Health AI : intelligent Healthcare assistant using IBM  granite

Model Selection and Architecture:

Importing Libraries;

import pandas as pd

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

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn.linear\_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier

Loading Data

# Load your dataset here

from sklearn.datasets import load\_iris

iris = load\_iris()

df = pd.DataFrame(data=iris.data, columns=iris.feature\_names)

df['target'] = iris.target

Splitting Data

X = df.drop('target', axis=1)

y = df['target']

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

Model Selection

models = {

'Logistic Regression': LogisticRegression(),

'Decision Tree Classifier': DecisionTreeClassifier(),

'Random Forest Classifier': RandomForestClassifier(),

'Support Vector Machine': SVC(),

'K-Nearest Neighbors': KNeighborsClassifier()

}

for name, model in models.items():

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

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

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

print(f'Model: {name}, Accuracy: {accuracy:.3f}')

print(classification\_report(y\_test, y\_pred))

Model Architecture

Let's take the example of a neural network architecture using Keras:

from keras.models import Sequential

from keras.layers import Dense

model = Sequential()

model.add(Dense(64, activation='relu', input\_shape=(4,)))

model.add(Dense(32, activation='relu'))

model.add(Dense(3, activation='softmax'))

model.compile(loss='sparse\_categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(X\_test, y\_test))

Choosing the Best Model

Based on the performance metrics (accuracy, precision, recall, F1-score), you can choose the best model for your problem.

Hyperparameter Tuning

You can use techniques like GridSearchCV or RandomizedSearchCV to tune hyperparameters for your chosen model.

from sklearn.model\_selection import GridSearchCV

param\_grid = {

'C': [0.1, 1, 10],

'kernel': ['linear', 'rbf', 'poly']

}

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

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

print(f'Best Parameters: {grid\_search.best\_params\_}')

print(f'Best Score: {grid\_search.best\_score\_}')

\*Research And Select The Appropriate AI Model

## Step 1: Define the Problem

Identify the type of problem you're trying to solve, such as classification, regression, or clustering. This will help you narrow down the choice of models.

## Step 2: Choose a Model Type

Based on your problem, select a suitable model type. Some popular options include:

- Supervised Learning Models:

- Logistic Regression: for binary classification problems

- Decision Trees: for classification and regression problems

- Random Forest: for ensemble learning and improving model accuracy

- Unsupervised Learning Models:

- K-Means Clustering: for grouping similar data points

- Principal Component Analysis (PCA): for dimensionality reduction

- Deep Learning Models:

- Convolutional Neural Networks (CNNs): for image classification and object detection

- Recurrent Neural Networks (RNNs): for sequence data and natural language processing

## Step 3: Select a Python Library

Choose a suitable Python library for your model, such as:

- Scikit-learn: for traditional machine learning models

- TensorFlow or PyTorch: for deep learning models

## Step 4: Implement and Evaluate the Model

Implement your chosen model using Python and evaluate its performance using metrics such as accuracy, precision, and recall.

Here's an example code snippet using Scikit-learn's Logistic Regression model:

from sklearn.datasets import load\_iris

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

# Load the iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split the data into training and testing sets

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

# Train a logistic regression model

model = LogisticRegression(max\_iter=1000)

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

# Make predictions and evaluate the model

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

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

print(f"Model Accuracy: {accuracy:.3f}")

Some popular resources for learning more about AI models in Python include ¹ ² ³:

- Toxigon Blog: provides step-by-step guides and tutorials on building AI models with Python

- CodeConductor: offers comprehensive guides on building generative AI models using Python

- Medium: features articles and tutorials on building AI models with Python, including beginner-friendly guides and advanced topics

By following these steps and exploring these resources, you can research and select an appropriate AI model using Python for your specific needs.

\*Core Functionalities Development

code:

calculator.py

def add(a, b):

"""Return the sum of a and b"""

return a + b

def subtract(a, b):

"""Return the difference of a and b"""

return a - b

def multiply(a, b):

"""Return the product of a and b"""

return a \* b

def divide(a, b):

"""Return the quotient of a and b"""

if b == 0:

raise ZeroDivisionError("Cannot divide by zero")

return a / b

You can import and use this module in another script:

# main.py

import calculator

result = calculator.add(5, 7)

print(result) # Output: 12

Develope core functionalities:

->Code:

Library Management System

class Book:

def \_init\_(self, title, author, status="Available"):

self.title = title

self.author = author

self.status = status

class Library:

def \_init\_(self):

self.books = []

def add\_book(self, title, author):

new\_book = Book(title, author)

self.books.append(new\_book)

print(f"Book '{title}' by {author} added successfully.")

def remove\_book(self, title):

for book in self.books:

if book.title.lower() == title.lower():

self.books.remove(book)

print(f"Book '{title}' removed successfully.")

return

print(f"Book '{title}' not found.")

