Jupyter Notebooks in Web Pages

thedatafrog.com/en/articles/jupyter-notebooks-web-pages/

Learn how to integrate jupyter notebooks in a web page, and how to highlight code and show nice equations.

Introduction

In this post, you'll learn how to set up a web page to communicate scientific results with style!

When you're done with this exercise, you will know how to:

- **integrate perfectly a jupyter notebook in a web page,**
- apply an elegant overall style with **Bootstrap**,
- highlight code with **Pygments**,
- render math equations with **MathJax**.

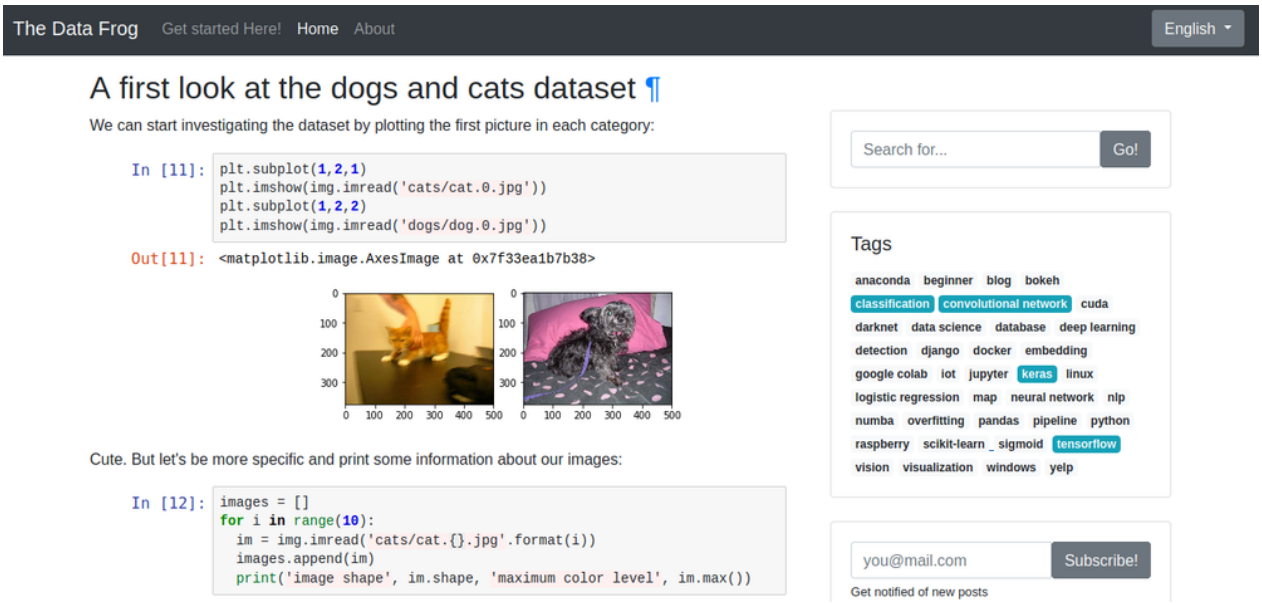
There are ways to do this with a content management system (CMS) like WordPress, and I was actually doing that before. But that's in fact more difficult and the result is not as good. You can try it if you like, and even come back here to ask questions in the comments if you need to. I'll be happy to help.

However, in the following, I will assume that you want to do better, and that you are writing your own HTML pages.

I personally use django as a web framework, with wagtail as CMS. You could decide to go for another python-based web framework like Flask, a PHP-based framework, or even stay away from web frameworks and write plain static pages. By the way, if you want a static blog, you should definitely consider Jekyll.

Here, I set up a static HTML page to be able to help you, whatever path you take.

And all the code is available in this github repository.



Jupyter nbconvert

If you're here, you most certainly know about jupyter nbconvert.

There are basically two ways to use this tool to convert a notebook to HTML.

The default template mode is invocated like this:

```
jupyter nbconvert --execute my_notebook.ipynb
```

The resulting file, `my_notebook.html`, can be loaded in a web browser, and looks exactly like a live jupyter notebook.

