Malnad College of Engineering, Hassan

(An Autonomous Institution affiliated to VTU, Belgavi)

Machine learning Report On

"Machine Learning-Based Prediction of Patient Readmission Using the Diabetes 130-US Hospitals Dataset"

Submitted in partial fulfilment of

the requirements for the award of the degree of

Bachelor of Engineering in

Computer Science and Engineering

Submitted by :

KANNIKA H K	**4MC22CS073**
KARAN N	**4MC22CS074**
KARTHIK PAI	**4MC22CS075**
KESHAV YADAV L	**4MC22CS076**
KHALANDAR BIBI	**4MC22CS077**

Submitted to :

DR. KEERTHI KUMAR H.M

Department of Computer Science and Engineering

2024-2025

Real Word Problem

“Can we predict the hospital readmission of diabetic patients using demographic, medical, and hospital encounter attributes?”

Dataset Chosen

The Diabetes 130-US hospitals dataset from the UCI Machine Learning Repository contains over 100,000 records of diabetic patient encounters. It includes both numerical and categorical attributes such as age, race, diagnosis, medications, and lab procedures. The goal is to classify patient readmission status (Yes, No, <30 days), simulating real-world hospital decision-making in managing chronic diseases like diabetes.

Dataset Description

The Diabetes 130-US Hospitals dataset is a large, real-world medical dataset sourced from the UCI Machine Learning Repository. It contains over 100,000 records of hospital encounters for diabetic patients collected across 130 hospitals in the United States over a 10-year period. Each instance includes detailed demographic information, admission details, diagnoses, medications, and lab results, enabling the study of patient outcomes and healthcare quality.

The primary objective of this dataset is to predict hospital readmission status—whether a patient is readmitted within 30 days, after 30 days, or not readmitted at all—based on these features. This classification task supports the development of predictive models aimed at improving hospital resource management, identifying high-risk patients, and enhancing chronic disease care. The dataset's real-world scale and diversity make it an excellent resource for exploring advanced supervised learning methods in the healthcare domain.

Key Features

- **Race** – The race of the patient (e.g., Caucasian, AfricanAmerican, Asian).
- **Age** – The age group of the patient (e.g., [0-10), [70-80), [80-90)).
- **Admission_type_id** – The type of hospital admission (e.g., Emergency, Elective).
- **discharge_disposition_id** – Patient's condition or destination after discharge (e.g., Discharged to home, Expired).
- **Num_lab_procedures** – Number of lab tests performed during the encounter.
- **Num_medications** – Total number of unique medications administered.
- **Number_diagnoses** – Number of distinct diagnoses recorded during the encounter.
- **Gender** – Gender of the patient (e.g., Male, Female).
- **DiabetesMed** – Indicates if diabetes medication was prescribed (Yes/No).
- **Readmitted** – Target variable indicating if the patient was readmitted: <30, >30, or NO.

Algorithms Used

- K-Nearest Neighbours (KNN)
- Decision Tree Classifier
- Support Vector Machine (SVM)
- Random Forest Algorithm
- **Linear Regression** Algorithm

K – Nearest Neighbours Algorithm

Introduction

K-Nearest Neighbours (KNN) is a simple yet powerful **supervised-learning** algorithm that can be used for both **classification** and **regression**. It assigns a label to a new instance by **examining the *k* most similar records** (its "nearest neighbours") in the training data. Because KNN is **non-parametric**—it makes no assumptions about how the data are distributed—it adapts well to complex, real-world datasets such as electronic health-record data from hospitals.

How KNN Works?

- **Feature Scaling** : Because KNN relies on distance calculations (e.g., Euclidean or Manhattan distance), features must be scaled so that variables with large numeric ranges (e.g., number of lab procedures) do not dominate those with smaller ranges (e.g., age group codes).
- **Distance Calculation** : For a test instance, KNN computes the chosen distance metric between that instance and every point in the training set.
- **Neighbour Voting**
    - **Classification:** The majority class among the *k* closest neighbours is assigned to the new instance.
    - **Regression (optional):** The predicted value is the average (or weighted average) of the neighbours' target values.

Why Use KNN for Cardio Dataset?

- **Straightforward & Intuitive** : KNN predicts a patient's **readmission status** by looking at past encounters with similar demographic, diagnostic, and treatment profiles—an approach that is easy for clinicians and analysts to understand.

- **No Explicit Training Phase** : KNN stores the entire training set and performs all computation at prediction time. This makes it trivial to **incorporate new hospital encounters** without retraining a model.

- **Handles Mixed Feature Types After Encoding** : The dataset contains many **categorical attributes** (e.g., admission type, discharge disposition). Once one-hot encoded, KNN naturally works with the resulting numerical vectors.

- **Flexible Distance Metrics** : Healthcare data can be unevenly distributed; experimenting with Euclidean, Manhattan, or even Mahalanobis distance helps identify the metric that best captures patient similarity.

- **Strong Baseline for Comparison** : Although KNN may not always outperform sophisticated models (e.g., SVM, Random Forest), it is an excellent **baseline**; improvements offered by more complex algorithms can be measured against KNN's performance.

- **Local Decision-Making Insight** : Because predictions are based on actual neighbouring cases, KNN can highlight **individual past encounters** that drove a decision—useful for explaining results to clinicians or hospital administrators.

Dataset Description

1. **Goal** : The objective is to **classify food products** into two quality categories—**Good (1)** or **Bad (0)**—based on their **sensory** (taste, smell, texture) and **packaging** (type, label) attributes.

2. **Features**:

   - **Product**: Name/type of food item (e.g., Chips, Juice)

- **Taste**: Dominant taste attribute (e.g., Sweet, Salty)
- **Smell**: Odor intensity or freshness (e.g., Mild, Strong)
- **Texture**: Surface or consistency (e.g., Crunchy, Soft)
- **Packaging**: Type of packaging used (e.g., Plastic, Bottle)

3. **Target**:
   - **Label**: Quality indicator — Good = 1, Bad = 0
4. **Type**:
   - A **supervised classification problem** using **categorical features** that are encoded numerically.

DATA PREPROCESSING STEPS

- **Label Encoding** : All **categorical features and target labels** are converted into **numerical values** using encoding techniques (e.g., Label Encoding or One-Hot Encoding), ensuring compatibility with machine learning algorithms that rely on numerical inputs.
- **Feature Scaling (Optional)** : While not always required for categorical data, **scaling** (e.g., MinMaxScaler or StandardScaler) can improve performance, especially for algorithms like **KNN** that are sensitive to distance-based calculations.
- **Data Splitting** : The dataset is divided into **training and testing sets**—typically using an **80:20 split**—to evaluate how well the trained model generalizes to unseen data.

KNN Model Building

- **Data Preparation** : The custom food dataset containing product attributes such as **taste, smell, texture, packaging**, and the **product quality label** is first loaded. All **categorical features** and labels are transformed into **numerical values** using **Label Encoding**, enabling distance-based calculations required by KNN. Features and target labels are then separated for model input.

- **Data Splitting** : The dataset is divided into **training and testing sets** using an **80:20 ratio**. The training set stores examples for the KNN classifier, while the test set evaluates model performance on unseen instances.
- **Feature Scaling** : Since KNN is sensitive to the magnitude of feature values, all encoded features are scaled using **normalization techniques** such as **Min-Max Scaling** or **Standardization**. This ensures that no feature dominates others during distance computation.
- **Model Initialization** : A **K-Nearest Neighbours classifier** is initialized with a chosen value of **k** (e.g., k=3). Parameters like the **distance metric** (commonly Euclidean) and **weighting strategy** are configured to define how neighbours influence the prediction.
- **Model Training** : KNN is a **lazy learning algorithm**, meaning it does not explicitly "train" in the traditional sense. Instead, it **stores the training data** and uses it directly during the prediction phase.
- **Prediction** : For each test instance, the KNN algorithm computes distances to all training samples, finds the **k closest neighbours**, and predicts the class label using **majority voting** among these neighbours.
- **Evaluation** : The model's accuracy is evaluated by comparing predicted labels against actual test labels. Metrics like **Accuracy Score**, **Confusion Matrix**, and **Classification Report** help measure classification performance.
- **Result Visualization** : Visual tools such as **confusion matrix heatmaps**, **bar plots** for class distribution, or **scatter plots** with colored predictions are used to understand the model's behavior and separability of classes.
- **Sample Prediction Output** : A table displaying sample test instances, their **actual labels**, and **predicted outputs** is shown to manually verify prediction correctness and check for misclassifications.

Model Performance and Visualization

- **Accuracy Evaluation** : The **KNN model's accuracy** is measured on the test dataset. Accuracy represents the proportion of correctly predicted food product labels (either "Yes" or "No") out of all predictions made. It gives a high-level view of how effectively the model can classify products as **acceptable (Yes)** or **not acceptable (No)** based on sensory and packaging features.
- **Classification Report** : A **classification report** is generated to provide more detailed performance metrics:

  - **Precision**: Indicates how many of the predicted "Yes" labels are actually correct.
  - **Recall**: Shows how many of the actual "Yes" labels were successfully identified.
  - **F1-Score**: A balance between precision and recall that helps assess the trade-off between false positives and false negatives.

These metrics are calculated for both classes:

- **Class 0**: "No" – Not acceptable food products
- **Class 1**: "Yes" – Acceptable food products

This report helps determine if the model is **biased toward a particular class** or performs equally well across both.

- **Confusion Matrix** : A **confusion matrix** visually represents the model's prediction outcomes:

  - **True Positives (TP)**: Correctly predicted "Yes"
  - **True Negatives (TN)**: Correctly predicted "No"
  - **False Positives (FP)**: Predicted "Yes" but actually "No"
  - **False Negatives (FN)**: Predicted "No" but actually "Yes"

This matrix is valuable for diagnosing specific errors and evaluating **model reliability**.

- **Accuracy vs. K Plot** : To determine the best value for k (number of neighbours), a **line plot** is generated that shows accuracy across various k values (e.g., 1 to 30). This analysis is performed using:

- **Distance Metric**: Minkowski (with p = 2, equivalent to Euclidean)
- **Weighting Scheme**: 'distance', giving closer neighbours more influence

The plot helps select the **optimal k** that yields the highest accuracy without overfitting, improving performance on unseen data.

