PRÁCTICA FINAL:
Predicción de la Demanda de Energía en Francia

APLICACIONES DEL SOFT-COMPUTING EN ENERGÍA, VOZ E IMAGEN

Álvaro Zamorano Ortega, Gabriel López Cuenca

Contenido

Enunciado

La práctica consiste en la predicción de la demanda total de energía a nivel nacional (datos de Francia en este caso) a partir de variables macroeconómicas. En este caso en particular hay 6 variables macroeconómicas para hacer la predicción (población, GPD, emisiones en CO2, producción eléctrica, consumo eléctrico y porcentaje de población sin empleo), con datos desde 1990 hasta 2016.

Las variables macroeconómicas deberán estar normalizadas, es decir, cada dato deberá ser dividido entre el dato de más valor de su mismo tipo.

El procedimiento para realizar la predicción involucra un algoritmo metaheurístico para optimizar los pesos de un modelo de tipo exponencial. El modelo de predicción, el cual emplea 13 variables predictivas (debido a las 6 variables macroeconómicas), es el siguiente:

$$\hat{E}(t+1) = \sum_{i=1}^{T} w_i X_i(t)^{w_{i+m'}} + w_0$$

Los pesos están normalizados entre -1 y 1, excepto w_0 que está entre -5 y 5.

A partir de este modelo de predicción se debe minimizar la función objetivo, en este caso el valor cuadrático medio entre la predicción y los valores reales de energía:

$$f(\mathcal{M}) = \frac{1}{n^*} \sum_{j=1}^{n^*} \left(E(j) - \hat{E}(j) \right)^2$$

En nuestro caso, el algoritmo metaheurístico es el **Harmony Search** y éste usará la energía de los años pares (variables macroeconómicas de los años impares) para ajustar los pesos, quedando la energía de los años impares (variables de los años pares) para la parte de test.

Proceso

En primer lugar, se han introducido las variables macroeconómicas en un archivo con extensión **.csv** para una mejor carga de datos en Matlab.

Seguidamente se ha realizado una función para **normalizar** los datos de este fichero. Para ello se ha recorrido cada columna de datos y se han dividido todos ellos entre la variable de mayor valor de la columna. Finalmente, esta función devuelve los datos normalizados y un array de factores por los que se ha dividido cada columna, es decir, el máximo de cada una de ellas. Dicha función es la siguiente:

```
function [datosNormalizados, factor] = normalizarDatos(datos)
    tam = size(datos);
    %Meter los años en la primera columna
    resultado = datos(:,1);

    factores = [];
    %Normalizamos cada columna
    for i=2:tam(2)
        columna = datos(:,i);
        maximo = max(columna);
        resultado = [resultado, columna./maximo];
        factores = [factores, maximo];
    end

    datosNormalizados = resultado;
    factor = factores;
end
```

Posteriormente, se han **separado** los datos de los años pares e impares. En nuestro caso, se han utilizado las variables macroeconómicas de los años impares para ajustar los pesos, por lo que las energías de los años pares se han utilizado para la función objetivo. Ambos conjuntos de datos se han guardado en varTrain y enTrain, respectivamente. Asimismo, en los arrays varTest y enTest se han guardado las variables macroeconómicas de los años pares y las energías de los años impares. Dichas funciones son las siguientes:

```
function [varTrain, enTrain] = separarDatosTrain(datosNormalizados)
    % Variables de años impares
    varTrain = datosNormalizados(2:2:end,:);
    varTrain = varTrain(:,3:end);

    % Resultado de años pares
    enTrain = datosNormalizados(3:2:end,:);
    enTrain = enTrain(:,2);
end
```

```
function [varTest, enTest] = separarDatosTest(datosNormalizados)
    % Variables de años pares
    varTest = datosNormalizados(1:2:end,:);
    varTest = varTest(:,3:end);
    varTest = varTest(1:(length(varTest)-1),:);

    % Resultado de años impares
    enTest = datosNormalizados(2:2:end,:);
    enTest = enTest(:,2);
end
```

Por tanto, se usará la energía para el entrenamiento desde el año 1992 hasta el 2016, y para la prueba desde el 1991 hasta el 2016.

Por otro lado, para el cálculo del **modelo predictivo** se han usado dos funciones que se llaman entre sí. La primera de ellas hace uso de la segunda y su función es construir el array de demandas calculadas. La segunda función, de acuerdo con la ecuación del modelo, y a las variables macroeconómicas y los diferentes pesos en un cierto momento de la ejecución, obtiene la energía para el año siguiente al que se correspondan las variables.

```
function demandasCalculadas = calcularDemanda(varReales, varPredictivas)
    conjuntoCalculado = [];
    tam = length(varReales);
    for i=1:tam
        conjuntoCalculado = [conjuntoCalculado;
cDemandaAux(varReales(i,:),varPredictivas)];
    end

    demandasCalculadas = conjuntoCalculado;
end

function demandaCalculada = cDemandaAux(varReales, varPredictivas)
    %Las varPredictivas son los pesos de la armonía
    suma = 0;
    tam = length(varReales);
    for i=2:tam
        suma = suma +
varPredictivas(i)*(varReales(i)^(varPredictivas(i+tam)));
    end

    %La bia se encuentra en la primera posición del vector
    demandaCalculada = suma + varPredictivas(1);
end
```

Se hará uso de la primera función, la cuál devuelve un array con las energías calculadas para todos los años, con el fin de calcular el valor de la función objetivo, es decir, el error cometido en el cálculo, y poder minimizarlo mediante el ajuste de los diferentes pesos. El error se obtiene a través de la siguiente función:

```
function error = funcionMinimizar(enReal, enCalculada)
    suma = 0;
    tam = length(enReal);
    for i=1:tam
        suma = suma + (enReal(i) - enCalculada(i))^2;
    end

    error = suma/tam;
end
```

Harmony Search

Previo a la explicación del algoritmo se explicarán funciones creadas únicamente para el mejor funcionamiento y comprensión de la ejecución.

Comenzamos por crear la memoria de **armonías inicial**. Está representada mediante un array de celdas donde cada una de las celdas contiene un vector de 13 pesos aleatorios, es decir, una bia más dos pesos para cada característica (multiplicador y exponente), dichos pesos se encuentran en el rango dado con el problema. El número de celdas será elegido arbitrariamente por el

usuario. En caso de querer linealizar el modelo se indica mediante un booleano y el vector de pesos (sin la bia) será tratado en otra función.

```
function armoniaInicial = crearArmoniaInicial(nCaracteristicas, nPadres,
linealizar)
    varPredictivas = cell(1,nPadres);
    for i=1:nPadres
        vector = -1 + 2*rand(1,nCaracteristicas*2);

        % Los exponentes del modelo se ponene a 1
        if linealizar
           vector = linealizarPesos(vector,nCaracteristicas);
        end

        w0 = -5 + 10*rand;
        varPredictivas{i} = [w0, vector];
    end

    armoniaInicial = varPredictivas;
end
```

La función que **linealiza** el modelo tiene el siguiente aspecto:

```
function pesosLinealizados = linealizarPesos(pesos,nCaracteristicas)
    longitudPesos = length(pesos);

    if (mod(length(pesos),2) == 0)
        suma = 0;
    else
        suma = 1;
    end

    %La bia y los pesos se deben conservar
    indicePrimerExponente = (longitudPesos/2)+suma;
    rangoAConservar = 1:indicePrimerExponente;

    pesosLinealizados = [pesos(rangoAConservar), ones(1,nCaracteristicas)];
end
```

Su funcionamiento se corresponde con encontrar donde comienzan los pesos usados como exponentes, esto es a partir de la posición 6 o 7, en caso de que se no se tenga en cuenta la bia o si, respectivamente, y a partir de ahí completa el vector con 1.

Además, cuando se muta una cierta nota es necesario comprobar que el valor generado no sobrepasa los límites anteriormente mencionados. Si esto es así, dicho valor será el límite que sobrepase.

```
function valor = comprobarValor(valorInicial, minimo, maximo)
    if(valorInicial<minimo)
        valor = minimo;
    else
        if(valorInicial>maximo)
            valor = maximo;
        else
            valor = valorInicial;
        end
    end
end
```

La última función auxiliar tiene como función principal comprobar que los 13 pesos ajustados durante la ejecución del algoritmo no sobrepasen los **límites** indicados. Dichos límites son: [-5,5] para la bia, y [-1,1] para el resto de los pesos. La función recibe el índice del array de pesos que se está tratando y devuelve los valores máximo y mínimo que en esa posición puede haber.

```
function [minimo,maximo] = limitesPesos(indice)
    %El primer indice siempre será w0
    if (indice==1)
       maximo = 5;
       minimo = -5;
    else
       maximo = 1;
       minimo = -1;
    end
end
```

Algoritmo Harmony Search

En la improvisación musical, cada músico toca una nota dentro de un posible rango, de tal manera que forman un vector armónico. Si el conjunto de notas tocadas por los músicos es considerado una buena armonía, esta es guardada en la memoria de cada músico, incrementando la posibilidad de hacer una buena armonía la próxima vez. Del mismo modo en el proceso de optimización en ingeniería, cada variable de decisión inicialmente toma valores aleatorios dentro del rango posible, formando un vector solución. Si dicho vector, es decir, dicho conjunto de valores que lo conforman son una buena solución, esta es almacenada en la memoria de cada variable, aumentando la posibilidad de encontrar mejores soluciones en la siguiente iteración.

El algoritmo consta de 3 pasos fundamentales para generar una nueva armonía a partir de una memoria de armonías.

- **HMCR**: consiste en la generación de una nueva harmonía (hijo). Cada nota (elemento del hijo) se genera independientemente, de manera que el HMCR es una probabilidad de que dicha nota sea obtenida de la Harmony Memory (matriz de padres). Si la probabilidad aleatoria es menor que HMCR, la nota será una nota aleatoria contenida en la columna que se está tratando de la Harmony Memory. En caso contrario, se genera una nota aleatoria en el dominio que corresponda. En nuestro caso HMCR es 0.8.
- **PAR**: consiste en la mutación de la nota en base a una probabilidad. En nuestro caso la mutación se hace a través de un operador gaussiano, su probabilidad es 0.3.
- **RSR**: algunas notas son reemplazadas por otras generadas de forma aleatoria, dependiendo también de una cierta probabilidad. La probabilidad escogida ha sido de 0.01.

Finalmente, la nueva armonía se compara con la peor del Harmony Memory, y si es mejor que ésta, la sustituye.

La ejecución del algoritmo se ha dividido en 3 grandes funciones. La primera de ellas realiza el bucle principal, es decir, mientras no se llegue al máximo de iteraciones:

1. Cálculo del **error** producido por cada una de las armonías.
2. Obtención del error **mínimo** (hasta el momento será el mejor conseguido).
3. Almacenamiento del mejor **fitness** conseguido en cada iteración.
4. Generación de una nueva **armonía**.
5. **Comparación** de la nueva armonía con la peor que hay en la memoría de armonías.

a. Sustitución en caso de que la generada sea mejor.

6. Asignación del **mejor** de los padres (el que menor error genera) como solución parcial.

```
function [allFitness,mejorFitness,solucion] =
algoritmoHS(varTrain,enTrain,padres,maxIteraciones,varianza,linealizar,nCara
cteristicas)
    %Calcular el error producido por todos los padres
    errorPadres = evaluarAux(padres,varTrain,enTrain);

    %Obtener el error minimo
    minimoPadres = min(errorPadres);

    %El mejor fitness conseguido hasta el momento es el mínimo
    mejorFitness = minimoPadres;

    %Vector para guardar los mejores fitness conseguidos durante la
    %ejecución
    allFitness = [];

    %Realizamos el algoritmo hasta el máximo de iteraciones indicado
    for j=1:maxIteraciones

        %Guardamos el fitness del mejor padre
        allFitness = [allFitness; mejorFitness];

        %Generamos una nueva armonia
        hijo = generarHijoRSR(padres,varianza);

        % Los exponentes del modelo se ponene a 1
        if linealizar
           hijo = linealizarPesos(hijo,nCaracteristicas);
        end

        %Calculamos el error del hijo
        enCalculadaHijo = calcularDemanda(varTrain, hijo);
        errorHijo = funcionMinimizar(enTrain, enCalculadaHijo);

        %Obtenemos el error maximo de los padres y su posición
        [valorPeorPadres, posPeorPadre]= max(errorPadres);

        %Si el hijo generado es mejor que el peor padre, se cambian
        if(valorPeorPadres>errorHijo)
            %Cambio
            padres{posPeorPadre} = hijo;

            %Calculamos otra vez los valores de los nuevos padres
            errorPadres = evaluarAux(padres,varTrain,enTrain);

            %Obtenemos el error mínimo y la posición del padre que lo
            %genera
            [minimoPadres, posMinimoPadres] = min(errorPadres);
            mejorFitness = minimoPadres;

            %La solución hasta el momento es el padre cuyo error producido
            %es el menor
            solucion = padres{posMinimoPadres};
        end
    end
end
```

La relización del 4º paso se realiza en una función que es llamada por esta y su funcionamiento es el de realizar los 3 pasos fundamentales para generar una nueva armonía en Harmony Search, es decir, HMCR, PAR y RSR.

```
function [hijo] = generarHijoRSR(padres,varianza)
    %Creación de un hijo vacío
    hijo = [];

    %Tamaño de un solo padre
    tam = size(padres{1});
    tam = tam(2);

    %Conocer el número de padres totales
    tamColumnas = size(padres);
    tamColumnas = tamColumnas(2);

    for i = 1:tam
        %Obtener límites para los pesos aleatorios
        [minimo,maximo] = limitesPesos(i);

        %Obtener probabilidad para HMCR
        probHMCR = rand;

        %Probabilidad HMCR para generar nota
        if(probHMCR<0.8)
            %Nota aleatoria de la misma columna de Harmony Memory
            numAleatorioFila = randi(tamColumnas);
            %Coger fila completa aleatoria
            fila = padres{numAleatorioFila};
            %Valor de la fila a introducir en el hijo
            valor = fila(i);
        else
            %Nota aleatoria en el rango de la misma columna
            valor = minimo + (maximo-minimo)*rand;
        end

        %Obtener probabilidad para PAR
        probPAR = rand;

        if(probPAR<0.3)
            %Mutamos nota
            ruido = 0 + varianza*rand;
            valor = valor + ruido;

            %Comprobar que el valor no se sale de los límites del peso
            valor = comprobarValor(valor, minimo, maximo);
        end
        hijo = [hijo,valor];
    end

    %Obtener probabilidad para RSR
    probCambio = rand;

    if(probCambio<0.01)
        %Reemplazamos num aleatorio de notas por un num aleatorio en el
        %rango de su misma columna
        for i=1:randi(tam)
            posicionAleatoria = randi(tam);

            %Obtener límites para los pesos aleatorios
            [minimo,maximo] = limitesPesos(posicionAleatoria);

            valor = rand*(maximo-minimo)+minimo;
            hijo(posicionAleatoria) = valor;
        end
    end
end
```

Por último, para calcular el **error** producido por cada una de las armonías (primer paso del bucle), se tiene una función auxiliar encargada de llamar a la función que modeliza la función objetivo (explicada con anterioridad).

```
function errores = evaluarAux(padres,varTrain,enTrain)
    errores = [];
    tam = size(padres);

    for i=1:tam(2)
        enCalculada = calcularDemanda(varTrain, padres{i});
        errores = [errores, funcionMinimizar(enTrain, enCalculada)];
    end
end
```

Todo ello se ejecuta a través del siguiente **script** de Matlab:

```
datos = csvread('data.csv',1,0);

[datosNormalizados, factoresNormalizado] = normalizarDatos(datos);
factorNormalizadoEnergia = factoresNormalizado(1);

annosTrain = 1992:2:2016;
annosTest = 1991:2:2015;

[varTrain, enTrain] = separarDatosTrain(datosNormalizados);
[varTest, enTest] = separarDatosTest(datosNormalizados);

% Linealizar Modelo (true/false)
linealizar = true;

nCaracteristicas = 6;
nPadres = 10;
armoniaInicial = crearArmoniaInicial(nCaracteristicas, nPadres, linealizar);

maxIteraciones = 10000;
varianza = 0.01;
evolucionFitness = [];

errorMinimo = 5;

for i=1:10
    [allFitness,mejorFitness,solucion] =
algoritmoHS(varTrain,enTrain,armoniaInicial,maxIteraciones,varianza,linealizar,nCaracterist
icas);

    if(mejorFitness<errorMinimo)
        errorMinimo = mejorFitness;
        solucionMejor = solucion;
        evolucionFitness = allFitness;
    end
end
disp(errorMinimo*factorNormalizadoEnergia^2);
enCalculadaTrain = calcularDemanda(varTrain, solucionMejor);

enCalculadaTest = calcularDemanda(varTest, solucionMejor);
errorTest = funcionMinimizar(enTest, enCalculadaTest);
disp(errorTest*factorNormalizadoEnergia^2);

figure;
subplot(2,1,1)
plot(annosTrain,enTrain.*factorNormalizadoEnergia,'-
+',annosTrain,enCalculadaTrain.*factorNormalizadoEnergia,'-o');
title('TRAIN');
legend('Real','Calculada','Location','south');

subplot(2,1,2)
plot(annosTest,enTest.*factorNormalizadoEnergia,'-
+',annosTest,enCalculadaTest.*factorNormalizadoEnergia,'-o');
title('TEST');
legend('Real','Calculada','Location','south');

figure;
plot(evolucionFitness*factorNormalizadoEnergia^2);
title("EVOLUCIÓN FITNESS");
```

Resultados

En primer lugar, se ha ejecutado el algoritmo 10 veces con un total de 100000 iteraciones, 50 armonías en el Harmony Memory y una varianza de 0.1, dando los siguientes resultados:

- Error en Train: 33.0539
- Error en Test: 38.9539
- Gráficas:

En segundo lugar, se ha ejecutado el algoritmo 10 veces con un total de 1000000 iteraciones, 50 armonías en el Harmony Memory y una varianza de 0.1, dando los siguientes resultados:

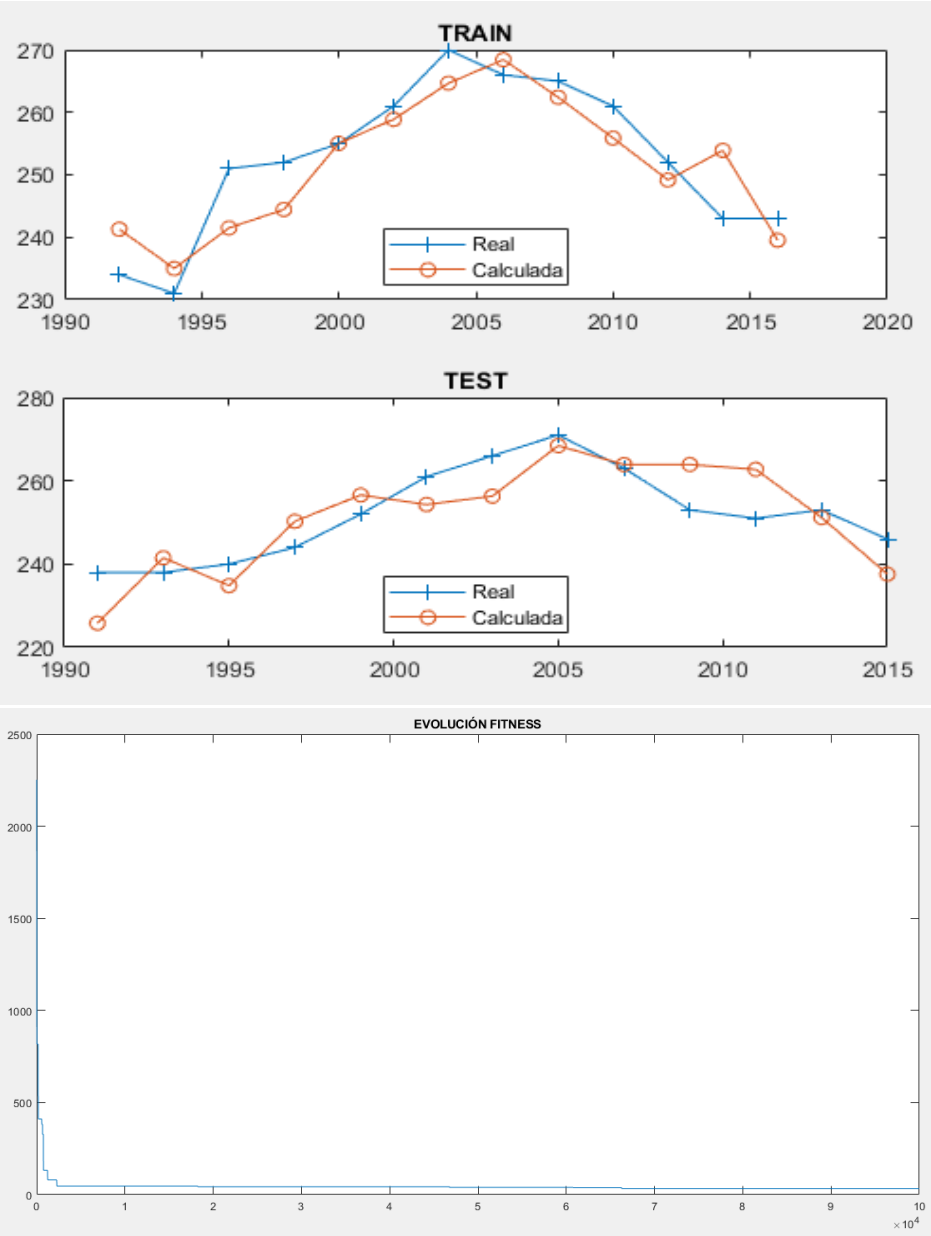
- Error en Train: 31.3909
- Error en Test: 60.3552
- Solución: 1.03395022234104, -0.0396830409104649, 0.502554494152303, -0.760054757541465, 0.195101103423410, 0.0943908597774077,

0.549589778634588, -0.778306935062829, 1, -0.364925961006354, 0.388576828557323, 0.367450208421355, 0.674018527704088

- Gráficas:

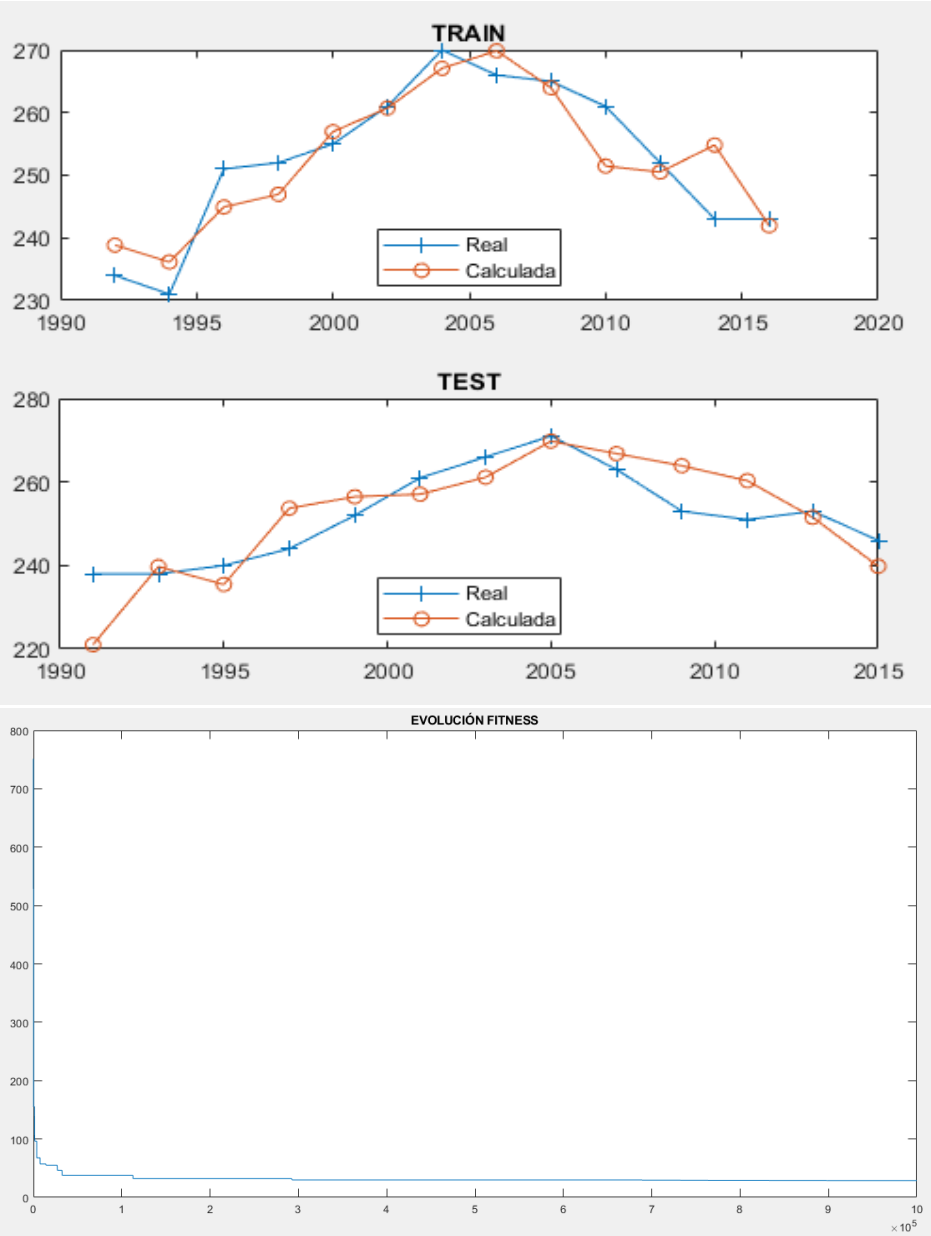
En tercer lugar, se ha ejecutado el algoritmo 10 veces con un total de 100000 iteraciones,100 armonías en el Harmony Memory y una varianza de 0.005, dando los siguientes resultados:

- Error en Train: 32.8115
- Error en Test: 55.8012
- Solución: 1.02122038776458, -0.0563892219210374, 0.630940079872278, 0.0141610702172770, -0.330131311278730, -0.255221353875998, -0.458435714656607, -0.655266561254742, 0.821445995181982, 0.271459417590469, -0.651273482472148, -0.0960520936376637, -0.493543192859249
- Gráficas:



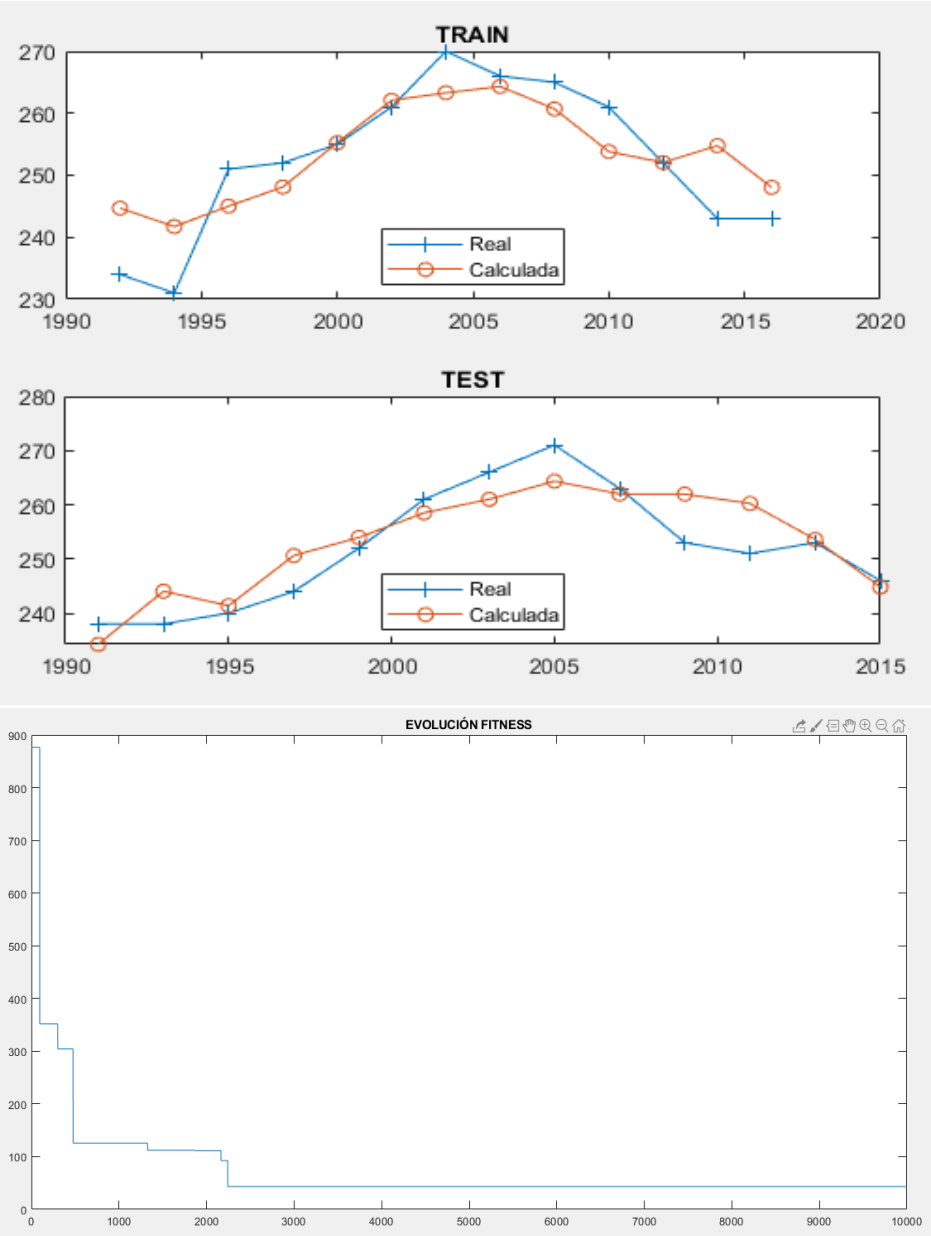
En cuarto lugar, se ha ejecutado el algoritmo 10 veces con un total de 1000000 iteraciones,100 armonías en el Harmony Memory y una varianza de 0.005, dando los siguientes resultados:

- Error en Train: 28.8345
- Error en Test: 55.8824
- Solución: 2.28939042829409, -0.0314911036115602, -0.458475832939426, -0.615181173974046, -0.108492629614068, -0.0629958594475037, -0.0997430085650631, -0.984259851891381, -0.887695275708103, -0.644775459494193, 0.867796166374109, -0.378100237733892, -0.176517227088102
- Gráficas:



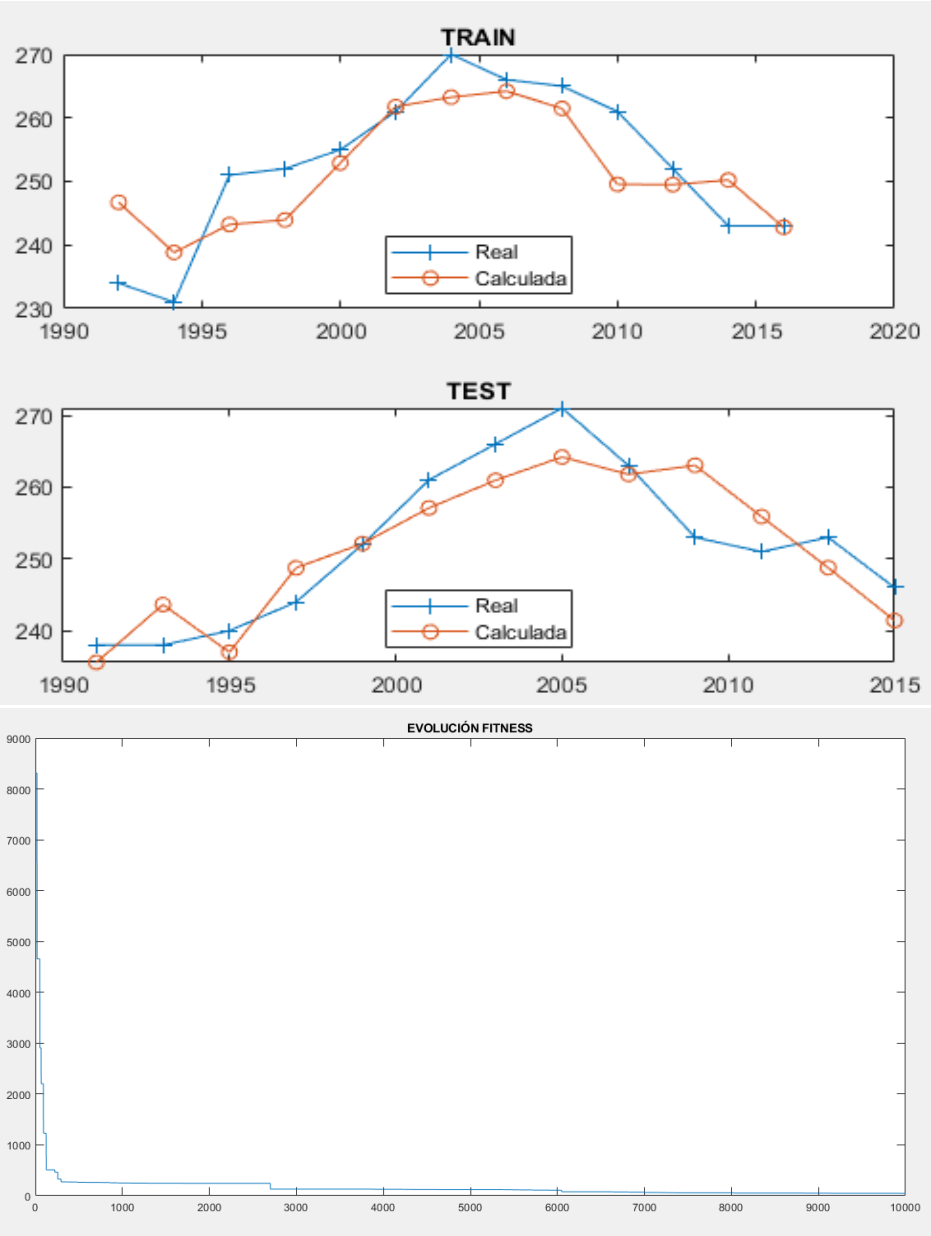
En quinto lugar, se ha ejecutado el algoritmo 10 veces con un total de 10000 iteraciones,50 armonías en el Harmony Memory y una varianza de 0.2, dando los siguientes resultados:

- Error en Train: 43.3610
- Error en Test: 26.4610
- Solución: 0.837356978814644, -0.452818745147446, -0.224005678519026, 0.275721434213902, -0.265890939487795, 0.808408433517817, -0.635541094639752, 0.0190935735087033, -0.879114439809641, 0.661089943110297, -0.498612061100123, -0.0225832859890875, 0.453546170439069
- Gráficas:



Por último, se ha realizado el mismo algoritmo, pero con la función modelo linealizada, es decir, con los exponentes a 1. El mejor resultado (10 veces con un total de 10000 iteraciones,50 armonías en el Harmony Memory y una varianza de 0.1) ha sido el siguiente:

- Error en Train: 46.4398
- Error en Test: 24.6816
- Solución: 0.378037831530686, 0.0365842967617848, 0.356129977356326, 0.332637855851294, -0.0615815818905862, -0.0661984419013626, 0.894410564736061, 1, 1, 1, 1, 1, 1
- Gráficas:



Conclusión

A la vista de los resultados, se ha observado que el mejor fitness se consigue cuando se realizan **menos iteraciones**, aunque este hecho provoca que el fitness sea muy aleatorio. Asimismo, cuando el error del train se hace pequeño (alrededor de 30), el error en test se hace muy grande. Esto se puede deber a que el algoritmo **aprende** los datos del conjunto de entrenamiento y no generaliza correctamente.

Por otro lado, al cambiar la **varianza** de valor, no se observan grandes cambios. Esto se debe a que el rango de los valores de los elementos es muy pequeño y, cuando algún elemento sobrepasa los límites, éste se cambia por el límite. Esta situación se puede dar al realizar el PAR. Asimismo, se han cambiado los valores de las **probabilidades** PAR (0.2-0.3) y HMCR (0-8-0.9) y no se han observado grandes cambios.

Por último, se ha examinado que al cambiar el modelo de predicción por uno **linealizado** se obtienen mejores resultados, aunque estos no hacen una gran aportación respecto al propuesto en el enunciado. De esta forma se ha obtenido la mejor de las soluciones, es decir, un error en test de 24.68; la configuración de la ejecución fue:

- Varianza: 0.1
- Harmony Memory: 50
- Iteraciones: 10000

El algoritmo Harmony Search funciona de una manera bastante buena en este problema debido a su proceso de improvisación con el que aumenta la posibilidad de encontrar una solución mejor en una iteración próxima.

La ventaja de los algoritmos armónicos frente a otros algoritmos evolutivos se basa en sus características, las cuales lo identifican y hacen de él una poderosa herramienta de optimización. Entre estas destacan:

- No requiere cálculos complejos.
- Obvia óptimos locales.
- Puede manejar variables discretas y continuas.

Todas ellas permiten diferenciarlo de los algoritmos genéticos y hacen de él una poderosa herramienta basada en cálculos matemáticos simples e **improvisación**.

Bibliografía

- Contenido estudiado en la asignatura “APLICACIONES DEL SOFT-COMPUTING EN ENERGÍA, VOZ E IMAGEN”.
- Cobos, Carlos & estupiñan, Dario. (2011). A survey of harmony search. Revista Avances en Sistemas e Informática (RASI). 8. 67-80