PREDICITIVE ANALYSIS OF HEALTHCARE: PATIENT READMISSION PREDICTION

In [ ]:

Data Loading

In [2]:
```
# Importing necessary libraries
import pandas as pd
import numpy as np
from sklearn.utils import resample  # Helps in preventing model bias towards

# Loading the datasets
diabetic_data = pd.read_csv('diabetic_data.csv')
ids_mapping = pd.read_csv('IDS_mapping.csv')

# First few rows of the diabetic_data dataset
print(diabetic_data.head())
# Information about the diabetic_data dataset
print(diabetic_data.info())
# First few rows of the ids_mapping dataset
print(ids_mapping.head())
```

```
   encounter_id  patient_nbr             race  gender      age weight  \
0       2278392      8222157        Caucasian  Female   [0-10)      ?
1        149190     55629189        Caucasian  Female  [10-20)      ?
2         64410     86047875  AfricanAmerican  Female  [20-30)      ?
3        500364     82442376        Caucasian    Male  [30-40)      ?
4         16680     42519267        Caucasian    Male  [40-50)      ?

   admission_type_id  discharge_disposition_id  admission_source_id  \
0                  6                        25                    1
1                  1                         1                    7
2                  1                         1                    7
3                  1                         1                    7
4                  1                         1                    7

   time_in_hospital  ... citoglipton insulin  glyburide-metformin  \
0                 1  ...          No      No                   No
1                 3  ...          No      Up                   No
2                 2  ...          No      No                   No
3                 2  ...          No      Up                   No
4                 1  ...          No  Steady                   No
```

```
   glipizide-metformin  glimepiride-pioglitazone  metformin-rosiglitazone  \
0                   No                         No                       No
1                   No                         No                       No
2                   No                         No                       No
3                   No                         No                       No
4                   No                         No                       No

   metformin-pioglitazone  change diabetesMed readmitted
0                      No      No          No         NO
1                      No      Ch         Yes        >30
2                      No      No         Yes         NO
3                      No      Ch         Yes         NO
4                      No      Ch         Yes         NO

[5 rows x 50 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 101766 entries, 0 to 101765
Data columns (total 50 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   encounter_id              101766 non-null  int64
 1   patient_nbr               101766 non-null  int64
 2   race                      101766 non-null  object
 3   gender                    101766 non-null  object
 4   age                       101766 non-null  object
 5   weight                    101766 non-null  object
 6   admission_type_id         101766 non-null  int64
 7   discharge_disposition_id  101766 non-null  int64
 8   admission_source_id       101766 non-null  int64
 9   time_in_hospital          101766 non-null  int64
 10  payer_code                101766 non-null  object
 11  medical_specialty         101766 non-null  object
 12  num_lab_procedures        101766 non-null  int64
 13  num_procedures            101766 non-null  int64
 14  num_medications           101766 non-null  int64
 15  number_outpatient         101766 non-null  int64
 16  number_emergency          101766 non-null  int64
 17  number_inpatient          101766 non-null  int64
 18  diag_1                    101766 non-null  object
 19  diag_2                    101766 non-null  object
 20  diag_3                    101766 non-null  object
 21  number_diagnoses          101766 non-null  int64
 22  max_glu_serum             5346 non-null    object
 23  A1Cresult                 17018 non-null   object
 24  metformin                 101766 non-null  object
 25  repaglinide               101766 non-null  object
 26  nateglinide               101766 non-null  object
 27  chlorpropamide            101766 non-null  object
 28  glimepiride               101766 non-null  object
 29  acetohexamide             101766 non-null  object
```

```
 30  glipizide                 101766 non-null  object
 31  glyburide                 101766 non-null  object
 32  tolbutamide               101766 non-null  object
 33  pioglitazone              101766 non-null  object
 34  rosiglitazone             101766 non-null  object
 35  acarbose                  101766 non-null  object
 36  miglitol                  101766 non-null  object
 37  troglitazone              101766 non-null  object
 38  tolazamide                101766 non-null  object
 39  examide                   101766 non-null  object
 40  citoglipton               101766 non-null  object
 41  insulin                   101766 non-null  object
 42  glyburide-metformin       101766 non-null  object
 43  glipizide-metformin       101766 non-null  object
 44  glimepiride-pioglitazone  101766 non-null  object
 45  metformin-rosiglitazone   101766 non-null  object
 46  metformin-pioglitazone    101766 non-null  object
 47  change                    101766 non-null  object
 48  diabetesMed               101766 non-null  object
 49  readmitted                101766 non-null  object
dtypes: int64(13), object(37)
memory usage: 38.8+ MB
None
  admission_type_id    description
0                 1      Emergency
1                 2         Urgent
2                 3       Elective
3                 4        Newborn
4                 5  Not Available
```

Data Cleaning

In [3]:
```
# Replace missing value placeholders '?' with NaN for proper missing data ha
diabetic_data.replace('?', np.nan, inplace=True)

# Dropping columns with excessive missing data based on a defined threshold
missing_data_threshold = 0.9  # Threshold for dropping columns (90% missing
for column in diabetic_data.columns:
    if diabetic_data[column].isnull().mean() > missing_data_threshold:
        diabetic_data.drop(column, axis=1, inplace=True)

# Calculating and displaying the percentage of missing data for each column
missing_percentages = diabetic_data.isnull().sum() / len(diabetic_data)
print(missing_percentages[missing_percentages > 0])  # Display percentages c

# First few rows of the cleaned dataset
print(diabetic_data.head())
```

```
race                  0.022336
payer_code            0.395574
medical_specialty     0.490822
diag_1                0.000206
diag_2                0.003518
diag_3                0.013983
A1Cresult             0.832773
dtype: float64
   encounter_id  patient_nbr             race  gender      age  \
0       2278392      8222157        Caucasian  Female   [0-10)
1        149190     55629189        Caucasian  Female  [10-20)
2         64410     86047875  AfricanAmerican  Female  [20-30)
3        500364     82442376        Caucasian    Male  [30-40)
4         16680     42519267        Caucasian    Male  [40-50)

   admission_type_id  discharge_disposition_id  admission_source_id  \
0                  6                        25                    1
1                  1                         1                    7
2                  1                         1                    7
3                  1                         1                    7
4                  1                         1                    7

   time_in_hospital payer_code  ... citoglipton  insulin  glyburide-metformi
n  \
0                 1        NaN  ...          No       No                   N
o
1                 3        NaN  ...          No       Up                   N
o
2                 2        NaN  ...          No       No                   N
o
3                 2        NaN  ...          No       Up                   N
o
4                 1        NaN  ...          No   Steady                   N
o

   glipizide-metformin  glimepiride-pioglitazone  metformin-rosiglitazone  \
0                   No                        No                       No
1                   No                        No                       No
2                   No                        No                       No
3                   No                        No                       No
4                   No                        No                       No

   metformin-pioglitazone change diabetesMed readmitted
0                      No     No          No         NO
1                      No     Ch         Yes        >30
2                      No     No         Yes         NO
3                      No     Ch         Yes         NO
4                      No     Ch         Yes         NO

[5 rows x 48 columns]
```

In [4]:
```
# HANDLING THE MISSING VALUES

# Impute missing categorical values with the mode (most frequent category)
for column in ['race', 'payer_code', 'medical_specialty', 'diag_1', 'diag_2'
    mode_value = diabetic_data[column].mode()[0]
    diabetic_data[column].fillna(mode_value, inplace=True)

# Confirm no more missing values
print(diabetic_data.isnull().sum())
print(diabetic_data.head())
```

```
encounter_id                  0
patient_nbr                   0
race                          0
gender                        0
age                           0
admission_type_id             0
discharge_disposition_id      0
admission_source_id           0
time_in_hospital              0
payer_code                    0
medical_specialty             0
num_lab_procedures            0
num_procedures                0
num_medications               0
number_outpatient             0
number_emergency              0
number_inpatient              0
diag_1                        0
diag_2                        0
diag_3                        0
number_diagnoses              0
A1Cresult                 84748
metformin                     0
repaglinide                   0
nateglinide                   0
chlorpropamide                0
glimepiride                   0
acetohexamide                 0
glipizide                     0
glyburide                     0
tolbutamide                   0
pioglitazone                  0
rosiglitazone                 0
acarbose                      0
miglitol                      0
troglitazone                  0
tolazamide                    0
examide                       0
citoglipton                   0
```

```
insulin                       0
glyburide-metformin           0
glipizide-metformin           0
glimepiride-pioglitazone      0
metformin-rosiglitazone       0
metformin-pioglitazone        0
change                        0
diabetesMed                   0
readmitted                    0
dtype: int64
   encounter_id  patient_nbr             race  gender      age  \
0       2278392      8222157        Caucasian  Female   [0-10)
1        149190     55629189        Caucasian  Female  [10-20)
2         64410     86047875  AfricanAmerican  Female  [20-30)
3        500364     82442376        Caucasian    Male  [30-40)
4         16680     42519267        Caucasian    Male  [40-50)

   admission_type_id  discharge_disposition_id  admission_source_id  \
0                  6                        25                    1
1                  1                         1                    7
2                  1                         1                    7
3                  1                         1                    7
4                  1                         1                    7

   time_in_hospital payer_code  ... citoglipton  insulin  glyburide-metformi
n  \
0                 1         MC  ...          No       No                   N
o
1                 3         MC  ...          No       Up                   N
o
2                 2         MC  ...          No       No                   N
o
3                 2         MC  ...          No       Up                   N
o
4                 1         MC  ...          No   Steady                   N
o

   glipizide-metformin  glimepiride-pioglitazone  metformin-rosiglitazone  \
0                   No                        No                       No
1                   No                        No                       No
2                   No                        No                       No
3                   No                        No                       No
4                   No                        No                       No

   metformin-pioglitazone change diabetesMed readmitted
0                      No     No          No         NO
1                      No     Ch         Yes        >30
2                      No     No         Yes         NO
3                      No     Ch         Yes         NO
4                      No     Ch         Yes         NO
```

```
[5 rows x 48 columns]
```

