Python

MEMORY USAGE AND MANAGEMENT

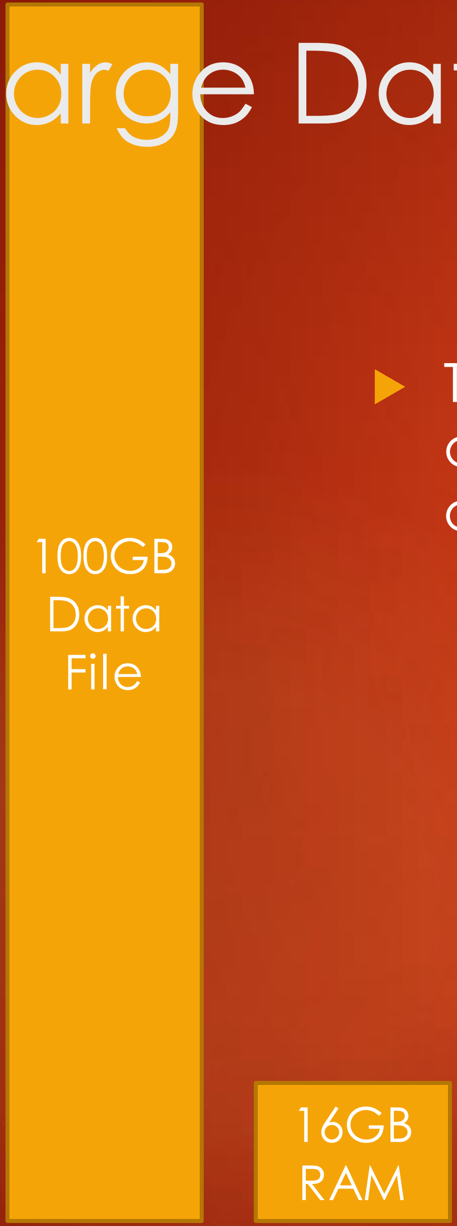
Gene Olafsen

Reference Material

- https://pythonspeed.com/
- Numbers Every Programmer Should Know By Year (colin-scott.github.io)
- Text and Image Compression based on Data Mining Perspective (codata.org)
- psutil documentation — psutil 2.2.1 documentation
- ltrace(1) - Linux manual page (man7.org)

Large Data Sets

- The demands of a large dataset on a typical development machine can push the system to run out of memory and allocations will fail.



Throw Out Some Data!

- Of course this is the first step and probably applies to all datasets. If there is obvious garbage data, incomplete data, or data that you can identify from an unreliable source… these rows may be dropped without affecting the accuracy of the trained model.

Cluster Option

- It is possible to move your processing to a small cluster of computers, but that comes with some challenges.
  - Spend money to rent the machines.
  - Possibility of a software re-write to use specific API's to harness the power of the cluster.

Out-of-core Computation

- Continue to process the data locally.
  - Compress
  - Chunk
  - Index

RAM/SSD and Latency

- While SSD's can indeed be much quicker than mechanical hard drives, they are still orders of magnitude slower than data retrieval from RAM.
  - RAM data access: 100ns
  - SSD data access: 16,000ns
- If you're current computer contains mechanical hard drives, it is time to replace those with SSDs. SSDs are typically 100X faster than mechanical drives. Even though the goal is not to have to read data from storage, there is still a need to have the fast access possible.

RAM/SSD and Latency Options

- Spend the money to upgrade your local system with more memory. Time is money and this may be a cheaper solution in the long run.
  - Current prices for 128GB RAM modules run about $1,000
- Rent a VM in the cloud.
  - A Google VM with 80 cores and 320GB RAM, for $3.82/hour.

Compression

- Data compression is typically divided into two categories:
  - Lossless: The compressed data has the exact same information as the original data.
  - Lossy: The compressed data loses some of the details in the original data, but in a way that ideally doesn't impact the results of your calculation very much.
- However for the sake of this discussion, we are tolking about "in memory" data representation.

Compression

- Simple substitution can be used to represent data equivalence.
  - "True" or "False" in the column of a file can be stored as a 1 or 0. This can either be a single byte in a column, or a single bit, if combined with other boolean values from other columns.

Click to add text

- Clustering based Huffman Encoding(CBH) or similar can be used to represent text; where an ID is assigned to each unique word.
- One-hot encoding may provide a smaller in-memory representation for some datasets.

Feature Hashing

- Feature hashing, also known as the hashing trick. is a fast and space-efficient way of vectorizing features, i.e. turning arbitrary features into indices in a vector or matrix.
- It works by applying a hash function to the features and using their hash values as indices directly.
- Scikit-learn provides a class named **FeatureHasher** which is a high-speed, low-memory vectorizer
- The **FeatureHasher** applies a hash function to the features to determine their column index in the matrices directly.
- **Note**: The hash function works one-way. The hasher does not remember what the input features looked like and has no inverse_transform method.

