BEST PRACTICES FOR A BLAZING FAST MACHINE LEARNING PIPELINE

David Liu, *Python Technical Consultant Engineer*

Intel Corporation

Overview

- Introduction
- Tools and Techniques
- Data preprocessing
- *Break (15 min)*
- Data exploration
- Machine Learning
- Options for scaling and pipelining
- *Break (15 min)*
- Hands-on: Advanced tools
- Hands-on: Chaining it together
- Summary

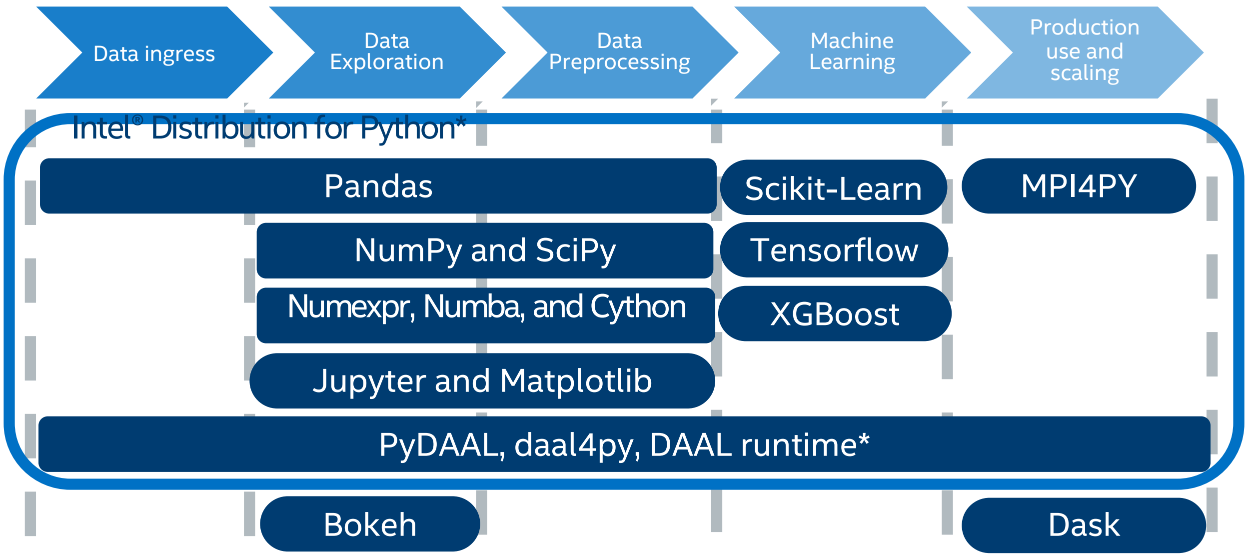
Introduction

- The use of Machine Learning (ML) in industry has risen to the point of which it is hard to ignore, but navigating to find the best practices is difficult
- Increasing and rapidly changing number of tools in frameworks in the space
- Dialing in a core set of tools and processes to handle ML in the day-to-day is the optimal way to handle things

Introduction: How the ML pipeline looks in practice

- ML Pipeline for a Data Scientist extends to many different steps with a wide range of tools
- Some tools range over each of the areas, but others only cover certain tasks
- No “perfect” tool, but choosing the right tool for the need is best practice

Introduction: How the ML pipeline looks in practice



Tools of the Trade

- One of concepts of design the Intel® Distribution for Python* is to provide an accelerated and well tested + constructed collection of necessary tools for the Data Science process
- Each step of the process is an expansive space (and deep dive) in its own right; today is just a primer on them
- The tools continue to change and evolve over time, so it is best to learn the techniques and fundamentals when possible
- Occasionally a mix of tools in each of the spaces is required to tackle different parts of the problem

THE BASICS

Understanding Python Performance

Python performance overview

Per the creator of the language (and the language direction), the focus of the language, it Python was not meant to be "fast"

The focus on the language was to be *expressive and quick to prototype*

However, its usage is only picking up in numerical, scientific, and machine learning world

Unusually, Python and C are the perfect pair; Python has been made to build and access C libraries with ease

Understanding Python Performance (Con't)

General Python behavior

Python works by providing an interpreter (Cpython) which runs one's Python commands from *Python Bytecode* (.pyc)

Pathway from one's code is: *Lexing, parsing, compiling, interpreting*

Splits into function and code objects

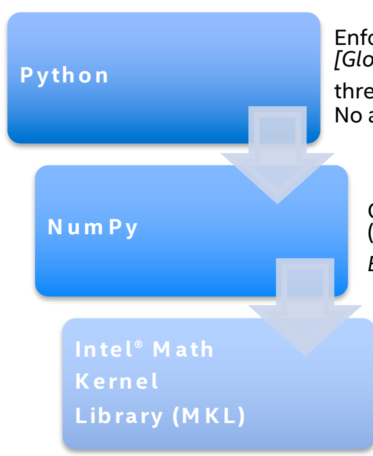
Compiling is not "standard"; doesn't go down to x86 instructions, but instead to the Python interpreter

This format allows for very *flexible* bytecode, and the Python interpreter is the main proponent of this

Understanding Python Performance (Con't)

**Why does this matter?
(Example with numerical flow)**

- The Python language is interpreted and has many type checks to make it flexible
- Each level has various tradeoffs; *NumPy*[NumPy]* value proposition is immediately seen
- For best performance, escaping the Python layer early is best method in this case



Enforces *Global Interpreter Lock* (GIL) *[Global Interpreter Lock]* and is single threaded, Abstraction overhead, No advanced types

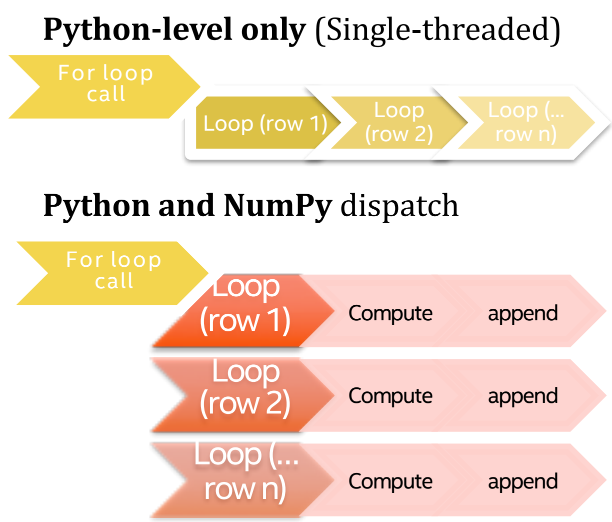
Gets around the GIL (multi-thread and multi-core)
BLAS API can be the bottleneck
*Basic Linear Algebra Subprograms (BLAS) [CBLAS]

Gets around BLAS API bottleneck
Much stricter typing
Fastest performance level
Dispatches to hardware vectorization

Understanding Python Performance (Con't)

Why does this matter? (Python levels)

- Example with array loops
- GIL will force loops to run in a single threaded fashion
- NumPy dispatch helps get around single-threaded by using C functions
- C functions can then call processor vectorization
- *Getting out of Python layer for performance is key*



Introducing the Intel® Distribution for Python* 2018

- The **Intel® Distribution for Python*** was created as a response to the needs of **Data Scientists, engineers**, and those in **HPC**
- It utilizes advanced *runtime libraries* to harness the power of the Intel® hardware transparent to the user, **so no code changes required**
- Accelerates popular packages such as **NumPy, Pandas, Scikit-Learn, Tensorflow** through direct code changes linking to the *runtime libraries*
- Available on **Anaconda** and **pip**, through **Docker**, or as **standalone installation**
- Distribution is **free, even for commercial use**