You get a nice style for the input and output cells, for the tables, and for the plots. Moreover, the code is highlighted, and the math equations written in markdown appear properly.

This works because all the necessary CSS (style sheets) and javascript are included either in the HTML header or in the HTML body. Also, the file contains opening and closing `<html>` tags.

But the contents of such a file cannot be seamlessly inserted in a web page, nor in a full blown website.

A solution could be to include this file in an iframe, but there are two important drawbacks: The style within the iframe is not going to be consistent with the style of the rest of the page, and you'll get a fixed iframe viewport with a vertical scrollbar that you are forced to use to read the whole jupyter notebook.

The basic template mode of nbconvert is the one we're going to use. One just needs to add an option:

```
jupyter nbconvert --execute --template basic my_notebook.ipynb
```

This time, only the contents of the body is written to the output file. You can still load this file in a web browser, but you'll see that all the nice styling is gone: fonts and page setup from the nineties, no more syntax highlighting, no more equations.

In this article, you will learn how to get back all this, and to integrate the jupyter notebook in your website with a fully consistent style.

For testing, we will use a jupyter html file that I generated with the basic template mode for one of my posts.

Installation

I created a github repository with all the files we need.

Just clone it and follow the instructions in the README to see the final results.

This may be all you need! but if you want to understand how all this is working so that you can perform this integration in your own website, please keep reading.

A simple test webpage

The goal of this article is to make it easy for you to understand what needs to be done to integrate a jupyter notebook in a webpage, so that you can do it by yourself in your own website.

For this reason, I didn't want to rely on complex web frameworks such as Django or Flask, and decided to stick with HTML and python code.

So we're going to start with a very simple test webpage:

```
<!DOCTYPE html>

<html>
  <head>
  </head>

  <body>
    <h1>Integrating a Jupyter Notebook in a Web Page</h1>
    <p>Some text.</p>

    {% include overfitting.html %}

  </body>
</html>
```

Our goal is now to replace the template tag `{% include overfitting.html %}` by the contents of our jupyter notebook, and to make all of this look nice.

Jinja2 as a template engine

Dynamic web sites such as this blog rely on a template engine. Its role is to generate web pages on the fly, depending on client requests. For example, take the tag box on the right side of this page. Under the hood, the template engine generates this box by looping on all existing tags. If the tag is associated with the current page, it is highlighted.

For this website, I use the django template engine.

Here, we're going to use Jinja2, a standalone python template engine, to replace the template tag `{% include overfitting.html %}` by the contents of the jupyter notebook HTML file.

So first install Jinja2, e.g. with pip:

```
pip install jinja2
```

Then, create a script called `render.py` with the following code:

```
import os

from jinja2 import Environment, FileSystemLoader

# this tells jinja2 to look for templates
# in the templates subdirectory
env = Environment(
    loader = FileSystemLoader('templates'),
)

input_file = 'main.html'
output_file = 'index.html'

# reading the template
template = env.get_template(input_file)
# render the template.
# in other words, we replace the template tag
# by the contents of the overfitting file
rendered = template.render()

# write the result to disk in index.html
with open(output_file, 'w') as ofile:
    ofile.write(rendered)
```

Please have a look at the inline comments to understand what this script is doing. You see, it's quite simple.

Then, run it:

```
python render.py
```

And open the resulting index.html file in your browser. It should basically work, but the result is ugly:

Building a small dataset¶

Let's create a sample of examples with two values x1 and x2, with two categories. For category 0, the underlying probability distribution is a 2D Gaussian centered on (0,0), with width = 1 along both directions. For category 1, the Gaussian is centered on (1,1). We assign label 0 to category 0, and label 1 to category 1.

