Tuning Random Forest Hyperparameters across Big Data Systems

By Ishna Kaul

Introduction

Motivation

The amount of data organizations are generating has skyrocketed. Businesses are eager to use all of this data to gain insights and improve processes; however, “big data” means big challenges. Entirely new technologies had to be invented to handle larger and larger datasets. These new technologies include the offerings of cloud computing service providers like Amazon Web Services (AWS) and open-source large-scale data processing engines like Apache Spark. As the amount of data generated continues to soar, aspiring data scientists who can use these “big data” tools will stand out from their peers in the market [1].As more and more data is being generated every second, data scientists need to understand how to best choose the best available tool for better performance. Through this project, we are trying to compare how Python works on local as well as how PySpark works on both local, and AWS.

Whether a regression or a classification task, random forest is an applicable model for your needs. It can handle binary features, categorical features, and numerical features. There is very little pre-processing that needs to be done. The data does not need to be rescaled or transformed and they are parallelizable and great with high dimensionality with decent training speed. Hence, for this project, we will be looking at Random Forest’s performance on various big data systems. This project is two-fold:

- In the first part it discuss Python vs Pyspark performance for Random Forest through various hyperparameters on local with a relatively decent sized data (about 100 MB csv file)
- Additionally, the first part also discusses how performance for data preparation tasks changes for different size of datasets on local Python and PySpark (100 MB vs 2.5GB)
- In the second part, we tune these parameters and understand the performance of pyspark on EMR cluster with different nodes and partitions using a much larger dataset (~6GB).

Evaluated System(s)

We evaluate the following systems here:

- Local: 2017 Mac which is 4 cores and 256GB SSD and 8GB of onboard memory
    - Python vs PySpark on Local
- AWS: 2 EMR cluster settings: 2 & 4 worker nodes having m5.xlarge 4 vCore, 16 GiB with Spark 2.4.4
    - PySpark with 4 nodes vs 2 nodes

Python on Local

The "local" here is my 2017 Mac which is 2.3GHz dual-core Intel Core i5, Turbo Boost up to 3.6GHz, with 64MB of eDRAM and 256 GB SSD and 8GB of 2133MHz LPDDR3 onboard memory.

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document [3].

In Python, the memory manager is responsible for these kinds of tasks by periodically running to clean up, allocate, and manage the memory. Unlike C, Java, and other programming languages, Python manages objects by using reference counting. This means that the memory manager keeps track of the number of references to each object in the program. When an object's reference count drops to zero, which means the object is no longer being used, the garbage collector (part of the memory manager) automatically frees the memory from that particular object.

PySpark on Local

Apache Spark has become the de facto unified analytics engine for big data processing in a distributed environment. Yet we are seeing more users choosing to run Spark on a single machine, often their laptops, to process small to large data sets, than electing a large Spark cluster. This choice is primarily because of the following reasons:

- A single, unified API that scales from “small data” on a laptop to “‘big data” on a cluster
- Polyglot programming mode, with support for Python, R, Scala, and Java
  ANSI SQL support
- Tight integration with PyData tools, e.g., Pandas through Pandas user-defined functions

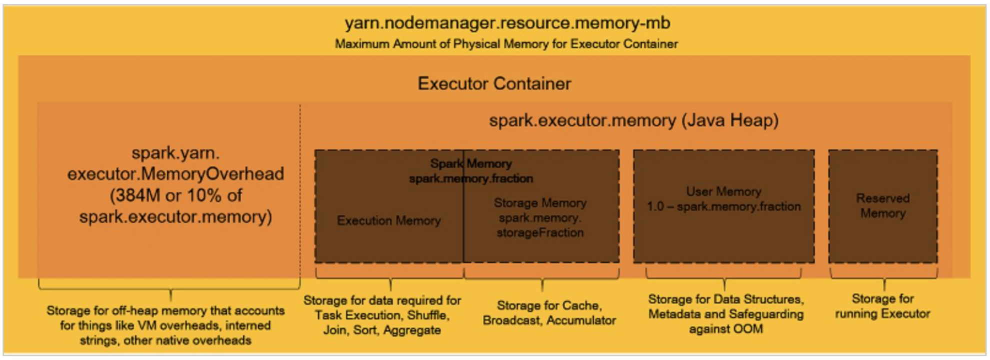
While the above might be obvious, users are often surprised to discover that:

- Spark installation on a single node requires no configuration (just download and run it).
- Spark can often be faster, due to parallelism, than single-node PyData tools.
- Spark can have lower memory consumption and can process more data than laptop ’s memory size, as it does not require loading the entire data set into memory before processing [4].

For Spark, it is easy to scale from small data set on a laptop to “big data” on a cluster with one single API. Even on a single node, **Spark’s operators spill data to disk if it does not fit in memory, allowing it to run well on any sized data.** Over few Spark releases, Pandas has contributed and integrated well with Spark. One huge win has been Pandas UDFs. In fact, because of Pandas API similarity with Spark DataFrames, many developers often combine both, as it’s convenient to interoperate between them. It’s been a few years since Intel was able to push CPU clock rate higher. Rather than making a single core more powerful with higher frequency, the latest chips are scaling in terms of core count. Hence, it is not uncommon for laptops or workstations to have 16 cores, and servers to have 64 or even 128 cores. In this manner, these multi-core single-node machines’ work resemble a distributed system more than a traditional single core machine.

