Covid-19 data analysis and new cases prediction with Machine Learning algorithms

Implemented in Hadoop MapReduce and Spark

Vera Yaseneva
University of Stavanger, Norway
v.yaseneva@stud.uis.no

Md Ruhul Islam
University of Stavanger, Norway
mr.islam@stud.uis.no

Isar Adnan
University of Stavanger, Norway
i.adnan@stud.uis.no

ABSTRACT

Coronavirus has become a serious concern because of its unpredictable and highly contagious nature. Diverse machine learning, computer vision and cloud computing techniques are being used in several studies to analyze the potential causal relationship among multiple factors, identification of infections and prognosticating the number of new cases of COVID-19 infection through extrapolation of the generated data. In this project diverse machine learning algorithms are implemented in Hadoop MapReduce and Spark frameworks to analyze the latest data of COVID-19 pandemic. In the exploratory data analysis, SVD - Singular Value Decomposition is used to compute PCA and K-Means Clustering algorithms are applied for dimensionality reduction and grouping of the data. In addition to this, an ensemble learning method, Random Forest regression is used to predict numbers of new cases of Coronavirus-infected patients for a set of countries.

KEYWORDS

Machine Learning, Random Forest Regression, K-Means Clustering, SVD, MapReduce, Hadoop, Spark

1 INTRODUCTION

Coronavirus is a novel severe acute respiratory disease that has become a major threat to the whole world. A lot of data related or possibly related with different aspects of pandemic is being collected for analysis in attempts to find key variables in spread and health related effects of the virus.

There is an ongoing search for Machine Learning based solution that could help to better understand and predict different aspect of the disease in such areas as forecasting, medical diagnostics, drug development, and tracing the spread of the infection among the population.

In this project we have applied several Machine Learning algorithms to Coronavirus related dataset. We implemented Singular Value Decomposition to compute PCA for dimensionality reduction from scratch, implemented the decomposition technique PCA, using machine learning library and, implemented K-Means clustering algorithm to perform clustering or grouping of the data. Additionally, we implemented Random Forest Regression algorithm to attempt prediction of the number of new cases of COVID-19 infection for January, February and March 2021 based on the data collected for the previous year from multiple countries.

Supervised by Thomasz Wiktorski.

Project in Data-intensive systems (DAT500), IDE, UiS
2021.

We have implemented the algorithms in Hadoop and Spark, two data processing frameworks designed specifically for working with large volumes of data. After that, we have analyzed the performance of the implemented algorithms to prove that our implementation supports the scalability and parallelization offered by the MapReduce programming paradigm.

Our implementation can be found in the git repository: **https://github.com/0xf8f8ff/DIS**.

2 BACKGROUND

In this section we introduce two main tools used for data-intensive processing: Apache Hadoop and Spark. We also present the Random Forest regression, K-Means clustering and Singular Value Decomposition Machine Learning algorithms that we want to implement in our project.

2.1 Apache Hadoop

Apache Hadoop is an open-source software framework that supports working with extra large volumes of data. Two main cores of a Hadoop system are HDFS and MapReduce. HDFS or Hadoop Distributed File System is the storage layer responsible for distributed storage of data. MapReduce is a layer responsible for processing of the stored data. Hadoop is specifically designed to be capable of handling large numbers of concurrent tasks or jobs due to its ability to run in distributed mode on clusters of multiple physical machines connected in a network[14].

2.1.1 *MapReduce.*

MapReduce is a programming paradigm designed to support parallel distributed data processing by converting datasets to sets of key-value pairs in the mapping phase, and combining and reducing these tuples in the consequent phases into smaller sets of tuples. Map and Reduce operations can be sequenced across a distributed set of cluster nodes by assigning data segments to nodes for parallel processing and aggregation[9].

2.2 Spark

Apache Spark is another data-intensive processing framework capable of running its tasks in distributed mode on multiple cluster nodes to achieve high-performance data processing. It is a successor framework to Hadoop aiming to overcome some of its traits negatively affecting the performance.

Hadoop uses disk reads and writes when managing intermediate results. This can significantly increase running time when it is necessary to perform data analysis consisting of multiple steps. Sometimes the impact of number of required steps between calculations has more significant impact on the performance than the size or complexity of the datasets the calculations are being performed on. Unlike Hadoop, Spark is capable of in-memory storage of intermediate data[14].

2.3 PCA Through SVD

Principal component analysis of a data matrix extracts the dominant patterns in the matrix in terms of a complementary set of score and loading plots. The results of the analysis depend on the scaling of the matrix, which therefore must be specified [15].

Singular value decomposition (SVD) can be looked at from three mutually compatible points of view. On the one hand, we can see it as a method for transforming correlated variables into a set of uncorrelated ones that better expose the various relationships among the original data items. At the same time, SVD is a method for identifying and ordering the dimensions along which data points exhibit the most variation. This ties in to the third way of viewing SVD, which is that once we have identified where the most variation is, it's possible to find the best approximation of the original data points using fewer dimensions. Hence, SVD can be seen as a method for data reduction [4].

Singular value decomposition takes a rectangular matrix of gene expression data (defined as A, where A is a n x p matrix) in which the n rows represents the genes, and the p columns represents the experimental conditions.The SVD theorem states [2]:

$$A_{nxp} = U_{nxn} S_{nxp} V^T_{pxp}$$

The objective of using the principal component analysis in this project is to reduce the dimensionality of the higher ranked vectorized data to a lower ranked vectorized data that contains few principal components comprising variations as maximum as possible. Most importantly we have used this dimensionality reduction algorithm to extrapolate information that is use for the groping and prediction of the number of new covid cases [1]. We have implemented SVD to compute PCA from scratch and implemented PCA using the python machine learning library too. Also, calculated the variance ratio (reconstruction error) of both of these two implementations to see the performance and accuracy.

2.4 K-Means Clustering

K-means algorithm is the most well-known and commonly used clustering method. It takes the input parameter, k, and partitions a set of n objects into k clusters so that the resulting intra-cluster similarity is high whereas the intercluster similarity is low. Cluster similarity is measured according to the mean value of the objects in the cluster, which can be regarded as the cluster's "center of gravity". The algorithm proceeds as follows: Firstly, it randomly selects k objects from the whole objects which represent initial cluster centers. Each remaining object is assigned to the cluster to which it is the most similar, based on the distance between the object and the cluster center. The new mean for each cluster is then calculated. This process iterates until the criterion function converges.In k-means algorithm, the most intensive calculation to occur is the calculation of distances. In each iteration, it would require a total of (nk) distance computations where n is the number of objects and k is the number of clusters being created [16].

In this project we have implemented K-means clustering to group the data of 190 countries to see the similarity between the infection spreading of novel corona virus among the countries that would help to develop a model further for a further analysis and predicting the further infection spreading rate of this virus. We have implemented this algorithm step by step on Hadoop architecture to see how the clustering works on a distributed environment.

2.5 Random Forest Regression

Random Forests is a statistical method introduces by Leo Breiman in 2001[5]. It is shown to have excellent performance for classification and regression problems alike when compared to a single prediction tree. The method gives good results for noisy data and data with large numbers of variables and relatively small number of observations[12].

A random forest is composed of a large number (at least one thousand, often more) of decision trees. Its randomness originates from two sources. First of all, each tree is based on a random bootstrap subset of the original training set. Additionally, each split within a tree is being made based on a random subset of all features. The final prediction of the forest is the average of all predictions from each individual tree.

2.5.1 Bootstrap sampling with aggregation. Bootstrap is a statistical procedure for estimating quantities from a data sample. An important part of the method is creating a large number of random subsamples of the original sample with replacement. Random Forests use bootstrap samples when building trees. Each tree gets a different subsample as its training data and then makes a prediction given a testing set. The final prediction will be the average of predictions from each individual tree model.

