Python Jupyter-Emacs and Org Documents Tutorial

Lenin G. Falconí

2026-02-19 Thu

Contents

1 Introduction

This a document to introduce the use of jupyter-emacs instead of ob-ipython inside an **Org** document and to render it as a PDF by exporting it to LaTeX. In this notebook we use some Python libraries and packages like `PyTorch`, `numpy` and `matplotlib` to show how to use **Org** documents to write a complex scientific document involving equations, code, images and text.

It is necessary that your Linux or WSL Operative System runs python inside an *Anaconda* or *Miniforge* distribution that handles the Python packages needed. Follow the steps introduced in this Github repository .

2 Writing Python inside Org Document

To write a Python section code inside an Org document use:

1. `C-c C-,` to insert an structured template. This key combination runs the command `org-insert-structure-template`.

2. Set it up the block with: `jupyter-python :session test :exports both` to control the behavior and the direction of the outputs of the block. Check more information about SRC Blocks.

```
#+begin_src jupyter-python :session test :resutls both
def say_hi(yourname):
    print(f"Hello {yourname}, what will we achieve today?")
say_hi("Isaac Asimov")
#+end_src
```

3. To edit the code inside the `#+begin_src` block you can either:

   - switch the **major mode** with `M-x python-mode`. This will change the major mode from `Org` to `Python`. The benefit of that is that you can work the whole document like it were a whole Python script. After writing your code you have to switch back with `M-x org-mode`.
   - Open a new window to edit the code by using `C-c '`. The latter executes the command `org-edit-special`, which is used to call the specific editor depending on the code involved (e.g. table, source code, LaTeX, footnote, etc.). After doing changes, switch back with `C-c '`. However, I have noticed that when using this option Emacs does not recall about other variables in the document or cannot see them.
   - Use a combination of both to boost productivity. If you just need to make a minor fix I would go with `C-c '`. But if I need make a major update to code, use libraries and prior variables, switch the major mode.

4. To execute the code inside the block just use `C-c C-c`. This combination of keys switches its behavior depending on the context. However, if performed on a code block it will evaluate the code inside of it. The use of `:session test` aids to use all the variables in the whole document.

5. To export the document to PDF use: `org-latex-export-to-pdf`. You need to have LaTeXinstalled. Also solve any dependency that your Operative System may need.

Remember you can ask Emacs for help about a combination of keys by calling `C-h k` and writing the combination (e.g. `C-x C-f`). Emacs will open a new buffer in a new window showing the corresponding help. Help on the web can be found at Org Manual and emacs docs on the Org Manual.

3 Problem

For this session, we address the problem of fitting a linear model $f(x) = ax+b$ to synthetic generated data. This is a regression problem. Consider the true linear model to be:

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

where $a = -2.3$ and $b = 7.8$

3.1 Data Generation

A 100 linearly spaced data points are generated to serve as the complete training set. As this is a demonstration, we do not consider the model's generalization performance, focusing instead on employing Org mode for scientific communication. The synthetic data incorporate normalized Gaussian noise in ϵ. The data are then shuffled to prevent the model from learning any inherent ordering.

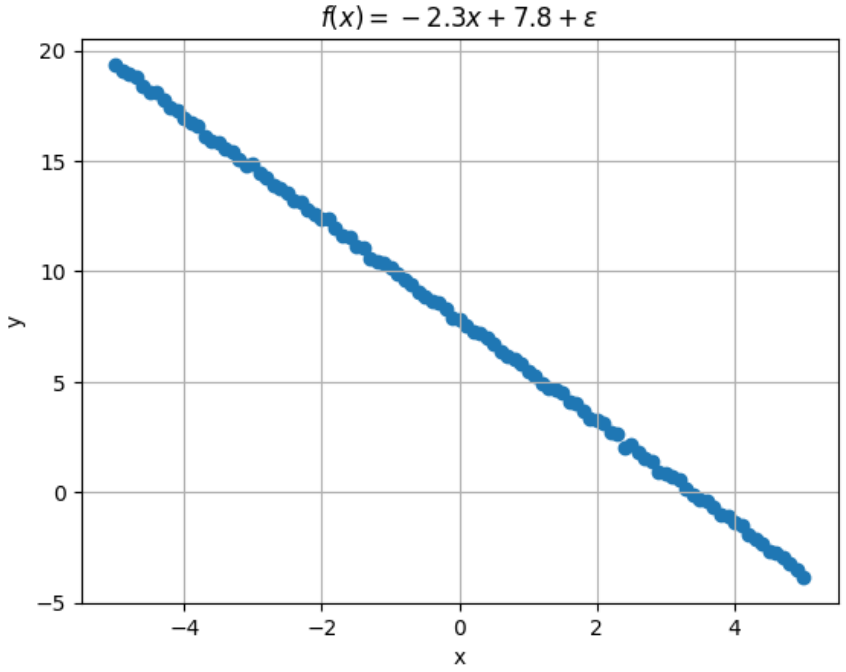
```
import numpy as np
from sklearn.utils import shuffle
seed = 42
np.random.seed(seed)
x = np.linspace(-5,5,101)
print(f"x:\n{x[:5]}")
y = -2.3*x+7.8+0.1*np.random.randn(len(x))
x = x.reshape(-1,1)
y = y.reshape(-1,1)
x,y = shuffle(x,y,random_state=42)
print(f"x:\n{x[:5]}")
print(f"y:\n{y[:5]}")
print(x.shape)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

The generated data is plotted using matplotlib. The configuration may vary depending if we are interested in displaying the image to the monitor or directly saving it to an image folder. By default, images are saved to directory `.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 Fitting a linear model with Pytorch and Gradient Descend

PyTorch is used to train a simple linear regression model using **Gradient Descent**. First, current numpy arrays are transformed to torch tensors and set to work with the GPU if available. The following code also prints details of the current GPU.