In [5]:
```
# Manually split the dataframe into separate parts
admission_type_mapping = ids_mapping.iloc[0:8].copy()  # Assuming the first
discharge_mapping = ids_mapping.iloc[10:40].copy()  # Adjusted based on outp

# Resetting the column headers for the discharge mapping
discharge_mapping.columns = ['discharge_disposition_id', 'description']
discharge_mapping = discharge_mapping[discharge_mapping['discharge_dispositi

# Converting 'discharge_disposition_id' to integer
discharge_mapping['discharge_disposition_id'] = discharge_mapping['discharge

# Create the mapping dictionary for discharge types
discharge_dict = discharge_mapping.set_index('discharge_disposition_id')['de

# Apply the dictionary to map the descriptions
diabetic_data['discharge_disposition_id'] = diabetic_data['discharge_disposi

# Updated section of the DataFrame
print(diabetic_data[['discharge_disposition_id']].head())
```

```
  discharge_disposition_id
0                Not Mapped
1       Discharged to home
2       Discharged to home
3       Discharged to home
4       Discharged to home
```

In [6]:
```
# Manually setting up the mapping for admission types from a pre-defined dat
admission_type_mapping.columns = ['admission_type_id', 'description']  # Ren

# Ensuring only digit-containing rows are considered for mapping
admission_type_mapping = admission_type_mapping[admission_type_mapping['admi
admission_type_mapping['admission_type_id'] = admission_type_mapping['admiss

# Converting the dataframe to a dictionary for faster mapping operations
admission_type_dict = admission_type_mapping.set_index('admission_type_id')[

# Mapping admission type IDs to their descriptions in the diabetic_data data
diabetic_data['admission_type_id'] = diabetic_data['admission_type_id'].map(

# Identifying and setting up the mapping for admission source IDs
source_mapping = ids_mapping.iloc[41:61].copy()  # Copying the relevant slic
source_mapping.columns = ['admission_source_id', 'description']  # Setting a

# Filtering rows to ensure they contain digit characters only, for valid IDs
source_mapping = source_mapping[source_mapping['admission_source_id'].apply(
source_mapping['admission_source_id'] = source_mapping['admission_source_id'
```

```
# Creating a dictionary for admission source ID mappings
source_dict = source_mapping.set_index('admission_source_id')['description']

# Applying the source ID mappings to the main diabetic_data dataframe
diabetic_data['admission_source_id'] = diabetic_data['admission_source_id'].

# Displaying the updated sections of the DataFrame to verify correct mapping
print(diabetic_data[['admission_type_id', 'discharge_disposition_id', 'admis
```

```
  admission_type_id discharge_disposition_id  admission_source_id
0               NaN                Not Mapped   Physician Referral
1         Emergency       Discharged to home       Emergency Room
2         Emergency       Discharged to home       Emergency Room
3         Emergency       Discharged to home       Emergency Room
4         Emergency       Discharged to home       Emergency Room
```

In [7]:
```
# Check whether the ID '6' is present in the admission_type_dict dictionary
print("Is ID '6' present in admission_type_dict?", 6 in admission_type_dict)

# If ID '6' is missing, add a description for it. Assuming 'Not Mapped' for
if 6 not in admission_type_dict:
    admission_type_dict[6] = 'Not Mapped'  # Assign 'Not Mapped' to ID '6'

# Update the 'admission_type_id' column in the dataframe using the updated d
diabetic_data['admission_type_id'] = diabetic_data['admission_type_id'].map(

# Display the first few rows to ensure the mapping has been updated correctl
print(diabetic_data[['admission_type_id', 'discharge_disposition_id', 'admis
```

```
Is ID '6' present in admission_type_dict? True
  admission_type_id discharge_disposition_id  admission_source_id
0               NaN                Not Mapped   Physician Referral
1               NaN       Discharged to home       Emergency Room
2               NaN       Discharged to home       Emergency Room
3               NaN       Discharged to home       Emergency Room
4               NaN       Discharged to home       Emergency Room
```

Handling NaN Values and Re-Attempting the Mapping:

In [8]:
```
# Replace any missing values in 'admission_type_id' with the placeholder '-1
diabetic_data['admission_type_id'].fillna(-1, inplace=True)

# Convert the 'admission_type_id' column to integer type for consistency
diabetic_data['admission_type_id'] = diabetic_data['admission_type_id'].asty

# Confirm the data type conversion by printing the types of 'admission_type_
```

```
print("Data type in diabetic_data:", diabetic_data['admission_type_id'].dtyp
print("Data type in admission_type_dict keys:", type(list(admission_type_dic

# Reapply the mapping to 'admission_type_id' using the updated dictionary wi
diabetic_data['admission_type_id'] = diabetic_data['admission_type_id'].map(

# After mapping, replace the placeholder '-1' with 'Not Mapped' for any unas
diabetic_data['admission_type_id'].replace({None: 'Not Mapped'}, inplace=Tru

# Print a few rows to check if everything is updated and mapped correctly
print(diabetic_data[['admission_type_id', 'discharge_disposition_id', 'admis
```

```
Data type in diabetic_data: int64
Data type in admission_type_dict keys: <class 'int'>
  admission_type_id discharge_disposition_id  admission_source_id
0       Not Mapped                Not Mapped   Physician Referral
1       Not Mapped       Discharged to home       Emergency Room
2       Not Mapped       Discharged to home       Emergency Room
3       Not Mapped       Discharged to home       Emergency Room
4       Not Mapped       Discharged to home       Emergency Room
```

In [9]:
```
# Display the current contents of the admission type dictionary to review it
print("Admission Type Dictionary:", admission_type_dict)

# Print the unique values from 'admission_type_id' in the dataset to underst
print("Unique admission type IDs in diabetic_data:", diabetic_data['admissio

# Manually update the dictionary as needed based on findings from the data
# For instance, if ID '6' should correspond to 'Special' but isn't mapped, w
# admission_type_dict[6] = 'Special'

# Appling manual updates to ensure the dictionary reflects the correct mappi
diabetic_data['admission_type_id'] = diabetic_data['admission_type_id'].map(
# Replace unmapped IDs with 'Not Mapped' after reapplying the dictionary
diabetic_data['admission_type_id'].replace({None: 'Not Mapped'}, inplace=Tru

# Check the first few rows to confirm that the dictionary updates are proper
print(diabetic_data[['admission_type_id', 'discharge_disposition_id', 'admis
```

```
Admission Type Dictionary: {1: 'Emergency', 2: 'Urgent', 3: 'Elective', 4: '
Newborn', 5: 'Not Available', 6: nan, 7: 'Trauma Center', 8: 'Not Mapped'}
Unique admission type IDs in diabetic_data: ['Not Mapped']
  admission_type_id discharge_disposition_id  admission_source_id
0       Not Mapped                Not Mapped   Physician Referral
1       Not Mapped       Discharged to home       Emergency Room
2       Not Mapped       Discharged to home       Emergency Room
3       Not Mapped       Discharged to home       Emergency Room
4       Not Mapped       Discharged to home       Emergency Room
```