PySpark on AWS

In the world of big data, a common use case is performing extract, transform (ET) and data analytics on huge amounts of data from a variety of data sources.One of the most popular cloud-based solutions to process such vast amounts of data is Amazon EMR. Amazon EMR is a managed cluster platform that simplifies running big data frameworks, such as Apache Hadoop and Apache Spark, on AWS. Amazon EMR enables organizations to spin up a cluster with multiple instances in a matter of few minutes. Apache Spark is a cluster-computing software framework that is open-source, fast, and general-purpose. It is widely used in distributed processing of big data. Apache Spark relies heavily on cluster memory (RAM) as it performs parallel computing in memory across nodes to reduce the I/O and execution times of tasks.


yarn.nodemanager.resource.memory-mb
Maximum Amount of Physical Memory for Executor Container
Executor Container
spark.executor.memory (Java Heap)
spark.yarn. executor.MemoryOverhead (384M or 10% of spark.executor.memory)
Spark Memory spark.memory.fraction
Execution Memory
Storage Memory spark.memory. storageFraction
User Memory 1.0 – spark.memory.fraction
Reserved Memory
Storage for off-heap memory that accounts for things like VM overheads, interned strings, other native overheads
Storage for data required for Task Execution, Shuffle, Join, Sort, Aggregate
Storage for Cache, Broadcast, Accumulator
Storage for Data Structures, Metadata and Safeguarding against OOM
Storage for running Executor

The executor container has multiple memory compartments. Of these, only one (execution memory) is actually used for executing the tasks. These compartments should be properly configured for running the tasks efficiently and without failure [5].

Problem Statement

Through the analysis, we try and find answers to the following questions:

Part 1: ON LOCAL MACHINE PySpark vs Python

2017 Mac which is 4 cores and 256GB SSD and 8GB of onboard memory.

- Given the 100 MB dataset, how does PySpark vs Python perform for basic data preparation operations such as loading the dataset, group by aggregations, joins, unions, sum, max, count etc?

- Given the 100 MB dataset, how does the performance of PySpark on local and Python on local vary as we tune different hyperparameters for a Random forest such as number of trees, depth of the tree, minimum instances for node split, feature sub-strategy and impurity?

- Given the 100 MB dataset, how does the performance of PySpark on local and Python on local vary with cross-validation?

- How does the performance of PySpark and Python on local differ as we change the dataset sizes from 100 MB to 2.5 GB?
  It is often believed that PySpark performance worse than Python on smaller datasets. But will this change if we change the size of our dataset and increase it?

- How does the python's performance change when we change the "n_jobs" in scikit learn?
  N_jobs set to -1 uses all available cores. If `n_jobs` was set to a value higher than one, the data is copied for each point in the grid (and not `n_jobs` times). This is done for efficiency reasons if individual jobs take very little time, but may raise errors if the dataset is large and not enough memory is available. A workaround in this case is to set `pre_dispatch`. Then, the memory is copied only `pre_dispatch` many times. A reasonable value for `pre_dispatch` is `2 * n_jobs`.

Part 2: AWS multiple worker node instances

2 EMR cluster settings: 2 & 4 worker nodes having m5.xlarge 4 vCore, 16 GiB with Spark 2.4.4

- Given the 7 GB dataset, how does PySpark vs Python perform for basic data preparation operations such as loading the dataset, group by aggregations, joins, unions, sum, max, counts etc?
- Given the 7 GB dataset, how does the performance of 2 worker nodes vs 4 workers nodes on local vary with cross-validation?
- How does tuning various Random Forest hyperparameters such as number of trees, depth of the tree, minimum instances for node split, feature sub-strategy and impurity change performance on 2 and 4 worker node settings in AWS on a 7 GB dataset?

Methodology

Datasets

Forest Cover Type, UCI ML Library [6]

We use this dataset on our local machine.