What's Inside Intel® Distribution for Python*

High Performance Python* for Scientific Computing, Data Analytics, Machine & Deep Learning

FASTER PERFORMANCE	GREATER PRODUCTIVITY	ECOSYSTEM COMPATIBILITY
Performance Libraries, Parallelism, Multithreading, Language Extensions	**Prebuilt & Accelerated Packages**	**Supports Conda & PIP**
Accelerated NumPy/SciPy/scikit-learn with Intel® MKL[1] & Intel® DAAL[2] Data analytics, machine learning & deep learning with scikit-learn, pyDAAL, Caffe*, Theano* Scale with Numba* & Cython* Includes optimized mpi4py, works with Dask* & PySpark* Optimized for latest Intel® architecture	Prebuilt & optimized packages for numerical computing, machine/deep learning, HPC, & data analytics Drop in replacement for existing Python - No code changes required Jupyter* notebooks, Matplotlib included Free download & free for all uses including commercial deployment	Compatible & powered by Anaconda*, supports conda & pip Distribution & individual optimized packages also available at conda & Anaconda.org, YUM/APT, Docker image on DockerHub Optimizations upstreamed to main Python trunk Priority Support through Intel® Parallel Studio XE

Intel® Architecture Platforms

Operating System: Windows*, Linux*, MacOS[1]*

1Intel® Math Kernel Library

2Intel® Data Analytics Acceleration Library

1Available only in Intel® Parallel Studio Composer Edition.

Installing the Intel® Distribution for Python* 2018

Stand-alone installer and anaconda.org/intel

Download full installer from
https://software.intel.com/en-us/intel-distribution-for-python

OR

```
> conda config --add channels intel
> conda install intelpython3_core
> conda install intelpython3_full
```

```
docker pull intelpython/intelpython3_full
```

Linux*

Windows*

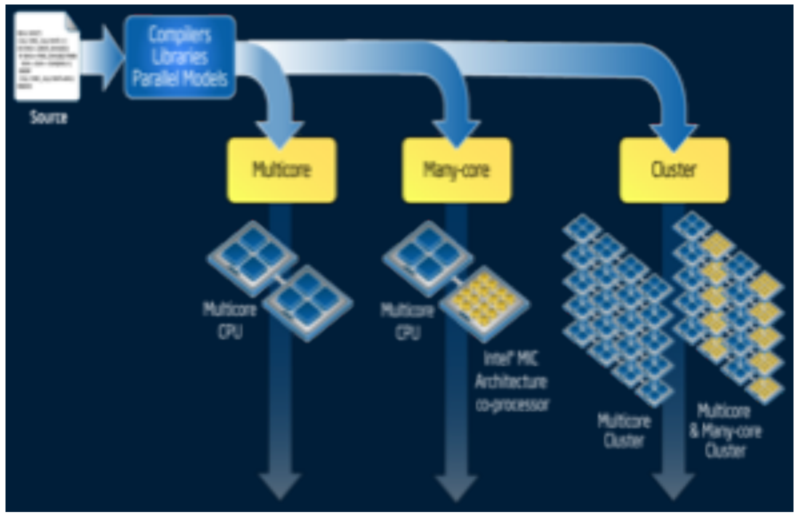
macOS*

```
Apt/Yum,
pip also
available
```

From Single Core, to Multicore, to Many Core

Purpose of libraries is to help scaling of code over various types of hardware

These are some of the ways we've accelerated NumPy*/SciPy* /Scikit-learn*



Tools of the Trade (slight return)

Technologies relied upon as a starter:

- **Base Language**: *Python*
- **Numerical/Scientific**: *NumPy, SciPy, Numba, Cython, Numexpr*
- **Data preprocessing and manipulation**: *Pandas, Dask, Intel® DAAL*
- **Machine Learning**: *Scikit-Learn, Intel® DAAL*
- **Distributed work:** *Dask, MPI4PY*
- **Visualization**: *Matplotlib, Bokeh*
- **IDE or work area:** *Jupyter Notebooks* and the shell w/ *IPYTHON or Python*
- **Code Profiler:** Optional cprofile, line_profile, Intel® VTune Amplifier

PREPROCESSING

Data ingress/egress

- Getting the data in and out of Python can be simple or be the bane of one's existence
- Several roadblocks:
  - Python Object size, Global Interpreter Lock (GIL), Serialization in and out of Python
- Formats
  - csv, xlsx, hdf5, txt…
  - Movement from one to another? Mixed formats? Not on one node?

Data ingress/egress (con't)

- Data movement is expensive, try to do it once and hold it there for data science tasks
  - Load into Pandas via Jupyter notebook
  - Use IPython
  - Load into dask or a dask dataframe
  - If in Spark, leave in cluster until ready to do the final calculation in engine
  - If one must exit the application, save it to a format that can be reloaded easily

Preprocessing

- Preprocessing: The real 90%
- Tools of the trade
- Types of parallelism
- Distributed: Dask and MPI4PY
- Vectorization: NumPy, Numba, Numexpr, Cython

The 90%

- Often, the majority of the time spent by data scientists or ML engineers is in preprocessing
- This has been made many times worse by the increasing size of datasets and feature complexity over the last few years
- Rather than just focus on the Training and Prediction, focus on growing task that precedes it as place of optimization and process improvement
- What ways are there to get the most out of one's preprocessing?

General Preprocessing

- General Data munging
  - **Pandas** is the de facto standard package when working with data. It is a framework that encompasses relational-style calls with series and dataframe primitives
  - The ability to quickly thin down datasets and correct for datatypes before analysis and machine learning is part of the "munging" process
  - Data is typically dirty, and as such this framework grants quick and easy methods of getting to one's initial "clean" dataset

General Preprocessing (con't)

- General Data transforms
  - Say for example one of the transforms to the data is expensive—a complicated mathematical function
    - Use **numba or numexpr** to transform the data with **vectorized funcs** that **exit the GIL**
    - Use accelerated capabilities of the **Intel® Data Analytics Acceleration Library (DAAL)** in Scikit-learn or **PyDAAL** for supported preprocessing, i.e. **Principal Component Analysis** (PCA) or **Singular value decomposition** (SVD)

Tools of the Trade for Preprocessing

- **Python**
  - Easy processing, string manipulation, i/o in a single language
- **Pandas + NumPy**
  - Dataframe munging and simple transforms
- **Scikit-Learn**
  - General preprocessing included with ML library
- **PyDAAL, daal4py**
  - Pipelining and some advanced preprocessing
- **Dask, MPI4PY**
  - Distributed work (multi-node or out of core)

Many types of Parallelism

Parallelism is the best way to achieve performance gains in Python

Examples:

- *Message passing*
  - MPI4Py*, Dask*
- *General parallelism*
  - multiprocessing, Dask*
- *Multi-format parallelism*
  - Cython*, Numba*
  - TBB, OpenMP are backends/runtimes
  - Numexpr*, NumPy*, et al.

At lower levels: OpenMP, TBB, and MKL, DAAL calls

Distributed computing landscape

mpi4py | pySpark | Dask/distributed | ...

