VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Digit Classification on Large Dataset using Logistic Regression on Hadoop

Final Report

COURSE CODE- ITE2013
COURSE NAME- BIG DATA ANALYTICS
FACULTY- PROF. RANICHANDRA C
SLOT- E2+TE2

SUBMITTED BY:
NAME- KRISHNA KUMAR MAHTO
REGISTRATION NUMBER- 16BIT0453

- **Phase 1- Discovery Phase**

a. Problem Statement

Logistic regression works fine on a single machine, but on a relatively small dataset. With large datasets, even the sophisticated Logistic Regression libraries, for example LogisticRegression class provided in SciKit Learn Python library, takes a very long time to converge on large training datasets with more than 10,000 examples and over 700 features. So the goal of the project is to implement Logistic Regression on a Distributed File System using HDFS and logically improve the speed of convergence.

b. Modules

Module 1: Computation of objective function, (the cost function) and its gradient

On a single machine, training logistic regression on large dataset appeared to never complete, while another algorithm called Random Forest finished training within a few seconds. However, reducing the number of training examples did work and Logistic Regression converged, but it still took some time, and also the training set had to be reduced to nearly (1/10)th of the original size. I tried training the *MNIST dataset for Digit Recognition* (the same dataset which will be used for this project) which has a training set of size 42,000x768 on my personal laptop which runs on intel core i5 7200 and 8 GB of DDR4 RAM. However, even after 10 minutes, the classifier did not converge. The reason for this maybe either the cost function is taking too long to be computed, or the gradient of the cost is taking it too long to converge. So, the objective is to allow each machine in the cluster to calculate an objective function on its own section of the dataset along with the corresponding gradient of the cost function, and then produce an aggregate cost for the entire dataset and also the consequent aggregate using the gradients computed by each machine in the cluster.

Module 2: Optimisation of objective function

Optimisation can be done either with a code written from scratch, or by using any sophisticated function already available in different libraries. The optimisation is done by first finding the cost for the dataset and then finding its gradient. This should be

successfully obtained from module 1, after which an optimisation function can use the results of 2nd module to calculate the parameters of Logistic Regression that optimise the cost for a minimum. With this, the hypothesis of the Logistic Regression model is finally learnt as at the end of both the modules, the Logistic Regression model will have fitted the dataset.
The algorithm that can be used to optimise the parameters is Stochastic Gradient descent which is considered the best for Big Data. However, other avenues such as Gradient Descent can also be explored.

- **Phase 2- Data Preparation**

c. Dataset

Dataset selected: MNIST Dataset for Digit Recognition
Training set:
Number of examples = 42,000
Number of features = 784
Url: https://www.kaggle.com/c/digit-recognizer/download/train.csv

Test set:
Number of examples = 28, 000
Number of features = 784
Url: https://www.kaggle.com/c/digit-recognizer/download/test.csv

d. S/W and H/W Requirements

Softwares:

1. Apache Hadoop DFS for storage

Hadoop Distributed File System will be used to store data on a cluster of machines.

2. Apache Spark using Python/R for analysis

For in-place analysis of the dataset (training and testing the model), Spark will be used. Spark can work directly on top of HDFS. The MapReduce that comes with HADOOP has been found to be slow for Machine Learning jobs. Spark ML library is found to be faster.

Hardware:

A cluster of appropriate number of machines that can work parallely on the dataset to train the Logistic Regression model. The number of nodes in the cluster would depend on the size of the dataset.

Phase 3: Model Planning:

Module 1: Computation of objective function, (the cost function) and its gradient

Logistic regression is a classification model. With the given dataset, it sets to fit a function that can make predictions about a data point belonging to one class or the other the best way possible. The function that is intended to be fitted is often called as a hypothesis. This hypothesis may be one of many suitable functions, however, sigmoid function is used more often, which instead of directly giving y = 0 or y = 1' as the predicted class, computes a probability of how likely is it that y = 1. We then, based on the context or the business problem can select an appropriate threshold below which y = 0 and beyond which y = 1.

The sigmoid function shall be used for this project.

The sigmoid function (fig 1):

$$h_\Theta(x) = g(\Theta^T x)$$

where,

$$g(z) = 1/(1+e^{-z}) \quad \text{(fig 1)}$$

and Θ is the parameter vector, or the weights for the feature values.

Logistic regression can also be conveniently interpreted as a function that carves out a separation between the data points of two classes. This separation is called as decision boundary.

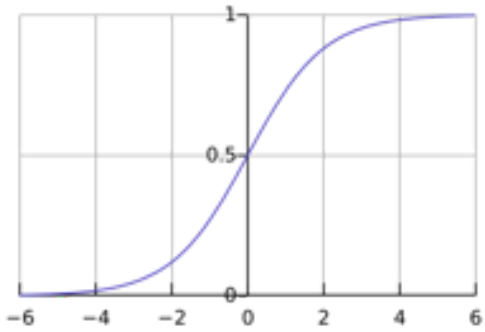

Fig. 1- Sigmoid function

The most common threshold that can be selected is g(z) = 0.5, as it explicitly makes out an intuitive definition of the decision boundary for any classification problem. From fig. 1, it is clear that any z>0, given a g(z) value greater than 0.5- this forms the decision boundary for a given problem. For classification problems, this z will be weighted sum of the attribute values taken from each example of the dataset. Logistic regression , although is seen as a model that associates with a linear decision boundary (fig-2), it can train non-linear decision boundaries as well.