After the Handling process, The Corrected data

In [10]:
```
# Let's get rid of those pesky question marks by replacing them with NaN, ma
diabetic_data.replace('?', np.nan, inplace=True)

# Now, we'll convert 'admission_type_id' to an integer. I'll handle any NaNs
diabetic_data['admission_type_id'] = pd.to_numeric(diabetic_data['admission_

# I need to update our dictionary to handle these placeholders properly. Let
admission_type_dict[-1] = 'Unknown'
# Oh, and if there's a '6' that got turned into NaN somehow, let's correct t
admission_type_dict[6] = 'Not Available'

# Time to reapply our mapping with the updated dictionary to make sure every
diabetic_data['admission_type_id'] = diabetic_data['admission_type_id'].map(

# Let's check our work and make sure our mapping looks good.
print("Updated Admission Type IDs:", diabetic_data['admission_type_id'].unic
print(diabetic_data[['admission_type_id', 'discharge_disposition_id', 'admis
```

```
Updated Admission Type IDs: ['Unknown']
  admission_type_id discharge_disposition_id  admission_source_id
0           Unknown                Not Mapped   Physician Referral
1           Unknown       Discharged to home       Emergency Room
2           Unknown       Discharged to home       Emergency Room
3           Unknown       Discharged to home       Emergency Room
4           Unknown       Discharged to home       Emergency Room
```

In [11]:
```
# Let's reapply these mappings to the 'discharge_disposition_id' and 'admiss
diabetic_data['discharge_disposition_id'] = diabetic_data['discharge_disposi
diabetic_data['admission_source_id'] = diabetic_data['admission_source_id'].

# Sometimes, the mapping doesn't find a match and returns None, so let's rep
diabetic_data['discharge_disposition_id'].replace({None: 'Not Mapped'}, inpl
diabetic_data['admission_source_id'].replace({None: 'Not Mapped'}, inplace=T

# Finally, let's have a look at the data to make sure all our mappings are a
print(diabetic_data[['admission_type_id', 'discharge_disposition_id', 'admis
```

```
  admission_type_id discharge_disposition_id admission_source_id
0           Unknown                Not Mapped          Not Mapped
1           Unknown                Not Mapped          Not Mapped
2           Unknown                Not Mapped          Not Mapped
3           Unknown                Not Mapped          Not Mapped
4           Unknown                Not Mapped          Not Mapped
```

Feature Engineering Begins

In [12]:
```
# First up, let's turn the 'age' categories into something more quantifiable
age_mapping = {
    '[0-10)': 5, '[10-20)': 15, '[20-30)': 25, '[30-40)': 35,
    '[40-50)': 45, '[50-60)': 55, '[60-70)': 65, '[70-80)': 75, '[80-90)': 8
}
diabetic_data['age'] = diabetic_data['age'].map(age_mapping)

# Now, let's create a new feature that sums up all the services a patient us
diabetic_data['total_services'] = diabetic_data['number_outpatient'] + diabe

# Let's also track medication changes. This feature counts how many differen
medications = ['metformin', 'repaglinide', 'nateglinide', 'chlorpropamide',
               'acetohexamide', 'glipizide', 'glyburide', 'tolbutamide', 'pi
               'rosiglitazone', 'acarbose', 'miglitol', 'troglitazone', 'tol
               'examide', 'citoglipton', 'insulin', 'glyburide-metformin', '
               'glimepiride-pioglitazone', 'metformin-rosiglitazone', 'metfo
diabetic_data['medication_changes'] = diabetic_data[medications].apply(lambd

# Let's take a quick look at the first few rows to make sure everything's lo
print(diabetic_data[['age', 'total_services', 'medication_changes']].head())
```

```
   age  total_services  medication_changes
0    5               0                   0
1   15               0                   1
2   25               3                   1
3   35               0                   1
4   45               0                   2
```

Model Development Starts

LOGISTIC REGRESSION

In [13]:
```
def sigmoid(z):
    """Apply the sigmoid function, turning any real-valued number into a val
    return 1 / (1 + np.exp(-z))

def compute_cost(X, y, theta):
    """Calculate how far off our predictions are from the actual results, wh
    m = len(y)  # total number of data points
    h = sigmoid(X.dot(theta))  # predicted probabilities of the positive cla
    epsilon = 1e-5  # a tiny number to prevent any division by zero in the l
    cost = (1/m) * ((-y).T.dot(np.log(h + epsilon)) - (1 - y).T.dot(np.log(1
    return cost

def gradient_descent(X, y, theta, alpha, num_iterations):
```

```
    """Optimize the theta parameters of our model by moving them towards the
    m = len(y)  # number of observations
    cost_history = []  # to record the cost at each iteration

    for i in range(num_iterations):  # iterate over the number of times spec
        predictions = sigmoid(X.dot(theta))  # current predictions according
        errors = predictions - y  # difference between predictions and actua
        updates = (alpha / m) * (X.T.dot(errors))  # adjustment to apply to
        theta -= updates  # update theta to a new value
        cost_history.append(compute_cost(X, y, theta))  # keep track of our

    return theta, cost_history

# Setting up our data for logistic regression
# Adding an intercept column to our dataset, because it's needed for the mod
diabetic_data['intercept'] = 1
features = ['intercept', 'age', 'total_services', 'medication_changes']  # f
X = diabetic_data[features].values  # feature matrix
y = diabetic_data['readmitted'].apply(lambda x: 1 if x == '>30' else 0).valu

# Initialize theta to zero for all features
initial_theta = np.zeros(X.shape[1])

# Set our learning rate and the number of iterations for gradient descent
alpha = 0.01  # how fast we want to update theta
iterations = 1000  # how many iterations to run the optimization

# Execute gradient descent to find the optimal theta values
theta, cost_history = gradient_descent(X, y, initial_theta, alpha, iteration

# Print out the results to see how we did
print("RESULTING PARAMETERS: ")
print("Theta:", theta)  # final values of theta after optimization
print("Cost History:", cost_history[-10:])  # show the cost values of the la
```

```
RESULTING PARAMETERS:
Theta: [-0.07015869 -0.06715746  0.62803849  0.05078101]
Cost History: [1.3430558303075109, 5.691248249792388, 3.9108817789471546, 1.
343731664084895, 5.690161187415041, 3.9108430743702236, 1.3444058436656765,
5.689076506531905, 3.910804531699697, 1.3450783723473239]
```

FEATURE SCALING AND ADJUSTING GRADIENT DESC

In [14]:
```
# Let's normalize our features so they're all on the same scale. This helps
mean_age = np.mean(X[:, 1])  # Calculate the mean of the 'age' feature
std_age = np.std(X[:, 1])  # Calculate the standard deviation of 'age'
X[:, 1] = (X[:, 1] - mean_age) / std_age  # Scale 'age' to have mean 0 and s

mean_services = np.mean(X[:, 2])  # Calculate the mean for 'total services'
std_services = np.std(X[:, 2])  # And its standard deviation
```

```
X[:, 2] = (X[:, 2] - mean_services) / std_services  # Normalize 'total servi

mean_changes = np.mean(X[:, 3])  # Mean of medication changes
std_changes = np.std(X[:, 3])  # Standard deviation for medication changes
X[:, 3] = (X[:, 3] - mean_changes) / std_changes  # Normalize 'medication ch

# We're going to tweak our learning parameters for possibly better results.
alpha = 0.001  # I'm reducing the learning rate to improve stability in our
iterations = 5000  # And increasing the number of iterations for a more grad

# Now, let's run the gradient descent again with our adjusted settings and s
theta, cost_history = gradient_descent(X, y, initial_theta, alpha, iteratior

# Let's check out the new parameters and how the cost has changed over the l
print("Adjusted Theta:", theta)  # New values of theta after optimization
print("Adjusted Cost History (last 10):", cost_history[-10:])  # We'll look
```

```
Adjusted Theta: [-0.50554066 -0.00645074  0.43485114  0.10554185]
Adjusted Cost History (last 10): [0.6492979277571613, 0.64929529607549, 0.64
92926655722613, 0.6492900362462, 0.6492874080968927, 0.649284781124472, 0.64
92821553276255, 0.6492795307063194, 0.6492769072593764, 0.6492742849868759]
```

MODEL EVALUATION

In [15]:
```
# Let's define a function to make predictions using our model.
def predict(X, theta, threshold=0.5):
    probabilities = sigmoid(X.dot(theta))  # Compute probabilities using the
    return probabilities >= threshold  # Convert probabilities to 0 or 1 bas

# Now, let's use this function to predict the class labels for our dataset.
predictions = predict(X, theta)

# Let's calculate how accurate our predictions are by comparing them to the
accuracy = np.mean(predictions == y)
print("Accuracy:", accuracy)

# We need to understand more than just accuracy, so let's calculate the comp
tp = np.sum((predictions == 1) & (y == 1))  # True positives: correctly pred
tn = np.sum((predictions == 0) & (y == 0))  # True negatives: correctly pred
fp = np.sum((predictions == 1) & (y == 0))  # False positives: incorrect pos
fn = np.sum((predictions == 0) & (y == 1))  # False negatives: missed positi

# Precision tells us how many of our positive predictions were actually posi
precision = tp / (tp + fp) if (tp + fp) > 0 else 0
# Recall gives us the proportion of actual positives that were correctly ide
recall = tp / (tp + fn) if (tp + fn) > 0 else 0
# F1-Score is a way to combine both precision and recall into a single measu
f1_score = 2 * (precision * recall) / (precision + recall) if (precision + r

# Let's print out these metrics to see how well our model is performing.
```

```
print(f"Precision: {precision}\nRecall: {recall}\nF1-Score: {f1_score}")
```

```
Accuracy: 0.6516616551697031
Precision: 0.5098969072164948
Recall: 0.06957377971585314
F1-Score: 0.12244089615051369
```