- **Sample Predictions Table** : A sample output table is presented showing:

  - **Test Samples**
  - **Actual Labels**
  - **Predicted Labels**

Conclusion and limitations

Conclusion:

The **K-Nearest Neighbours (KNN)** algorithm, when combined with essential preprocessing techniques such as **feature encoding**, **scaling**, and optional **dimensionality reduction (e.g., PCA)**, has shown to be effective in classifying food products based on **sensory** (taste, smell, texture) and **packaging** attributes. Utilizing **Grid Search CV** for hyperparameter tuning allowed the identification of the optimal configuration for:

- Number of neighbours (k)
- Distance metric (e.g., Euclidean)
- Weighting scheme (e.g., uniform or distance-based)

These optimizations enhanced the classifier's accuracy and generalization. Additionally, **visualization tools** such as the **confusion matrix** and **accuracy vs. K plots** offered meaningful insights into model behaviour and performance, validating its suitability for food quality prediction.

Limitations

1. **Computational Cost** : KNN is computationally expensive at prediction time, as it calculates distances from all training samples for each query. This makes it **inefficient for real-time** or **large-scale applications**.
2. **Curse of Dimensionality** : With high-dimensional feature spaces (e.g., after one-hot encoding categorical attributes), the **distance measures become less meaningful**, potentially degrading model performance.
3. **Class Imbalance Sensitivity** : KNN is sensitive to **imbalanced datasets**. If the majority of samples belong to one class (e.g., "Yes"), the model may become biased and underperform on minority classes (e.g., "No"), particularly affecting **recall and F1-score**.
4. **Lazy Learning** : As a **non-parametric lazy learner**, KNN does not build an internal model during training. It stores the entire dataset, leading to **slower predictions** and **higher memory usage**.
5. **Parameter Sensitivity** : The model's success is highly dependent on carefully chosen parameters:

    - Optimal **value of K**
    - Appropriate **distance metric**
    - Effective **weighting strategy**
    - Poor parameter choices can result in **underfitting** or **overfitting**.

6. **Sensitive to Noise and Outliers** Noisy or mislabelled data points (e.g., a sweet product incorrectly marked as "No") can significantly **distort predictions**, since KNN relies heavily on local data neighbourhoods.

Code:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
```

```
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap

# Load dataset
df = pd.read_csv('diabetes_subset_25k.csv')
df.columns = df.columns.str.strip()

# Sample smaller dataset for faster processing
df = df.sample(1000, random_state=42).reset_index(drop=True)

# Target variable
target_column = 'readmitted'
X = df.drop(target_column, axis=1)
y = df[target_column]

# Categorical and numerical columns
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()
numerical_cols = X.select_dtypes(include=[np.number]).columns.tolist()

# Preprocessing
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_cols),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_cols)
    ])

# Pipeline
knn_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', KNeighborsClassifier())
])

# Reduced hyperparameter grid for faster tuning
param_grid = {
    'classifier__n_neighbors': [3, 5, 7],
    'classifier__weights': ['uniform'],
    'classifier__metric': ['euclidean']
}

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# GridSearchCV with parallel jobs
```

```
grid_search = GridSearchCV(knn_pipeline, param_grid, cv=5, scoring='accuracy',
n_jobs=-1)

# Fit grid search
print("Starting Grid Search...")
grid_search.fit(X_train, y_train)
print("Grid Search complete!")

print("Best Parameters:", grid_search.best_params_)

# Best model
best_model = grid_search.best_estimator_
best_model.fit(X_train, y_train)

# Predictions
y_pred = best_model.predict(X_test)

# Evaluation
acc = accuracy_score(y_test, y_pred)
print(f"\nKNN Accuracy: {acc*100:.2f}%")
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# ----------------------------
# 📊 VISUALIZATIONS
# ----------------------------

# 1. Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Oranges')
plt.title("KNN Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

# 2. Prediction Count (Bar Plot)
plt.figure(figsize=(6,4))
sns.countplot(x=y_pred, palette='Set2')
plt.title('Predicted Class Distribution')
plt.xlabel('Predicted Class')
plt.ylabel('Count')
plt.show()

# 3. Actual Class Proportions (Pie Chart)
labels = y.unique()
sizes = [sum(y_test == label) for label in labels]

plt.figure(figsize=(5,5))
```

```
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140,
colors=sns.color_palette('Set2'))
plt.title("Actual Class Distribution")
plt.axis('equal')
plt.show()

# 4. Feature Importance Proxy (Variance)
ohe = best_model.named_steps['preprocessor'].named_transformers_['cat']
encoded_cols = ohe.get_feature_names_out(categorical_cols)
all_feature_names = numerical_cols + list(encoded_cols)

X_train_transformed =
best_model.named_steps['preprocessor'].transform(X_train)
X_dense = X_train_transformed.toarray() if hasattr(X_train_transformed,
'toarray') else X_train_transformed

variances = np.var(X_dense, axis=0)
top_idx = np.argsort(variances)[-10:]
top_features = [all_feature_names[i] for i in top_idx]
top_values = variances[top_idx]

plt.figure(figsize=(10,5))
sns.barplot(x=top_values, y=top_features, palette='coolwarm')
plt.title("Top 10 Influential Features (by Variance)")
plt.xlabel("Variance")
plt.ylabel("Features")
plt.show()

# 5. Decision Boundary (only 2 numerical features)
features_2d = ['time_in_hospital', 'num_medications']
if all(f in numerical_cols for f in features_2d):
    X_vis = df[features_2d]
    y_vis = df[target_column]
    y_vis_encoded = pd.factorize(y_vis)[0]

    scaler_vis = StandardScaler()
    X_vis_scaled = scaler_vis.fit_transform(X_vis)

    knn_vis =
KNeighborsClassifier(n_neighbors=grid_search.best_params_['classifier__n_neigh
bors'])
    knn_vis.fit(X_vis_scaled, y_vis_encoded)

    x_min, x_max = X_vis_scaled[:, 0].min() - 1, X_vis_scaled[:, 0].max() + 1
    y_min, y_max = X_vis_scaled[:, 1].min() - 1, X_vis_scaled[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.2),
                         np.arange(y_min, y_max, 0.2))
```