In [17]:

```
def make_sample(nexamples, means=([0.,0.],[1.,1.]), sigma=1.):
    normal = np.random.multivariate_normal
    # squared width:
    s2 = sigma**2.
    # below, we provide the coordinates of the mean as
    # a first argument, and then the covariance matrix
    # which describes the width of the Gaussian along the
    # two directions.
    # we generate nexamples examples for each category
    sgx0 = normal(means[0], [[s2, 0.], [0.,s2]], nexamples)
    sgx1 = normal(means[1], [[s2, 0.], [0.,s2]], nexamples)
    # setting the labels for each category
    sgy0 = np.zeros((nexamples,))
    sgy1 = np.ones((nexamples,))
    sgx = np.concatenate([sgx0,sgx1])
    sgy = np.concatenate([sgy0,sgy1])
    return sgx, sgy
```

Here, we create a very small training sample with only 30 examples per category, and a test sample with 200 examples per category. We're using such a small training sample because, as will be shown in this post, small samples are very easy to overfit.

In [18]:

```
sgx, sgy = make_sample(30)
tgx, tgy = make_sample(200)
```

So we're going to work on the style.

Slick styling with bootstrap

When it comes to web styling, bootstrap is the way to go. We're going to add just a few lines to our main template, render again, and get a modern web page. So edit `templates/main.html` so it looks like this:

```
<!DOCTYPE html>

<html>
  <head>
    <link rel="stylesheet"

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
          integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
          crossorigin="anonymous">
  </head>

  <body>
    <div class="container">
      <h1>Integrating a Jupyter Notebook in a Web Page</h1>
      <p>Some text.</p>

      {% include 'overfitting.html' %}

    </div>
  </body>
</html>
```

Render again, and refresh your browser. Much nicer!

Building a small dataset¶

Let's create a sample of examples with two values x1 and x2, with two categories. For category 0, the underlying probability distribution is a 2D Gaussian centered on (0,0), with width = 1 along both directions. For category 1, the Gaussian is centered on (1,1). We assign label 0 to category 0, and label 1 to category 1.

In [17]:

```
def make_sample(nexamples, means=([0.,0.],[1.,1.]), sigma=1.):
    normal = np.random.multivariate_normal
    # squared width:
    s2 = sigma**2.
    # below, we provide the coordinates of the mean as
    # a first argument, and then the covariance matrix
    # which describes the width of the Gaussian along the
    # two directions.
    # we generate nexamples examples for each category
    sgx0 = normal(means[0], [[s2, 0.], [0.,s2]], nexamples)
    sgx1 = normal(means[1], [[s2, 0.], [0.,s2]], nexamples)
    # setting the labels for each category
    sgy0 = np.zeros((nexamples,))
    sgy1 = np.ones((nexamples,))
    sgx = np.concatenate([sgx0,sgx1])
    sgy = np.concatenate([sgy0,sgy1])
    return sgx, sgy
```

Here, we create a very small training sample with only 30 examples per category, and a test sample with 200 examples per category. We're using such a small training sample because, as will be shown in this post, small samples are very easy to overfit.

In [18]:

```
sgx, sgy = make_sample(30)
tgx, tgy = make_sample(200)
```

Still, we want our web page to be perfect, and we aren't quite yet there. The input and output prompts of our notebook don't look like the ones in the jupyter notebook, and we have no code syntax highlighting.