def borrow\_book(self, title):

for book in self.books:

if book.title.lower() == title.lower():

if book.status == "Available":

book.status = "Borrowed"

print(f"Book '{title}' borrowed successfully.")

else:

print(f"Book '{title}' is currently borrowed.")

return

print(f"Book '{title}' not found.")

def return\_book(self, title):

for book in self.books:

if book.title.lower() == title.lower():

if book.status == "Borrowed":

book.status = "Available"

print(f"Book '{title}' returned successfully.")

else:

print(f"Book '{title}' is already available.")

return

print(f"Book '{title}' not found.")

def list\_books(self):

if not self.books:

print("No books in the library.")

else:

for book in self.books:

print(f"Title: {book.title}, Author: {book.author}, Status: {book.status}")

# Create a library

library = Library()

# Add books

library.add\_book("To Kill a Mockingbird", "Harper Lee")

library.add\_book("1984", "George Orwell")

# List books

library.list\_books()

# Borrow a book

library.borrow\_book("To Kill a Mockingbird")

# Return a book

library.return\_book("To Kill a Mockingbird")

# Remove a book

library.remove\_book("1984")

# List books again

library.list\_books()

App.Py Development:

->code

App.py Code

from flask import Flask, render\_template, request

app = Flask(\_name\_)

# Home page

@app.route("/")

def home():

return render\_template("home.html")

# About page

@app.route("/about")

def about():

return render\_template("about.html")

# Contact page

@app.route("/contact", methods=["GET", "POST"])

def contact():

if request.method == "POST":

name = request.form["name"]

email = request.form["email"]

message = request.form["message"]

# Handle form submission (e.g., send email or save to database)

print(f"Name: {name}, Email: {email}, Message: {message}")

return "Thank you for contacting us!"

return render\_template("contact.html")

if \_name\_ == "\_main\_":

app.run(debug=True)

Main application logic:

->code

Banking System

class BankAccount:

def \_init\_(self, account\_number, balance=0):

self.account\_number = account\_number

self.balance = balance

def deposit(self, amount):

self.balance += amount

return self.balance

def withdraw(self, amount):

if amount > self.balance:

raise ValueError("Insufficient funds")

self.balance -= amount

return self.balance

def get\_balance(self):

return self.balance

class BankingSystem:

def \_init\_(self):

self.accounts = {}

def create\_account(self, account\_number, initial\_balance=0):

if account\_number in self.accounts:

raise ValueError("Account already exists")

self.accounts[account\_number] = BankAccount(account\_number, initial\_balance)

return self.accounts[account\_number]

def get\_account(self, account\_number):

return self.accounts.get(account\_number)

def deposit(self, account\_number, amount):

account = self.get\_account(account\_number)

if account:

return account.deposit(amount)

else:

raise ValueError("Account not found")

def withdraw(self, account\_number, amount):

account = self.get\_account(account\_number)

if account:

return account.withdraw(amount)

else:

raise ValueError("Account not found")

def get\_balance(self, account\_number):

account = self.get\_account(account\_number)

if account:

return account.get\_balance()

else:

raise ValueError("Account not found")

def main():

banking\_system = BankingSystem()

while True:

print("1. Create account")

print("2. Deposit")

print("3. Withdraw")

print("4. Check balance")

print("5. Exit")

choice = input("Enter your choice: ")

if choice == "1":

account\_number = input("Enter account number: ")

initial\_balance = float(input("Enter initial balance: "))

account = banking\_system.create\_account(account\_number, initial\_balance)

print(f"Account created with account number {account\_number} and balance {initial\_balance}")

elif choice == "2":

account\_number = input("Enter account number:

Design and develop the user interface:

->code

import tkinter as tk

root = tk.Tk()

root.title("My First Python UI App")

label = tk.Label(root, text="Hello, World!")

label.pack()

def button\_click():

label.config(text="Button clicked!")

button = tk.Button(root, text="Click me", command=button\_click)

button.pack()

root.mainloop()

Design Considerations:

Deployment:

->code

using Python

import tkinter as tk

class Application(tk.Frame):

def \_init\_(self, master=None):

super().\_init\_(master)

self.master = master

self.pack()

self.create\_widgets()

def create\_widgets(self):

self.hi\_there = tk.Button(self)

self.hi\_there["text"] = "Hello World\n(click me)"

self.hi\_there["command"] = self.say\_hi

self.hi\_there.pack(side="top")

self.quit = tk.Button(self, text="QUIT", fg="red",

command=self.master.destroy)

self.quit.pack(side="bottom")

def say\_hi(self):

print("hi there, everyone!")

root = tk.Tk()

app = Application(master=root)

app.mainloop()

Prepare for deployment:

🡪code

import os

import logging

# Configure logging

logging.basicConfig(level=logging.INFO)

# Use environment variables

api\_key = os.environ.get('API\_KEY')

def main():

logging.info('Application started')

# Use the API key

print(f'API Key: {api\_key}')

if \_name\_ == '\_main\_':

main()