Your project, the WEAR system, is an innovative application of technology in fashion and personal organization. It incorporates machine learning, sensor data, and user preferences to recommend outfits. Let's break down the components and structure of your codebase for this project. We will focus on the machine learning and Python code part, as requested.

**1. Project Structure Overview**

The project can be divided into several key components:

* **Data Collection and Preprocessing**
* **Machine Learning Model**
* **API Integration**
* **User Interface and Output**
* **Database Management**

**2. Data Collection and Preprocessing**

This component deals with gathering and preparing data for the machine learning model.

* **File:** **sensor\_data\_handler.py**
  + **Purpose:** To interface with sensors and APIs to collect data on temperature, humidity, and weather conditions.
  + **Functionality:** Parse data from the DHT22 sensor and APIs like OpenWeatherMap and Google Calendar.
  + **Output:** Structured data in a format suitable for the ML model.
* **File:** **data\_preprocessing.py**
  + **Purpose:** To preprocess the data for the machine learning model.
  + **Functionality:** Cleansing, normalizing, and transforming data. Dealing with missing values, encoding categorical data, etc.
  + **Output:** Cleaned and preprocessed data ready for training and testing.

**3. Machine Learning Model**

This is the core of your application, where the outfit recommendation logic is implemented.

* **File:** **model\_training.py**
  + **Purpose:** To train the machine learning model.
  + **Functionality:** Use algorithms like KNN to train the model on attributes such as Type, Color, Fabric, etc.
  + **Output:** A trained model that can make predictions.
* **File:** **model\_evaluation.py**
  + **Purpose:** To evaluate the trained model.
  + **Functionality:** Apply various metrics to evaluate model performance.
  + **Output:** Evaluation reports (accuracy, precision, recall, etc.).
* **File:** **recommendation\_engine.py**
  + **Purpose:** To generate outfit recommendations.
  + **Functionality:** Use the trained model to recommend outfits based on current conditions and user preferences.
  + **Output:** Recommended outfits, which will be displayed on the Arduino screen and read aloud via the text-to-speech module.

**4. API Integration**

This section will handle the integration with external APIs like OpenWeatherMap and Google Calendar.

* **File:** **api\_integration.py**
  + **Purpose:** To fetch data from external APIs.
  + **Functionality:** API calls to fetch weather data and calendar events.
  + **Output:** Data that will be used for outfit recommendations.

**5. User Interface and Output**

This component will handle the interaction with the user and the presentation of the recommended outfits.

* **File:** **user\_interface.py**
  + **Purpose:** To manage the interaction between the user and the system.
  + **Functionality:** Display recommendations on the screen and through the text-to-speech module.
  + **Output:** User-friendly interface for displaying recommendations.

**6. Database Management**

This part will deal with the storage and retrieval of user clothing data.

* **File:** **database\_manager.py**
  + **Purpose:** To manage interactions with the MySQL database.
  + **Functionality:** CRUD (Create, Read, Update, Delete) operations on the clothing database.
  + **Output:** A well-managed database storing all clothing items and attributes.

**7. Utilities and Helpers**

You might also want to have a separate directory for utility functions and shared resources.

* **Directory:** **utils/**
  + **Purpose:** To store utility functions and shared resources.
  + **Functionality:** Functions for logging, error handling, data conversions, etc.
  + **Common Files:** **logger.py**, **exceptions.py**, **helpers.py**, etc.

**8. Color Matching**

* **File:** **color\_matching.py**
  + **Purpose:** To handle the logic for color matching based on your matrix.
  + **Functionality:** Apply color matching algorithms to recommend aesthetically pleasing outfit combinations.
  + **Output:** Integration into the recommendation engine to enhance outfit suggestions.

**Conclusion**

Organizing your code into modular, purpose-driven components not only makes the development process more manageable but also ensures maintainability and scalability. Each file or module should have a clear responsibility, making it easier to debug, test, and collaborate on the project. Remember to document each part of the code thoroughly. This structure is a guideline and can be adapted as per the specific needs of your project.

Code we didn’t use since bad ui:  
import tkinter as tk

from tkinter import simpledialog, Toplevel, messagebox

from tkcalendar import Calendar

from datetime import datetime

import csv

import os

# Specify the CSV file path

csv\_file\_path = r"C:\Users\User\Desktop\Laucourses\this semester\FYP2\fromFYP1\fypCode\data1\events.csv"

# Ensure the directory exists

os.makedirs(os.path.dirname(csv\_file\_path), exist\_ok=True)

# Utility function to read events from CSV

def read\_events():

    if os.path.exists(csv\_file\_path):

        with open(csv\_file\_path, 'r', newline='', encoding='utf-8') as file:

            return list(csv.reader(file))

    return []

