

Name: Sudhir Singh
NetID: ss4552
Email: ss4552scarletmail.rutgers.edu

30-Day Readmission Prediction for Diabetic Patients

Introduction

Hospital readmissions are a costly and often preventable issue, with 30-day readmission rates serving as a key quality metric in healthcare. In patients with diabetes, they have been documented at levels of 14% to 22% which is significantly higher than the national average. The goal of this project is to predict if a diabetic patient will be readmitted within 30 days of discharge from hospital using the Diabetes 130-US hospitals (1999–2008) dataset. The dataset is available on Kaggle: <https://www.kaggle.com/datasets/brandao/diabetes>. It comes in a ZIP file containing `diabetic_data.csv` (original dataset) and a `description.pdf`, which is a reference to a research paper titled: "Impact of HbA1c Measurement on Hospital Readmission Rates: Analysis of 70,000 Clinical Database Patient Records". This research investigates whether HbA1c testing at the time of hospital admission affects 30-day readmission in diabetic patients. Using over 70,000 inpatient records from the Cerner Health Facts database, the study found that patients who had an HbA1c test during their stay were significantly less likely to be readmitted within 30 days. The diabetic data set includes 101,766 inpatient stays in 130 hospitals and a wide range of variables including the patient demographics (e.g., race, gender, age), hospital encounter details (e.g., admission type, discharge disposition, length of stay), clinical measures (e.g., diagnoses, A1C results), medication usage (e.g., insulin and diabetes drugs), prior healthcare utilization, and a binary indicator of 30-day readmission.

To fulfill this task, I will create an end-to-end machine learning pipeline so I created this project using Google Colab Notebook. The procedure includes data cleaning and preprocessing with Pandas, storage of data in a structured SQLite database, exploratory data analysis (EDA), and training two machine learning algorithms, logistic regression and decision tree, to make a prediction about 30-day readmissions. These models are selected for their balance of accuracy and interpretability, making them suitable for clinical applications where explainability is crucial.

This is the roadmap for the project:

1. Data Loading: Download the dataset in Google Colab and load it into Pandas DataFrames.
2. Data Preprocessing: Clean the data by handling missing values and removing irrelevant records; encode categorical variables for modeling.
3. Database Storage: Design a relational schema and store the cleaned data in SQLite, with separate tables for patients, encounters, and category mappings.
4. SQL Queries: Run and demonstrate SQL queries within Colab using the SQLite database (via `sqlite3` or `SQLAlchemy`).
5. Exploratory Data Analysis (EDA): Use Pandas and Seaborn/Matplotlib to visualize readmission trends by age group, number of medications, and insulin usage.

6. Model Training: Split the data into training and test sets, then train a logistic regression and a decision tree classifier using scikit-learn.
7. Model Evaluation: Evaluate both models using accuracy, precision, recall, F1-score, and a confusion matrix.

Here's the github repo link for the project: <https://github.com/SudhirSingh407337/Projects.git>

Here's the video link for the project:

https://drive.google.com/file/d/1YTUjCEv0fL3MWo3TTn3jqvcCO1Q_cILL/view?usp=sharing

Data Loading In Colab:

```
import pandas as pd
import numpy as np

# Load dataset from local or Google Drive path
df = pd.read_csv("/content/diabetic_data.csv")
print("Dataset loaded with shape:", df.shape)
```

```
Dataset loaded with shape: (101766, 50)
```

In this code block, I simply load the `diabetic_data.csv` file into a Pandas DataFrame and print its shape. This gives us an overview of the raw, uncleaned data before I begin preprocessing.

Data Preprocessing:

```
df.replace("?", np.nan, inplace=True)

df.drop(columns=["weight", "payer_code", "medical_specialty"], inplace=True, errors='ignore')

expired_codes = [11, 19, 20, 21]
df = df[~df['discharge_disposition_id'].isin(expired_codes)].copy()

admission_type_map = {
    1: "Emergency", 2: "Urgent", 3: "Elective", 4: "Newborn",
    5: "Not Available", 6: "NULL", 7: "Trauma Center", 8: "Not Mapped"
}
discharge_disposition_map = {
    1: "Discharged to home", 2: "Short-term hospital", 3: "SNF", 4: "ICF", 5: "Another facility",
    6: "Home health", 7: "Left AMA", 8: "Home/org care", 9: "Readmitted",
    11: "Expired", 13: "Hospice / Home", 14: "Hospice / Facility", 20: "Expired"
}
admission_source_map = {
    1: "Physician Referral", 2: "Clinic Referral", 3: "HMO Referral",
    4: "Transfer from hospital", 5: "Transfer from SNF", 6: "Other facility",
    7: "Emergency Room", 8: "Court/Law", 9: "Not Available"
}

df['admission_type'] = df['admission_type_id'].replace(admission_type_map)
df['discharge_disposition'] = df['discharge_disposition_id'].replace(discharge_disposition_map)
df['admission_source'] = df['admission_source_id'].replace(admission_source_map)

df.drop(['admission_type_id', 'discharge_disposition_id', 'admission_source_id'], axis=1, inplace=True)

df['readmit_30'] = np.where(df['readmitted'] == "<30", 1, 0)

df.drop(columns=["encounter_id"], inplace=True, errors='ignore')
constant_cols = [col for col in df.columns if df[col].nunique() <= 1]
df.drop(columns=constant_cols, inplace=True)

df.dropna(subset=["race", "gender", "age"], inplace=True)

print("Cleaned dataset shape:", df.shape)
```

Cleaned dataset shape: (97875, 46)

Next, I clean and preprocess the dataset to prepare it for analysis and modeling. Data preprocessing steps include handling missing values, removing irrelevant entries, and encoding categorical variables for modeling.

1. Handling Missing Values:

First, missing values are addressed. In this dataset, the placeholder "?" is used to represent missing data. These are replaced with `np.nan` so that Pandas can correctly recognize and process them as missing. Several columns have a high percentage of missing entries like `weight`, `payer_code`, and `medical_specialty`. Due to the unreliability and sparsity of these features, they are dropped entirely from the dataset. This is a typical data cleaning technique where columns containing excessive missing data, typically around 30%, are deleted to avoid skewing analysis or undermining model performance.

