```
import pandas as pd
import numpy as np
import pymc as pm
import arviz as az
import statsmodels.api as sm
import matplotlib.pyplot as plt
import scipy.stats as stats
from collections import Counter
from sklearn.linear_model import ElasticNet
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_absolute_error, mean_squared_error
from typing import List, Dict, Tuple
import pdb
import numpy as np
import pymc as pm
import arviz as az


def Probabilidad_Caidas(
    datos,      # lista de enteros (p.ej. 2465 valores)
    inicio,     # 0 o 1
    cantidad,   # p.ej. 10 para valores 0..9
    desde=50,   # índice 0-based de Python
    ventana=15, # tamaño de las ventanas históricas
    decimales=3 # dígitos en el resultado
):
    # 1) Rango de valores a evaluar
    rango = list(range(inicio, inicio + cantidad))

    # 2) Para cada ventana, cuento cuántas veces sale el dato siguiente
    conteos_siguientes = []
    for i in range(desde, len(datos)):
        grupo = datos[i - ventana : i]
        cnt   = Counter(grupo)[ datos[i] ]  # entero de 0 a ventana
        conteos_siguientes.append(cnt)

    # 3) Histograma de esos conteos (0..ventana → cuántas ventanas tuvieron ese conteo)
    hist_counts = Counter(conteos_siguientes)
    total       = len(conteos_siguientes)
    print("todos los datos datos")
    print(hist_counts)

    # 4) Conteo de cada valor v en la ÚLTIMA ventana de 15
    ult_grupo = datos[-ventana:]
    cnt_last  = Counter(ult_grupo)       # {v: veces que sale v en los últimos 15}
    print("Ultimos datos")
    print(cnt_last)

    # 5) Preparo p_last sólo para que puedas verlo si quieres
    p_last = {
        v: round(cnt_last.get(v, 0) / ventana, decimales)
        for v in rango
    }
    print("Probabilidades en la última ventana:", p_last)
```

```
    # 6) Para cada valor v, calculo su probabilidad final
    resultado = {}
    for v in rango:
        c = cnt_last.get(v, 0)          # cuántas veces cayó v en la última ventana
        # cuántas ventanas tuvieron ese mismo conteo, dividido entre el total
        prob = hist_counts.get(c, 0) / total
        resultado[v] = prob

    return resultado


def sumar_diccionarios(*dicts, Divisor=10):
    """
    Recibe N diccionarios con las mismas llaves y devuelve uno
    donde cada llave = suma de sus valores en los dicts recibidos.
    """
    # Asumo que hay al menos un dict y todos comparten exactamente las mismas llaves.
    claves = dicts[0].keys()
    return {k: sum(d[k]/Divisor for d in dicts) for k in claves}


def porcentaje_coincidencias(F_d: dict, datos: list) -> dict:
    conteo = Counter(datos)
    n = len(datos)
    return {
        k: round(conteo[val] / n , 4)
        for k, val in F_d.items()
    }

def aplicar_svr(lista_30, lista_15, lista_6, lista_sig):
    # Convertir las listas a arrays y preparar la matriz de características X.
    # Cada fila de X corresponde a una observación con tres características.
    X = np.column_stack((lista_30, lista_15, lista_6))
    y = np.array(lista_sig)

    # Definir el modelo SVR con kernel RBF (muy usado para relaciones no lineales)
    svr_model = SVR(kernel='rbf')

    # Definir una rejilla de hiperparámetros para ajustar el parámetro de penalización C y el gamma del kernel.
    param_grid = {
        "C": [ 118.509189,118.509191, 118.509192],
        "gamma": ['scale', 'auto', 0.0175003, 0.0175004]
    }

    # Usar GridSearchCV para buscar los mejores hiperparámetros usando validación cruzada
    grid_search = GridSearchCV(svr_model, param_grid, cv=5, scoring='neg_mean_squared_error')
    grid_search.fit(X, y)

    # Extraemos el mejor modelo encontrado, la mejor puntuación, y los parámetros óptimos.
    best_svr = grid_search.best_estimator_
    cv_score = grid_search.best_score_
    best_params = grid_search.best_params_

    preds_all = best_svr.predict(X)
    errors_pct_all = (preds_all - y) / (y + 1e-8)
    mpe_all = np.mean(errors_pct_all)
    preds_prev = preds_all[-20:]
    reales_prev = y[-20:]
```

```
        # Cálculo del error porcentual CON signo
    errores_pct10 = (preds_prev - reales_prev) / reales_prev
    mpe_last10  = np.mean(errores_pct10)

    return best_svr, cv_score, best_params, mpe_all, errores_pct10



def prediccion_bayesiana(lista_30, lista_15, lista_6, lista_sig):

    # Convertir listas a arrays
    np_lista_30 = np.array(lista_30)
    np_lista_15 = np.array(lista_15)
    np_lista_6  = np.array(lista_6)
    np_lista_sig = np.array(lista_sig)

    # Preparar datos de entrenamiento (todos menos el último)
    X_train = np.column_stack((np_lista_30[:-1], np_lista_15[:-1], np_lista_6[:-1]))
    y_train = np_lista_sig[:-1]

    # x_new: el último registro para predecir
    x_new = np.array([np_lista_30[-1], np_lista_15[-1], np_lista_6[-1]])

    # Definir el modelo bayesiano
    with pm.Model() as modelo:
        alpha = pm.Normal("alpha", mu=0, sigma=10)
        beta = pm.Normal("beta", mu=0, sigma=10, shape=3)
        sigma = pm.HalfNormal("sigma", sigma=1)

        mu = alpha + pm.math.dot(X_train, beta)
        y_obs = pm.Normal("y_obs", mu=mu, sigma=sigma, observed=y_train)

        trace = pm.sample(2000, tune=2000, chains=4, target_accept=0.98, return_inferencedata=True)

    # Usar sample_posterior_predictive y obtener un diccionario simple
    with modelo:
        ppc = pm.sample_posterior_predictive(
            trace,
            var_names=["alpha", "beta"],
            random_seed=42,
            return_inferencedata=False
        )

    # Extraer muestras usando el diccionario
    alpha_samples = ppc["alpha"]
    beta_samples = ppc["beta"]

    # Calcular predicciones para cada muestra
    predicciones = alpha_samples + np.dot(beta_samples, x_new)

    # Calcular la predicción final y el intervalo del 95%
    prediccion_media = np.mean(predicciones)
    pred_int2_5 = np.percentile(predicciones, 1)
    pred_int97_5 = np.percentile(predicciones, 99)

    return {
        "prediccion_media": prediccion_media,
        "int_95": (pred_int2_5, pred_int97_5),
        "trace": trace
```

```
    }

def leer_datos_excel(file_path):
    df = pd.read_excel(file_path)
    columna = pd.to_numeric(df['A'], errors='coerce').dropna()
    return columna

def obtener_siguiente_numero(columna):
    ultima_caida = columna.iloc[-1]
    return [columna[i + 1] for i in range(len(columna) - 1) if columna[i] == ultima_caida]

def Siguientes_lista(lista):
    """
    Busca todas las veces que el último valor de la lista aparece (salvo en la última posición)
     y devuelve los elementos que le siguen inmediatamente.
    """
    ultima = lista[-1]
    # zip(lista, lista[1:]) empareja (lista[0],lista[1]), (lista[1],lista[2]), …
    return [siguiente for actual, siguiente in zip(lista, lista[1:])
            if actual == ultima]


def obtener_historial_caidas(columnas):

    caidas_columna = []
    ultimas_posiciones = [-1] * 10
    for i, valor in enumerate(columnas):
        if ultimas_posiciones[valor] == -1:
            jugadas = i + 1
        else:
            jugadas = i - ultimas_posiciones[valor]
        if jugadas > 40:
            jugadas = 40 if jugadas % 2 == 0 else 39
        caidas_columna.append(jugadas)
        ultimas_posiciones[valor] = i
    return caidas_columna

def obtener_siguiente_caidas(columnas):
    siguiente_caidas = []
    caidas = obtener_historial_caidas(columnas)
    ultima_caida = caidas[-1]
    for i in range(len(caidas) - 1):
        if caidas[i] == ultima_caida:
            siguiente = min(caidas[i + 1], 40)
            siguiente_caidas.append(siguiente)
    return siguiente_caidas

def Semanas(columna):
    grupo = columna.tail(50)
     # Calcular cuántas jugadas han pasado desde la última aparición de cada número
    apariciones = {}
    for num in range(10):
        if num in grupo.tolist():
            # Encontrar la posición de la última aparición y calcular la distancia desde el final
            ultima_posicion = len(grupo) - 1 - grupo[::-1].tolist().index(num)
            distancia = len(grupo)  - ultima_posicion
        else:
            # Si el número no aparece en el grupo, asignar 40 como default
            if num % 2 == 0:
```

```
            distancia = 40
        else :
            distancia = 39

    apariciones[num] = distancia
  return apariciones

def calcular_alpha_prior(columna):

  limite = int(len(columna) * 0.9)  # Define el 90% del total
  grupo = columna.iloc[:limite]  # Obtiene la parte inicial de la columna
  alpha_prior = {num: grupo.tolist().count(num) for num in range(10)}  # Cuenta ocurrencias
  return alpha_prior

def calcular_alpha_prior_Lista(columna):

  limite = int(len(columna) * 0.9)  # Define el 90% del total
  print("Cantidad de Numeros Siguientes ",len(columna))
  grupo = columna[:limite]  # Extrae los últimos 50 elementos de la lista
  frecuencias = {num: grupo.count(num) for num in range(10)}  # Cuenta ocurrencias

  #print(frecuencias)  # Muestra las frecuencias en consola
  return frecuencias

def ultima_jerarquia(columna):
  grupo = columna.tail(50)
  frecuencias = {num: grupo.tolist().count(num) for num in range(10)}  # Usa .count() en la lista
  #print(frecuencias)  # Ahora imprimirá las frecuencias correctamente
  return frecuencias

def ultima_jerarquia_Lista(columna):
  """
  Calcula la frecuencia de aparición de cada dígito en los últimos 50 elementos de una lista.
