```python
from flask import Flask, render_template
import pandas as pd
import re
import xgboost as xgb
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import os

app = Flask(__name__)
```

Iniciamos nuestro **algoritmo** haciendo las importaciones de las librerías que vamos a utilizar, importamos la librería de **flask** haciendo énfasis que vamos a utilizar una aplicación tipo **Flask** y utilizamos **render_templates** para renderizar una vista.

Importamos **Pandas** para la lectura de archivos es este caso vamos a leer archivos CSV

Importamos **re** para trabajar con expresiones regulares y manejar cadenas de texto si así lo necesitamos.

Impartamos nuestro modelo **xgboost** y le damos un alisas en este caso **xgb**

Importamos la librería **sklearn.model_selection** pero vamos a utilizar la función **train_test_split** esto nos ayuda a dividir los datos en conjuntos y entrena nuestro modelo, esta división de datos es esencial para el aprendizaje automático.

Importamos nuestra librería **matplotlib.pyplot** que simplemente función para Proporciona una forma fácil de generar gráficos de alta calidad

Por último, importamos nuestra librería **os** la cual nos ayudara a interactuar con el sistema, manejar archivos, carpetas y demás, como en este caso dependemos del sistema operativo, entonces tendremos que utilizarla.

```python
@app.route('/')
def index():
    # Cargar datos
    fileHistoricOne = pd.read_csv('load/Report Historic - Ene-Feb-Mar-2023.csv')
    fileHistoricTwo = pd.read_csv('load/Report Historic - Abr-May-Jun 2023.csv')
    fileHistoricThre = pd.read_csv('load/Report Historic - Jul-Ago-Sep 2023.csv')
    fileHistoricFour = pd.read_csv('load/Report Historic - Oct-Nov-Dic 2023.csv')
    fileHistoricFive = pd.read_csv('load/Report Historic - Ene-Feb-2024.csv')

    # Concatenar los dos DataFrames
    fileHistoric = pd.concat([fileHistoricOne, fileHistoricTwo, fileHistoricThre, fileHistoricFour, fileHistoricFive], ignore_index=True)

    # Eliminar columnas innecesarias
    columns_to_drop = ['CODIGO SIC', 'MEDIDOR', 'YEARX', 'MONX', 'DAYX', 'HOURX']
    fileHistoric = fileHistoric.drop(columns=columns_to_drop)

    # Convertir la columna DATEX1 a tipo fecha (date)
    fileHistoric['DATEX1'] = pd.to_datetime(fileHistoric['DATEX1'], format='%d/%m/%Y').dt.date

    # Convertir la columna CLIENTE a tipo categoría
    fileHistoric['CLIENTE'] = fileHistoric['CLIENTE'].astype('category')
    fileHistoric['CLIENTE_CODE'] = fileHistoric['CLIENTE'].cat.codes + 1

    # Convertir la columna DATETIME a tipo datetime
    fileHistoric['DATETIME'] = pd.to_datetime(fileHistoric['DATETIME'], format='%m/%d/%Y %H:%M', errors='coerce')
```

Creamos una ruta de acceso principal o bien se conoce como ruta raíz (/), definimos nutra función Índex y comenzamos con la carga de cada uno de nuestros archivos, estos los guardamos en diferentes variables. Se puede evidenciar que son 5 archivos, pero utilizamos la función **concat** y pasamos cada una de nuestras variables para crear una sola variable con todos los archivos concatenados. Esto nos va generar la variable **fileHistoric.**

Creamos un arreglo con las columnas que no vamos a utilizar o ente caso son columnas innecesarias para nuestro análisis, y utilizamos el parámetro **drop** para eliminarlas de nuestro dataframe.

	DATEX1	CLIENTE	VARIABLE	DATETIME	USAGE_DATA
0	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kVarhD	2023-01-01 00:00:00	3.20
2	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kWhD	2023-01-01 00:00:00	19.84
4	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kVarhD	2023-01-01 00:15:00	3.52
6	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kWhD	2023-01-01 00:15:00	19.20
8	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kVarhD	2023-01-01 00:30:00	3.84
10	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kWhD	2023-01-01 00:30:00	19.84
12	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kVarhD	2023-01-01 00:45:00	3.20
14	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kWhD	2023-01-01 00:45:00	19.84
16	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kVarhD	2023-01-01 01:00:00	3.52
18	2023-01-01	INDUSTRIA DE ELECTRODOMESTICOS	kWhD	2023-01-01 01:00:00	19.84

Luego procedemos a darle formato a nuestros datos en este caso deben quedar de la siguiente manera.