2. Removing Irrelevant or Out-of-scope Records:

Next, I remove irrelevant or non-actionable records from the dataset. In particular, the analysis aims to predict 30-day hospital readmissions. Therefore, patients who died during their hospital stay are not relevant to this goal. Such encounters are identified using the `discharge_disposition_id` field, where codes like 11, 19, 20, and 21 indicate death or hospice. These records are filtered out to ensure the analysis focuses only on patients who were actually at risk of being readmitted after discharge.

3. Encoding the Target Variable:

The target variable for prediction is whether a patient is readmitted within 30 days. The original column `readmitted` contains three values: "<30", ">30", and "NO". To reduce this to a binary classification problem, I defined a new column `readmit_30` where 1 denotes readmission within 30 days, and 0 otherwise (both ">30" and "NO" cases). This binary target allows normal classification models to be applied for prediction.

4. Additional Cleaning:

Additional cleaning steps are applied to remove unnecessary or uninformative data. The column `encounter_id`, which uniquely identifies each hospital stay, is dropped because it does not contribute to the prediction task. Similarly, columns that show no variance such as `examide` and `citoglipton`, which have only one unique value ("No")—are also removed, since they do not help distinguish between readmitted and non-readmitted patients. Finally, records that are missing essential demographic information such as `race`, `gender`, or `age` are excluded to ensure complete and consistent input data for modeling.

5. Mapping Encoded Variables to Human-Readable Labels

The dataset includes several integer-coded categorical fields that need to be translated into descriptive text to make the data more interpretable. Using Python dictionaries, I turned the `admission_type_id`, `discharge_disposition_id`, and `admission_source_id` to human-readable categories such as "Emergency", "Discharged to home", or "Physician Referral". After creating

new columns with these descriptive labels, the original ID columns are dropped, as they no longer add value and may confuse interpretation.

Overall, the updated code automates the transformation of categorical codes into descriptive categories and more thoroughly cleans the data. It concludes by printing the shape of the cleaned dataset to verify the reductions in rows and columns (ensuring no unintended data loss).

Clean-data set:

```
[20] print("Columns in dataset after cleaning:", df.columns.tolist())
      print("Sample of unique values in key columns:")
      print("  - race:", df['race'].unique())
      print("  - gender:", df['gender'].unique())
      print("  - age:", sorted(df['age'].unique()))
      print("  - readmit_30 (target) distribution:\n", df['readmit_30'].value_counts(normalize=True))

Columns in dataset after cleaning: ['patient_nbr', 'race', 'gender', 'age', 'time_in_hospital', 'num_lab_procedures', '
Sample of unique values in key columns:
  - race: ['Caucasian' 'AfricanAmerican' 'Other' 'Asian' 'Hispanic']
  - gender: ['Female' 'Male' 'Unknown/Invalid']
  - age: ['[0-10)', '[10-20)', '[20-30)', '[30-40)', '[40-50)', '[50-60)', '[60-70)', '[70-80)', '[80-90)', '[90-100)']
  - readmit_30 (target) distribution:
      readmit_30
0      0.885885
1      0.114115
Name: proportion, dtype: float64
```

1. `print("Columns in dataset after cleaning:", df.columns.tolist())`

This displays the full list of columns remaining after cleaning. This confirms that unnecessary, constant, or encoded columns were successfully removed.

2. `print("Sample of unique values in key columns:")`

This prepares to give a quick overview of diversity in important fields.

3. `print(" - race:", df['race'].unique())`

This shows the unique racial categories (e.g., 'Caucasian', 'AfricanAmerican', etc.), which confirms that missing or invalid values were handled.

4. `print(" - gender:", df['gender'].unique())`

This lists all gender entries, useful to verify if odd values like 'Unknown/Invalid' remain

5. `print(" - age:", sorted(df['age'].unique()))`

This displays sorted age brackets to ensure they cover the expected range and are formatted consistently (e.g., '[0-10)', '[90-100)').

5. `print(" - readmit_30 (target) distribution:\n", df['readmit_30'].value_counts(normalize=True))`

This gives the proportion of patients readmitted within 30 days (label 1) vs. those not readmitted within 30 days (label 0). This is useful to observe class imbalance here, only 11.4% were readmitted, confirming a strong class imbalance that may affect model performance.

Relational Database Schema (SQLite):

As part of this project, I show how to store the cleaned diabetes dataset in a relational SQL database using SQLite. SQLite is a lightweight, file-based database engine that integrates seamlessly with Python via the sqlite3 library and works directly in Colab.

```
[24] import sqlite3
import pandas as pd

conn = sqlite3.connect("diabetes_readmissions.db")

patients_df = df[["patient_nbr", "race", "gender", "age"]].drop_duplicates(subset="patient_nbr").copy()
patients_df.rename(columns={"patient_nbr": "patient_id"}, inplace=True)
patients_df.to_sql("Patients", conn, if_exists="replace", index=False)
print("Loaded Patients table:", patients_df.shape)

encounters_df = df.copy()
encounters_df.drop(columns=["race", "gender", "age", "readmitted"], inplace=True, errors='ignore')
encounters_df.rename(columns={"patient_nbr": "patient_id"}, inplace=True)

for col in encounters_df.columns:
    if encounters_df[col].apply(lambda x: isinstance(x, pd.Interval)).any():
        encounters_df[col] = encounters_df[col].astype(str)

encounters_df.to_sql("Encounters", conn, if_exists="replace", index=False)
print("Loaded Encounters table:", encounters_df.shape)

cur = conn.cursor()
cur.execute("SELECT name FROM sqlite_master WHERE type='table';")
print("SQLite tables:", [x[0] for x in cur.fetchall()])
```

Loaded Patients table: (68613, 4)
Loaded Encounters table: (97875, 44)
SQLite tables: ['Patients', 'Encounters']

The Patients table is designed to contain a single, unique record for every patient to make every individual show up only once in the database. It contains four demographic columns that are key: patient_id (previously patient_nbr), race, gender, and age. These attributes are considered static for each patient and are separated into their own table to avoid redundancy, ensuring that repeated hospital encounters do not duplicate demographic data.