- New distributed computing technologies appear almost every year
- These technologies help Python achieve task-based parallelism and mitigate the issues that many people have with Python

Two different flavors of Distributed: Dask and MPI4PY

MPI4PY*

- Access to the MPI Library at the Python level
- Accelerated with Intel® MPI Library
- Best for composing things that have complex relationships

Dask*

- Framework that uses distributed futures to construct tasks graphs and execute via a scheduler
- Specialized for computational workloads (numerical Python parallelism), and comes with a lot of built-in functionality

MPI4PY

- Allows one to utilize the Message Passing Interface (MPI) with the Python language
- Designed for the parallel computing world
- Can handle very complex relationships that don't necessarily fit "templates" of other distributed task frameworks

```
from mpi4py import MPI
import numpy

def matvec(comm, A, x):
    m = A.shape[0] # local rows
    p = comm.Get_size()
    xg = numpy.zeros(m*p, dtype='d')
    comm.Allgather([x,  MPI.DOUBLE],
                   [xg, MPI.DOUBLE])
    y = numpy.dot(A, xg)
    return y
```

Image From MPI readthedocs

Dask

- Easy way of accessing distributed task-parallelism in the NumPy*/SciPy* ecosystem
- Comes with Task Graphs, Delayed wrappers, diagnostic server
- Can scale up and down quickly depending on needs (local computer, full cluster)

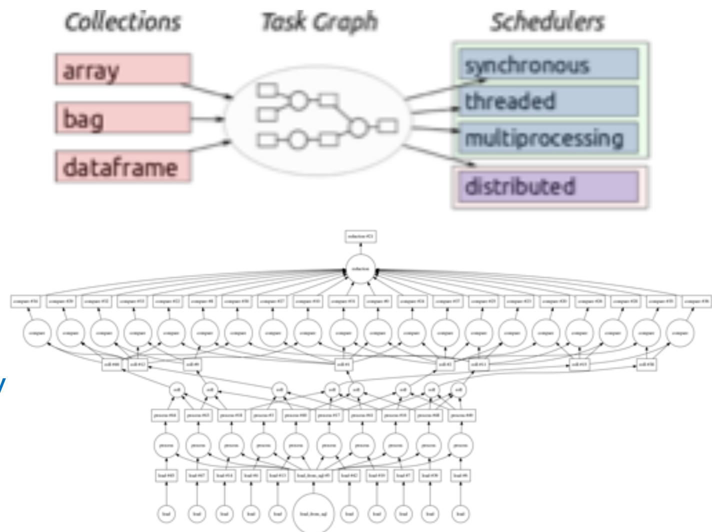


Image From Dask documentation*

Dask (Con't)

- Extremely easy to integrate in places where NumPy* and SciPy* already exist
- Is a bit "heavier" of a solution than MPI, so use accordingly
- It does well with Task graph (i.e. Task parallel) or concurrent future-style of async
- Works best when tasks have little intercommunication between workers

Other Python-level Accelerators

Cython*

- Optimizing static compiler
- Similar syntax to Python
- Can interact with NumPy* pretty well
- Supports calling C/C++ well

Numba*

- Just-in-time (JIT) certain functions in Python
- Optimizes down to Low Level Virtual Machine (LLVM) code
- Useful for code that can be instantiated once and reused

Vectorization

- Special form of parallelism converted from an initial scalar form
- Hardware supported parallelism of SIMD which can greatly assist numerical pipelines
- Main two components are numexpr* and the NumPy* that use vectorization
- Intel® Distribution for Python* does this for you with changes to NumPy*, SciPy*, Scikit-learn* etc.
- Occasionally using the raw numexpr* might fit one's use case

Cython

- Can statically compile native code
- Can utilize static typing for faster code
- Compiles to C files
- Can pre-compile and import Cython code/modules
- Accessed with a package or via the %%cython in Jupyter notebooks

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Code from the Cython documentation

Caveats

From the Cython docs:

“The general recommendation is that you should only try to compile the critical paths in your code. If you have a piece of performance-critical computational code amongst some higher-level code, you may factor out the performance-critical code in a separate function and compile the separate function with Numba. Letting Numba focus on that small piece of performance-critical code has several advantages:

- *it reduces the risk of hitting unsupported features;*
- *it reduces the compilation times;*
- *it allows you to evolve the higher-level code which is outside of the compiled function much easier.”*

NUMEXPR: the numerical Evaluator

- Multi-core, multi-threaded vectorization performance through Vector Math Library (VML), part of the Intel® MKL
- Best on large array size calculations, and transcendent expressions
- Callable from the Python-level
- Great for making changes that could call down to vectorization code without moving one's code to C++ level

```
In [1]: import numpy as np

In [2]: import numexpr as ne

In [3]: a = np.random.rand(1e6)

In [4]: b = np.random.rand(1e6)

In [5]: timeit 2*a + 3*b
10 loops, best of 3: 18.9 ms per loop

In [6]: timeit ne.evaluate("2*a + 3*b")
100 loops, best of 3: 5.83 ms per loop   # 3.2x: medi

In [7]: timeit 2*a + b**10
10 loops, best of 3: 158 ms per loop

In [8]: timeit ne.evaluate("2*a + b**10")
100 loops, best of 3: 7.59 ms per loop   # 20x: large
```


NUMEXPR (Con't)

- Easy to intermix with NumPy* and SciPy* code
- Requires that you understand the numerical implications of your code
- This was one of the methods we accelerated NumPy* and SciPy* in our optimized IDP Package

```
>>> import numpy as np
>>> import numexpr as ne

>>> a = np.arange(1e6)   # Choose large arrays for better speedups
>>> b = np.arange(1e6)

>>> ne.evaluate("a + 1")   # a simple expression
array([  1.00000000e+00,   2.00000000e+00,   3.00000000e+00, ...,
         9.99998000e+05,   9.99999000e+05,   1.00000000e+06])

>>> ne.evaluate('a*b-4.1*a > 2.5*b')   # a more complex one
array([False, False, False, ...,  True,  True,  True], dtype=bool)

>>> ne.evaluate("sin(a) + arcsinh(a/b)")   # you can also use functions
array([        NaN,  1.72284457,  1.79067101, ...,  1.09567006,
        0.17523598, -0.09597844])

>>> s = np.array(['abba', 'abbb', 'abbcdef'])
>>> ne.evaluate("'abba' == s")   # string arrays are supported too
array([ True, False, False], dtype=bool)
```

Numba

- Accessed by using the @jit decorator
- May need special compilation options to get best out of it
- Can cache the function with cache=True
- Access vectorization with @vectorization decorator

```
from numba import jit

@jit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255
```

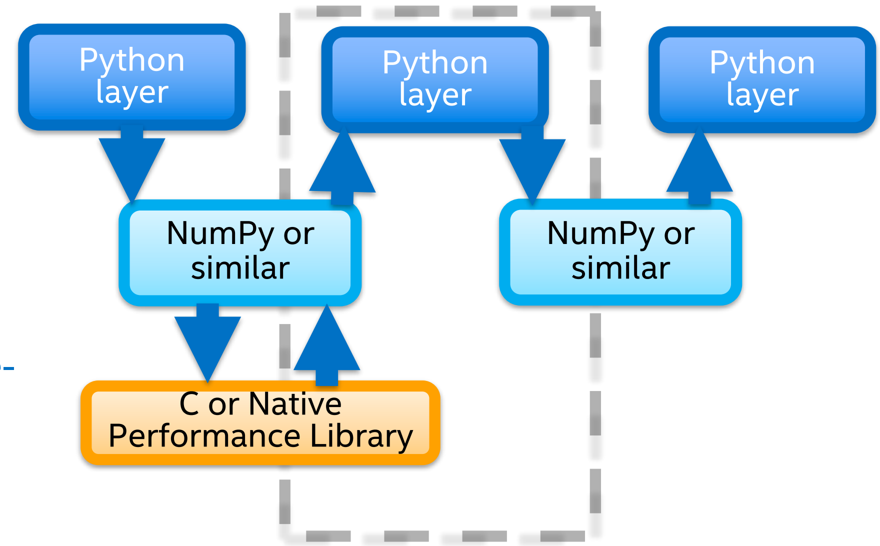
Code snipit from the Numba documentation