DATEX1 = Variable Fecha

CLIENTE = Variable Categorica

DATETIME = Variable de Fecha Tiempo

```
# Filtrar filas para mantener solo las variables 'kVarhD' y 'kWhD'
fileHistoric = fileHistoric[fileHistoric['VARIABLE'].isin(['kVarhD', 'kWhD'])]

# Función para limpiar y convertir a float
def clean_and_convert(value):
    cleaned_value = re.sub(r'[^\d.]', '', str(value))
    try:
        return float(cleaned_value)
    except ValueError:
        return None

fileHistoric['USAGE_DATA'] = fileHistoric['USAGE_DATA'].apply(clean_and_convert)

# Convertir la columna VARIABLE a tipo categoría
fileHistoric['VARIABLE'] = fileHistoric['VARIABLE'].astype('category')
fileHistoric['VARIABLE_CODE'] = fileHistoric['VARIABLE'].cat.codes + 1
```

Segmentamos las variables a utilizar es decir que solo dejamos la variable **KVarhD** y la variable **KWhD.**

Declaramos la función **clean_and_convert** que como su nombre lo indica va a limpiar el dato y lo va convertir en **FLOAT.** Esta toma el valor, quita puntos y caracteres extraños si los tiene si no entonces no retorna nada, pero si retorna algo es el valor limpio y formateado.

USAGE_DATA = Variable limpia y queda FLOAT

Por último, tomamos las variables de energía activa y reactiva (**KVarhD, KWhD**) y las categorizamos con un numero para poder obtener una mejor lectura.

```
# Entrenamiento de modelos
models = {}
for client in fileHistoric['CLIENTE'].unique():
    client_data = fileHistoric[fileHistoric['CLIENTE'] == client]
    X = client_data.drop(columns=['USAGE_DATA', 'DATEX1', 'DATETIME', 'CLIENTE', 'CLIENTE_CODE'])
    y = client_data['USAGE_DATA']

    # Convertir la columna 'VARIABLE' a tipo numérico (por ejemplo, categoría)
    X['VARIABLE'] = X['VARIABLE'].astype('category').cat.codes

    # Dividir el conjunto de datos en entrenamiento y prueba
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Crear DMatrix de XGBoost, habilitando el manejo de variables categóricas
    dtrain = xgb.DMatrix(X_train, label=y_train, enable_categorical=True)
    dtest = xgb.DMatrix(X_test, label=y_test, enable_categorical=True)

    # Definir los parámetros del modelo
    params = {
        'objective': 'reg:squarederror',
        'max_depth': 4,
        'eta': 0.1,
        'eval_metric': 'rmse'
    }

    # Entrenar el modelo
    bst = xgb.train(params, dtrain, num_boost_round=100)

    # Guardar el modelo en el diccionario
    models[client] = bst
```

Esta es una de las partes más importantes porque es la aplicación del modelo **XGBOOST,** creamos un arreglo llamado models con la intención de generar un resultado por cada uno de los clientes y poder devolver cada uno de ellos.

Creamos un ciclo FOR para iterar en cada uno de nuestros clientes y utilizamos la función unique() para decirle que ese cliente es único.

Declaramos X y también declaramos Y:

X = a cada una de nuestras variables en este caso variable activa y reactiva (**KVarhD, KWhD**)

Y = Es el consumo por cada uno de los registros.

X	Y
kVarhD	3.20
kWhD	19.84
kVarhD	3.21
kWhD	19.83

Estos son los datos que estaría declarando el algoritmo.

Procedemos a hacer la división de datos y guardándolos en estas 4 variables que van hacer importantes, a continuación, se explicara cada una de las funciones utilizadas en el algoritmo y su explicación.

X_train, X_test, y_train, y_test

Se utiliza la función ya mencionada **train_test_split** para agrupar los datos y los parámetros que se deben enviar son nuestra variable X y nuestra variable Y.

Aprte de eso pasamos 2 parametros importante slo cuales son

test_size=0.2, random_state=42

test_size=0.2: Esto significa que del 100% de los datos utilizara el 20% de los datos se utilizarán para la prueba y el 80% para el entrenamiento.

random_state=42: Esto indica el total de datos por grupo, es decir que cada grupo va tener de 42 datos, se hicieron de 42 datos porque cada vez que subía o bajaba el grupo bajaba la exactitud del modelo.

xgb.DMatrix: DMatrix optimizada de XGBoost que se utiliza para almacenar datos de entrenamiento y prueba.

X_train y y_train: Son los datos de entrenamiento y sus etiquetas correspondientes.

X_test y y_test: Son los datos de prueba y sus etiquetas correspondientes.

enable_categorical=True: Este parámetro permite que XGBoost maneje variables categóricas de manera nativa.

objective: Define el objetivo de la tarea de aprendizaje. En este caso, reg:squarederror indica que se trata de una regresión.

max_depth: La profundidad máxima de los árboles. Un valor mayor puede llevar a modelos más complejos.

eta: La tasa de aprendizaje. Este parámetro controla la contribución de cada árbol adicional (reduciendo la velocidad del aprendizaje para prevenir el sobreajuste).

eval_metric: La métrica de evaluación. rmse (Root Mean Squared Error) se usa para medir el rendimiento del modelo.

xgb.train: Esta función entrena el modelo con los parámetros definidos.

params: Los parámetros del modelo definidos anteriormente.

dtrain: Los datos de entrenamiento en formato DMatrix.

num_boost_round: El número de iteraciones de boosting. Esto define cuántos árboles se entrenarán.