The Encounters table captures the details of each hospital visit or admission. It includes a patient_id column that serves as a foreign key linking back to the Patients table. In addition to this, it stores a variety of clinical and administrative features such as time_in_hospital, num_lab_procedures, diag_1, and num_medications, among others. Importantly, it also contains


the `readmit_30` column, which indicates whether the patient was readmitted within 30 days. Columns like race, gender, and age are intentionally excluded from this table to prevent redundancy, as they are already available in the Patients table.

In terms of execution results, the Patients table was successfully loaded with 68,613 unique patient records across 4 columns. The Encounters table contains 97,875 rows, each representing a hospital admission, across 41 columns. Both tables were stored in a SQLite database named `diabetes_readmissions.db`, making them readily accessible for querying and integration in later analysis stages.

Querying the Database with SQL:

With the data now stored in a structured SQL database, I leveraged SQL syntax within Colab to perform powerful and efficient queries. This is particularly useful for summarizing large datasets, performing group-level aggregations, or conducting joins across multiple tables. Below are several examples demonstrating the analytical capabilities enabled by the relational schema.

Query 1 examines how many hospital encounters resulted in a 30-day readmission versus those that did not. This serves as a quick check on the class balance of the prediction target (`readmit_30`). Out of 97,875 total encounters, only 11.4% resulted in a readmission within 30 days. This confirms that the dataset is imbalanced, the majority of patients are not readmitted within the 30-day window.

```
 # Query 1: Readmission counts
query = """
SELECT readmit_30, COUNT(*) as count
FROM Encounters
GROUP BY readmit_30;
"""

result = pd.read_sql_query(query, conn)
print("\nReadmission counts:\n", result)
```

```

Readmission counts:
      readmit_30  count
0                0  86706
1                1  11169

```

Query 2 explores the average number of medications and average length of hospital stay (time_in_hospital) for patients based on their readmission status. Patients who were readmitted within 30 days had, on average, one more medication and 0.4 more days in the hospital compared to those who were not readmitted. This indicates that higher treatment complexity and longer hospitalizations may be correlated with a greater risk of early readmission.

```

#Query 2: What is the average number of medications for patients who were readmitted within 30 days versus those who were not?
query = """
SELECT readmit_30,
       AVG(num_medications) as avg_medications,
       AVG(time_in_hospital) as avg_time_in_hosp
FROM Encounters
GROUP BY readmit_30;
"""
print("\nAverage meds/time by readmission:\n", pd.read_sql_query(query, conn))

```

```

Average meds/time by readmission:
      readmit_30  avg_medications  avg_time_in_hosp
0                0      15.865430         4.342987
1                1      16.927926         4.770973

```

Query 3 joins the Encounters and Patients tables to calculate the 30-day readmission rate by age group. For each age bracket (e.g., [50-60), [60-70), etc.), the query counts total encounters and those resulting in early readmission, then computes the proportion. The results show that younger patients (under 20) have the lowest readmission rates, while older adults (ages 70–90) exhibit readmission rates above 12%. This suggests that older diabetic patients may be more susceptible to post-discharge complications and could benefit from targeted follow-up care or support.


```
# Query 3: Readmission rate by age group using join
query = """
SELECT P.age,
       COUNT(E.readmit_30) as total_encounters,
       SUM(E.readmit_30) as readmit_30_count,
       (1.0 * SUM(E.readmit_30) / COUNT(E.readmit_30)) as readmit_rate
FROM Encounters E
JOIN Patients P ON E.patient_id = P.patient_id
GROUP BY P.age
ORDER BY P.age;
"""

age_readmit = pd.read_sql_query(query, conn)
print("\nReadmission rate by age:\n", age_readmit)
```

Readmission rate by age:

| | age | total_encounters | readmit_30_count | readmit_rate |
|---|----------|------------------|------------------|--------------|
| 0 | [0-10) | 161 | 3 | 0.018634 |
| 1 | [10-20) | 714 | 59 | 0.082633 |
| 2 | [20-30) | 1629 | 229 | 0.140577 |
| 3 | [30-40) | 3757 | 418 | 0.111259 |
| 4 | [40-50) | 9660 | 1067 | 0.110455 |
| 5 | [50-60) | 16876 | 1657 | 0.098187 |
| 6 | [60-70) | 21762 | 2476 | 0.113776 |
| 7 | [70-80) | 25027 | 3025 | 0.120869 |
| 8 | [80-90) | 15921 | 1973 | 0.123924 |
| 9 | [90-100) | 2368 | 262 | 0.110642 |

Exploratory Data Analysis (EDA):

Now, I performed exploratory data analysis (EDA) on the dataset to identify patterns associated with 30-day hospital readmissions. I used visualizations to explore the relationship between readmission rates and variables such as age group, number of medications, HbA1c levels, and insulin usage, utilizing Python libraries like Matplotlib and Seaborn.

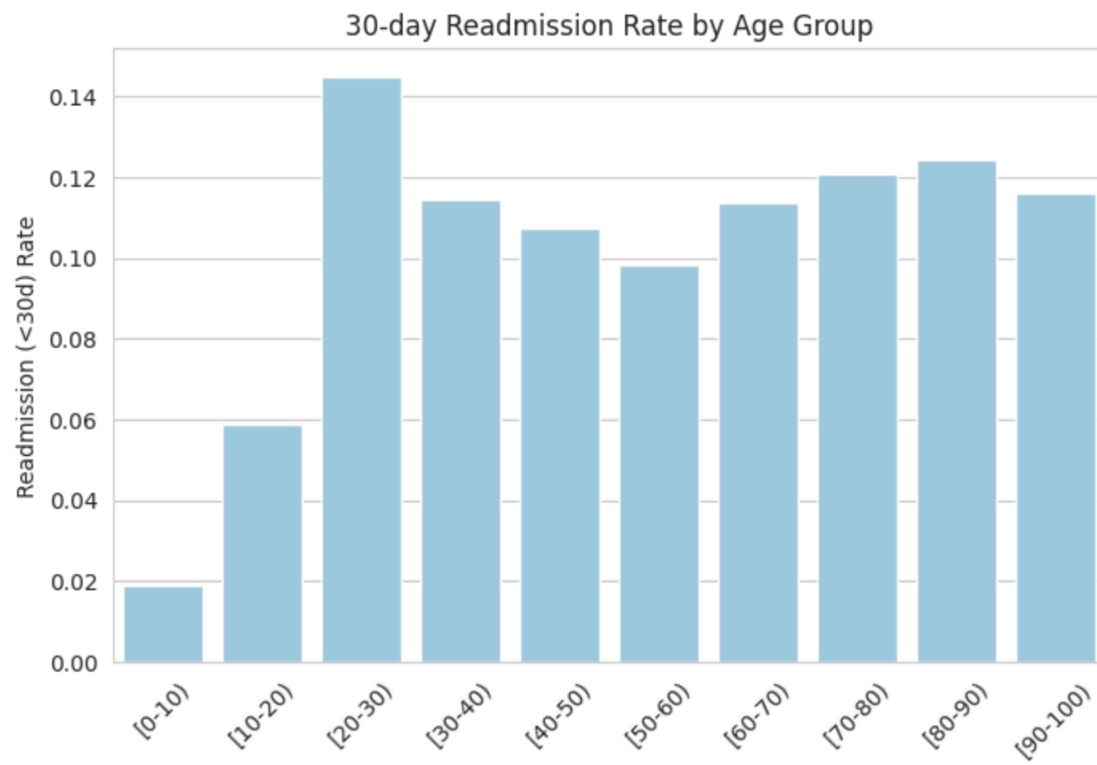
1. Readmission Rate by Age Group:

