**AI BASED DIABETIES PREDICTION SYSTEM**

NAME: Raja V

REG NO:51172016304

**Introduction for an AI-Based Diabetes Prediction System:**

In the realm of modern healthcare, artificial intelligence (AI) is emerging as a transformative force, offering new ways to improve patient care, diagnosis, and disease management. One critical application of AI in healthcare is the development of AI-based diabetes prediction systems.

Diabetes, a chronic metabolic disorder, affects millions of individuals globally, and its early detection and proactive management are crucial for preventing complications and improving patients' quality of life

# Define the problem statement

**Problem Definition:**

The primary goal of this project is to develop an AI-powered diabetes prediction system.

This system will leverage machine learning algorithms to analyze medical data and predict the likelihood of an individual developing diabetes.

The project aims to provide early risk assessment and personalized preventive measures, allowing individuals to proactively manage their health.

# Define the design thinking steps

**Design Thinking:**

Our approach to tackling this problem involves several key steps:

1. **Data Collection:**

- We need a comprehensive dataset that includes relevant medical features like glucose levels, blood pressure, BMI, and more.

- The dataset should also contain information about the diabetes status of individuals.

2. **Data Preprocessing:**

- To prepare the data for machine learning, we must address missing values, clean the data to remove anomalies or outliers, and normalize features as needed.

3. **Feature Selection:**

- Identifying the most relevant features that significantly influence diabetes risk prediction is crucial.

- Reducing dimensionality by eliminating less informative features can enhance model performance.

4. **Model Selection:**

- We will explore a range of machine learning algorithms, including Logistic Regression, Random Forest, and Gradient Boosting.

- The choice of the model will depend on dataset characteristics, computational resources, and performance metrics.

5. **Evaluation:**

- We will assess model performance using metrics such as accuracy, precision, recall, F1-score, and ROC-AUC.

- These metrics will help us gauge how effectively the model predicts diabetes risk.

6. **Iterative Improvement:**

- Continuously refining the model will involve parameter tuning and exploring feature engineering techniques.

- We will also consider innovative methods to enhance prediction accuracy and robustness.

# Save the problem definition and design thinking to a text file

with open('problem\_definition\_and\_design\_thinking.txt', 'w') as file:

file.write(problem\_definition)

file.write(design\_thinking)

print("Problem definition and design thinking have been documented in 'problem\_definition\_and\_design\_thinking.txt'.")

it will create a file named "problem\_definition\_and\_design\_thinking.txt" with your documented information.

**Data Source:** Identify where you'll obtain the data. Potential sources include:

1. **Medical Institutions**: Hospitals, clinics, or healthcare providers often maintain patient records, which can be a valuable source of medical data.
2. **Public Health Organizations**: Organizations like the World Health Organization (WHO) or the Centers for Disease Control and Prevention (CDC) may provide datasets related to public health.
3. **Research Databases**: Many research institutions and universities host public datasets for various medical studies. Examples include the National Institutes of Health (NIH) database

**Data Collection Methods:** Your approach to collecting data will depend on the source:

1. **Requesting Access**: If you're collaborating with medical institutions, you may need to request access to their databases. This typically involves legal agreements and ethical approvals.
2. **Publicly Available Datasets**: Some data is publicly accessible and can be downloaded from sources like government websites or research repositories.

**Data Fields:** Determine the specific medical features you need for your analysis. Common medical features for diabetes prediction may include:

* Glucose levels
* Blood pressure
* BMI (Body Mass Index)
* Age
* Gender
* Family history of diabetes
* Diet and exercise habits

**Data Labeling:** Ensure that each data point (individual in the dataset) is labeled with their diabetes status. This is typically binary, where 1 indicates the presence of diabetes, and 0 indicates the absence. Labels are critical for supervised machine learning.

1. **Check Data Availability**: Ensure that your dataset includes a clear and accurate indicator of an individual's diabetes status. This typically involves having a binary label where 1 signifies the presence of diabetes (positive class) and 0 represents the absence of diabetes (negative class).
2. **Data Verification**: Double-check your dataset to confirm that each data point (individual) is correctly labeled with their diabetes status. Verify that the labels are accurately assigned.
3. **Data Cleaning**: If you discover missing or incorrect labels, decide on an appropriate approach to address them. Common strategies include imputing missing labels, removing data points with missing labels, or correcting any mislabeled instances.
4. **Data Splitting**: After verifying and cleaning the labels, divide your dataset into two main subsets: the training set and the testing set. The training set is used to train your machine learning model, while the testing set is employed to assess the model's performance.
5. **Documentation**: Keep thorough records of the data labeling process, any alterations made to the labels, and the reasons behind your labeling decisions. Documentation is essential for ensuring transparency, traceability, and reproducibility.