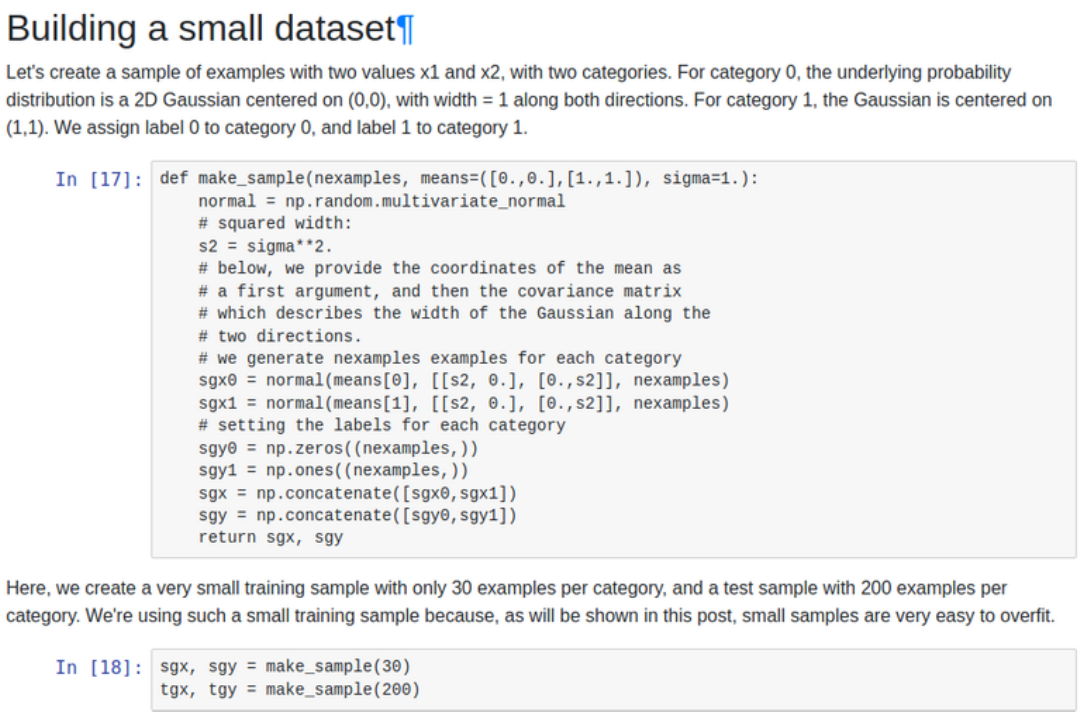
Let's first work on the notebook style.

Jupyter notebook CSS

I extracted the important part of the jupyter notebook CSS, and put that in `css/notebook.css`. I did that by hand, and there might be a way to do it more cleanly. Anyway, let's add this CSS to our template. Just edit the head section to add this CSS as shown below. And I take the occasion to improve the style even further with `css/main.css`.

```
<head>
    <link rel="stylesheet"

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
          integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
          crossorigin="anonymous">
    <link href="css/notebook.css" rel="stylesheet">
    <link href="css/main.css" rel="stylesheet">
  </head>
```

Render again, and refresh your browser. Now we get this:



Building a small dataset¶

Let's create a sample of examples with two values x1 and x2, with two categories. For category 0, the underlying probability distribution is a 2D Gaussian centered on (0,0), with width = 1 along both directions. For category 1, the Gaussian is centered on (1,1). We assign label 0 to category 0, and label 1 to category 1.

In [17]:

```
def make_sample(nexamples, means=([0.,0.],[1.,1.]), sigma=1.):
    normal = np.random.multivariate_normal
    # squared width:
    s2 = sigma**2.
    # below, we provide the coordinates of the mean as
    # a first argument, and then the covariance matrix
    # which describes the width of the Gaussian along the
    # two directions.
    # we generate nexamples examples for each category
    sgx0 = normal(means[0], [[s2, 0.], [0.,s2]], nexamples)
    sgx1 = normal(means[1], [[s2, 0.], [0.,s2]], nexamples)
    # setting the labels for each category
    sgy0 = np.zeros((nexamples,))
    sgy1 = np.ones((nexamples,))
    sgx = np.concatenate([sgx0,sgx1])
    sgy = np.concatenate([sgy0,sgy1])
    return sgx, sgy
```

Here, we create a very small training sample with only 30 examples per category, and a test sample with 200 examples per category. We're using such a small training sample because, as will be shown in this post, small samples are very easy to overfit.

In [18]:

```
sgx, sgy = make_sample(30)
tgx, tgy = make_sample(200)
```

This starts to look seriously like a jupyter notebook. And now, let's deal with ...

Jupyter code syntax highlighting with Pygments

Pygments is a (actually THE) python syntax highlighter. You give it a string with some code, it uses a "lexer" to parse the code, and generates html with syntax highlighting for the code. You can see how to use it in the Introduction and Quickstart section of the pygments documentation.

It turns out that jupyter is using pygments to highlight code. And we can find out in `templates/overfitting.html` that code cells have the class `highlight`, as these elements start with `<div class=" highlight hl-ipython3">` .