I begin by examining the readmission rate by age group. The bar chart displays the percentage of encounters that resulted in a 30-day readmission for each age bracket (e.g., [0–10), [10–20), ..., [90–100)). The [0–10) age group has the lowest rate at under 2%, followed by a moderate increase to around 6% for [10–20). The highest rate is observed in the [20–30) group, peaking at over 14%. Beyond that, rates stabilize between 9.8% and 12.5% across the adult and elderly groups. Notably, readmission rates trend slightly upward again from age 70 onward, likely due to comorbid conditions or age-related complications, which make older adults more prone to rehospitalization.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")

age_groups = df['age'].unique()
age_groups.sort()
readmit_rates = []
for age in age_groups:
    group = df[df['age'] == age]
    rate = group['readmit_30'].mean()
    readmit_rates.append(rate)

plt.figure(figsize=(8,5))
sns.barplot(x=age_groups, y=readmit_rates, color="skyblue")
plt.xticks(rotation=45)
plt.ylabel("Readmission (<30d) Rate")
plt.title("30-day Readmission Rate by Age Group")
plt.show()
```

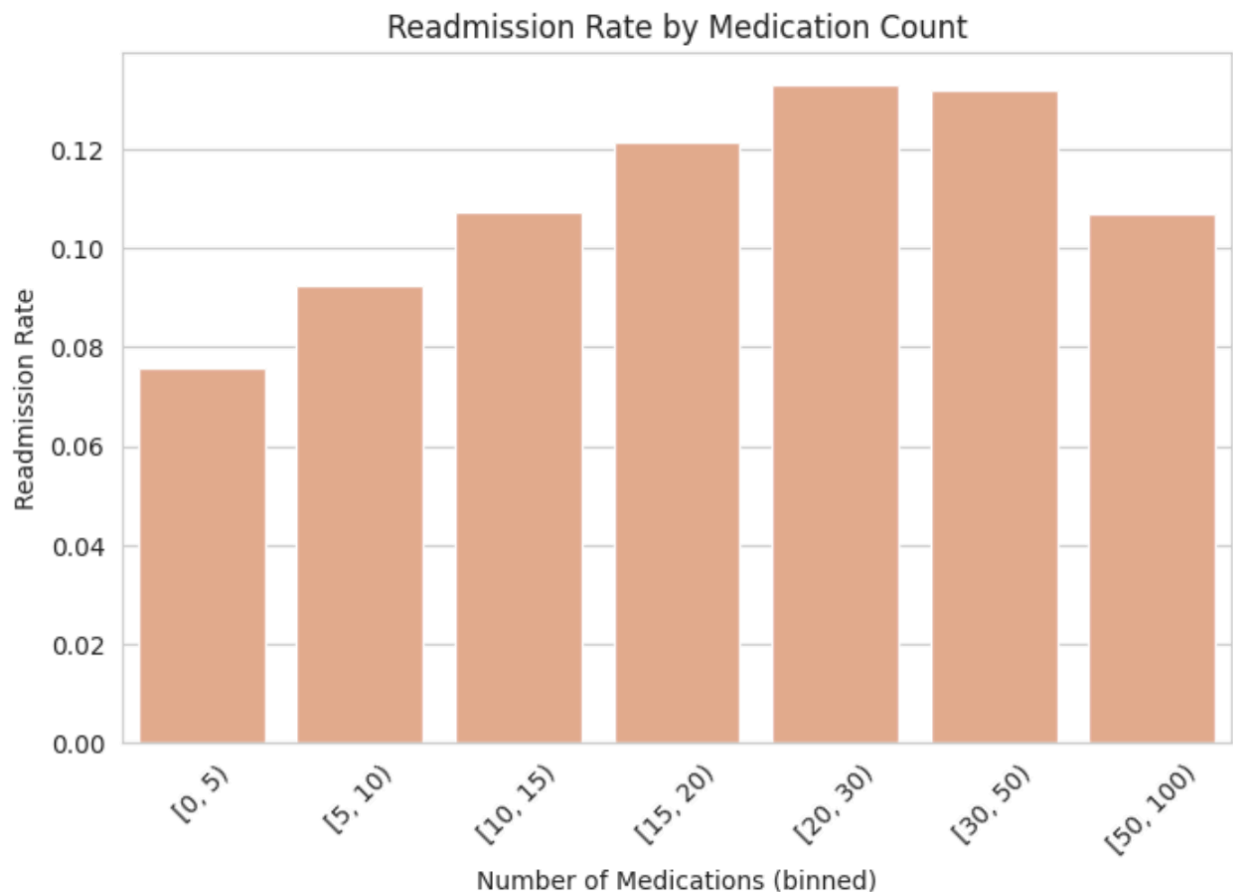


2. Readmission Rate by Number of Medications:

Next, I explored how the number of medications relates to readmission. Patients are grouped into bins based on the number of medications prescribed during their hospital stay (e.g., [0–5), [5–10), etc.) and the average readmission rate is computed for each group. The chart shows a clear upward trend—from about 7.5% in the [0–5) bin to a peak of roughly 13.5% in the [20–30) and [30–50) bins. There is a slight decrease in the [50–100) bin. This pattern suggests that patients receiving more medications, possibly due to more severe or complex health conditions, are more likely to be readmitted within 30 days.