  Parámetro: columna: Lista de números.
  Retorna: Un dicc con la cantidad de veces que ha aparecido cada dígito en los últimos 50 elementos.
  """
  grupo = columna[-50:]  # Extrae los últimos 50 elementos de la lista
  frecuencias = {num: grupo.count(num) for num in range(10)}  # Cuenta ocurrencias
  #print(frecuencias)  # Muestra las frecuencias en consola

  return frecuencias

def calcular_jerarquias(columna):

  jerarquia = []
  posiciones = []
  for i in range(len(columna) - 2, 51,-1 ):
    grupo = columna[max(0, i - 49):i + 1]
    frecuencias = {num: grupo.value_counts().get(num, 0) for num in range(10)}
    primeras_apariciones = {num: (len(grupo) - 1 - grupo[::-1].tolist().index(num) if num in grupo.tolist() else 60)
for num in range(10)}
    ordenados = sorted(frecuencias.items(), key=lambda x: (x[1], primeras_apariciones[x[0]]))
    jerarquia.append(ordenados)
    #print(ordenados)
    if i < len(columna) - 1:
      siguiente_dato = columna[i + 1]
      for pos, (num, _) in enumerate(ordenados):
        if num == siguiente_dato:
          posiciones.append(pos + 1)
```

```
                #print(pos)
                break
    jerarquia.reverse()
    return jerarquia, posiciones

def calcular_mayores_pares(columna):
    mayores = [num for num in columna if num > 4]
    pares = [num for num in columna if num % 2 == 0]
    return mayores, pares

def aplicar_regresion_logistica_mayor_menor(columna):
    if len(columna) < 50:
        print("Error: Insuficientes datos para regresión logística (mayor/menor).")
        return None
    X = np.array(columna[:-1]).reshape(-1, 1)
    y = np.array([1 if num > 4 else 0 for num in columna[1:]])
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    modelo = LogisticRegression().fit(X_train, y_train)
    ultimo_numero = np.array(columna.iloc[-1]).reshape(1, 1)
    return modelo.predict_proba(ultimo_numero)[0][1]

def aplicar_regresion_logistica_par_impar(columna):
    if len(columna) < 50:
        print("Error: Insuficientes datos para regresión logística (par/impar).")
        return None
    X = np.array(columna[:-1]).reshape(-1, 1)
    y = np.array([1 if num % 2 == 0 else 0 for num in columna[1:]])
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    modelo = LogisticRegression().fit(X_train, y_train)
    ultimo_numero = np.array(columna.iloc[-1]).reshape(1, 1)
    return modelo.predict_proba(ultimo_numero)[0][1]

def calcular_promedios_y_errores(columna):
    promedio_general = np.mean(columna)
    # Calcular lista_30 y errores_30
    lista_30 = [np.mean(columna[i - 30:i]) for i in range(35, len(columna))]
    errores_30 = [(p - promedio_general) / promedio_general for p in lista_30]

    # Calcular lista_15 y errores_15 con índice correcto
    lista_15 = [np.mean(columna[i - 8:i]) for i in range(35, len(columna))]
    errores_15 = [(p - promedio_general) / promedio_general for p in lista_15]

    # Calcular lista_6 y errores_6
    lista_6 = [np.mean(columna[i - 3:i]) for i in range(35, len(columna))]
    errores_6 = [(p - promedio_general) / promedio_general for p in lista_6]

    # Calcular lista_sig
    lista_sig = [np.mean(columna[i - 14:i + 1]) for i in range(35, len(columna) + 1)]
    lista_sig = lista_sig[:-1]  # Ajustar tamaño

    lista_14=sum(columna[-14:])
    l5=sum(columna[-5:])

    return errores_30, errores_15, errores_6, lista_sig, lista_14, l5, promedio_general


def promedios_y_errores_lista(data):
    """
    data: lista de números (floats o ints)
```

```
    Devuelve: (errores_30, errores_15, errores_6, lista_sig, suma_14, promedio_general)
    """
    n = len(data)
    if n == 0:
        raise ValueError("La lista no puede estar vacía.")

    # 1) Promedio general
    promedio_general = sum(data) / n

    # 2) Ventanas deslizantes y errores relativos
    #    Para i en [40 .. n-1], calculamos la media de los últimos k elementos
    def medias_ventana(k):
        return [ sum(data[i-k: i]) / k for i in range(30, n) ]

    lista_30 = medias_ventana(30)
    lista_15 = medias_ventana(15)
    lista_6  = medias_ventana(6)

    def errores(lista_medias):
        return [ (m - promedio_general) / promedio_general for m in lista_medias ]

    errores_30 = errores(lista_30)
    errores_15 = errores(lista_15)
    errores_6  = errores(lista_6)

    # 3) lista_sig: media de ventana de tamaño 15, pero alineada como en tu código original
    #    Para i en [40 .. n], media de data[i-14 : i+1], luego descartamos el último
    lista_sig = [ sum(data[i-14: i+1]) / 15 for i in range(30, n+1) ]
    lista_sig = lista_sig[:-1]  # para que mida igual que errores_30

    # 4) suma de los últimos 14 valores
    suma_14 = sum(data[-14:])

    return errores_30, errores_15, errores_6, lista_sig, suma_14, promedio_general


def aplicar_regresion_ponderada(lista_30, lista_15, lista_6, lista_sig):
    if not all([lista_30, lista_15, lista_6, lista_sig]) or not len(lista_30) == len(lista_15) == len(lista_6) ==
len(lista_sig):
        print("Error en los datos para regresión ponderada.")
        return None

    # Preparación de datos
    X = np.array([lista_30, lista_15, lista_6]).T
    y = np.array(lista_sig)
    X_ols = sm.add_constant(X)

    # Regresión OLS inicial
    modelo_ols = sm.OLS(y, X_ols).fit()
    residuos = modelo_ols.resid

    ### □ MÉTODO 1: Pesos inversos al error (el que ya tenías)
    pesos_1 = 1 / (residuos ** 2 + 1e-6)
    modelo_wls_1 = sm.WLS(y, X_ols, weights=pesos_1).fit(cov_type='HC0')

    ### ● MÉTODO 2: Pesos con raíz cuadrada del error
    pesos_2 = 1 / np.sqrt(residuos ** 2 + 1e-6)
    modelo_wls_2 = sm.WLS(y, X_ols, weights=pesos_2).fit(cov_type='HC0')
```

```
    # Extraer coeficientes para ambas versiones
    coeficientes_1 = modelo_wls_1.params
    coeficientes_2 = modelo_wls_2.params

    # Calcular métricas para ambas versiones
    resultados = {
        "Método 1: Pesos inversos al error": {
            "Intercepto": coeficientes_1[0],
            "Coef. lista_30": coeficientes_1[1],
            "Coef. lista_15": coeficientes_1[2],
            "Coef. lista_6": coeficientes_1[3],
            "Porcentaje de variabilidad explicada por el modelo (cuanto mayor, mejor)": modelo_wls_1.rsquared,
            "Promedio del error al cuadrado entre lo predicho y lo real (menores valores son mejores)":
mean_squared_error(y, modelo_wls_1.predict(X_ols)),
            "Promedio absoluto de error entre lo predicho y lo real (indica desviación en las mismas unidades)":
mean_absolute_error(y, modelo_wls_1.predict(X_ols))
        },
        "Método 2: Pesos con raíz cuadrada del error": {
            "Intercepto": coeficientes_2[0],
            "Coef. lista_30": coeficientes_2[1],
            "Coef. lista_15": coeficientes_2[2],
            "Coef. lista_6": coeficientes_2[3],
            "Porcentaje de variabilidad explicada por el modelo (cuanto mayor, mejor)": modelo_wls_2.rsquared,
            "Promedio del error al cuadrado entre lo predicho y lo real (menores valores son mejores)":
mean_squared_error(y, modelo_wls_2.predict(X_ols)),
            "Promedio absoluto de error entre lo predicho y lo real (indica desviación en las mismas unidades)":
mean_absolute_error(y, modelo_wls_2.predict(X_ols))
        }
    }

    return resultados


def aplicar_regresion_elasticnet(lista_30, lista_15, lista_6, lista_sig, alpha=0.8, l1_ratio=0.3, max_iter=2000,
tol=0.00001,):
    if not all([lista_30, lista_15, lista_6, lista_sig]) or not len(lista_30) == len(lista_15) == len(lista_6) ==
len(lista_sig):
        print("Error en los datos para Elastic Net.")
