Spark-lean: an interactive PySpark-based Data Cleaning Library

DS-GA 1004 Project: Final Report

Qintai Liu
Center of Data Science, NYU
New York, NY, USA
ql819@nyu.edu

Shuaiji Li
Center of Data Science, NYU
New York, NY, USA
sl6486@nyu.edu

Yicheng Pu
Center of Data Science, NYU
New York, NY, USA
yp653@nyu.edu

ABSTRACT

This paper provides a scalable and interactive Spark-based Python library that contributes to the routinedata cleaning process. Our library, Spark-lean, provides essential features that are capable of handling most of the typical data cleaning and wrangling tasks on massive datasets. Raw input data preprocessed through our library should be ready for future use, such as analytical illustration and predictive modeling.

KEYWORDS

Spark, Python, PySpark, SparkSQL, Interactive data cleaning

1 INTRODUCTION

The idea of Big Data [5] often referred as the massive datasets which are too large to be processed or stored through traditional toolkits, and the computing resources and strategies to handle those datasets. Currently, the most popular frameworks used by public for implementing large-scale data-intensive applications are MapReduce [4] (Apache Hadoop) and Apache Spark [11].

Apache Spark is not only compatible with Apache Hadoop but also highly effective through the in-memory computing mechanism. Due to the optimized overall data processing workflow, it is convenient to query and manipulate data with user's favored programming language APIs. For Python users specifically, Apach Spark provides the Spark Python API (PySpark) which integrates essential functionalities of core Spark engine and Spark SQL [2]. With the help of Spark SQL, users can load data from different sources and query structured data as a resilient distributed dataset (RDD). RDD tracks the past transformations to recover potential crashes on a single node [11]. DataFrame object can also be easily constructed for interactive structured data manipulations.

Since a qualified data pre-processing is desirable for any subsequent data related tasks, we leverage the advantages of Spark SQL and develop user-friendly data cleaning toolkit through Python API. Compared with the Java API, PySpark is relatively new and does not contain any holistic data cleaning module. The only PySpark-based cleaning library contributed by the community, Optimus[7], does not automatically produce intermediate results between consecutive operations, which is less effective from a debugging standpoint. For this project, we aim to build a cleaning library focusing on flexible operations and interactivity. Users should be able to easily execute commands and examine results with supportive descriptions.

2 RELATED WORK

Although there is a huge demand for Spark in both industry and academia. We have not found many qualified related work.

Optimus[7] is a high-quality open-source project based on PySpark which covered a lot of functionalities like data cleaning, profiling and feature engineering. With Optimus, users could implement functions like replacing null values, distinguishing column types and checking data distributions. Optimus makes data cleaning much easier for Pyspark users by packaging some common operations.

Pandas[9] is a well-known Python library which has a unique "Pandas.DataFrame" object. The Pandas dataframe provides Python users with a more convenient way to deal with tables, and also many other help functions for data cleaning and visualization. Pandas has been so popular that it is one of the best examples for Python data cleaning libraries. However, it is based on Python and does not support Spark.

Dora[6] is a data-cleaning library based on Pandas, it is a higher-level abstraction of Pandas' common operations. Operations like replacing missing-values and dropping columns sometimes are confusing in syntax for users, but in Dora they could be easily solved with a simple command like "immputing missing values". Dora's another interesting feature is its function does not return anything but only changes the data inside the Dora class. So users do not need to assign a new variable to store results. However, just like Pandas, Dora is also not designed to support Spark.

3 ARCHITECTURE

Although large scale data cleaning library like Optimus[7] has been developed, we are still looking for a better solution. With experience about different scale's data cleaning, we found that for small-scale data users could pursue more flexibility and transparency, but for large-scale data users need to worry about computational time and operations' complexity first. Providing user with full-transparency and flexibility i.e., a Pandas Dataframe, becomes very expensive. Also for small datasets many people like to do data-cleaning and exploration at the same time, but people dealing with large-scale datasets have to concentrate on the data-cleaning process, which makes doing data cleaning and exploration at the same time much harder.

Based on these observations, our goal is to create a scalable data cleaning library that provides users with flexibility and transparency to the maximum, helps users explore the datasets by interactive interfaces and optimizes the data cleaning process with algorithm.