```
    Z = knn_vis.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(8,6))
    cmap = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
    plt.contourf(xx, yy, Z, cmap=cmap, alpha=0.5)
    plt.scatter(X_vis_scaled[:, 0], X_vis_scaled[:, 1], c=y_vis_encoded,
edgecolors='k', cmap=cmap)
    plt.title("KNN Decision Boundary (2 Features)")
    plt.xlabel(features_2d[0])
    plt.ylabel(features_2d[1])
    plt.show()
else:
    print("Features for decision boundary not found in numerical columns.")
```

OUTPUT:

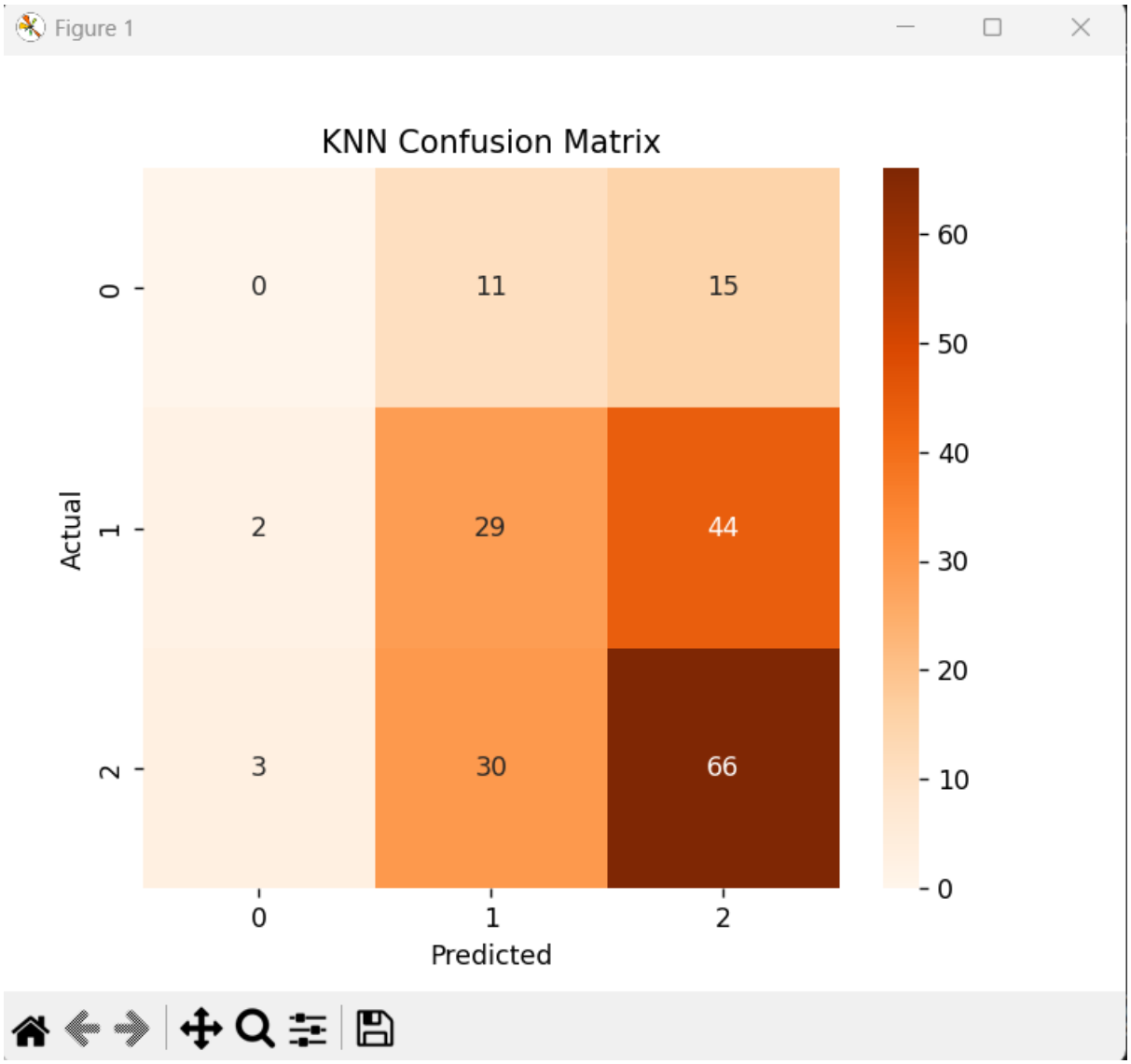


Figure 1
KNN Confusion Matrix
Actual
Predicted
0
1
2
0
11
15
2
29
44
3
30
66
60
50
40
30
20
10
0

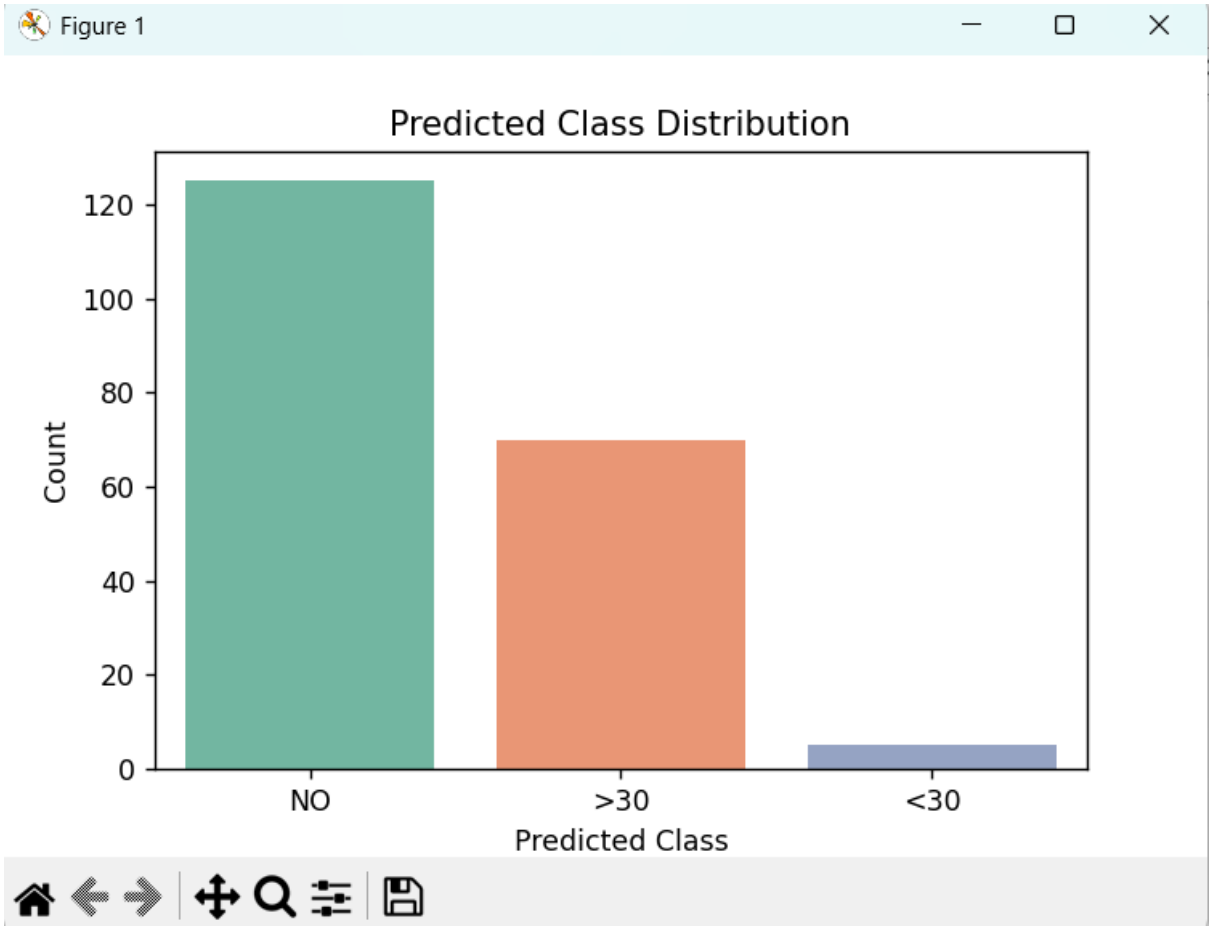
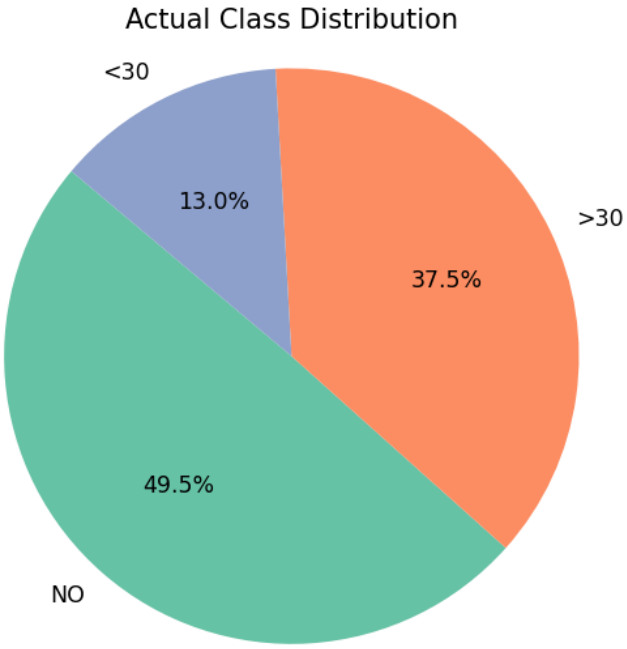
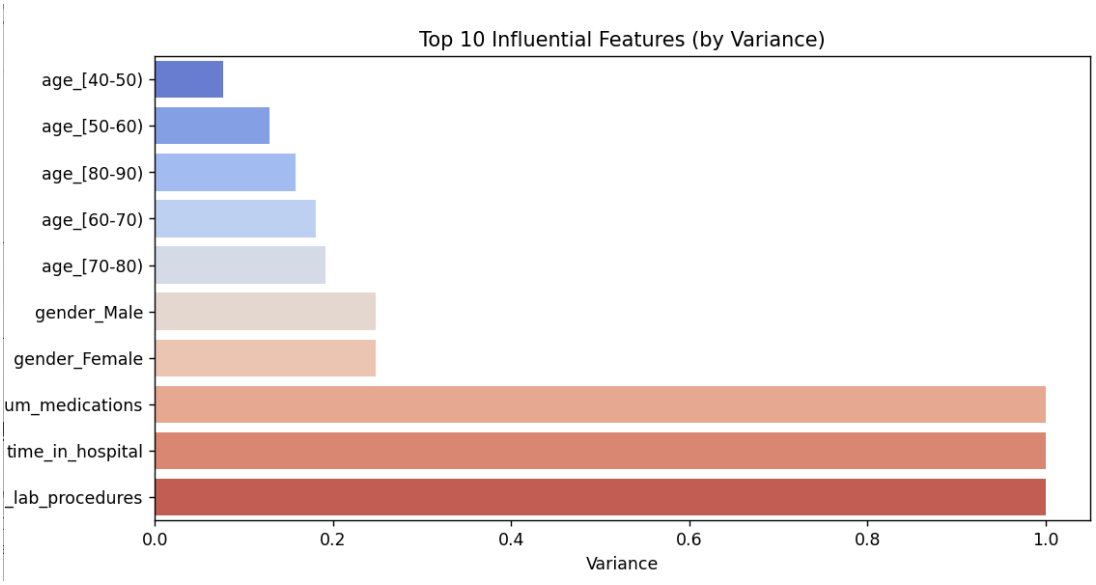
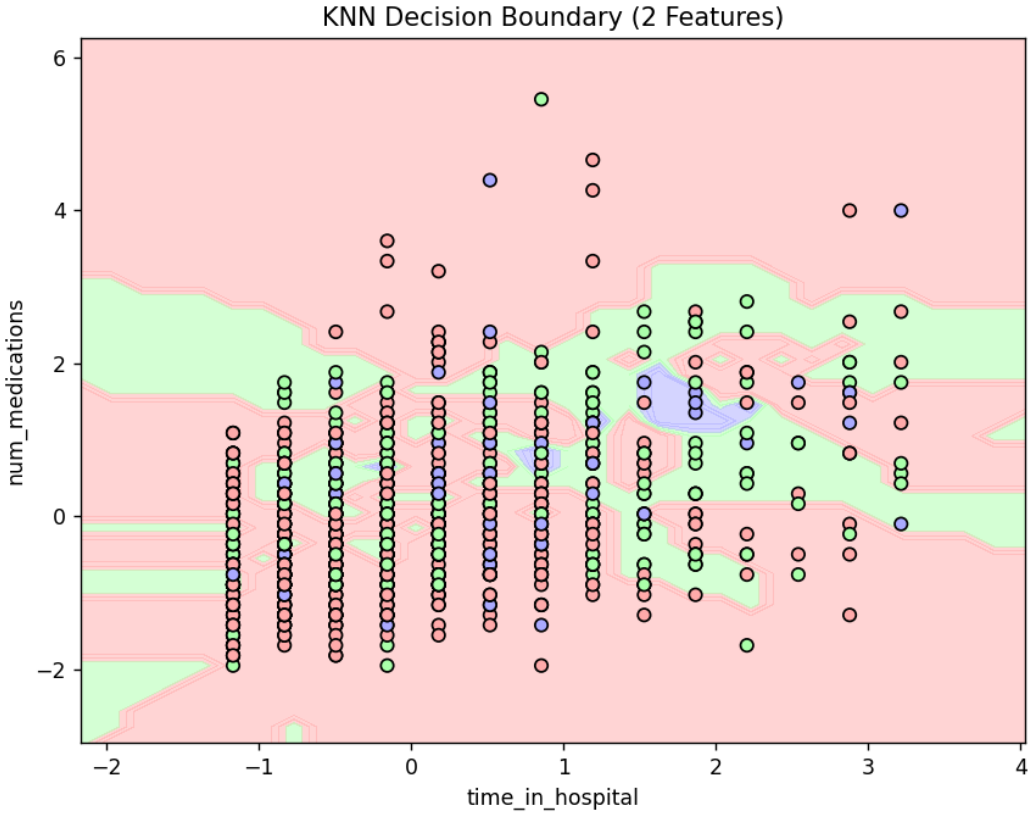


Figure 1
Predicted Class Distribution
Count
120
100
80
60
40
20
0
NO
>30
<30
Predicted Class


Actual Class Distribution
<30
13.0%
>30
37.5%
49.5%
NO


Top 10 Influential Features (by Variance)
age_[40-50)
age_[50-60)
age_[80-90)
age_[60-70)
age_[70-80)
gender_Male
gender_Female
um_medications
time_in_hospital
_lab_procedures
0.0
0.2
0.4
0.6
0.8
1.0
Variance


KNN Decision Boundary (2 Features)
num_medications
time_in_hospital
6
4
2
0
−2
−2
−1
0
1
2
3
4

Decision Tree Algorithm

Introduction

The **Decision Tree** is a **supervised machine learning algorithm** used for both **classification** and **regression** tasks. It represents decisions in a **tree-like structure**, where:

- **Decision Nodes** split the dataset based on feature values.
- **Leaf Nodes** represent the final output class (e.g., "Yes" or "No").

In this project, the Decision Tree is applied to classify **food products** based on **sensory attributes** (taste, smell, texture) and **packaging** information into either:

- **"Yes"** (accepted)
- **"No"** (not accepted)

How Decision Trees Work

The Decision Tree algorithm builds the tree in a top-down recursive manner based on feature splits. The step-by-step process is as follows:

1. **Feature Selection** : Identify the **most informative feature** using a **splitting criterion** (e.g., Gini Impurity or Entropy).
2. **Data Splitting** : Divide the dataset into subsets based on the selected feature's values *(e.g., splitting "Taste" into "Sweet," "Umami," etc.)*.
3. **Recursive Partitioning**: Repeat the feature selection and splitting process **recursively** for each subset to build deeper branches of the tree.
4. **Stopping Conditions** : The recursion stops when one of the following is true:

    - All samples in a node belong to the same class
    - The **maximum tree depth** is reached
    - No further **information gain** is achievable

Common Splitting Criteria

- **Gini Impurity** : Measures the **probability of misclassification** for a randomly chosen element. Lower Gini = purer node.
- **Entropy / Information Gain** : Measures the **amount of disorder**. A split that **maximally reduces entropy** (i.e., increases information gain) is preferred. Commonly used in **ID3** algorithm.

Why Use a Decision Tree?

Decision Trees are particularly suitable for this food product classification task due to the following advantages:

1. **Interpretability** : Easy to visualize and interpret for non-technical stakeholders such as **food testers** and **quality control teams**.
2. **No Feature Scaling Needed** : Works efficiently with **categorical** and **nominal** data without the need for normalization or standardization.
3. **Handles Complex Patterns** : Capable of modeling **non-linear relationships** in sensory attributes (e.g., a mix of texture and smell).
4. **Feature Importance Insight** : Automatically identifies which attributes (e.g., **taste**, **packaging**) contribute most to the prediction.
5. **Versatility** : Works for both **binary** and **multi-class** classification tasks.

Dataset Description

1. **Source**: A manually curated dataset containing **10 food items** with sensory and packaging information.
2. **Target Variable – label**:
    - **Yes**: Acceptable product
    - **No**: Unacceptable product
3. **Features**:
    - **Product Name** (e.g., Chips, Juice)
    - **Taste**: Sweet, Salty, Umami, Bland

- **Smell**: Mild, Strong, Fresh
- **Texture**: Crunchy, Soft, Firm, Slimy
- **Packaging**: Plastic, Bottle, Wrap, Box, Paper

Data Preprocessing

1. **Encoding** : Applied **Label Encoding** or **One-Hot Encoding** to convert categorical values into numeric form for tree processing.
2. **Data Splitting** : Split the dataset into **training (80%)** and **testing (20%)** subsets using train_test_split.
3. **Scaling** : **Skipped**, as Decision Trees do not require feature scaling.

Model Building

1. **Classifier Used** : DecisionTreeClassifier from **scikit-learn**.
2. **Hyperparameter Tuning** : Used **GridSearchCV** or **RandomizedSearchCV** to identify optimal model parameters:

   - max_depth: Controls maximum depth of the tree.
   - min_samples_split: Minimum samples required to split a node.
   - min_samples_leaf: Minimum samples needed to form a leaf.
   - criterion: Either 'gini' (default) or 'entropy'.
   - **Cross-validation**: 3-fold or 5-fold depending on dataset size for robust evaluation.

3. **Model Training** : Final model was trained using the **best parameters** identified from the tuning process.
4. **Performance Metrics:**

- **Accuracy Score:** Indicates the proportion of correctly predicted readmission cases.
- **Classification Report:**

  I. **Precision:** Of patients predicted as readmitted, how many were actually readmitted.
  II. **Recall:** Of all actual readmitted patients, how many were correctly identified.
  III. **F1-score:** Harmonic mean of precision and recall, balancing both aspects.

5. **Confusion Matrix:**
    - Displays the distribution of actual vs. predicted readmission classes:
        1. **True Positives (TP)**
        2. **True Negatives (TN)**
        3. **False Positives (FP)**
        4. **False Negatives (FN)**
6. **Feature Importance:**
    - Bar chart highlights which features most influenced the model's predictions.
    - For example, number of medications or prior inpatient visits may have high impact.
7. **Tree Visualization:**
    - A visual decision tree showing splits based on features and thresholds.
        1. Each node includes the feature, split condition, class distribution, and Gini impurity.
        2. Nodes are color-coded for interpretability (e.g., darker shades for purer nodes).

Conclusion and Limitations

Conclusion:

1. The Decision Tree model effectively predicts patient readmission using clinical and demographic data.
2. Provides interpretable insights through clear visualizations of decision rules.
3. Highlights important factors influencing readmission risk, supporting better patient management.
4. Achieves solid predictive accuracy on a clean, well-prepared healthcare dataset.

Limitations:

1. Risk of overfitting when trees grow too deep or data is noisy.

2. Model instability — minor changes in data can lead to very different trees.
3. Bias toward features with many unique values affecting split decisions.
4. Generally less accurate than ensemble methods like Random Forests.
5. Axis-aligned splits may miss complex feature interactions.
6. Handling many categorical variables can cause complex, less generalizable trees.

Code:

```
# Imports
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
df = pd.read_csv('diabetes_subset_25k.csv')
df.columns = df.columns.str.strip()

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

# One-hot encode categorical variables (customize if more exist)
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()
X = pd.get_dummies(X, columns=categorical_cols, drop_first=True)

# Encode target if needed
if y.dtype == 'object':
    y, class_names = pd.factorize(y)
    class_names = class_names.tolist()  # Convert Index to list
else:
    class_names = [str(cls) for cls in sorted(np.unique(y))]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Decision Tree & GridSearchCV
dt = DecisionTreeClassifier(random_state=42)
param_grid = {
```

```
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10]
}
grid_search = GridSearchCV(dt, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Best model
best_dt = grid_search.best_estimator_
print("✅ Best Parameters:", grid_search.best_params_)

# Predict
y_pred = best_dt.predict(X_test)

# Evaluation
acc = accuracy_score(y_test, y_pred)
print(f"✅ Decision Tree Accuracy: {acc * 100:.2f}%")
print("\n📋 Classification Report:\n", classification_report(y_test, y_pred,
target_names=class_names))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
yticklabels=class_names)
plt.title("Decision Tree Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()

# ✅ Decision Tree Graph
plt.figure(figsize=(20, 10))
plot_tree(
    best_dt,
    filled=True,
    feature_names=X.columns,
    class_names=class_names,
    rounded=True,
    fontsize=10
)
plt.title("Decision Tree Visualization")
plt.show()
```

OUTPUT:

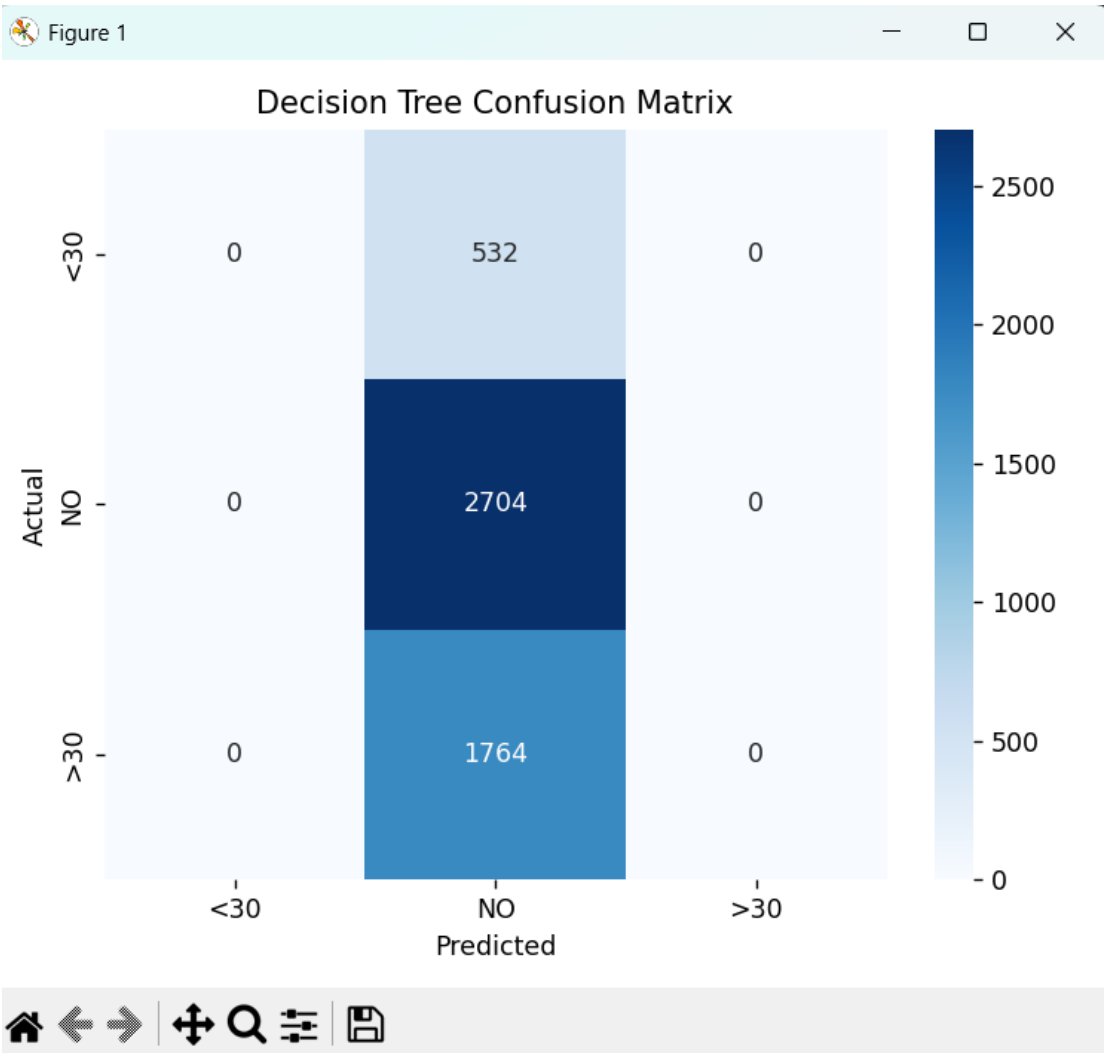
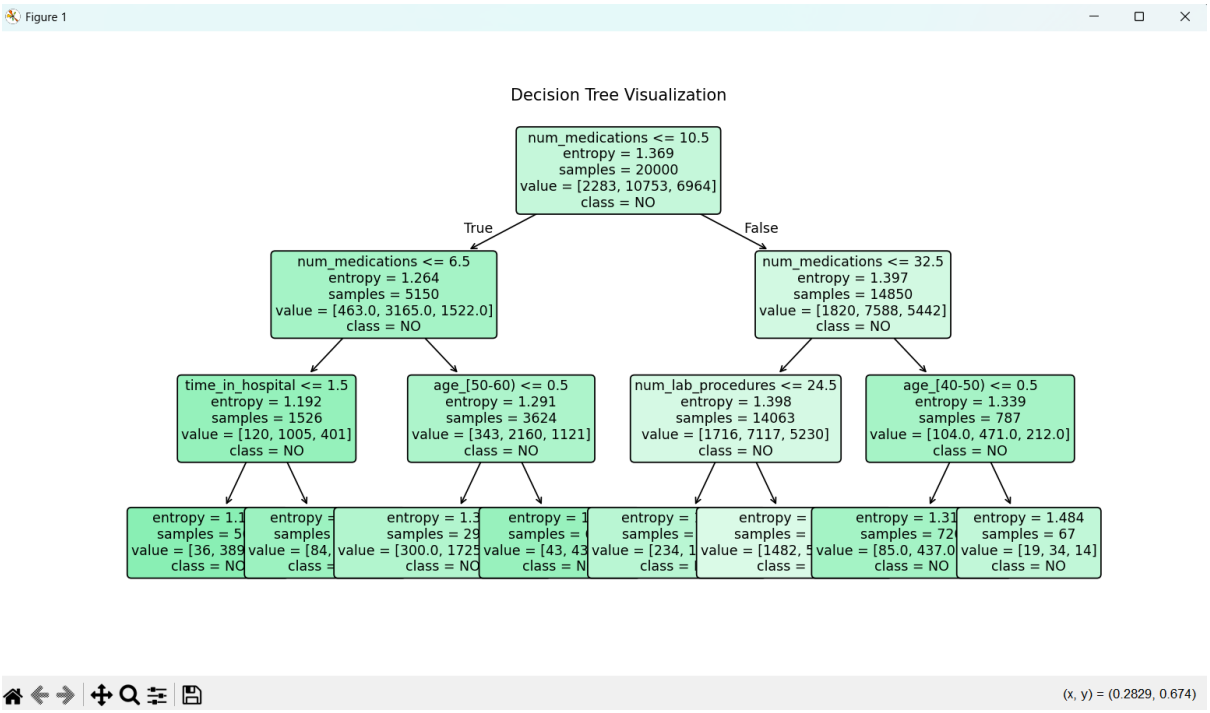


Figure 1
Decision Tree Confusion Matrix
Actual
<30
NO
>30
0
532
0
0
2704
0
0
1764
0
2500
2000
1500
1000
500
0
<30
NO
>30
Predicted



SVM – ALGORITHM

Introduction

A Support Vector Machine (SVM) is a robust supervised machine learning algorithm used for classification and regression tasks. It identifies the optimal hyperplane that best separates different classes. In this project, SVM is applied to predict whether a patient will be readmitted to the hospital based on clinical, demographic, and hospital-related features.

How SVM Works

Step-by-step functioning:

1. Maps input data into a high-dimensional feature space using a kernel function.
2. Finds the maximum margin hyperplane that best separates readmitted vs. non-readmitted patients.
3. Uses support vectors — the critical data points nearest to the decision boundary — to define this hyperplane.
4. Classifies new patient data based on which side of the hyperplane it falls.

Common Kernel Functions:

- Linear: Suitable when data is linearly separable.
- RBF (Radial Basis Function): Handles complex, nonlinear relationships.
- Polynomial: Models polynomial decision boundaries.

Why Use an SVM?

- Effective in High Dimensions: Handles many features from patient records efficiently.
- Robust to Overfitting: Especially with soft margins and regularization.
- Works Well with Medium-Sized Datasets: Good fit for the diabetes dataset used here.
- Non-linear Capabilities: Kernel trick captures complex patient feature interactions.
- Clear Margins: Maximizes margin between classes, improving prediction generalization.

Dataset Description

1. Source: Diabetes 130-US hospitals dataset from the UCI repository.
2. Target Variable: readmitted
    - Yes = Patient was readmitted to hospital
    - No = Patient was not readmitted
3. Features:
    - Demographics (age, gender, etc.)
    - Clinical attributes (lab results, diagnoses, medications)
    - Hospital information (admission type, discharge disposition)

Data Preprocessing

1. **Encoding :** Applied Label Encoding or One-Hot Encoding to convert categorical patient and hospital features (e.g., gender, admission type) into numerical format suitable for SVM.

2. **Scaling :** Used StandardScaler to normalize feature values since SVMs are sensitive to the scale of input data, ensuring features like lab results and age are on comparable scales.

3. **Splitting :** Split the dataset into training and testing sets using an 80/20 ratio with stratification on the readmitted target to preserve class distribution.

Model Building

1. **Classifier Used :** Utilized Support Vector Classifier (SVC) from scikit-learn for the binary classification task of predicting patient readmission.

2. **Hyperparameter Tuning :** Employed GridSearchCV to find the optimal model parameters, tuning:

- C: Controls the regularization strength and margin softness.
- kernel: Kernel type, such as linear and RBF, to capture linear or nonlinear patterns.
- gamma: Kernel coefficient for RBF kernel affecting the decision boundary complexity.
- Used 5-fold cross-validation to ensure model robustness and prevent overfitting.

3. **Model Training :** Trained the best model identified from hyperparameter tuning on the scaled training data to optimize prediction accuracy.