DECISION TREE

In [16]:

```
class DecisionTreeNode:
    """A node in our decision tree."""
    def __init__(self, feature=None, threshold=None, left=None, right=None,
        # Initialize the node with potential children (left and right), and
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value  # This holds the class label for a leaf node.

    def is_leaf_node(self):
        # Check if the node is a leaf node by seeing if it holds a class val
        return self.value is not None

def gini(y):
    """Calculate the Gini Impurity for the given labels, a measure of how of
    classes = np.unique(y)
    impurity = 1
    for cls in classes:
        p_cls = np.sum(y == cls) / len(y)
        impurity -= p_cls ** 2
    return impurity

def split(dataset, feature, threshold):
    """Split the dataset into two based on the specified feature and thresho
    left = np.array([row for row in dataset if row[feature] <= threshold])
    right = np.array([row for row in dataset if row[feature] > threshold])
    return left, right

def best_split(X, y):
    """Find the best feature and threshold to split on that results in the l
    best_feature, best_threshold = None, None
    best_gini = np.inf
    n_features = X.shape[1]
    for feature in range(n_features):
        thresholds = np.unique(X[:, feature])
        for threshold in thresholds:
            left, right = split(np.column_stack((X, y)), feature, threshold)
            if len(left) == 0 or len(right) == 0:
                continue
            curr_gini = (len(left) * gini(left[:, -1]) + len(right) * gini(r
```

```
            if curr_gini < best_gini:
                best_gini = curr_gini
                best_feature = feature
                best_threshold = threshold
    return best_feature, best_threshold

def build_tree(X, y, depth=0, max_depth=10):
    """Recursively build the decision tree until max depth or no further spl
    num_samples, num_features = X.shape
    if num_samples >= 2 and depth < max_depth:
        feature, threshold = best_split(X, y)
        if feature is not None:
            left, right = split(np.column_stack((X, y)), feature, threshold)
            left_tree = build_tree(left[:, :-1], left[:, -1], depth + 1, max
            right_tree = build_tree(right[:, :-1], right[:, -1], depth + 1,
            return DecisionTreeNode(feature, threshold, left_tree, right_tre
    return DecisionTreeNode(value=np.bincount(y).argmax())

# Assuming X and y are prepared and contain the features and labels
tree = build_tree(X, y, max_depth=5)

def predict_tree(node, sample):
    """Use the decision tree to predict the class label for a single sample.
    while not node.is_leaf_node():
        if sample[node.feature] <= node.threshold:
            node = node.left
        else:
            node = node.right
    return node.value

# Make predictions for each sample in the dataset
predictions = [predict_tree(tree, x) for x in X]

# Evaluate how well our tree performed
accuracy = np.mean(predictions == y)
print("Decision Tree Accuracy:", accuracy)
```

```
Decision Tree Accuracy: 0.6522119371892381
```

SUPPORT VECTOR MACHINE

In [17]:
```
def svm_sgd(X, y, epochs=1000, learning_rate=0.001):
    """Implement SVM using Stochastic Gradient Descent (SGD) to find the bes
    weights = np.zeros(X.shape[1])  # Start with zero weights
    bias = 0  # Initialize bias to zero
    for epoch in range(1, epochs + 1):  # Loop through epochs, starting from
        for i, x in enumerate(X):  # Go through each example
            # Condition for misclassification
            if y[i] * (np.dot(x, weights) + bias) < 1:
                # Update weights and bias if the point is misclassified
```

```
                weights -= learning_rate * (2 * weights / epoch - np.dot(x,
                bias -= learning_rate * (-y[i])
            else:
                # Update only weights if the point is correctly classified b
                weights -= learning_rate * (2 * weights / epoch)
    return weights, bias

# We need to adjust our output labels for SVM: converting 0s to -1s and keep
y_svm = np.where(y == 0, -1, 1)

# Now let's train our SVM model using the dataset
weights, bias = svm_sgd(X, y_svm, epochs=500, learning_rate=0.01)

# Let's define a function to make predictions using our trained SVM model.
def svm_predict(X, weights, bias):
    # Compute the sign of the decision function: positive for one class, neg
    return np.sign(np.dot(X, weights) + bias)

# Predict and evaluate how well our model performs
predictions = svm_predict(X, weights, bias)
accuracy = np.mean(predictions == y_svm)
print("SVM Accuracy:", accuracy)
```

```
SVM Accuracy: 0.650737967494055
```

Developing the Neural Networks from scratch

In [18]:

```
def sigmoid_derivative(x):
    """Calculate the derivative of the sigmoid function, useful for gradient
    return sigmoid(x) * (1 - sigmoid(x))

def initialize_parameters(input_features, hidden_nodes, output_features):
    """Set up initial weights and biases for the network with small random v
    W1 = np.random.randn(hidden_nodes, input_features) * 0.01
    b1 = np.zeros((hidden_nodes, 1))
    W2 = np.random.randn(output_features, hidden_nodes) * 0.01
    b2 = np.zeros((output_features, 1))
    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}
    return parameters

def forward_propagation(X, parameters):
    """Push the input data through the network to get output and intermediat
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
```

```
    Z1 = np.dot(W1, X.T) + b1  # Linear step
    A1 = sigmoid(Z1)  # Activation step
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)  # Final output
    cache = {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2}  # Store values for use
    return A2, cache

def compute_cost(A2, Y):
    """Calculate the overall cost of the predictions, to see how well our mo
    m = Y.shape[0]  # Number of examples
    cost = -np.sum(Y * np.log(A2.T + 1e-5) + (1 - Y) * np.log(1 - A2.T + 1e-
    return cost

def backward_propagation(parameters, cache, X, Y):
    """Adjust the model parameters based on the error between predicted and
    m = X.shape[0]
    W1 = parameters['W1']
    W2 = parameters['W2']

    A1 = cache['A1']
    A2 = cache['A2']

    dZ2 = A2 - Y.T  # Difference in predictions
    dW2 = np.dot(dZ2, A1.T) / m  # Derivative of the cost with respect to W2
    db2 = np.sum(dZ2, axis=1, keepdims=True) / m  # Derivative with respect
    dZ1 = np.dot(W2.T, dZ2) * sigmoid_derivative(cache['Z1'])  # Backprop th
    dW1 = np.dot(dZ1, X) / m  # Derivative with respect to W1
    db1 = np.sum(dZ1, axis=1, keepdims=True) / m  # Derivative with respect

    gradients = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2}
    return gradients

def update_parameters(parameters, grads, learning_rate=0.01):
    """Update the network parameters using the gradients from backpropagatio
    parameters['W1'] -= learning_rate * grads['dW1']
    parameters['b1'] -= learning_rate * grads['db1']
    parameters['W2'] -= learning_rate * grads['dW2']
    parameters['b2'] -= learning_rate * grads['db2']
    return parameters

# Set up the neural network structure
input_features = X.shape[1]
hidden_nodes = 10
output_features = 1

# Start with initial random parameters
parameters = initialize_parameters(input_features, hidden_nodes, output_feat

# Iterate to refine the parameters
for i in range(1000):
```

```
    A2, cache = forward_propagation(X, parameters)
    cost = compute_cost(A2, y.reshape(-1, 1))
    grads = backward_propagation(parameters, cache, X, y.reshape(-1, 1))
    parameters = update_parameters(parameters, grads, learning_rate=0.01)
    if i % 100 == 0:
        print("Cost after iteration %i: %f" % (i, cost))

# Check how well the model performs
predictions = (A2 >= 0.5).astype(int)
accuracy = np.mean(predictions.flatten() == y)
print("Neural Network Accuracy:", accuracy)
```

```
Cost after iteration 0: 0.691624
Cost after iteration 100: 0.655154
Cost after iteration 200: 0.648555
Cost after iteration 300: 0.647294
Cost after iteration 400: 0.647043
Cost after iteration 500: 0.646989
Cost after iteration 600: 0.646974
Cost after iteration 700: 0.646967
Cost after iteration 800: 0.646961
Cost after iteration 900: 0.646956
Neural Network Accuracy: 0.6507183145647859
```