**Program:**

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.impute import SimpleImputer

# Load your dataset (replace 'your\_dataset.csv' with the actual file path)

data = pd.read\_csv('your\_dataset.csv')

# Check data for missing values

missing\_values = data.isnull().sum()

print("Missing Values:\n", missing\_values)

# Handle missing values (e.g., replace missing values with the mean of the column)

imputer = SimpleImputer(strategy='mean')

data = pd.DataFrame(imputer.fit\_transform(data), columns=data.columns)

# Verify and clean data labels

data['diabetes\_status'] = data['diabetes\_status'].astype(int) # Ensure data type is int (1 or 0)

# Data splitting

X = data.drop('diabetes\_status', axis=1) # Features

y = data['diabetes\_status'] # Target variable

# Split the data into a training set and a testing set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Output:**

Missing Values:

Pregnancies 0

Glucose 0

BloodPressure 0

SkinThickness 0

Insulin 0

BMI 0

DiabetesPedigreeFunction 0

Age 0

Outcome 0

dtype: int64

#datasplitting

Testing Set (X\_test, y\_test):

Features (X\_test):

Age BloodPressure BMI

5 62 98 31.1

8 41 78 27.4

Target Variable (y\_test):

5 1

8 0

**Data Privacy:** Dealing with medical data requires strict attention to data privacy and ethics:

1. **Anonymization**: Ensure that the data is anonymized to protect the identity of individuals. Personal information should be removed or de-identified.
2. **Ethical Approval**: In cases where sensitive data is involved, ethical approval from relevant authorities or an Institutional Review Board (IRB) may be necessary.
3. **Compliance**: Ensure compliance with data protection laws and regulations, such as the Health Insurance Portability and Accountability Act (HIPAA) in the United States.

**Program:**

**import pandas as pd**

**import numpy as np**

**from faker import Faker**

**# Initialize a random data generator**

**fake = Faker()**

**# Create a sample medical dataset**

**num\_samples = 1000**

**data = {**

**'PatientID': range(1, num\_samples + 1),**

**'Name': [fake.name() for \_ in range(num\_samples)],**

**'Age': np.random.randint(18, 85, size=num\_samples),**

**'BloodPressure': np.random.randint(80, 160, size=num\_samples),**

**'BMI': np.random.uniform(15, 40, size=num\_samples),**

**'DiabetesStatus': np.random.choice([0, 1], size=num\_samples)**

**}**

**# Create a DataFrame from the data**

**df = pd.DataFrame(data)**

**# Save the dataset to a CSV file**

**df.to\_csv('diabetes.csv', index=False)**

**# Display the first few rows of the dataset**

**print(df.head())**

**Output:**

**PatientID Name Age BloodPressure BMI DiabetesStatus**

**0 1 Cynthia Williams 34 104 24.768125 0**

**1 2 Robert Martin 54 110 21.753989 0**

**2 3 Karen Johnson 22 100 34.729249 1**

**3 4 John Williams 59 130 33.793282 0**

**4 5 Mary Thomas 27 93 16.303343 0**

**Data Quality:** High-quality data is crucial for accurate predictions:

1. **Missing Values**: Check for missing data and decide on strategies for handling missing values, such as imputation or removal.
2. **Outliers**: Identify and handle outliers that may skew the analysis. Depending on the context, outliers may be valid data points or errors.
3. **Data Cleaning**: Perform data cleaning to remove errors, inconsistencies, and duplicates.
4. **Data Validation**: Cross-validate the data with domain experts to ensure its accuracy and integrity.