So we need to find a CSS file to style elements of this class. I already generated the CSS file to do this, in `css/pygments/notebook/colorful.css`. If you want to re-generate the file, just do:

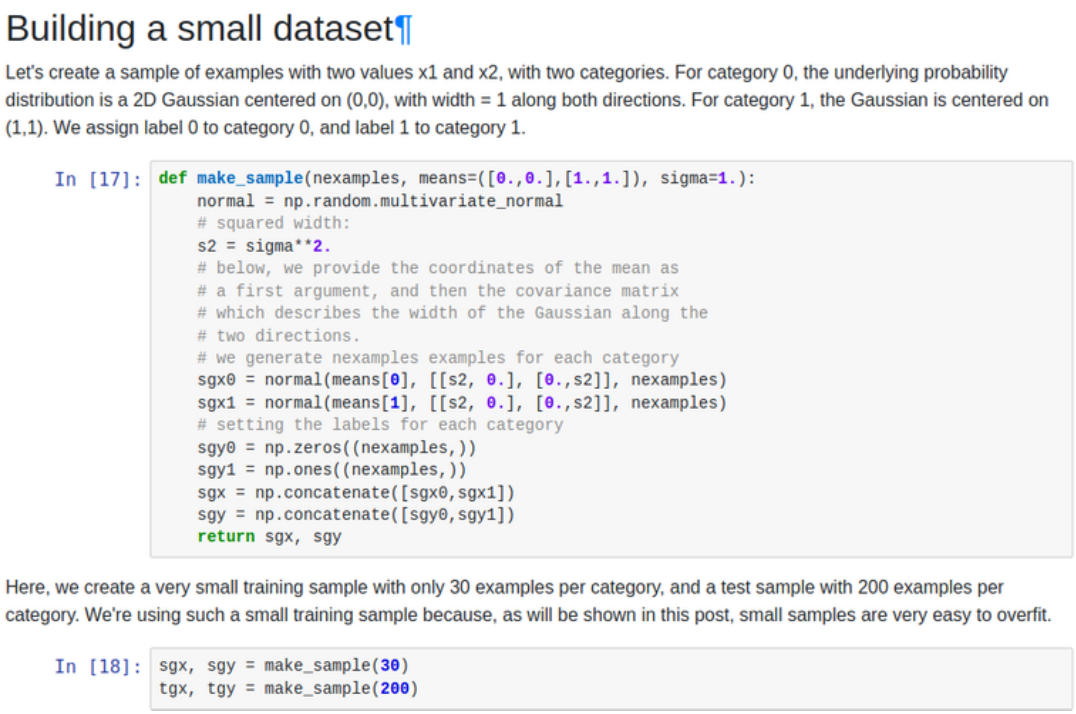
```
pygmentize -f html -S colorful -a .highlight > colorful.css
```

Then, include this CSS in your HTML header in `templates/main.html` like this:

```
<head>
    <link rel="stylesheet"

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
          integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
          crossorigin="anonymous">
    <link href="css/notebook.css" rel="stylesheet">
    <link href="css/pygments/notebook/colorful.css" rel="stylesheet">
    <link href="css/main.css" rel="stylesheet">
  </head>
```

Please note that I'm keeping `css/main.css` at the end of the header, because I want to have the final word on the style in this file.

As usual, render and refresh, and voilà! Syntax highlighting:



Building a small dataset¶

Let's create a sample of examples with two values x1 and x2, with two categories. For category 0, the underlying probability distribution is a 2D Gaussian centered on (0,0), with width = 1 along both directions. For category 1, the Gaussian is centered on (1,1). We assign label 0 to category 0, and label 1 to category 1.