We defined all our functions in a single cleaner class, and store the raw data inside of cleaner when we initialize the cleaner. The

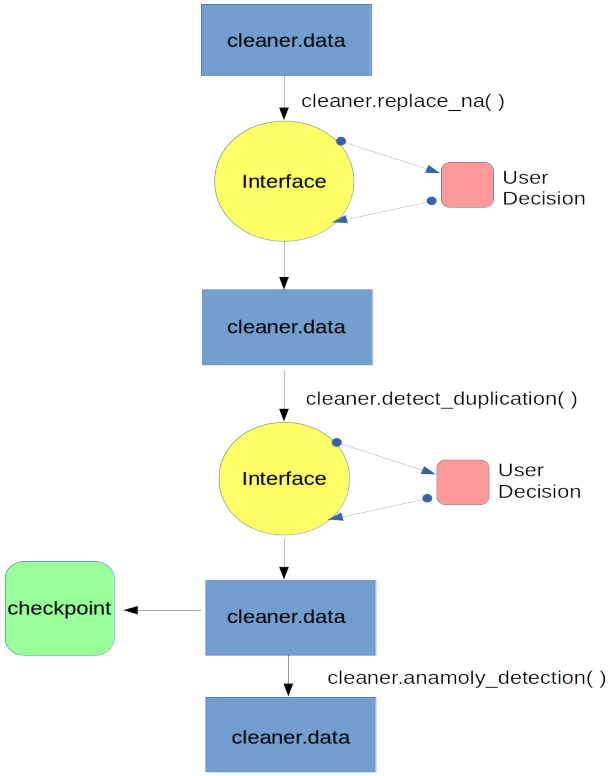

Figure 1: A sample diagram of Spark-lean's data cleaning process.

ideal data cleaning process should be a pipeline that tackles problems and explores data step by step, and all users need to do is to call methods of our cleaner class. Every method returns no object but changes the raw data stored inside the class. This design was inspired by Dora's[6] implementation, we believe it saves users' time from assigning the returned data frame to a new object. If a function needs to generate outputs, we would also store the outputs inside of the class, so users could call cleaner.out() to retrieve it.

Figure 1 is a diagram of our typical pipeline. The raw data is saved in cleaner object, users call methods like replace_na to the cleaner, get feedback like which columns contain missing values in the interface, then users could choose a pre-defined approach to deal with the problem. In the data cleaning process, users can freely save or load data into disk as checkpoints at any time. This version control function helps users to withdraw any operations and save backup copies.

3.1 An innovation in User-friendly Design

Let's take a closer look at our library using the function replacing_na as an example:

```
c=cleaner("data.csv","checkpoints")
c.detect_missing_value(['na','n/a'])

Output:
column _c8 has 523 null values!
column _c9 has 474 null values!
column _c0 has 4 rows that include
keyword: na
column _c0 has 30 rows that include
keyword: n/a
_________________
Please select an approach:
1. Delete all suspicious rows
2. Replacing suspicious rows with 0
3. Replacing suspicious rows with column
average
......
7. Do nothing
_________________
Input(number):1
_________________
All deleted!
```

During this process, users only need to execute one command to get diagnosis about missing values for the whole dataset. And the predefined approaches has covered most of the common operations, users could select an approach by just typing a number. If all the predefined approaches do not meet user's expectation, the result of all suspicious columns and null value columns are also stored in cleaner.out(), so the user could also implement customized operations based on that.

Compared with other libraries like Pandas or Optimus, our library simply use one-line code to finish both data exploration and data clean for missing values, the same task could take multiple operations for other libraries.

Another big advantage of such interface is that user do not need to memorize complicated syntax or spending searching online. Today there are dozens of popular packages and even more new packages are coming out. It is impossible to memorize all syntax of those functions. Our interactive library minimizes the pain of searching documentations and makes data scientist's life easier.

4 PROBLEM FORMULATION

Below we present detailed formulations of current major functions within Spark-lean library.

4.1 Indexing

The default data structure of Spark-lean is PySpark DataFrame object which does not support slicing by index like what Pandas library has. To provide users with this feature, we develop a series of functions that helps to construct, drop or reset a index column on every DataFrame object. Users may opt to implement further operations such as replacing or removing a target instance by utilizing the index column created through this function.