Parallelism and other tools: Usage Details

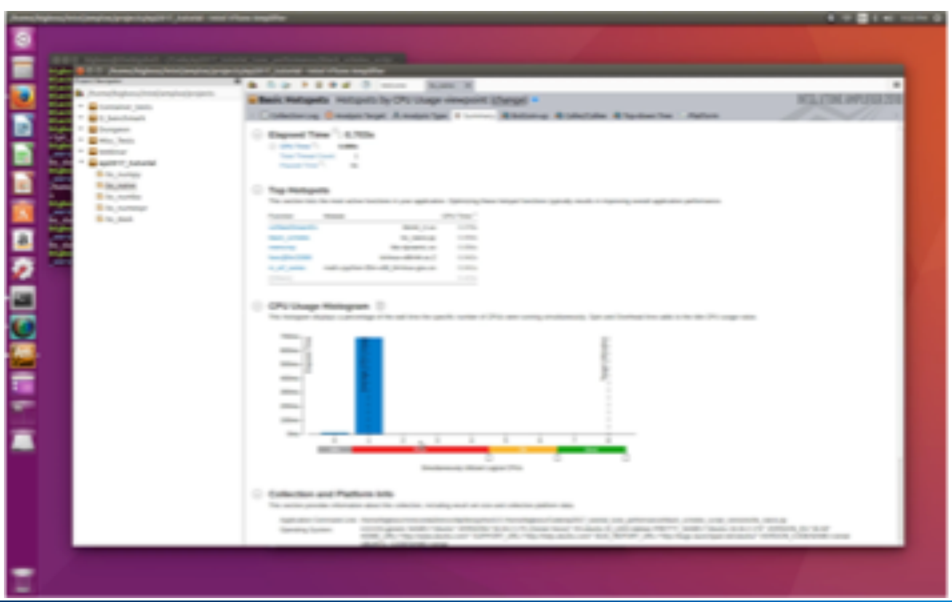
- Clearly understand one's workload and algorithms before implementing anything with these tools
- Profile one's code to more accurately understand where to make code changes
- Try different strategies and mixes of optimization to see where balance point is
- Documentation is you friend: many of these technologies have lots of gotchas and implementation quirks

Python computation behavior

- Worst case-you have to make multiple trips through the top layer of Python
- This extra trip bottlenecks the code back to single-threaded land as it goes back to Python



Intel® VTune Amplifier example



Intel® VTune™ Amplifier Details

Line-level profiling details:

- Uses sampling profiling technique
- Average overhead ~1.1x-1.6x (on certain benchmarks *)

Cross-platform:

- Windows and Linux (Viewer-only on OSX)
- Python 32- and 64-bit; 2.X, 3.X versions

* Measured against Grand Unified Python Benchmark

Machine specs: HP EliteBook 850 G1; Intel® Core™ i5-4300U @ 1.90 Ghz (4 cores with HT on) CPU; 16 GB RAM; Windows 8.1 x86_64

Profiler Summary

Profilers should be the first step when after a visual inspection does not net performance advantages

Without Code Profilers, one is pretty much lost without the insight provided by them, *especially with the complexity of Python*

Each of the open source profilers have different aspects they are good at (or that they can see), so use accordingly

Tools such as Intel VTune™ provide source, function, and hardware level information if the open source profilers aren't enough

Test often, and if in doubt profile your code!

BREAK

43

DATA EXPLORATION

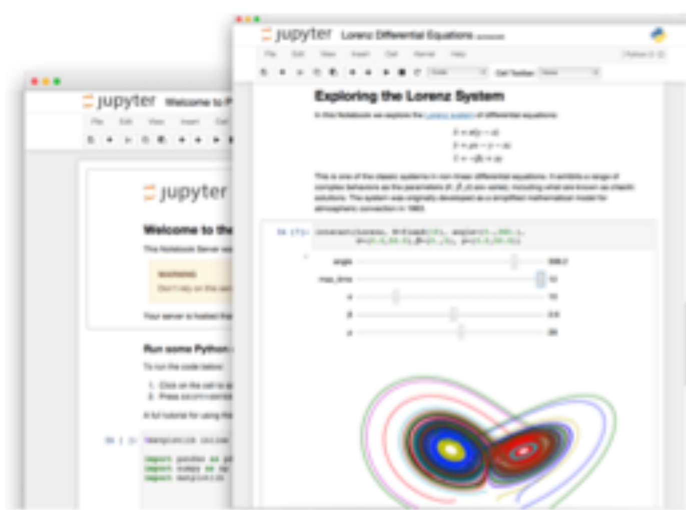
44

Data Exploration

- One of the most important things to do is *visualize* the data you are working with
- This means working with it in an iterative and journalistic way, which is where **Jupyter Notebooks** come into handy
- Integrated features from **Pandas** and **Matplotlib** give easy and interactive access to datasets quickly within Jupyter Notebooks
- Frameworks such as **Bokeh** do a good job on making interactive visualizations for those who need to utilize it
- Saving and sharing the notebooks makes for useful collaboration technique

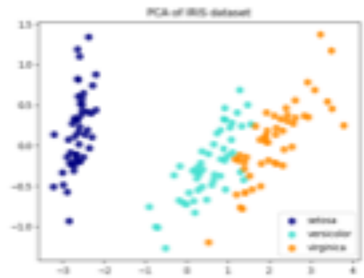
Data Exploration: on Jupyter

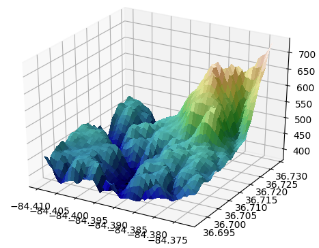
- Built with the IPython Kernel and *feature-rich plugins*, this Display system allows for **Browser-based development** of Python
- The tool of choice because of the *iterative* nature of running the cells and the **markup options** for **documentation**



Data Exploration: on Matplotlib

- One of the original visualization libraries created for the NumPy/SciPy community
- Advantage of having rich integration with the scientific and numerical datatypes, as well as plugin integration into Jupyter with *%matplotlib inline*