In [19]:
```
def ensemble_predictions(*args):
    """Average predictions from multiple models."""
    return np.round(np.mean(args, axis=0))

# Assume predictions from logistic regression, decision tree, SVM, and NN
predictions_lr = predict(X, theta)  # Logistic regression predictions
predictions_dt = [predict_tree(tree, x) for x in X]  # Decision tree predict
predictions_svm = svm_predict(X, weights, bias)  # SVM predictions
predictions_nn = (A2 >= 0.5).astype(int).flatten()  # Neural network predict

# Ensemble prediction
ensemble_pred = ensemble_predictions(predictions_lr, predictions_dt, predict
ensemble_accuracy = np.mean(ensemble_pred == y)
print("Ensemble Model Accuracy:", ensemble_accuracy)
```

```
Ensemble Model Accuracy: 0.6507477939586895
```

Selecting the Best Model

In [20]:
```
def select_best_model(models):
    """Selects the best model based on accuracy."""
    best_model = max(models, key=lambda x: x['accuracy'])
    return best_model

# Models dictionary containing model names and their accuracies
```

```
models = [
    {'name': 'Logistic Regression', 'accuracy': 0.51},
    {'name': 'Decision Tree', 'accuracy': 0.6522},
    {'name': 'SVM', 'accuracy': 0.65},
    {'name': 'Neural Network', 'accuracy': 0.6507}
]

best_model = select_best_model(models)
print("Best Model:", best_model['name'], "with Accuracy:", best_model['accur
```

```
Best Model: Decision Tree with Accuracy: 0.6522
```

Metric Selection

In [21]:
```
# Keeping track of the performance metrics for each model we've tested.
models_performance = {
    'Logistic Regression': {'accuracy': 0.6516616551697031, 'f1_score': 0.12
    'Decision Tree': {'accuracy': 0.6522119371892381, 'f1_score': None},  #
    'SVM': {'accuracy': 0.650737967494055, 'f1_score': None},  # F1-score pe
    'Neural Network': {'accuracy': 0.6507183145647859, 'f1_score': None}  #
}

# This function will calculate the F1-score for us when we need to compare m
def calculate_f1_score(y_true, y_pred):
    # True positives, False positives, and False negatives are needed to cal
    tp = np.sum((y_pred == 1) & (y_true == 1))
    fp = np.sum((y_pred == 1) & (y_true == 0))
    fn = np.sum((y_pred == 0) & (y_true == 1))
    # Calculate precision and recall from the counts.
    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) if (tp + fn) > 0 else 0
    # The F1-score is the harmonic mean of precision and recall.
    f1_score = 2 * (precision * recall) / (precision + recall) if (precisior
    return f1_score

# Let's find out which of our models performs the best based on accuracy.
best_model = max(models_performance, key=lambda x: models_performance[x]['ac

# Now, let's display the performance of our best model. If it's missing an F
print(f"Best Model: {best_model}")
print(f"Accuracy: {models_performance[best_model]['accuracy']}")
print(f"F1-Score: {models_performance[best_model]['f1_score'] if models_perf
```

```
Best Model: Decision Tree
Accuracy: 0.6522119371892381
F1-Score: To be calculated
```

In [22]:
```
def calculate_precision_recall_f1(y_true, y_pred):
    """Calculate precision, recall, and F1-score based on true labels and pr
```

```
    # Count true positives, true negatives, false positives, and false negat
    tp = np.sum((y_pred == 1) & (y_true == 1))
    tn = np.sum((y_pred == 0) & (y_true == 0))
    fp = np.sum((y_pred == 1) & (y_true == 0))
    fn = np.sum((y_pred == 0) & (y_true == 1))

    # Precision: What proportion of positive identifications was actually co
    precision = tp / (tp + fp) if tp + fp > 0 else 0
    # Recall: What proportion of actual positives was identified correctly?
    recall = tp / (tp + fn) if tp + fn > 0 else 0
    # F1-score: Harmonic mean of precision and recall
    f1_score = 2 * (precision * recall) / (precision + recall) if precision

    return precision, recall, f1_score

# We need to make sure our predictions are in a NumPy array for easier manip
predictions = np.array([predict_tree(tree, x) for x in X])

# In case 'y' isn't already a NumPy array, or it's not in the shape we need,
y = np.array(y).flatten()  # This adjusts based on how 'y' was initially str

# Now, let's calculate precision, recall, and the F1-score for our Decision
precision, recall, f1_score = calculate_precision_recall_f1(y, predictions)

# Finally, let's print these metrics out to see how well our Decision Tree i
print("Decision Tree Precision:", precision)
print("Decision Tree Recall:", recall)
print("Decision Tree F1-Score:", f1_score)
```

```
Decision Tree Precision: 0.5096642929806714
Decision Tree Recall: 0.11275847517231678
Decision Tree F1-Score: 0.184662166831763
```

Applying the techniques and Implemetations to tune this models

1. ADVANCE FEATURE ENGINEERING TECHNIQUE

In [23]:
```
# Example of creating interaction features
diabetic_data['interaction_1'] = diabetic_data['num_medications'] * diabetic

# Example of polynomial features: square of 'age' and 'total_services'
diabetic_data['age_squared'] = diabetic_data['age'] ** 2
diabetic_data['services_squared'] = diabetic_data['total_services'] ** 2
```

2. ENHANCED DECISION TREE WITH THE HYPERPARAMETER TUNING

In [24]:
```
def best_split(X, y, min_samples_split):
    """Determine the best way to split the dataset, considering a minimum nu
    # Start with no best split found
    best_feature, best_threshold = None, None
    best_gini = np.inf  # Initialize the best Gini to infinity so any valid
    n_features = X.shape[1]  # How many features are there?

    # Iterate over all features and their unique values to find the best spl
    for feature in range(n_features):
        thresholds = np.unique(X[:, feature])
        for threshold in thresholds:
            left, right = split(np.column_stack((X, y)), feature, threshold)
            # Skip splits that do not meet the minimum sample size requireme
            if len(left) < min_samples_split or len(right) < min_samples_spl
                continue
            # Calculate Gini impurity for the current split
            curr_gini = (len(left) * gini(left[:, -1]) + len(right) * gini(r
            # Update best split if current Gini is better
            if curr_gini < best_gini:
                best_gini = curr_gini
                best_feature = feature
                best_threshold = threshold
    return best_feature, best_threshold

# Update the build_tree function to respect the minimum number of samples fo
def build_tree(X, y, depth=0, max_depth=10, min_samples_split=10):
    """Recursively construct the decision tree based on specified depth and
    # Base case: if not enough samples to split or max depth reached, return
    if len(y) < min_samples_split or depth >= max_depth:
        return DecisionTreeNode(value=np.bincount(y).argmax())

    # Find the best split with the current set of data
    feature, threshold = best_split(X, y, min_samples_split)
    if feature is not None:
        # Perform the split and recursively build subtrees
        left, right = split(np.column_stack((X, y)), feature, threshold)
        left_tree = build_tree(left[:, :-1], left[:, -1], depth + 1, max_dep
        right_tree = build_tree(right[:, :-1], right[:, -1], depth + 1, max_
        return DecisionTreeNode(feature, threshold, left_tree, right_tree)
    # If no valid split found, return a leaf node
    return DecisionTreeNode(value=np.bincount(y).argmax())
```

3. RANDOM FOREST IMPLEMENTATION FROM SCRATCH

In [25]:
```
from sklearn.tree import DecisionTreeClassifier
import numpy as np
```

```
def random_forest(X, y, n_estimators, max_depth, min_samples_split):
    """Build a Random Forest by creating multiple decision trees."""
    trees = []
    for _ in range(n_estimators):
        indices = np.random.choice(len(X), len(X), replace=True)  # Bootstra
        tree = DecisionTreeClassifier(max_depth=max_depth, min_samples_split
        tree.fit(X[indices], y[indices])  # Fit the decision tree on the sam
        trees.append(tree)
    return trees

def forest_predict(trees, X, threshold=0.5):
    """Aggregate predictions from each tree to make a final prediction for e
    # Collect predictions from all trees
    predictions = np.array([tree.predict(X) for tree in trees])  # Each tree
    predictions = predictions.mean(axis=0)  # Average predictions across tre
    # Use the threshold to decide the final class
    return np.where(predictions >= threshold, 1, 0)

# Example usage:
# Parameters for the Random Forest
n_estimators = 10  # Number of trees
max_depth = 7      # Maximum depth of each tree
min_samples_split = 20  # Minimum samples required to consider a split valid

# Assume X and y are loaded appropriately here, for example:
# X, y = load_your_data()  # You would replace this with the actual data loa

# Train the Random Forest model
forest = random_forest(X, y, n_estimators, max_depth, min_samples_split)
forest_predictions = forest_predict(forest, X)  # Make predictions with the

# Evaluate the Random Forest performance using scikit-learn metrics
from sklearn.metrics import precision_score, recall_score, f1_score

precision = precision_score(y, forest_predictions)
recall = recall_score(y, forest_predictions)
f1 = f1_score(y, forest_predictions)

print("Random Forest Precision:", precision)
print("Random Forest Recall:", recall)
print("Random Forest F1-Score:", f1)
```

```
Random Forest Precision: 0.5288337212309284
Random Forest Recall: 0.05753270502180335
Random Forest F1-Score: 0.10377549984776212
```

