```
# Step 1: Import Libraries and Load Dataset
import pandas as pd
import numpy as np
from zipfile import ZipFile
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA
import gym
from gym import spaces
import random
from collections import defaultdict

# Unzip and Load the dataset
file_path = 'diabetes+130-us+hospitals+for+years+1999-2008.zip'
with ZipFile(file_path, 'r') as zip_ref:
    csv_filename = zip_ref.namelist()[0]
    with zip_ref.open(csv_filename) as file:
        data = pd.read_csv(file)

# Display basic dataset information
print("Dataset Information:")
print(data.info())
print("\nFirst 5 Rows:")
print(data.head())
```

```
# Step 2: Data Preprocessing and Target Separation
# Define the target column
target_column = 'readmitted'  # Change if needed
X = data.drop(columns=[target_column])
y = data[target_column]

# Encode categorical columns
X = pd.get_dummies(X, drop_first=True)

# Encode the target if it's categorical
if y.dtype == 'object':
    label_encoder = LabelEncoder()
    y = label_encoder.fit_transform(y)

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

# Standardize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 3: Classification Using Random Forest
# Train Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=42)
rf_classifier.fit(X_train, y_train)

# Predictions and metrics
y_pred = rf_classifier.predict(X_test)
print("\nRandom Forest Classifier Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix for Random Forest Classifier")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

# Step 4: Clustering Using K-Means
# K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X_train)
clusters = kmeans.predict(X_train)
silhouette_avg = silhouette_score(X_train, clusters)
print("\nSilhouette Score for K-Means Clustering:", silhouette_avg)

# PCA for Visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_train)
plt.figure(figsize=(10, 8))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=clusters, cmap='viridis', s=50, alpha=0.6)
plt.title("K-Means Clustering Visualization (PCA-reduced data)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.colorbar(label="Cluster")
plt.show()

# Step 5: Reinforcement Learning Environment Setup and Q-Learning
# Define a custom gym environment for feature selection
```

```
class DiabetesEnv(gym.Env):
    def __init__(self, data, target):
        super(DiabetesEnv, self).__init__()
        self.data = data.select_dtypes(include=[np.number])  # Only numeric features
        self.target = target
        self.action_space = spaces.Discrete(self.data.shape[1])  # One action per feature
        self.observation_space = spaces.Box(low=0, high=1, shape=(self.data.shape[1],))
        self.reset()

    def reset(self):
        self.state = np.random.choice(self.data.index)
        return self.data.iloc[self.state].values

    def step(self, action):
        reward = np.corrcoef(self.data.iloc[:, action], self.target)[0, 1]
        done = True  # Single-step environment
        return self.data.iloc[self.state].values, reward, done, {}

# Initialize the environment
env = DiabetesEnv(pd.DataFrame(X_train), y_train)

# Q-Learning parameters
num_episodes = 100
learning_rate = 0.1
discount_factor = 0.9
epsilon = 0.1  # Epsilon-greedy

# Q-table
q_table = defaultdict(lambda: np.zeros(env.action_space.n))

for episode in range(num_episodes):
```

```
    state = env.reset()
    done = False
    while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(q_table[state.tobytes()])

        next_state, reward, done, _ = env.step(action)
        best_next_action = np.argmax(q_table[next_state.tobytes()])
        td_target = reward + discount_factor * q_table[next_state.tobytes()][best_next_action]
        td_error = td_target - q_table[state.tobytes()][action]
        q_table[state.tobytes()][action] += learning_rate * td_error
        state = next_state

# Display sample Q-values after training
sample_q_values = dict(list(q_table.items())[:5])
print("\nSample Q-table values after training:", sample_q_values)

# Graph of rewards per action
actions = list(range(env.action_space.n))
rewards = [np.corrcoef(env.data.iloc[:, a], env.target)[0, 1] for a in actions]
plt.figure(figsize=(10, 6))
plt.bar(actions, rewards, color='blue', alpha=0.7)
plt.xlabel("Action (Feature Index)")
plt.ylabel("Correlation with Target (Reward)")
plt.title("Reward per Action (Feature)")
plt.show()
```