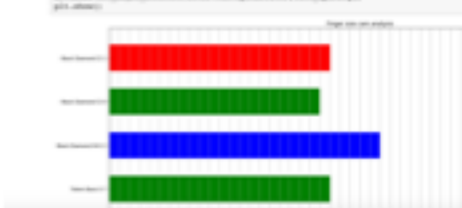


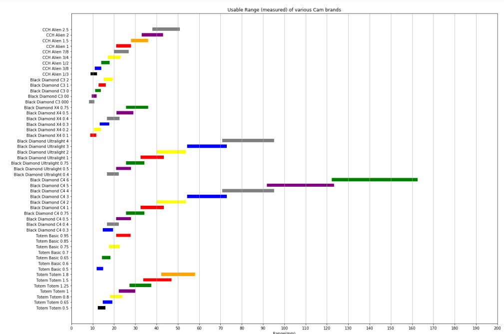
Data Exploration

- General flows within the exploration process
  - Move it into a **dataframe** to make it easy to explore, describe, and munge through the data
  - If using Python, **pandas** is the preferred framework to do this in
  - If requiring a *large transform or normalization* of the data, using **NumPy** or more powerful technologies (**numba, numexpr**) to do the **mathematical transforms** may be necessary
  - *Append* the dataframe or replace the attribute to process further
  - Evaluate quality of dataset and current data setup

Data Exploration

	Manufacturer	Model	Size	Color	Range_low	Range_high	Cost	weight_oz	head_width	strength_kn	usable_min	usable_max	current_available
0	Totem	Totem	0.5	Black	11.7	18.9	84	2.4	30.0	6	12.42	16.03	yes
1	Totem	Totem	0.65	Blue	13.8	22.5	84	2.6	30.0	8	14.67	19.18	yes
2	Totem	Totem	0.8	Yellow	17.0	27.7	84	2.9	32.0	9	18.07	23.89	yes
3	Totem	Totem	1	purple	20.9	34.2	84	3.3	34.0	10	22.23	29.88	yes
4	Totem	Totem	1.25	green	25.7	42.3	84	3.8	37.0	13	27.36	37.55	yes





49

MACHINE LEARNING

50

Machine Learning

- One of the most important areas of recent data analysis it the increasing use of ML in the space
- Availability of compute and an easy interface language to utilize it (Python) are the main drivers of this recent increase in use
- While it is a buzzwordy area, there is an approach to get the best lessons out of the area
- Today's focus will be on Classical Machine Learning, which is the most useful type because of model complexity and explainability

Machine Learning Frameworks Overview

- **Scikit-Learn** is the most fleshed out ecosystem, with well thought out APIs, metric and grading tools, and supported algorithms
- **XGBoost** is a favorite of those who use Kaggle, as the boosted trees give relatively good performance out of the box but assume one already knows the data well enough
- **Tensorflow** and similar frameworks are meant for Neural Networks and Deep Learning, which trade model explainability for a costly but accurate model
- Many others in the space, but this is a great overview of the popular ones!

Understanding Scikit-Learn optimizations on Intel® Distribution for Python*

Intel® Distribution for Python*

Numpy* | Scipy* | Scikit-learn* | pyDAAL | Pandas* | Mpi4py* | ...

Intel® Performance Libraries

Intel® IPP | Intel® TBB | Intel® MKL | Intel® DAAL | Intel® MPI Library

Speedup Analytics & Machine Learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)

- Highly tuned functions for classical machine learning and analytics performance across a spectrum of Intel® architecture devices
- Optimizes data ingestion together with algorithmic computation for highest analytics throughput
- Includes Python*, C++, Java* APIs, and connectors to popular data sources including Spark* and Hadoop*

