EECS16ML: Best Practices for Writing/Documenting code

Often times, “good” code does not mean writing code in the least amount of lines, or writing one method that does everything. The best practices for coding not only includes documenting everything you do, but also writing code that is readable and reusable. This is especially true when dealing with machine learning in either industry or research. Documentation may sound very boring for future data scientists and machine learning engineers, however, it is one of the most important practices when working as part of a team. In this note, we will go over the importance of clean documentation and reusable data workflows in order to create reproducible code in the context of big data and machine learning.

1. Examples

As of now, you should have a basic understanding of Pandas and it’s methods to work with data as well as clean and query data. Let’s take a look at examples of how to write good code with good documentation in the Pandas context.

Take a look at the following two coding blocks:

```
df = “some dataframe”
sum = 0
for _, row in df.iterrows():
   if row.count <= 100:
       sum += row.amount
```

```
df = "some dataframe"
#finds total sum of counts less than 100
df[df.count <= 100].amount.sum()
```

The second block not only looks much cleaner than the first but also takes less time computationally. When working with datasets in machine learning contexts, you will have to filter and clean a lot of data before actually being able to deploy the data into the models. Getting used to simplicity when using methods in Pandas will serve you very well.

Let's look at another example. Assume we have a dataframe stored in df with different tax rate information for different cities, and we want to go through and fix the tax rates to apply a certain rate while making sure Seattle contains a tax rate of 0. At first thought, you may think to just for through the rows of df and changing the values manually such as the first block. However, the second block, which uses the pandas apply method turns out to be much faster. Using the apply method to clean and filter data is another technique to develop greater code readability as well as efficiency.

```
for i, row in df.iterrows():
   if row.city == 'seattle':
       df.loc[i, 'income_tax'] = 0
   else:
       df.loc[i, 'income_tax'] = row.tax * r.rate
```

```
#helper func
def fix_income_tax(r)
  if row.city == 'seattle':
       return 0
  else:
       return  r.tax * r.rate
df['income_tax'] = df.apply(calculate_sales_factor, axis=1)
```

2. Helper Files

A key part of writing good code is reducing the number of redundant tasks to do. This is especially true in machine learning contexts. If you know you are dealing with multiple data sets that need operations such as data cleaning or filtering, then creating a helper file that has easy to access methods is very useful. Let's take a look at this example file called clean_helper.py:

```python
import numpy as np

def clean(df):

    df.columns = df.columns.str.strip().str.lower().str.replace('(',
'').str.replace(')', '').replace(' ', '_')
```

This file has a method called clean that removes unnecessary characters from column names. This could very helpful and an annoying thing to do every time you import a new dataset in your system and attempt to clean it up. Then, simply import this file:

```python
from clean_helper import clean
```

Now, anytime you deal with a new dataset that needs some cleaning, you can import this file and call the clean method accordingly. Now, this is a very simple example of a helper file, but you can see how as handling data gets more complicated, this practice will become much more useful, especially in terms of saving time.

3. Comments

One important thing missing in the coding example in the last section is the addition of comments. In machine learning contexts, you will be writing methods and functions that many other people want to use. To do this efficiently, you should add comments briefly explaining the purpose of the function and why it will help whoever wants to use it. The comments should include:

```python
import numpy as np
#this function cleans up the column names of a dataframe by replacing
spaces with underscores and removing parentheses and capitals
def clean(df):
    df.columns = df.columns.str.strip().str.lower().str.replace('(',
'').str.replace(')', '').replace(' ', '_')
```

One thing a machine learning engineer should always keep in mind is not only how he or she is going to use the code, but how others will use their code as well. Adding comments to functions is very important and will make you a more respectable programmer.

4. Pipelines

When talking about writing good and reusable code in the contexts of machine learning, data pipelines are very important. A goal of machine learning is to predict things faster and better. One thing that often gets in the way of training models faster is the actual process of collecting the data and making it ready to use. A clean and efficient pipeline that allows this process can save a lot of time. The initial stages of the workflow which include data collection and data cleaning are particularly important when designing pipelines. The data should be of good quality and easily accessible. Although you will learn sci-kit learn later on in the class, we will walk through how a simple pipeline can be constructed using the library.

First, we will need to import the library and methods:

```
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
df=pd.read_csv('data')
```

Let’s assume in this process that we want to convert data values into integers. The pipeline method essentially takes a stack of functions in the order they should be called. You can use both functions that are built-in to a library or your own custom function.

```
pipeline = Pipeline([
    ('name', name_of_function(parameters))
])

data = pipeline.fit_transform(dataframe)
```

This above piece of code is how you would construct the actual pipeline. The first argument would be the name and the second would be your function name. Let’s assume in this case it’s some function that converts values to integers, perhaps such as one-hot encoding, to make categorical variables able to be more easily classified. You can see why this process makes the data pipeline easier and reusable as in more complex contexts, once you have the pipeline set up, the process is certainly reusable.