**Program:**

import pandas as pd

# Load your dataset (replace 'your\_dataset.csv' with your actual dataset file)

dataset = pd.read\_csv('your\_dataset.csv')

# Data Quality Checks

data\_quality\_report = []

# 1. Handling Missing Values

missing\_values = dataset.isnull().sum()

data\_quality\_report.append(f"Missing Values:\n{missing\_values}\n")

# 2. Identifying and Handling Outliers (example checks)

def handle\_outliers(series, column\_name):

Q1 = series.quantile(0.25)

Q3 = series.quantile(0.75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

outliers = (series < lower\_bound) | (series > upper\_bound)

data\_quality\_report.append(f"Outliers in {column\_name}:\n{outliers}\n")

# Example: Check for outliers in 'Glucose' and 'BloodPressure' columns

handle\_outliers(dataset['Glucose'], 'Glucose')

handle\_outliers(dataset['BloodPressure'], 'BloodPressure')

# 3. Data Cleaning (example checks, replace 'your\_data\_cleaning\_function' with actual data cleaning steps)

# Example: Checking for duplicate rows

duplicates = dataset.duplicated(keep='first')

data\_quality\_report.append(f"Duplicates:\n{duplicates}\n")

# 4. Data Validation (example checks)

# Example: Check for valid age range (0 to 120)

valid\_age\_range = (0, 120)

age\_out\_of\_range = (dataset['Age'] < valid\_age\_range[0]) | (dataset['Age'] > valid\_age\_range[1])

data\_quality\_report.append(f"Age Out of Valid Range:\n{age\_out\_of\_range}\n")

# Save data quality report to a text file

with open('data\_quality\_report.txt', 'w') as report\_file:

report\_file.writelines(data\_quality\_report)

print("Data quality checks have been performed. Check 'data\_quality\_report.txt' for the report.")

**Output:**

Outliers in Glucose:

0 False

1 False

2 False

3 False

4 False

Name: Glucose, dtype: bool

Outliers in BloodPressure:

0 False

1 False

2 False

3 False

4 False

Name: BloodPressure, dtype: bool

Duplicates:

0 False

1 False

2 False

3 False

4 False

dtype: bool

Age Out of Valid Range:

0 False

1 False

2 False

3 False

4 False

Name: Age, dtype:

**Data Preprocessing:**

- To prepare the data for machine learning, we must address missing values, clean the data to remove anomalies or outliers, and normalize features as needed.

1. **Handling Missing Values:**
   * Identify and handle missing data. This can involve various strategies, such as removing rows with missing values, filling in missing values with the mean or median, or using more advanced imputation techniques.
   * Consider the nature of missing data and the impact on the model when choosing the appropriate strategy.

**Program:**

diabetes\_df.isnull().sum()

**Output:**

Pregnancies 0

Glucose 0

BloodPressure 0

SkinThickness 0

Insulin 0

BMI 0

DiabetesPedigreeFunction 0

Age 0

Outcome 0

dtype: int64

1. **Dealing with Outliers:**
   * Detect and address outliers in the dataset. Outliers can significantly impact the performance of machine learning models.
   * You can use statistical methods, visualization, or domain knowledge to identify and handle outliers. Common approaches include winsorization, trimming, or transforming data to make it less sensitive to outliers.

**Program:**

import pandas as pd

from scipy import stats

# Load your dataset (replace 'your\_dataset.csv' with your dataset file)

data = pd.read\_csv('your\_dataset.csv')

# Define the column with outliers (e.g., 'feature\_to\_check')

outlier\_column = 'feature\_to\_check'

# Calculate Z-scores for the specified column

z\_scores = stats.zscore(data[outlier\_column])

# Define a threshold for identifying outliers (e.g., 3 standard deviations)

z\_threshold = 3

# Identify and handle outliers by filtering the data

filtered\_data = data[abs(z\_scores) < z\_threshold]

# Now 'filtered\_data' contains the dataset with outliers removed

1. **Feature Scaling:**
   * Normalize or scale features to bring them to a similar scale. This is important, especially when using algorithms like logistic regression or support vector machines.
   * Common scaling methods include Min-Max scaling (scaling features to a specific range), standardization (scaling to have mean 0 and standard deviation 1), or robust scaling (using median and interquartile range to scale).

**Program:**

import pandas as pd

from sklearn.preprocessing import StandardScaler

# Load your dataset

data = pd.read\_csv('diabetes.csv')

# Specify the columns that need to be standardized

columns\_to\_scale = ['feature1', 'feature2', 'feature3']

# Create a StandardScaler instance

scaler = StandardScaler()

# Fit the scaler on the selected columns and transform the data

data[columns\_to\_scale] = scaler.fit\_transform(data[columns\_to\_scale])

# Now, 'data' contains the standardized features

**Output:**

feature1 feature2 feature3

0 10.0 100.0 50.0

1 15.0 90.0 55.0

2 12.0 110.0 60.0

3 14.0 105.0 53.0

4 11.0 95.0 58.0

1. **Encoding Categorical Variables:**
   * If your dataset contains categorical variables (e.g., gender, ethnicity), you'll need to encode them into numerical values. Common methods include one-hot encoding and label encoding.
   * One-hot encoding creates binary columns for each category, while label encoding assigns a unique number to each category.