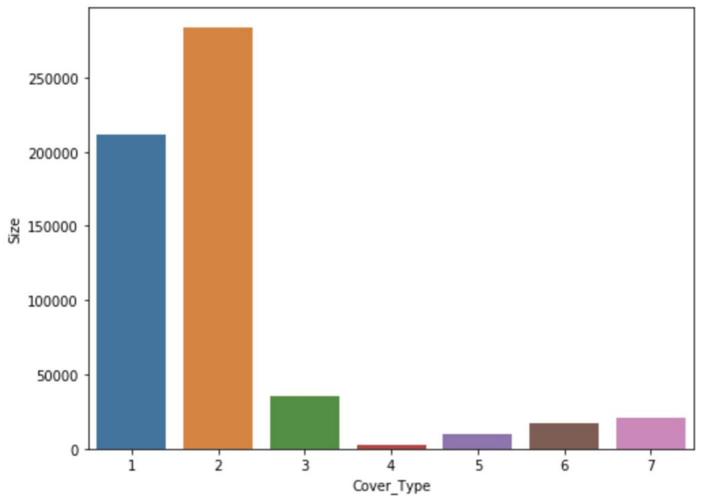
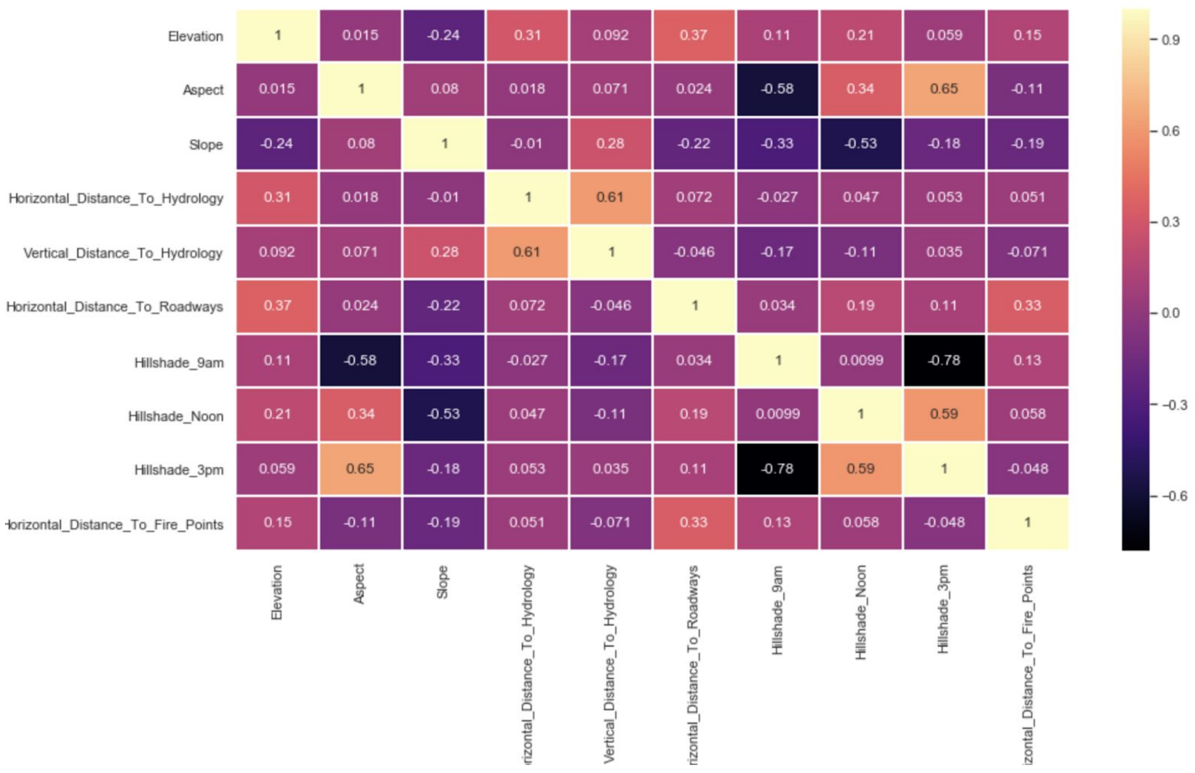


Figure showing the difference in the Cover types in the data

This dataset contains tree observations from four areas of the Roosevelt National Forest in Colorado. All observations are cartographic variables (no remote sensing) from 30 meter x 30 meter sections of forest. There are over half a million measurements total. This dataset includes information on tree type, shadow coverage, distance to nearby landmarks (roads etcetera), soil type, and local topography. This dataset is part of the UCI Machine Learning Repository. The original database owners are Jock A. Blackard, Dr. Denis J. Dean, and Dr. Charles W. Anderson of the Remote Sensing and GIS Program at Colorado State University. It is approximately 80 MB in size.