FeatureHasher

```
import pandas as pd
from sklearn.feature_extraction import FeatureHasher

test = pd.DataFrame({'type': ['apple', 'bear', 'adam', 'comet', 'bean','zoo','ye
llow','xray']})
h = FeatureHasher(n_features=3, input_type='string')
f = h.transform(test.type)
f.toarray()

array([[-2., 1., 2.],
       [ 0., 2., 0.],
       [ 0., 2., 2.],
       [-1., 3., -1.],
       [-1., 1., 0.],
       [-1., 2., 0.],
       [ 0., 2., 2.],
       [ 0., 2., 2.]])
```

Chunking

- Chunking is useful when you need to process all the data, but don’t need to load all the data into memory at once. Instead you can load it into memory in chunks, processing the data one chunk at time.

Incremental PCA

- **Incremental Principal Component Analysis** (IPCA) is used to address the biggest limitation of **Principal Component Analysis** (PCA) and that is PCA only supports batch processing, means all the input data to be processed should fit in the memory.
- The Scikit-learn ML library provides **sklearn.decomposition.IPCA** module that makes it possible to implement Out-of-Core PCA either by using its **partial_fit** method on sequentially fetched chunks of data or by enabling use of **np.memmap**, a memory mapped file, without loading the entire file into memory.

Indexing

- Indexing is useful when you only need to use a subset of the data, and you expect to be loading different subsets of the data at different times.
- Imagine you had text about different topics, including: "fashion", "real estate" and "food". Indexing would tell you where to look for the specific passages. (Whereas 'chunking' would require you to parse through the entire corpus to locate the text that satisfies the category of interest.)

Memory Management

- Python memory allocation can be checked with the psutil library.

```
import psutil
psutil.Process().memory_info().rss / (1024 * 1024)
123.567  # displayed in MB
```

- Machine learning applications typically load datests into a NumPy array or a Pandas DataFrame.

What is psutil?

- psutil is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network).
- It is useful mainly for system monitoring, profiling and limiting process resources and management of running processes.
- It implements many functionalities offered by command line tools such as: ps, top, lsof, netstat, ifconfig, who, df, kill, free, nice, ionice, iostat, iotop, uptime, pidof, tty, taskset, pmap.

Memory Management

- Python memory allocation can be checked with the psutil library.
- In the following case, we will consider the current process and return a tuple representing the Resident Set Size (RSS) and Virtual Memory Size (VMS).

```
import psutil
psutil.Process().memory_info().rss / (1024 * 1024)
123.567  # displayed in MB
```

- Note: On UNIX rss and vms are the same values shown by ps. Whereas on Windows, rss and vms refer to the “Mem Usage” and “VM Size” columns of taskmgr.exe.

Virtual Memory Lesson

- Behind the scenes, Python uses the C library malloc and mmap functions to allocate memory from the operating system. (malloc used for smaller memory requests, mmap for larger ones)
- The operating systems that host Python implement a virtual memory scheme. Where 'memory' is paged in and out of physical memory to disk. Those pages that are memory resident are known as the *working set*.
- Next, let's look at a utility that can be used to monitor memory allocations...

ltrace

- ltrace is a program that simply runs the specified command until it exits. It intercepts and records the dynamic library calls which are called by the executed process and the signals which are received by that process. (It can also intercept and print the system calls executed by the program.)
- Given the following Python program:

```
import numpy as np
arr = np.ones((123_000,), dtype=np.uint8)
```

- We see the following result when running Python under ltrace:

```
$ ltrace -e malloc python memoryalloc.py
...
_multiarray_umath.cpython-39-x86_64-linux-gnu.so-
>malloc(123000) = 0x4527862a23e0
```

Working Set and Memory Pressure

- The operating system will decide which memory pages to locate in RAM (fast) and those that are paged to disk (slow).
- We can see the effect of this by opening a few applications which will demand RAM-backed memory allocations and executing psutil again to view *working set* memory.

```
import psutil
psutil.Process().memory_info().rss / (1024 * 1024)
88.132  # displayed in MB
```

- Here we see that the memory in use reported from dropped from 123MB to 88MB.

Take Aways

- There are tools that can help report on an application's memory utilization.
- Operating systems will swap 'unused' memory pages to disk when not being directly accessed.
- When your Pythong application is executing as the 'focused' application, the operating system will prioritize resources for the process.

Program Structure and Memory

- There are a number of steps that a data scientist or machine learning specialist can do to help reduce memory pressure on an application.

A Normalize Function

- Consider the following code:

```
import numpy

def normalize(array: numpy.ndarray) -> numpy.ndarray:
    """
    Accepts a floating point array.
    Returns a normalized array with values between 0 and 1.
    """
    low = array.min()
    high = array.max()
    return (array - low) / (high - low)
```

Executing Normalize

- Executing the code yields:

```
arr = np.array ( [1, 2, 3, 4, 5])
k = normalize(arr)
print(k)

[0. 0.25 0.5 0.75 1. ]
```

- Looks good... but from a memory perspective, what happened?
  - What happens if we print variable *arr*?