2.5.2 Feature selection and splitting rules. A single CART tree selects a split point by comparing all variables and choosing the one which is most optimal for the current node[6]. Because of that, even multiple CART trees trained on bootstrap subsamples will have structural similarities and significant correlation of their predictions. A Random Forest tree is limited in its choice to a set of candidate variables randomly chosen from the set of all dataset features. As a result, the predictions from multiple trees are uncorrelated or weakly correlated[10].

2.6 Related works

In this subsection we will give a brief review of related works we used to build a knowledge and understanding of different algorithms necessary in creation of our project.

Applicability of Machine Learning models to COVID-19 data. [7] gives us some insights about using Machine Learning methods in analysis of data collected in relation to the COVID-19 pandemic. In particular, the authors present a comparison of two different models: Random Forests and Gradient Boosting Machine used for regression modelling.

Random Forests. To understand the basics of Random Forests we first study two works of Leo Breiman, the creator of the method. In [6] the author explains CART: classification and regression tree models. In [5] he introduces Random Forests and explores different approaches to defining rules for splitting the tree nodes.

Further, we study a few additional works on Random Forests to understand the details and reasoning behind the different implementations of the algorithm. The author of [10] explores methods of defining the importance of the dataset features when looking for the best possible split for each new tree node. The author also talks about the effect of pruning of the decision trees on the performance and precision of the algorithm.

In [12] the author is discussing various splitting rules for decision trees in random forests and their effect on processing of noisy data and how the different rules affect the precision of predictions.

A practical application of the Random Forest Regression in MapReduce architecture is presented in [9] and serves as an inspiration for our work.

Finally, [14] is our guide to Hadoop MapReduce and Spark frameworks.

3 IMPLEMENTATION

In this section we will present the datasets we use in the project and our implementation of the Machine Learning algorithms. We will describe the challenges we have met and explain the choices we have made to overcome those challenges in the course of our work.

3.1 Datasets

Our main dataset is a collection of COVID-19 data maintained by Our World in Data[11]. However, due to Coronavirus being a new disease, there is roughly one year of data, and the dataset is only 21 MB in size. To perform calculations on a dataset this small there is hardly any need for a distributed cluster of several machines.

To make the task more challenging we use a second dataset. This additional dataset is in fact a collection of monthly datasets of air traffic records from across the world collected by The OpenSky Network 2020[3]. It is created to illustrate the development of air traffic during the COVID-19 pandemic, is 4.3 GB in size and constantly growing. We decided to preprocess this additional collection of data to extract and add two new columns to the original dataset. One column will contain daily numbers of international arrivals for each country, another column will hold data about internal flights on the same date. This way we are getting the experience of working with larger datasets and, at the same time, provide two new variables for analysis of the original data.

There are also three helper datasets that we will use in the preprocessing phase. One is provided by the swedish sources and supplies information about the numbers of new tests for COVID-19 infection in Sweden for the first part of 2020[8]. This data is mostly missing in the original dataset. Two other datasets are published by OwrAirports and are required to associate the source and destination airports of the flights with their respective countries[13].

3.2 Preprocessing

The goal of the preprocessing stage in our project is to clean the original dataset and to add two new columns with daily data: one for domestic flights inside the country, another one for the number of international flights arriving in the country on this day. Most of the preprocessing is written in Python using MapReduce framework with Python's *mrjob* library. Only the last step was implemented in two variants, one with *pandas* frameworks, another one with *mrjob*. Unfortunately, the mrjob variant of this part has not yet been fully debugged as we were short on time and decided to focus on better implementation and optimization of more complex algorithms.

3.2.1 Cleaning. The main dataset is in csv format and, at first glance, appears to be perfectly structured and not in need of any additional preprocessing. However, when studied closely, some data appears to be missing, especially for the first months of year 2020.

Some countries barely have any data at all. We want to completely remove entries for countries that miss a lot of dat and only consider a subset of all countries represented in the dataset. A file that lists all the countries we want to select is being loaded by all mappers during the *mapper_init* stage.

```python
def mapper_init(self):
    self.locations = list()
    for country in csv.reader(open("country_list", "r")):
        self.locations.append(country)
```

Having a separate file with countries instead of hardcoding all the country names allows us to edit the list by adding and removing countries and restarting the preprocessing routine to adjust the size of the resulting dataset.

Some columns in the main dataset, like $handwashing\_facilities$, $icu\_patients\_per\_million$, and others, do not have any valid data for many of the present countries. Another columns are just for commodity and do not add any statistical significance to the data, like ISO codes of the countries. We want to remove such columns from the dataset.

```python
df = df.drop([
    'iso_code', 'continent', 'new_cases_per_million', '
     new_deaths_per_million',
    'icu_patients_per_million', '
     hosp_patients_per_million',
    'weekly_icu_admissions', '
     weekly_icu_admissions_per_million',
    'weekly_hosp_admissions', '
     weekly_hosp_admissions_per_million',
    'new_tests', 'total_tests_per_thousand', '
     new_tests_per_thousand',
    'new_tests_smoothed', '
     new_tests_smoothed_per_thousand', 'tests_units',
    'handwashing_facilities', 'hosp_patients', '
     icu_patients', 'positive_rate',
    'tests_per_case'
], axis=1)
```

After checking the resulting dataset for null values we can see that some datapoints are still missing. For example, there is no values at all for extreme poverty in Germany. We take this information from The World Bank Groups to get a reasonable estimate.

```python
    df.loc[df.location == 'Germany', 'extreme_poverty'] =
      0.2
```

Data for testing numbers in Sweden is also incomplete. We supplement it from the first helper dataset[8].

```python
 df_total_tests_Sweden = df_tests_Sweden.groupby('date').
     sum().cumsum().reset_index()
 df_Sweden = df.loc[df.location == 'Sweden']

```

```
df_Sweden = pd.merge(df_Sweden, df_total_tests_Sweden,
    how='left', on='date')
df_Sweden.drop('total_tests', axis=1, inplace=True)
df_Sweden = df_Sweden.rename(columns={'count': '
    total_tests'})

df = df.loc[~(df.location == 'Sweden')]
df = df.append(df_Sweden, ignore_index=True)
```

3.2.2 *Lookup table: **preprocess_ISO_codes.py**.*

After cleaning the original dataset we want to add two new columns with daily data: one for domestic flights inside the country and one for the number of international flights arriving in the country on this day. The first problem is how to connect airport data from the second dataset with the country records in the main dataset. The main dataset has the name of the country and 3-letter ISO code in its records. The second dataset only has 4-letter ICAO code of the airport.

To identify airports belonging to the specific countries we need to create a lookup table. For this we need to join the two helper datasets, one of them lists airport ICAO codes with 2-letter ISO codes of the countries, the other one associates ISO codes with the country names. Both datasets are renamed to $codes0.csv$, $codes1.csv$ and moved to HDFS to be read simultaneously in the mapper stage.

The first mapper reads both files and emits ISO code as key and either country name from the first dataset, or the airport code from the second as a value.

```
def mapper_ISO(self, _, line):
    row = line.split(',')
    # skip first line
    if row[0] != "id":
        # the row is from the ICAO code dataset
        if len(row) >= 9:
            if row[2] in airport_types:
                yield (row[8], row[1])
        # the row is from country ISO dataset
        elif len(row) > 4 and len(row[0]) == 6:
            yield (row[1], row[2])
```

Reduces aggregates the data for each ISO code and sends it to the next step. In fact we perform a join of two datasets. Reading both datasets into the mapper line by line is not a very efficient way of joining, but because both datasets are very small it is not crucial to the running time of the joining process.

```
def reducer_ISO(self, key, values):
        yield key, list(values)
```

The second mapper constructs the required lookup table of the format

$$[ICAO\ code,\ Country\ name]$$

```
def mapper_ICAO(self, key, codes):
    # ignore empty keys
    if key != "" and len(key) > 2:
        if len(codes) > 1:
            countryName = ""
            for code in codes:
                if code in self.locations:
                    countryName = code
            if countryName != "":
                for code in codes:
                    if code != countryName:
                        line = ','.join([code,
    countryName])
                        yield (None, line)
```

3.2.3 *Processing flight data: **preprocess_flights.py**.*

In the next step we use the created lookup table to process all 4GB of flight data into a single dataset with only the relevant data.