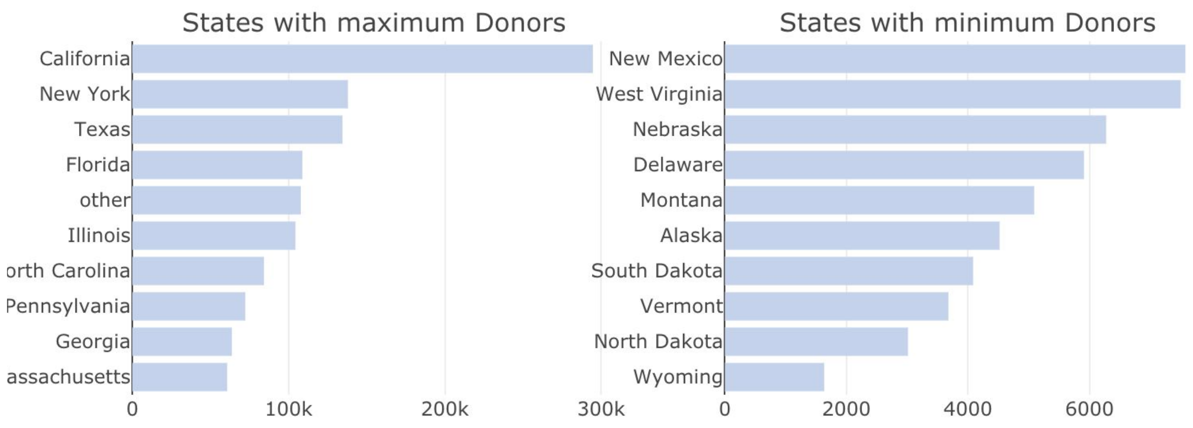


	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hillshade_Noon	Hillshade_3pm	Horizontal_Distance_To_Fire_Points
Elevation	1	0.015	-0.24	0.31	0.092	0.37	0.11	0.21	0.059	0.15
Aspect	0.015	1	0.08	0.018	0.071	0.024	-0.58	0.34	0.65	-0.11
Slope	-0.24	0.08	1	-0.01	0.28	-0.22	-0.33	-0.53	-0.18	-0.19
Horizontal_Distance_To_Hydrology	0.31	0.018	-0.01	1	0.61	0.072	-0.027	0.047	0.053	0.051
Vertical_Distance_To_Hydrology	0.092	0.071	0.28	0.61	1	-0.046	-0.17	-0.11	0.035	-0.071
Horizontal_Distance_To_Roadways	0.37	0.024	-0.22	0.072	-0.046	1	0.034	0.19	0.11	0.33
Hillshade_9am	0.11	-0.58	-0.33	-0.027	-0.17	0.034	1	0.0099	-0.78	0.13
Hillshade_Noon	0.21	0.34	-0.53	0.047	-0.11	0.19	0.0099	1	0.59	0.058
Hillshade_3pm	0.059	0.65	-0.18	0.053	0.035	0.11	-0.78	0.59	1	-0.048
Horizontal_Distance_To_Fire_Points	0.15	-0.11	-0.19	0.051	-0.071	0.33	0.13	0.058	-0.048	1

EDA: Correlation matrix for continuous variables in the dataset

Donorschoose.org, hosted on Kaggle [7]

We use this data on AWS and Azure. We also use a part of it (Projects dataset) on our local.



Example of results from a query where we find out states with maximum and minimum donor

Founded in 2000 by a Bronx history teacher, DonorsChoose.org has raised $685 million for America's classrooms. To date, 3 million people and partners have funded 1.1 million DonorsChoose.org projects. The dataset has 6 csv files:

- Donations: For every project in the Projects.csv dataset, there are one or more donations. This dataset contains each donation and is joined with the dataset above using the "Project ID" .
  The size of the dataset is ~600 MB.
- Donors: For every project in the Projects.csv dataset, there are one or more donations. This dataset contains each donation from a citizen donor, information on that donor and is joined with the dataset above using the "Project ID".
  The size of the dataset is ~120 MB
- Projects: For every project in the Projects.csv dataset, there are one or more donations. This dataset contains each donation from a citizen donor and is joined with the dataset above using the "Project ID".
  The size of the dataset is ~2.5 GB
- Resources: For every project in the Projects.csv file, there are one or more resources that is requested. This dataset contains the names of each resource in the project request and is joined with the dataset above using the "Project ID" column.
  The size of the dataset is ~800 MB
- Schools: More information at a school level. Each row represents a single school. This dataset is joined with the project data using the "School ID" column.
  The size of the dataset is ~10 MB
- Teachers: More information at a teacher level. Each row represents a single teacher. This dataset is joined with the project data using the "Teacher ID" column.
  The size of the dataset is ~20MB

LOCAL : Python vs Pyspark

Ingesting data

We will observe differences in time taken to load the datasets of various sizes (100 MB and 3 GB) both in Python and in Spark and then compare these.

Data Preprocessing

We will observe differences in time taken to conduct various data preparation and EDA tasks such as joins, aggregations, etc., both in Python and in Spark and then compare these.

Random Forest Tuning

We will compare different parameter based random forests built on both pyspark and python and compare how time to build these varies as we change the hyperparameters. There are a lot of hyperparameters we can tune for random forests. Some of them that we will be playing around with are:

- Trees = number of trees in the forest
- Feature subset strategy = max number of features considered for splitting a node
- Maximum depth = max number of levels in each decision tree

- Minimum samples in leaf = min number of data points allowed in a leaf node for a split to happen
- Impurity = metric to select between gini and entropy to find the split node

Cross-Validation

Computing a 3 fold cross-validation on both scikit-learn and spark's ml.classification

Varying n_jobs on local Python

We will set n_jobs to both None and -1 to see the difference in scikit-learn's performance in local Python.

AWS: 2 worker nodes vs 4 worker nodes

Ingesting data

We will observe differences in time taken to load the datasets in both a 2 node and 4 worker node EMR cluster setting.

Data Preparation

We will observe differences in time taken to conduct various data preparation and EDA tasks such as joins, aggregations, etc., both in 2 and 4 worker nodes and then compare these.

Random Forest Tuning

- We will compare different parameter based random forests built on both 2 and 4 node setting and compare how time to build these varies as we change the hyperparameters
- Computing a 3 fold cross-validation on both spark's ml.classification

Results

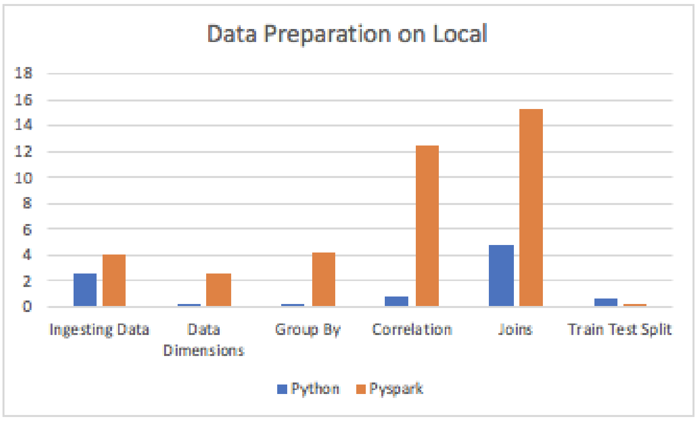
Note: All time units mentioned in the graphs are in seconds. **We have not focused on improving the performance of the model in terms of accuracy or other evaluation metrics. We have simply observed performance under various settings.**

