Benchmarking Performance of Hyperparameter Tuning for Tree-based Algorithms

Sanjana Gupta
University of Washington

DATA 516

Fall 2020

Introduction

To improve the performance of a machine learning algorithm, strategies like gathering more data and performing more feature engineering can be done.
Hyperparameter Tuning is the process of choosing a set of optimal hyperparameters to solve a machine learning problem. A hyperparameter is a parameter whose value is used to control the learning process, and it cannot be learnt from the data. It is a “meta” optimization task where the outcome of tuning is the best hyperparameter setting and the outcome of the model is the best model parameter setting. These are set before training a model and it is typically a time consuming and computationally expensive process.

Ensemble models like Random Forest and Gradient Boosting are good at handling data with large dimensionality (features) and can handle complex, non-linear relationships. They are powerful and can be trained relatively quickly. Tuning hyperparameters can further improve accuracy of these models and can result in greater predictions. However, tuning hyperparameters for large datasets is generally not possible on a local machine since it requires substantial computational resources.

The goal of this project is to tune hyperparameters for Ensemble based Learning methods and to compare the performance of Python’s scikit-learn library with PySpark’s ML Library.

Evaluated Systems

1. Local machine

The local machine used is a 2020 MacBook Pro with 16GB RAM, Quad-core and 512GB SSD.

2. Databricks

A Standard Edition Databricks account was used to test the performance of Apache Spark’s PySpark library. Performance was captured by varying the number of worker nodes.

Python on Local Machine:

Unlike other programming languages like C/C++, Python performs memory management and users do not have any control over it. Memory management in Python involves the use of a private heap that keeps a collection of all Python objects and data structures. The python memory manager internally ensures the management of this private heap. Managing memory leaks in Python is tough and requires additional tools like guppy or Objgraph.
It is also harder to handle large datasets/Big Data (larger than the local machine’s RAM) on local machines.

Parallelism can be achieved on local machines through multiprocessing by specifying the number of concurrent processes that can be run. But, this can lead to memory problems if there aren't enough resources available.

PySpark on Local Machine:

PySpark is the Python API for Apache Spark; a big data system. Apache Spark is a distributed cluster computing framework for big data. PySpark provides a cloud based platform and can integrate and work efficiently with Resilient Distributed Datasets (RDD) in Python. It provides impressive disk persistence and powerful caching. It also processes faster when compared to traditional Big Data frameworks (up to 100x faster than traditional Hadoop MapReduce).

PySpark on Databricks:

Databricks provides a clean notebook interface (similar to Jupyter) which is preconfigured to hook into a Spark cluster. It allows collaborative working as well as working in multiple languages like Python, Spark, R and SQL. Working on Databricks offers the advantages of cloud computing - scalable, lower cost, on demand data processing and data storage. It also provides a cluster manager and a storage manager. Clusters can be easily managed in Databricks with options to display, filter, pin clusters, as well as view cluster configurations. Clusters can also be easily edited and cloned, and access to clusters can also be easily managed. There are also 2 options for the termination of a cluster - manual or automatic termination. To configure clusters, a cluster policy, mode, pool, runtime, python version, as well as worker nodes, cluster size, and auto scaling can be specified.

Problem Statement

Through analysis, the following questions are answered:

Part 1: On Local Machine

- How does Python vs PySpark perform for basic data loading using data stored locally?
- How does Python vs PySpark perform for data imputation tasks?
- How does the performance of Python vary as we tune hyperparameters using a Grid Search and Random Search for a Random Forest Model and a Gradient Boosted Tree Model?

- How does performance of PySpark vary as we tune hyperparameters using Cross Validation and Train Validation techniques for a Random Forest Model and a Gradient Boosted Tree Model?
- How does the performance of Python vs PySpark vary with change in the size of the dataset?

Part 2: On Databricks

- How does PySpark perform for basic data loading with data stored on an S3 bucket?
- How does PySpark perform for data imputation tasks?
- How does PySpark perform as we vary hyperparameters using Cross Validation and Train Validation techniques for a Random Forest Model and Gradient Boosted Model?
- How does PySpark perform for the conditions listed above when we vary the number of worker nodes from 2 vs 4 vs 8?