4.2 Instance Cleaning

So far, we have developed several instance-based functions that can be employed on user's favor. For any particular or a group of instances, users can choose to replace the original value with a new

value by either the original value or the index of the target instances. In case of faulty operations, a brief summary about number of items being replaced will be provided.

In addition, we provide users with two functions that allows them to drop instances by index or values, which facilitates data cleaning process, because users may also want to directly remove some instances with specific index or original values.

4.3 Feature Cleaning

Since this section involves analysing running time of a function, some common notation should be introduced first.

$\Theta - notation$ asymptotically bounds a function from above and below.

$O - notation$ gives an asymptotic upper bound on a function

$\Omega - notation$ provides an asymptotic lower bound on a function

$T - notation$ represents the whole data frame

4.3.1 Distinguish Numerical Features. PySpark supports various approaches to fetch or load csv files. However, one of the default loading function:

```
SQLContext.read.csv("yourfile.csv")
```

automatically covert all attributes to 'string' type. The resulting DataFrame object may contain misleading schema types that influence subsequent operations. For instance, if a numerical feature is falsely identified as a string type feature, users may ignore any numerical feature engineering process such as normalization, and instead falsely implement one-hot encoding for predictive modeling purpose. Besides, it's also possible that some attributes in the raw input csv file are incorrectly collected due to technical or human errors.

We hence provide this function to help users identify real numerical attributes. One thing to note is that if we examine all instances row by row, then the computational complexity is $\Theta(n)$, which is expensive for a large dataset. Instead, we check the data type on 10 randomly sampled instances from each attribute. The computational load thus decreases drastically without loss of much accuracy.

4.3.2 Duplicate Features Detection. In practice, it's likely that two features or more have exactly the same value in each row, implying that these features are duplicate. If users use duplicate features to do data mining, these features may negatively influence the model's performance. Here we provide a function that detects whether there is any pair of two features with exactly same value, and remind users with the detected pairs. We provide three subsequent options: the first two are about deleting duplicated pairs, and the last one is doing nothing.

The main challenge of this function is the complexity. Suppose m is the number of features, n is the number of rows. For a big dataset, where m is greater than one hundred and n is greater than 1 million, the detecting process is extremely time consuming. What even worse is that PySpark does not provide any function allowing users to get the i-th row value in $\Theta(1)$. The running time for querying a specific row is $\Theta(n)$ on default. And for each pair, the running time is $\Theta(n^2)$ if we compare row by row. If we wish to compare all possible pairs of all the features, there are $\Theta(m^m)$ pairs and the total complexity of detecting pairs with duplicated feature values is $\Theta(m^m n^2)$.

To reduce the complexity of detection, we take into account the fact that there is a relatively small chance that two features share the exactly same values across all rows. Instead of comparing all pairs of features like above, we select a candidate pair which shares same values in their first 20 rows. Any pair of features which doesn't match the first 20 rows certainly can't match all rows. Then we compare the next 20 rows and so on. We immediately stop searching and comparing once a single difference appears within this candidate pair, and move on to another pair.

Another improvement we make is utilizing the inner join to check each candidate pair $(L\_feature, R\_feature)$. The following relational algebra is how we implement this function:

$$L\_table = \Pi_{L\_feature}(T)$$
$$R\_table = \Pi_{R\_feature}(T)$$
$$result = F_{count(x)}(L\_table \bowtie_{L\_feature=R\_eature} R\_table)$$

If the result is equal to the number of row, then the candidate pair

$$(L\_feature, R\_feature)$$

has the same values for all rows. By doing this, we reduce the running time of each pair of features' comparison from $\Theta(n^2)$ to $\Theta(n)$. With these two improvements, we are able to lower the best running time from $\Omega(m^m n^2)$to $\Omega(m^m)$ if no same features exist. Suppose there are x pairs of features having the same value, the running time for this method is $\Theta(m^m nx)$

4.3.3 Features with All-Same-Values Detection. We also developed another function to detect any column containing only one unique value. If a feature has the same value in all rows, it is highly possible that the feature is created by mistake or something is wrong on data collection phase. Such a column may not be able to bring us with much valuable information.