Additionally, by "default" setting, the hyperparameter setting for the cases below is as follows:

- Number of trees = 100
- Maximum depth = 10
- Feature subset strategy = square root of total number of features
- Minimum samples in leaf = 2
- Impurity = gini

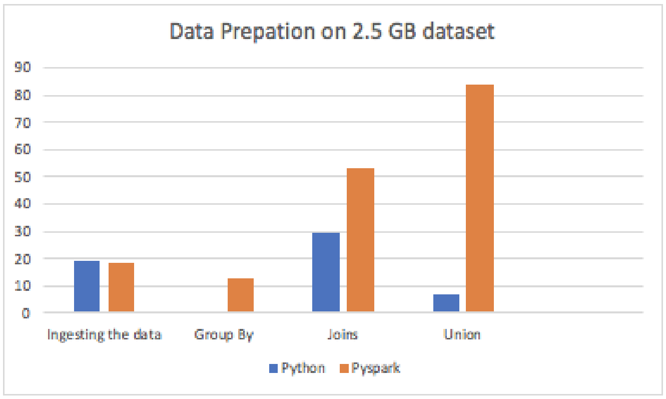
Local: Python vs PySpark

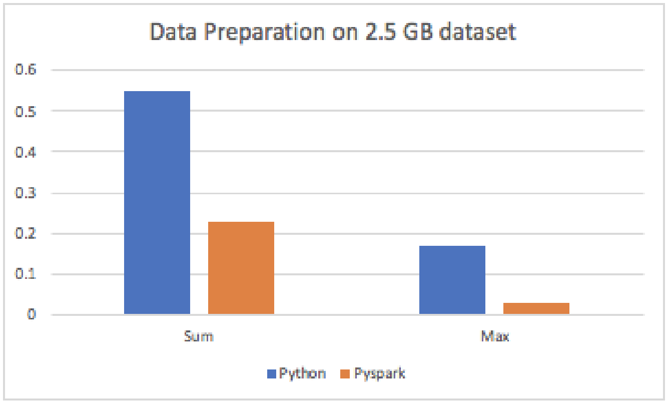
I. Data Preparation: Comparing PySpark and Python data preparation and EDA on 100 MB dataset



Observation: Python will definitely perform better compared to pyspark on smaller data sets.

II. Data Preparation and EDA tasks on PySpark and Python for a ~2.5GB dataset which is the "Projects" data from the donors.org data mentioned above



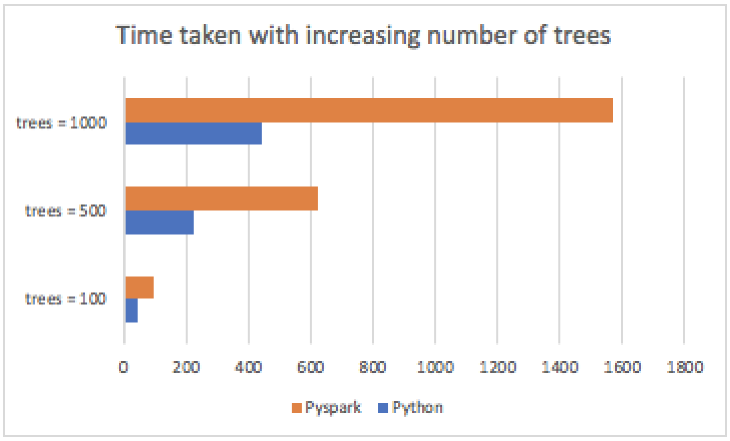


Observation: You will see the difference when you are dealing with larger data sets. For smaller datasets, Python on local performs significantly better than PySpark. However, as we increase the size of our dataset, we can study the threshold around which PySpark's performance starts improving and for some operations, even getting better than Python on local. For example, when we are dealing with the dataset of

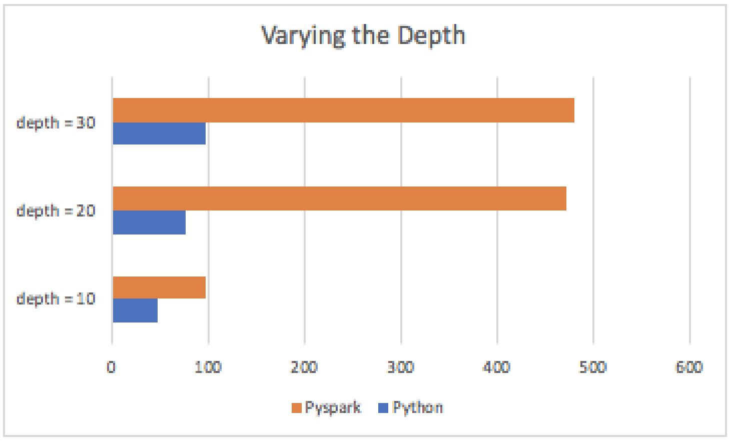
~2.5 GB, operations such as sum of a column (associative and communicative), take a lesser time on PySpark than on Python