```
# Readmission vs. Number of Medications
df['med_bins'] = pd.cut(df['num_medications'], bins=[0,5,10,15,20,30,50,100], right=False)
med_readmit = df.groupby('med_bins', observed=True)['readmit_30'].mean().reset_index()

plt.figure(figsize=(8,5))
sns.barplot(x='med_bins', y='readmit_30', data=med_readmit, color="lightsalmon")
plt.xticks(rotation=45)
plt.xlabel("Number of Medications (binned)")
plt.ylabel("Readmission Rate")
plt.title("Readmission Rate by Medication Count")
plt.show()
```



3. Distribution of HbA1c Results:

I also examined HbA1c levels, which reflect long-term blood glucose control by creating two charts, Bar plot: distribution of HbA1c results and Stacked bar plot of readmission proportion per HbA1c category. The two charts together provide insight into how HbA1c levels relates to 30-day hospital readmission rates among diabetic patients. The first chart shows the distribution of HbA1c results, where the majority of patients fell into the ">8" category (7,897 patients), indicating poor glucose control. This was followed by patients with normal HbA1c levels ("Norm," 4,856 patients) and those in the ">7" category (3,702 patients). These findings highlight the high prevalence of elevated HbA1c levels among hospitalized diabetic patients. The second chart illustrates the 30-day readmission rate by HbA1c category. Across all groups, the proportion of readmitted patients is relatively similar, with only a slight increase in the ">7" and ">8" groups compared to the "Norm" group. This suggests that while poor glycemic control may be associated with a modestly higher risk of readmission, the difference is not substantial. Overall, HbA1c alone does not appear to be a strong standalone predictor of readmission but can still offer valuable clinical context when used alongside other features in a predictive model.

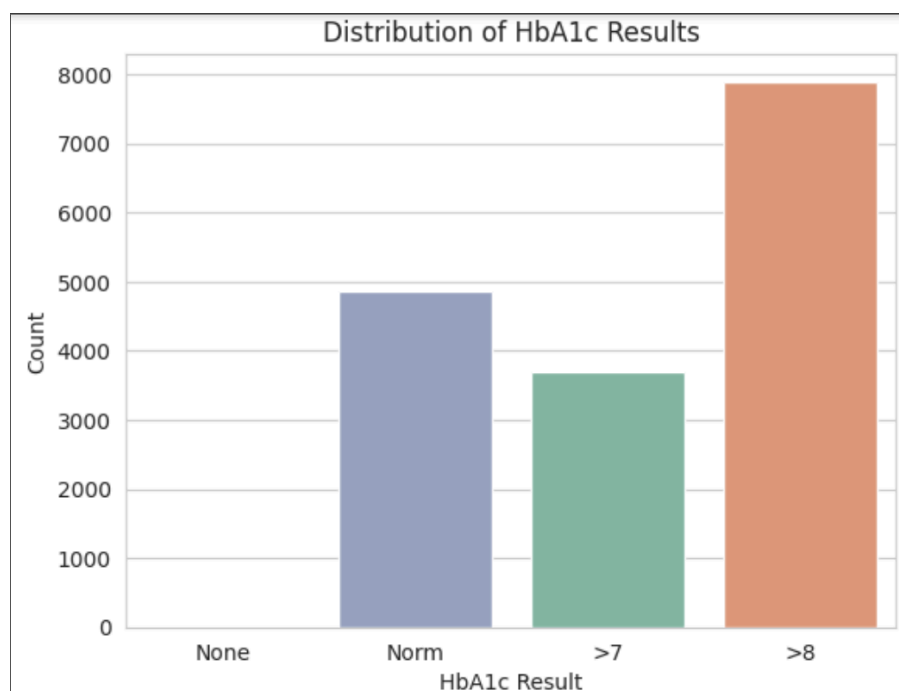
```
# Bar plot: distribution of HbA1c results
sns.countplot(data=df, x='A1Cresult', hue='A1Cresult', order=['None', 'Norm', '>7', '>8'], palette='Set2', legend=False)
plt.title("Distribution of HbA1c Results")
plt.xlabel("HbA1c Result")
plt.ylabel("Count")
plt.show()

print(df['A1Cresult'].value_counts())

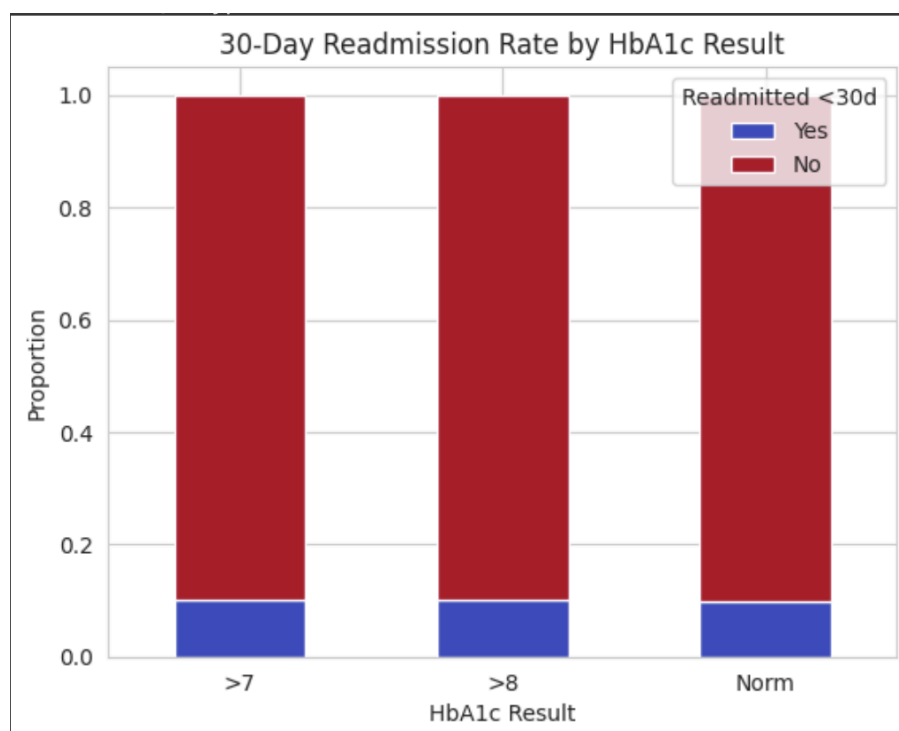
df['readmit_30_label'] = df['readmit_30'].map({0: "No", 1: "Yes"})

hb_readmit = df.groupby(['A1Cresult', 'readmit_30_label']).size().unstack().fillna(0)
hb_readmit_percent = hb_readmit.div(hb_readmit.sum(axis=1), axis=0)