        return None
    X = np.array([lista_30, lista_15, lista_6]).T
    y = np.array(lista_sig)
    modelo_enet = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, max_iter=max_iter, tol=tol).fit(X, y)
    return modelo_enet.intercept_, modelo_enet.coef_



def aplicar_regresion_robusta(lista_30, lista_15, lista_6, lista_sig):
    if not all([lista_30, lista_15, lista_6, lista_sig]) or not len(lista_30) == len(lista_15) == len(lista_6) ==
len(lista_sig):
        print("Error en los datos para regresión robusta.")
        return None

    # Preparación de datos
    X = np.array([lista_30, lista_15, lista_6]).T
    y = np.array(lista_sig)
    X_ols = sm.add_constant(X)  # Agregar intercepto

    # Aplicar regresión robusta con método HuberT
```

```
    modelo_rlm = sm.RLM(y, X_ols, M=sm.robust.norms.HuberT()).fit()

    # Calcular un pseudo R² manualmente:
    ss_res = np.sum(modelo_rlm.resid ** 2)
    ss_tot = np.sum((y - np.mean(y)) ** 2)
    r2_pseudo = 1 - ss_res / ss_tot if ss_tot != 0 else None

    coeficientes = modelo_rlm.params

    return {
        "Intercepto": coeficientes[0],
        "Coef. lista_30": coeficientes[1],
        "Coef. lista_15": coeficientes[2],
        "Coef. lista_6": coeficientes[3],
        "Estimación de variabilidad explicada por el modelo (Pseudo R²)": r2_pseudo
    }


def analizar_siguientes_numeros_para_probabilidades(siguiente_numeros):
    if not siguiente_numeros:
        return {i: 0 for i in range(10)}
    frecuencia = {i: siguiente_numeros.count(i) for i in range(10)}
    total = len(siguiente_numeros)
    #print("Prior Numeros/Siguientes")
    #print(frecuencia)
    return {num: freq / total if total > 0 else 0 for num, freq in frecuencia.items()}


def calcular_probabilidades_regresion(params_wls, params_enet, lista_30, lista_15, lista_6):
    if params_wls is None or params_enet is None or not lista_30 or not lista_15 or not lista_6:
        return {"WLS": None, "ElasticNet": None}
    if len(lista_30) != len(lista_15) or len(lista_30) != len(lista_6):
        print("Error: Las listas de promedios tienen longitudes inconsistentes.")
        return {"WLS": None, "ElasticNet": None}

    ultima_media_30 = lista_30[-1] if lista_30 else 0
    ultima_media_15 = lista_15[-1] if lista_15 else 0
    ultima_media_6 = lista_6[-1] if lista_6 else 0

    prediccion_wls = params_wls[0] + params_wls[1] * ultima_media_30 + params_wls[2] * ultima_media_15 +
params_wls[3] * ultima_media_6

    intercepto_enet, coefs_enet = params_enet
    prediccion_enet = intercepto_enet + coefs_enet[0] * ultima_media_30 + coefs_enet[1] * ultima_media_15 +
coefs_enet[2] * ultima_media_6

    return {"WLS": prediccion_wls, "ElasticNet": prediccion_enet}


def calcular_regresion_wls_metodo1(lista_30, lista_15, lista_6, lista_sig):
    """
    Aplica la regresión ponderada y extrae los coeficientes del Método 1.
    """
    params = aplicar_regresion_ponderada(lista_30, lista_15, lista_6, lista_sig)
    if params:
        return {
            "Intercepto": params["Método 1: Pesos inversos al error"]["Intercepto"],
            "Coef_lista_30": params["Método 1: Pesos inversos al error"]["Coef. lista_30"],
            "Coef_lista_15": params["Método 1: Pesos inversos al error"]["Coef. lista_15"],
```

```
            "Coef_lista_6": params["Método 1: Pesos inversos al error"]["Coef. lista_6"]
        }
    else:
        return None


def calcular_regresion_wls_metodo2(lista_30, lista_15, lista_6, lista_sig):
    """
    Aplica la regresión ponderada y extrae los coeficientes del Método 2.
    """
    params = aplicar_regresion_ponderada(lista_30, lista_15, lista_6, lista_sig)
    if params:
        return {
            "Intercepto": params["Método 2: Pesos con raíz cuadrada del error"]["Intercepto"],
            "Coef_lista_30": params["Método 2: Pesos con raíz cuadrada del error"]["Coef. lista_30"],
            "Coef_lista_15": params["Método 2: Pesos con raíz cuadrada del error"]["Coef. lista_15"],
            "Coef_lista_6": params["Método 2: Pesos con raíz cuadrada del error"]["Coef. lista_6"]
        }
    else:
        return None


def calcular_regresion_robusta(lista_30, lista_15, lista_6, lista_sig):
    """
    Aplica la regresión robusta y extrae los coeficientes (se puede usar el Método 1 como referencia).
    """
    params = aplicar_regresion_robusta(lista_30, lista_15, lista_6, lista_sig)
    if params:
        return {
            "Intercepto": params["Intercepto"],
            "Coef_lista_30": params["Coef. lista_30"],
            "Coef_lista_15": params["Coef. lista_15"],
            "Coef_lista_6": params["Coef. lista_6"],
            "R2": params["Estimación de variabilidad explicada por el modelo (Pseudo R²)"]
        }
    else:
        return None


def predecir_con_regresion(parametros, params_enet, lista_30, lista_15, lista_6):
    """
    Calcula las probabilidades de predicción usando la función existente
    'calcular_probabilidades_regresion.'
    Se espera que 'parametros' sea un diccionario con las llaves "Intercepto", "Coef_lista_30", etc.
    """
    # Primero, convierte el diccionario 'parametros' en una lista en el orden esperado:
    if parametros is None:
        return None
    parametros_lista = [
        parametros["Intercepto"],
        parametros["Coef_lista_30"],
        parametros["Coef_lista_15"],
        parametros["Coef_lista_6"]
    ]
    return calcular_probabilidades_regresion(parametros_lista, params_enet, lista_30, lista_15, lista_6)

def procesar_regresiones(datos):

    lista_30, lista_15, lista_6, lista_sig = datos
```

```
    # Aseguramos que se usan segmentos consistentes:
    datos_regresion = (
        lista_30[-len(lista_sig):],
        lista_15[-len(lista_sig):],
        lista_6[-len(lista_sig):],
        lista_sig
    )

    # Extraer parámetros para cada método
    params_wls1 = calcular_regresion_wls_metodo1(*datos_regresion)
    params_wls2 = calcular_regresion_wls_metodo2(*datos_regresion)
    params_rlm  = calcular_regresion_robusta(*datos_regresion)
    params_enet = aplicar_regresion_elasticnet(*datos_regresion)  # Suponiendo que esta función ya existe

    # Calcular predicciones
    resultados_m1 = predecir_con_regresion(params_wls1, params_enet, lista_30, lista_15, lista_6)
    resultados_m2 = predecir_con_regresion(params_wls2, params_enet, lista_30, lista_15, lista_6)
    resultados_rlm = predecir_con_regresion(params_rlm, params_enet, lista_30, lista_15, lista_6)

    return {
        "WLS_Metodo1": resultados_m1,
        "WLS_Metodo2": resultados_m2,
        "RLM": resultados_rlm
    }


def inferir_probabilidades_bayesianas(frecuencias, alpha_prior):
    # Usamos un prior uniforme si no se proporciona
    if alpha_prior is None:
        alpha_prior = {i: 1 for i in range(10)}

    # Calculamos la suma total del prior (en este caso, 10, ya que 1 para cada dígito)
    total_alpha = sum(alpha_prior.values())
    # Total de observaciones (suma de todas las frecuencias)
    total_frecuencias = sum(frecuencias.get(i, 0) for i in range(10))
    # Suma total del posterior
    total_posterior = total_alpha + total_frecuencias
    # Calculamos la media del posterior para cada dígito:
    probabilidades_posterior = {}
    for i in range(10):
        # alpha posterior: prior + observación
        alpha_post = alpha_prior.get(i, 1) + frecuencias.get(i, 0)
        probabilidades_posterior[i] = alpha_post / total_posterior

    return probabilidades_posterior


def ultimos_promedios_list(data: List[float]) -> List[float]:
    n = len(data)
    min_needed = 15 + 5 - 1
    if n < min_needed:
        raise ValueError(
            f"Se requieren al menos {min_needed} elementos, pero hay {n}."
