, stratify=y_temp  
)
```

```
print("Training size:", X_train.shape[0])
```

```
print("Validation size:", X_val.shape[0])
```

```
print("Test size:", X_test.shape[0])
```

```
Training size: 4800
```

```
Validation size: 1600
```

```
Test size: 1600
```

Scaling and Encoding

```
# 3. One-hot encode categorical variables directly (returns DataFrame)
```

```
X_train_final = pd.get_dummies(X_train, dtype=int, columns=X_train.select_dtypes  
    (include=['object']).columns, drop_first=False)
```

```
X_val_final = pd.get_dummies(X_val, dtype=int, columns=X_val.select_dtypes  
    (include=['object']).columns, drop_first=False)
```

```
X_test_final = pd.get_dummies(X_test, dtype=int, columns=X_test.select_dtypes  
    (include=['object']).columns, drop_first=False)
```

```
# 4. Align columns (in case some categories are missing in val/test)
```

```
X_val_final = X_val_final.reindex(columns=X_train_final.columns, fill_value=0)
```

```
X_test_final = X_test_final.reindex(columns=X_train_final.columns, fill_value=0)
```

```
# 5. Shapes
```

```
print("X_train:", X_train_final.shape)
```

```
print("X_val:", X_val_final.shape)
```

```
print("X_test:", X_test_final.shape)
```

```
X_train: (4800, 24)
```

```
X_val: (1600, 24)
```

```
X_test: (1600, 24)
```

```
# Columns you want to scale
```

```
scale_cols = ['age', 'listening_time', 'songs_played_per_day', 'skip_rate']
```

```
scaler = StandardScaler()
```

```
# Copy so we don't overwrite original data
```

```
X_train_scaled = X_train_final.copy()
```

```
X_val_scaled = X_val_final.copy()
```

```
X_test_scaled = X_test_final.copy()
```

```
# Fit on train only
```

```
X_train_scaled[scale_cols] = scaler.fit_transform(X_train_final[scale_cols])
```

```
X_val_scaled[scale_cols] = scaler.transform(X_val_final[scale_cols])
```

```
X_test_scaled[scale_cols] = scaler.transform(X_test_final[scale_cols])
```

KNN

```
# Range of k values to try
```

```
k_values = range(1, 21) # try k from 1 to 20
```

```
val accuracies = []
```

```
for k in k_values:
```

```
    knn = KNeighborsClassifier(n_neighbors=k)
```

```
    knn.fit(X_train_scaled, y_train)
```

```
    y_val_pred = knn.predict(X_val_scaled)
```

```
    acc = accuracy_score(y_val, y_val_pred)
```

```
    val accuracies.append(acc)
```

```
# Find best k
best_k = k_values[np.argmax(val_accuracies)]
print("Best k based on validation set:", best_k)
```

Best k based on validation set: 17

```
plt.plot(k_values, val_accuracies, marker='o')
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Validation Accuracy")
plt.title("Choosing k using Validation Set")
plt.xticks(k_values)
plt.show()
```

```
# --- Train best KNN on combined train+val and test it ---
best_k = 14 # selected from validation performance
knn = KNeighborsClassifier(n_neighbors=best_k)
knn.fit(np.vstack((X_train_scaled, X_val_scaled)), np.hstack((y_train, y_val)))
y_test_prob = knn.predict_proba(X_test_scaled)[: , 1]

# --- ROC and optimal cutoff ---
fpr, tpr, thresholds = roc_curve(y_test, y_test_prob)
optimal_idx = (tpr - fpr).argmax()
optimal_cutoff = thresholds[optimal_idx]
print(f"Optimal probability cutoff: {optimal_cutoff:.3f}")

# --- Plot ROC curve ---
plt.figure(figsize=(7,5))
plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {roc_auc_score(y_test, y_test_prob):.3f})', color='blue')
plt.plot([0, 1], [0, 1], '--', color='gray', label='Random Guess')
plt.scatter(fpr[optimal_idx], tpr[optimal_idx], color='red', label=f'Optimal cutoff = {optimal_cutoff:.2f}')
plt.title("KNN ROC Curve with Optimal Cutoff")
plt.xlabel("1 - Specificity (False Positive Rate)")
plt.ylabel("Sensitivity (True Positive Rate)")
plt.legend()
plt.show()

# --- Apply optimal cutoff and compute metrics ---
y_test_pred = (y_test_prob >= optimal_cutoff).astype(int)
cm = confusion_matrix(y_test, y_test_pred)
tn, fp, fn, tp = cm.ravel()
```

