Lab 19

Today, we will dig a bit more into evaluating the efficacy of our code via *benchmarking.* Today'a goals are:

1. Define different kinds of benchmarking
2. Deploy different kinds of benchmarking

Machine Learning at Scale

Part of working on large data and complex code is to explore how long each piece of code is taking. Today, we will return to our implementation of gradient descent and use that as our example. To prepare for that, please import the data from Lab 9 and add to the below functions (from Lab 11).

Warning!

You may get import errors for a few of the new lines. If you do, please stop your `jupyter notebook` and run the following in your **command line**:

```
conda install line_profiler
conda install memory_profiler
```

In [ ]:

```
%matplotlib inline

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

import numpy as np
from numpy import linalg as LA
import pandas as pd

import timeit ## <-- New line!
import time

import line_profiler ## <-- New line!
import memory_profiler ## <-- New line!

from sklearn import linear_model
from sklearn.ensemble import RandomForestClassifier
```

In [ ]:

```
## Functions for later use

def compute_mse(truth_vec, predict_vec):
    return np.mean((truth_vec - predict_vec)**2)

def compute_m_partial(in_vals, truth_vec, predict_vec):
    return -2*np.mean(in_vals*(truth_vec-predict_vec))

def compute_b_partial(truth_vec, predict_vec):
    return -2*np.mean(truth_vec-predict_vec)

def grad_des(input_data, truth_vec, max_steps):
    # Add your implementation for gradient descent here
    pass

def minibatch_gd(input_data, truth_vec, batch_size, max_steps):
    # Add your implementation for mini-batch gradient descent here
    pass

def stochastic_gd(input_data, truth_vec, max_steps):
    # Add your implementation for stochastic gradient descent here
    pass
```

Data

This time, we will use both the data from Labs 9 and 16. Please add a copy of each to this directory.

In [ ]:

```
## Import Data

employ_data = pd.read_csv("lab9data.csv", sep = ",")

## numpy vectors of our inputs
neuro = employ_data[["neuroticism"]].to_numpy()
perform = employ_data[["performance"]].to_numpy()
```

In [ ]:

```
## Import Data
dog_pd = pd.read_csv("lab16data.csv", sep = ",", index_col = "Breed Name")

#Change all the booleans to numbers
dog_pd.iloc[:,7:11] = dog_pd.iloc[:,7:11].astype("int")

# Export to numpy
dog_full_np = dog_pd.to_numpy()

# Split into the input variables and the target classes
in_dog_data = dog_full_np[:,:-1]
out_class = dog_full_np[:,-1]

# Get the variable names
var_names = list(dog_pd.columns)[:-1]
```

Timing our implementations with `timeit`

Last time, we used the `time` module to time how long it takes an implementation to run. Another option is to use **magic** built in python commands to check individual lines. These commands take the form of `%command` for a single line and `%%command` for a block of code. Today, we will use a few of them starting with `%%timeit`.

Noting that there can be small changes in run time, in `timeit`, we run the code several times to find the average run time. Notice how the output between `%%time` and `%%timeit` are different:

In [ ]:
```
%%timeit

# Specify and fit the model
grove = RandomForestClassifier(n_estimators=10, max_features = 3, max_depth
grove.fit(in_dog_data, out_class)
```

In [ ]:
```
%%time

# Specify and fit the model
grove = RandomForestClassifier(n_estimators=10, max_features = 3, max_depth
grove.fit(in_dog_data, out_class)
```

What do you notice?

Did one of the blocks take longer than the other one? Why do you think this is?

Your thoughts

Whare are two things that you notice that are similar about these outputs?

1.
2.

What are two things that are different about these outputs?

1.
2.

Hint Consult the prose above the code blocks for an explanation as to `time` and `timeit` offer different outputs!

Benchmarking

Last time (and above), we looked at how long a whole algorithm would take to do its work. This is a good first step, but when we *benchmark* code, we examine how fast each piece of the code is. This means that we need a method that will tell us how each piece runs. We could do this by running a time line for each individual piece of our code, OR we could use a *profiler* which gives us more nuanced information about the run times without us having to insert additional timing lines.

Benchmarking as a whole allows us to determine if we need to edit or change any lines due to *bottlenecking* (or places where the code slows down).

As a visual, consider a bottle like a glass soda bottle with a long neck. Suppose you filled the bottle with little balls, and then turned the bottle upside down quickly. Would all the balls fall out at once or would there be a sort of traffic jam at one place where the balls would have to wait before falling the rest of the way? This place where they are waiting is the *bottleneck,* hence the term "bottlenecking."

Today, we will look at three kinds of *profilers.* Specifically, we are looking at:

1. Profiling a whole script
2. Line by line Profiling
3. Memory Profiling

For this part, we need to conda install a few things:

- `line_profiler`
- `memory_profiler`

The remainder of this lab follows this blog post.

(First) Profiler

The first *magic* line that we will use is `%prun` which is the profiler command. This is akin to `cProfile` or `profile` in usual profilers for python scripting. This will tell you every piece (including deep parts of the base python) that are touched by your code.

This profiler will give us various timing information. Run the below line and consult the linked helpfile to see what the profiler is telling you.

In [ ]:
```
%prun grove.fit(in_dog_data, out_class)
```

What does it all mean?

This is a lot of information. `%prun` gives us **every** piece and **every** line (even those in deep, deep python). This output can be super overwhelming.