# Utility function to write events to CSV

def write\_events(events):

    with open(csv\_file\_path, 'w', newline='', encoding='utf-8') as file:

        writer = csv.writer(file)

        writer.writerows(events)

def sort\_events():

    events = read\_events()

    events.sort(key=lambda row: datetime(int(row[0]), int(row[1]), int(row[2]), int(row[3]), int(row[4])))

    write\_events(events)

def event\_popup(action):

    def execute\_action(event\_index):

        events = read\_events()

        if action == "edit":

            open\_event\_editor(events[event\_index], event\_index)

        elif action == "delete":

            del events[event\_index]

            write\_events(events)

            load\_events()

        popup.destroy()

    popup = Toplevel(root)

    popup.title(f"{action.capitalize()} Event")

    popup.geometry("600x400")  # Double the default size for popups

    for index, row in enumerate(read\_events()):

        event\_date = f"{row[0]}-{row[1]}-{row[2]} {row[3]}:{row[4]}"

        event\_button = tk.Button(popup, text=f"{event\_date}: {row[5]}", command=lambda index=index: execute\_action(index))

        event\_button.pack(fill=tk.X, padx=10, pady=5)

def add\_event():

    open\_event\_editor()

def open\_event\_editor(event\_data=None, event\_index=None):

    def save\_event():

        try:

            selected\_date = cal.selection\_get()

            year, month, day = selected\_date.year, selected\_date.month, selected\_date.day

            hour = int(hour\_entry.get())

            minute = int(minute\_entry.get())

            event\_description = event\_entry.get()

            event\_datetime = datetime(year, month, day, hour, minute)

            if event\_datetime < datetime.now():

                messagebox.showerror("Error", "Event date and time must be in the future.")

                return

            new\_event = [year, month, day, hour, minute, event\_description]

            if event\_index is not None:  # Edit existing event

                events = read\_events()

                events[event\_index] = new\_event

                write\_events(events)

            else:  # Add new event

                write\_events(read\_events() + [new\_event])

            top.destroy()

            load\_events()

        except ValueError:

            messagebox.showerror("Error", "Invalid date, time, or event description.")

    top = Toplevel(root)

    button\_text = "Edit Event" if event\_data else "Save Event"

    top.title(button\_text)

    top.geometry("800x600")

    cal = Calendar(top, selectmode='day', year=datetime.now().year, month=datetime.now().month, day=datetime.now().day)

    cal.pack(pady=20)

    time\_frame = tk.Frame(top)

    time\_frame.pack(pady=5)

    hour\_entry = tk.Entry(time\_frame, width=5)

    hour\_entry.pack(side=tk.LEFT)

    tk.Label(time\_frame, text=":").pack(side=tk.LEFT)

    minute\_entry = tk.Entry(time\_frame, width=5)

    minute\_entry.pack(side=tk.LEFT)

    event\_label = tk.Label(top, text="Event Description:")

    event\_label.pack()

    event\_entry = tk.Entry(top, width=20)

    event\_entry.pack()

    if event\_data:  # Pre-fill form for editing

        year, month, day, hour, minute, event\_description = event\_data

        cal.set\_date(datetime(year, month, day))

        hour\_entry.insert(0, str(hour))

        minute\_entry.insert(0, str(minute))

        event\_entry.insert(0, event\_description)

    save\_button = tk.Button(top, text=button\_text, command=save\_event)

    save\_button.pack(pady=20)

def load\_events():

    sort\_events()

    text\_area.config(state=tk.NORMAL)  # Enable the text area to modify its content

    text\_area.delete('1.0', tk.END)  # Clear the text area

    events = read\_events()

    for event in events:

        year, month, day, hour, minute, event\_description = event

        text\_area.insert(tk.END, f"Date: {year}-{month}-{day} {hour}:{minute}, Event: {event\_description}\n")

    text\_area.config(state=tk.DISABLED)  # Disable the text area to prevent user edits

root = tk.Tk()

root.title("Simple Calendar App")

root.geometry("960x600")  # 1.5 times bigger than the default size

main\_frame = tk.Frame(root)

main\_frame.pack(pady=20)

text\_area = tk.Text(main\_frame, width=120, height=30)  # Adjusted for the bigger main window

text\_area.pack(pady=20)

text\_area.config(state=tk.DISABLED)

add\_button = tk.Button(main\_frame, text="Add Event", command=add\_event)

add\_button.pack(side=tk.LEFT, padx=10, pady=10)

edit\_button = tk.Button(main\_frame, text="Edit Event", command=lambda: event\_popup("edit"))

edit\_button.pack(side=tk.LEFT, padx=10, pady=10)

delete\_button = tk.Button(main\_frame, text="Delete Event", command=lambda: event\_popup("delete"))

delete\_button.pack(side=tk.LEFT, padx=10, pady=10)

sort\_events()

load\_events()

root.mainloop()