At first we copy the lookup table to each worker node and save it as a dictionary.

```
def mapper_init(self):
    self.lookup_table = {}
    for icao, country in csv.reader(open("lookupfile.csv"
    , "r")):
        self.lookup_table[icao] = country
```

After that, the mapper reads all the datasets with flight data and emits rows formatted as

$$[date,\ country\ of\ origin,\ country\ of\ destination]$$

```
def mapper(self, _, line):
    row = line.split(',')
    # skip the first line
    if row[0] != "callsign":
        relevant = False
        # replace ICAO codes of airports with country
    names
        if len(row) > 9:
            if row[5] in self.lookup_table:
                relevant = True
                row[5] = self.lookup_table[row[5]].
    replace("\t", "")
            if row[6] in self.lookup_table:
                relevant = True
                row[6] = self.lookup_table[row[6]].
    replace("\t", "")
            if relevant:
                date = row[9].split(" ")[0]
                key = ','.join([date, row[5], row[6]])
                yield (None, key)
```

3.2.4 *Extracting flight data columns.*

At last, we have to aggregate and extract all the information about flights in form of two columns with daily data for each country, one for the number of international arrivals, and one for the number of internal flights on the same day. The resulting data will be ready for joining with the main dataset based on the date and country name. The intermediate format is:

$$[date, country, internal\ flights, international\ arrivals]$$

Mapper emits a list of the date and country values as a key. The value is a two element boolean array indicating whether the flight was internal or international. A value of **[1,0]** indicates a domestic flight, while **[0,1]** is for international arrivals.

```
# yield format [internal, international]
    def mapper(self, _, line):
        row = line.split(',')
        # ignore too short lines or lines with empty
    destination, date
        if len(row) > 2 and row[0] != "" and row[2].
    replace("\t", "") in self.locations:
            key = ','.join([row[0], row[2].replace("\t",
    "")])
            if row[1] == row[2].replace("\t", ""):
                yield (key, [1,0])
            else:
                yield (key, [0,1])
```

The reducer then counts the number of internal flights and international arrivals for each [*country*, *date*] key.

```
def reducer(self, key, values):
    inflights = 0
    intflights = 0
    for val in values:
        inflights += val[0]
        intflights += val[1]
    line = ','.join([key, str(inflights), str(intflights)
     ])
    yield (None, line)
```

We are using the *RawValueProtocol* output protocol that ignores keys overall in our preprocessing code. This way it is easy to write files in csv compatible format without implementing any custom protocols for csv.

After the preprocessing stage the new dataset is ready to use in the development and application of the different Machine Learning algorithms we want to create.

3.2.5 Preprocessing for PCA and Clustering. For PCA and Clustering we have considered data of all countries and taken the maximum values of each country. Feature-wise we have taken all 39 features before doing the decomposition. For the rest of the features there were no values and we have dropped then as mentioned earlier. Upon preparing the final cleaned dataset we have added the id as the "country_id" for each of the data and saved the data set in both csv and text file for further utilization. Here is the code doing these preprocessing as mentioned:

```
country_list = df['location'].unique()
countries = country_list.tolist()
df = df[df['location'].isin(countries)]
df = df.reset_index(drop=True)
df = df.fillna(0)
df_grouped = df.groupby('location')
taking_maximums = df_grouped.max()
final_df = taking_maximums.reset_index()
final_df = final_df.drop(['location','date'],axis=1)
matDF = final_df.to_numpy()
data_for_mapreduce_pca = np.vstack((ids, matDF.T)).T
np.savetxt('pcaorgdata.txt', data_for_mapreduce_pca)
```

Once the cleaned data is ready along with tagging the country then the data is formatted for the PCA to segregate the values as key-value pair for mapper and reducer functions. While adding the ID into the data matrix the IDs are automatically converted as floating and while saving into the text file these values were converted as string which has been handled here and once again saved in a text file:

```
for line in lines:
 re_key1 = re.compile(' ')
 re_key5 = re.compile("\n")
 line = re_key5.sub('', line)
 a = re_key1.split(line)
 if i <= 9: a[0] = a[0][a[0].find('.') - 1:a[0].find('.')
     ]
 if i > 9 and i <= 99:
 ID1 = a[0][a[0].find('.') - 1:a[0].find('.')]
 ID2 = a[0][a[0].find('.') + 1:a[0].find('.') + 2]
 a[0] = ID1 + ID2
 if i > 99 and i <= 190:
 ID1 = a[0][a[0].find('.') - 1:a[0].find('.')]
 ID2 = a[0][a[0].find('.') + 1:a[0].find('.') + 3]
 a[0] = ID1 + ID2
 output = a[0] + '|' to a[39] + '\n'
 i += 1
```

In addition to the above formatting for the preprocessing part, we have formatted the data for the K-Means clustering too. Initially we have done the PCA and K-means clustering independently with the same source of the data but at the second step we have feed the data to the clustering process which is generated from PCA through SVD with lower dimension and maximum variance. We have set up the ID for each row as the country ID and labeled each row of data with a random cluster initially as a cluster initialization and randomly picked 3 centroids to initialize the K-Means clustering process. The formatted data has been saved into a text file for further processing and given as input the Hadoop MapReduce method for clustering.

```
I = 1
for line in lines:
 re_key1 = re.compile(' ')
 re_key5 = re.compile("\n")
 line = re_key5.sub('',line)
 a = re_key1.split(line)
 if i <= 9: a[0] = a[0][a[0].find('.') - 1:a[0].find('.')
     ]
 if i > 9 and i <= 99:
 ID1 = a[0][a[0].find('.') - 1:a[0].find('.')]
 ID2 = a[0][a[0].find('.') + 1:a[0].find('.') + 2]
 a[0] = ID1 + ID2
 #print(a[0])
 if i > 99 and i <= 190:
 ID1 = a[0][a[0].find('.') - 1:a[0].find('.')]
 ID2 = a[0][a[0].find('.') + 1:a[0].find('.') + 3]
 a[0] = ID1 + ID2
 output = a[0] + '|' + a[1][a[1].find('.') - 1:a[1].find(
     '.')] + '|' +
a[2] + ',' + a[3] + '\n'
 i += 1
```

3.3 PCA through SVD

3.3.1 Hadoop MapReduce. Now the data is ready for doing the PCA through SVD. We have implemented this with Hadoop MapReduce along with developing multiple functions to perform various steps to do the generalized eigen decomposition or singular value decomposition to reduce the dimensionality along with generating the reconstruction error and identified the best number of components with the minimum or benchmark reconstruction error. Here is the mapper function to read the data from text file:

```
def mapper_PCAbySVD(self, _, line):
 data_ID, features = line.split('|')
 feature_data = features.strip('\r\n')
 Features_arr = np.array(feature_data.split(','), dtype=
     float)
 row = Features_arr.tolist()
 yield None, (data_ID, row)
```

There are number of helper methods developed that are called from reducer method to perform the steps while doing the PCA through SVD. The following two method are developed identify the SVD to decompose the given main data matrix and to generate the matrix with a reduced dimension which is main objective of the singular value decomposition and this matrix will be used for the clustering algorithm. Here in the get_SVD method, we have decomposed the m x n matrix to the three singular vectors as the followings:

$$A_{[m*n]} = U_{[m*r]} \sum_{[r*r]} (V_{[n*r]})^T$$

where:
a. U, Left Singular Vector: m x r matrix
b. S, Singular values: r x r diagonal matrix
c. V, Right Singular vectors: n x r matrix

```
def get_SVD(self, cov_matrix):
 u, s, v = np.linalg.svd(cov_matrix, full_matrices=False)
 return u, s, v
def get_KComponents(self, u, x, k=2):
 z = np.dot(x, u[:, :k])
 u_reduced = u[:, :k]
 return z#, u_reduced
```