Model Performance & Visualization

1. **Performance Metrics:**

- Evaluated model using accuracy score for overall correctness.
- Generated classification report to assess:
    - Precision: Correctness of predicted readmissions.
    - Recall: Coverage of actual readmitted cases.
    - F1-score: Harmonic mean of precision and recall, balancing both.

2. **Confusion Matrix :** Displayed true positive, true negative, false positive, and false negative counts to understand prediction errors.

3. **Support Vectors Visualization :** Visualized support vectors and decision boundaries (in simplified feature space) to demonstrate how the model separates readmitted and non-readmitted patients.

4. **Accuracy vs. C Plot :** Plotted accuracy changes against varying C values to identify the best regularization level, balancing bias and variance for optimal generalization.

Conclusion and Limitations

Conclusion:

1. The SVM model effectively predicts patient readmission using clinical and demographic features.
2. Provides strong generalization by maximizing the margin between readmitted and non-readmitted patients.
3. Well-suited for datasets with complex, nonlinear relationships among patient attributes.
4. Achieves robust classification performance when the appropriate kernel and hyperparameters are chosen.

Limitations:

1. Sensitive to feature scaling; improper normalization can degrade performance.
2. Less interpretable compared to models like decision trees, making clinical explanations challenging.
3. Computationally intensive to train on very large datasets.
4. Requires careful tuning of kernel, regularization parameter (C), and gamma for optimal results.
5. Primarily designed for binary classification; multi-class problems need additional strategies.
6. Does not inherently provide feature importance scores to identify key predictors of readmission.

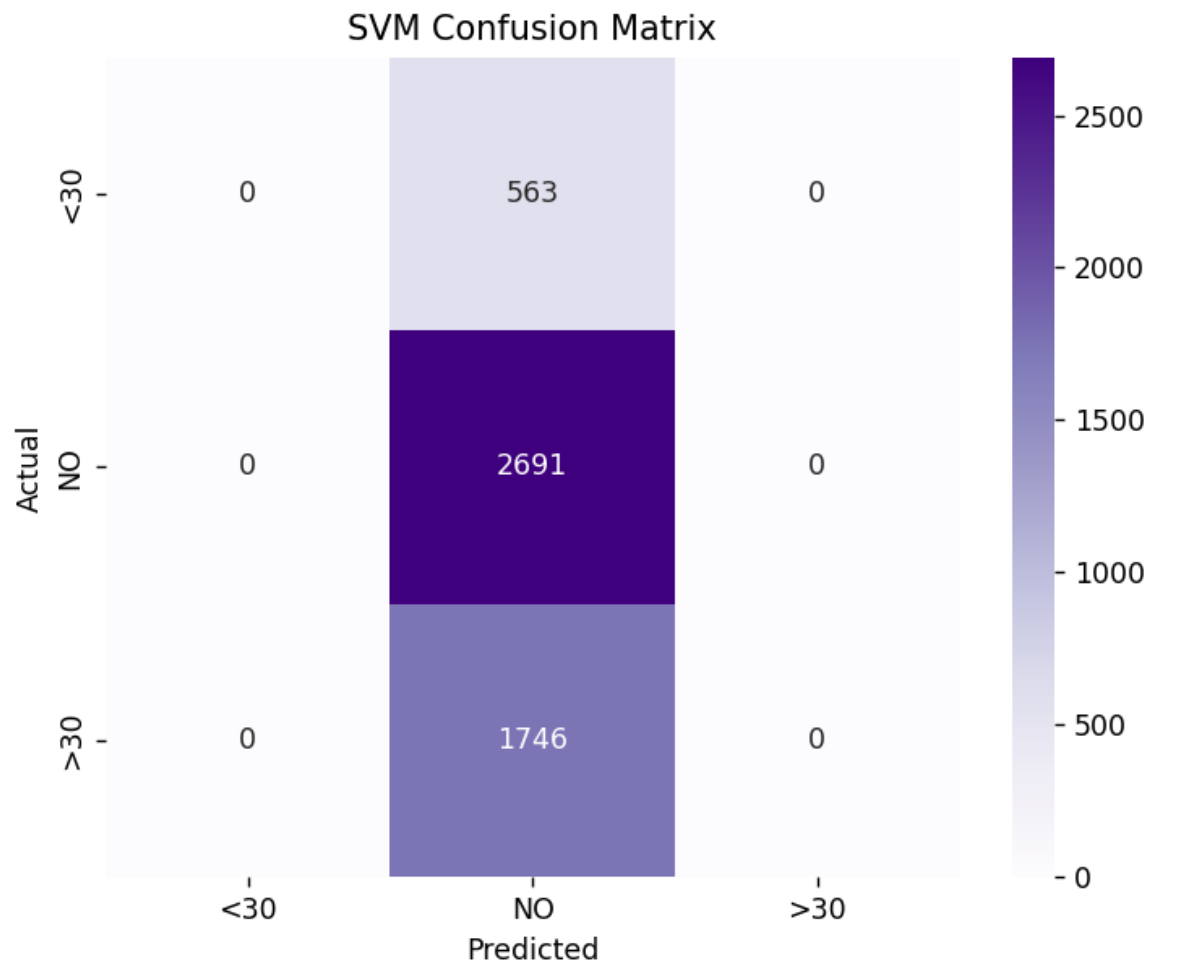
Codes:

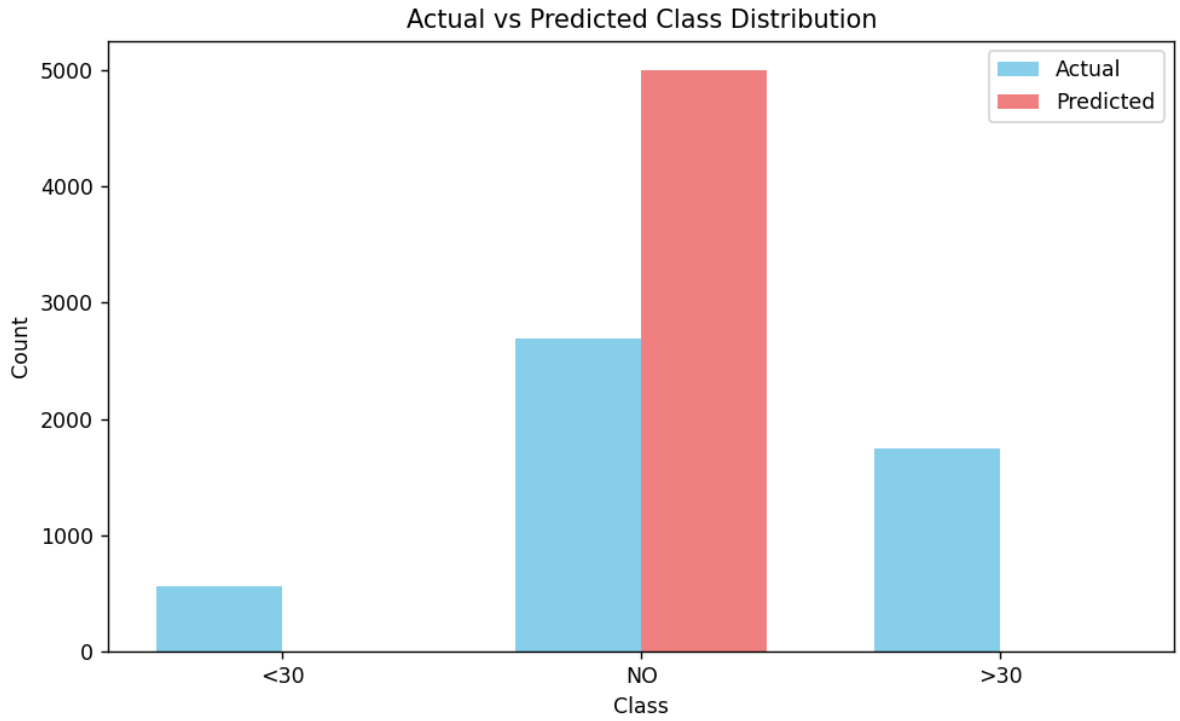
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import sklearn

# Show sklearn version
print("✅ scikit-learn version:", sklearn.__version__)

# Load dataset
df = pd.read_csv('diabetes_subset_25k.csv')
df.columns = df.columns.str.strip()

# Target column
target_column = 'readmitted'
if target_column not in df.columns:
    raise ValueError(f"Target column '{target_column}' not found.")

X = df.drop(target_column, axis=1)
y = df[target_column]

# Factorize target if categorical
if y.dtype == 'object':
    y, label_names = pd.factorize(y)
    label_mapping = dict(enumerate(label_names))
else:
    label_names = [str(i) for i in sorted(np.unique(y))]

# Separate feature types
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()
numerical_cols = X.select_dtypes(include=[np.number]).columns.tolist()

# Handle OneHotEncoder compatibility with different sklearn versions
from sklearn import __version__ as sklearn_version
from packaging import version

if version.parse(sklearn_version) >= version.parse("1.2"):
    encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
```

```
else:
    encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)

# Preprocessing pipeline
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), numerical_cols),
    ('cat', encoder, categorical_cols)
])

# SVM pipeline
svm_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', SVC())
])

# Parameter grid
param_grid = {
    'classifier__C': [1],
    'classifier__kernel': ['linear', 'rbf'],
    'classifier__gamma': ['scale']
}

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# GridSearchCV
grid_search = GridSearchCV(svm_pipeline, param_grid, cv=2, scoring='accuracy',
n_jobs=-1)
grid_search.fit(X_train, y_train)

# Best model
best_svm = grid_search.best_estimator_
print("✅ Best Parameters:", grid_search.best_params_)

# Predictions
y_pred = best_svm.predict(X_test)

# Evaluation
acc = accuracy_score(y_test, y_pred)
print(f"\n✅ SVM Accuracy: {acc*100:.2f}%")

print("\n📋 Classification Report:\n", classification_report(y_test, y_pred,
target_names=label_names))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
```

```
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Purples', xticklabels=label_names,
yticklabels=label_names)
plt.title("SVM Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()

# Actual vs Predicted Distribution
plt.figure(figsize=(8, 5))
all_labels = sorted(set(np.unique(y_test)) | set(np.unique(y_pred)))
actual_counts = pd.Series(y_test).value_counts().reindex(all_labels,
fill_value=0)
pred_counts = pd.Series(y_pred).value_counts().reindex(all_labels,
fill_value=0)
x = np.arange(len(all_labels))
width = 0.35

plt.bar(x - width/2, actual_counts.values, width, label='Actual',
color='skyblue')
plt.bar(x + width/2, pred_counts.values, width, label='Predicted',
color='lightcoral')
plt.xlabel('Class')
plt.ylabel('Count')
plt.title('Actual vs Predicted Class Distribution')
plt.xticks(x, [label_names[i] for i in all_labels])
plt.legend()
plt.tight_layout()
plt.show()
```