Methodology

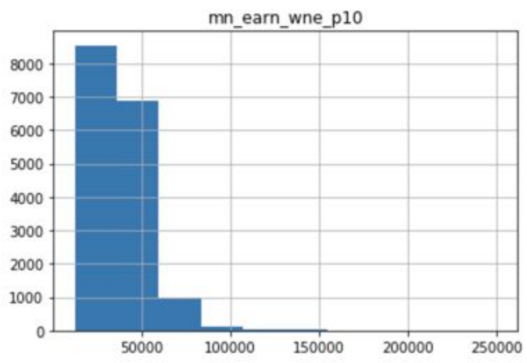
Dataset:

The dataset used is the US College Scorecard data that was released by the US Department of Education. College Scorecard provides a publicly available data set consisting of approximately 2000 metrics for 7805 degree-granting institutions. These metrics include demographic data, test scores, family income data, data about the percentages of students in each major, financial aid information, debt and debt repayment values, earnings of alumni several years after graduation, and more.

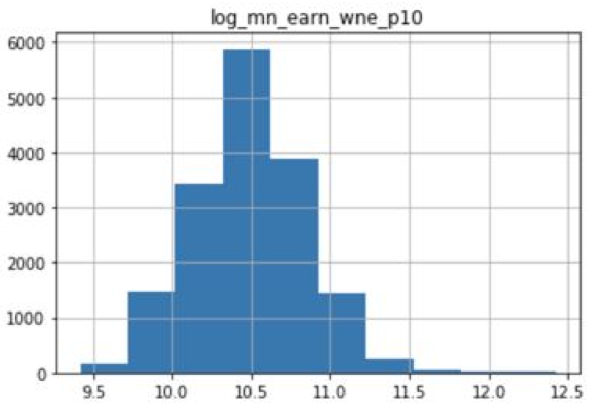
The goal of this project was to build a model that would accurately predict the earnings of a college's graduates after ten years based on the specific features.
To perform this regression task, the response variable used is:

- *mn_earn_wne_p10:* Median earnings of students working and not enrolled 10 years after entry.

The response variable was not found to have a uniform distribution.



A log transformation was applied, which resulted in the response variable being more uniformly distributed:



The predictor variables were identified as follows:

- ADM_RATE_ALL: Admission rate across all campuses
- COSTT4_A: Cost of Attendance for academic year institutions
- UGDS: Number of undergraduate students
- UGDS_BLACK: Total number enrolled undergraduate students who are black
- UGDS_WHITE: Total number enrolled undergraduate students who are white
- PPTUG_EF: Total number of students enrolled part time
- UG25abv: Undergraduate Student Body by Age
- PAR_ED_PCT_1STGEN: First generation students
- PCTFLOAN: Percent of Undergraduates Receiving Federal Loans
- C150_4: Percentage Completion Rate for a 4 year school
- TUITIONFEE_OUT: Tuition Fee for an out-of-state student
- TUITIONFEE_IN: Tuition Fee for an in-state student
- TUITFTE: Revenue/Cost of School
- AVGFACSAL: Average Faculty Salary
- INC_PCT_LO: Undergraduate students by Family Income - lower limit
- INC_PCT_H2: Undergraduate students by Family Income - upper limit
- PCTPELL: Percentage of Pell Students

- DEBT_MDN: Cumulative Median Debt
- PCTFLOAN: Percentage of Undergraduates Receiving Federal Loans

Part 1: Ingesting Data

To ingest data in Python, a pandas dataframe is used. The data is stored locally.
To ingest data in PySpark, a spark dataframe is used. The data is read from an S3 bucket on AWS.

Part 2: Data Imputation

To perform data imputation for missing values in Python, a SimpleImputer() function from sklearn's impute package is used. The missing values are replaced using the mean along each column.
To perform data imputation for missing values in PySpark, an Imputer() function from pyspark's ml package is used. The missing values are also replaced using the mean along each column.

Part 3: Training Models

To train a random forest and gradient boosted tree model in Python, RandomForestRegressor() and GradientBoostingRegressor() functions are used respectively from sklearn's ML regression packages.
To train a random forest and gradient boosted tree model in PySpark, RandomForestRegressor() and GBTRegressor() functions are used respectively from pyspark's ml package.

Part 4: Tuning Hyperparameters

To tune hyperparameters in Python for a Random Forest and Gradient Boosted Tree model, a Grid Search and Random Search was used. Grid Search is implemented using GridSearchCV() function in sklearn's model_selection package. Random Search is implemented using RandomizedSearchCV() function available in the same package.