        )
    # i recorre count-1, count-2, …, 0
    return [
        sum(data[n - 15 - i : n - i]) / 15
        for i in reversed(range(10))
    ]
```

```
def ultimos_promedios_series(data: pd.Series) -> List[float]:
    if len(data) < 15:
        # no hay ni un solo promedio completo
        return pd.Series(dtype=float)

    medias = data.rolling(15).mean().dropna()
    return medias.iloc[-10:].tolist()



def procesar_e_imprimir_regresion(titulo, Lista, start=0,stop=10, medio=5, ante=2,fin=7):
    L_30, L_15, L_6, L_sig, Sum14, S5, PROM = calcular_promedios_y_errores(Lista)
    Bloque= ultimos_promedios_list(Lista)

    print("\033[93m           * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - * - *\033[0m")
    print(aviso_ansi(f"Resultados para {titulo}:", fg=37, bg=101))
    #print(f"Resultados para {titulo}:")
    best_svr, cv_score, best_params,PromT,Err6 = aplicar_svr(L_30, L_15, L_6, L_sig)
    print("Parámetros para SVR:", best_params)
    print(f"\033[31mPromedio :\033[0m", end="\t")
    print(f"\033[1;31;47m{PROM:.3f}\033[0m", end="\t\t")
    S14=(Sum14+medio)/15
    S142=(Sum14+ante)/15
    S146=(Sum14+fin)/15
    S5=(S5+medio)/6
    # Se toma el último valor de cada lista para formar una nueva observación.
    nuevo_dato = np.array([[L_30[-1], L_15[-1], L_6[-1]]])
    prediccion = best_svr.predict(nuevo_dato)
    Pprom=prediccion[0]

    #if Pprom - S14 > 0.3:
    #        Pprom=S14+0.3
    #if S14-Pprom > 0.3:
    #    Pprom=S14-0.3

    datos_regresion = (L_30, L_15, L_6, L_sig)

    # Códigos ANSI
    BG_BLUE = '\033[46m'
    RED_ON_YELLOW = "\033[37;45m"
    RESET       = "\033[0m"
    # Verificar que todas las listas contengan datos, tengan la misma longitud y mayor a 50.
    if (L_30 and L_15 and L_6 and L_sig and
        len(L_30) == len(L_15) == len(L_6) == len(L_sig) and len(L_30) > 50):

        resultados = procesar_regresiones(datos_regresion)

        # Extraer el valor de ElasticNet (se asume que es consistente en todos los métodos)
        primer_metodo = next(iter(resultados))
        valor_elasticnet = resultados[primer_metodo]["ElasticNet"]

        print(f"\033[1;91;43mRegresion : {Pprom:.3f}\033[0m", end="\t")
        #print("* Nuevo dato: ", f"{Pprom:.4f}", end="\t")
        print(f"'ElasticNet': {valor_elasticnet:.3f}", end="\t\t")

        resultados = [Pprom * factor for factor in (0.9, 0.95, 1.05, 1.1)]
        # Unimos con dos espacios como separador y lo imprimimos en la misma línea
```

```
        print("  ".join(f"{RED_ON_YELLOW}{valor:.3f}{RESET}" for valor in resultados))
        print("")
        print(f"\033[34m\t\tProm + {ante}: {S142:.3f}\t\033[0m", end="\t")
        print(f"\033[31mPromedio Actual: {S14:.3f}\t\033[0m", end="\t")
        print(f"\033[34mProm + {fin}: {S146:.3f}\t\033[0m")

        # Extrae y formatea los 12 últimos
        formato=[]
        for x in Err6[-15:]:
            form = f"{x:.3f}"
            print(colorear2(x), end="\t")
        print(" ")
        Prom10 = sum(Err6[-15:]) / 15
        Prom6 = sum(Err6[-4:]) / 4

        print(colorear(PromT, "Promedio "), end="  ")
        print(colorear(Prom10, "P. 10 "), end="    ")
        print(colorear(Prom6, "P. 4 "))
        print("\033[93mUltimos: \033[0m",end=" ")

        # Formateo de Bloque y Err6
        bloque_fmt = [f"{x:.2f}" for x in Bloque[-15:]]
        err6_fmt   = [f"{x:.3f}" for x in Err6[-7:]]

        # Une todo en una línea, coloreando de azul el fondo de Err6
        all_values = bloque_fmt + ["\t"]+ [f"{BG_BLUE}{v}{RESET}" for v in err6_fmt]
        print(*all_values, sep="  ")
        print(" -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ")

        nuevos_valores, errores_ajustados = imprimir_valores_y_errores(Sum14, Pprom,start,stop)
        minY=min(nuevos_valores)
        maxY=max(nuevos_valores)
        Prom_Gral = solicitar_nuevo_prom(S14, minY, maxY)
        nuevos_valores, errores_ajustados = imprimir_valores_y_errores(Sum14, Prom_Gral, start, stop,
col_val="\033[92m", col_err="\033[92m")
        print("")
        return  errores_ajustados, nuevos_valores, Sum14, Prom_Gral

    else:
        print(f"No hay datos suficientes o las listas no cumplen las condiciones para {titulo}.")
        return None, None, None, None


def aviso_ansi(texto, fg=37, bg=41, style=1):
    # 1 = negrita, 37 = texto blanco, 41 = fondo rojo
    return f"\033[{style};{fg};{bg}m{texto}\033[0m"


def solicitar_nuevo_prom(s14, min_val, max_val):
    prompt  = aviso_ansi(
        f"⚠  Introduce nuevo Prom_Gral "
        f"(entre {min_val:.3f} y {max_val:.3f}): "
    )
    while True:
        try:
            nuevo = float(input(prompt))
        except ValueError:
            print(aviso_ansi("→ Entrada no válida. Sólo números."))
```

```
            continue

        if min_val <= nuevo <= max_val:
            return nuevo
        print(aviso_ansi(
            f"→ Fuera de rango. Debe ser entre {min_val:.3f} y {max_val:.3f}."
        ))



def colorear(valor, etiqueta):
    # amarillo brillante para la etiqueta
    azul  = "\033[34m"
    rojo  = "\033[31m"
    amarillo = "\033[92m"
    reset = "\033[0m"

    num_color = azul if valor >= 0 else rojo
    return f"{amarillo}{etiqueta}{reset} = {num_color}{valor:.3f}{reset}"

def colorear2(valor):
    fondo = "\033[104m"  # Fondo gris claro
    if valor >= 0:
        texto = "\033[30m" if valor > 0 else "\033[30m"  # Azul si >0, negro si ==0
    else:
        texto = "\033[31m"  # Rojo si negativo
    return f"{fondo}{texto}{valor:.3f}\033[0m"


def imprimir_valores_y_errores(s14, p_gral, start=0, stop=10, col_val="\033[93m", col_err="\033[96m"):
    formateados = []
    errores    = []
    nuevos     = []
    ajustados  = []

    for i in range(start, stop):
        val   = (s14 + i) / 15
        err   = (val - p_gral) / p_gral

        formateados.append(f"{val:.3f}")
        errores.append(f"{err:.3f}")
        ajustados.append(err * -0.99 if err < 0 else err)
        nuevos.append(val)

    print(f"{col_val}{'   '.join(formateados)}\033[0m")
    print("  ".join(colorear2(float(e)) for e in errores))
    return nuevos, ajustados


#def solicitar_nuevo_prom(s14, min, max):
#    prompt  = (f"Introduce el nuevo Prom_Gral "
#               f"(entre {min:.3f} y {max:.3f}): ")

#    while True:
#        try:
#            nuevo = float(input(prompt))
#        except ValueError:
#            print("→ Entrada no válida. Usa un número.")
#            continue
```