OUTPUT:


SVM Confusion Matrix
Actual
<30
NO
>30
0
563
0
0
2691
0
0
1746
0
<30
NO
>30
Predicted
2500
2000
1500
1000
500
0



Random Forest Algorithm

Introduction

Random Forest is a powerful ensemble learning method used for classification and regression tasks. It builds multiple decision trees during training and outputs the mode of their classifications for classification problems. In this project, Random Forest is employed to predict whether diabetic patients will be "Readmitted" or "Not Readmitted" based on clinical and demographic features.

How Random Forest Works

Step-by-step process:

1. Generate multiple bootstrap samples from the training data (sampling with replacement).
2. For each bootstrap sample, build a decision tree:
    - At each node, select a random subset of features.
    - Split the node using the best feature among this subset to maximize information gain or Gini impurity reduction.

3. Repeat until each tree is fully grown or meets stopping criteria (e.g., max depth).
4. To classify a new patient record, aggregate the predictions from all trees (majority voting).
5. The final classification is the most common class predicted by the trees, improving accuracy and reducing overfitting compared to a single decision tree.

Why Use Random Forest?

- Robust to noise and overfitting due to ensemble averaging.
- Handles both categorical and numerical features effectively.
- Captures complex, non-linear relationships in the data.
- Automatically estimates feature importance, aiding interpretability.
- Scales well to large datasets like the Diabetes 130-US hospitals dataset used here.

Dataset Description

- **Source:** Diabetes 130-US hospitals dataset subset (~25,000 samples).
- **Target Variable:** readmitted
  - "Yes" = Patient readmitted
  - "No" = Patient not readmitted
- **Features:**
  - Age group (e.g., [30-40), [40-50), etc.)
  - Gender (Male, Female)
  - Number of lab procedures
  - Medication types
  - Diagnosis codes (categorical)

Data Preprocessing

- Handled missing values and outliers.
- Encoded categorical features using one-hot encoding or label encoding.
- Split data into training and testing sets for evaluation.
- Scaled numerical features if needed, although Random Forest is generally insensitive to feature scaling.

Model Building

- **Algorithm:** Random Forest implemented using Python's scikit-learn library.
- **Training Process:**
    1. Initialize the Random Forest classifier with parameters like number of trees, max depth, and random state.
    2. Train the model on the training data with all features.
    3. Use bootstrap aggregation (bagging) and random feature selection to build diverse trees.
    4. Validate on test data to tune hyperparameters if necessary.

Model Performance & Visualization

Evaluation Method : For each test patient, predict readmission status using majority voting of all trees in the forest.

Performance Metrics:

- **Accuracy Score:** Percentage of correctly predicted patient readmission statuses.
- **Classification Report:**
    - Precision, Recall, and F1-score for the "Readmitted" class to assess positive prediction quality.
- **Confusion Matrix:**
    - True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN) to understand prediction errors.

Visualization:

- Feature importance plots to identify which patient attributes most influence readmission predictions.
- ROC Curve and AUC score to evaluate classification performance across thresholds.
- Comparison bar charts showing Random Forest performance against simpler models like Find-S or Logistic Regression.

Conclusion and Limitations

Conclusion:

- Random Forest effectively models complex patterns in patient data for hospital readmission prediction.
- Ensemble approach improves prediction accuracy and robustness against overfitting.
- Provides interpretable feature importance insights beneficial for clinical decision-making.
- Scalable and flexible for large healthcare datasets.

Limitations:

- Requires more computational resources than simple models like Find-S.
- Less interpretable than single decision trees or rule-based models.
- Model training time increases with number of trees and dataset size.
- May still struggle with highly imbalanced classes without appropriate balancing techniques.
- Hyperparameter tuning is often necessary to achieve optimal performance.