From the above section, we mentioned that the running time of retrieving a specific row in Spark is $\Theta(n)$. If we keep comparing the first instance with all other instances in the column as long as no difference showing up, the lower bound time complexity is $\Omega(n)$. The issue is that if the feature does has all rows with the same value, the running time for this function will increase dramatically to $\Theta(n^2)$. We make use of some PySpark SQL built-in functions to optimize the detecting process. First, we obtain the first row's value for the target feature x. Then we use the following relational algebra

$$result = F_{count(x)}(\sigma_{targetfeature=x}(T))$$

to see how many row has the same value with the first row. We can check whether the result is equal to the number of rows. If they are equal, then it means this feature has the same value in all rows. The running time of this function is $\Theta(n)$.

4.4 Advanced Methods

4.4.1 Outlier Detection. Based on IQR rule[12], we mark any instance smaller than $Q_1 - 1.5\text{IQR}$ or larger than $Q_3 + 1.5\text{IQR}$ as outliers, where Q_1 is the first quantile, Q_3 is the third quantile, and

IQR is $Q_3 - Q_1$. To provide users with more flexibility, we set 1.5 as a threshold parameter. The function will return cleaned dataset without potential outliers if users request

4.4.2 Similar string matching. This function allows users to input a string and find highly similar words in a certain column. Implementation of this method was combined with character Bi-gram vectorization and Min-Hash. For a given string, we split it into multiple sets of character Bi-grams. For example, the string "new york" is a combination of $\{ne, ew, w\_, \_y, yo, or, rk\}$.

With the Bi-grams from all strings in that column, we could build a count-vectorizer of bi-grams, and then apply it to each string. For "new york", its corresponding vector would be:

ad	wc	xt	...	ne	ew	ws
0	0	0	...	1	1	0

The vectorized representation of strings enables us to perform nearest neighbor clustering like KNN[1]. However, KNN compares the Euclidean Distance between each pair of strings and could be pretty time-consuming for large datasets. So here we chooses Min-Hash as an alternation for KNN.

Min-Hash[3] applies n random hash functions to each string, and then keep the min value. It is proved that the probability of set A and B has the same Min-Hash value, is equal to their Jaccard Similarity $J(A, B)$.

$$J(A, B) = \frac{A \cap B}{A \cup B}$$
$$Pr[h_{min}A = h_{min}B] = J(A, B)$$

4.4.3 Anomaly Detection by KMeans. The function will detect anomalous data by K-means[8]. The pseudo code is listed in Algorithm 1.

The advantage of this function is that it is able to standardize features before doing the K-means clustering to deal with features having different scale. In addition, when it is doing anomaly detection, only points belong to the same cluster as p are considered, so the result should be more accurate and meaningful.

Algorithm 1 Anomaly Detection by KMeans

procedure AnomalyDetectionByKMeans(k, columns, standardize, threshold)
  k: the number of clusters
  columns: a list of columns' names fitted into the K-mean
  standardize: whether standardize features
  threshold: the threshold for outliers
  if *standardize* = *True* **then**
    standardize each feature in *columns*
  end if
  call K-means clustering(k=*k*)
  clusters ← all clusters
  for each $C \in$ *clusters* **do**
    $mean_C$ ← calculate the average distance from all points belonging to C to its centroid
    std_C ← calculate the std of distance from all points belonging to C to its centroid
  end for
  for each $p \in$ all data **do**
    $C \leftarrow$ which cluster p belonging to
    $d_p \leftarrow$ the distance from p to the center of C
    if $(d_p - mean_C)/std_C > threshold$ **then**
      p is considered as anomaly
    end if
  end for
end procedure

5 RESULTS

We did experiment with 3 basic functions we have finished with the NYC parking-violation dataset [10]. The dataset has 1,014,017 rows and 22 columns.

The duplicated column detection function successfully found the duplicated index we manually added into the dataset. Also, by observing the running process, we found the program spent a lot of time inspecting column "violation precinct" and "violation location". We then looked into the data and found that these two columns are highly similar but not exactly the same. The total running time for this function on parking violation data is 180 seconds.

The missing value detecting function was tested with three keywords "NA", "n/a" and "N/A", the function successfully found 10 columns containing NULL values and three columns containing "N/A". The process took 241 seconds including both detecting missing values and deleting missing values.

The useless feature detection function did not detect any single-value column (useless-feature) in the original dataset. The process took 98 seconds. However, after we remove all the null values, this functions detected three single-value columns: violation code, violation description and issuing agency. This gives us an insight that only certain kinds of violation and certain agency's records are completed.