# Stacked bar plot of readmission proportion per HbA1c category
hb_readmit_percent[['Yes', 'No']].plot(kind='bar', stacked=True, colormap='coolwarm')
plt.title("30-Day Readmission Rate by HbA1c Result")
plt.xlabel("HbA1c Result")
plt.ylabel("Proportion")
plt.legend(title="Readmitted <30d")
plt.xticks(rotation=0)
plt.show()
```



```
A1Cresult
>8      7897
Norm    4856
>7      3702
Name: count, dtype: int64
```



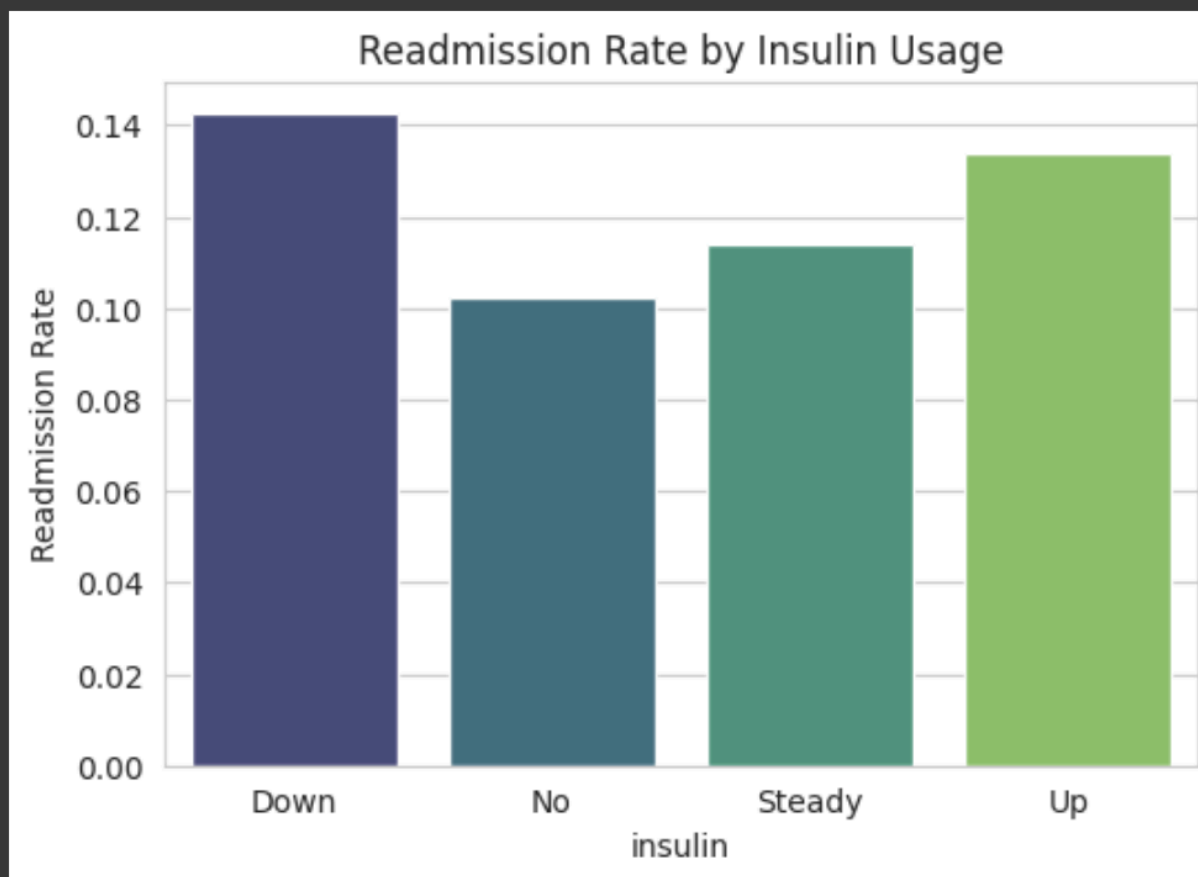
4. Readmission Rate by Insulin Usage:

Finally, I analyzed the effect of insulin therapy. The 'insulin' variable has four values: 'No', 'Steady', 'Up', and 'Down'. The bar chart shows that patients with decreased insulin doses ('Down') had the highest readmission rate at 14.1%, while those not on insulin had the lowest at 10.2%. Patients with unchanged ('Steady') and increased ('Up') doses had rates of 11.4% and 13.3%, respectively. When I simplified insulin usage into a binary form, used vs. not used, patients on insulin had a readmission rate of 12.4%, compared to 10.2% for those not on insulin. This supports the idea that insulin use, especially dose changes, may signal more severe diabetes, which could elevate the risk of hospital readmission.

```
# Readmission vs. Insulin Use
insulin_readmit = df.groupby('insulin')['readmit_30'].mean().reset_index()

plt.figure(figsize=(6,4))
sns.barplot(x='insulin', y='readmit_30', hue='insulin', data=insulin_readmit, palette="viridis", legend=False)
plt.ylabel("Readmission Rate")
plt.title("Readmission Rate by Insulin Usage")
plt.show()

df['insulin_used'] = (df['insulin'] != 'No').astype(int)
insulin_usage_rate = df.groupby('insulin_used')['readmit_30'].mean()
print("Readmission rate with insulin:", insulin_usage_rate[1])
print("Readmission rate without insulin:", insulin_usage_rate[0])
```



```
Readmission rate with insulin: 0.12443104342315005
Readmission rate without insulin: 0.10238833340610401
```

Modeling: Logistic Regression vs Decision Tree

With the exploratory analysis complete, I moved on to predictive modeling. The objective was to predict whether a patient would be readmitted within 30 days (`readmit_30 = 1`) or not (`readmit_30 = 0`) using hospital encounter data. I built and compared two classification models: Logistic Regression, a linear model that estimates the probability of readmission, and a Decision Tree, a non-linear model that learns decision rules from the data.

These models were chosen for their balance between predictive performance and interpretability. Logistic regression offers interpretable coefficients that quantify the impact of each feature, while decision trees produce rule-based structures that are easy to visualize and explain.

