In [ ]:

```
# COMPUTACIÓN BLANDA - Sistemas y Computación
#KAREN POSADA Y JUAN CAMILO LOPERA MARTÍNEZ
# ------------------------------------------------------------------
# AJUSTES POLINOMIALES
# ------------------------------------------------------------------
# Lección 06
#
#   ** Se importan los archivos de trabajo
#   ** Se crean las variables
#   ** Se generan los modelos
#   ** Se grafican las funciones
#
# ------------------------------------------------------------------

# Se importa la librería del Sistema Operativo
# Igualmente, la librería utils y numpy
# ------------------------------------------------------------------
import os

# Directorios: chart y data en el directorio de trabajo
# DATA_DIR es el directorio de los datos
# CHART_DIR es el directorio de los gráficos generados
# ------------------------------------------------------------------
from utils import DATA_DIR, CHART_DIR
import numpy as np

# Se eliminan las advertencias por el uso de funciones que
# en el futuro cambiarán
# ------------------------------------------------------------------
np.seterr(all='ignore')

# Se importa la librería scipy y matplotlib
# ------------------------------------------------------------------
import scipy as sp
import matplotlib.pyplot as plt

# Datos de trabajo
# ------------------------------------------------------------------
data = np.genfromtxt(os.path.join(DATA_DIR, "web_traffic.tsv"),
                     delimiter="\t")

# Se establece el tipo de dato
data = np.array(data, dtype=np.float64)
print(data[:10])
print(data.shape)

# Se definen los colores
# g = green, k = black, b = blue, m = magenta, r = red
# g = verde, k = negro, b = azul, m = magenta, r = rojo
colors = ['g', 'k', 'b', 'm', 'r']

# Se definen los tipos de líneas
# los cuales serán utilizados en las gráficas
linestyles = ['-', '-.', '--', ':', '-']

# Se crea el vector x, correspondiente a la primera columna de data
# Se crea el vercot y, correspondiente a la segunda columna de data
x = data[:, 0]
y = data[:, 1]

# la función isnan(vector) devuelve un vector en el cual los TRUE
# son valores de tipo nan, y los valores FALSE son valores diferentes
# a nan. Con esta información, este vector permite realizar
# transformaciones a otros vectores (o al mismo vector), y realizar
# operaciones como sumar el número de posiciones TRUE, con lo
# cual se calcula el total de valores tipo nan
print("Número de entradas incorrectas:", np.sum(np.isnan(y)))

# Se eliminan los datos incorrectos
# ------------------------------------------------------------------

# Los valores nan en el vector y deben eliminarse
# Para ello se crea un vector TRUE y FALSE basado en isnan
# Al negar dichos valores (~), los valores que son FALSE se vuelven
# TRUE, y se corresponden con aquellos valores que NO son nan
# Si el vector x, que contiene los valores en el eje x, se afectan
# a partir de dicho valores lógicos, se genera un nuevo vector en
# el que solos se toman aquellos que son TRUE. Por tanto, se crea
# un nuevo vector x, en el cual han desaparecido los correspondientes
# valores de y que son nan
```

```
# Esto mismo se aplica, pero sobre el vector y, lo cual hace que tanto
# x como y queden completamente sincronizados: sin valores nan
x = x[~np.isnan(y)]
y = y[~np.isnan(y)]

# CON ESTA FUNCIÓN SE DEFINE UN MODELO, EL CUAL CONTIENE
# el comportamiento de un ajuste con base en un grado polinomial
# elegido
# ----------------------------------------------------------------
def plot_models(x, y, models, fname, mx=None, ymax=None, xmin=None):
    ''' dibujar datos de entrada '''

    # Crea una nueva figura, o activa una existente.
    # num = identificador, figsize: anchura, altura
    plt.figure(num=None, figsize=(8, 6))

    # Borra el espacio de la figura
    plt.clf()

    # Un gráfico de dispersión de y frente a x con diferentes tamaños
    # y colores de marcador (tamaño = 10)
    plt.scatter(x, y, s=10)

    # Títulos de la figura
    # Título superior
    plt.title("Tráfico Web en el último mes")

    # Título en la base
    plt.xlabel("Tiempo")

    # Título lateral
    plt.ylabel("Solicitudes/Hora")

    # Obtiene o establece las ubicaciones de las marcas
    # actuales y las etiquetas del eje x.

    # Los primeros corchetes ([]) se refieren a las marcas en x
    # Los siguientes corchetes ([]) se refieren a las etiquetas

    # En el primer corchete se tiene: 1*7*24 + 2*7*24 + ..., hasta
    # completar el total de puntos en el eje horizontal, según
    # el tamaño del vector x

    # Además, se aprovecha para calcular los valores de w, los
    # cuales se agrupan en paquetes de w*7*24. Esto permite
    # determinar los valores de w desde 1 hasta 5, indicando
    # con ello que se tiene un poco más de 4 semanas

    # Estos valores se utilizan en el segundo corchete para
    # escribir las etiquetas basadas en estos valores de w

    # Por tanto, se escriben etiquetas para w desde 1 hasta
    # 4, lo cual constituye las semanas analizadas
    plt.xticks(
        [w * 7 * 24 for w in range(10)],
        ['semana %i' % w for w in range(10)])

    # Aquí se evalúa el tipo de modelo recibido
    # Si no se envía ninguno, no se dibuja ninguna curva de ajuste
    if models:

        # Si no se define ningún valor para mx (revisar el
        # código más adelante), el valor de mx será
        # calculado con la función linspace

        # NOTA: linspace devuelve números espaciados uniformemente
        # durante un intervalo especificado. En este caso, sobre
        # el conjunto de valores x establecido
        if mx is None:
            mx = np.linspace(0, x[-1], 1000)

        # La función zip () toma elementos iterables
        # (puede ser cero o más), los agrega en una tupla y los devuelve

# HASTA AQUÍ ESTÁ RESUELTO

# ----------------------------------------------------------------------

# AQUÍ INICIA LA TAREA DE DOCUMENTACIÓN

# ----------------------------------------------------------------------
```

```
        #Haremos un cico con ayuda de la función zip() que hace:
        #Toma como argumento dos o más objetos iterables retorna un nuevo iterable cuyos elementos
        #son tuplas que contienen un elemento de cada uno de los iteradores originales.
        #En este caso nuestros iterables son models,linestyles y colors


        for model, style, color in zip(models, linestyles, colors):
            # print "Modelo:",model
            # print "Coeffs:",model.coeffs
            plt.plot(mx, model(mx), linestyle=style, linewidth=2, c=color) #se grafica los iterables

        plt.legend(["d=%i" % m.order for m in models], loc="upper left") #La función legend() coloca una leyenda
en los ejes y además la ubica en la posición inferior

    plt.autoscale(tight=True) #Autoescala la vista del eje a los datos
    plt.ylim(ymin=0) #Establece los limites de los ejes, en este caso que el y minimo sea cero
    if ymax: #preguntamos si tenemos un y max
        plt.ylim(ymax=ymax) #entonces que establezca de limite que ymax = ymax
    if xmin: #Pregunta si tenemos un xmin
        plt.xlim(xmin=xmin) #entonces que establezca un xmin
    plt.grid(True, linestyle='-', color='0.75') # Para establecer un orden y que los datos se vean bien, agregare
mos una cuadricula con la función grid()
    plt.savefig(fname) #guardamos los datos

# Primera mirada a los datos
# -------------------------------------------------------------------
 #Llamremos la función que nos grafica y le mandaremos nuestros datos x & y, no usaremos ningun modelo (seran los
datos iniciales) y además la guardaremos
plot_models(x, y, None, os.path.join(CHART_DIR, "1400_01_01.png"))

# Crea y dibuja los modelos de datos
# -------------------------------------------------------------------
fp1, res1, rank1, sv1, rcond1 = np.polyfit(x, y, 1, full=True) #Ajusta un polinomio a la los valores de x & Y ...
. y nos devulve los valores del polinomio, residuos, rango, valores_individuales, segundo
print("Parámetros del modelo fp1: %s" % fp1)
print("Error del modelo fp1:", res1)
f1 = sp.poly1d(fp1) #la función poly1d() ayuda a definir una función polinomial. Facilita la aplicación de "opera
ciones naturales" en polinomios.

fp2, res2, rank2, sv2, rcond2 = np.polyfit(x, y, 2, full=True) #Ajustamos un polinomio de grado dos a los valores
x & y
print("Parámetros del modelo fp2: %s" % fp2)
print("Error del modelo fp2:", res2)
f2 = sp.poly1d(fp2) #Deifinimos la función polinomial en este caso de grado 2

f3 = sp.poly1d(np.polyfit(x, y, 3)) #Ajustamos un polinomio de grado 3 a los valores x & y
f10 = sp.poly1d(np.polyfit(x, y, 10)) #Ajustamos un polinomio de grado 10 a los valores x & y
f100 = sp.poly1d(np.polyfit(x, y, 100)) ##Ajustamos un polinomio de grado 100 a los valores x & y

# Se grafican los modelos
# -------------------------------------------------------------------
plot_models(x, y, [f1], os.path.join(CHART_DIR, "1400_01_02.png")) #Graficamos el modelo de grado 1 y lo guardamo
s como "1400_01_02.png"
plot_models(x, y, [f1, f2], os.path.join(CHART_DIR, "1400_01_03.png")) #Graficamos los modelos de grado 1 y 2 y l
oguardamos como "1400_01_03.png"
plot_models(
    x, y, [f1, f2, f3, f10, f100], os.path.join(CHART_DIR,
                                                "1400_01_04.png")) # Graficamos los modelos de grado 1,2,3,10 y 1
00 y lo guardamos como  "1400_01_04.png"

# Ajusta y dibuja un modelo utilizando el conocimiento del punto
# de inflexión
# -------------------------------------------------------------------
#Otra forma de resolver el problema es usando dos funciones en vez de una, así que vamos encontrar un punto de do
nde podamos dividir las dos funciones
#Con ese punto de inflexión haremos 4 vectores dos que representaran la función a y dos que representaran la func
ión b
inflexion = 3.5 * 7 * 24
xa = x[:int(inflexion)]
ya = y[:int(inflexion)]
xb = x[int(inflexion):]
yb = y[int(inflexion):]

# Se grafican dos líneas rectas
# -------------------------------------------------------------------
fa = sp.poly1d(np.polyfit(xa, ya, 1)) #Ajustamos una polinomio de grado 1 a los valores de la primera función
fb = sp.poly1d(np.polyfit(xb, yb, 1)) #Ajustamos un polinomio de grado 1 los valores de la segunda función

# Se presenta el modelo basado en el punto de inflexión
# -------------------------------------------------------------------
plot_models(x, y, [fa, fb], os.path.join(CHART_DIR, "1400_01_05.png")) #Graficaremos las funciones x & y, además
de los das dos rectas que hicimos anteriormente y las guardaremos como "1400_01_05.png"
```

```
# Función de error
# -------------------------------------------------------------------
def error(f, x, y):  #Vamos a usar una función de error para saber cuanto nuesotrs ajustes tienen de diferencia c
on los datos reales
    return np.sum((f(x) - y) ** 2) #sumaremos la diferencia cuadrada entre f(x) y Y

# Se imprimen los errores
# -------------------------------------------------------------------
print("Errores para el conjunto completo de datos:")
for f in [f1, f2, f3, f10, f100]: #Vamos a hacer un ciclo para los modelos de grado 1,2,3,10 y 100
    print("Error d=%i: %f" % (f.order, error(f, x, y))) #Vamos a imprimir en orden los eerores

print("Errores solamente después del punto de inflexión")
for f in [f1, f2, f3, f10, f100]:#Vamos a hacer un ciclo para los modelos de grado 1,2,3,10 y 100
    print("Error d=%i: %f" % (f.order, error(f, xb, yb))) #Pero en este caso imprimimos los errores solo para la
función después del punto de inflexión

print("Error de inflexión=%f" % (error(fa, xa, ya) + error(fb, xb, yb))) #vamos a imprirmir el error del punto de
inflexión sumando las dos funciones que sacamos de la inflexión

# Se extrapola de modo que se proyecten respuestas en el futuro
# -------------------------------------------------------------------
plot_models(