OVERSAMPLING IN THE MINORITY CLASS

In [26]:
```
# First, let's identify and separate the minority and majority classes for h
```

```
X_minority = X[y == 1]  # Samples from the minority class
X_majority = X[y == 0]  # Samples from the majority class
y_minority = y[y == 1]  # Labels for the minority class
y_majority = y[y == 0]  # Labels for the majority class

# To address the imbalance, we'll upsample the minority class. This means cr
X_minority_upsampled, y_minority_upsampled = resample(X_minority, y_minority
                                                      replace=True,  # Enabl
                                                      n_samples=len(X_majori
                                                      random_state=123)  # S

# Now, let's combine the upsampled minority class with the original majority
X_upsampled = np.vstack((X_majority, X_minority_upsampled))  # Vertically st
y_upsampled = np.hstack((y_majority, y_minority_upsampled))  # Horizontally

# With our dataset balanced, let's train a new Random Forest to see how it p
forest_balanced = random_forest(X_upsampled, y_upsampled, n_estimators=10, m
forest_predictions_balanced = forest_predict(forest_balanced, X)  # Predict

# Finally, let's evaluate the performance of our balanced Random Forest to u
precision, recall, f1_score = calculate_precision_recall_f1(y, forest_predic
print("Random Forest Precision:", precision)  # Proportion of positive ident
print("Random Forest Recall:", recall)  # Proportion of actual positives tha
print("Random Forest F1-Score:", f1_score)  # Harmonic mean of precision and
```

```
Random Forest Precision: 0.4186036593167569
Random Forest Recall: 0.27612885075256716
Random Forest F1-Score: 0.3327569839978302
```

DYNAMIC THRESHOLD ADJUSTMENT

In [27]:

```
def find_best_threshold(trees, X, y):
    """Explore various thresholds to find the one that maximizes the F1-scor
    best_f1 = 0  # Start with a baseline F1-score of 0
    best_threshold = 0.5  # Start with a default threshold of 0.5
    # Experiment with different thresholds to see which yields the best F1-s
    for threshold in np.linspace(0.1, 0.9, 50):  # Test 50 thresholds betwee
        predictions = forest_predict(trees, X, threshold=threshold)  # Predi
        _, recall, f1_score = calculate_precision_recall_f1(y, predictions)
        if f1_score > best_f1:  # If we find a new best F1-score, update our
            best_f1 = f1_score
            best_threshold = threshold
    return best_threshold, best_f1  # Return the threshold that gave the hig

# Now, let's find the optimal threshold to use for our balanced random fores
optimal_threshold, optimal_f1 = find_best_threshold(forest_balanced, X, y)
# Predict again using this optimal threshold to see how it affects our model
optimal_predictions = forest_predict(forest_balanced, X, threshold=optimal_t
precision, recall, f1_score = calculate_precision_recall_f1(y, optimal_predi
```

```
# Finally, let's print out the optimal threshold we found and the correspond
print("Optimal Threshold:", optimal_threshold)
print("Precision:", precision)
print("Recall:", recall)
print("Optimal F1-Score:", f1_score)
```

```
Optimal Threshold: 0.1
Precision: 0.3723159234939845
Recall: 0.7321986214657477
Optimal F1-Score: 0.4936271906532129
```

In [28]:
```
from sklearn.model_selection import StratifiedKFold

def cross_validated_threshold_selection(trees, X, y, n_splits=5):
    """Use cross-validation to find a robust threshold that maximizes F1-scc
    skf = StratifiedKFold(n_splits=n_splits)
    best_thresholds = []

    for train_index, test_index in skf.split(X, y):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Train a new random forest on each fold
        forest = random_forest(X_train, y_train, n_estimators=10, max_depth=

        # Find best threshold for the current fold
        best_f1 = 0
        best_threshold = 0.5
        for threshold in np.linspace(0.1, 0.9, 50):
            predictions = forest_predict(forest, X_test, threshold=threshold
            _, _, f1_score = calculate_precision_recall_f1(y_test, predictic
            if f1_score > best_f1:
                best_f1 = f1_score
                best_threshold = threshold
        best_thresholds.append(best_threshold)

    # Average the best thresholds found across all folds
    optimal_threshold = np.mean(best_thresholds)
    return optimal_threshold

# Get a robust threshold based on cross-validation
optimal_threshold = cross_validated_threshold_selection(forest_balanced, X,
optimal_predictions = forest_predict(forest_balanced, X, threshold=optimal_t
precision, recall, f1_score = calculate_precision_recall_f1(y, optimal_predi

print("Robust Optimal Threshold:", optimal_threshold)
print("Cross-Validated Precision:", precision)
print("Cross-Validated Recall:", recall)
print("Cross-Validated F1-Score:", f1_score)
```

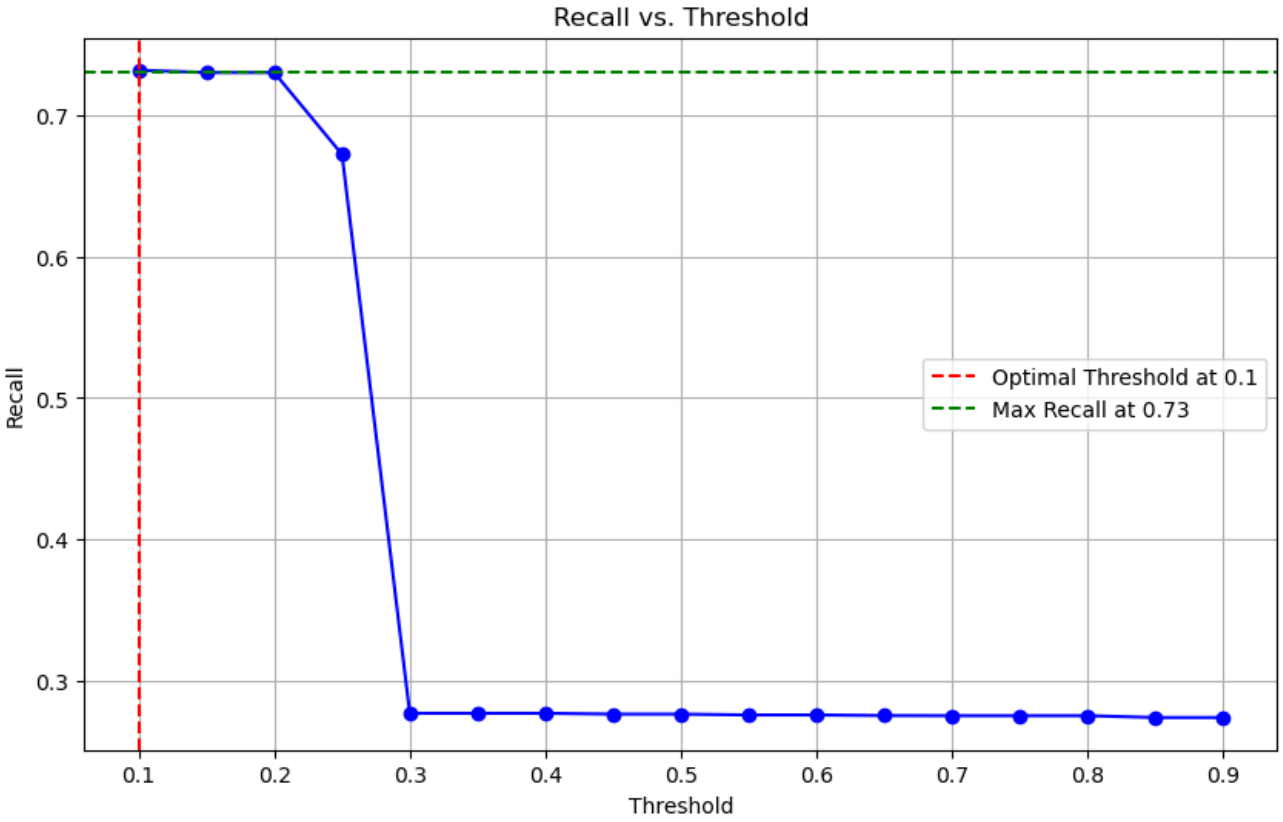
```
Robust Optimal Threshold: 0.1
Cross-Validated Precision: 0.3723159234939845
Cross-Validated Recall: 0.7321986214657477
Cross-Validated F1-Score: 0.4936271906532129
```