The following two methods are used to calculate the variance ratio or the reconstruction error:

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} \leq 0.01 \qquad \textbf{(1\%)}$$

The get_varianceRatio method calculates the reconstruction error using the above equation and return the variance ratio of the lower dimensioned vector. The other function iterates the process with increasing the dimension of the Right Singular Vector until the benchmark of the variance ratio is achieved. This reconstruction error or variance ration will be compared to the same that will be generated through the PCA algorithm using python ML library.

```
def find_BestK(self, initial, step, u, x):
 for k in range(initial, 30, step):
 z = self.get_KComponents(u, x, k)
 U_reduced = u[:, :k]
 ratio = self.get_VarianceRatio(z, U_reduced, x, k)
 if ratio <= 0.0019:
 break
 return k, ratio
def get_VarianceRatio(self, z, u, x, k):
 u = u[:, :k]
 X_approx_pca = np.dot(z, np.transpose(u))
 ratio = np.mean((x - X_approx_pca).T.dot(x -
     X_approx_pca)) /
np.mean(x.T.dot(x))
 return ratio
```

As the next step we have developed a Reducer method to get the keys and Values from teh mapper method and call the helper function to do the generalized eigen decomposition, generate a lower dimension matrix, identify the reconstruction error, and re-iterating the process the identify the best number of features that will give the most variance of the datapoint. Also saved the lower dimensional data in the text file along with Here is the reducer method. To get the singular values initially the covariance matrix is generated to make the original rectangular shaped matrix to a square matrix and then singular value decomposition is done with that covariance matrix.

```
def reducer_PCAbySVD(self, data_ID, values):
 dt = []
 for data_id, row in values:
 dt.append(row)
 val = []
 x = np.array(dt)
 np.savetxt('~/outpcaX.txt', x)
 #x = x.T
 np.savetxt('/~/outpcaXT.txt', x)
 cov_matrix = np.cov(x.T, bias=False)
 u, s, v = self.get_SVD(cov_matrix)
 k = 2
 z = self.get_KComponents(u, x, k)
 np.savetxt('~/outpcaZ.txt', z)
 clusterss = np.random.randint(1, 4, z.shape[0])
 ids = list(range(1, 1 + len(z)))
 ids = np.array(ids) # .to_numpy()
 ids = ids.tolist()
 lowerdim_data_z_with_ID_CID = np.vstack((clusterss, z.T)
     ).T
 lowerdim_data_z_with_ID_CID = np.vstack((ids,
lowerdim_data_z_with_ID_CID.T)).T
 np.savetxt('~/outpcaZWithIDCLID.txt',
     lowerdim_data_z_with_ID_CID)
 ratio = self.get_VarianceRatio(z, u, x, k)
 best_k, best_ratio = self.find_BestK(1, 1, u, x)
 val.append(ratio)
 val.append(best_k)
 val.append(best_ratio)
 np.savetxt('~/outpcaRatio.txt', val)
```

Once the decomposition is done, we have calculated Z by doing a. multiplication between the original matrix with the lower dimension U, left singular vector. This Z is the lower dimension matrix with the maximum variance which we will use for further processing. This Z can be used to do clustering and can develop any forecasting or predicting model for any of the practical problems. Here is the result of the reconstruction error calculation and best number of features and best variance:

```
Variance Ratio: 0.0010575742932206484
Best Number of Features: 2
Best Ratio: 0.0010575742932206484
```

3.4 PCA using python ML Library

The same that has been done in the above step has been done using the python machine learning library to check and compare the result. The following mapper method reads the data from the given file and passes the keys and values to the reducer method to compute the PCA and generate the lower dimensional matrix.

```
def mapper_builtinPCA(self, _, line):
 data_ID, features = line.split('|')
 feature_data = features.strip('\r\n')
 Features_arr = np.array(feature_data.split(','), dtype=
     float)
 row = Features_arr.tolist()
 yield None, (data_ID, row)
```

Here is the reducer method to calculate the PCA:

```
def reducer_builtinPCA(self, data_ID, values):
 dt = []
 for data_id, row in values:
 dt.append(row)
 val = []
 x = np.array(dt).T
 k = 2
 pca = PCA(n_components=k)
 z_pca = pca.fit_transform(x)
```

```
np.savetxt('~/KMTestData_by_outpcaZWithIDCLID.txt', z_pca
    )
 X_approx_pca = pca.inverse_transform(z_pca)
 ratio_pca = np.mean((x - X_approx_pca).T.dot(x -
    X_approx_pca)) /
np.mean(x.T.dot(x))
 val.append(ratio_pca)
 np.savetxt('/~/outpcaBuitinRatio.txt', val)
```

The reconstruction error or variance ratio is as below:

```
Variance ratio: 0.0003433886170064675376
```

3.5 K-Means Clustering

Along with the data formatting initially we have randomly assigned the cluster numbers for each row of the data and selected 3 centroids randomly to start the K-Means clustering process and saved the data into text files. At the beginning of the project, we have implemented the PCA and K-Means clustering algorithm independently and there was no integration in between these two implementations. At a later phase we have formatted the data which we got as the outcome of PCA through SVD for the clustering algorithm and have feed the data to the process using Hadoop MapReduce. We have used several helper methods and for the process along with one Mapper, Combiner and Reducer methods. Here is the mapper method that reads the data from input file:

```
def mapper_readataa(self, _, line):
    data_ID, Cluster_ID, Coordinate = line.split('|')
    Coordinate = Coordinate.strip('\r\n')
    Coordinate_array = np.array(Coordinate.split(','),
     dtype=float)
    global Centroid
    Centroid = self.get_centroids()
    Centroid_arr = np.reshape(Centroid, (-1, len(
     Coordinate_array)))
    global nclass
    nclass = Centroid_arr.shape[0]
    global ndim
    ndim = Centroid_arr.shape[1]
    Distance = ((Centroid_arr - Coordinate_array) ** 2).
     sum(axis=1)
    Cluster_ID = str(Distance.argmin() + 1)
    Coord_arr = Coordinate_array.tolist()
    yield Cluster_ID, (data_ID, Coord_arr)
```

Initially we have defined a method named get_centroid which reads the initial randomly picked Centroids from a file. These centroids are written in the same line in the file. Here we have used Numpy array to store the data except the when we passed the data from mapper to combiner and combiner to reducer. We have used list here instead to store the data and pass through from one method to another. In the mapper method we have read each of the rows of the file which was formatted earlier under preprocessing and formatting, and sliced the data in to parts as per the data id, cluster id and feature values. Then we read the centroids and calculated the distance. From the mapper methods we have output the cluster id as key, and data id and list of features as coordinate as values. We have used some global variables here so that other methods can get access to that variables along with convergence of the clustering process can be verified.