```
[34]
feature_cols = [
    'race', 'gender', 'age',
    'admission_type', 'discharge_disposition', 'admission_source',
    'time_in_hospital', 'num_lab_procedures', 'num_medications',
    'number_outpatient', 'number_emergency', 'number_inpatient',
    'A1Cresult', 'insulin', 'change', 'diabetesMed'
]

model_df = df[feature_cols + ['readmit_30']].copy()

cat_cols = ['race', 'gender', 'age', 'admission_type', 'discharge_disposition',
            'admission_source', 'A1Cresult', 'insulin', 'change', 'diabetesMed']
model_df = pd.get_dummies(model_df, columns=cat_cols, drop_first=True)

X = model_df.drop('readmit_30', axis=1)
y = model_df['readmit_30']
```

I selected a meaningful subset of features to ensure the model captures relevant clinical and administrative factors. These features include demographics such as race, gender, and age; admission characteristics like admission type, discharge disposition, and admission source; and a range of clinical indicators including time spent in the hospital, the number of lab procedures and medications, prior outpatient, emergency, and inpatient visits, as well as key diabetes-related markers like A1C result, insulin usage, medication changes, and whether the patient was prescribed diabetes medication. This combination offers a well-rounded view of each patient's condition and care context.

Categorical variables were encoded using one-hot encoding (`pd.get_dummies`), which converts each category into binary columns (e.g., `age_[20-30]`, `insulin_Up`). This approach is compatible with both logistic regression and decision tree models.

Train-Test Split

I split the data using an 80/20 ratio for training and testing. Specifically, the line `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)` divides the dataset into 80% for training (`X_train, y_train`) and 20% for testing (`X_test, y_test`). The `test_size=0.2` parameter ensures a consistent split ratio, while `random_state=42` sets a fixed seed to guarantee reproducibility. Given the class imbalance in the target variable (~11% of patients were readmitted within 30 days), I applied SMOTE (Synthetic Minority Over-sampling Technique) to the training data only, ensuring the model was trained on a balanced dataset without introducing information from the test set.

Train-Test Split

```
from sklearn.model_selection import train_test_split

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

Model Training

To address the class imbalance (only ~11% of patients are readmitted within 30 days), I applied SMOTE (Synthetic Minority Over-sampling Technique) to the training set only. This ensures the model sees a balanced distribution during learning without leaking synthetic data into the test set.

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from imblearn.over_sampling import SMOTE

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_scaled, y_train)

log_reg = LogisticRegression(max_iter=5000, solver='lbfgs')
log_reg.fit(X_train_resampled, y_train_resampled)

y_pred_lr = log_reg.predict(X_test_scaled)

tree_clf = DecisionTreeClassifier(max_depth=5, random_state=42)
tree_clf.fit(X_train_resampled, y_train_resampled)

y_pred_tree = tree_clf.predict(X_test_scaled)
```

I trained both models using scikit-learn:

Logistic Regression: Trained using the 'lbfgs' solver with max_iter=5000 to ensure convergence on the large resampled dataset. Since logistic regression is sensitive to feature magnitudes, I standardized the input features by using StandardScaler.

Decision Tree: Trained with max_depth=5 to reduce overfitting and enhance interpretability. Decision trees are not affected by feature scaling, so no standardization was applied to their input features.

This setup balances performance with explainability and avoids overfitting to the majority class.

Model Evaluation

I evaluate both models using key classification metrics to assess their performance. Accuracy measures the overall correctness of the model's predictions. Precision indicates the proportion of predicted positive cases that are actually correct, while recall captures the proportion of actual positive cases that the model successfully identifies. The F1-score, a harmonic mean of precision and recall, provides a balanced measure especially useful in the presence of class imbalance. Finally, the confusion matrix offers a detailed breakdown of true positives, true negatives, false positives, and false negatives, allowing for a deeper understanding of the types of errors each model makes.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report

# Metrics for Logistic Regression
acc_lr = accuracy_score(y_test, y_pred_lr)
prec_lr = precision_score(y_test, y_pred_lr)
recall_lr = recall_score(y_test, y_pred_lr)
f1_lr = f1_score(y_test, y_pred_lr)
cm_lr = confusion_matrix(y_test, y_pred_lr)

# Metrics for Decision Tree
acc_tree = accuracy_score(y_test, y_pred_tree)
prec_tree = precision_score(y_test, y_pred_tree)
recall_tree = recall_score(y_test, y_pred_tree)
f1_tree = f1_score(y_test, y_pred_tree)
cm_tree = confusion_matrix(y_test, y_pred_tree)

print("Logistic Regression - Accuracy: {:.3f}, Precision: {:.3f}, Recall: {:.3f}, F1: {:.3f}".format(acc_lr, prec_lr, recall_lr, f1_lr))
print("Decision Tree - Accuracy: {:.3f}, Precision: {:.3f}, Recall: {:.3f}, F1: {:.3f}".format(acc_tree, prec_tree, recall_tree, f1_tree))
print("\nConfusion Matrix (Logistic Regression):\n", cm_lr)
print("Confusion Matrix (Decision Tree):\n", cm_tree)
```

```
Logistic Regression - Accuracy: 0.678, Precision: 0.179, Recall: 0.522, F1: 0.266
Decision Tree - Accuracy: 0.854, Precision: 0.258, Recall: 0.161, F1: 0.198
```

```
Confusion Matrix (Logistic Regression):
```

```
[[12119  5263]
 [ 1049  1144]]
```

```
Confusion Matrix (Decision Tree):
```

```
[[16371  1011]
 [ 1841   352]]
```

Classification Report:

- Logistic Regression achieves higher recall (0.52), meaning it captures more true readmissions.
- Decision Tree achieves higher precision (0.26) and higher overall accuracy, but has lower recall (0.16), indicating more missed positives.

| Metric | Logistic Regression | Decision Tree |
|-----------|---------------------|---------------|
| Accuracy | 0.678 | 0.854 |
| Precision | 0.179 | 0.258 |
| Recall | 0.522 | 0.161 |
| F1 Score | 0.266 | 0.198 |

The logistic regression model, although less accurate overall (67.8% vs. 85.4%), had a much higher recall (52.2% vs. 16.1%), meaning it successfully identified over half of the patients who were readmitted within 30 days.