```
#       if min_val <= nuevo <= max_val:
#           return nuevo
#       print(f"→ Fuera de rango. Debe estar entre {min_val:.3f} y {max_val:.3f}.")


def procesar_regresion_Histo(titulo, Lista, Valores, medio=7, ante=3,fin=11,start=1, stop=15):
    L_30, L_15, L_6, L_sig, Sum14, PROM = promedios_y_errores_lista(Lista)
    Bloque= ultimos_promedios_list(Lista)

    print("*******************************")
    print(f"\033[31mPromedio {titulo}:\033[0m", end="\t")
    print(f"\033[1;91;47m{PROM:.4f}\033[0m", end="\t\t")
    S14=(Sum14+medio)/15
    S142=(Sum14+ante)/15
    S1410=(Sum14+fin)/15
    S5=(sum(L_sig[-5:])+medio)/6

    best_svr, cv_score, best_params,PromT,Err6 = aplicar_svr(L_30, L_15, L_6, L_sig)
    print("Parámetros para SVR:", best_params)
    print()

    # Se toma el último valor de cada lista para formar una nueva observación.
    nuevo_dato = np.array([[L_30[-1], L_15[-1], L_6[-1]]])
    prediccion = best_svr.predict(nuevo_dato)
    Pprom=prediccion[0]
    datos_regresion = (L_30, L_15, L_6, L_sig)
    # Verificar que todas las listas contengan datos, tengan la misma longitud y mayor a 50.
    if (L_30 and L_15 and L_6 and L_sig and
        len(L_30) == len(L_15) == len(L_6) == len(L_sig) and len(L_30) > 50):

        resultados = procesar_regresiones(datos_regresion)
        print(f"Resultados para {titulo}:")

        # Extraer el valor de ElasticNet (se asume que es consistente en todos los métodos)
        primer_metodo = next(iter(resultados))
        valor_elasticnet = resultados[primer_metodo]["ElasticNet"]

        valores_wls = [float(datos["WLS"]) for datos in resultados.values()]
        Prom_wls = sum(valores_wls) / len(valores_wls)
        Prom_Gral=(Pprom+Prom_wls+2*valor_elasticnet)/4
        print(f"\033[1;91;46mRegresion : {Prom_Gral:.4f}\033[0m", end="\t")
        print("** Nuevo dato: ", f"{Pprom:.4f}", end="\t")
        print(f"'ElasticNet': {valor_elasticnet:.4f}", end="\t")
        print(f"Prom WLS: {Prom_wls:.4f}")
        print(f"\033[34mPromedio + {ante}: {S142:.3f}\t\033[0m", end="\t")
        print(f"\033[31mPromedio Actual: {S14:.3f}\t\033[0m", end="\t")
        print(f"\033[34mPromedio + {fin}: {S1410:.3f}\t\033[0m")

        Prom10 = sum(L_15[-10:]) / len(L_15[-10:])
        Prom6 = sum(L_15[-6:]) / len(L_15[-6:])

        print("")
        print(colorear(PromT, "Promedio "), end="  ")
        print(colorear(Prom10, "P. 10 "), end="    ")
        print(colorear(Prom6, "P. 6 "), end="    ")
        print(colorear(S5, "Prom. Medio 6"))

        BG_BLUE = '\033[44m'
```

```
        RESET   = '\033[0m'
        # Formateo de Bloque y Err6
        bloque_fmt = [f"{x:.2f}" for x in Bloque[-8:]]
        err6_fmt   = [f"{x:.3f}" for x in Err6[-8:]]

        # Une todo en una línea, coloreando de azul el fondo de Err6
        all_values = bloque_fmt + ["\t"]+ [f"{BG_BLUE}{v}{RESET}" for v in err6_fmt]
        print(*all_values, sep="  ")
        prom_er6 = sum(Err6) / len(Err6)
        print(f"Prom errores 6: {prom_er6:.3f}")
        print(" -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ")

        nuevos, errores, textoN,textoE = calcular_nuevos_y_errores(Valores, Sum14, Prom_Gral)

        linea = "\t".join(textoN)
        linea1 = "\t".join(textoE)
        print(f"\033[92m{linea}\033[0m")
        print(f"\033[95m{linea1}\033[0m")
        v_min = min(nuevos.values())
        v_max = max(nuevos.values())
        Prom_Gral = solicitar_nuevo_prom(S14, v_min, v_max)
        nuevos, errores, texto,textoE = calcular_nuevos_y_errores(Valores, Sum14, Prom_Gral)
        linea = "\t".join(textoN)
        linea1 = "\t".join(textoE)
        print(f"\033[92m{linea}\033[0m")
        print(f"\033[95m{linea1}\033[0m")

        print()
        # Retornamos ambas listas para que el main pueda reordenarlas si es necesario.
        return  errores, nuevos, Sum14

    else:
        print(f"No hay datos suficientes o las listas no cumplen las condiciones para {titulo}.")
        return None, None, None, None



def calcular_nuevos_y_errores(Valores: Dict[str, float], Sum14: float, Prom_Gral: float) -> Tuple[Dict[str, float],
Dict[str, float], List[str]]:

    errores_por_clave = {}
    nuevos_por_clave  = {}
    formateados1      = []
    formateados2      = []

    for clave, incremento in Valores.items():
        nuevo_valor = (Sum14 + incremento) / 15
        error = (nuevo_valor - Prom_Gral) / Prom_Gral
        if error < 0:
            error *= -0.99

        nuevos_por_clave[clave]  = nuevo_valor
        errores_por_clave[clave] = error
        formateados1.append(f"{nuevo_valor:.2f}")
        formateados2.append(f"{error:.3f}")

    return nuevos_por_clave, errores_por_clave, formateados1,formateados2
```

```
def inferir_probabilidades_bayesianas1(orden_digitos, historial_posiciones):

    num_posiciones = len(orden_digitos)  # Número total de posiciones
    print("Cantidad Posiciones",num_posiciones)
    # Calculamos los totales de prior y evidencia
    total_evidence = sum(orden_digitos.values())
    print("Total Prior",total_prior)
    total_prior = sum(historial_posiciones.values())
    print("Total Evidence",total_evidence)
    total_combined = total_prior + total_evidence

    # Calculamos probabilidades posteriores
    posterior_probs = {}
    for digito in orden_digitos:
        posterior_probs[digito] = (orden_digitos[digito] + historial_posiciones[digito]) / total_combined
        print(f"Dígito {digito} => orden: {orden_digitos[digito]}, historial: {historial_posiciones[digito]}")

    # Devolvemos el diccionario con las mismas claves (0, 1, 2, …)
    return posterior_probs


def imprimir_tabla(Titulo, data, es_decimal=False, highlight_key=None):
    # ANSI colors
    RED = "\033[1;31m"
    YELLOW = "\033[33m"
    RESET = "\033[0m"

    # Caso diccionario: se mantienen las claves en el orden de inserción
    if isinstance(data, dict):
        claves = list(data.keys())
        cabecera = [str(k) for k in claves]
        valores = [data[k] for k in claves]
    # Caso lista: se usan los índices de la lista como cabecera
    elif isinstance(data, list):
        cabecera = [str(i) for i in range(len(data))]
        valores = data
    else:
        print("Tipo de dato no soportado (se esperaba diccionario o lista).")
        return

    # Formateo de la fila de datos:
    if es_decimal:
        # Se formatean los números a 4 dígitos decimales
        fila_datos = [f"{v:.4f}" for v in valores]
    else:
        fila_datos = [str(v) for v in valores]

    # Determina el ancho mínimo que se necesita para cada celda (según la mayor longitud entre cabecera y
datos)
    ancho_min = max(max(len(s) for s in cabecera), max(len(s) for s in fila_datos))
    # Se añade un pequeño padding (2 espacios adicionales)
    ancho_celda = ancho_min + 2
    num_cols = len(cabecera)

    # índices de las 4 columnas centrales
    mid         = num_cols // 2
    offset      = 1 if num_cols % 2 else 0
    centro_idxs = list(range(mid - 2 + offset, mid + 2))
```

```
    # índice de la clave a resaltar en rojo
    idx_hl = None
    if highlight_key is not None:
        try:
            idx_hl = cabecera.index(str(highlight_key))
        except ValueError:
            pass

    # función de formateo de cada celda
    def fmt(s, i):
        cell = f"{s:>{ancho_celda}}"
        if i == idx_hl:
            return f"{RED}{cell}{RESET}"
        if i in centro_idxs:
            return f"{YELLOW}{cell}{RESET}"
        return cell

    # línea de borde
    borde = "-" * ((ancho_celda + 1) * num_cols + 1)

    # --- 4) Impresión de la tabla ---
    print(f"\n******  {Titulo}  *****")
    print(borde)
    print(" ".join(fmt(c, i) for i, c in enumerate(cabecera)) + " │")
    print(borde)
    print(" ".join(fmt(d, i) for i, d in enumerate(fila_datos)) + " │")
    print(borde)


def imprimir_Nmedios(lista):
    #ANSI
    RED = "\033[1;31m"
    RESET = "\033[0m"

    # Formatear siempre como decimales de 4 dígitos
    vals_fmt = [f"{v:.4f}" for v in lista]
    # Encontrar índice del mínimo (numérico)
    min_idx = min(range(len(lista)), key=lambda i: lista[i])
    # Cabeceras (1,2,3,4,5)
    headers = [str(i) for i in range(1,6)]

    # Calcular ancho de celda
    ancho = max(max(len(h) for h in headers),
                max(len(v) for v in vals_fmt)) + 3

    # Borde
    borde = "-" * ((ancho + 1) * len(headers) + 1)

    # Función de formateo con color para el mínimo
    def fmt(s, idx):
        cell = f"{s:>{ancho}}"
        if idx == min_idx:
            return f"{RED}{cell}{RESET}"
        return cell

    # Imprimir
    print("\n***** Valores Numeros Medios *****")
    print(borde)
    # Headers
```

```
    print("│ " + " ".join(fmt(h, i) for i, h in enumerate(headers)) + " │")
    print(borde)
    # Valores
    print("│ " + "  ".join(fmt(v, i) for i, v in enumerate(vals_fmt)) + " │")
    print(borde)


def imprimir_tabla_N(titulo: str, data, es_decimal: bool = False, color_titulo: str = "default", blink: bool = False,
    light: bool = False):

    # ——— Configuración ANSI de colores/blink ———
    base = {
        "black": 30, "red": 31, "green": 32, "yellow": 33,
        "blue": 34, "magenta": 35, "cyan": 36, "white": 37,
        "default": 39
    }
    bright = light or color_titulo.startswith("light_")
    clave = color_titulo.replace("light_", "") if bright else color_titulo
    codigo_color = base.get(clave.lower(), base["default"])
    if bright:
        codigo_color += 60
    codigos = []
    if blink:
        codigos.append("5")
    codigos.append(str(codigo_color))
    seq_inicio = f"\033[{';'.join(codigos)}m"
    seq_fin    = "\033[0m"

    # ——— Extraer cabecera y valores ———
    if isinstance(data, dict):
        claves = list(data.keys())
        cabecera = [str(k) for k in claves]
        valores  = [data[k] for k in claves]
    elif isinstance(data, list):
        cabecera = [str(i) for i in range(len(data))]
        valores  = data
    else:
        print("Tipo no soportado (esperado dict o list).")