```
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 The training Loop

A training loop is the core iterative process in machine learning where a model learns from data by adjusting its parameters to minimize a predefined loss function. The following steps outline a standard implementation:

1. Parameters a and b are initialized with random values, and gradient tracking is enabled for these tensors in PyTorch.
2. Predictions are generated using the current model based on the parameters a and b.
3. A **cost function** quantifies the **error** between true and predicted values. The **Mean Squared Error** is used here.
4. A defined number of training epochs is set, and backpropagation is applied to compute the gradients of the **loss** with respect to each parameter. The gradient of the cost function J with respect to the parameter vector $\boldsymbol{\theta}$ is calculated as:
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{2}$$
5. Parameters are updated according to the gradient descent rule using a η learning rate:
$$\boldsymbol{\theta}^{\text{new}} = \boldsymbol{\theta}^{\text{old}} - \eta \frac{\partial J}{\partial \boldsymbol{\theta}} \tag{3}$$
6. The process is repeated for the specified number of epochs.

Proper initialization and explicit enabling of gradients are critical. This ensures the computational graph is built correctly from the start, allowing PyTorch to track all operations for accurate gradient computation during backpropagation.

The following code initializes the parameters a and b, sets a random seed for reproducibility, defines a cost function (mean squared error), selects an optimizer (stochastic gradient descent) with a specified learning rate, and implements a training loop for a predetermined number of epochs.

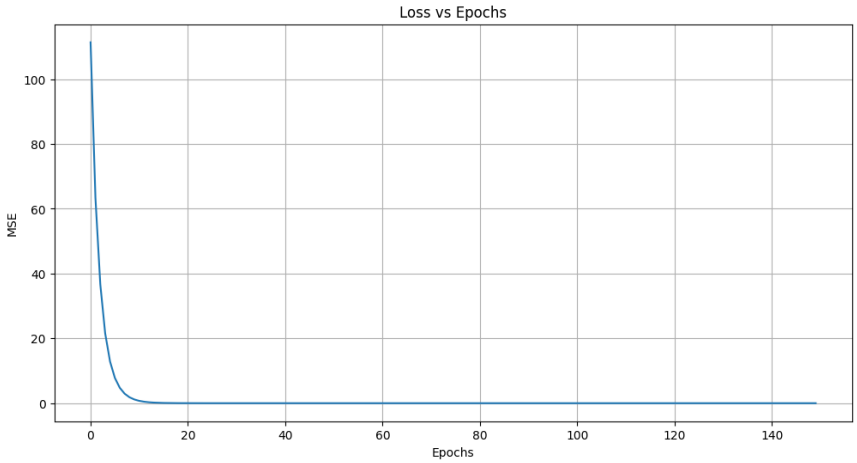
```
import torch

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
```

Finally, we plot the loss against the number of epochs. It can be observed

that as the number of iterations increases, the cost function decreases, approaching a set of parameters that approximate the true values of a and b for this 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 Conclusion

This document serves as an introduction to producing technical documentation using Org Mode. We have accomplished the following objectives:

- We have integrated mathematical notation, programming code, text, and graphics.
- We have minimized the use of LaTeXcode by relying on Org Mode's straightforward markup language.
- We have generated a PDF by allowing Org Mode to handle the compilation and LaTeXcode generation automatically.