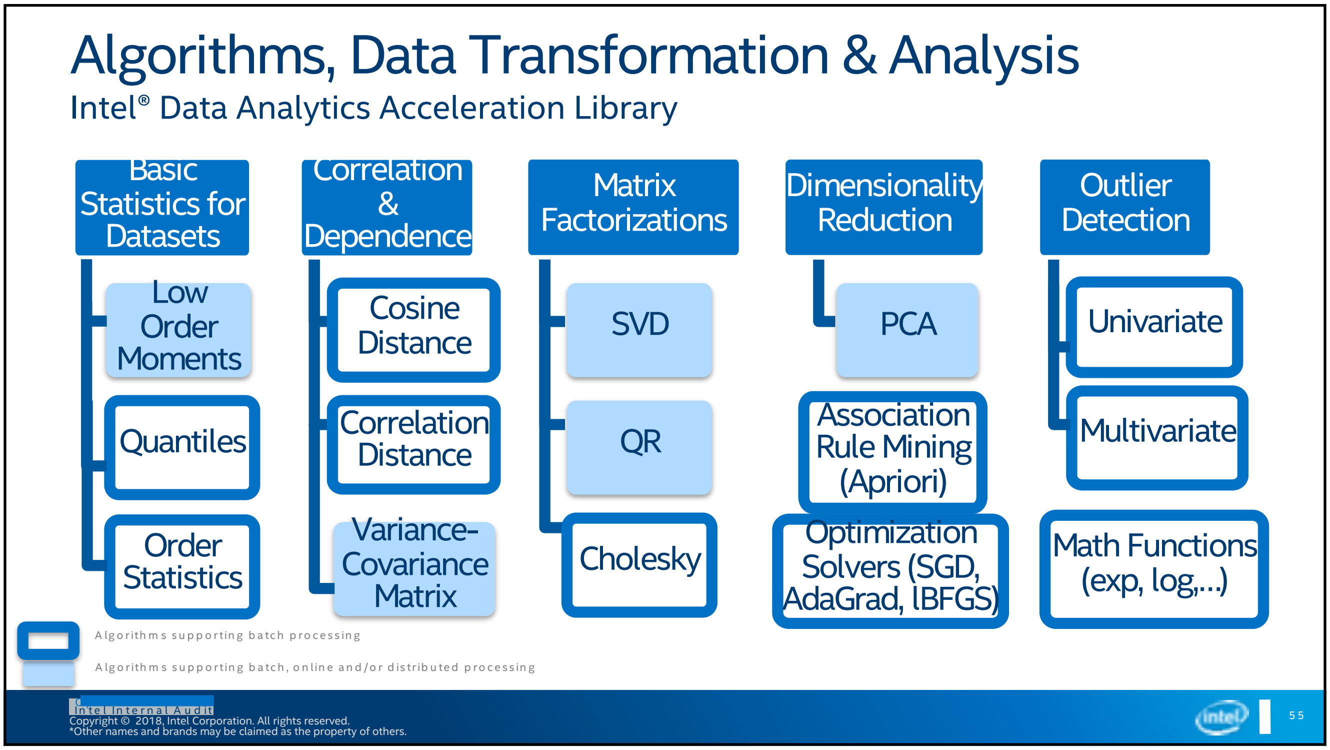
Learn More: software.intel.com/daal

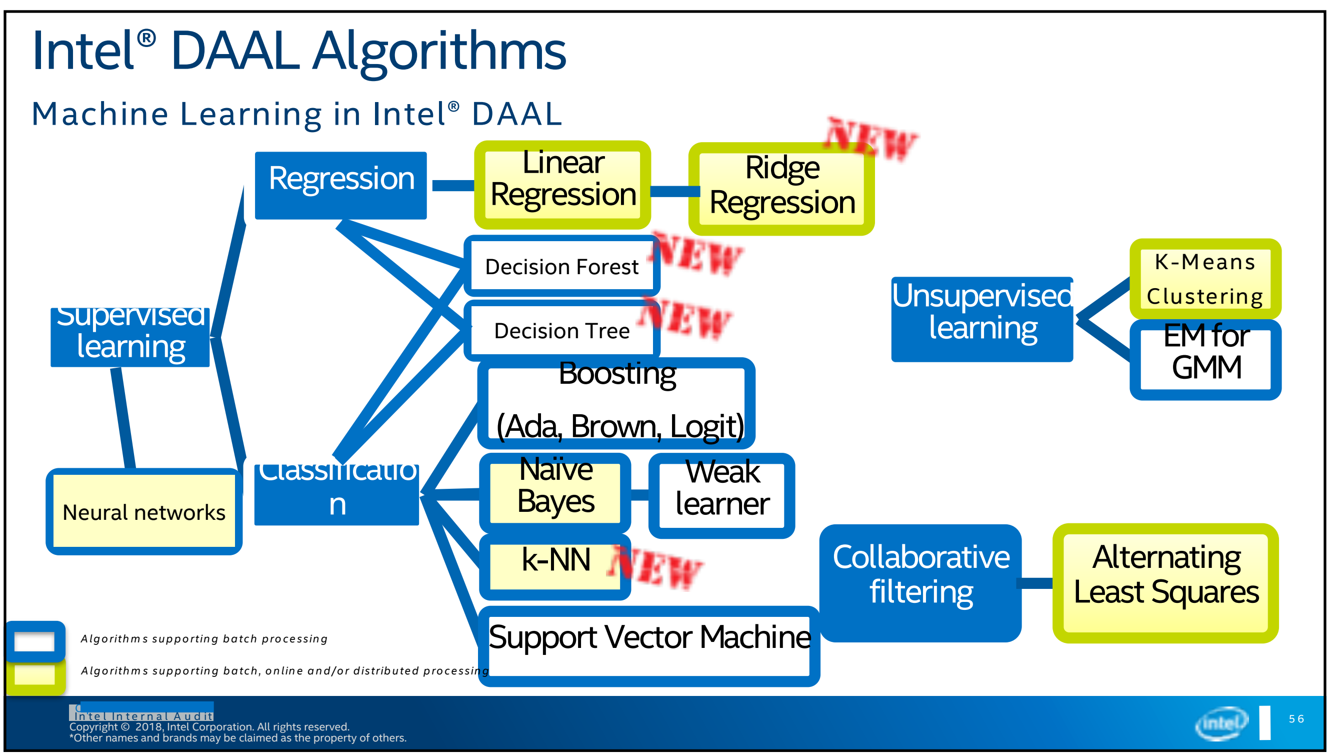
What's New in the 2018 Release

- New Algorithms
  - Classification & Regression Decision Tree and Forest
  - k-NN
  - Ridge Regression
- Spark* MLlib-compatible API wrappers for easy substitution of faster Intel® DAAL functions
- Improved APIs for ease of use
- Repository distribution via YUM, APT-GET, and Conda

Pre-processing → Transformation → Analysis → Modeling → Validation → Decision Making

Pre-processing	Transformation	Analysis	Modeling	Validation	Decision Making
Decompression, Filtering, Normalization	Aggregation, Dimension Reduction	Summary Statistics Clustering, etc.	Machine Learning (Training) Parameter Estimation Simulation	Hypothesis Testing Model Errors	Forecasting Decision Trees, etc.


Algorithms, Data Transformation & Analysis
Intel® Data Analytics Acceleration Library
Basic Statistics for Datasets
Low Order Moments
Quantiles
Order Statistics
Correlation & Dependence
Cosine Distance
Correlation Distance
Variance-Covariance Matrix
Matrix Factorizations
SVD
QR
Cholesky
Dimensionality Reduction
PCA
Association Rule Mining (Apriori)
Optimization Solvers (SGD, AdaGrad, lBFGS)
Outlier Detection
Univariate
Multivariate
Math Functions (exp, log,...)
Algorithms supporting batch processing
Algorithms supporting batch, online and/or distributed processing
Intel Internal Audit
Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
55


Intel® DAAL Algorithms
Machine Learning in Intel® DAAL
Regression
Linear Regression
Ridge Regression
NEW
Supervised learning
Decision Forest
NEW
Decision Tree
NEW
Boosting (Ada, Brown, Logit)
Classification
Naïve Bayes
Weak learner
k-NN
NEW
Support Vector Machine
Neural networks
Unsupervised learning
K-Means Clustering
EM for GMM
Collaborative filtering
Alternating Least Squares
Algorithms supporting batch processing
Algorithms supporting batch, online and/or distributed processing
Intel Internal Audit
Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
56

Utilizing the best the advanced Intel® runtime libraries through Scikit-learn

- If using the Intel® Distribution of Python* variant of Scikit-learn, the optimizations are directly built into Sklearn for you—no code changes required
- This is the best way of utilizing these advanced libraries and runtimes without having to write one's own code to interface with them in C
- The dynamic runtimes detect what hardware you are on and deploy the appropriate instructions for the CPU
- Just as easy as *conda install scikit-learn –c intel*

Options outside of scikit-learn for general ML

- **PyDAAL** – a SWIG-based wrapper around the entire DAAL library, which allows you to use the majority of the DAAL library for general pipelining the online/batch modes of supported ML models
- **Daal4py** – A simplified abstraction of the DAAL library, with some of the distributed "wiring" with MPI done under the hood (*currently Linux only*)
- Other frameworks built on top of NumPy and SciPy can inherit some of the performance benefits of the Intel® Distribution for Python*, which include frameworks such as *Statsmodels and XGBoost*

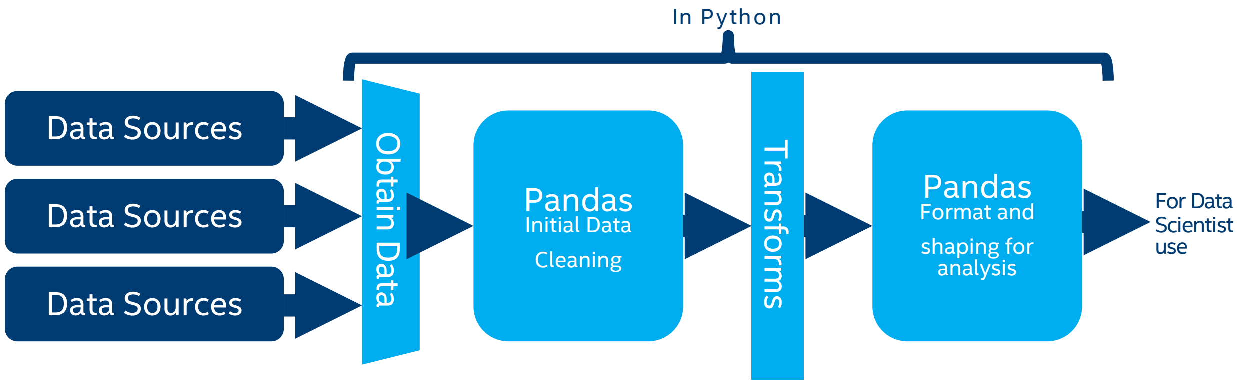
Where ML fits into the equation

- Once preprocessing is out of the way, one is ready to pipe things into ML
- One can iteratively experiment with ML to explore models to find a best performant model variant
- Use the performance, accuracy, or grade of model to determine if more model work is needed
- Take the result of the ML and use it for prediction in some task
- Repeat, re-retrain, re-deploy!