        return

    # ——— Formatear valores ———
    fila_datos = []
    for v in valores:
        if es_decimal and isinstance(v, float):
            fila_datos.append(f"{v:.4f}")
        else:
            fila_datos.append(str(v))

    # ——— Calcular anchos por columna (contenido vs. cabecera) + padding ———
    n = len(cabecera)
    anchos = []
    for i in range(n):
        ancho_max = max(len(cabecera[i]), len(fila_datos[i]))
        anchos.append(ancho_max + 2)   # +2 espacios de padding

    # ——— Construir líneas de borde ———
    # Ej: +----+------+----+
    partes = ["+" + "-" * a for a in anchos]
    borde = "".join(partes) + "+"
```

```
    # ——— Construir filas de texto ———
    # Cabecera: | key0 | key1 | ...
    cab = "|"
    datos = "|"
    for i in range(n):
        cab += f" {cabecera[i].center(anchos[i]-2)} |"
        datos += f" {fila_datos[i].center(anchos[i]-2)} |"

    # ——— Impresión final ———
    print()
    # título centrado sobre la tabla
    ancho_tabla = len(borde)
    print(seq_inicio + titulo.center(ancho_tabla) + seq_fin)
    print(borde)
    print(cab)
    print(borde)
    print(datos)
    print(borde)



def calcular_probabilidades_desde_historial(orden_digitos, historial_posiciones):

    # 2. Inicializamos un diccionario para contar apariciones, para cada dígito
    conteos = {digito: 0 for digito in orden_digitos}

    # 3. Recorrer el historial.
    #    Se asume que los números del historial son posiciones 1-indexadas.
    for pos in historial_posiciones:
        index = pos - 1  # Convertir a índice 0-indexado
        if 0 <= index < len(orden_digitos):
            digito = orden_digitos[index]
            conteos[digito] += 1
        else:
            print(f"Advertencia: posición {pos} fuera de rango en el historial.")
    #print("Prior Jerarquias")
    #print(conteos)

    # 4. Normalizamos los conteos para obtener probabilidades.
    total = sum(conteos.values())
    if total > 0:
        probabilidades = {digito: conteos[digito] / total for digito in conteos}
    else:
        # Si no hay registros en el historial, puede hacerse una distribución uniforme u otra política
        probabilidades = {digito: 0.005 for digito in conteos}

    return probabilidades



def inferir_probabilidades_bayesianas(orden_digitos, historial_posiciones):

    evidence_history = historial_posiciones[-40:]
    prior_history = historial_posiciones[:-40]
    print()
    # Inicializamos conteos para cada posición (usaremos posiciones 1 a N, donde N = len(orden_digitos))
    num_posiciones = len(orden_digitos)  # normalmente 10
    prior_counts = {pos: 0 for pos in range(1, num_posiciones + 1)}
```

```
    evidence_counts = {pos: 0 for pos in range(1, num_posiciones + 1)}

    # Contar ocurrencias en el prior
    for pos in prior_history:
        if 1 <= pos <= num_posiciones:
            prior_counts[pos] += 1
        else:
            print(f"Advertencia: posición {pos} fuera de rango en el prior.")

    # Contar ocurrencias en la evidencia
    for pos in evidence_history:
        if 1 <= pos <= num_posiciones:
            evidence_counts[pos] += 1
        else:
            print(f"Advertencia: posición {pos} fuera de rango en la evidencia.")

    # Aplicamos el modelo Bayesiano (conjugado Dirichlet):
    # La distribución posterior para la posición i es:
    # posterior(prob_i) = (prior_counts[i] + evidence_counts[i]) / (total_prior + total_evidence)
    total_prior = sum(prior_counts.values())
    total_evidence = sum(evidence_counts.values())  # debería ser 40 si todo está bien
    total_combined = total_prior + total_evidence

    posterior_probs = {}
    for pos in range(1, num_posiciones + 1):
        posterior_probs[pos] = (prior_counts[pos] + evidence_counts[pos]) / total_combined

    # Mostramos información de chequeo
    print("-- Prior counts (por posición) --")
    print(prior_counts)
    print("-- Evidence counts (últimas 40 jugadas) --")
    print(evidence_counts)
    print("-- Posterior (distribución de posiciones) --")
    print(posterior_probs)

    # Reconversión: asignamos la probabilidad calculada para cada posición
    # al dígito correspondiente segun el orden en orden_digitos.
    # Si la posición es 1 (1-indexado), corresponde a orden_digitos[0].
    final_probabilidades = {}
    for pos in range(1, num_posiciones + 1):
        digito = orden_digitos[pos - 1]
        final_probabilidades[digito] = posterior_probs[pos]

    return final_probabilidades


def ordenar_por_valor(d, ascendente=True):
    return dict(
        sorted(
            d.items(),
            key=lambda par: par[1],
            reverse=not ascendente
        )
    )

def mostrar_dict(d):
    for clave, valor in d.items():
        print(f"{clave}: {valor}")
```

```
def mostrar_formato(num):
    entero = int(num)
    decimal = round(num - entero, 4)
    dec_str = f"{decimal:.4f}"[2:]  # '1456', sin '0.'

    if num < 1:
        print(f".{dec_str}")
    else:
        print(f"{entero}.{dec_str}")

def Lista2_con_map(lista):
    """Aplica y = x//2 + 1 a cada elemento usando map+lambda."""
    return list(map(lambda x: x // 2 + 1, lista))

def Histo2_con_map(lista, T):
    """Aplica y = (x-1)//2 + 1 a cada elemento usando map+lambda."""
    return list(map(lambda x: (x-1) // T + 1, lista))

def ordenar_lista(lista: list, ascendente: bool = True) -> list:
    return sorted(lista, reverse=not ascendente)

def split_segments(H: List[int], n: int = 3) -> List[List[int]]:
    """Divide H en n trozos lo más parejos posible."""
    L = len(H)
    size = L // n
    segments = []
    for i in range(n-1):
        segments.append(H[i*size : (i+1)*size])
    segments.append(H[(n-1)*size : ])
    return segments

def compute_percentages(seg: List[int], possible: List[int]) -> Dict[int, float]:
    """
    Cuenta cuántas veces aparece cada valor en 'possible'
    dentro de 'seg' y devuelve porcentaje (0–1).
    """
    cnt = Counter(seg)
    total = len(seg) if seg else 1
    return {v: cnt.get(v, 0)/total for v in possible}

def mean_percentages(per_list: List[Dict[int, float]]) -> Dict[int, float]:
    """Dado un listado de dicts {v: pct}, devuelve su promedio por clave."""
    keys = per_list[0].keys()
    n = len(per_list)
    return {k: sum(d[k] for d in per_list)/n for k in keys}

def compute_fd_errors(
    mean_p: Dict[int,float],
    last_p: Dict[int,float],
    F_d:    Dict[int,int]
) -> Tuple[Dict[int,float], Dict[int,float]]:
    """
    Devuelve dos dicts:
      error_abs_fd[k] = abs(last_p[x] - mean_p[x])
      error_rel_fd[k] = (last_p[x] - mean_p[x]) / mean_p[x]
    donde x = F_d[k].