A grid search performs an exhaustive search for selecting the best parameters. The model is trained for every combination of hyperparameters specified in the grid. By contrast, a random search selects random combinations of the hyperparameters to train the model. Typically, a random search will run faster, and finds the best set of hyperparameters.
The hyperparameters selected for tuning:

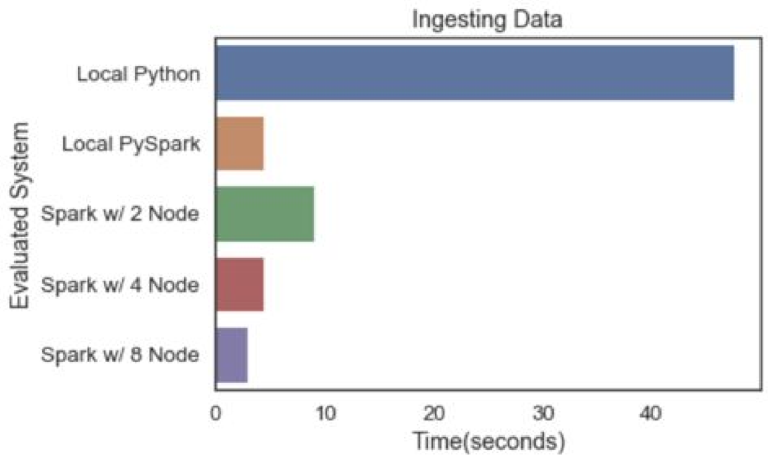
- Bootstrap
- max_depth
- Max_features
- min_samples_leaf

- min_samples_split
- N_estimators

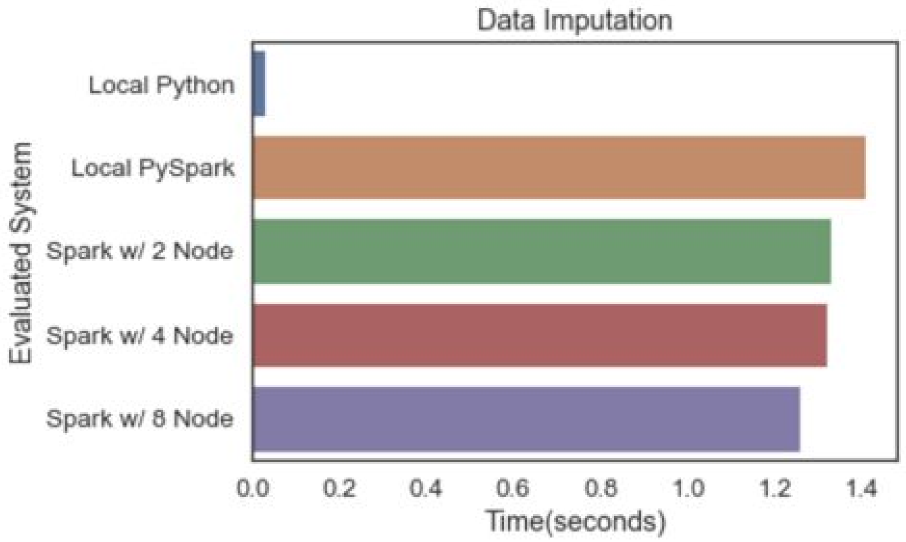
To tune hyperparameters in PySpark for a Random Forest and Gradient Boosted Tree Model, a CrossValidator() and TrainValidationSplit() was used from pyspark's ML Tuning package. CrossValidation begins by splitting the dataset into a set of folds which are used as separate training and test datasets. CrossValidator is pretty expensive since it is required to evaluate every possible combination of the specified hyperparameter values. In contrast, a TrainValidationSplit evaluates only a single random train/test data split per combination.