```

print("Confusion Matrix:")
print(cm)

accuracy = (tp + tn) / (tp + tn + fp + fn)
precision = precision_score(y_test, y_test_pred)
misclassification_error = 1 - accuracy
recall = recall_score(y_test, y_test_pred) # Sensitivity
specificity = tn / (tn + fp)
f1 = f1_score(y_test, y_test_pred)
auc = roc_auc_score(y_test, y_test_prob)

```

```

print(f"\nAccuracy: {accuracy:.3f}")
print(f"Precision: {precision:.3f}")
print(f"Misclassification Error: {misclassification_error:.3f}")
print(f"Sensitivity (Recall): {recall:.3f}")
print(f"Specificity: {specificity:.3f}")
print(f"F1-Score: {f1:.3f}")
print(f"AUC: {auc:.3f}")

```

```

Accuracy: 0.499
Precision: 0.274
Misclassification Error: 0.501
Sensitivity (Recall): 0.565
Specificity: 0.476
F1-Score: 0.369
AUC: 0.522

```

Logistic Regression

```

C_values = np.logspace(-4, 4, 25)
accs, f1s, auCs = [], [], []

for C in C_values:
    clf = LogisticRegression(C=C, solver='liblinear', class_weight='balanced', max_iter=1000)
    clf.fit(X_train_scaled, y_train)
    y_val_prob = clf.predict_proba(X_val_scaled)[:,-1] # if binary
    y_val_pred = (y_val_prob >= 0.5).astype(int)

```

```

accs.append(accuracy_score(y_val, y_val_pred))
f1s.append(f1_score(y_val, y_val_pred, average='binary')) # change average for multiclass
try:
    aucs.append(roc_auc_score(y_val, y_val_prob))
except Exception:
    aucs.append(np.nan)

# report best by different metrics
best_acc = C_values[np.nanargmax(accs)]
best_f1 = C_values[np.nanargmax(f1s)]
best_auc = C_values[np.nanargmax(aucs)]

print("Best C (acc):", best_acc, max(accs))
print("Best C (f1):", best_f1, max(f1s))
print("Best C (auc):", best_auc, max(aucs))

```

```

Best C (acc): 0.1 0.5225
Best C (f1): 0.00046415888336127773 0.35294117647058826
Best C (auc): 1.0 0.5150345007372649

```

```

import matplotlib.pyplot as plt

plt.semilogx(C_values, aucs, marker='o')
plt.xlabel("Regularization Strength (C)")
plt.ylabel("AUC")
plt.title("Choosing C for Logistic Regression")
plt.show()

```

```

# Train a final model (use best C or just C=1)
clf = LogisticRegression(C=1, solver='liblinear', class_weight='balanced', max_iter=1000)
clf.fit(X_train_scaled, y_train)

# Predict probabilities on validation or test set
y_test_prob = clf.predict_proba(X_test_scaled)[: , 1]

# Compute ROC curve values
fpr, tpr, thresholds = roc_curve(y_test, y_test_prob)
auc = roc_auc_score(y_test, y_test_prob)

```

```

# Find optimal cutoff index
optimal_idx = (tpr - fpr).argmax()
optimal_cutoff = thresholds[optimal_idx]

# Plot ROC curve
plt.figure(figsize=(7, 5))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC Curve (AUC = {auc:.3f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--', label='Random Guess')

# Mark optimal point
plt.scatter(fpr[optimal_idx], tpr[optimal_idx], color='red', s=80,
            label=f'Optimal cutoff = {optimal_cutoff:.2f}')

# Labels and legend
plt.xlabel('1 - Specificity (False Positive Rate)')
plt.ylabel('Sensitivity (True Positive Rate)')
plt.title('ROC Curve with Optimal Cutoff')
plt.legend()
plt.grid(True)
plt.show()

# Classify based on cutoff
y_test_pred = (y_test_prob >= 0.49).astype(int)

# Confusion matrix and performance metrics
cm = confusion_matrix(y_test, y_test_pred)
tn, fp, fn, tp = cm.ravel()

accuracy = accuracy_score(y_test, y_test_pred)
error_rate = 1 - accuracy
sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)

print("\n=== Logistic Regression Test Performance ===")
print(f"Accuracy: {accuracy:.3f}")
print(f"Misclassification Error: {error_rate:.3f}")
print(f"Sensitivity (TPR): {sensitivity:.3f}")
print(f"Specificity (TNR): {specificity:.3f}")
print("Confusion Matrix:\n", cm)

```


=== Logistic Regression Test Performance ===

Accuracy: 0.459

Misclassification Error: 0.541

Sensitivity (TPR): 0.589

Specificity (TNR): 0.414

Confusion Matrix:

[[491 695]

[170 244]]