    """
    error_abs_fd = {}
```

```
    error_rel_fd = {}

    for k, x in F_d.items():
        # Si x no está en mean_p o last_p, saltamos o le ponemos 0
        m = mean_p.get(x)
        l = last_p.get(x)
        if m is None or l is None or m == 0:
            error_abs_fd[k] = None
            error_rel_fd[k] = None
        else:
            abs_err = abs(l - m)
            rel_err = (l - m) / m
            error_abs_fd[k] = abs_err
            error_rel_fd[k] = rel_err

    return error_abs_fd, error_rel_fd

def analyze_frecuencias(
    H: List[int],
    F_d: Dict[int,int],
    max_val: int,
    n_segments: int =3,
    last_n: int =45
) -> Tuple[Dict[int,float], Dict[int,float], Dict[int,float], Dict[int,float]]:
    """
    Retorna:
      mean_p   = promedio de los n_segments porcentajes históricos
      last_p   = porcentaje de los últimos last_n valores
      errors   = error absoluto por valor
      error_fd = asigna a cada clave de F_d su error correspondiente
    """
    possible = list(range(1, max_val+1))
    # 1) Segmentar y calcular porcentajes históricos
    segs = split_segments(H, n_segments)
    historical = [compute_percentages(s, possible) for s in segs]
    mean_p = mean_percentages(historical)

    # 2) Porcentaje últimos N
    last_seg = H[-last_n:]
    last_p = compute_percentages(last_seg, possible)

    # 3) Errores absolutos por valor
    error_abs_fd, error_rel_fd = compute_fd_errors(mean_p, last_p,F_d)

    # 4) Mapear errores según F_d
    error_fd = {k: error_abs_fd.get(k, None) for k in F_d.keys()}

    return mean_p, last_p, error_abs_fd, error_rel_fd, error_fd


def Dicc_probabilidad_ordenado(lista_numeros, ventanas=(15, 20, 20, 25), intervalo_inicial=0,
intervalo_final=10, cantidad=30):
    diccionarios = [Probabilidad_Caidas(lista_numeros, intervalo_inicial, intervalo_final, cantidad, ventana, True)
                    for ventana in ventanas]
    #return diccionarios

def sumar_diccionarios(*diccionarios, Divisor=1):
    diccionario_sumado = sumar_diccionarios(*diccionarios)
    return ordenar_por_valor(diccionario_sumado, ascendente=False)
```

```
def remapear_por_posicion(claves_ordenadas: list, dic_posiciones: dict)-> dict:
    print("Hi 5")
    resultado = {}
    n = len(claves_ordenadas)
    for pos, val in dic_posiciones.items():

        i = pos - 1
        print(f"  probando pos={pos} → i={i}, rango 0–{len(claves_ordenadas)-1}")
        if 0 <= i < n:
            resultado[claves_ordenadas[i]] = val
        else:
            # opcional: lanzar warning si hay posición fuera de rango
            # print(f"Warning: posición {pos} fuera de rango 1–{n}")
            pass
    return resultado



def main(file_path):
    Numeros = leer_datos_excel(file_path)
    Nume=Numeros.tolist()
    Nume2=Lista2_con_map(Nume)
    Histog=obtener_historial_caidas(Numeros)
    Sig_Histo=obtener_siguiente_caidas(Numeros)
    Sig_numeros = obtener_siguiente_numero(Numeros)
    #print(Sig_numeros)
    H2=Histo2_con_map(Histog,2)
    Sig_H2=Siguientes_lista(H2)
    F_datos=Semanas(Numeros)
    print(F_datos)
    Ultima_Jerarquia=ultima_jerarquia(Numeros)
    Prior1=calcular_alpha_prior(Numeros)
    Prior=ordenar_por_valor(Prior1, ascendente=False)
    print()

    Ultima_J_Sig=ultima_jerarquia_Lista(Sig_numeros)
    Ultima_Jer_Sig=ordenar_por_valor(Ultima_J_Sig, ascendente=False)
    #imprimir_tabla("Ultima Jerarquia Numeros Siguientes ", Ultima_Jer_Sig,es_decimal=False)
    PriorSig=calcular_alpha_prior_Lista(Sig_numeros)
    jerarquias, Posic = calcular_jerarquias(Numeros)
    print(Ultima_Jerarquia)
    claves_ordenadas = sorted(Ultima_Jerarquia.keys(), key=lambda k: (Ultima_Jerarquia[k], -F_datos[k]))

    # Imprimir el resultado ordenado
    print()
    print("Orden de jerarquías:")
    for k in claves_ordenadas:
        print(f"Id: {k}\t Repet: {Ultima_Jerarquia[k]}\t Aparición: {F_datos[k]}")
    print()

    # Diccionario de ranking basado en claves ordenadas
    ranking_dict = {rank: clave for rank, clave in enumerate(claves_ordenadas[:10])}

    #Procesamiento probabilidades ultimas
    #DiccNu = Dicc_probabilidad_ordenado(Nume)

    #DiccSig = Dicc_probabilidad_ordenado(Sig_numeros)
```

```
    #DiccJer = Dicc_probabilidad_ordenado(Posic,(15, 20, 25), intervalo_inicial=1, intervalo_final=10,
cantidad=30)
    #imprimir_tabla("Probabilidad Numeros con 15, 20 y 25 ", DiccNu, es_decimal=True)
    #imprimir_tabla("Probabilidad Siguientes Numeros con 15, 20 y 25 ", DiccSig, es_decimal=True)
    #DicJ=remapear_por_posicion(claves_ordenadas, DiccJer)
    #imprimir_tabla("Probabilidad Jerarquia ", DicJ, es_decimal=True)
    #DicT=sumar_diccionarios(DiccSig, DiccNu, DiccNu, DiccNu, DiccNu, DicJ, Divisor=6)
    #DicT=sumar_diccionarios(DiccSig, DiccNu, DiccNu, DiccNu, DiccNu, Divisor=5)
    print()

    # Procesamiento para "Numeros"
    Pr_Num2, _, _,  _ = procesar_e_imprimir_regresion("Numeros Medios", Nume2, 1, 6,3,2,4)
    #imprimir_Nmedios(Pr_Num2)
    print()
    Pr_Num, _, _,  _ = procesar_e_imprimir_regresion("Numeros", Numeros)
    Pr_Sig, _, _,  _ = procesar_e_imprimir_regresion("Siguientes", Sig_numeros)
    Pr_Pos_val, Pr_Pos_err, Sum14, PromGral = procesar_e_imprimir_regresion("Jerarquía", Posic)

    # Ahora generamos los valores correctos usando ranking_dict
    nuevos_valores_dict = {}
    errores_dict = {}

    for rank, clave in ranking_dict.items():
        nuevo_valor = (Sum14 + rank) / 15
        error = (nuevo_valor - PromGral) / PromGral
        if error < 0:
            error *= -0.99
        nuevos_valores_dict[clave] = nuevo_valor
        errores_dict[clave] = error
    print()

    # Para el Total final: queremos que la “columna” Pr_Pos sea la lista de errores reordenada por clave (en
orden ascendente)
    # Es decir, obtenemos las claves (numéricas) del ranking en orden ascendente:
    sorted_keys = sorted(ranking_dict.values())  # esto da [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    Pr_Pos_err_ordered = [errores_dict[k] for k in sorted_keys]

    # Códigos ANSI para rojo y reset
    RED   = "\033[31m"
    RESET = "\033[0m"
    # Calcula los mínimos
    min_jer = min(errores_dict.values())
    min_num = min(Pr_Num)
    min_num2 = min(Pr_Num2)
    min_sig = min(Pr_Sig)

    print("\tJerarquías\tNumeros\t\t\tSiguientes\tError Num")
    print("**************************************************************************")
    ErrorNUm={}
    for k in sorted_keys:
        jer = errores_dict[k]
        num = Pr_Num[k]
        sig = Pr_Sig[k]
        ErrorNUm[k] = (sig + 4*num + 3*jer) / 8

        # Formatea cada celda, poniendo rojo si coincide con el mínimo
        s_jer = f"{jer:.4f}"
        if jer == min_jer:
            s_jer = f"{RED}{s_jer}{RESET}"
```

```
        s_num = f"{num:.4f}"
        if num == min_num:
            s_num = f"{RED}{s_num}{RESET}"

        s_sig = f"{sig:.4f}"
        if sig == min_sig:
            s_sig = f"{RED}{s_sig}{RESET}"

        print(f"{k}\t{s_jer}\t\t{s_num}\t\t{s_sig}\t\t{ErrorNUm[k]:.4f}")

    ErrorOrdenado=ordenar_por_valor(ErrorNUm, ascendente=True)