In the combiner method we have extracted the id and coordinates that we repassed from all machines by mapper method and the coordinate summations are calculated of the all the data from each machine. We have packed the values into a list and then passed to the reducer method for further calculating the distance between the centroid and data points and update the centroids. Here is the combiner method:

```
def combiner_nodecal(self, Cluster_ID, values):
    member = []
    Coordinate_set = []
    Coordinate_sum = np.zeros(ndim)
    for data_ID, Coordinate_array in values:
        Coordinate_set.append(','.join(str(e) for e in
     Coordinate_array))
        Coordinate_array = np.array(Coordinate_array,
     dtype=float)
        member.append(data_ID)
        Coordinate_sum += Coordinate_array
        Coordinate_sum = Coordinate_sum.tolist()
    yield Cluster_ID, (member, Coordinate_sum,
     Coordinate_set)
```

The reducer method works in all the machines and here in our case we have defined a method as reducer named update_centroid that actually update the centroids in a file upon calculating the distances between the data point and centroids. In the formatted data the data id, cluster id and data points are separated by '|'. Coordinates, and cluster members has been updated until the there is no update in the distance between the data points and calculated centroids. This whole process iterates based on the configuration argument in the argument configuration method. Here is code for the Reducer method:

```
def update_centroid(self, Cluster_ID, values):

    final_member = []
    final_Coord_set = []
    final_Coord_sum = np.zeros(ndim)

    f = open('~/KMclusterd_output.txt', 'a')

    for member, Coord_sum, Coord_set in values:
        final_Coord_set += Coord_set
        Coord_sum = np.array(Coord_sum, dtype=float)
        final_member += member
        final_Coord_sum += Coord_sum

    n = len(final_member)
    new_Centroid = final_Coord_sum / n
    Centroid[ndim * (int(Cluster_ID) - 1): ndim * int(
     Cluster_ID)] = new_Centroid
    if int(Cluster_ID) == nclass:
        self.write_centroids(Centroid)

    for ID in final_member:
        clustered_data1 = ID + ',' + Cluster_ID + ',' +
     final_Coord_set[ind] + ',' + "\n"

        f.write(clustered_data1)
        yield None, (ID + '|' + Cluster_ID + '|' +
     final_Coord_set[ind])

    f.close()
```

Multistep MapReduce codes are run by MRJob here in our code and in terms of the output of the mapper which has to be in the exact same format as the reduce steps are worked properly. In the argument configuration method, we have set up the number of iterations that the kmeans algorithm will run to achieve the optimum clustering.

```
def configure_args(self):
 super(MRKmeans, self).configure_args()
 self.add_file_arg('--infile')
 self.add_file_arg('--outfile')
 self.add_passthru_arg('-n', '--iterations', default=10,
    type=int,
help='number of iterations')
def steps(self):
 return [MRStep(mapper=self.mapper_readataa,
 combiner=self.combiner_nodecal,
 reducer=self.update_centroid)] * self.options.iterations
```

Here is the sample output of the clustered data where the 1st part is the ID or Country ID, 2nd portion is the cluster number where the country fall in and the rest coordinate or features were considered.

```
"153|3|-25587419.53101131,-13135782.51560188"
"163|3|-17602548.811211735,-181164.42030308917"
"183|3|-13039427.449006088,843442.5507865637"
"1|2|-32956168.235943224,-1411787.9917041105"
"2|2|-108990838.16919181,-2629810.726334124
"182|2|-45983684.67544772,-8385038.857376007"
"190|2|-1391453271.867793,-218879240.6876602"
"4|1|-10633440.04728168,-9674159.326911634"
"5|1|-3024137.3238387266,-1942046.0413679536"
```

Upon generating the centroids and completing the clustering we have drawn the plot for the clustering. Here is the code that has been developed for the plotting. We have loaded the centroids into a list and also the read the clustered data to get the coordinates. Then we have calculated the Euclidian distances from the centroids from the coordinates and created a plot for clustered data. Here is code for the plotting:

```
filepath = '/Users/ruhulislam/PycharmProjects/
    pythonProject/KMCentroid_plot.txt'
with open(filepath) as fp:
    line = fp.readline()
    while line:
        if line:
            line = line.strip()
            cord = line.split(',')
            centroids.append((float(cord[0]), float(cord
    [1])))
            Cent[0].append((float(cord[0])))
            Cent[1].append((float(cord[1])))
        else:
            break
        line = fp.readline()
fp.close()
list1 = []
filepath = '/Users/ruhulislam/PycharmProjects/
    pythonProject/KMclusterd_output_plot.txt'
with open(filepath) as fp:
    line = fp.readline()
    while line:
        if line:
            line = line.strip()
            list1 = line.split(',')
            labels = float(list1[1])
            x = (float(list1[2]), float(list1[3]))
            dist = 100000000000000
            selected_m = -1
            for m in centroids:
                test_distance = distance.euclidean(x, m)
                if test_distance < dist:
                    dist = test_distance
                    selected_m = centroids.index(m)
            X[selected_m][0].append(x[0])
            X[selected_m][1].append(x[1])
        else:
            break
        line = fp.readline()
fp.close()
plt.plot(X[0][0], X[0][1], 'ro')
plt.plot(X[1][0], X[1][1], 'go')
plt.plot(X[2][0], X[2][1], 'bo')
plt.plot(Cent[0], Cent[1], 'yo')  # centroids are yellow
    color
plt.show()
```

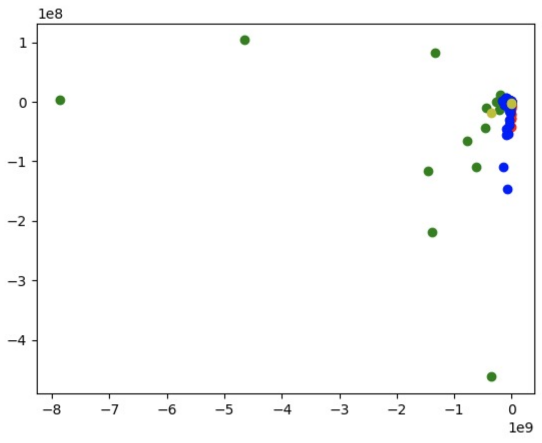

Figure 1: Figure: Clustered Data

3.6 Random Forest Regression

When implementing the Random Forest regression our goal was to split the dataset into two parts: one with data from year 2020 to be used as a training set for the regression model, and another, with fresh data from the current year 2021, as a testing set for predicting the number of new Coronavirus cases. We wanted to start with the Spark's *mllib* implementation of Random Forest Regression to get the predictions. After that, we would develop a Random Forest Regression model in MapReduce paradigm with Python's *mrjob*, and use it to predict numbers of new disease cases for the same dates in 2021 and then plot the resulting values against each other and the actual values for comparison.

3.6.1 Dataset preparation. The dataset we have created as a result of the main preprocessing step still requires some minor additional preparations for each specific algorithm. In case of Random Forest regression we wanted to remove columns with strongly correlated features, replace a few remaining NaN values with zeroes because of Random Forest's inability to deal with null values, and split the remaining dataset into testing and training subsets with data for year 2020 and 2021 respectively.

```
data = data[['location','date','total_deaths','new_deaths
    ','new_deaths_smoothed','total_cases_per_million',
'total_deaths_per_million','
    new_deaths_smoothed_per_million','reproduction_rate'
    ,'total_tests',
'stringency_index','population','population_density','
    median_age','aged_65_older','aged_70_older',
'gdp_per_capita','extreme_poverty','cardiovasc_death_rate
    ','diabetes_prevalence','female_smokers',
'male_smokers','hospital_beds_per_thousand','
    life_expectancy','human_development_index','
    new_tests',
'days_after_100_cases','internal_flights','
    international_arrivals','new_cases']].fillna(0)

training_data = data[data['date'].str.contains("2020")]
testing_data = data[data['date'].str.contains("2021")]
```

3.6.2 RF with MapReduce. First step was to express the possible algorithm structure in terms of map and reduce operations. Random Forest implies building multiple, preferably over thousand, randomised decision tree models. This part offers highest potential for scalability, because trees are being trained independently and can easily be grown simultaneously in distributed mode.

Tree-building and prediction routines look like a natural fit into the mapper stage. This way the predicted results from each individual tree would have been sent to the reducer stage, where it is easy to take an average of all predictions for each row of the training data and yield the final result.

However, the nature of Hadoop mapper requires it to read a file, usually line by line. It is crucial if the task needs some mapping operation to be applied to each row of the dataset. This was our first approach at designing the Random Forest algorithm. Mapper would read the full dataset row by row, apply the additional processing steps and yield the year value as a key and the whole cleaned row as a value. Reducer would collect the values into testing and training sets, generate the required number of subsamples equal to the number of decision trees to build, an then pass both datasets to the next step.

Next mapper would then train the tree models and predict values, and the next reducer would aggregate the predictions. It took approximately 20 minutes to make the final predictions in the containerized Hortonworks Sandbox solution, and slightly under 15 minutes on the cluster of one master and three worker nodes. Our measurements indicated that at least 3 minutes of this time was spent during the first step and streaming the intermediate results to the second step.