III. Applying Random Forest algorithm on the ~100 MB Forest Cover dataset and tuning various hyperparameters

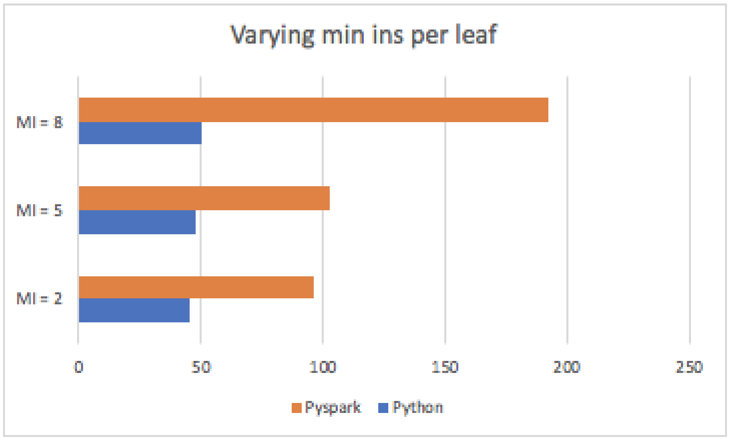
- Varying number of trees while other values are set at default:



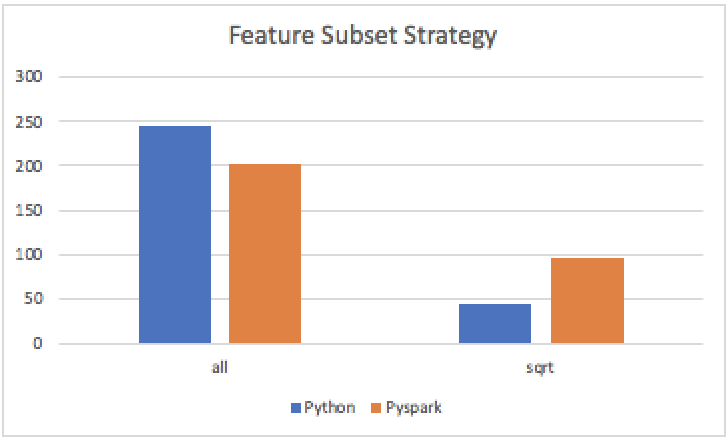
- Varying the depth at number of trees at default setting:



- Varying number of minimum instances at default setting:

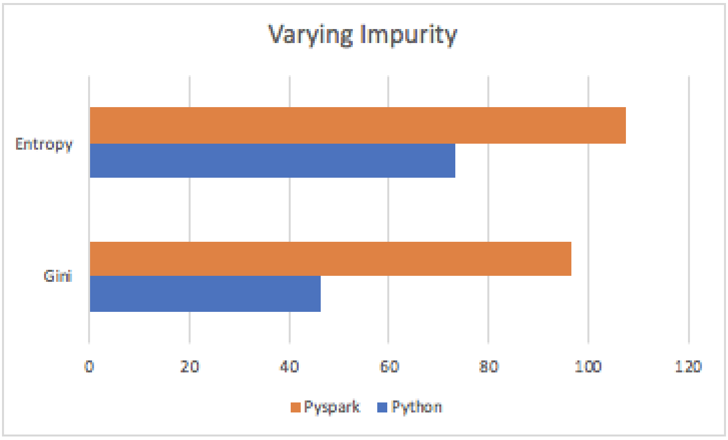


- Feature Subset Strategy at default:



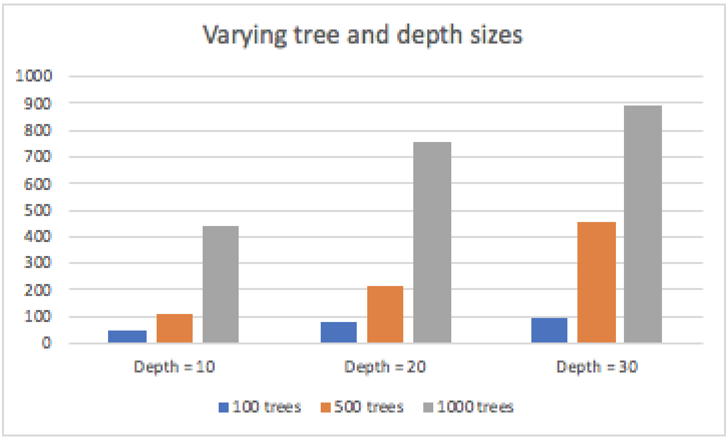
Observation: PySpark's performance was better when the sub-strategy was set to "all" features.

- Varying Impurity at default setting:



Observation: Calculating on entropy took significantly more time than gini calculations in both PySpark and Python.

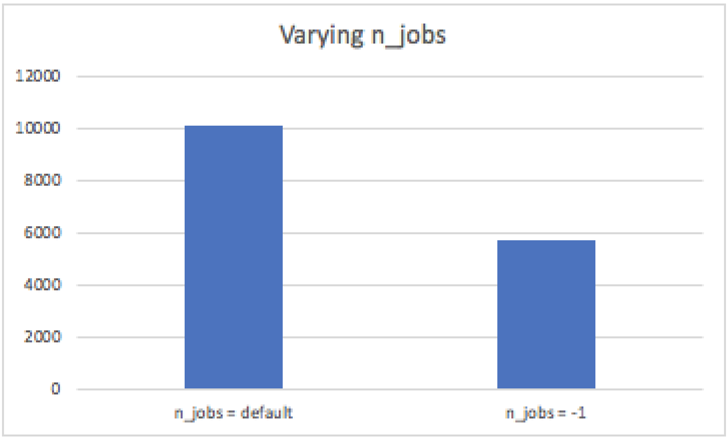
- On Python only, increasing number of trees gradually with the depth of the trees:



Observation: Increasing the number of trees, followed by depth had the highest impact on time consumed to build a random forest.