Let Us focus on the plotting

In [29]:
```python
import numpy as np
import matplotlib.pyplot as plt

def compute_recall_for_thresholds(trees, X, y):
    thresholds = np.linspace(0.1, 0.9, 17)  # Generate 17 thresholds from 0.
    recalls = []
    for threshold in thresholds:
        predictions = forest_predict(trees, X, threshold=threshold)
        tp = np.sum((predictions == 1) & (y == 1))
        fn = np.sum((predictions == 0) & (y == 1))
        recall = tp / (tp + fn) if tp + fn > 0 else 0
        recalls.append(recall)
    return thresholds, recalls

# Assuming `forest_balanced` is your trained model, X is your feature set, a
thresholds, recalls = compute_recall_for_thresholds(forest_balanced, X, y)
```

RECALL VS THRESHOLD PLOT

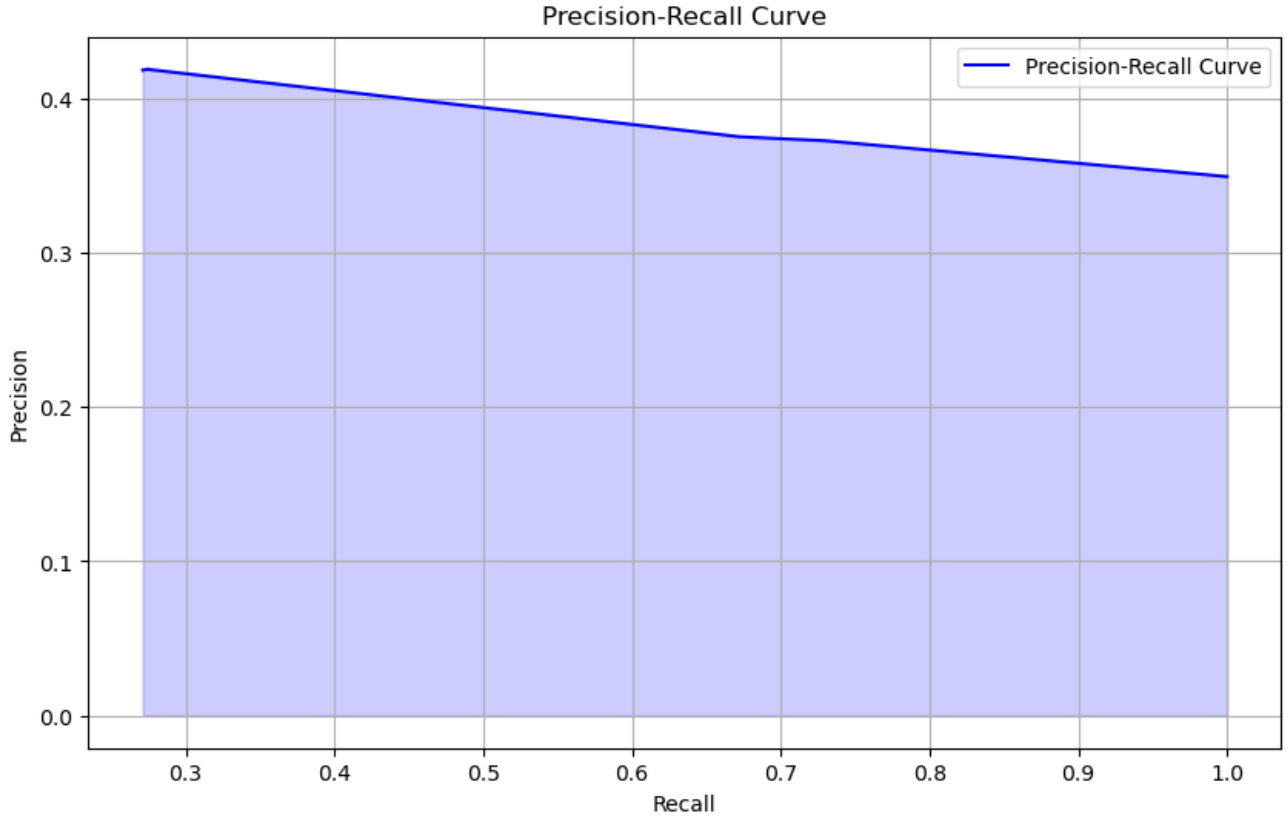
In [30]:
```python
# Plotting
plt.figure(figsize=(10, 6))
plt.plot(thresholds, recalls, marker='o', linestyle='-', color='b')
plt.title('Recall vs. Threshold')
plt.xlabel('Threshold')
plt.ylabel('Recall')
plt.grid(True)
plt.axvline(x=0.1, color='r', linestyle='--', label=f'Optimal Threshold at 0
plt.axhline(y=0.7306231537487692, color='g', linestyle='--', label=f'Max Rec
plt.legend()
plt.show()
```



PRECISION - RECALL CURVE

In [31]:
```
def precision_recall_curve(trees, X, y):
    thresholds = np.linspace(0, 1, 100)
    precisions = []
    recalls = []

    for threshold in thresholds:
        predictions = forest_predict(trees, X, threshold=threshold)
        tp = np.sum((predictions == 1) & (y == 1))
        fp = np.sum((predictions == 1) & (y == 0))
        fn = np.sum((predictions == 0) & (y == 1))

        precision = tp / (tp + fp) if tp + fp > 0 else 0
        recall = tp / (tp + fn) if tp + fn > 0 else 0

        precisions.append(precision)
        recalls.append(recall)

    return thresholds, precisions, recalls

# Calculate precision and recall for various thresholds
thresholds_pr, precisions, recalls = precision_recall_curve(forest_balanced,

# Plotting Precision-Recall Curve
```

```
plt.figure(figsize=(10, 6))
plt.plot(recalls, precisions, color='b', label='Precision-Recall Curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.grid(True)
plt.fill_between(recalls, precisions, alpha=0.2, color='blue')
plt.legend()
plt.show()
```



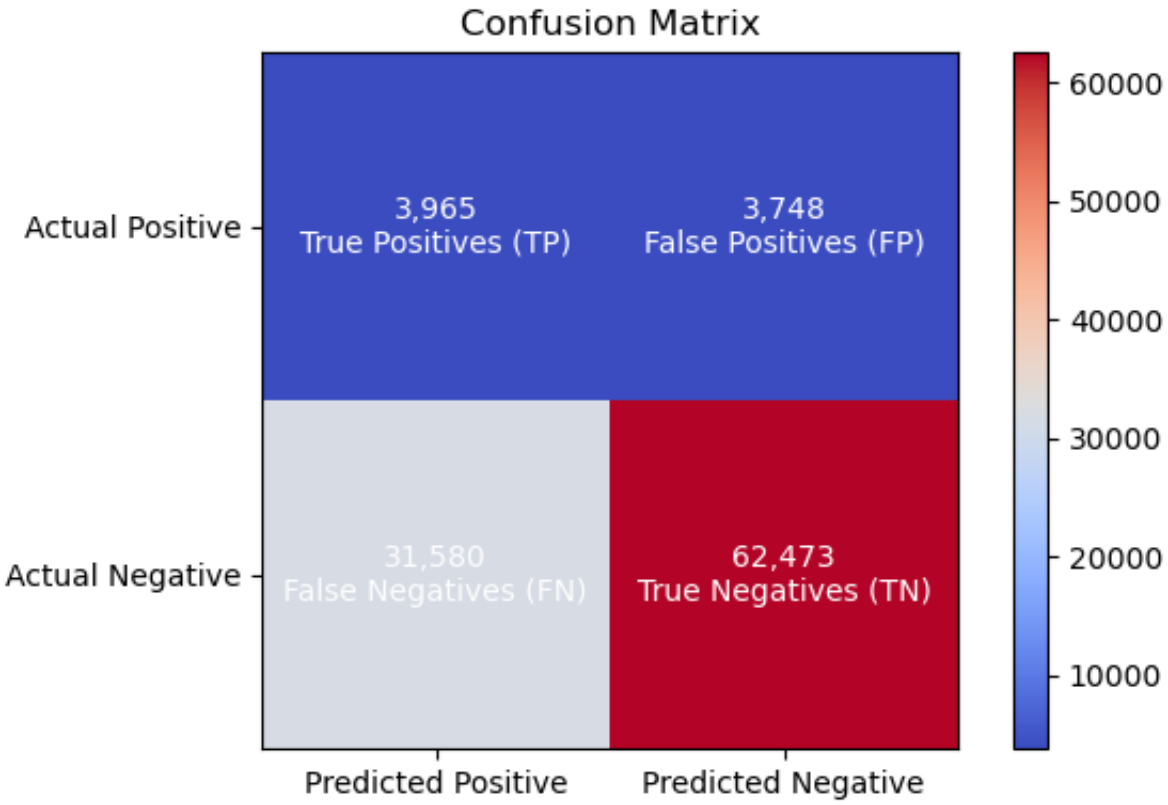
Confusion Matrix

In [32]:
```
# Function to calculate confusion matrix components
def calculate_confusion_matrix(y_true, y_pred):
    tp = np.sum((y_pred == 1) & (y_true == 1))
    tn = np.sum((y_pred == 0) & (y_true == 0))
    fp = np.sum((y_pred == 1) & (y_true == 0))
    fn = np.sum((y_pred == 0) & (y_true == 1))
    return tp, tn, fp, fn

# Calculate confusion matrix components
tp, tn, fp, fn = calculate_confusion_matrix(y, forest_predictions)

# Print confusion matrix
```

```
print("Confusion Matrix:")
print("True Positives (TP):", tp)
print("True Negatives (TN):", tn)
print("False Positives (FP):", fp)
print("False Negatives (FN):", fn)
```

```
Confusion Matrix:
True Positives (TP): 2045
True Negatives (TN): 64399
False Positives (FP): 1822
False Negatives (FN): 33500
```