    print()
    imprimir_tabla("Errores Promedios Numeros ", ErrorOrdenado, es_decimal=True)

    # Imprimir resultados ajustados según el ranking (en orden de ranking)
    print("\nResultados ajustados de 'Jerarquía' reordenados mediante ranking_dict:")
    for rank in sorted(ranking_dict.keys()):
        clave = ranking_dict[rank]
        print(f"Id {rank}:\tNum {clave},\t Prom15: {nuevos_valores_dict[clave]:.4f},\t Er: {errores_dict[clave]:.4f}")
    print()

    Pb_Num = analizar_siguientes_numeros_para_probabilidades(Nume)
    Jer_orde=ordenar_por_valor(Pb_Num, ascendente=False)
    imprimir_tabla("Probabilidades Numeros Bayes ", Jer_orde, es_decimal=True)
    print()

    Pb_Sig = analizar_siguientes_numeros_para_probabilidades(Sig_numeros)
    Jer_orde=ordenar_por_valor(Pb_Sig, ascendente=False)
    imprimir_tabla("Probabilidades Siguientes Bayes ", Jer_orde, es_decimal=True)
    print()

    Pb_jerarquia = calcular_probabilidades_desde_historial(claves_ordenadas,Posic)
    Jer_orde=ordenar_por_valor(Pb_jerarquia, ascendente=False)
    imprimir_tabla("Probabilidades Jerarquia Bayes ", Jer_orde, es_decimal=True)
    print()

    print("\tJerarquias\tNumeros\t\tSiguientes\tTotal")
    Pb_por_Numero = {}
    for num in Pb_Sig:
        # Se asume que la clave también existe en Probab_Jerar_bayes
        Pb_por_Numero[num] = (Pb_Sig[num] + 3*Pb_Num[num]+5*Pb_jerarquia[num]) / 9
        if num % 2==0:

print("\033[33m"+f"{num}\t{Pb_jerarquia[num]:.4f}\t\t{Pb_Num[num]:.4f}\t\t{Pb_Sig[num]:.4f}\t\t{Pb_por_Numero[n
um]:.4f}" + "\033[0m")
        else:

print("\033[34m"+f"{num}\t{Pb_jerarquia[num]:.4f}\t\t{Pb_Num[num]:.4f}\t\t{Pb_Sig[num]:.4f}\t\t{Pb_por_Numero[n
um]:.4f}" + "\033[0m")

    print()
    imprimir_tabla("Errores Prom. Ordenados Numeros Siguientes Jerarquia ", ErrorOrdenado, es_decimal=True)
    print("---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---")

    ProNUm=ordenar_por_valor(Pb_por_Numero, ascendente=False)
    imprimir_tabla("Prob. Bayes Ordenadas Numeros Siguientes Jerarquia  ", ProNUm, es_decimal=True)
    print()
```

```
    Probab_mayor = aplicar_regresion_logistica_mayor_menor(Numeros)
    if Probab_mayor is not None:
        print(f"\nProbabilidad (Reg. Logística) de que el siguiente número sea mayor que 4: {Probab_mayor:.4f}")

    Probab_par = aplicar_regresion_logistica_par_impar(Numeros)
    if Probab_par is not None:
        print(f"Probabilidad (Reg. Logística) de que el siguiente número sea par: {Probab_par:.4f}")
    print()

    Histo_nuevo = Histo2_con_map(Histog,3)
    F_datos_modificado = {k: (v - 1) // 3 + 1 for k, v in F_datos.items()}
    F_datos_2 = {k: (v - 1) // 2 + 1 for k, v in F_datos.items()}
    Error_val2, Nuevo_valor2, Sum14=procesar_regresion_Histo("Histograma", Histo_nuevo,
F_datos_modificado,4,2,6)
    CaidasOrdenadas=ordenar_por_valor(F_datos_modificado, ascendente=True)

    PromOrdenados=ordenar_por_valor(Error_val2, ascendente=True)
    llave_max = min(PromOrdenados, key=PromOrdenados.get)
    imprimir_tabla("Caidas ", CaidasOrdenadas, es_decimal=False, highlight_key=llave_max)
    imprimir_tabla("Promedio Histograma 1/3 ", PromOrdenados, es_decimal=True)

    mean_p, last_p, errors, error_abs_fd, error_rel_fd= analyze_frecuencias(
        Histog, F_datos, max_val=40, n_segments=3, last_n=45
    )

    mean_p2, last_p2, errors2, error_abs_fd2, error_rel_fd2= analyze_frecuencias(
        H2, F_datos_2, max_val=20, n_segments=3, last_n=35
    )
    imprimir_tabla("Probabilidad Semanas Caidas  ", error_abs_fd, es_decimal=True)
    imprimir_tabla("Probabilidad Semanas  20 ", error_abs_fd2, es_decimal=True)
    print("                          *************          ***********")



    Error_val, Nuevo_valor, Sum14=procesar_regresion_Histo("Histograma", Histog, F_datos)
    CaidasOrdenadas=ordenar_por_valor(F_datos, ascendente=True)
    PromOrdenados=ordenar_por_valor(Error_val, ascendente=True)
    llave_max = min(PromOrdenados, key=PromOrdenados.get)
    #print("LLave max ",llave_max)
    imprimir_tabla("Caidas ", CaidasOrdenadas, es_decimal=False, highlight_key=llave_max)
    imprimir_tabla("Promedio Histograma ", PromOrdenados, es_decimal=True)
    print()



    Error_val1, Nuevo_valor, Sum14=procesar_regresion_Histo("Histograma 2", H2, F_datos_2)
    CaidasOrdenadas=ordenar_por_valor(F_datos_2, ascendente=True)
    PromOrdenados1=ordenar_por_valor(Error_val1, ascendente=True)
    llave_max = min(PromOrdenados1, key=PromOrdenados1.get)
    imprimir_tabla("Caidas ", CaidasOrdenadas, es_decimal=False, highlight_key=llave_max)
    imprimir_tabla("Promedio Histograma ", PromOrdenados1, es_decimal=True)


    Error_val2, Nuevo_valor, Sum14=procesar_regresion_Histo("Sig. Histograma", Sig_Histo, F_datos)
    CaidasOrdenadas=ordenar_por_valor(F_datos, ascendente=True)
    PromOrdenados2=ordenar_por_valor(Error_val2, ascendente=True)
    llave_max = min(PromOrdenados2, key=PromOrdenados2.get)
    imprimir_tabla("Caidas ", CaidasOrdenadas, es_decimal=False, highlight_key=llave_max)
```

```
    imprimir_tabla("Promedio Histograma ", PromOrdenados2, es_decimal=True)



    Bayes_Histo=porcentaje_coincidencias(F_datos, Histog)
    PromOrdenados=ordenar_por_valor(Bayes_Histo, ascendente=False)

    imprimir_tabla("PORCENTAJE caidas Semanas", PromOrdenados, es_decimal=True)
    imprimir_Nmedios(Pr_Num2)
    print()
    imprimir_tabla("Errores Prom. Ordenados Numeros Siguientes Jerarquia ", ErrorOrdenado, es_decimal=True)
    print()
    ProNUm=ordenar_por_valor(Pb_por_Numero, ascendente=False)
    imprimir_tabla("Prob. Bayes Ordenadas Numeros Siguientes Jerarquia  ", ProNUm, es_decimal=True)
    print()
    #imprimir_tabla("Probabilidad Total con 15, 20 y 25 ", DicT, es_decimal=True)



if __name__ == "__main__":
    print("Hello World")
    file_path = 'D:/loter.xlsx'
    main(file_path)

    #DiccHS15={}
    #DiccH15=Probabilidad_Caidas(Histog,1,41,60,40, True)
    #imprimir_tabla("Probabilidad Numeros con 15 ", DiccS15, es_decimal=True)
    #DiccH20={}
    #DiccH20=Probabilidad_Caidas(Histog,1,41,50,30, True)
    #imprimir_tabla("Probabilidad Numeros con 20 ", DiccS20, es_decimal=True)
    #DiccH25={}
    #DiccH25=Probabilidad_Caidas(Histog,1,41,40,25, True)
    #imprimir_tabla("Probabilidad Numeros con 25 ", DiccS25, es_decimal=True)
    #SDicH=sumar_diccionarios(DiccH15, DiccH20, DiccH25)
    #DicHi=ordenar_por_valor(SDicH, ascendente=False)
    #imprimir_tabla("Probabilidad Numeros con 15, 20 y 25 ", DicHi, es_decimal=True)




    #modelo_ajustado = validate_residuals(lista_30, lista_15, lista_6, lista_sig)
    #graficar_residuos(lista_30, lista_15, lista_6, lista_sig)
    #reporte = reporte_regresiones(lista_30, lista_15, lista_6, lista_sig)

    #print("Procesando regresión bayesiana con PyMC...")
    #resultado_bayes = prediccion_bayesiana(lista_30, lista_15, lista_6, lista_sig)
    #print("Predicción media (modelo bayesiano):", resultado_bayes["prediccion_media"])
    #print("Intervalo 98% de credibilidad:", resultado_bayes["int_95"])
```