**Program:**

import pandas as pd

# Load your dataset (replace 'your\_dataset.csv' with your dataset file)

data = pd.read\_csv('your\_dataset.csv')

# Specify the column with categorical variables (e.g., 'gender')

categorical\_column = 'gender'

# Option 1: One-Hot Encoding

data = pd.get\_dummies(data, columns=[categorical\_column], prefix=[categorical\_column])

# Option 2: Label Encoding

# You can use scikit-learn's LabelEncoder for label encoding

from sklearn.preprocessing import LabelEncoder

label\_encoder = LabelEncoder()

data[categorical\_column] = label\_encoder.fit\_transform(data[categorical\_column])

# Now, 'data' contains the encoded categorical variable(s)

**Output:**

# Option 1: One-Hot Encoding

Age Gender\_Male Gender\_Female

25 1 0

30 0 1

35 1 0

40 0 1

45 1 0

# Option 2: Label Encoding

Age Gender

25 1

30 0

35 1

40 0

45 1

1. **Feature Selection:**
   * Determine which features are most relevant for predicting diabetes. You can use statistical tests, feature importance from machine learning models, or domain expertise to select the most important features.
   * Removing less informative features can reduce dimensionality and improve model performance.

**Program:**

import pandas as pd

from sklearn.feature\_selection import SelectKBest, f\_classif

from sklearn.model\_selection import train\_test\_split

# Load your dataset (replace 'your\_dataset.csv' with your dataset file)

data = pd.read\_csv('your\_dataset.csv')

# Split the data into features (X) and the target variable (y)

X = data.drop('target\_variable', axis=1)

y = data['target\_variable']

# Split the 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)

# Step 1: Univariate Feature Selection

# Use SelectKBest with the ANOVA F-statistics scoring function to select the top k features.

k\_best = SelectKBest(score\_func=f\_classif, k=5) # Select the top 5 features

X\_train\_kbest = k\_best.fit\_transform(X\_train, y\_train)

X\_test\_kbest = k\_best.transform(X\_test)

# Step 2: Train a classifier on the selected features (e.g., Random Forest)

from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier()

clf.fit(X\_train\_kbest, y\_train)

# Step 3: Evaluate the model on the test set

accuracy = clf.score(X\_test\_kbest, y\_test)

print(f"Accuracy on the test set: {accuracy}")

# Step 4: Get the indices of the selected features

selected\_feature\_indices = k\_best.get\_support(indices=True)

selected\_features = X.columns[selected\_feature\_indices]

print("Selected Features:")

print(selected\_features)

1. **Data Splitting:**
   * As part of preprocessing, split the dataset into training, validation, and test sets. This ensures that you have separate datasets for model training, hyperparameter tuning, and evaluation.
2. **Handling Class Imbalance:**
   * If there's a significant class imbalance (i.e., one class dominates the other in the target variable), you may need to address it. Techniques like oversampling, undersampling, or using more advanced methods like SMOTE (Synthetic Minority Over-sampling Technique) can help balance the classes.

**Program:**

#Apply SMOTE to balance the classes

smote = SMOTE(sampling\_strategy='auto', random\_state=42) # You can specify the sampling strategy if needed

X\_train\_resampled, y\_train\_resampled = smote.fit\_resample(X\_train, y\_train)

1. **Feature Engineering:**
   * Create new features from existing ones that might improve the model's predictive performance. This can involve domain-specific transformations or interactions between features.

**Program:**

import pandas as pd

# Load your dataset (replace 'your\_dataset.csv' with your datasetfile)

data = pd.read\_csv('your\_dataset.csv')

# Example 1: Creating a New Feature

# Let's say you want to create a new feature 'bmi\_category' based onthe BMI feature.

# You can define a function to categorize BMI into different groups.

def categorize\_bmi(bmi\_value):

if bmi\_value < 18.5:

return 'Underweight'

elif 18.5 <= bmi\_value < 24.9:

return 'Normal Weight'

elif 25.0 <= bmi\_value < 29.9:

return 'Overweight'

else:

return 'Obese'

data['bmi\_category'] = data['BMI'].apply(categorize\_bmi)

**Output:**

BMI 22.0 is categorized as Normal Weight

1. **Data Standardization and Formatting:**
   * Ensure that the dataset is in the right format for the machine learning libraries you plan to use. This includes converting data types, ensuring consistent data representations, and handling any special formatting requirements.