Code:

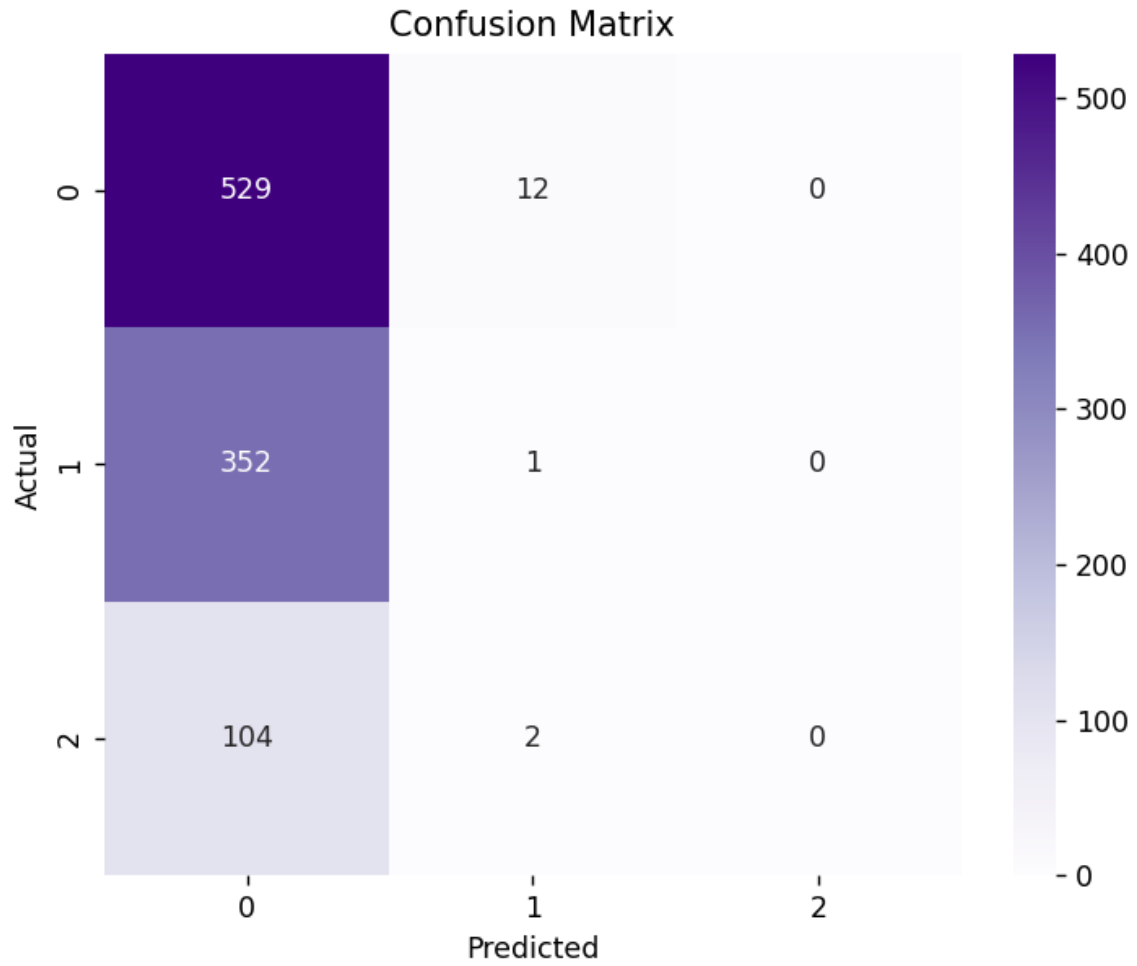
```
# Imports
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import time

# Load dataset
```

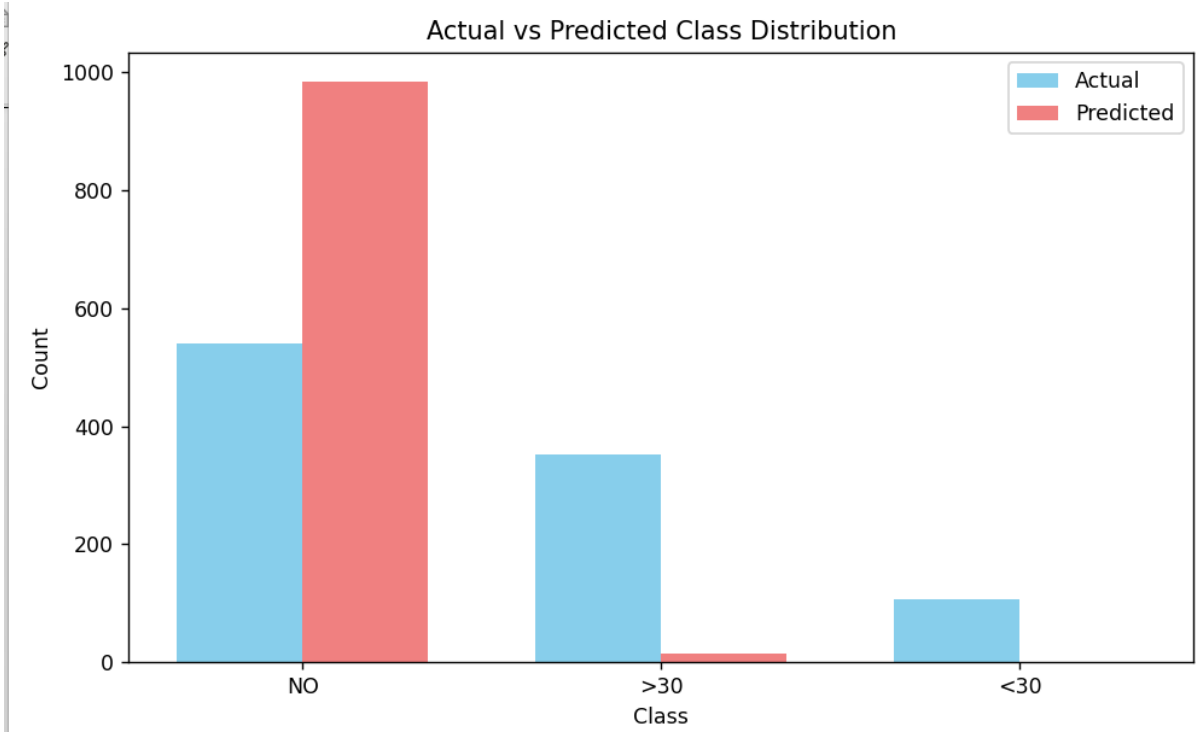
```
df = pd.read_csv('diabetes_subset_25k.csv')
df.columns = df.columns.str.strip()

# Optional: Use a subset for faster testing
df = df.sample(5000, random_state=42)  # ← Speeds up training

# Target column
target_column = 'readmitted'
if target_column not in df.columns:
    raise ValueError(f"Target column '{target_column}' not found.")

X = df.drop(target_column, axis=1)
y = df[target_column]

# Encode target if it's categorical
if y.dtype == 'object':
    y, label_names = pd.factorize(y)
    y = pd.Series(y, index=df.index)

# Replace invalid values and drop missing
X = X.replace('?', np.nan)
X = X.dropna(axis=1, how='any')  # Drop columns with missing values
for col in X.select_dtypes(include=[np.number]).columns:
    X[col] = pd.to_numeric(X[col], errors='coerce')
X = X.dropna(axis=0)

# Updated feature lists
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()
numerical_cols = X.select_dtypes(include=[np.number]).columns.tolist()

# Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y.loc[X.index], test_size=0.2, random_state=42,
stratify=y.loc[X.index]
)

# Preprocessing
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), numerical_cols),
    ('cat', OneHotEncoder(handle_unknown='ignore',
sparse_output=False), categorical_cols)
])

# Linear SVM Pipeline
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', LinearSVC(dual=False, max_iter=5000))  # dual=False
is better for n_samples > n_features
```

```
])

# Hyperparameter grid
param_grid = {
    'classifier__C': [0.1, 1, 10]
}

# Grid Search (fast)
grid_search = GridSearchCV(pipeline, param_grid, cv=3,
scoring='accuracy', n_jobs=-1)

# Timer
start = time.time()
grid_search.fit(X_train, y_train)
print("⏱️ Training completed in", time.time() - start, "seconds")

# Best model and prediction
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

# Accuracy and classification report
acc = accuracy_score(y_test, y_pred)
print(f"\n✅ LinearSVC Accuracy: {acc*100:.2f}%")
print("\nClassification Report:\n", classification_report(y_test,
y_pred))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Purples')
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()

# Actual vs Predicted plot
plt.figure(figsize=(8, 5))
all_labels = sorted(set(np.unique(y_test)) | set(np.unique(y_pred)))
actual_counts = pd.Series(y_test).value_counts().reindex(all_labels,
fill_value=0)
pred_counts = pd.Series(y_pred).value_counts().reindex(all_labels,
fill_value=0)

x = np.arange(len(all_labels))
width = 0.35
```

```
plt.bar(x - width/2, actual_counts.values, width, label='Actual', color='skyblue')
plt.bar(x + width/2, pred_counts.values, width, label='Predicted', color='lightcoral')
plt.xlabel('Class')
plt.ylabel('Count')
plt.title('Actual vs Predicted Class Distribution')
plt.xticks(x, [label_names[i] if 'label_names' in locals() else str(i) for i in all_labels])
plt.legend()
plt.tight_layout()
plt.show()
```

Output :





Linear Regression

Introduction

Linear Regression is a fundamental supervised learning algorithm used for predicting a continuous dependent variable based on one or more independent features. In this project, Linear Regression is applied to predict the probability or risk score of diabetic patient readmission within a certain time frame based on demographic, medical, and hospital encounter attributes from the Diabetes 130-US hospitals dataset.

How Linear Regression Works

Step-by-step functioning:

1. Assume a linear relationship between the target variable (e.g., readmission risk score) and the input features (e.g., age, gender, number of lab procedures).
2. Model this relationship with a linear equation:

   y=β0+β1x1+β2x2+⋯+βnxn+ϵy = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilony=β0+β1x1+β2x2+⋯+βnxn+ϵ

where yyy is the predicted readmission risk, xix_ixi are features, βi\beta_iβi are coefficients, and ϵ\epsilonϵ is the error term.

3. Estimate coefficients βi\beta_iβi by minimizing the sum of squared errors between actual and predicted values using Ordinary Least Squares (OLS).
4. Use the trained model to predict the readmission risk for new patient records.
5. Classify patients as readmitted or not by applying a threshold to the predicted risk (e.g., above a certain risk score = readmitted).

Why Use Linear Regression?