OPTIONS FOR PIPELINING

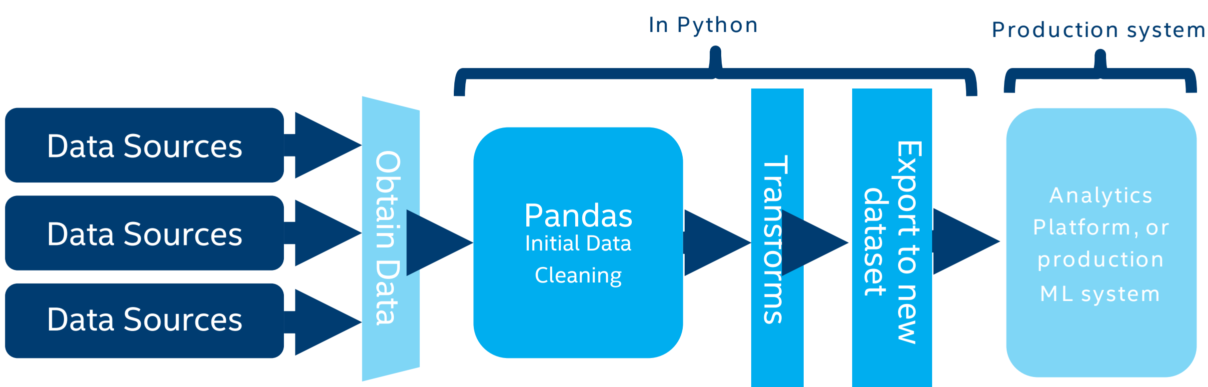
Pipelining into Automation

- General flows assume most of your data wrangling happens in Python, as happens with most Data Scientists when they start on a dataset



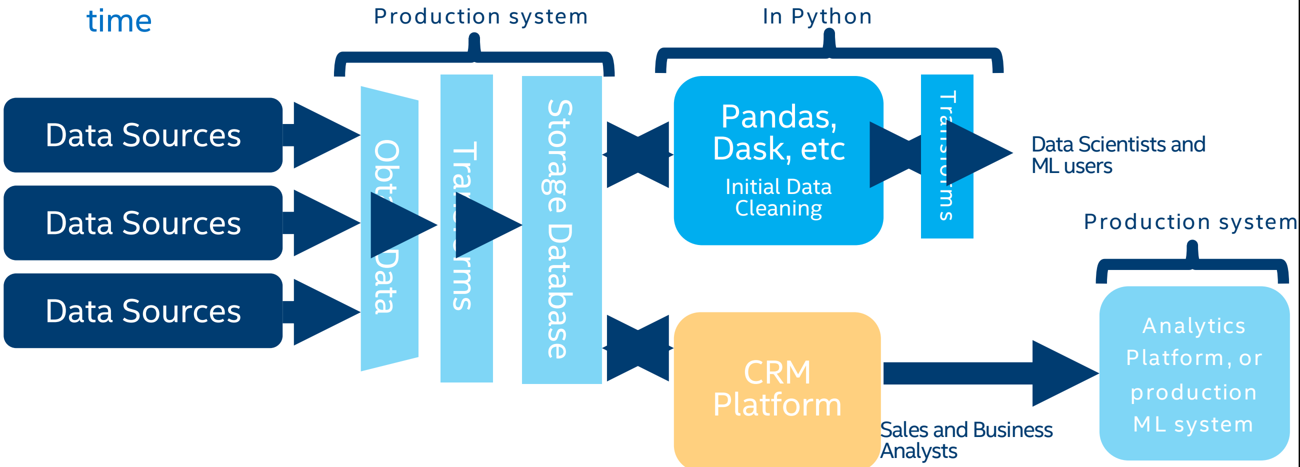
Pipelining into Automation (Con't)

- Flows can change if the use cases start changing, or if production/deployment is necessitated
- Example below for production systems



Pipelining into Automation (Con't)

- Example below is for multiple user dataset(s), with both Data Scientists, Sales, and Business Analysts accessing data at the same time



BREAK

HANDS ON: ADVANCED TOOLS

65

Things you'll need for the exercises

- Linux, Mac, or Windows (some tools not available on Mac or Windows)
  - *Docker container variant is Linux*
- Intel® Distribution for Python*
- Conda or Miniconda
- ~8GB of RAM
- Minimum Core i5 or greater Intel® Processor
- Internet access and Git

66

Advanced tools: Repo

- https://github.com/IntelPython/workshop
- Conda command to create it:
  - *conda create -n idp2018 python=3.6.2 intelpython3_full -c intel*
  - Then *conda install line_profiler*
  - *conda install dask*, *conda install dask distributed*
- *We'll be running a few items from this workshop*
- *NumPy, Numba, Numexpr, Dask examples*

The Black Scholes* Algorithm

A financial options trading formula used for investment price estimates

The formula calculates the price of a *European 'put'* and *'call'* options

Is a partial differential equation (PDE) which describes the *price of the option over time*

Is a great example of some of the optimization problems that exist in real-world

Black-Scholes* (Con't)

Algorithm is a PDE in general form

Solvable for Call and Put options

Goal is to solve for Call and Put options

Putting it into Python is next step

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

$$C(S_t, t) = N(d_1)S_t - N(d_2)Ke^{-r(T-t)}$$

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right]$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

The price of a corresponding put option based on put-call parity is:

$$P(S_t, t) = Ke^{-r(T-t)} - S_t + C(S_t, t)$$
$$= N(-d_2)Ke^{-r(T-t)} - N(-d_1)S_t$$

For both, as above:

- $N(\cdot)$ is the cumulative distribution function of the standard normal distribution
- $T - t$ is the time to maturity (expressed in years)
- S_t is the spot price of the underlying asset
- K is the strike price
- r is the risk free rate (annual rate, expressed in terms of continuous compounding)
- σ is the volatility of returns of the underlying asset

Black-Scholes* (Con't)

Code generates the intermediates of the formula, and gives the corresponding call/put

Generates for as many options that exist (nopt)

Calculates final call/put at the last two lines

```
from math import log, sqrt, exp, erf
import numpy as np
invsqrt = lambda x: 1.0/sqrt(x)

def black_scholes ( nopt, price, strike, t, rate, vol, call, put )
    mr = -rate
    sig_sig_two = vol * vol * 2

    for i in range(nopt):
        P = float( price [i] )
        S = strike [i]
        T = t [i]

        a = log(P / S)
        b = T * mr

        z = T * sig_sig_two
        c = 0.25 * z
        y = invsqrt(z)

        w1 = (a - b + c) * y
        w2 = (a - b - c) * y

        d1 = 0.5 + 0.5 * erf(w1)
        d2 = 0.5 + 0.5 * erf(w2)

        Se = exp(b) * S

        call [i] = P * d1 - Se * d2
        put [i] = call [i] - P + Se
```

One form of optimization: NumPy*-specific math calls

Exercise: In this example, replace the functions from the math library with NumPy* equivalents:

- log
- exp
- erf
- invsqrt

Re-run the profiling to see what you can find

- Total time?
- A change in what the bottlenecks were?

Black Sholes*: NumPy* Variant (vectorized)

- Loop removal helps by allowing use of NumPy's native array capabilities
- Individually going through loops, even with NumPy* arrays is VERY expensive
- Loop-parallel has a few options, and this is one of them: vectorization!
- On line_profiler, how many times did the code hits changes in this new version?

Black Scholes*: NUMEXPR*

- By interacting directly with numexpr*, you are calling out to the vectorization capabilities without going through the NumPy* layer
- By compressing the entire vectorization command of one's calculation in one expression, the vectorization engine can do significantly more
- This is one of the ways we did some of our optimization work on NumPy* itself for the Intel® Distribution for Python*!

Black Scholes*: NUMBA*

- **Exercise**: Using the Numba example, test with same methods: timeit, cProfile, line_profiler
- What do you notice about the functions being imported?
- Why do you think it uses the "nopython=True" option?
- What works? What doesn't work?