Let's look at a few pieces:

- First, let's look at the column names (as described here:
  - `ncalls` is the number of classes to that line.
  - `tottime` is the total running time per call (without including time calling helper functions)
  - `cumtime` is the total running time per call (including time calling helper functions)
- What gets called a lot? What gets called less? (Check out the `ncalls` column)
- Next isolate the lines are taking up a lot of time in the `tottime` column
- Now isolate the lines are taking up a lot of time in the `cumtime` column

Are these all the same lines? Does this surprise you?

Your turn

Run `%prun` on your implementation of `grad_des` and on `minibatch_gd`.

In [ ]:
```
# Run %prun on grad_des
```

Which lines are taking the most time?

Your thoughts

Which lines are called the most?

Your thoughts

What surprises you about this analysis?

Your thoughts

In [ ]:
```
# Run %prun on minibatch_gd
```

Which lines are taking the most time?

Your thoughts

Which lines are called the most?

Your thoughts

What surprises you about this analysis?

Your thoughts

Line Profiler

Instead of looking at a full script and every python code that it calls, we might want to look how much time each line (just in our program) takes to execute. This will tells us how long each line we coded takes to run.

To start this process, we will must load in the `line_profiler` as "magic" functions

In [ ]:
```
%load_ext line_profiler
```

Once loaded as "magic" we can use `%lprun` which will run our functions, timing them line by line. Let's do a silly example. Before running the line profiler, which line do you think will take longer: setting up the `grove` (line 1 of `tune_fit`) or fitting grove (line 2 of `tune_fit`)?

Add Your Thoughts *BEFORE* running the code

In [ ]:
```
def tune_fit(in_vars,classes):
    grove = RandomForestClassifier(n_estimators=10, max_features = 3, max_d
    grove.fit(in_vars,classes)
```

In [ ]:
```
%lprun -f tune_fit tune_fit(in_dog_data, out_class)
```

What does it all mean?

The output of `%lprun` gives us a line by line timing (including number of times called). Notice that the code is not running for over 30K seconds (which would be more than 8 hours), rather that the `Timer unit` is 0.00000001 of a second (or `1e-06`).

Your turn

Run `%lprun` on your implementation of `grad_des` and on `minibatch_gd` . What is surprising about these two functions?

In [ ]:
```
# Run %lprun on grad_des
```

Which lines are taking the most time?

Your thoughts

Which lines are called the most?

Your thoughts

What surprises you about this analysis?

Your thoughts

In [ ]:
```
# Run %lprun on minibatch_gd
```

Which lines are taking the most time?

Your thoughts

Which lines are called the most?

Your thoughts

What surprises you about this analysis?

Your thoughts

Memory Profiler

Instead of looking at timing of each line, we might want to look how much memory each line takes to execute as well as the total memory for the function.

To start this process, we will must load in the `memory_profiler` as "magic" functions

In [ ]:
```
%load_ext memory_profiler
```

Now that it is loaded as "magic", we can use two functions `%memit` which gives use the total memory usage and `%mprun` which gives us a line by line assessment of memory. (See bonus section below for `%mprun` .)

In [ ]:
```
%memit tune_fit(in_dog_data, out_class)
```

Your turn

Run `%memit` on your implementation of `grad_des` and on `minibatch_gd` . What is surprising about these two functions?

In [ ]:
```
# Run %memit on grad_des
```

In [ ]:
```
# Run %memit on minibatch_gd
```

Compare your results from above. What was your peak memory for each? Which one uses memory in a more efficient manner?

Your thoughts

(If you would like to see a line by line readout for memory usage, check out `mprun` below the final thoughts section.)

Why do we care?

In this lab, we've used very small examples. But as we scale up our data, it is important to pay attention to what lines use more memory and take more time. These bottlenecks can make or break our machine learning implementations, and resolving them are critical to scaling up implementations to use on larger and larger datasets.

Final Thoughts

To finish up this lab, answer two of the following three questions:

1. **What did you learn by benchmarking two versions of gradient descent? Is there anything you want to re-examine in your code?**
2. **Which is more important to consider when programming a machine learning algorithm: time or memory? Why?**
3. **We have several notions of time: seconds, iterations, and epochs. Which of these do you think is the best to use for comparing the speed of different algorithms? Why that one?**

Share your thoughts in a post on **#lab-19_submission** channel on slack with your answer.

If your have questions from this lab, post them to #lab_questions with the same preamble (i.e. starting with **Lab19**). If you have the same question, please use one of the emoji's to upvote the question. If you would like to answer someone's question, please use the thread function. This will tie your answer to their question.

Next Time

We will start our journey into deep learning.

Bonus - `mprun`

Like `prun` and `lprun`, `mprun` does a line by line read out on memory usage.

To run `mprun`, we do have to create a file for our example before we can run it. Again following the earlier blog post:

In [ ]:

```
%%file mprun_demo.py

from sklearn.ensemble import RandomForestClassifier
import numpy as np
from numpy import linalg as LA
import pandas as pd

def tune_fit(in_vars,classes):
    grove = RandomForestClassifier(n_estimators=10, max_features = 3, max_d
    grove.fit(in_vars,classes)
```

Let's again turn to our a silly example. Before running the memory profiler that we just created, which line do you think will take more memory? Setting up the `grove` (line 1 of `tune_fit`) or fitting grove (line 2 of `tune_fit`)

Your thoughts

In [ ]:

```
from mprun_demo import tune_fit
%mprun -f tune_fit tune_fit(in_dog_data, out_class)
```

Resources consulted

1. Benchmarking your code
2. IPython Magic Commands
3. Profiling and Timing Code
4. How do I get time of a Python program's execution?
5. Measure Time in Python – time.time() vs time.clock()
6. I also googled 1e-6 to check for the correct number of 0's after the decimal