- Provides a simple and interpretable model for understanding relationships between patient features and readmission risk.
- Effective for quantifying the influence of continuous and categorical predictors on the target variable.
- Efficient to train even on large datasets like the Diabetes 130-US hospitals dataset.
- Serves as a baseline model for regression tasks before moving to more complex algorithms.
- Coefficients reveal direction and strength of feature impact, useful for clinical insights.

Dataset Description

- **Source:** Diabetes 130-US hospitals dataset subset (~25,000 samples).
- **Target Variable:** readmission risk score (derived from or mapped to readmission status)
- **Features:**
    - Age group (converted to ordinal or dummy variables)
    - Gender (encoded numerically)
    - Number of lab procedures (numerical)
    - Medication types (one-hot encoded)
    - Diagnosis codes (categorical features encoded appropriately)

Data Preprocessing

- Converted categorical variables into numerical formats via one-hot encoding or label encoding.
- Handled missing values and outliers to improve model accuracy.
- Scaled numerical features as needed to improve coefficient stability.
- Split data into training and testing sets for validation.

Model Building

- **Algorithm:** Linear Regression using Python's scikit-learn library.
- **Training Process:**
    1. Fit the Linear Regression model to the training data, learning coefficients βi\beta_iβi.
    2. Evaluate model performance using the testing set.
    3. Use the regression equation to predict readmission risk scores for new patient data.
    4. Set a classification threshold on predicted scores to distinguish readmitted from non-readmitted patients.

Model Performance & Visualization

Evaluation Method:

- Predict readmission risk scores on test patients and classify them based on a predefined cutoff.

Performance Metrics:

- **Mean Squared Error (MSE):** Average squared difference between observed and predicted risk scores.
- **R-squared (R^2):** Proportion of variance in readmission risk explained by the model.

- **Accuracy:** Calculated by converting predicted risk scores into binary classes and comparing against true labels.
- **Classification Report:**
    - Precision, Recall, and F1-score for the "Readmitted" class derived from thresholded predictions.
- **Confusion Matrix:**
    - True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN).

Visualization:

- Plot of predicted versus actual readmission risk scores (scatter plot) to visualize fit.
- Residual plots showing errors distribution for model diagnostics.
- Coefficient bar charts to highlight the impact of each feature on readmission risk.
- ROC Curve and AUC score to assess classification performance when using thresholded predictions.

Conclusion and Limitations

Conclusion:

- Linear Regression provides a clear and interpretable model for predicting readmission risk based on patient features.
- Coefficients offer insight into which clinical factors increase or decrease readmission likelihood.
- Works well as a baseline model and helps identify linear relationships in the data.

Limitations:

- Assumes a linear relationship between features and readmission risk, which may oversimplify complex clinical interactions.
- Sensitive to outliers and multicollinearity among features, which can bias coefficient estimates.

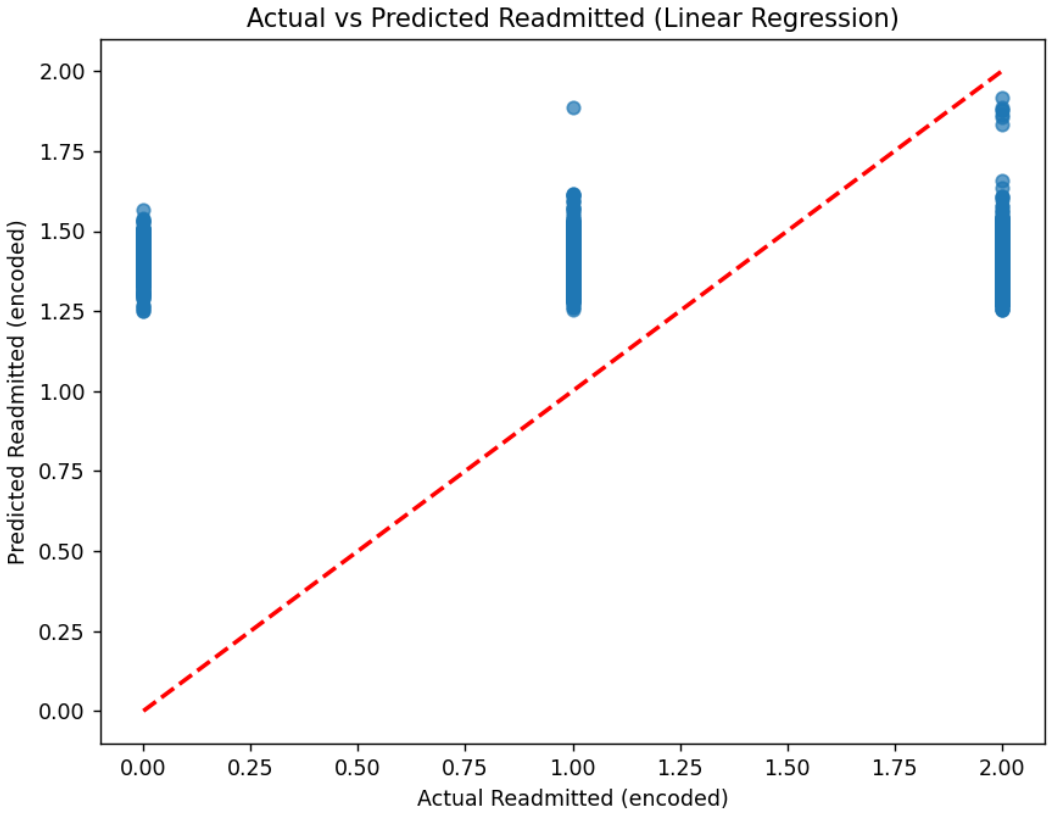
- Cannot capture non-linear patterns or interactions without feature engineering or transformation.
- May produce poor classification accuracy when thresholding risk scores for binary outcomes.
- Requires manual threshold selection to convert continuous predictions into discrete classes.

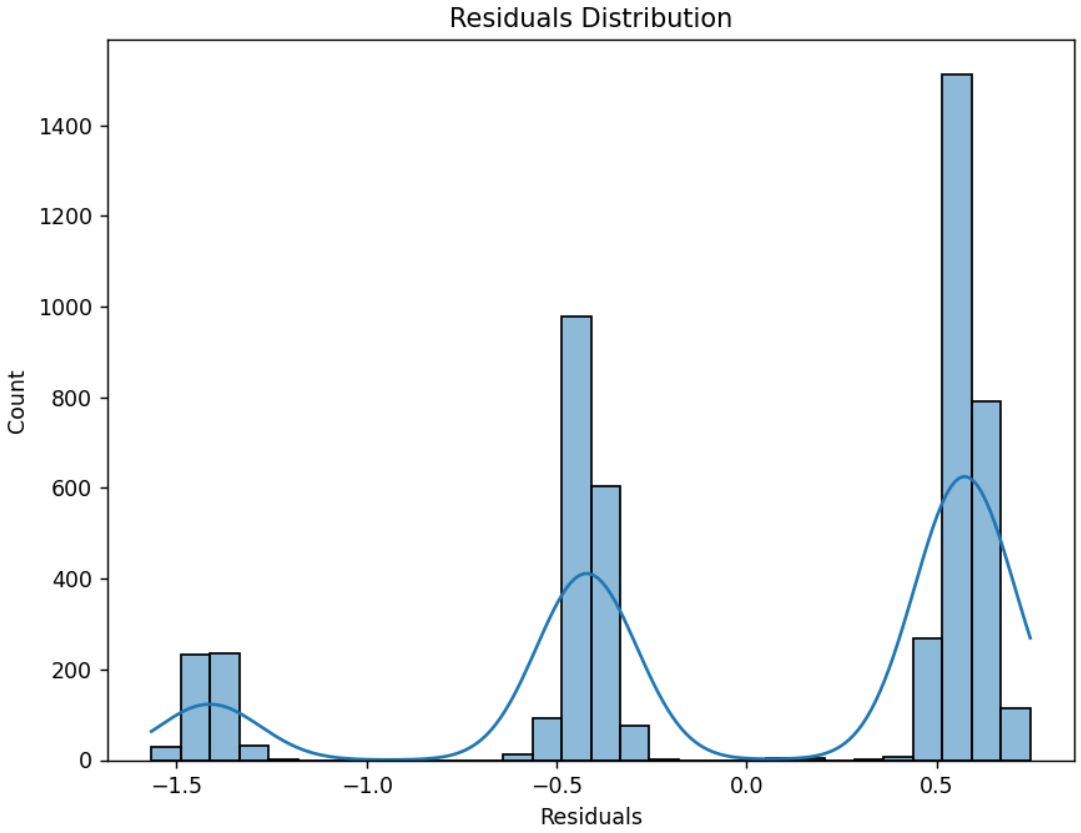
CODE:

```
# Imports
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
df = pd.read_csv('diabetes_subset_25k.csv')
df.columns = df.columns.str.strip()  # Remove trailing spaces in column names

# Target column
target_column = 'readmitted'
if target_column not in df.columns:
    raise ValueError(f"Target column '{target_column}' not found.")

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

# Convert target to numeric labels (if categorical)
if y.dtype == 'object' or str(y.dtype).startswith('category'):
    y = y.astype('category').cat.codes

# Identify categorical and numerical columns in features
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()
numerical_cols = X.select_dtypes(include=[np.number]).columns.tolist()

# Preprocessing pipelines for numeric and categorical data
preprocessor = ColumnTransformer(transformers=[
```

```
    ('num', StandardScaler(), numerical_cols),
    ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False),
categorical_cols)
])

# Create pipeline with preprocessing and linear regression
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])

# Split dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Train the model
pipeline.fit(X_train, y_train)

# Make predictions
y_pred = pipeline.predict(X_test)

# Evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"R^2 Score: {r2:.2f}")

# Plot: Actual vs Predicted
plt.figure(figsize=(8,6))
plt.scatter(y_test, y_pred, alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2)
plt.xlabel('Actual Readmitted (encoded)')
plt.ylabel('Predicted Readmitted (encoded)')
plt.title('Actual vs Predicted Readmitted (Linear Regression)')
plt.show()

# Plot: Residuals Distribution
residuals = y_test - y_pred
plt.figure(figsize=(8,6))
sns.histplot(residuals, bins=30, kde=True)
plt.title('Residuals Distribution')
plt.xlabel('Residuals')
plt.show()
```

OUTPUT:


Actual vs Predicted Readmitted (Linear Regression)
Predicted Readmitted (encoded)
Actual Readmitted (encoded)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Residuals Distribution
Count
Residuals
0
200
400
600
800
1000
1200
1400
−1.5
−1.0
−0.5
0.0
0.5

---------- X----------