Black Scholes*: NUMBA*

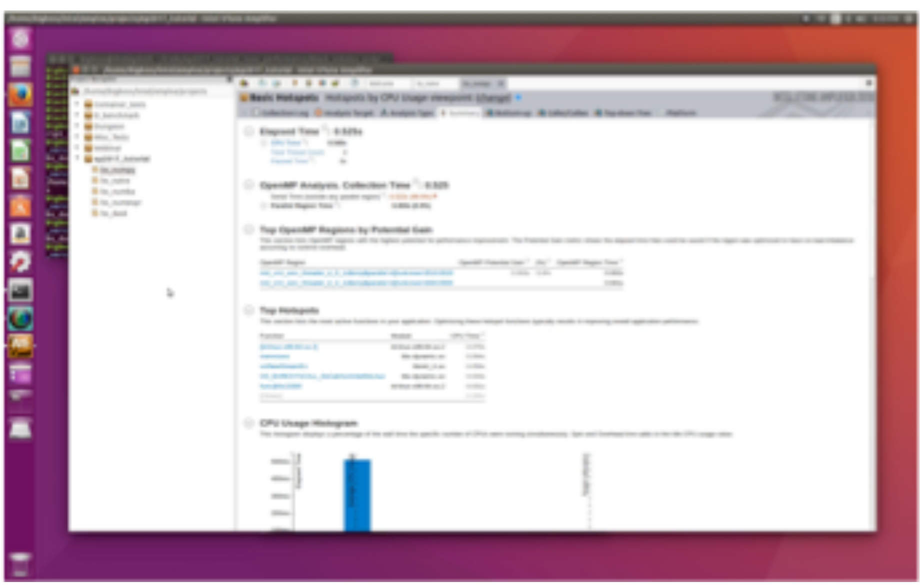
- This example uses Just-In-Time(JIT) compiling to achieve performance gains
- Because of this, profiling can become VERY difficult
- The first run is slow because you pay for the compilation time, but the function is cached afterwards
- Many times this require writing in pure Python before utilizing Numba

Black Scholes*: DASK* (NumPy* mods)

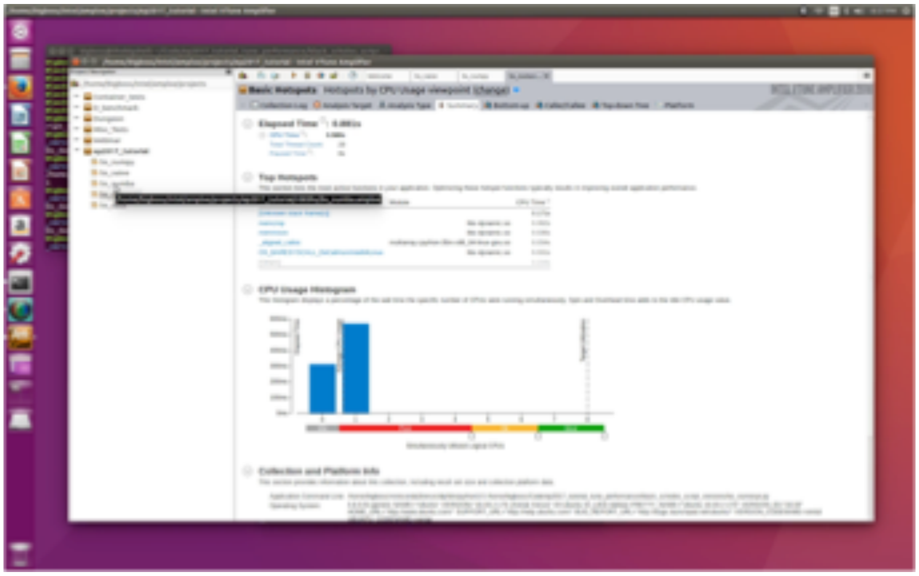
- What is different in this example? What does it change?
- Using this example, test with same methods: timeit, cProfile, line_profiler
- How does the diagnostic server help?
- What works? What doesn't work?

Vtune Analysis of Black Scholes* with NUMPY*

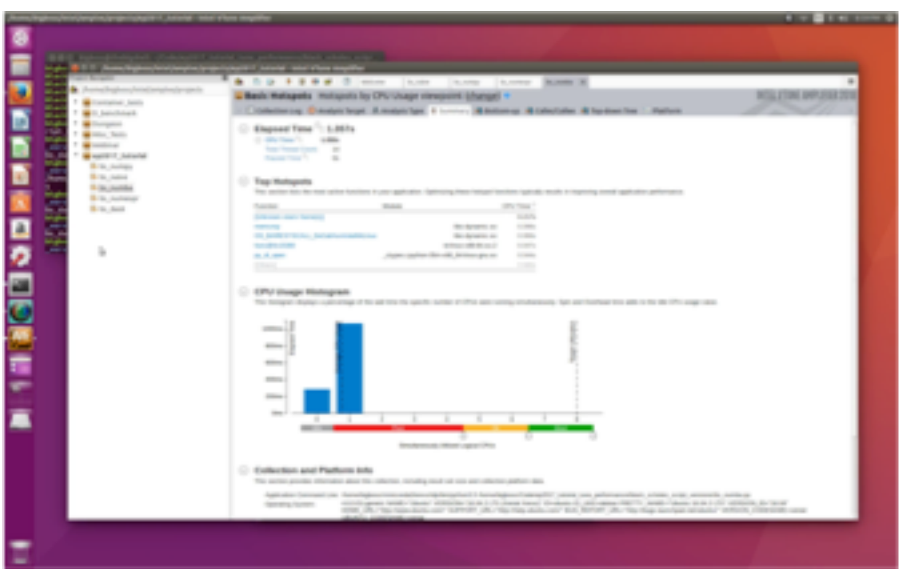
Vtune Analysis of Black Scholes* with NUMEXPR



Vtune Analysis of Black Scholes* with NUMBA*



Vtune Analysis of Black Scholes* with DASK



Why Review these? I thought it was about ML?

- The reason for these items first is to lay the foundation of how the Data Scientist workflow is correctly dealt with: with 90% as preprocessing, it is VERY important to know how to use these tools
  - Next we will look at some other tools in the ML space, and play with datasets as well
- A lot of what goes on in daily work means the fastest possible iterations when sifting through data, which the tools here can help with—and IDP makes it even faster
  - Optimizations throughout the ecosystem used by Data Scientists is one of the main tenants of the Intel® Distribution for Python*!

81

HANDS ON: CHAINING IT TOGETHER

82

Chaining all the skills together

- For the applied portion of this tutorial, we are going to take a look at one of my very old Github projects: pyworkout-toolkit
  - Go here and download the repo
    - https://github.com/triskadecaepyon/pyworkout-toolkit
  - *pip install pyworkout-toolkit* or *conda install -c triskadecaepyon pyworkout=0.0.1*
    - I might *eventually* get it on conda-forge ☺
  - *Conda install bokeh*
  - *Pip install or conda install graphviz (you might need the actual binary too)*
  - *http://graphviz.org/download/*

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Additional Information

Intel® Distribution for Python* Documentation

- https://software.intel.com/en-us/intel-distribution-for-python-support/documentation

cProfile:

- https://docs.python.org/3.5/library/profile.html

Line_profiler:

- https://github.com/rkern/line_profiler