google calendar:  
import requests

import msal

import datetime

def authenticate\_client(client\_id, client\_secret, tenant\_id):

    authority = f"https://login.microsoftonline.com/{tenant\_id}"

    scope = ["https://graph.microsoft.com/.default"]

    app = msal.ConfidentialClientApplication(client\_id, authority=authority, client\_credential=client\_secret)

    token\_response = app.acquire\_token\_for\_client(scopes=scope)

    if "access\_token" in token\_response:

        return token\_response["access\_token"]

    else:

        print("Error acquiring token:")

        print(token\_response.get("error"))

        print(token\_response.get("error\_description"))

        return None

def get\_calendar\_events(access\_token):

    endpoint = "https://graph.microsoft.com/v1.0/me/events"

    headers = {

        "Authorization": f"Bearer {access\_token}",

        "Content-Type": "application/json"

    }

    params = {

        "startDateTime": datetime.datetime.utcnow().isoformat(),

        "endDateTime": (datetime.datetime.utcnow() + datetime.timedelta(days=7)).isoformat(),

        "$select": "subject,start,end",

        "$orderby": "start/dateTime",

        "$top": 10

    }

    response = requests.get(endpoint, headers=headers, params=params)

    events = response.json().get("value", [])

    if events:

        for event in events:

            print(f"Subject: {event['subject']}")

            print(f"Start: {event['start']['dateTime']}")

            print(f"End: {event['end']['dateTime']}\n")

    else:

        print("No upcoming events found.")

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

    # These values must be replaced with your actual client ID, client secret, and tenant ID

    CLIENT\_ID = "YOUR\_CLIENT\_ID"

    CLIENT\_SECRET = "YOUR\_CLIENT\_SECRET"

    TENANT\_ID = "YOUR\_TENANT\_ID"

    access\_token = authenticate\_client(CLIENT\_ID, CLIENT\_SECRET, TENANT\_ID)

    if access\_token:

        get\_calendar\_events(access\_token)

From the files you've provided, here are the primary functions, their inputs, and outputs:

**calendarEndpoint.py**

Functions:

* **add\_event(year, month, day, hour, minute, description)**  
  **Inputs:** year (int), month (int), day (int), hour (int), minute (int), description (str)  
  **Outputs:** None. Adds a new event to the CSV file and sorts all events chronologically.
* **delete\_event(year, month, day, hour, minute)**  
  **Inputs:** year (int), month (int), day (int), hour (int), minute (int)  
  **Outputs:** None. Deletes an event from the CSV file matching the provided date and time.

**color\_matching.py**

Functions:

* **color\_matches(top, bottom)**  
  **Inputs:** top (str), bottom (str)  
  **Outputs:** Boolean. Returns **True** if the bottom color matches with the top based on a predefined matching matrix, otherwise **False**.

**data\_preprocessing.py**

Functions:

* **preprocess\_data(data, target\_column)**  
  **Inputs:** data (DataFrame), target\_column (str)  
  **Outputs:** Tuple (preprocessed\_features\_df, target). Returns preprocessed features as a DataFrame and the target variable.

**database\_manager.py**

Functions:

* **add\_data(file\_path, data)**  
  **Inputs:** file\_path (str), data (list)  
  **Outputs:** None. Adds a new row to the dataset with an auto-incremented ID.
* **remove\_data(file\_path, id)**  
  **Inputs:** file\_path (str), id (int)  
  **Outputs:** None. Removes a row from the dataset based on the provided ID.

**model\_training\_and\_evaluation.py**

Functions:

* **train\_model(X\_train, y\_train)**  
  **Inputs:** X\_train (array-like), y\_train (array-like)  
  **Outputs:** Trained KNN model. Trains a KNeighborsClassifier model on the training data.
* **evaluate\_model(model, X\_test, y\_test, X, y)**  
  **Inputs:** model (sklearn estimator), X\_test (array-like), y\_test (array-like), X (array-like), y (array-like)  
  **Outputs:** None. Prints out the model's accuracy, classification report, confusion matrix, ROC-AUC score (if binary classification), and cross-validation accuracy scores.

**OpenAI\_api.py**

Functions:

* **classify\_event\_with\_gpt3(event\_text)**  
  **Inputs:** event\_text (str)  
  **Outputs:** Label (str). Returns a classification label (**'Sports Activity'**, **'Formal Event'**, or **'Casual Outing'**) for the provided event text using OpenAI's GPT-3.

**openWeather\_api.py**

Functions:

* **get\_weather\_data(coord, api\_key)**  
  **Inputs:** coord (dict with 'lat' and 'lon'), api\_key (str)  
  **Outputs:** None. Prints the weather data including main weather, description, temperature, pressure, and humidity for the given coordinates.

These function summaries provide a clear understanding of each file's main functionalities, inputs, and outputs, which can be used for further implementation or integration tasks.