In attempt to improve the performance we have decided to use a raw mapper that reads the whole file directly, without addressing each individual line. But still the additional preprocessing would be applied in the mapper stage, this time using the pandas dataframes. Other steps were performed as before.

This approach did not show any significant improvement of the algorithm performance. Moreover, the first mapper-reducer step was beginning to look really unnecessary, as this simple preprocessing routine could be done externally before starting the algorithm, and bootstrap sampling could be easily done by the second mapper.

To avoid the unnecessary passing of the data between two mrjob steps we decided to try and fit the whole algorithm into a single step without losing any scalability. Now an external Python script would preprocess data, and another script would create a dummy file with requested number of lines corresponding to the number of decison trees.

```
if len(sys.argv) != 2:
    print("Usage: provide number of trees for a Random
     Forest")
else:
    trees = int(sys.argv[1])
    with open("dummy", "a") as dummy:
        for i in range(trees):
            dummy.write(str(i+1) + "\n")
```

The dummy file would then be copied to HDFS and read by the single mapper line by line. The testing and training datasets would be loaded during the mapper initialization step and converted to numpy's two-dimensional ndarrays.

```
def mapper_init(self):
        self.training = pd.read_csv("rf_training.csv").
    to_numpy()
        self.testing = pd.read_csv("rf_testing.csv").
    to_numpy()
```

For each line of the dummy file the mapper creates a new bootstrap sample, trains a tree on this sample, gets predictions for training data and passes them to the reducer stage.

```
def mapper(self, _, value):
    training = np.delete(self.training, obj = [0, 1, 2],
     axis = 1)
    records_total = training.shape[0]
    features_total = training.shape[1] - 1
    # number of features to consider at a new split
    sample_features = int(math.floor(math.sqrt(
     features_total)))
    # bootstrap sample records with replacement
    samples = np.random.choice(records_total,
     records_total)
    bootstrap_samples = training[samples, :]
    tree = RFTree(training = bootstrap_samples, testing =
      self.testing, depth = self.options.tree_depth,
     min_samples = self.options.min_samples,
     sample_features = sample_features)
    model = tree.train()
    predicted = tree.get_predictions(model)
    yield "results", predicted
```

Reducer, in its turn, aggregates the predicted data, takes the average of all predictions, adds the final result to the columns with date, country name, and actual value, and then writes the resulting array to a file in csv format. The format of the data:

$$[\textit{country name},\ \textit{date},\ \textit{actual value},\ \textit{predicted value}]$$

```
def reducer(self, key, values):
        combined_predictions = np.array(list(values))
        if combined_predictions.shape[1] == self.
    dates_locations.shape[0]:
            result = np.mean(combined_predictions, axis
    =0)
            result = np.atleast_2d(result).T

            with_date = np.append(self.dates_locations,
    np.array(result), 1)
            for row in with_date:
                yield None, "{},{},{},{}".format(row[0],
    row[1], row[2], int(row[3]))
```

This way we were able to recover at least 2-3 minutes of running time when running the algorithm with one thousand decision trees.

Moreover, we have achieved some additional performance gain by moving the Python code with the Random Forest class from a separate module which would need to be loaded to each node along with the training and testing sets, directly to the file containing mrjob steps.

The final architecture of the algorithm is presented on the figure.

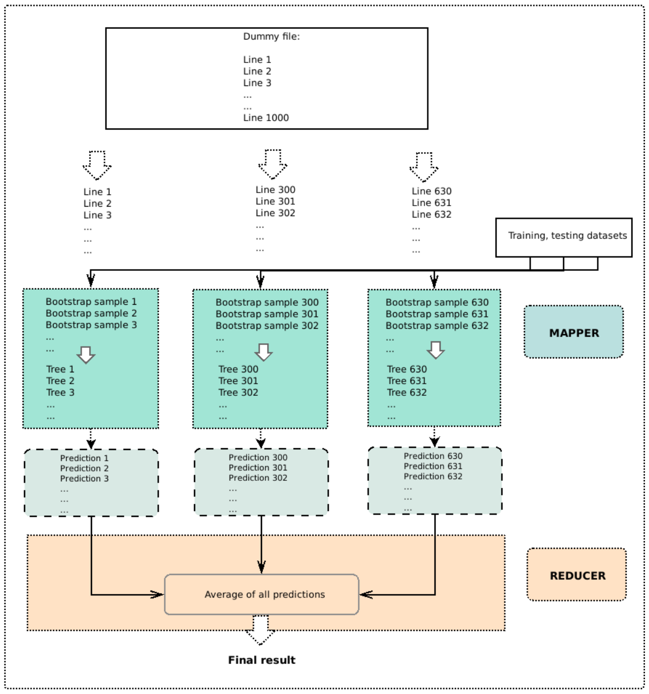

Figure 2: Single step design of the Random Forest regression algorithm

Parameters. There are three parameters that can be defined by the user when starting the job execution. *Number of trees* can be decided when running the *make_dummy.py* script. The desired number of decision trees to build must be provided as a command line argument to the script. Two other parameters can be added as options when starting the job execution.

– – *tree_depth* option allows to limit the depth of the decision trees. According to the authors of [12] and [10] limiting the depth of the trees or pruning the trees after they were allowed to fully grow can negatively affect the precision of the prediction, but can also improve the performance. The default depth value in our implementation is equal to 1000 which in practice allows the trees to grow unstopped, at least for a dataset of this size.

– – *min_samples* options allows to decide how many records should a node have in total to stop splitting and form a leaf node. The default value is set to 5, changing this value inside the range of recommended values (1-10) gives no visible effect on the performance of the algorithm.

3.6.3 *RF with Spark's mllib.*

MLlib is Spark's library offering a range of Machine Learning models. We use its implementation of the Random Forests to predict numbers of new cases for the same dates we have already predicted with the former algorithm.

```python
training = spark.read.csv("rf_training.csv", inferSchema=True, header=True)
testing = spark.read.csv("rf_testing.csv", inferSchema=True, header=True)
training = training.drop('_c0', 'location', 'date').fillna(0)
testing = testing.drop('_c0', 'location', 'date').fillna(0)

feature_cols = training.schema.names[:-1]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
tmp = assembler.transform(training)
training = tmp["features", "new_cases"]

t_assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
ttmp = t_assembler.transform(testing)
testing = ttmp["features", "new_cases"]

tree = RandomForestRegressor(labelCol="new_cases", featuresCol="features", numTrees=1000, impurity='variance')
#pipeline = Pipeline(stages=[t_assembler, tree])
model = tree.fit(training)
predictions = model.transform(testing)
```

The predicted values are returned in the same format as in the previous implementation. After that we have two sets of predictions ready for comparison.

3.6.4 *Random Forest Regression algorithm.*

The design of the Random Forest algorithm itself also posed quite a few challenges.

We implement a single Random Forest decision tree as a Python class with methods for training and predictions and a set of parameters such as tree depth, maximum number of records in a node to form a leaf node, and number of features to choose as candidates for a split condition.

```python
class RFTree:
def __init__(self, training, testing, depth, min_samples, sample_features):
    self.training_set = np.array(training)
    self.testing_set = np.array(testing)
    self.tree_depth = depth
    self.min_samples = min_samples
    self.sample_features = sample_features
```

A tree is growing recursively by identifying a subset of candidate variables for a split, evaluating the importance of each variable, and choosing the one that promises the lowest variance for each tree node. Unlike in a CART tree[6], which analyses the whole set of features for each new split, a Random Forest tree only selects a subset of all features.

The choice of the splitting rule can have a significant impact on the predictive abilities of the algorithm. An interesting alternative to selecting a random subset of features and choosing one giving the best split (resulting in highest gain in variance reduction) is to pick just one random feature without any evaluation and use the mean of its values as the split condition, mentioned in [12].

In our variant of the split rule we set the number of random features considered for each new split to the square root of the number of all features in the dataset. This is a standard approach that can be seen in most software implementations of Random Forests.