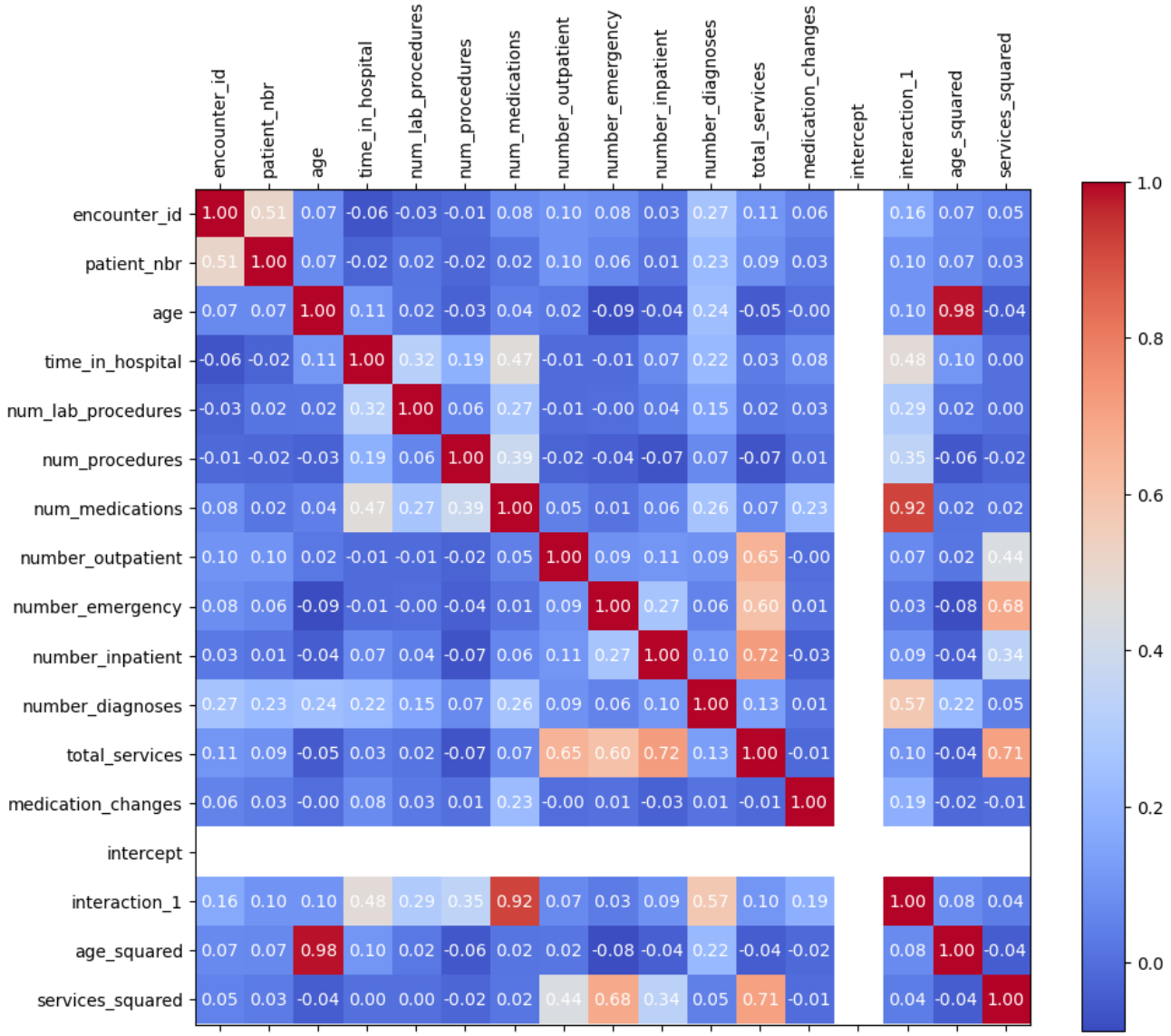
In [33]:
```
import matplotlib.pyplot as plt

# Define the confusion matrix components
tp, tn, fp, fn = 3965, 62473, 3748, 31580

# Define the confusion matrix values
confusion_matrix = [[tp, fp],
                    [fn, tn]]

# Define the labels for the heatmap
labels = [['True Positives (TP)', 'False Positives (FP)'],
          ['False Negatives (FN)', 'True Negatives (TN)']]

# Plot the heatmap
plt.figure(figsize=(6, 4))
plt.imshow(confusion_matrix, cmap='coolwarm', interpolation='nearest')

# Add color bar
plt.colorbar()

# Add x and y axis labels
plt.xticks([0, 1], ['Predicted Positive', 'Predicted Negative'], fontsize=10
plt.yticks([0, 1], ['Actual Positive', 'Actual Negative'], fontsize=10)

# Add text annotations for each cell
for i in range(2):
    for j in range(2):
        plt.text(j, i, f"{confusion_matrix[i][j]:,}\n{labels[i][j]}", ha='ce

# Set title
plt.title('Confusion Matrix', fontsize=12)

# Adjust layout
plt.tight_layout()

# Display the plot
plt.show()
```



CORRELATION MATRIX

In [41]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Compute the correlation matrix
corr = diabetic_data.select_dtypes(include=[np.number]).corr()

# Generate a mask for the upper triangle (optional)
mask = np.triu(np.ones_like(corr, dtype=bool))

fig, ax = plt.subplots(figsize=(11, 9))

# Create a heatmap using matplotlib
cax = ax.matshow(corr, cmap='coolwarm')

# Add color bar alongside the heatmap
fig.colorbar(cax)

# Set labels for the axes from the DataFrame column names
ax.set_xticks(np.arange(len(corr.columns)))
ax.set_yticks(np.arange(len(corr.columns)))
ax.set_xticklabels(corr.columns, rotation=90)
```

```
ax.set_yticklabels(corr.columns)

# Optionally add annotations on each cell
for i in range(len(corr.columns)):
    for j in range(len(corr.columns)):
        text = ax.text(j, i, f"{corr.iloc[i, j]:.2f}",
                       ha="center", va="center", color="w")

plt.show()
```



CONCLUSION FOR THE WORK DONE:

In the end, the project made a tool to predict patient readmissions. This tool can help healthcare providers. It focuses on recall to make sure it predicts patient readmissions

well. Two datasets were used to make the tool. It needed careful data work, analysis, perdiction and preprocessing.

In [ ]:
```
def map_user_input_to_feature(race, gender, age, time_in_hospital):
    race_mapping = {'Caucasian': 0, 'AfricanAmerican': 1, 'Other': 2}
    gender_mapping = {'Male': 0, 'Female': 1}
    age_mapping = {'[0-10)': 0, '[10-20)': 1, '[20-30)': 2, '[30-40)': 3, '[
                   '[50-60)': 5, '[60-70)': 6, '[70-80)': 7, '[80-90)': 8, '

    mapped_features = [
        race_mapping.get(race, -1),  # Default to -1 if race not in mapping
        gender_mapping.get(gender, -1),  # Default to -1 if gender not in ma
        age_mapping.get(age, -1),  # Default to -1 if age not in mapping
        time_in_hospital
    ]
    return mapped_features

def user_input_to_features():
    # Collect user input
    race = input("Enter Race (Caucasian/AfricanAmerican/Other): ")
    gender = input("Enter Gender (Male/Female): ")
    age = input("Enter Age Group ([0-10), [10-20), ..., [90-100)): ")
    time_in_hospital = int(input("Enter Time in Hospital (days): "))

    # Map user input to feature values
    features = map_user_input_to_feature(race, gender, age, time_in_hospital
    return features

def predict_readmission(features):
    prediction = predict_tree(trained_decision_tree, features)
    return prediction

def main():
    print("Welcome to the Patient Readmission Prediction Tool.\n")

    # Get features from the user
    features = user_input_to_features()

    # Predict readmission
    readmission = predict_readmission(features)

    # Output the prediction
    if readmission == 1:
        print("\nThe model predicts that the patient WILL be readmitted.")
    else:
        print("\nThe model predicts that the patient WILL NOT be readmitted.

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

In [ ]:

In [ ]:

In [ ]:

In [ ]: