Tutorial de Python, Jupyter-Emacs y Documentos Org

Lenin G. Falconí

2026-02-22

Contents

1 Introducción

Este es un documento para introducir el uso de jupyter-emacs en lugar de ob-ipython dentro de un documento **Org** y para renderizarlo como un PDF exportándolo a LaTeX. En este cuaderno utilizamos algunas bibliotecas y paquetes de Python como `PyTorch`, `numpy` y `matplotlib` para mostrar cómo usar documentos **Org** para escribir un documento científico complejo que incluya ecuaciones, código, imágenes y texto.

Es necesario que tu sistema operativo Linux o WSL ejecute Python dentro de una distribución *Anaconda* o *Miniforge* que gestione los paquetes de Python requeridos. Sigue los pasos introducidos en este repositorio de Github.

2 Escribiendo Python en un Documento Org

Para escribir una sección de código Python dentro de un documento Org, utilice:

1. `C-c C-,` para insertar una plantilla estructurada. Esta combinación de teclas ejecuta el comando `org-insert-structure-template`.
2. Configure el bloque con: `jupyter-python :session test :exports both` para controlar el comportamiento y la dirección de las salidas del bloque. Consulta más información sobre los Bloques SRC.

```
#+begin_src jupyter-python :session test :results both
def say_hi(yourname):
    print(f"Hola {yourname}, ¿qué lograremos hoy?")
say_hi("Isaac Asimov")
#+end_src
```

3. Para editar el código dentro del bloque `#+begin_src`, puede:
   - Cambiar el **modo principal** con `M-x python-mode`. Esto cambiará el modo principal de `Org` a `Python`. La ventaja es que puede trabajar en todo el documento como si fuera un script completo de Python. Después de escribir tu código, debes volver a cambiar con `M-x org-mode`.
   - Abrir una nueva ventana para editar el código usando `C-c '`. Esto último ejecuta el comando `org-edit-special`, que se utiliza para llamar al editor específico dependiendo del código involucrado (por ejemplo, tabla, código fuente, LaTeX, nota al pie, etc.). Después de hacer cambios, vuelva con `C-c '`. Sin embargo, he notado que al usar esta opción, Emacs no recuerda otras variables en el documento o no puede verlas.
   - Usar una combinación de ambos para aumentar la productividad. Si solo necesitas hacer una corrección menor, usaría `C-c '`. Pero si necesito hacer una actualización importante del código, usar bibliotecas y variables previas, cambio el modo principal.
4. Para ejecutar el código dentro del bloque, simplemente use `C-c C-c`. Esta combinación de teclas cambia su comportamiento dependiendo del contexto. Sin embargo, si se realiza en un bloque de código, evaluará el código dentro de él. El uso de `:session test` ayuda a utilizar todas las variables en todo el documento.

5. Para exportar el documento a PDF, use: `org-latex-export-to-pdf`. Necesita tener LaTeXinstalado. También resuelva cualquier dependencia que su sistema operativo pueda necesitar.

Recuerde que puede pedir ayuda a Emacs sobre una combinación de teclas llamando `C-h k` y escribiendo la combinación (por ejemplo, `C-x C-f`). Emacs abrirá un nuevo búfer en una nueva ventana mostrando la ayuda correspondiente. Puede encontrar ayuda en la web en el Manual de Org y en emacs docs sobre el Manual de Org.

3 Problem

Para esta sesión, abordamos el problema de ajustar un modelo lineal $f(x) = ax + b$ a datos generados sintéticamente. Este es un problema de regresión. Consideremos que el modelo lineal verdadero es:

$$y = -2.3x + 7.8 \tag{1}$$

donde $a = -2.3$ y $b = 7.8$

3.1 Generación de Datos

Se generan 100 puntos de datos espaciados linealmente para servir como el conjunto de entrenamiento completo. Como se trata de una demostración, no se considera el rendimiento de generalización del modelo, centrándose en cambio en emplear el modo Org para la comunicación científica. Los datos sintéticos incorporan ruido gaussiano normalizado en ϵ. Posteriormente, los datos se mezclan para evitar que el modelo aprenda cualquier orden inherente.

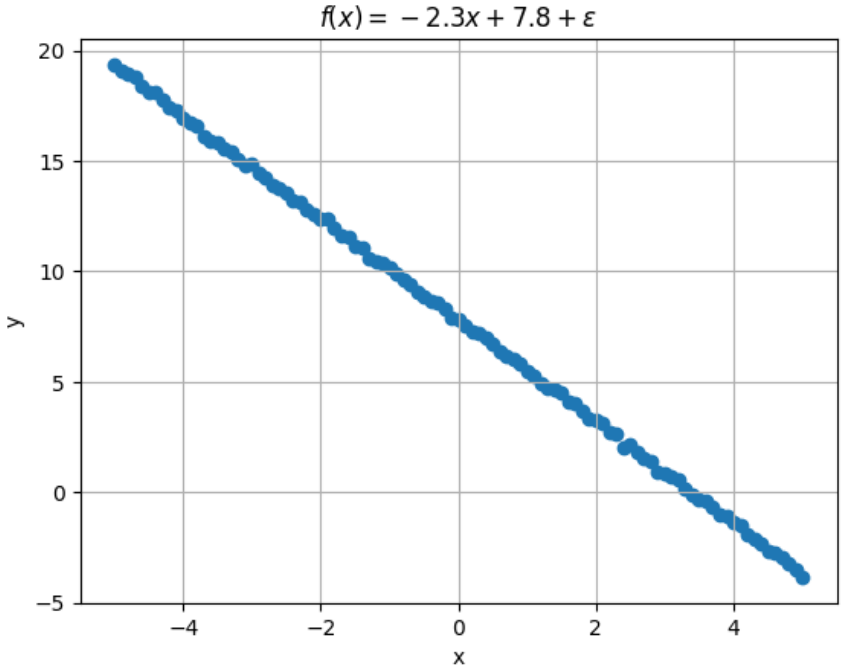
```
import numpy as np
from sklearn.utils import shuffle
seed = 42
np.random.seed(seed)
x = np.linspace(-5,5,101)
print(f"x:\n{x[:5]}")
y = -2.3*x+7.8+0.1*np.random.randn(len(x))
x = x.reshape(-1,1)
y = y.reshape(-1,1)
x,y = shuffle(x,y,random_state=42)
print(f"x:\n{x[:5]}")
```

```
print(f"y:\n{y[:5]}")
print(x.shape)
print(y.shape)
```

```
x:
[-5.  -4.9 -4.8 -4.7 -4.6]
x:
[[ 3.4]
 [ 0.5]
 [ 1.6]
 [ 1.7]
 [-0.5]]
y:
[[-0.10084936]
 [ 6.74312801]
 [ 4.11279899]
 [ 3.99035329]
 [ 8.87801558]]
(101, 1)
(101, 1)
```

Los datos generados se grafican utilizando matplotlib. La configuración puede variar dependiendo de si estamos interesados en mostrar la imagen en el monitor o en guardarla directamente en una carpeta de imágenes. Por defecto, las imágenes se guardan en el directorio `.ob-jupyter`

```
import matplotlib.pyplot as plt
plt.scatter(x,y)
plt.xlabel("x")
plt.ylabel("y")
plt.title("$f(x) = -2.3x+7.8+\epsilon$")
plt.grid()
plt.show()
```



3.2 Ajuste de un modelo lineal con Pytorch y Descenso de Gradiente

PyTorch se utiliza para entrenar un modelo simple de regresión lineal usando **Descenso de Gradiente**. Primero, los arrays de numpy actuales se transforman en tensores de torch y se configuran para trabajar con la GPU si está disponible. El siguiente código también imprime detalles de la GPU actual.

```
import torch
device = "cuda" if torch.cuda.is_available() == True else "cpu"
print(device)
print(f"is cuda available?: {torch.cuda.is_available()}")
print(f"cuda device name if available: {torch.cuda.get_device_name()}")
print(f"Current Device: {torch.cuda.current_device()}")
x_tensor = torch.tensor(x,dtype=torch.float, device=device)
y_tensor = torch.tensor(y,dtype=torch.float, device=device)
print(x_tensor.type())
print(y_tensor.type())
```

```
cuda
is cuda available?: True
cuda device name if available: NVIDIA GeForce GTX 1660 Ti
Current Device: 0
torch.cuda.FloatTensor
torch.cuda.FloatTensor
```

3.3 Lazo de entrenamiento

Un bucle de entrenamiento es el proceso iterativo central en el aprendizaje automático donde un modelo aprende de los datos ajustando sus parámetros para minimizar una función de pérdida predefinida. Los siguientes pasos describen una implementación estándar:

1. Los parámetros a y b se inicializan con valores aleatorios, y se habilita el seguimiento de gradientes para estos tensores en PyTorch.

2. Se generan predicciones usando el modelo actual basado en los parámetros a y b.

3. Una **función de costo** cuantifica el **error** entre los valores verdaderos y los predichos. Aquí se utiliza el **Error Cuadrático Medio**.

4. Se establece un número definido de épocas de entrenamiento, y se aplica retropropagación para calcular los gradientes de la **pérdida** con respecto a cada parámetro. El gradiente de la función de costo J con respecto al vector de parámetros $\boldsymbol{\theta}$ se calcula como:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{2}$$

5. Los parámetros se actualizan según la regla del descenso de gradiente usando una tasa de aprendizaje η:

$$\boldsymbol{\theta}^{\text{nuevo}} = \boldsymbol{\theta}^{\text{viejo}} - \eta \frac{\partial J}{\partial \boldsymbol{\theta}} \tag{3}$$

6. El proceso se repite durante el número especificado de épocas.

Una inicialización adecuada y la habilitación explícita de gradientes son críticas. Esto asegura que el grafo computacional se construya correctamente desde el principio, permitiendo a PyTorch rastrear todas las operaciones para un cálculo preciso de gradientes durante la retropropagación.

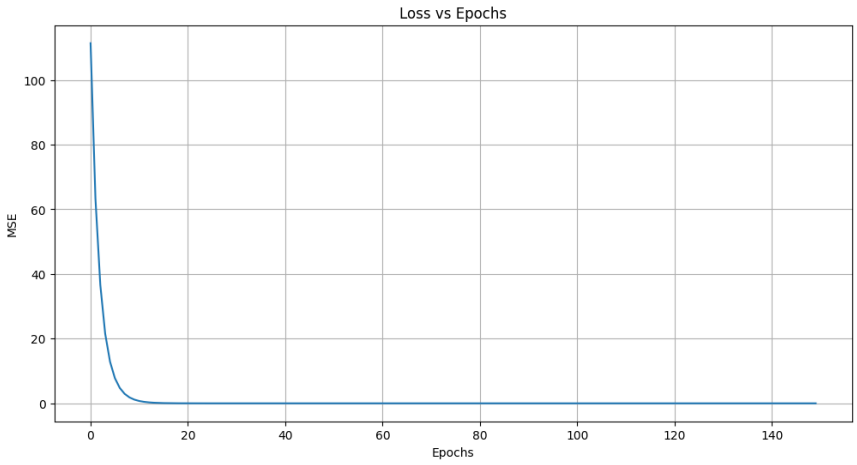
El siguiente código inicializa los parámetros a y b, establece una semilla aleatoria para reproducibilidad, define una función de costo (error cuadrático medio), selecciona un optimizador (descenso de gradiente estocástico) con una tasa de aprendizaje especificada, e implementa un bucle de entrenamiento para un número predeterminado de épocas.

```
import torch

torch.manual_seed(seed)
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
print(f"initial a:{a}")
print(f"initial b:{b}")

lr=0.1
optimizer = torch.optim.SGD([a,b], lr=lr)
loss_fn = torch.nn.MSELoss(reduction='mean')
epochs = 150

loss_values = []
for epoch in range(epochs):
    # prediction
    y_predict = a*x_tensor + b
    # calculate error
    loss = loss_fn(y_predict, y_tensor)
    # print(f"Loss MSE: {loss.item()}")
    loss_values.append(loss.item())
    # backpropagation
    loss.backward()
    # update parameters
    optimizer.step()
    # gradients are accumulative in PyTorch so make them 0 for new epoch
    optimizer.zero_grad()

print("Adjusted Parameters:")
print(f"final a: {a}")
print(f"final b: {b}")
print(f"Final Loss: {loss.item()}")

initial a:tensor([0.1940], device='cuda:0', requires_grad=True)
```

```
initial b:tensor([0.1391], device='cuda:0', requires_grad=True)
Adjusted Parameters:
final a: tensor([-2.2994], device='cuda:0', requires_grad=True)
final b: tensor([7.7883], device='cuda:0', requires_grad=True)
Final Loss: 0.00825003907084465
```

Finalmente, graficamos la pérdida en función del número de épocas. Se puede observar que a medida que aumenta el número de iteraciones, la función de costo disminuye, aproximándose a un conjunto de parámetros que se acercan a los valores reales de a y b para este tutorial.

```
plt.figure(figsize=(12,6))
plt.plot(range(epochs), loss_values)
plt.xlabel("Epochs")
plt.ylabel("MSE")
plt.title("Loss vs Epochs")
plt.grid()
plt.show()
```



4 Conclusión

Este documento sirve como una introducción a la producción de documentación técnica usando Org Mode. Se han logrado los siguientes objetivos:

- Se ha integrado notación matemática, código de programación, texto y gráficos.

- Se ha minimizado el uso de código LaTeXal confiar en el lenguaje de marcado directo de Org Mode.
- Se ha generado un PDF permitiendo que Org Mode maneje automáticamente la compilación y la generación del código LaTeX.