In contrast, the decision tree model had higher accuracy and precision, but only caught about 16% of actual readmissions, missing many at-risk patients.

Confusion Matrices

| | |
|---------------------------------|-----------------------------|
| True Negatives (No Readmission) | False Positives |
| False Negatives | True Positives (Readmitted) |

Logistic Regression:

| | |
|-------|------|
| 12119 | 5263 |
| 1049 | 1144 |

Decision Tree:

| | |
|-------|------|
| 16371 | 1011 |
| 1841 | 352 |

These matrices show that:

- Logistic Regression produced more false positives where it predicted readmission when there wasn't but produced fewer false negatives, meaning it missed fewer actual readmissions.
- The Decision Tree was more conservative in predicting readmission and achieved a cleaner separation of non-readmissions but at the cost of high false negatives.

```
print("\nClassification Report (Logistic Regression):\n", classification_report(y_test, y_pred_lr, target_names=["No Readmission", "Readmitted <30d"]))
print("Classification Report (Decision Tree):\n", classification_report(y_test, y_pred_tree, target_names=["No Readmission", "Readmitted <30d"]))
```

| Classification Report (Logistic Regression): | | | | |
|--|-----------|--------|----------|---------|
| | precision | recall | f1-score | support |
| No Readmission | 0.92 | 0.70 | 0.79 | 17382 |
| Readmitted <30d | 0.18 | 0.52 | 0.27 | 2193 |
| accuracy | | | 0.68 | 19575 |
| macro avg | 0.55 | 0.61 | 0.53 | 19575 |
| weighted avg | 0.84 | 0.68 | 0.73 | 19575 |
| Classification Report (Decision Tree): | | | | |
| | precision | recall | f1-score | support |
| No Readmission | 0.90 | 0.94 | 0.92 | 17382 |
| Readmitted <30d | 0.26 | 0.16 | 0.20 | 2193 |
| accuracy | | | 0.85 | 19575 |
| macro avg | 0.58 | 0.55 | 0.56 | 19575 |
| weighted avg | 0.83 | 0.85 | 0.84 | 19575 |

The logistic regression model achieved an overall accuracy of 68%, with strong performance in identifying patients who were not readmitted: it had 92% precision and 70% recall for the "No Readmission" class. For the "Readmitted <30d" class, the model showed limited precision (18%) but relatively high recall (52%), meaning it caught more true positives but at the cost of many false positives. Its F1-score of 0.27 for this class reflects a modest yet meaningful improvement indicating some balance between precision and recall.

The decision tree model achieved higher overall accuracy at 85% and was more effective at identifying non-readmitted patients, with 90% precision and 94% recall for that class. However, it struggled significantly more with the "Readmitted <30d" group, achieving only 26% precision and 16% recall, leading to a lower F1-score of 0.20.

In summary, logistic regression is more effective at capturing readmissions, while the decision tree excels in overall accuracy and performance for the No Readmission class but is weaker at identifying patients who are at risk of readmission within 30 days.

Changes from Project Proposal:

Compared to the original project proposal, several key optimizations and improvements were made along the way during final deployment. Most notably, the final model pipeline made use of SMOTE (Synthetic Minority Over-sampling Technique) in an effort to deal with the enormous class imbalance of readmission data, something not described in great detail in the proposal. The evaluation strategy was also extended beyond standard metrics to include confusion matrices, precision, recall, and F1-scores—providing a more fine-grained view of model performance, especially for the minority class. In practice, technical decisions other than the above were also documented, such as using the lbfgs solver and max_iter increment for logistic regression, scaling features using StandardScaler, and capping the decision tree with max_depth=5 to increase interpretability. The exploratory data analysis in the final report also went deeper than intended such as plotting distributions of HbA1c test results and how they relate to readmission. Furthermore, the final report simulated real-world deployment more thoroughly by highlighting key predictive features, generating risk flags for clinicians, and outlining how the model might integrate into discharge planning workflows details that were only briefly touched upon in the original proposal. These additions enhance the clarity, clinical relevance, and practicality of the final solution.

Conclusion:

This project presents a complete end-to-end data science pipeline, beginning with structured hospital encounter data and culminating in predictive modeling for 30-day patient readmissions. The workflow includes data cleaning, transformation, relational database integration, exploratory visualization, model training, and performance evaluation all within the context of clinical relevance. By focusing on interpretable models such as logistic regression and decision trees, I maintained transparency in feature influence and model decision-making. This is especially important in healthcare, where trust, accountability, and explainability are critical. The evaluation emphasized not just accuracy, but also precision and recall which are metrics that reflect the model's real-world usefulness in identifying high-risk patients.

Though this model is not yet deployed in a live clinical setting, I simulated practical application by identifying key predictive features to support clinical review, generating risk flags to assist care teams, and exploring how such a model could be integrated into discharge planning workflows. Identifying key features helps clinicians understand why a prediction was made, building trust and enabling informed decision-making. Generating risk flags translates model output into actionable alerts, helping care teams prioritize follow-up for high-risk patients. Exploring discharge integration ensures the model aligns with existing hospital workflows, making it both usable and valuable in real-world settings without causing disruption.

Overall, this project demonstrates how existing data science and machine learning techniques can be applied directly to solve healthcare problems. Its simplicity, clarity, and real-world grounding make it both an effective learning tool and a stepping stone toward clinical decision support systems. By structuring raw hospital data, engineering meaningful features, and evaluating interpretable models, this project bridges the gap between technical modeling and clinical utility. It highlights how predictive analytics can inform care delivery, such as identifying high-risk patients at discharge and enabling targeted follow-up. Additionally, the transparency of the models fosters trust and facilitates integration into clinical workflows, which is essential for adoption in real-world healthcare settings.