Por último, **models[client] = bst** agregamos cada cliente en el arreglo generado y le igualamos el algoritmo de predicción

Esto nos genera los siguientes resultados.

```
Cliente: INDUSTRIA DE ELECTRODOMESTICOS, RMSE: 86.29503896791164
Cliente: INDUSTRIA DE IMPRESIONES II, RMSE: 57.497500767132266
Cliente: INDUSTRIA DE CERAMICA, RMSE: 153.57247274797925
Cliente: INDUSTRIA DE CONCENTRADOS, RMSE: 25.080559904675923
Cliente: INDUSTRIA ALIMENTICIA, RMSE: 19.939181791720095
Cliente: INDUSTRIA DE PAPEL, RMSE: 25.77567047155107
Cliente: INDUSTRIA DE PVC, RMSE: 157.92254964165576
Cliente: INDUSTRIA ALIMENTICIA II, RMSE: 26.130855137312686
Cliente: INDUSTRIA DE CONCENTRADOS II, RMSE: 70.45136209424012
Cliente: INDUSTRIA DE CEMENTO II, RMSE: 218.39779420297958
Cliente: INDUSTRIA DE CEMENTO I , RMSE: 102.90387141240316
Cliente: INDUSTRIA DE TUBERIAS I, RMSE: 4.691043424808039
Cliente: INDUSTRIA DE TUBERIAS II, RMSE: 21.185864049488536
Cliente: INDUCTRIA DE CERAMICAS II, RMSE: 8.839752482306688
Cliente: INDUSTRIA DE EMPAQUES, RMSE: 62.21842683957717
Cliente: INDUSTRIA DE PUERTOS, RMSE: 94.0568871989926
```

Por último, utilizamos nutra librería **matplotlib.pyplot** para empezar a generar graficas con nustros resultados obtenidos.

```python
# Generar gráficos de torta y guardarlos
plot_paths = []

# Calcular el consumo total por cliente para cada variable
total_kVarhD = fileHistoric[fileHistoric['VARIABLE'] == 'kVarhD'].groupby('CLIENTE')['USAGE_DATA'].sum()
total_kWhD = fileHistoric[fileHistoric['VARIABLE'] == 'kWhD'].groupby('CLIENTE')['USAGE_DATA'].sum()

# Crear la gráfica de torta para kVarhD
plt.figure(figsize=(8, 6))
plt.pie(total_kVarhD, explode=[0.1] * len(total_kVarhD), labels=total_kVarhD.index, autopct='%1.1f%%', shadow=True, startangle=140)
plt.title('Consumo de kVarhD por cliente')
plt.axis('equal')
plot_path_kVarhD = os.path.join('static', 'plots', 'pie_kVarhD.png')
plt.savefig(plot_path_kVarhD)
plt.close()
plot_paths.append('plots/pie_kVarhD.png')

# Crear la gráfica de torta para kWhD
plt.figure(figsize=(8, 6))
plt.pie(total_kWhD, labels=total_kWhD.index, autopct='%1.1f%%', shadow=True, startangle=140)
plt.title('Consumo de kWhD por cliente')
plt.axis('equal')
plot_path_kWhD = os.path.join('static', 'plots', 'pie_kWhD.png')
plt.savefig(plot_path_kWhD)
plt.close()
plot_paths.append('plots/pie_kWhD.png')
```

Tomamos la suma de cada una de las variables activa y reactiva. Ajustamos el tamaño de la imagen por pulgadas, le indicamos que va ser una gráfica de **torta(pie)** y pasamos como parámetros cada una de las sumas de nuestras variables, definimos los nombres que va tener cada grafica y devolvemos un arreglo con el nombre de las dos graficas de pastel.

```python
return render_template('index.html', plot_paths=plot_paths, bar_plot_paths=bar_plot_paths,
                       total_kWhD=total_kWhD, total_kVarhD=total_kVarhD, plots=plots,
                       clientes_info=clientes_info)
```

Retornamos cada uno de los arreglos de las grafica a nuestro archivo html.

```
{% for plot_path in plot_paths %}
    <div class="plot">
        <h2>{{ 'Consumo de kVarhD por cliente' if loop.index == 1 else 'Consumo de kWhD por cliente' if loop.index == 2 else 'Distribució
        <img src="{{ url_for('static', filename=plot_path) }}" alt="Pie chart">
    </div>
{% endfor %}
```

Recorremos el arreglo que devuelve Python y le indicamos que por cada variable se va generar un gráfico.

Para lograr la correcta ejecución del prototipo se debe hacer lo siguiente.

```
PS C:\Users\Usuario\Desktop\prototipo> env\Scripts\activate
(env) PS C:\Users\Usuario\Desktop\prototipo>
```

Ejecutar el entorno virtual donde se instalaron todas las librerías.

```
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 971-789-633
```

Ejecutamos el api con el comando **Python app.py** donde app.py es el nombre del archivo que estamos utilizando. Procedemos a ingresar el URL generado por el código.