In [17]:
```
def make_sample(nexamples, means=([0.,0.],[1.,1.]), sigma=1.):
    normal = np.random.multivariate_normal
    # squared width:
    s2 = sigma**2.
    # below, we provide the coordinates of the mean as
    # a first argument, and then the covariance matrix
    # which describes the width of the Gaussian along the
    # two directions.
    # we generate nexamples examples for each category
    sgx0 = normal(means[0], [[s2, 0.], [0.,s2]], nexamples)
    sgx1 = normal(means[1], [[s2, 0.], [0.,s2]], nexamples)
    # setting the labels for each category
    sgy0 = np.zeros((nexamples,))
    sgy1 = np.ones((nexamples,))
    sgx = np.concatenate([sgx0,sgx1])
    sgy = np.concatenate([sgy0,sgy1])
    return sgx, sgy
```

Here, we create a very small training sample with only 30 examples per category, and a test sample with 200 examples per category. We're using such a small training sample because, as will be shown in this post, small samples are very easy to overfit.

In [18]:
```
sgx, sgy = make_sample(30)
tgx, tgy = make_sample(200)
```

Well done! but note that we are now highlighting only the elements with class `highlight`. If you want to also write code in your HTML page (not in the jupyter notebook), you probably want this code to be highlighted in exactly the same way. That's easy, and that's what we're going to do next.

Syntax highlighting of HTML code with pygments

Let's pose the problem. We would like all the code written between `<pre>... <\pre>` tags to be properly highlighted. And we don't want to use our template engine for this, in order to limit manual editing of the template HTML code.

To do this, we will

- use beautiful soup in our rendering script to parse the HTML code, to find the `<pre>` tags;
- use pygments to produce `div` tags with highlighted code;
- replace the `<pre>` tags with these `<div>` tags, again with beautiful soup.

First, install beautiful soup:

```
pip install beautifulsoup4
```

Then, add a code section to `templates/main.html:`

```
<body>
    <div class="container">
      <h1>Integrating a Jupyter Notebook in a Web Page</h1>
      <p>Some text.</p>

      <h2>A code section </h2>
      <p>Here is some code:</p>
      <pre>
import foo
print foo.bar()
      </pre>
...
```

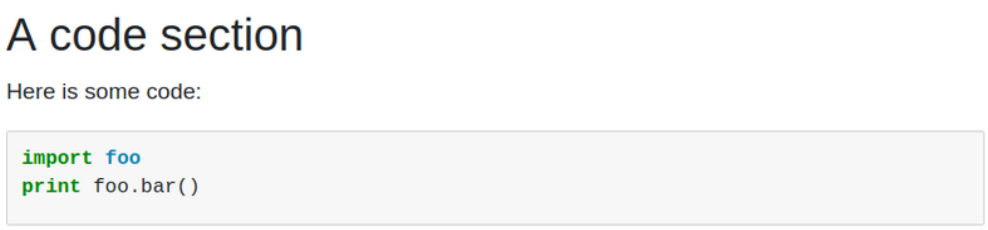
It's important to note that tabs or spaces at the beginning of each code line will appear in the formatting. So I'm taking care not to put any here.

Render again to see what it looks like:

Now, we're going to modify our rendering script to replace the `<pre>` tags by highlighted code. Here is the code. It might seem a bit long, but only because I added a lot of comments so that you can understand it easily. Please read them if you want an explanation.

```
import os

from jinja2 import Environment, FileSystemLoader

from bs4 import BeautifulSoup
from pygments import highlight
from pygments.lexers import PythonLexer, guess_lexer
from pygments.formatters import HtmlFormatter

html_formatter = HtmlFormatter()

# this tells jinja2 to look for templates
# in the templates subdirectory
env = Environment(
    loader = FileSystemLoader('templates'),
)

input_file = 'main.html'
output_file = 'index.html'

# reading the template
template = env.get_template(input_file)
# render the template
# in other words, we replace the template tag
# by the contents of the overfitting file
rendered = template.render()

# replace the pre tags by highlighted code
soup = BeautifulSoup(rendered, 'html.parser')
for pre in soup.find_all('pre'):
    # escaping pres in the jupyter notebook
    # either they're already formatted (input code),
    # or they should remain unformatted (ouput code)
    if pre.parent.name == 'div':
        pclass = pre.parent.get('class')
        if pclass and \
           ('highlight' in pclass or \
            'output_text' in pclass):
            continue
    # highlighting with pygments
    lexer = guess_lexer(pre.string)
    code = highlight(pre.string.rstrip(),
                     lexer, html_formatter)
    # replacing with formatted code
    new_tag = pre.replace_with(code)
    print(new_tag)

# create the final html string.
# formatter None is used to preserve
# the html < and > signs
rendered = soup.prettify(formatter=None)

# write the result to disk in index.html
with open(output_file, 'w') as ofile:
    ofile.write(rendered)
```