- Varying the n_jobs parameter on a 3 fold cross validation with 90 models to fit on the 100 MB data:
  If `n_jobs` was set to a value higher than one, the data is copied for each point in the grid (and not `n_jobs` times). This is done for efficiency reasons if individual jobs take very little time, but may raise errors if the dataset is large and not enough memory is available. A workaround in this case is to set `pre_dispatch`. Then, the memory is copied only `pre_dispatch` many times. A reasonable value for `pre_dispatch` is `2 * n_jobs`.

  Observation: Python's performance improved tremendously when we put n_jobs to -1

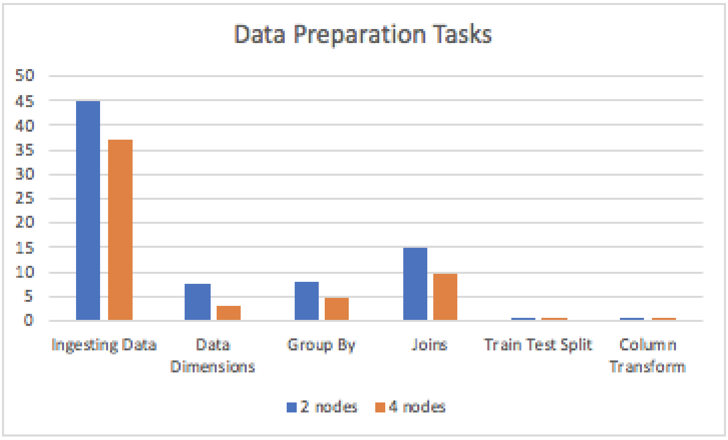


  `n_jobs=-1` it uses 100% of the cpu of *one of the cores*. Each process is run in a different core. Each process takes the 100% usage of a given core, but if you have `n_jobs=1` only one core is used.

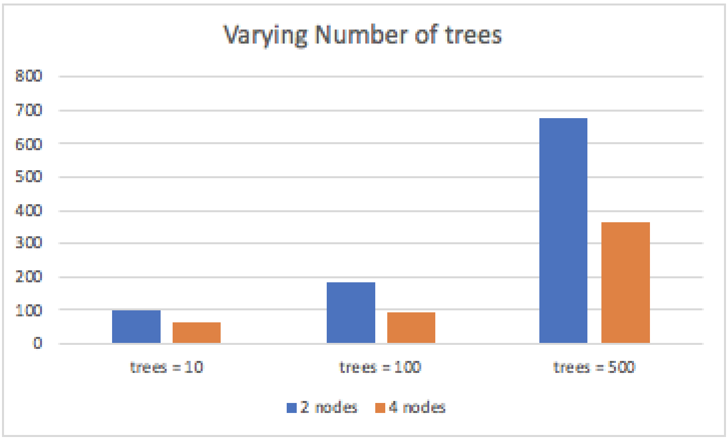
AWS: 2-node vs 4-node

AWS comparison on a 2-node vs 4-node EMR cluster setting on a 6.4 GB donor.org dataset

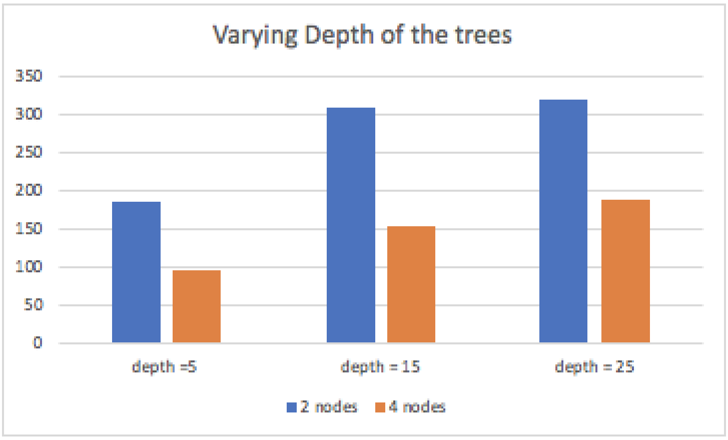
I. Data Preparation tasks:



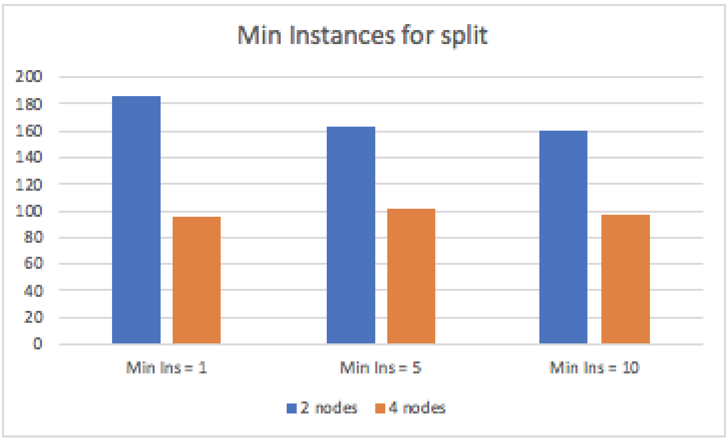
II. Applying Random Forest algorithm on the ~6.5 GB donor.org dataset and tuning various hyperparameters:
   - Varying the number of trees, keeping everything at default