The splitting rule is applied inside the private method of the Random Forest tree class. At first it selects a predefined number of random features from all the features of the dataset ($\mathbf{m} = \sqrt{total\_features}$).

```python
def __split(self, tree_node_records):
    if isinstance(tree_node_records, list) or
       isinstance(tree_node_records, np.ndarray):
            # choose n features (without replacement)
            random_features = np.random.choice(np.arange
        (0, self.training_set.shape[1]-1), self.
        sample_features, replace=False)
```

Then, for each of m candidate features, we split the node data into two candidate children subsets according to the mean value of the current candidate variable. Each candidate split is saved into a dictionary to later be scored against other candidates.

```python
# split for each of the chosen features: dict candidates
    {i: branches}
candidates = {}
for i in random_features:
    condition = np.nanmean(np.array(tree_node_records)[:,
      i].astype('float'))
    left_branch = []
    right_branch = []
    for record in tree_node_records:
        if record[i] < condition:
            left_branch.append(record)
        else:
            right_branch.append(record)
    candidates[i] = list([left_branch, right_branch])
```

The most used method of selecting one feature among the candidates in Random Forest regressions is measuring the split's contribution to the overall variance reduction of the label data. In our implementation we use a simplified measure of pooled standard deviation of the label data inside the two children nodes that would form as a result of the current split. Then we pick a candidate split that will offer the best score, i.e, the lowest value of the pooled standard deviation.

```python
def __get_feature_importance(self, branches):
    # POOLED STANDARD DEVIATION
    if len(branches) == 2:
        sd1, sd2 = (0, 0)
        if len(branches[0]) > 0:
            left = np.array(branches[0])[:,-1]
            sd1 = np.std(left)
        if len(branches[1]) > 0:
            right = np.array(branches[1])[:,-1]
            sd2 = np.std(right)
        pooled = math.sqrt((sd1**2 + sd2**2)/2)
        return pooled
```

Another challenge is posed by the the "variables" that only vary in the context of multiple countries, but are constant for a country, like median age, population density and so on. After multiple splits the node will often contain data for only one country. If such a *constant* variable is chosen as a candidate for split, it must be ignored and not added to the candidate splits because it no longer offers any statistical significance for splitting.

```python
feature_list = list(np.array(tree_node_records)[:, i].
    astype('float'))
count = feature_list.count(condition)
if count == len(feature_list) or len(left_branch) == 0 or
      len(right_branch) == 0:
    pass
```

If none of the randomly selected features offers any significance for splitting of the node data, the node becomes a leaf node by taking the mean value of the label column.

```python
if len(candidates.items()) == 0:
    new_node = np.nanmean(np.array(tree_node_records)[:,
     -1].astype('float'))
    return new_node
```

Otherwise, the best candidate is chosen and a new internal node is created. Unlike the leaf node, whose type is just a float, an internal node is a dict storing the information about the chosen split and corresponding split conditions that later will be used for predictions.

```python
# compare importance of the features, choose one that
    gives lowest pooled sd of children node
sds = {}
for k , v in candidates.items():
    sds[k] = self.__get_feature_importance(v)
chosen_feature = min(sds.items(), key=lambda x: x[1])
new_node = {}
branches = {}
branches["left"] = candidates[chosen_feature[0]][0]
branches["right"] = candidates[chosen_feature[0]][1]
new_node["branches"] = branches
new_node["feature"] = chosen_feature[0] # index of the
    chosen feature
new_node["condition"] = np.nanmean(np.array(
    tree_node_records)[:, chosen_feature[0]].astype('
    float'))
return new_node
```

After that the algorithm creates two children nodes based on the selected split and repeats the same steps for each of the children nodes in a recursive manner.

```python
def __branch_out(self, tree_node, depths):
    if isinstance(tree_node, dict):
        for side, branch in tree_node["branches"].items()
    :
            if depths >= self.tree_depth or np.array(
    branch).shape[0] <= self.min_samples:
                # take mean value of results (y) for this
     leaf, converting to floats first
                branch = np.nanmean(np.array(branch)[:,
    -1].astype('float'))
                tree_node[side] = branch
            # else make a node and branch out
            else:
                branch = self.__split(branch)
                self.__branch_out(branch, depths+1)
                tree_node[side] = branch
```

To train a new tree we take the whole training dataset as a root node, perform the first split and branch out. After the recursive part is complete the root node is a complete Random Forest tree model ready to make predictions.

```python
def train(self):
    root = self.__split(self.training_set)
    self.__branch_out(root, 1)
    return root
```

A tree model predicts the label value for each single record in the training set.

```python
def get_predictions(self, tree):
    test = self.testing_set
    predicted_values = list()
    for row in test:
        datarow = row[3:-1]
        predicted = self.__predict(tree, datarow)
        predicted_values.append(predicted)
    return predicted_values
```

The prediction is made based on the split conditions of each single node of the decision tree.

```
def __predict(self, tree_node, datarow):
    if datarow[tree_node["feature"]] < tree_node["
     condition"]:
        if isinstance(tree_node["left"], dict):
            return self.__predict(tree_node["left"],
     datarow)
        else:
            return tree_node["left"]
    else:
        if isinstance(tree_node["right"], dict):
            return self.__predict(tree_node["right"],
     datarow)
        else:
            return tree_node["right"]
```

At the end each individual tree makes its predictions that are later sent to the reducer and aggregated to produce the final prediction.

4 RESULTS AND EVALUATION

In this chapter we want to evaluate the results of the developed algorithms applied to our dataset. We will discuss the possible shortcomings of our implementations and possible ways of improvement. We will also study the performance of the algorithms in terms of running time with respect to changes in the hardware configurations and algorithm parameters.

4.1 Cluster setup

We use two different setups while working on the project. The first one is running in pseudo-distributed mode inside a docker container with Sandbox by Hortonworks. We use this setup to develop and test the code on a smaller dataset samples to debug any possible issues.

The second setup is a network of 4 VMs, each with 4GB RAM. Four VMs are configured as a single cluster of one master node and three slave nodes. Hadoop 3.2.1 is installed on the cluster and we use Yarn as our resource manager. Spark version is 3.1.1. We also use Yarn as a cluster manager for Spark jobs.

We use the cluster to run debugged code on the full datasets to preprocess data, apply the algorithms, get results, and evaluate the performance.

4.2 Performance evaluation of the PCA

In the above analysis, we found that the variance ratio while doing PCA through SVD is as below which is slightly differ from the variance rations calculated by using machine learning. This doesn't mean that PCA through SVD is unacceptable rather this is also acceptable because the variance ratio is less than 0.01 or 1 %. Thus, we used the Z matrix which is constructed through multiplying the original matrix with the left singular vector which is generated through SVD. Results from SVD:

```
Variance Ratio: 0.0010575742932206484
Best Number of Features: 2
Best Ratio: 0.0010575742932206484
```

Results from python ML PCA:

```
Variance ratio: 0.0003433886170064675376
```

4.3 Random Forest regression

Our goal was to apply the Random Forest regression algorithm developed by us in MapReduce paradigm to the COVID-19 dataset to attempt predicting daily numbers of new cases of infection by the disease and compare the results with the values predicted by the built-in Random Forest regression method of the Spark's mllib library.

As a final check, we made predictions on the same dataset with the Python's most known Machine Learning library *sklearn* and its implementation of the Random Forest regression and added the resulting predictions to the comparison.