Results

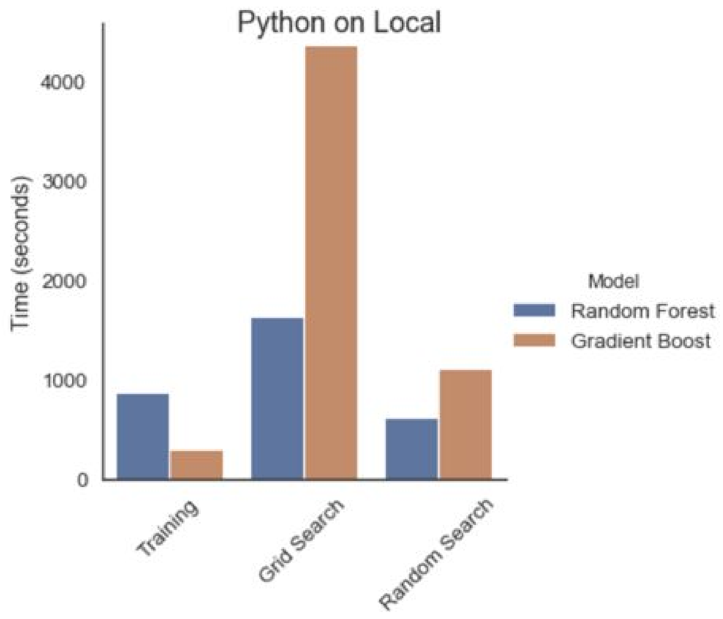
1. Ingesting Data



2. Data Imputation

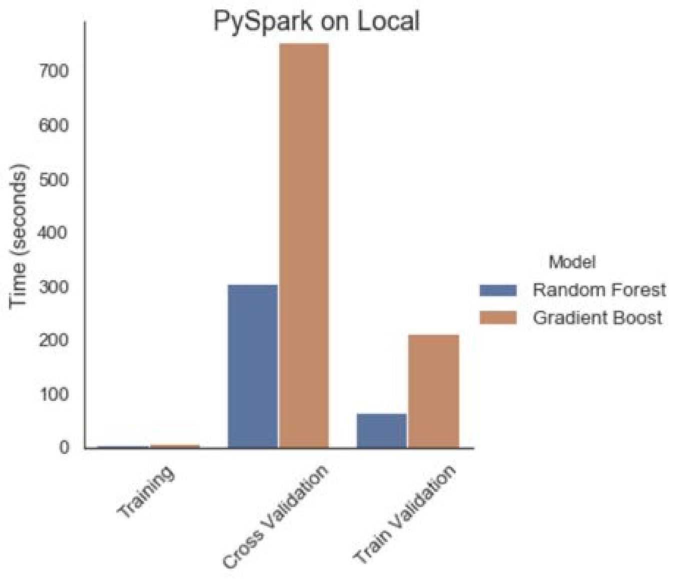


3. On Local Machine using Python ML Libraries

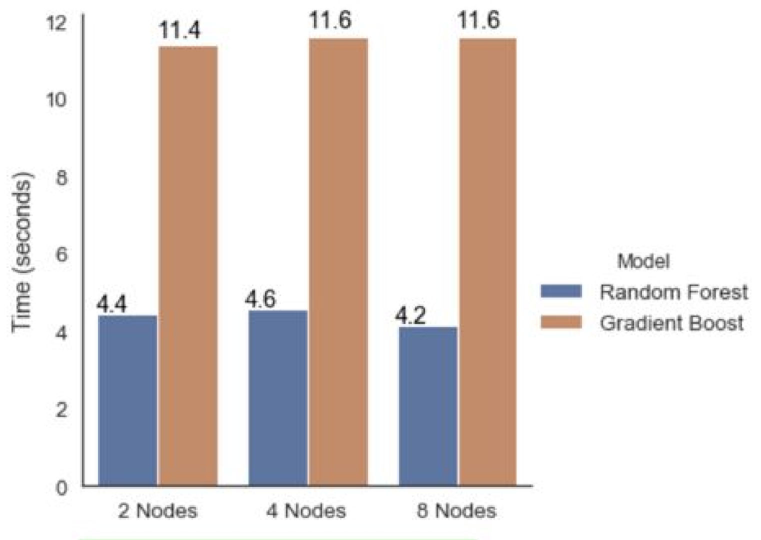


* Tuning parameters on the local machine for the entire dataset resulted in memory leak errors. Results are observed only for a smaller subset for the dataset (only for 20% of the data)