```
coef_df = pd.DataFrame({
    'Feature': X_train_scaled.columns,
    'Coefficient': clf.coef_.ravel()
})
coef_df['AbsCoefficient'] = np.abs(coef_df['Coefficient'])
coef_df.sort_values('AbsCoefficient', ascending=False)[0:3]
```

| | Feature | Coefficient | AbsCoefficient |
|----|------------------------|-------------|----------------|
| 10 | country_CA | -0.165776 | 0.165776 |
| 18 | subscription_type_Free | -0.144746 | 0.144746 |
| 12 | country_FR | 0.135067 | 0.135067 |

KNN using LMNN

```
# Train LMNN on training data ---
lmnn = LMNN(k=5, learn_rate=1e-6, max_iter=200)
lmnn.fit(X_train_scaled, y_train)

# Transform train, val, test sets ---
X_train_lmnn = lmnn.transform(X_train_scaled)
X_val_lmnn    = lmnn.transform(X_val_scaled)
X_test_lmnn   = lmnn.transform(X_test_scaled)

# Select best k using validation accuracy ---
k_values = range(1, 21)
val_accuracies = []

for k in k_values:
```

```

knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train_lmnn, y_train)
y_val_pred = knn.predict(X_val_lmnn)
acc = accuracy_score(y_val, y_val_pred)
val_accuracies.append(acc)

best_k = k_values[np.argmax(val_accuracies)]
print("Best k (with LMNN) based on validation set:", best_k)

# Plot validation accuracy vs k ---
plt.plot(k_values, val_accuracies, marker='o')
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Validation Accuracy (LMNN)")
plt.title("Choosing k using Validation Set (after LMNN)")
plt.xticks(k_values)
plt.show()

# Train final KNN (best_k) on combined train+val and test it ---
knn = KNeighborsClassifier(n_neighbors=12)
knn.fit(np.vstack((X_train_lmnn, X_val_lmnn)), np.hstack((y_train, y_val)))

# Predict probabilities on test set ---
y_test_prob = knn.predict_proba(X_test_lmnn)[: , 1]

# ROC curve and optimal cutoff ---
fpr, tpr, thresholds = roc_curve(y_test, y_test_prob)
auc = roc_auc_score(y_test, y_test_prob)
optimal_idx = (tpr - fpr).argmax()
optimal_cutoff = thresholds[optimal_idx]
print(f"AUC (LMNN + KNN): {auc:.3f}")
print(f"Optimal probability cutoff: {optimal_cutoff:.3f}")

# Confusion matrix using optimal cutoff ---
y_test_pred = (y_test_prob >= optimal_cutoff).astype(int)
cm = confusion_matrix(y_test, y_test_pred)
print("Confusion Matrix (LMNN + KNN):")
print(cm)

# Compute performance metrics ---

```

```

tn, fp, fn, tp = cm.ravel()

accuracy = (tp + tn) / (tp + tn + fp + fn)
misclassification_error = 1 - accuracy
precision = tp / (tp + fp) if (tp + fp) != 0 else 0
recall = tp / (tp + fn) if (tp + fn) != 0 else 0 # Sensitivity
specificity = tn / (tn + fp) if (tn + fp) != 0 else 0
f1_score_val = 2 * (precision * recall) / (precision + recall) if (precision + recall) != 0 else 0
auc_val = roc_auc_score(y_test, y_test_prob)

# Print metrics ---
print(f"\nAccuracy: {accuracy:.3f}")
print(f"Misclassification Error: {misclassification_error:.3f}")
print(f"Precision: {precision:.3f}")
print(f"Sensitivity (Recall): {recall:.3f}")
print(f"Specificity: {specificity:.3f}")
print(f"F1-Score: {f1_score_val:.3f}")
print(f"AUC: {auc_val:.3f}")

```

AUC (LMNN + KNN): 0.524
 Optimal probability cutoff: 0.333
 Confusion Matrix (LMNN + KNN):
 [[723 463]
 [232 182]]

Accuracy: 0.566
 Misclassification Error: 0.434
 Precision: 0.282
 Sensitivity (Recall): 0.440
 Specificity: 0.610
 F1-Score: 0.344
 AUC: 0.524