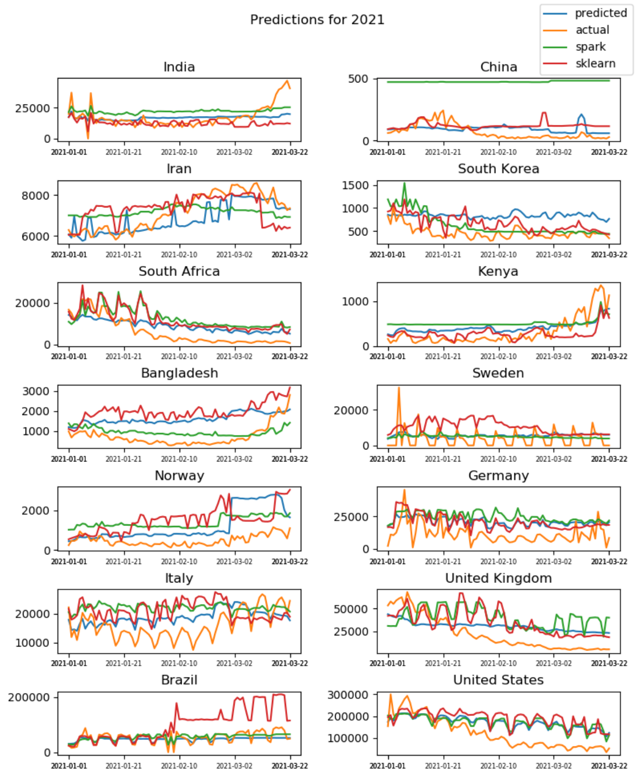

Figure 3: Comparison of the numbers of new cases of COVID-19 predicted by our algorithm, Spark's mllib and Python's sklearn with the actual numbers

Here we can see that the predictions returned by our algorithm are far from the real numbers. However, the predictions made by both sklearn and mllib are not extremely precise either, at least for most countries. It is possible that among the factors negatively affecting the precision are not just the flaws of our implementation of the Random Forest regression, but also some specifics of the dataset.

To analyze the scalability and performance of the MapReduce based implementation we ran the algorithm on different hardware setups and with different parameters. To compare the scalability we altered the configuration of our cluster from three to just one worker node. Additionally we ran the algorithm for different number of decision trees and with pruned and fully grown trees. Here we want to present the graph with the results of the performance test.

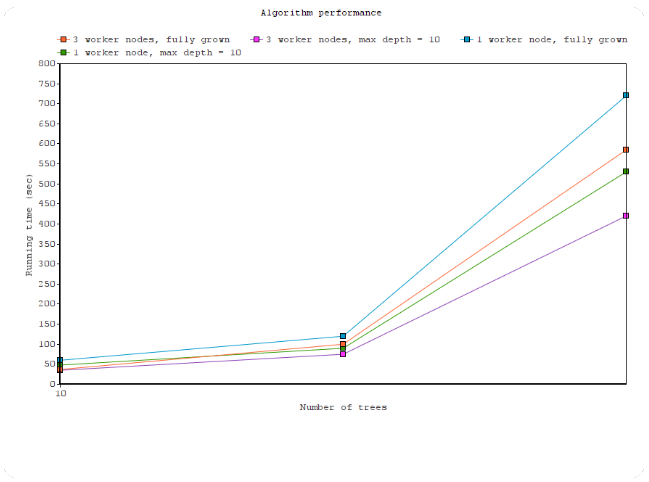

Figure 4: Algorithm performance depending on changes to number of worker nodes, number of decision trees and limiting tree depth

For better overview we also present the same data in form of a table.

Setup	Tree depth	10 trees	100 trees	1000 trees
3 workers	unlimited	37	100	585
	10	35	75	420
1 worker	unlimited	60	120	720
	10	48	90	530

Table 1: Running time (sec) depending on changes to the setup and parameters

From the results we can see that both tree depth and number of three parameters affect the performance of the algorithm. The running time of the algorithm is also noticeably scaling with the number of slave nodes.

However, it seems that the total number of trees is the factor that affects execution time of the algorithm the most.

5 CONCLUSION

In this project we have developed several Machine Learning algorithms using MapReduce paradigm. The algorithms we chose are PCA through SVD,K-Means clustering, and Random Forest regression. We applied the algorithms to the collection of data connected to the different aspects of the Coronavirus pandemic, using an Apache Hadoop cluster as running environment.

We have analyzed the results of the implemented algorithms, compared them with the results produced on the same dataset by well established and optimized libraries such as Spark's *mllib* and Python's *sklearn*. We present the comparison of the prediction results by plotting them against each other on a set of graphs. We have found that the predictions made by our regression algorithm are not very precise, but comparable to those predicted by other libraries.

We have also studied the performance of the algorithms by comparing the execution times on different hardware setups and with varying parameters to analyze how well the algorithms are adjusted to running in distributed mode. The results demonstrate that at least the performance of the Random Forest regression algorithm does indeed scale with the number of worker nodes.

From future enhancement perspective of clustering, other parallel algorithms will be implemented and compared the accuracy and performance among them to identify the best one along with working on the enhancement of any these core algorithms from big data perspective.

Also, for future work, there are still many opportunities to improve the Random Forest model accuracy by improving the splitting rules of the regression or adding more data points to the dataset, and also achieve better distributed performance by optimizing different aspects of the cluster configuration.

REFERENCES

[1] [n. d.]. Dimensionality Reduction using Factor Analysis in Python! https://www.analyticsvidhya.com/blog/2020/10/dimensionality-reduction-using-factor-analysis-in-python/
[2] [n. d.]. Singular Value Decomposition (SVD) tutorial. https://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm
[3] The OpenSky Network 2020. 2021. Crowdsourced air traffic data. https://zenodo.org/record/4601479#.YIsqPXVfiHs
[4] Kirk Baker. 2005. Singular value decomposition tutorial. *The Ohio State University* 2005 (2005), 24. https://doi.org/10.1021/jo0008901
[5] Leo Breiman. 2001. Machine Learning, Volume 45, Number 1 - SpringerLink. *Machine Learning* 45 (10 2001), 5–32. https://doi.org/10.1023/A:1010933404324
[6] L Breiman, JH Friedman, RA Olshen, and CJ Stone. 1984. CART. *Classification and Regression Trees, Wadsworth and Brooks/Cole, Monterey, CA* (1984).
[7] Suman Chakraborti, Arabinda Maiti, Suvamoy Pramanik, Srikanta Sannigrahi, Francesco Pilla, Anushna Banerjee, and Dipendra Nath Das. 2021. Evaluating the plausible application of advanced machine learnings in exploring determinant factors of present pandemic: A case for continent specific COVID-19 analysis. *Science of The Total Environment* 765 (2021), 142723. https://doi.org/10.1016/j.scitotenv.2020.142723
[8] DataGraphics. 2021. Dataset collection. https://datagraphics.dckube.scilifelab.se/api/dataset/bbbaf64a25a1452287a8630503f07418.csv
[9] Shu-Kai Fan, Chuan-Jun Su, Han-Tang Nien, Pei-Fang Tsai, and Chen-Yang Cheng. 2018. Using machine learning and big data approaches to predict travel time based on historical and real-time data from Taiwan electronic toll collection. *Soft Computing* 22 (09 2018). https://doi.org/10.1007/s00500-017-2610-y
[10] Ulrike Grömping. 2009. Variable Importance Assessment in Regression: Linear Regression versus Random Forest. *The American Statistician* 63 (11 2009), 308–319. https://doi.org/10.1198/tast.2009.08199
[11] Our World in Data. 2021. Data on COVID-19. https://github.com/owid/covid-19-data/tree/master/public/data
[12] Hemant Ishwaran. 2014. The Effect of Splitting on Random Forests. *Machine Learning* 99 (04 2014). https://doi.org/10.1007/s10994-014-5451-2
[13] OurAirports. 2021. Open data downloads. https://ourairports.com/data/
[14] Tomasz Wiktorski. 2019. *Data-intensive Systems: Principles and Fundamentals using Hadoop and Spark.* https://doi.org/10.1007/978-3-030-04603-3_2
[15] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems* 2, 1-3 (aug 1987), 37–52. https://doi.org/10.1016/0169-7439(87)80084-9
[16] Weizhong Zhao and Qing He. 2009. Parallel K -Means Clustering Based on. *Springer-Verlag Berlin Heidelberg* (2009), 674–679.