During the experiment, we surprisingly found this interesting insight and the high-similarity between those two columns. The results are summarized in table 1. Previously we have used the same dataset in two homeworks but no one in our team has noticed those characteristics due to the huge volume of data. With our library we made it by just three short one-line commands:

```
c.drop_duplicate_column()
c.detect_missing_value()
c.drop_column_with_single_value()
```

Task	Time	Achievement
Duplicated Column	180s	High-similarity relation
Missing Value	241s	Columns with missing values
Useless feature	98s	Insights about data completeness

Table 1: Performance of Basic Functions

Besides basic functions, we also tested our advanced functions on two datasets: a small dataset with around 240,000 rows and 32 columns and a large dataset with around 4,400,000 rows and 56 columns. We tested 5 advanced functions on both datasets to compare the running time. For the most time-consuming functions we only tested it on smaller dataset because they are not supposed to be used on large dataset without simplification. The results are summarized in table 2.

Task	Small dataset	Large dataset
Distinguish Numeric Format	4.51s	125s
Detect Missing Value	21.23	617s
Anomaly Detection	20.04s	57m
Detect Outliers	2.23s	53.21s
Detect Useless Features	20.48s	N/A
Drop Duplicated Column	252s	N/A
Similar Words Matching	2s	49.45s

Table 2: Comparison between Small Dataset(130MB) and Large Dataset(3.5GB)

In the end, we also compared Pyspark with another Spark-based data cleaning library Optimus in table 3. Optimus is not only a data-cleaning library but also a machine learning library, here we only compared the data-cleaning related functions. Both Spark-lean and Optimus covered basic functions of data cleaning process like replacing or removing successfully, however Spark-lean cover more advanced functions like anomaly detection.

Function	Optimus	Spark-lean
Normalize Feature	×	×
Clean String	×	×
Replace	×	×
Remove	×	×
Distinguish Numeric Format		×
Detect Missing Value	×	×
Anomaly Detection		×
Detect Outliers	×	×
Detect Useless Features		×
Drop Duplicated Column		×
Similar Words Matching		×
Outlier Detection	×	×

Table 3: Comparison between Optimus and Spark-lean

6 FUTURE WORK

Currently we have finished all the major architecture of this library and some advanced functions like similar string matching and anomaly detection. This is clearly not the end of our development phase. We have packaged and uploaded our library to PyPI where all the public libraries can be maintained and contributed by the community.

Our current limitation primarily lies on effectively applying some advanced functions on large datasets. In the next step we would refine our on-line documentation and further optimize the main cleaner class. We would continuously focus on improving its computational performance and user experience. If possible, we would devote more time to expand the range of acceptable input format such as unstructured data type (i.e. text), and develop a GUI for interactivity.

7 REPOSITORY

All the code could be found in the github repository:
https://github.com/qltf8/1004_LPL_project

To apply the library in practice, visit PyPI for more information:
https://pypi.org/project/Spark-lean

REFERENCES

[1] Naomi S Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394. DOI: http://dx.doi.org/10.1145/2723372.2742797

[3] Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*. IEEE, 21–29.

[4] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. DOI:http://dx.doi.org/10.1145/1327452.1327492

[5] Justin Ellingwood. 2016. An Introduction to Big Data Concepts and Terminology. (2016). https://www.digitalocean.com/community/tutorials/an-introduction-to-big-data-concepts-and-terminology

[6] Nathan Epstein. 2017. Dora. (2017). https://github.com/NathanEpstein/Dora

[7] Iron. 2018. Optimus. (2018). https://github.com/ironmussa/Optimus

[8] James MacQueen and others. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.

[9] Wes McKinney. 2010. pandas: a Foundational Python Library for Data Analysis and Statistics. (2010).

[10] City of New York. 2016. Parking Violations Issued - Fiscal Year 2016. (2016). data retrieved from NYC Open Data, https://data.cityofnewyork.us/City-Government/Parking-Violations-Issued-Fiscal-Year-2016/kiv2-tbus/data.

[11] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. 10 (07 2010), 10–10.

[12] Kokoska. S Zwillinger. D. 2000. *standard probability and Statistics tables and formulae* (1st ed.). CRC Press.