Copying Data Requires Memory

- So, variable *arr* will remain untouched.
- Another array (k) was allocated to contain the result of the function.
- Now imagine if the array was 3GB, or 30GB!

Normalize In-Situ

- Here is the normalize function reworked so that the array which is passed in, is modified *in place* and a new array is not returned.

```
def normalize_in_place(array: numpy.ndarray):
    low = array.min()
    high = array.max()
    array -= low
    array /= high - low

arr = np.array ( [1, 2, 3, 4, 5])
normalize_in_place(arr)
print(arr)
```

- What do you expect to be printed?

Not Expected...

- The program fails to execute.

```
TypeError Traceback (most recent call last)
 <ipython-input-1-54b391eb8e32> in <module>()
     29
     30
---> 31 normalize_in_place(arr)
     32 print(arr)
<ipython-input-1-54b391eb8e32> in normalize_in_place(array)
      6 high = array.max()
      7 array -= low
----> 8 array /= high-low
      9
     10 def normalize(array: np.ndarray) -> np.ndarray:
TypeError: No loop matching the specified signature and casting was found for
ufunc true_divide
```

Type Issue

- The /= operator is not allowed with an integer- because the operation will result in a float. Change the input array to be defined as float.

```
arr = np.array ( [1, 2, 3, 4, 5], dtype=np.float64)
normalize_in_place(arr)
print(arr)

[0. 0.25 0.5 0.75 1. ]
```

- The arr variable is now modified in-place and an extra allocation does not take place to contain the result.
- If we ran psutil before and after this operation, the results would indicate an additional memory allocation.

Mutation

- Many NumPy APIs include an out keyword argument, allowing you to write the results to an existing array, often including the original one.
- Pandas operations usually have an inplace keyword argument that modifies the object instead of returning a new one.

NumPy, Numbers and Memory

- If you thought storing a million integers into an array, where each integer comfortably fits in a 64-bit integer, would consume about 8MB of memory… think again.

```
list_of_numbers = []
for i in range(1000000):
    list_of_numbers.append(i)
```

- Python, like other weakly yped languages (think JavaScript) store numbers as objects.

Python GetSize

- So if we ask Python to get the size of an integer, it returns the 28 bytes. As such, we would expect one million integers to consume 28MB of memory…

```
import sys
sys.getsizeof(4000)

>> 28
```

- Why 28 bytes for an integer?

Numbers as Objects

- Every Python object (in the default CPython implementation) is back bye a C struct named PyObject

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject;
```

- These are:
  - The data that is to be stored.
  - A reference count (for garbage collection)
  - A pointer to the object type.

NumPy is Lean

- NumPy arrays are not collections of Python objects.
- NumPy stores numbers in their 'native' format.
- Memory usage for one million integers will be 8MB, plus a little more to account for the overhead of NumPy.

```
import numpy as np
arr = np.zeros((1000000,), dtype=np.uint64)
for i in range(1000000):
    arr[i] = i
sys.getsizeof(arr)

>>8000096
```

Reducing Python Memory

- As we have seen in the previous slides, Python can allocate much more memory than expected. Part of this is because Python is a weakly typed (duck-typed) language, and also because it relies on a garbage collector to release unused (un-referenced) objects.
- The object overhead may not be apparent with a few hundred objects. Once the allocations happen in the millions, then there are techniques that can be helpful.
- We will look at additional ways to recognize and reduce Python memory usage.

Eliminate a Python Object's Hidden Dictionary

- Behind the scenes, every Python class stores its instance attributes in a dictionary.
- Having a dictionary for every object makes sense if you want to add arbitrary attributes to any given object... HOWEVER, most of the time the class attributes are known and don't change.

```
class MyClass(object):
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
        self.set_up()
```

__slots__

- The second piece of code will reduce the burden on your RAM.
- In fact, some people have seen almost 40 to 50% reduction in RAM usage by using this technique.

```
class MyClass(object):
    __slots__ = ['name', 'identifier']
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
        self.set_up()
```

Object Pivot

- Consider a class that stores coordinates for a three-dimensional point, x,y and z.
- Assume that you need to manage 1M points.
- A point class that maintains each point separately.

```
class Point:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

Object Pivot Cont.

- Manage a structures that collects the points and then dereference the points by index.

```
points = {
    "x": [1,2,3],
    "y": [4,5,6],
    "z": [7,8,9]
}

print(points["x"][1])
print(points["y"][1])
print(points["z"][1])

2
5
8
```

Object Pivot Results

- Such an approach may decrease memory usage by 85%
- Of course for even better memory utilization, espcially when working with floating point numbers, you can reduce memory usage even further by using a Pandas DataFrame to store the information: it will use NumPy arrays to efficiently store the numbers internally.
- (All roads lead back to NumPy!)