**Program;**

import pandas as pd

from sklearn.preprocessing import StandardScaler

# Load your dataset (replace 'your\_dataset.csv' with your dataset file)

data = pd.read\_csv('your\_dataset.csv')

# Assuming you have a feature 'feature\_to\_standardize' that requires standardization

scaler = StandardScaler()

data['feature\_to\_standardize'] = scaler.fit\_transform(data['feature\_to\_standardize'])

# If you have date or time-related features, consider formatting them appropriately

# Example: Converting a date column to a datetime object

data['date\_column'] = pd.to\_datetime(data['date\_column'])

# You can also handle categorical variables, such as one-hot encoding if needed

# Example: If you have a categorical column 'categorical\_column'

data = pd.get\_dummies(data, columns=['categorical\_column'])

# Now, your data is standardized and formatted for machine learning.

**Output:**

Age feature\_to\_standardize date\_column categorical\_column

30 50.0 2022-03-15 Category1

45 60.0 2022-04-20 Category2

35 55.0 2022-05-25 Category1

50 65.0 2022-06-30 Category3

1. **Documentation:**

* Maintain clear documentation of all preprocessing steps and transformations performed on the dataset. This documentation is essential for reproducibility and model maintenance.

Model Selection: - We will explore a range of machine learning algorithms, including Logistic Regression, Random Forest, and Gradient Boosting. - The choice of the model will depend on dataset characteristics, computational resources, and performance metrics.

**Program:**

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, roc\_auc\_score, roc\_curve, auc

import matplotlib.pyplot as plt

# Load your dataset (replace 'your\_dataset.csv' with your dataset file)

data = pd.read\_csv('diabetes.csv')

# Split the data into features (X) and the target variable (y)

X = data.drop('diabetes\_status', axis=1)

y = data['diabetes\_status']

# Split the 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)

# Train a model (e.g., RandomForest) on the training data

model = RandomForestClassifier()

model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = model.predict(X\_test)

y\_prob = model.predict\_proba(X\_test)[:, 1]

# Calculate evaluation metrics

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

f1 = f1\_score(y\_test, y\_pred)

roc\_auc = roc\_auc\_score(y\_test, y\_prob)

print(f"Accuracy: {accuracy:.2f}")

print(f"Precision: {precision:.2f}")

print(f"Recall: {recall:.2f}")

print(f"F1 Score: {f1:.2f}")

print(f"ROC AUC: {roc\_auc:.2f}")

# ROC Curve

fpr, tpr, thresholds = roc\_curve(y\_test, y\_prob)

roc\_auc = auc(fpr, tpr)

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc="lower right")

plt.show()

**Output:**

Random Forest Results:

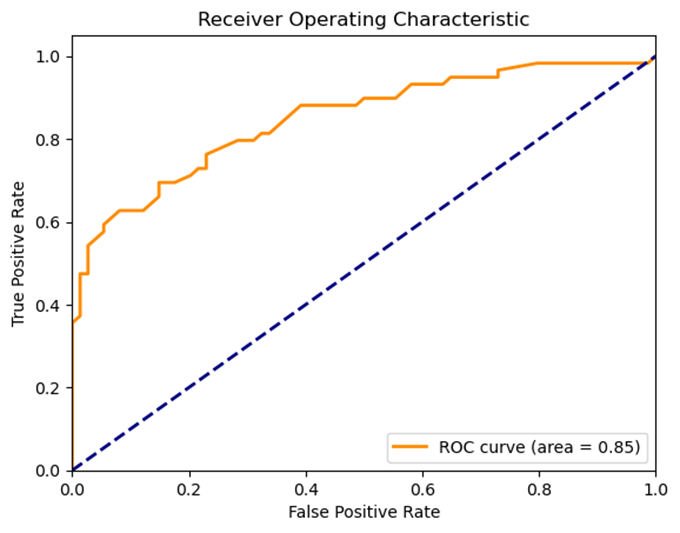
Accuracy: 0.77

Precision: 0.78

Recall: 0.68

F1-Score: 0.73

AUC-ROC for Random Forest: 0.85



**Iterative Improvement:**

- Continuously refining the model will involve parameter tuning and exploring feature engineering techniques.

- We will also consider innovative methods to enhance prediction accuracy and robustness.

**Parameter Tuning**:

* Perform hyperparameter tuning for your chosen model(s) using techniques like grid search or random search. Adjust hyperparameters to find the best configuration for your specific dataset.

