Gonzalo García Vence, Manuel Milde y Sofía Ferrari

Trabajo Práctico 1: Aproximación de datos vía funciones lineales continuas a trozos

Introducción:

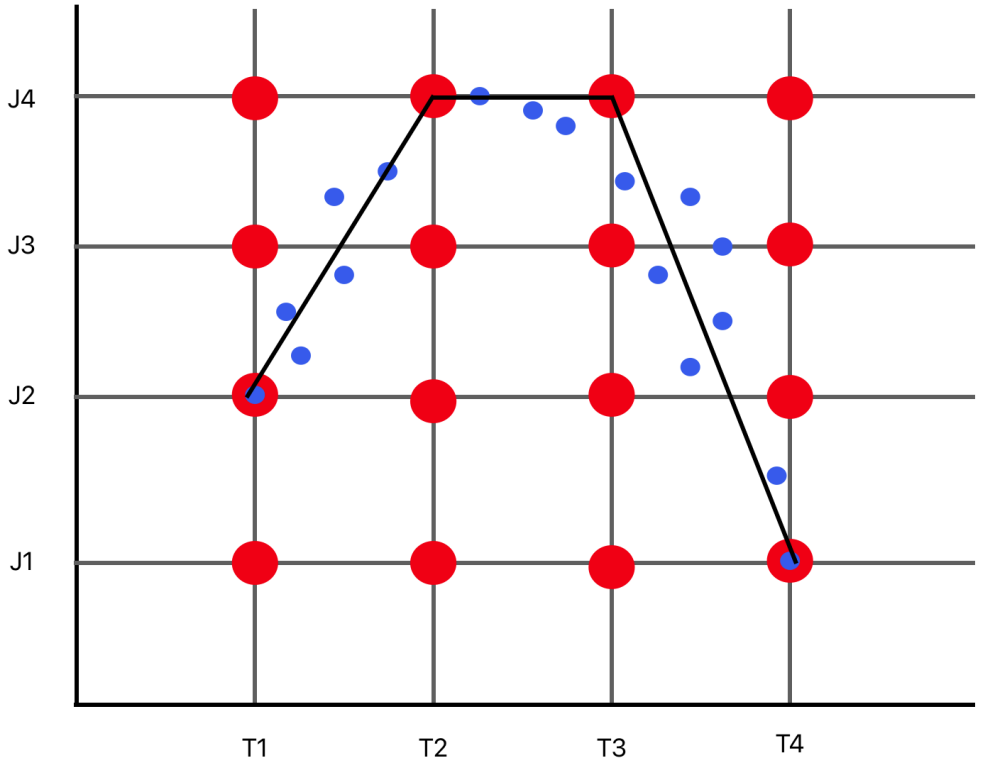
En el siguiente trabajo, se nos pidió como objetivo central crear algoritmos los cuales fueran capaces de aproximar un conjunto de datos mediante funciones *piecewise linear*. Para esto, teníamos diferentes conjuntos de datos para trabajar (Aspen, Ethanol, Titanium y Optimistic) y diferentes parámetros como grillas y N cantidad de breakpoints (o de piezas, depende cómo definamos la función).

Las grillas y los breakpoints son muy importantes, ya que las grillas nos darán el espacio de X e Y discretizado, de manera de que podamos tomar puntos para formar cada función lineal. El armado de estas funciones lineales, va a depender de estas grillas y los puntos del conjunto de datos. Nuestro objetivo, explicado brevemente, será encontrar funciones lineales que se 'conecten' entre breakpoints de tal manera de que:

1. **Cumpla con la restricción de máximos breakpoints.**
2. **Minimice el error con el conjunto de datos.**

En principio, el armado de grilla y la cantidad de breakpoints las seteamos default como venían en los archivos, de manera de poder armar una base en los algoritmos, para luego experimentar con ellos en un futuro en el punto 4.

A continuación, dejo un ejemplo de cómo podemos visualizar un ejemplo de las grillas (de X e Y, con tamaño 4), los puntos que podemos elegir como breakpoints, y la aproximación (como ejemplo) de esos datos:



Teniendo esto en mente, lo primero que hicimos fue pensar el ejercicio número 1. Sabíamos que íbamos a necesitar funciones auxiliares o complementarias al algoritmo principal, así que el primer paso será explicar que hicimos en cada función:

Función F(X) en tramo:

Esta es la función más sencilla. Simplemente pasados 2 breakpoints (X_0,Y_0 y X_1,Y_1) calcula el valor de la función lineal formada entre esos dos puntos evaluada en x. Esto nos va a servir en un futuro para evaluar cada punto y calcular errores.

De esta manera, queda definida como:

$$F(x) = \frac{Y_1 - Y_0}{X_1 - X_0}(x - X_0) + Y_0$$

De manera que nos quedaría igual al presentado en el enunciado.

```
def f_en_tramo(x0,y0,x1,y1,x): #Calcula el valor de la recta
     return (((y1-y0)/(x1-x0))*(x-x0)) + y0 # Simplemente calcula con
la formula provista en el PDF
```

(Ejemplo del código de la implementación en Python)

Calcular error entre V_1 y V_2:

Con esta función, buscamos obtener el error absoluto entre 2 vectores (en el caso nuestro, será equivalente a ir calculando los errores de estimación de una pieza determinada). Esto será de utilidad al comparar Y con Ŷ. En primer lugar consigue el valor absoluto de la diferencia entre cada valor, y luego suma todos los valores.

$$e(x_i, y_i) = |y_i - f(x_i)|$$

Cálculo de error absoluto de una observación.

De esta manera, obtenemos una expresión como la mostrada en el enunciado:

$$D((r_k, z_k), (r_{k+1}, z_{k+1})) = \sum_{r_k < x_i \leq r_{k+1}} |y_i - f_k(x_i)|$$

Cálculo de error absoluto de una pieza formada por (r_k,z_k) y (r_{k+1},z_{k+1}).

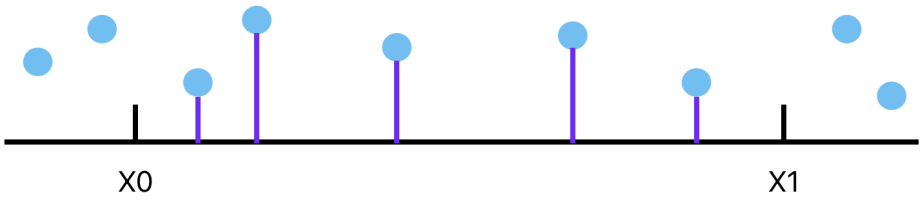
```python
import numpy as np
def calcular_error(vector1, vector2): # Tomo como entrada el vector predicción y vector 'y' reales
      diferencia = np.abs(vector1 - vector2) # Calcula diferencia en valor absoluto de cada predicción
      error = np.sum(diferencia) # Sumamos todos los errores
      return error # Retornamos suma de errores
```

Implementación de la función en Python utilizando la librería numpy. Utilizando C++ hicimos un for iterando por cada valor e ir sumando la diferencia en valor absoluto.

Subconjunto:

En la función subconjunto lo que buscamos hacer es encontrar los valores de X entre X_0 y X_1, para luego encontrar los valores correspondientes de Y a esos X. Esto nos va a servir para evaluar correctamente cada conjunto de X en la función de a tramos, y posteriormente calcular su error.

Imágen de cómo funciona esto:



Lo que vemos en la imagen es el conjunto de datos (marcado con los círculos celestes). Lo que hacemos es obtener aquellas observaciones que estén entre X_0 y X_1 para así obtener el subconjunto deseado.

```python
def subconjunto(x,y,x0,x1):
      sub_X = [x_i for x_i in x if x0 <= x_i <= x1] # Generamos subconjunto de x entre X_0 y X_1
      indice_inferior = x.index(sub_X[0]) # Conseguimos indice inferior
      indice_superior = indice_inferior + len(sub_X) # De igual manera para el indice Superior
      sub_Y = y[indice_inferior:indice_superior] # Generamos subconjunto de y respecto al subconjunto de x
      return sub_X, sub_Y # Retornamos ambos subconjuntos
```

Ejemplo de implementación en Python. En C++ es una implementación similar utilizando un for para ir iterando sobre el conjunto de datos.

Estimar error de Y:

Por último, quizás la función más 'compleja' del código. En esta, lo que buscamos es dada una solución (sea parcial o final) obtener su error total. Como input vamos a recibir x e y (conjunto de datos) y sol, el cual va a ser un array de tuplas con cada breakpoint de la solución (como ya mencionamos, parcial o final).

El código básicamente itera entre los breakpoints, obtiene la predicción $\hat{Y}$ para cada recta (o pieza), compara con Y, obtiene el error y lo suma a 'error', el cual va a sumar el

error de cada 'tramo' de la solución y cuando sea retornado será igual al error total de la solución.

```
def estimar_error_y(sol,x, y): #Calcula el valor de la recta
      i = 0  # Establecemos i = 0
      error = 0 # Establecemos error = 0 para sumar cada error del
tramo

      while i < (len(sol) - 1): # Desde i=0 hasta sol-1 (ya que tomamos
el valor i e i+1 de la sol)
            sub_x, sub_y = subconjunto(x, y, sol[i][0], sol[i+1][0]) #
Generamos subconjunto de X e Y correspondiente al tramo
            sub_x = np.array(sub_x) # Convertimos a Array de numpy
            prediccion = f_en_tramo(sol[i][0], sol[i][1], sol[i+1][0],
sol[i+1][1], sub_x) # Calculamos la estimación para cada punto
            error = error + calcular_error(prediccion,sub_y) #
Calculamos el error de ese tramo y sumamos al anterior            i = i
+ 1 #Pasamos de valor de i
      return error # Retornamos la suma de errores
```

Implementación en Python. En C++ la implementación es prácticamente igual.

Algoritmos:

Teniendo todo esto, buscamos como objetivo encontrar un algoritmo que encuentre la solución cuyos tramos (armados por diferentes breakpoints) tengan error mínimo.

De cierta manera, esta última función (*estimar_error_y)* es **clave**, ya que será la que más utilizaremos para comparar soluciones y 'escoger' la óptima globalmente.

Teniendo esto en mente, lo primero que hicimos, fue el algoritmo de **Brute Force**:

```
sol = [] #Inicializo Solucion como vacio

def fuerza_bruta(grid_x, grid_y, x, y, N, sol_parcial):
     if(len(grid_x) < N - (len(sol_parcial)) ): #Caso base de grilla
con menos de los necesarios
           return {'error':1e10}

     elif(len(sol_parcial) == N): # Si el largo de la solucion es el
necesario
           error_actual = estimar_error_y(sol_parcial, x, y)
           return {'error':error_actual,'puntos':sol_parcial.copy()}
     else:
           sol_global = {'error':1e10} # Dejamos error alto para que
vaya siendo remplazado
           if(N - (len(sol_parcial)) == 1): # Si queda un solo valor
de sol, tiene que ser el ultimo x de la grilla
                grid_x = [grid_x[-1]] # Hacemos que grid_x solo sea
ese valor
           for i in grid_y: # Recorremos la grilla en Y
                sol_parcial.append((grid_x[0], i)) # Sumamos valores
de Y
                parcial = fuerza_bruta(grid_x[1:], grid_y, x, y, N,
sol_parcial) # Evaluamos en ese valor de Y
                if(parcial['error'] < sol_global['error']): #
Evaluamos error ultima iteracion es mejor o peor y remplazamos
                     sol_global = parcial
                sol_parcial.pop() # Quitamos para seguir probando con
el resto
           if(len(sol_parcial) > 0): # Si la sol_parcial no es 0 (es
decir, ya se agrego x1) puedo ir evaluando opciones salteando valores
de x
                parcial = fuerza_bruta(grid_x[1:], grid_y, x, y, N,
sol_parcial)
           if(parcial['error'] < sol_global['error']): # Si tienen
mejor error, remplazo
                sol_global = parcial
     return  sol_global
```

Implementación del algoritmo en Python. Nombraremos paso a paso las decisiones de código en cada caso y las diferencias con el código de C++.

Nota importante:

Cuando realizamos este algoritmo, no nos dimos cuenta y sin querer adoptamos podas de factibilidad (como veremos, siempre llegamos a 'hojas' factibles). Cuando lo hablamos con Juanjo, nos dijo que estaba perfecto tomarlo como Fuerza Bruta, y que para Backtracking incorporemos Podas por Optimalidad.

Otro punto a tomar, es que tanto en Brute Force como en Backtracking buscamos soluciones de 0 -> K y el enunciado plantea de K -> 0. También lo charlamos con Juanjo, y nos dijo que no habría problema en hacer los algoritmos con este planteo.

Ahora sí, pasemos a la explicación del código:

Para empezar, la función toma como parámetro grid_x y grid_y (las grillas de X e Y), x e y (el conjunto de datos a aproximar), N (En nuestra solución, número de Breakpoints) y Sol (inicializado como un array vacío []).

Casos Bases:

Nuestra función recursiva va a tener dos casos bases:

- El primero de ellos se va a encargar de chequear que la cantidad de puntos en la grilla de X sea igual o mayor a la cantidad de puntos necesarios. Esto ya que (como veremos después) en cada llamado recursivo, la grilla de X se va achicando. Esto tiene lógica, ya que si pensamos una solución con 6 Breakpoints pero tenemos una grilla de 4 valores, nunca vamos a poder dar una solución factible (Poda por factibilidad). La decisión de devolver un diccionario con la clave *'error'* con valor alto es algo estratégico, para que nuestro algoritmo entienda que esa solución es infactible.
- El segundo caso base verifica si la solución es final (es decir, que contiene N Breakpoints). Si es así, calcula el error de la solución, y devuelve un diccionario (en caso de C++ un Pair) con los valores de error (donde guardamos el error de la solución) y puntos (donde guardamos los puntos seleccionados).

Si nuestra solución no entra en el caso base, entra en el else. Allí, lo primero que sucede es que se establece una 'solución global' con un error alto para que vaya siendo reemplazada a medida que las soluciones se vayan armando. Inmediatamente después, tenemos el primero de los chequeos (también de factibilidad). Esto verifica que si nos queda por asignar un solo breakpoint, automáticamente se tome el último valor de la grilla de x. Esto es para asegurarnos de tomar todo el dominio de la función y siempre asignar soluciones factibles.

Luego, iteramos sobre cada valor de la grilla de y:

- Allí buscaremos probar cada valor de la grilla de y asignándolo al breakpoint. Se llama al llamado recursivo con la grilla de x sin el valor que agregamos a la solución (otra poda de factibilidad, ya que aseguramos que no se repitan x en la solución y

que el valor de las mismas sea creciente), y si la solución a la que llegamos es mejor que la solución global, se reemplaza. Luego se quita ese valor de la solución parcial para chequear otros valores de la grilla de y.

Para terminar, el código realiza un último chequeo para probar todas las combinaciones posibles de soluciones. Aquí chequeamos que **si la solución parcial es de tamaño 1 o mayor**, podemos hacer llamados recursivos saltando algunos valores de la grilla de X. Esto nos va a servir para armar conjuntos de soluciones en las cuales no tomemos todos los valores de la grilla de X, y solo seleccionamos algunos. Pero adicionalmente, aseguramos que nunca falte el primer valor de la grilla de x a la solución, ya que esto derivaría en una solución no factible (Otra poda por optimalidad). Luego, se hace el chequeo de si esta solución es mejor que la anteriormente almacenada (y si es mejor la reemplaza), para luego terminar devolviendo la solución.

Backtracking:

```
def backtracking(grid_x, grid_y, x, y, N, sol_parcial):
     if(len(grid_x) < N - (len(sol_parcial)) ): #Caso base de grilla
con menos de los necesarios
          return {'error':1e10}

     elif(len(sol_parcial) == N): # Si el largo de la solucion es el
necesario
          error_actual = estimar_error_y(sol_parcial, x, y)
          return {'error':error_actual,'puntos':sol_parcial.copy()}
     else:
          sol_global = {'error':1e10}
          if(N - (len(sol_parcial)) == 1): # Si queda un solo valor
de sol, tiene que ser el ultimo x de la grilla
               grid_x = [grid_x[-1]] # Hacemos que grid_x solo sea
ese valor
          for i in grid_y: # Recorremos la grilla en Y
               sol_parcial.append((grid_x[0], i)) # Sumamos valores
de Y
               if(estimar_error_y(sol_parcial, x, y) <
sol_global['error']): #Poda por Optimalidad
                    parcial = backtracking(grid_x[1:], grid_y, x, y,
N, sol_parcial) # Evaluamos en ese valor de Y
                    if(parcial['error'] < sol_global['error']): #
Evaluamos error ultima iteracion es mejor o peor y remplazamos
                         sol_global = parcial
               sol_parcial.pop() # Quitamos para seguir probando con
el resto
          if(len(sol_parcial) > 0): # Si la sol_parcial no es 0 (es
decir, ya se agrego x1) puedo ir evaluando opciones salteando valores
de x
               parcial = backtracking(grid_x[1:], grid_y, x, y, N,
sol_parcial)
          if(parcial['error'] < sol_global['error']): # Si tienen
mejor error, remplazo
               sol_global = parcial
     return  sol_global
```

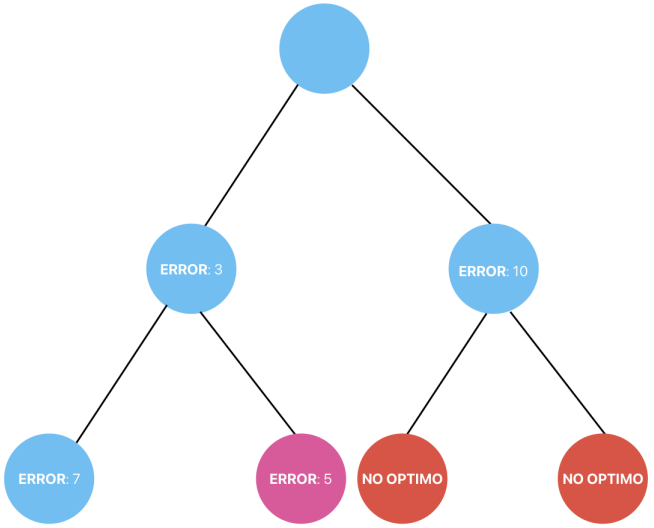
Algoritmo codeado en Python. Su diferencia con C++ es prácticamente nula (más allá de traducciones del lenguajes).

Para el algoritmo de backtracking, simplemente se copió el código de Fuerza Bruta y se le agregó una poda por Optimalidad. En este caso, si observamos el código, la poda busca no seguir expandiendo la solución si lo que llevamos armado tiene más error que el mejor error encontrado hasta ahora.

Tiene lógica tratar esta poda de esta manera, ya que imaginemos que nuestra solución (global) óptima hasta ahora tiene un error de 5. En una entrada recursiva posterior, tenemos una solución parcial con un error de 10. Sabemos que esta solución parcial nunca

será óptima globalmente, porque si el error parcial es 10, el error global será mayor o igual a 10. De esta manera, nos ahorramos seguir con ramas de soluciones que nunca serán óptimas, y nos ahorramos cálculos innecesarios.

En la siguiente imagen se explica cómo pensamos esta poda. El nodo violeta muestra el óptimo global encontrado hasta ese momento, y la rama derecha muestra lo que se está evaluando en el momento.



Programación Dinámica:

Primero de todo, definimos una función auxiliar únicamente para este algoritmo. Esta función (make_cube) nos va a generar un cubo para que funcione como memoria, y así ir almacenando las soluciones que vamos a ir generando con la programación dinámica.

```
def make_cube(N, M, Z):
     matriz = [[[{'error':1e10, 'puntos':[(None,None),(None, None)]}
for _ in range(Z)] for _ in range(M)] for _ in range(N)]
     #Creamos un cubo de NxMxZ asignando a cada entrada un diccionario
que representan soluciones 'vacias', ya que el error es alto. Nos
servira para que vayan siendo remplazadas
     return matriz
```

Implementación de make_cube en Python.

Lo que buscamos es crear un cubo de NxMxZ. Esto para guardar las mejores soluciones de ir a X_m Y_z con N breakpoints.

Por otro lado, tenemos el código completo de **Programación Dinámica**:

```
def prog_dinamica(grid_x, grid_y, x, y, N):
      N = N - 1 #Resto 1 así todos quedan definidos como N = Cantidad
de Breakpoints
      Z = len(grid x)
      M = len(grid_y)
      memoria = make_cube(N,Z,M) #Creamos el cubo que funcionará como
memória
      x0 = grid x[0] # Guardamos primer valor de grilla de X para el
'caso base' de la P. Dinámica, donde buscaremos la mejor recta desde
X0 Yi a Xj Yl
      for k in range(1,Z): #Iteramos grilla de X desde el 1 (ya que el
0 esta fijado en X0)
            for i in range(M): #Iteramos 2 veces sobre grilla de Y para
obtener todos los breakpoints posibles
                  for p in range(M):
                        if memoria[0][k][p]['error'] >
estimar error y([(x0, grid y[i]), (grid x[k], grid_y[p])], x, y): #Si
es menor al anterior guardado, remplazar
                              memoria[0][k][p] = {'error':
estimar_error_y([(x0, grid_y[i]), (grid_x[k], grid_y[p])], x,
y),'puntos': [(x0, grid_y[i]), (grid_x[k], grid_y[p])]}
      for i in range(1,N): # Agarro desde '2 piezas' para adelante, ya
que los valores con 1 sola pieza los calcule en el caso base
            for k in range(2,Z): # Tomo a partir de la tercer posicion
de la grilla de X ya que la de las segunda ya estan calculados
                  for p in range(M): # Itero sobre Grilla de y para
probar todos los valores
                        for l in range(k): #Itero sobre rango de K, ya
que l nunca puede ser mayor a k. Ya que se tiene que cumplir que los X
sean crecientes
                              for t in range(M): #Itero sobre grilla de y
para tomar el segundo valor
                                    error = memoria[i - 1][l][t]['error']
# Me guardo el error de ir hasta Xl Yt
                                    if(memoria[i][k][p]['error'] > error):

                                          error = memoria[i - 1][l][t]
['error'] + estimar error y([(grid x[l], grid y[t]), (grid x[k],
grid_y[p])], x, y) # Guardamos error de ir Xk Yp como el anterior
calculado más la suma de ir de ese a Xk Yp
                                          if(memoria[i][k][p]['error'] >
error): #Remplazo solución si es mejor
                                                memoria[i][k][p]['error'] =
error
                                                puntos = memoria[i - 1][l]
[t]['puntos']
                                                puntos.append((grid_x[k],
grid_y[p]))
                                                memoria[i][k][p]['puntos']
= puntos.copy()
                                                puntos.pop()
      diccionario menor error = min(memoria[N - 1][Z - 1], key=lambda
x: x["error"]) #Devuelvo el mínimo error llegado la ultima posición de
grilla de X, con los breakpoints que pedimos
      return diccionario_menor_error
```

Implementación en Python. No se aprecia bien indentado por cómo se exporta el código a pdf. Aún así, en el código de la entrega se ve perfectamente indentado.

En el código de la foto no se ve correctamente. Aún así, nuestro código sigue las bases de la presentación del problema en el PDF siguiendo esta fórmula:

$$F_M((t_i, z_j)) = \min_{\substack{(t_k,z_l):t_1<t_k<t_i \\ l=1,\dots,m_2}} \Big\{ D((t_k, z_l), (t_i, z_j)) + F_{M-1}((t_k, z_l)) \Big\},$$

con el correspondiente caso base (con $i > 1$)

$$F_1((t_i, z_j)) = \min_{\substack{(t_1,z_l) \\ l=1,\dots,m_2}} D((t_1, z_l), (t_i, z_j)).$$

De esta manera, podemos dividir el ‘problema grande’ en ‘pequeños subproblemas’. De manera coloquial, lo que buscamos hacer es:

- Guardar errores de ir hasta cierto punto con cierta cantidad de breakpoints para utilizarlos en futuros puntos.
- Sabemos que el error de la recta que llega a X_jY_i es igual que el error de la recta a $X_{j-1}Y_z$ + el de la recta que une $X_{j-1}Y_z$ con X_jY_i.

Así, nuestro código lo que busca hacer es generar los casos bases mediante el primer grupo de fors anidados. Allí busca la mejor recta que una un punto de con $X = X_0$ con cualquier otro punto de la grilla (en *criollo*, busca rectas que salgan del inicio hasta cualquier punto con una única pieza).

Luego, en el segundo grupo de fors anidados, usa la fórmula mostrada anteriormente. Entonces, así es posible generar todas las rectas hasta todos los puntos (incluidas las que llegan hasta $X = X_m$). Finalmente, buscamos la recta que llegue hasta el último punto de la grilla y contenga menor error, para así retornarla.

Experimentos:

Calidad:

Dentro de este primer grupo de experimentos, vamos a ver qué ocurre con nuestras predicciones a medida que varía el tamaño de la grilla y la cantidad de piezas (o breakpoints). Como punto de partida, buscamos intentar ‘suponer’ que podría ocurrir al aumentar la cantidad de breakpoints y/o grillas. En este caso:

Hipótesis: *‘A mayor tamaño de grilla (mayor granulación) y/o mayor cantidad de breakpoints, nuestra predicción va a ser siempre mejor’*.

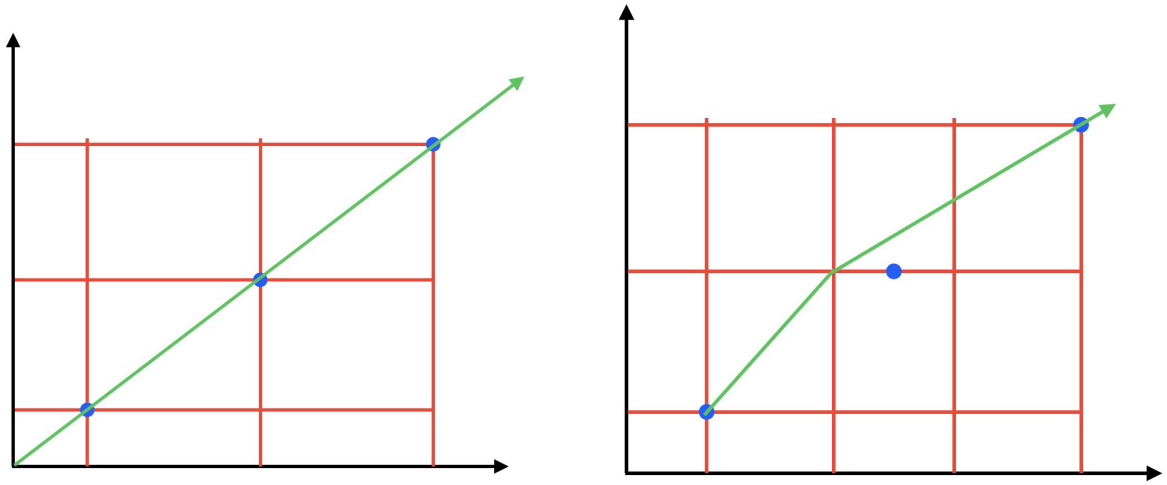
Para probar esto, armamos un pequeño experimento donde para todas las instancias (Optimistic, Titanium, Ethanol y Aspen) probamos diferentes combinaciones de tamaños de grilla y de cantidad de breakpoints, para chequear si en los que son mayores los tamaños tenemos mejores predicciones. Este experimento está hecho en los dos lenguajes con la función exp1(), pero por simplicidad, vamos a mostrar solo el resultado en Python ya que son iguales.

```
Experimento Número 1: Calidad de Predicción
----Set de Datos 1: Titanium----
Primero, con Grillas de Tamaño 6 y 5 Breakpoints:
Brute Force:  {'error': 5.927733333333334, 'puntos': [(595.0, 0.601), (787.0, 0.601), (883.0, 1.2282), (979.0, 0.601), (1075.0, 0.601)]}
Backtracking:  {'error': 5.927733333333334, 'puntos': [(595.0, 0.601), (787.0, 0.601), (883.0, 1.2282), (979.0, 0.601), (1075.0, 0.601)]}
Programación Dinámica:  {'error': 5.927733333333334, 'puntos': [(595.0, 0.601), (787.0, 0.601), (883.0, 1.2282), (979.0, 0.601), (1075.0, 0.601)]}
Ahora, con Grillas de Tamaño 8 y 6 Breakpoints:
Brute Force:  {'error': 6.648666666666671, 'puntos': [(595.0, 0.601), (800.7142857142858, 0.601), (869.2857142857142, 1.2730000000000001), (937.8571428571429, 0.825), (1006.4285714285714, 0.601), (1075.0, 0.601)]}
Backtracking:  {'error': 6.648666666666671, 'puntos': [(595.0, 0.601), (800.7142857142858, 0.601), (869.2857142857142, 1.2730000000000001), (937.8571428571429, 0.825), (1006.4285714285714, 0.601), (1075.0, 0.601)]}
Programación Dinámica:  {'error': 6.648666666666671, 'puntos': [(595.0, 0.601), (800.7142857142858, 0.601), (869.2857142857142, 1.2730000000000001), (937.8571428571429, 0.825), (1006.4285714285714, 0.601), (1075.0, 0.601)]}
----Set de Datos 2: Aspen----
Primero, con Grillas de Tamaño 5 y 3 Breakpoints:
Brute Force:  {'error': 14.785618502041615, 'puntos': [(0.859905, 12.433), (0.9646012500000001, 16.5179725), (0.9995, 24.6879175)]}
Backtracking:  {'error': 14.785618502041615, 'puntos': [(0.859905, 12.433), (0.9646012500000001, 16.5179725), (0.9995, 24.6879175)]}
Programación Dinámica:  {'error': 14.785618502041615, 'puntos': [(0.859905, 12.433), (0.9646012500000001, 16.5179725), (0.9995, 24.6879175)]}
Ahora, con Grillas de Tamaño 6 y 4 Breakpoints:
Brute Force:  {'error': 15.320245941831729, 'puntos': [(0.859905, 12.433), (0.943662, 15.700978), (0.971581, 15.700978), (0.9995, 25.504912)]}
Backtracking:  {'error': 15.320245941831729, 'puntos': [(0.859905, 12.433), (0.943662, 15.700978), (0.971581, 15.700978), (0.9995, 25.504912)]}
Programación Dinámica:  {'error': 15.320245941831729, 'puntos': [(0.859905, 12.433), (0.943662, 15.700978), (0.971581, 15.700978), (0.9995, 25.504912)]}
----Set de Datos 3: Ethanol----
Primero, con Grillas de Tamaño 6 y 5 Breakpoints:
Brute Force:  {'error': 0.5789000000000003, 'puntos': [(0.0, 0.2), (0.2, 0.6000000000000001), (0.4, 0.6000000000000001), (0.8, 0.8), (1.0, 1.0)]}
Backtracking:  {'error': 0.5789000000000003, 'puntos': [(0.0, 0.2), (0.2, 0.6000000000000001), (0.4, 0.6000000000000001), (0.8, 0.8), (1.0, 1.0)]}
Programación Dinámica:  {'error': 0.5789000000000003, 'puntos': [(0.0, 0.2), (0.2, 0.6000000000000001), (0.4, 0.6000000000000001), (0.8, 0.8), (1.0, 1.0)]}
Ahora, con Grillas de Tamaño 7 y 6 Breakpoints:
Brute Force:  {'error': 0.6731999999999995, 'puntos': [(0.0, 0.16666666666666666), (0.16666666666666666, 0.5), (0.3333333333333333, 0.6666666666666666), (0.5, 0.6666666666666666), (0.8333333333333333, 0.8333333333333333), (1.0, 1.0)]}
Backtracking:  {'error': 0.6731999999999995, 'puntos': [(0.0, 0.16666666666666666), (0.16666666666666666, 0.5), (0.3333333333333333, 0.6666666666666666), (0.5, 0.6666666666666666), (0.8333333333333333, 0.8333333333333333), (1.0, 1.0)]}
Programación Dinámica:  {'error': 0.6731999999999995, 'puntos': [(0.0, 0.16666666666666666), (0.16666666666666666, 0.5), (0.3333333333333333, 0.6666666666666666), (0.5, 0.6666666666666666), (0.8333333333333333, 0.8333333333333333), (1.0, 1.0)]}
----Set de Datos 4: Optimistic----
Primero, con Grillas de Tamaño 5 y 3 Breakpoints:
Brute Force:  {'error': 15386.505226480835, 'puntos': [(0.0, 703.0), (64575.0, 920.5), (86100.0, 703.0)]}
Backtracking:  {'error': 15386.505226480835, 'puntos': [(0.0, 703.0), (64575.0, 920.5), (86100.0, 703.0)]}
Programación Dinámica:  {'error': 15386.505226480835, 'puntos': [(0.0, 703.0), (64575.0, 920.5), (86100.0, 703.0)]}
Ahora, con Grillas de Tamaño 6 y 3 Breakpoints:
Brute Force:  {'error': 15993.029616724738, 'puntos': [(0.0, 703.0), (68880.0, 877.0), (86100.0, 703.0)]}
Backtracking:  {'error': 15993.029616724738, 'puntos': [(0.0, 703.0), (68880.0, 877.0), (86100.0, 703.0)]}
Programación Dinámica:  {'error': 15993.029616724738, 'puntos': [(0.0, 703.0), (68880.0, 877.0), (86100.0, 703.0)]}
----Probamos aumentar mucho el tamaño de grilla y número de breakpoints----
----Set de Datos 1: Titanium----
Grillas de Tamaño 20 y 10 Breakpoints:
Programación Dinámica:  {'error': 1.0526982456140375, 'puntos': [(595.0, 0.601), (721.3157894736842, 0.6835263157894736), (797.1052631578948, 0.6835263157894736), (847.6315789473684, 0.7660526315789473), (872.8947368421052, 1.2612105263157893), (898.1578947368421, 2.169), (923.421052631579, 1.2612105263157893), (948.6842105263158, 0.6835263157894736), (973.9473684210527, 0.601), (1075.0, 0.601)]}
```

Como vemos, tenemos 4 sets de datos. En todos ellos, probamos 2 combinaciones de tamaño de grilla y de cantidad de breakpoints. Escogimos estos ejemplos ya que todos son representativos de lo mismo: *más cantidad de breakpoints y mayor tamaño de grilla* ***no*** *asegura mejor predicción.* En cambio, cuando subimos significativamente la cantidad de breakpoints y tamaño de grilla (por ejemplo, el que pusimos en el experimento de ir a tamaño 20 de grilla y 10 breakpoints) obtenemos una solución mejor.

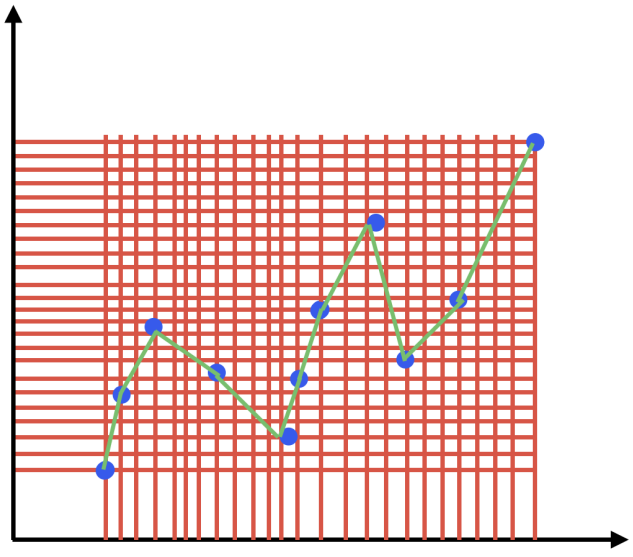
Antes de hacer el experimento, nos hacía sentido que a mayor cantidad de breakpoints o grilla, intuitivamente iba a ser similar al **overfitting** a la hora de trabajar con machine learning. De igual manera, si llevamos la grilla a tamaño muy grande, eventualmente tendremos errores muy bajos ya que estaríamos casi generando un espacio continuo (cosa que no entraba necesariamente en el experimento).

Con este resultado, generamos un ejemplo gráfico de por qué puede ocurrir de que con mayor tamaño de grilla y más breakpoints, nuestra predicción no sea necesariamente mejor:



En este caso, vemos que con un ejemplo bastante básico demostramos que no (necesariamente) necesitamos más tamaño de grilla para predecir mejor. En celeste: las observaciones del conjunto de datos. En ambos gráficos 3 breakpoints.

En conclusión: No **necesariamente** más breakpoints y mayor tamaño de grilla significa mejor resultado. Pero como se mencionó antes, hacer un espacio discretizado en un número muy grande, vamos a terminar generando *prácticamente* un espacio continuo, pudiendo generar predicciones con error muy bajo si así deseamos.



Ejemplo de cómo podría verse una predicción con gran tamaño de grilla (no viable computacionalmente)

Rendimiento:

Dentro de este experimento, vamos a correr sobre los cuatro conjuntos de datos con diferentes cantidades de breakpoints, tamaño de grilla y lenguaje de programación.

El objetivo es encontrar insights de cómo se ven modificadas los tiempos de cómputo según se modifican las variables. Para esto, en el código vamos a encontrar una función llamada *exp2(),* donde vamos a encontrar todos las combinaciones que vayamos explicando.

```
------------------------------------
Experimento Número 2: Perfomance
----Set de Datos 1: Titanium----
Primero, con Grillas de Tamaño 6 y 5 Breakpoints:
Brute Force:  2.166649103164673  segundos
Backtracking:  0.8367798328399658  segundos
Programación Dinámica:  0.0165560245513916  segundos
Ahora, con Grillas de Tamaño 8 y 6 Breakpoints:
Brute Force:  259.52489948272705  segundos
Backtracking:  105.16120076179504  segundos
Programación Dinámica:  0.05992937088012695  segundos
----Set de Datos 2: Aspen----
Primero, con Grillas de Tamaño 5 y 3 Breakpoints:
Brute Force:  0.008343935012817383  segundos
Backtracking:  0.009534358978271484  segundos
Programación Dinámica:  0.0031054019927978516  segundo
Ahora, con Grillas de Tamaño 9 y 3 Breakpoints:
Brute Force:  0.11598801612854004  segundos
Backtracking:  0.18131089210510254  segundos
Programación Dinámica:  0.045069217681884766  segundos
----Set de Datos 3: Ethanol----
Primero, con Grillas de Tamaño 4 y 4 Breakpoints:
Brute Force:  0.008641719818115234  segundos
Backtracking:  0.017253398895263672  segundos
Programación Dinámica:  0.0034019947052001953  segundo
Ahora, con Grillas de Tamaño 7 y 7 Breakpoints:
Brute Force:  45.27163028717041  segundos
Backtracking:  51.60078406333923  segundos
Programación Dinámica:  0.024335622787475586  segundos
----Set de Datos 4: Optimistic----
Primero, con Grillas de Tamaño 3 y 2 Breakpoints:
Brute Force:  0.0008540153503417969  segundos
Backtracking:  0.0009708404541015625  segundos
Programación Dinámica:  0.0017347335815429688  segundo
Ahora, con Grillas de Tamaño 10 y 2 Breakpoints:
Brute Force:  0.008172750473022461  segundos
Backtracking:  0.010759115219116211  segundos
Programación Dinámica:  0.05711650848388672  segundos
```

```
------------------------------------
Experimento Número 2: Performance
----Set de Datos 1: Titanium----
Reading file ../../data/titanium.json
Primero, con Grillas de Tamaño 6 y 5 Breakpoints:
Tiempo de ejecución (fuerza bruta): 0.26749 segundos
Tiempo de ejecución (backtracking): 0.094595 segundos
Tiempo de ejecución (programacion dinamica): 0.00215 segundos
Ahora, con Grillas de Tamaño 8 y 6 Breakpoints:
Tiempo de ejecución (fuerza bruta): 28.1471 segundos
Tiempo de ejecución (backtracking): 12.6691 segundos
Tiempo de ejecución (programacion dinamica): 0.009271 segundos
----Set de Datos 2: Aspen----
Reading file ../../data/aspen_simulation.json
Primero, con Grillas de Tamaño 5 y 3 Breakpoints:
Tiempo de ejecución (fuerza bruta): 0.00038 segundos
Tiempo de ejecución (backtracking): 0.0006 segundos
Tiempo de ejecución (programacion dinamica): 0.000436 segundos
Ahora, con Grillas de Tamaño 9 y 3 Breakpoints:
Tiempo de ejecución (fuerza bruta): 0.002006 segundos
Tiempo de ejecución (backtracking): 0.003502 segundos
Tiempo de ejecución (programacion dinamica): 0.004431 segundos
----Set de Datos 3: Ethanol----
Reading file ../../data/ethanol_water_vle.json
Primero, con Grillas de Tamaño 4 y 4 Breakpoints:
Tiempo de ejecución (fuerza bruta): 0.00159 segundos
Tiempo de ejecución (backtracking): 0.002378 segundos
Tiempo de ejecución (programacion dinamica): 0.000273 segundos
Ahora, con Grillas de Tamaño 7 y 7 Breakpoints:
Tiempo de ejecución (fuerza bruta): 7.55507 segundos
Tiempo de ejecución (backtracking): 8.00082 segundos
Tiempo de ejecución (programacion dinamica): 0.003679 segundos
----Set de Datos 4: Optimistic----
Reading file ../../data/optimistic_instance.json
Primero, con Grillas de Tamaño 3 y 2 Breakpoints:
Tiempo de ejecución (fuerza bruta): 0.000146 segundos
Tiempo de ejecución (backtracking): 0.000234 segundos
Tiempo de ejecución (programacion dinamica): 0.000289 segundos
Ahora, con Grillas de Tamaño 10 y 2 Breakpoints:
Tiempo de ejecución (fuerza bruta): 0.002188 segundos
Tiempo de ejecución (backtracking): 0.002162 segundos
Tiempo de ejecución (programacion dinamica): 0.00928 segundos
```

Print de los experimentos en C++ (Derecha) y Python (Izquierda).

Como primer aspecto a analizar, la diferencia entre la velocidad en los diferentes lenguajes de programación es **inmensa.** En ciertos casos, podemos estar esperando por más de 250 segundos en Python, mientras que en C++ solo demora 28 segundos. En el 100% de los casos, la implementación en C++ es mejor, y podemos de cierta manera ver la variación de tiempo por Algoritmo. En el caso de los algoritmos de fuerza bruta, el cálculo nos dió una variación del más del 1000% en tiempo de ejecución en promedio.

C++	0.26	28.14	0.00038	0.002	0.0015	7.55	0.0001	0.002
Python	2.16	259	0.008	0.1159	0.0086	45.27	0.00085	0.0081

%	~730%	~820%	~2014%	~5695%	~473%	~499%	~750%	~305%

Ejemplo: Implementación de Fuerza Bruta de Py vs C++. Generando así una variación promedio por encima del 3000%

Python	0.83	105	0.009	0.1813	0.017	51.60	0.0009	0.01075
C++	0.09	12.66	0.0006	0.0035	0.0023	8	0.000234	0.002162
%	~822%	~729%	~1400%	~5000%	~630%	~545%	~284%	~397%

Ejemplo: Implementación de Backtracking de Py vs C++. Generando así una variación promedio por encima del 2000%

Python	0.0165	0.059	0.0031	0.045	0.003	0.024	0.0017	0.057
C++	0.002	0.009	0.00043	0.0044	0.00027	0.0036	0.000289	0.00928
%	~725%	~555%	~620%	~922%	~1000%	~566%	~488%	~514%

Ejemplo: Implementación de Programación Dinámica de Py vs C++. Generando así una variación promedio por encima del 500%

Esto era un resultado que nos esperábamos, ya que C++ es conocido por su rapidez, mientras que Python es conocido por su versatilidad para diferentes áreas como álgebra lineal, machine learning, etcétera.

Por otro lado, es interesante ver cómo los algoritmos de Programación Lineal son muy superiores a los otros dos en tiempo de ejecución. Puedo analizar que mientras más sube el tamaño de grilla (x e y) y/o la cantidad de breakpoints, los tiempos en los algoritmos de Backtracking y Fuerza Bruta suben de manera sorprendente. Una intuición de esto puede deberse a la naturaleza recursiva, haciendo que el árbol que se genera con las soluciones sea muy grande. Además, estos algoritmos no logran aprovechar la naturaleza de soluciones de subproblemas, cosa que Programación Dinámica incorpora de manera óptima.

Algo para destacar de los algoritmos recursivos es que vemos el mayor aumento de los tiempos de ejecución cuando la cantidad de breakpoints sube de manera abrupta. Para cerrar esto, estos algoritmos son bastante buenos cuando tenemos pocos breakpoints. Inclusive cuando solamente tenemos dos breakpoints funcionan mejor que programación dinámica (Ver último set de datos).

Esto tiene lógica, ya que los algoritmos recursivos poseen podas para casos en donde queden 1 solo breakpoint por añadir lo que hace que no considere la gran mayoría de espacios en la grilla de x.

Para terminar, vemos que el lenguaje con más varianza en tiempo de ejecución es fuerza bruta, seguido de backtracking y luego Programación dinámica.

Con esto, podemos resumir cosas que observamos:

- C++ es **mucho** mejor a Python para este tipo de tareas

- Al aumentar la cantidad de breakpoints (y lógicamente la cantidad de grilla de x), nuestros algoritmos recursivos sufren muchísimo más que los de Programación Dinámica (los cuales aguantan tranquilamente grillas de más de 25).
- Si tenemos pocos breakpoints (ejemplo dos, es decir una función lineal única), los algoritmos recursivos parecen ser mejores que los dinámicos.Una intuición extra es que aquí los recursivos no pueden aprovechar tanto las soluciones de los subproblemas ya que solo contamos con 2 dimensiones en la profundidad de memoria.
- En los casos de los recursivos, encontramos puntos alrededor de los 7-8-9-10 de largo de grilla donde el tiempo de ejecución sube hasta un punto molesto (literalmente tener que esperar usando el celular hasta que termine de correr).
- En muchos casos vemos como Fuerza Bruta es mejor que Backtracking. Esto se puede deber a que al ya implementar podas de factibilidad, haya casos donde el chequear la solución parcial sea más costoso que no hacerlo en el completo.

Pequeño análisis integral:

Un pequeño análisis integral que es interesante analizar es pensar el trade off rendimiento - calidad. Como vimos en el primer ejemplo, no siempre tener más breakpoints asegura mejor predicción. Pero en cierto punto, ya es mejor tener más breakpoints y tamaño de grilla por el hecho de que discretizan el espacio. Podemos ver en el código de C++ y/o Python, que correr un ejemplo con veinte de tamaño de grilla y diez breakpoints podemos obtener resultados muy buenos, demorando menos de un minuto. Es difícil pensar en un límite con Fuerza Bruta y/o Backtracking, ya que quedan atrás mucho antes, pero como vimos con el ejemplo anterior, obtener de error total aproximadamente 1 es un resultado más que bueno con un tiempo de ejecución muy corto.

A partir de ahí, va a depender plenamente de nuestra decisión sobre a partir de qué error es buena idea tomar una predicción. Y también cuánto tiempo estamos dispuestos a ceder para agregar más breakpoints o tamaño de grilla.

Recomendación y cierre:

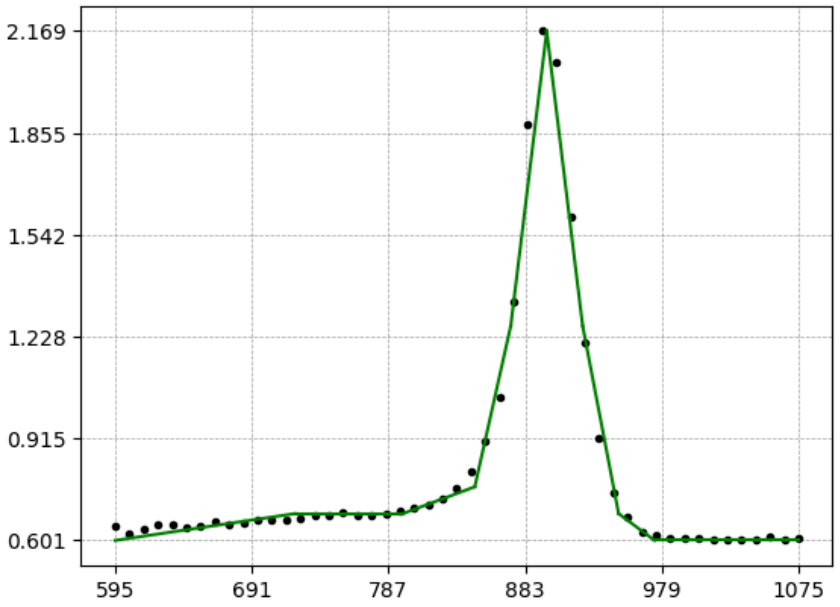
Con todo lo que estuvimos viendo durante el codeo de los algoritmos, trabajar con dos lenguajes, diferentes tipos de datos y demás, fuimos recolectando consejos para estos algoritmos:

- El primero, sería claramente inclinarnos por un modelo de programación dinámica, ya que estos aprovechan al 100% la posibilidad de resolver problemas mediante subproblemas.
- Si no estamos dispuestos a ceder tanta memoria, quizás sea mejor opción alguno de los algoritmos recursivos.
- Por el lado de los lenguajes, si estamos trabajando en un marco donde podemos llegar a utilizar estos algoritmos con aplicaciones de otra índole (Data Visual por ejemplo), quizás Python nos sirva más. Por otro lado, si buscamos eficientar procesos y no nos molesta el hecho de tener que compilar manualmente el código,

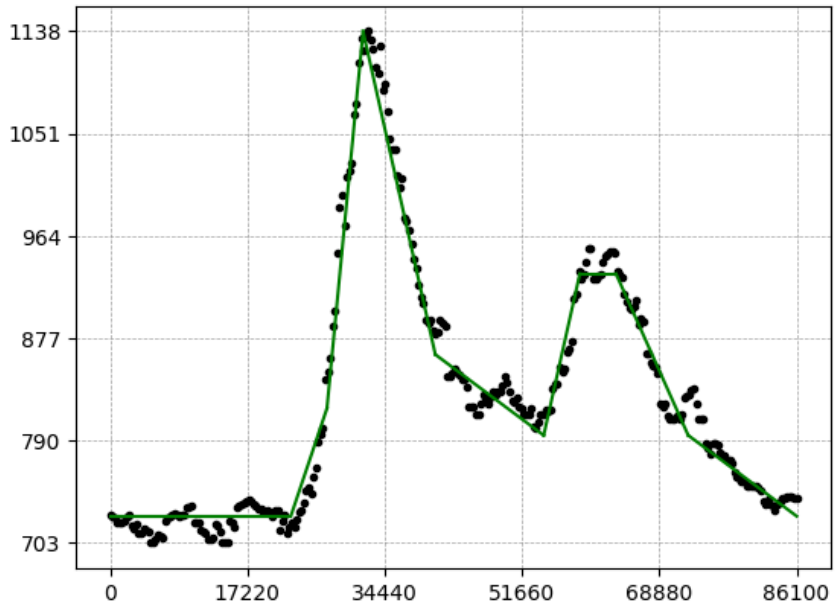
C++ es ***muy*** superior a Python. Los resultados que mostramos son simplemente aplastantes.

- Por último, sería interesante mirar un enfoque con un error medio. El error absoluto a veces puede engañar un poco, ya que tiene muchas desventajas a la hora de interpretarlo. Por ejemplo, un error medio sería muchísimo más conveniente. O inclusive si no queremos depender de distribuciones, MSE puede servir.

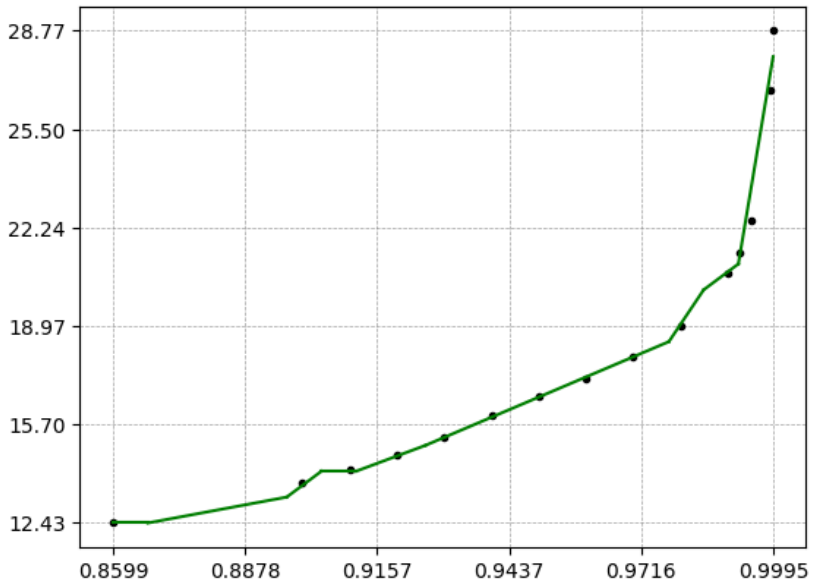
<u>Algunos gráficos de soluciones con tamaño de grilla grande:</u>



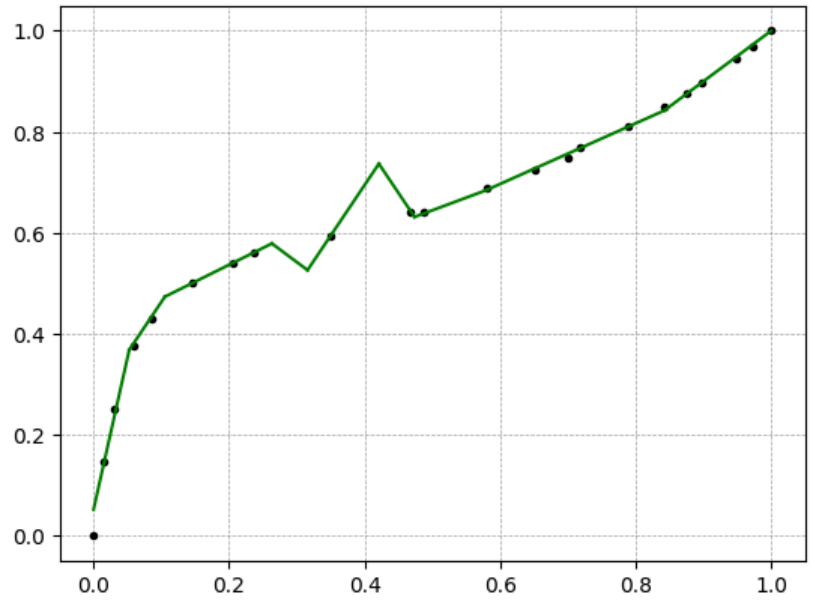
Set de datos de Titanium con tamaño de grilla 20 (x e y) y 10 breakpoints, resultante en error = 1.05.



Set de datos de Optimistic con tamaño de grilla 20 (x e y) y 10 breakpoints, resultante en error = 3286.



Set de datos de Aspen con tamaño de grilla 20 (x e y) y 10 breakpoints, resultante en error = 3.28.



Set de datos de Ethanol con tamaño de grilla 20 (x e y) y 10 breakpoints, resultante en error = 0.11.

Compilación y ejecución del código:

El código de C++ debe ser compilado mediante el comando ‘make’ y luego ser ejecutado con la línea “./pwl_fit”. En el caso de Python, al ser un lenguaje interpretado, simplemente corremos el código.

Testing:

El testing que realizamos fue probar el mismo input para todos los algoritmos en ambos lenguajes, y si obteníamos el mismo resultado, era un buen indicador de que la solución era correcta. También probamos conjuntos de inputs, los cuales ya sabíamos su output correspondiente, y luego corroboramos que coincidan.