You can now render again, refresh your browser, and see the results:



Please note that, in order to get the style of this code fully consistent with the one that's coming from the notebook, I added the following fragment to `css/main.css` :

```
.highlight {
  border: 1px solid #cfcfcf;
  padding-top: 10px;
  padding-left: 10px;
  padding-right: 10px;
  border-radius: 2px;
  background: #f7f7f7;
  line-height: 1.21429em;
  margin-bottom: 10px;
}
```

Scientific equations with MathJax (and a bit of particle physics)

These are obviously a must in any scientific blog! Many scientists use LaTeX to typeset and render scientific papers in pdfs, including equations.

In web pages, we can keep using LaTeX to typeset the equations, and we will render them with MathJax.

First , add a math section to `templates/main.html` :

```
<h2>Maths</h2>

      <p>Here is a simple equation inline equation: $E=mc^2$.</p>

      <p>And for longer equations, use the standard mode math mode:</p>

      $$
      \mathcal{L}=\bar{\psi}\left(i\gamma^{\mu}\partial_  {\mu}-m\right)\psi
      $$
```

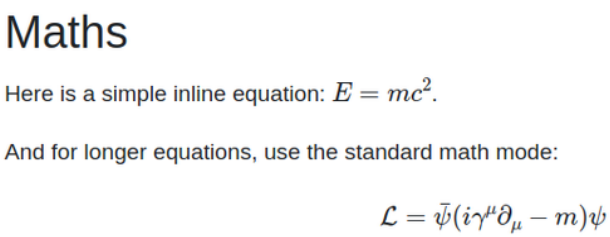
The LaTeX syntax can feel a bit awkward at first, but people get used to it!

Then, at the end of the body, add scripts to enable MathJax:

```
<!-- mathjax -->
    <script id="MathJax-script" async
src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-chtml.js"></script>
    <script src="https://polyfill.io/v3/polyfill.min.js?features=es6"></script>
    <script>
      MathJax = {
        tex: {
          inlineMath: [['$', '$'], ['\\(', '\\)']]
        },
        svg: {
          fontCache: 'global'
        }
      };
    </script>
```

The last script is a configuration fragment for MathJax. This means that I want to typeset inline equations between `$` signs. And as a consequence, I won't be able to use the dollar sign on this page anymore. If needed, I can escape it and write prices like this `\$200`.

Render and refresh again, and you will get this:



For the record, the first well-known equation shows the equivalence between mass and energy. Since the speed of light c is a constant, we just set it to one and forget about it (yes, that's what particle physicists do!). And we conclude that a particle of mass m at rest has an energy $E=m$.

And this explains why we build large colliders like the LHC at CERN. This collider accelerates two beams of protons in opposite directions so that they reach very high energies. Then, the protons are directed to collide at specific interaction points. The energy stored in the two protons that collide is converted to mass in the collision, to produce new particles. Most of these particles are well known, but if the available energy is large enough, we have a chance to create and discover heavy particles that have never been observed before by Mankind.

This is how we discovered the Higgs boson in 2012.

The second equation is the Dirac Lagrangian for free matter particles, which are called fermions. If you develop it, its first term corresponds to their kinetic energy, and the second term to their mass.

Conclusion

In this article, you have learnt how to:

- **integrate perfectly a jupyter notebook in a web page,**
- apply an elegant overall style with **Bootstrap**,
- highlight code with **Pygments**,
- render math equations with **MathJax**.

I hope you liked it, and that it will save you a few hours of work!

Please let me know what you think in the comments! I'll try and answer all questions.

And if you liked this article, you can subscribe to my mailing list to be notified of new posts (no more than one mail per week I promise.)

Back Home