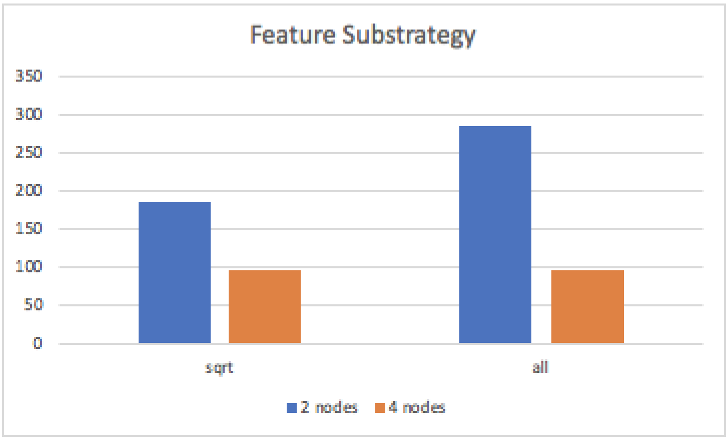
- Varying the depth of the trees, keeping everything at default



- Minimum instances for node split

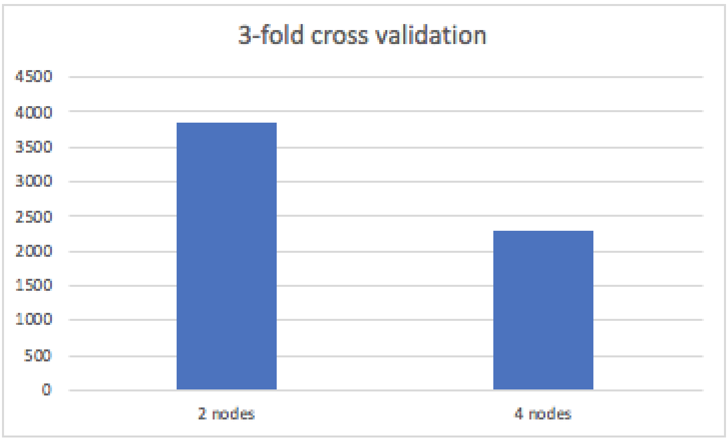


- Varying Feature Selection Sub-strategy:



Observation: Time taken by a 4 node system for a feature sub-strategy set to "sqrt" was the same as for "all", however for a 2 node system, the difference was significant.

- 3-fold cross validation for 18 models:



Observation: 2 node system took almost twice as much time as a 4 node system for a 3 fold cross-validation with 18 fits

Conclusion

Through our analysis we found a lot of expected as well as unexpected. Here is the summary of them as follows:

- **Python will definitely perform better compared to pyspark on smaller data sets.** You will see the difference when you are dealing with larger data sets. For smaller datasets, Python on local performs significantly better than PySpark. However, as we **increase the size of our dataset, we can study the threshold around which PySpark's performance starts improving and for some operations, even getting better than Python on local**. For example, when we are dealing with the dataset of ~2.5 GB, operations such as sum of a column (associative and communicative), take a lesser time on PySpark than on Python

- **Python's performance improved tremendously when we put n_jobs to -1.**
- Increasing the number of trees, followed by depth had the highest impact on time consumed to build a random forest.
- Additionally, **calculating on entropy took significantly more time than gini calculations.**
- Surprisingly, **PySpark performed comparably well as Python when we set the feature subset strategy to "all".** This could mean the parallelisation, even for a smaller dataset was helping there.
- A 4 node system was significantly faster than a 2 node EMR cluster on AWS
- Time taken by a **4 node system for a feature sub-strategy set to sqrt was the same as for "all", however for a 2 node system, the difference was significant.**
- Additionally, a 2 node system took almost twice as much time as a 4 node system for a 3 fold cross-validation with 18 fits.

Given everything, There are at least two advantages of Pandas wherever possible that PySpark could not overcome:

- stronger APIs
- more libraries, for example, matplotlib for data visualization

In a nutshell, **I would prefer Python for datasets ~2 GB in size for ease of use and flexibility, beyond that, pyspark catches up and parallelisation works beautifully.** Random Forest are a great example to test your system on to understand how parallelisation is working in the background and I would definitely recommend testing settings on this if someone is curious to find out how various big data system settings work.

References

[1] https://towardsdatascience.com/getting-started-with-pyspark-on-amazon-emr-c85154b6b921
[2] https://docs.python.org/2/using/mac.html
[3] https://docs.python.org/2/c-api/memory.html
[4] https://databricks.com/blog/2018/05/03/benchmarking-apache-spark-on-a-single-node-machine.html
[5] https://aws.amazon.com/blogs/big-data/best-practices-for-successfully-managing-memory-for-apache-spark-applications-on-amazon-emr/
[6] https://archive.ics.uci.edu/ml/datasets/covertype
[7] https://www.kaggle.com/donorschoose/io/data#_=_