4. On Local Machine Using PySpark MLLib Libraries

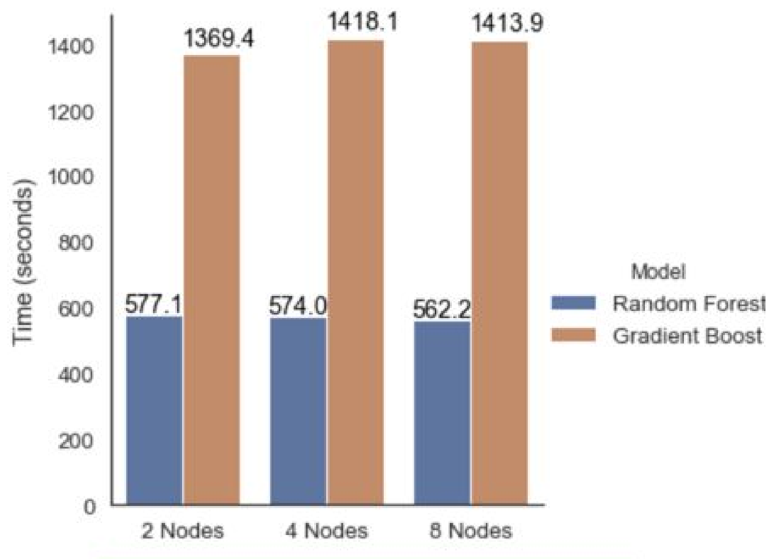


5. Databricks
    5.1. Model Training



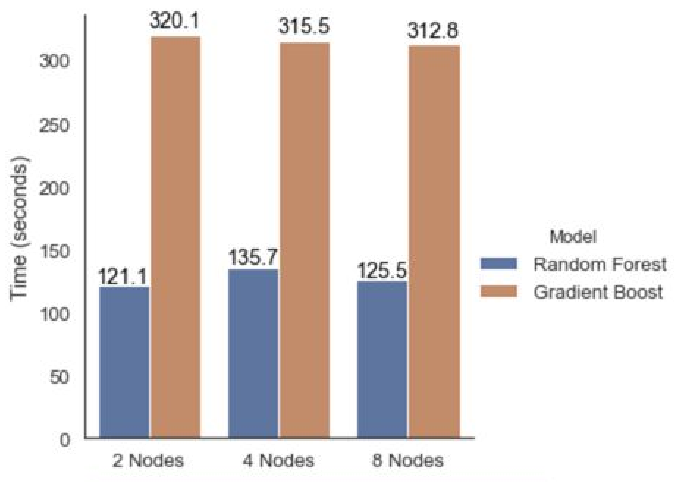
Model Training on Databricks

5.2. Cross Validation Tuning



Cross Validation Tuning on Databricks

5.3. Train Validation Tuning



Train Validation Tuning on Databricks

Conclusion

1. Data Ingestion
    - A Spark dataframe reads data faster than a pandas dataframe.
    - A Spark dataframe can also read data from cloud storage (S3 bucket) without any configuration required, while pandas requires boto3 library.

Conclusion: Reading data into a spark dataframe is quicker, and it should be used to achieve parallelism. Pandas will also run out of memory if the dataset is too large.

2. Data Preparation:
    - Spark dataframe defaults the data type for all columns as “string” and has to be converted to appropriate data types explicitly, while pandas infers them from the data.
    - Pandas has better data wrangling, filtering, aggregation functions compared to Spark.
    - Pandas requires a lot of memory resources to load data files compared to Spark whose operators spill data to disk if it does not fit in memory, allowing them to run well on any sized data.
    - Data Imputation is faster in Python compared to PySpark.

Conclusion: Pandas performs well with small datasets only; for large datasets Spark should be used. A spark dataframe can be converted to a pandas dataframe for data wrangling and preparation tasks to make use of the extensive functionalities that pandas provides. Pandas also works better for building visualizations (Graphs in the Result section were built in Pandas).

3. Training ML Models and Hyperparameter Tuning

   3.1. On Local Machine

   - Training models and tuning hyperparameters using PySpark’s ML Library is a lot faster than Pandas’ sklearn libraries, even with all available resources being used on the local machine *(n_jobs=-1*).
   - Grid Search Hyperparameter Tuning takes longer to compute than Random Search, as expected.
   - Tuning hyperparameters for a Gradient Boosted Tree model takes a lot longer than a Random Forest model. This is because a GBT model builds trees sequentially based on weak learners (high bias, low variance), whereas a random forest builds trees independently.

   3.2. Databricks

   - Time taken to train ML models improves when the number of worker nodes are varied from 2 to 4 to 8.
   - Random forest models are faster to train since they build decision trees parallely, compared to gradient boosted tree models which build trees one at a time.
   - Tuning hyperparameters using Cross Validation technique is costlier than Train Validation. This is because it only evaluates each combination of parameters once, as opposed to k times in the case of CrossValidator.

Conclusion: Databricks provides an easy interface to manage and configure clusters. Resizing clusters is also relatively quick.