    x, y, [f1, f2, f3, f10, f100], #vamos a llamar la función plot_models() que nos ayudara a graficar los datos
x & y más los modelos de grado 1,2,3,10 y 100
    os.path.join(CHART_DIR, "1400_01_06.png"), #Lo guardaremos en "1400_01_06.png"
    mx=np.linspace(0 * 7 * 24, 6 * 7 * 24, 100),#vamos a espaciar los numeros durante el intervalo
    ymax=10000, xmin=0 * 7 * 24) #Tendremos como ymax 10000 que sera el valor a encontrar y valor minimo de x la
semana 0

# -------------------------------------------------------------------

# HASTA AQUÍ ES LA TAREA EN SU FASE DE ENTENDIMIENTO Y GENERACIÓN
# DE COMENTARIOS POR LÍNEA

# La parte que sigue es relativa al entrenamiento del modelo
# y la predicción

print("Entrenamiento de datos únicamente despúes del punto de inflexión")
fb1 = fb
fb2 = sp.poly1d(np.polyfit(xb, yb, 2))
fb3 = sp.poly1d(np.polyfit(xb, yb, 3))
fb10 = sp.poly1d(np.polyfit(xb, yb, 10))
fb100 = sp.poly1d(np.polyfit(xb, yb, 100))

print("Errores después del punto de inflexión")
for f in [fb1, fb2, fb3, fb10, fb100]:
    print("Error d=%i: %f" % (f.order, error(f, xb, yb)))

# Gráficas después del punto de inflexión
# -------------------------------------------------------------------
plot_models(
    x, y, [fb1, fb2, fb3, fb10, fb100],
    os.path.join(CHART_DIR, "1400_01_07.png"),
    mx=np.linspace(0 * 7 * 24, 6 * 7 * 24, 100),
    ymax=10000, xmin=0 * 7 * 24)

# Separa el entrenamiento de los datos de prueba
# -------------------------------------------------------------------
frac = 0.3 #El porcentaje de los datos que vamos a tomar para hacer el muestreo
split_idx = int(frac * len(xb)) #Numero de datos de muestreo para entranmiento
shuffled = sp.random.permutation(list(range(len(xb)))) #Hacemos una lista de indices que tenga el tamaño de xb y
luego hace una permutación aleatoria, haciendo que se desordene
test = sorted(shuffled[:split_idx])
train = sorted(shuffled[split_idx:]) #Tome los indices y tome solo 44 de los indices desordenados y ordenelos
fbt1 = sp.poly1d(np.polyfit(xb[train], yb[train], 1))
fbt2 = sp.poly1d(np.polyfit(xb[train], yb[train], 2)) # tomamos xb que sacamos de train y también sacalos los de
b, lo hacemos de grado dos, ajustamos la función matemática y la creamos
print("fbt2(x)= \n%s" % fbt2)
print("fbt2(x)-100,000= \n%s" % (fbt2-100000))
fbt3 = sp.poly1d(np.polyfit(xb[train], yb[train], 3))
fbt10 = sp.poly1d(np.polyfit(xb[train], yb[train], 10))
fbt100 = sp.poly1d(np.polyfit(xb[train], yb[train], 100))

print("Prueba de error para después del punto de inflexión")
for f in [fbt1, fbt2, fbt3, fbt10, fbt100]:
    print("Error d=%i: %f" % (f.order, error(f, xb[test], yb[test])))

plot_models(
    x, y, [fbt1, fbt2, fbt3, fbt10, fbt100],
    os.path.join(CHART_DIR, "1400_01_08.png"),
    mx=np.linspace(0 * 7 * 24, 6 * 7 * 24, 100),
```

```
    ymax=10000, xmin=0 * 7 * 24)


from scipy.optimize import fsolve #La instrucción para hacer la predicción del problema viene de fsolve que es ca
paz de restar de una función el valor que necesitamos llegar para que el corte de la función sea el x que necesit
amos
print(fbt2) #Imprimos la función  matemática  después del punto de inflexión
print(fbt2 - 100000) #Imprimimos la resta de la función con el valor que queremos llegar
alcanzado_max = fsolve(fbt2 - 100000, x0=800) / (7 * 24) #Entonces le restamos a la función predicción el valor q
ue queremos llegar,le decimos además que haga la tarea desde cierto punto para que sea más sencillo y después lo
vamos a dividir en el total de horas de una semana, para saber en que semana ocurre el evento
print("\n100,000 solicitudes/hora esperados en la semana %f" %
      alcanzado_max[0]) #Imprimimos la semana en que alcanzara los 100.000
```