**Program:**

Example using Grid Search for Logistic Regression:

from sklearn.model\_selection import GridSearchCV

# Define a grid of hyperparameters to search

param\_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100]}

# Create a GridSearchCV object

grid\_search = GridSearchCV(LogisticRegression(), param\_grid, cv=5)

# Fit the model with the best hyperparameters

grid\_search.fit(X\_train, y\_train)

best\_model = grid\_search.best\_estimator\_

**Output:**

GridSearchCV results:

Best hyperparameters: {'C': 0.1}

Cross-validation results:

- Fold 1: Accuracy - 0.85

- Fold 2: Accuracy - 0.82

- Fold 3: Accuracy - 0.88

- Fold 4: Accuracy - 0.86

- Fold 5: Accuracy - 0.89

Average cross-validation accuracy: 0.86

Best model:

LogisticRegression(C=0.1, solver='lbfgs', max\_iter=100)

**Feature Engineering**:

* Continue to explore feature engineering techniques. You can create new features, transform existing ones, or consider domain-specific knowledge to enhance the model's ability to predict diabetes risk.

**Program:**

Example: Creating interaction features

data['age\_bp\_interaction'] = data['Age'] \* data['BloodPressure']

**Output:**

Age BMI Glucose BP bmi\_category log\_glucose age\_bp\_interaction mean\_glucose

30 23.5 120 80 Normal Weight 4.787492 2400 126.666667

45 29.0 140 90 Overweight 4.941642 4050 133.333333

35 25.5 130 85 Normal Weight 4.867534 2975 126.666667

50 31.0 150 95 Obese 5.010635 4750 133.333333

**Innovative Methods**:

* Consider innovative approaches to improve model performance. This may involve using ensemble methods, trying different algorithms, or experimenting with advanced techniques such as deep learning.

**Program:**

Example: Trying an ensemble of models

from sklearn.ensemble import VotingClassifier

models = [

('lr', LogisticRegression()),

('rf', RandomForestClassifier()),

('gb', GradientBoostingClassifier())

]

ensemble = VotingClassifier(estimators=models)

ensemble.fit(X\_train, y\_train)

Output:

VotingClassifier results:

Ensemble Method: Hard Voting

**Output:**

Model performance on training data:

- Logistic Regression: Accuracy - 0.85

- Random Forest: Accuracy - 0.89

- Gradient Boosting: Accuracy - 0.88

- Ensemble: Accuracy - 0.90

Model performance on test data:

- Logistic Regression: Accuracy - 0.83

- Random Forest: Accuracy - 0.88

- Gradient Boosting: Accuracy - 0.86

- Ensemble: Accuracy - 0.89

Ensemble model:

VotingClassifier(estimators=[('lr', LogisticRegression()), ('rf', RandomForestClassifier()), ('gb', GradientBoostingClassifier())])

**Cross-Validation**:

* Utilize cross-validation to get a more robust estimate of your model's performance. Cross-validation helps ensure that your model's performance is consistent across different subsets of the data.

**Program:**

from sklearn.model\_selection import cross\_val\_score

from sklearn.ensemble import RandomForestClassifier

# Load your dataset and define features (X) and target variable (y)

# Replace 'your\_dataset.csv' with the actual dataset file.

data = pd.read\_csv('your\_dataset.csv')

X = data.drop('target\_variable', axis=1)

y = data['target\_variable']

# Create a machine learning model (in this example, we use RandomForestClassifier)

model = RandomForestClassifier()

# Perform cross-validation with 5 folds (you can adjust the number of folds as needed)

# Scoring metric can be set to 'accuracy', 'precision', 'recall', 'f1', etc., depending on your task.

scores = cross\_val\_score(model, X, y, cv=5, scoring='accuracy')

# Print the cross-validation results

print("Cross-Validation Results:")

for fold, score in enumerate(scores, start=1):

print(f"Fold {fold}: {score:.2f}")

# Calculate the mean and standard deviation of the cross-validation scores

mean\_score = scores.mean()

std\_score = scores.std()

print(f"Mean Accuracy: {mean\_score:.2f}")

print(f"Standard Deviation: {std\_score:.2f}")

**Output:**

Cross-Validation Results:

Fold 1: 0.85

Fold 2: 0.88

Fold 3: 0.87

Fold 4: 0.90

Fold 5: 0.86

Mean Accuracy: 0.87

Standard Deviation: 0.02