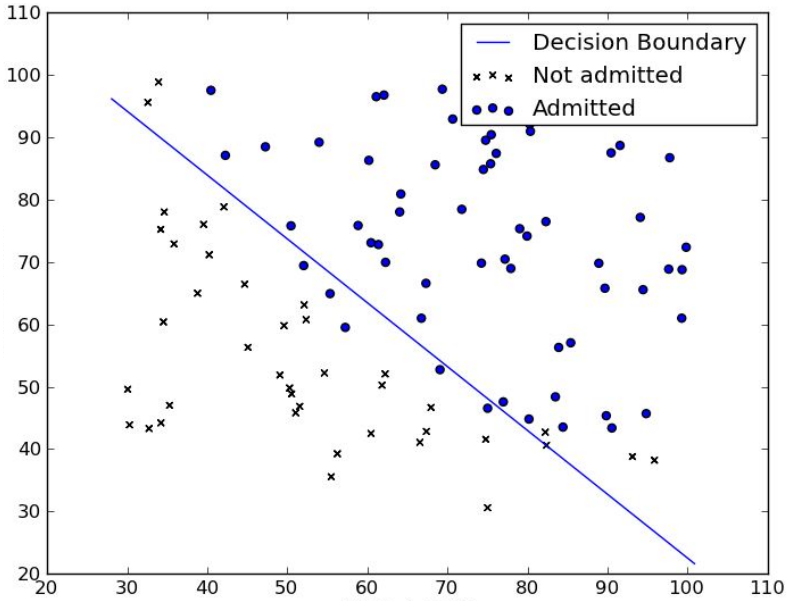

Fig. 2- Decision boundary

The objective function:

The function that we generally aim to minimize is the mean squared error function calculated on the training dataset. The cost function, J(Θ) is given as

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

where m = number of training examples

But for the logistic regression models, we tend to using the following cost function, which uses logarithms.

$$J(\theta) = -\frac{1}{m} [\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

We intend to fit the model to the training dataset by computing the values of the parameters Θ so that the objective function is minimized.
Nevertheless, using the above formula, we can compute the cost function, which also is the objective function for our case.

<u>One-vs-all classification:</u>

The problem at hand has 10 classes (since there are 10 digits from 0 to 9, inclusive). The idea of a binary classification (discerning two classes) can be conveniently extended to multi-class classification using an idea called One-vs-all classification. The idea is very simple, yet very powerful. In this method, the probability of that y = i (ie, sigmoid(X) > 0.5 for that class), for i^{th} class is calculated against all the other classes which are considered different from the i^{th} class, but of the same entity type with respect to each other.

Regularization:

Overfitting is one of the most common things that you would not want your model to get troubled by. Overfitting is the situation when a model becomes well-trained, but not well-generalized, i.e., the model may fit the training dataset very well, but may fail to generalize for the unseen examples. In such cases, we resort to introducing a regularization term that may counteract the effects of overfitting. In this project, regularization may be needed if the performance on the test set is not good.

Regularized cost function:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2.$$

Module 2: Optimisation of Objective Function

The gradient descent algorithm will be used to optimize the cost function.

Gradient descent is perhaps the most widely used optimisation algorithm in Machine Learning. As the name suggests, it optimizes the function by going down the slope of the graph of the function so as to achieve a local minimum and if possible, a global minimum.

Our cost function is:

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

We aim to minimize this objective function:

$$\min_\theta J(\theta)$$

Gradient Descent Algorithm:

1. Start with some randomly initialised values for Θ vector.
2. Repeat for a given number of steps or until a minimum is reached {
   i. Find $temp_j := \Theta_j - \alpha\, \partial J(\Theta)/\partial\Theta_j$ for each j.
   ii. Update $\Theta_j := temp_j$

}

Gradient descent algorithm is based on the idea of gradient, ie, the derivative of the function of maximum rate of increase. The cost function is taken as a function of the weight vector Θ. At every iteration of the above algorithm, the parameters Θ are either incremented or decremented (fig. 3b) based on its initialised or the current values and the sign of the gradient at the current point (fig. 3a).

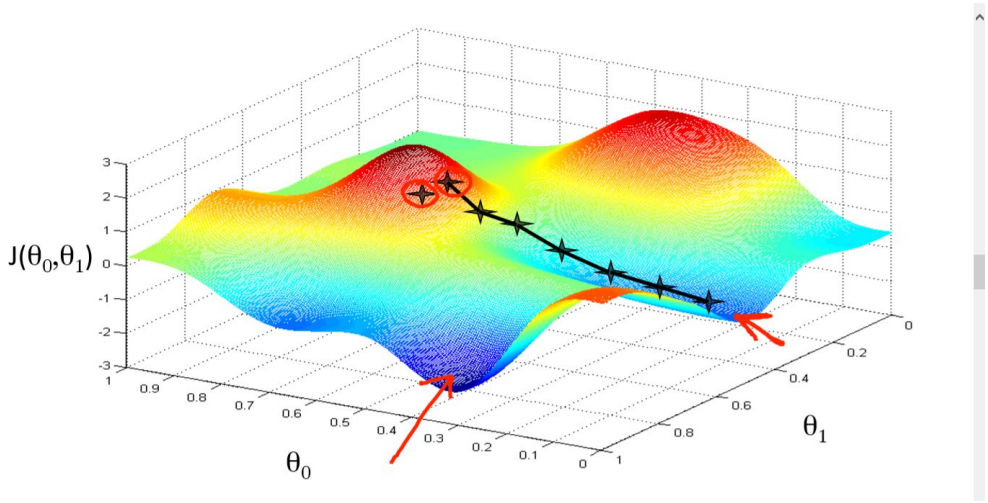

Fig. 3a- Gradient descent to minimize cost function J wrt $\Theta = (\Theta_0, \Theta_1)$

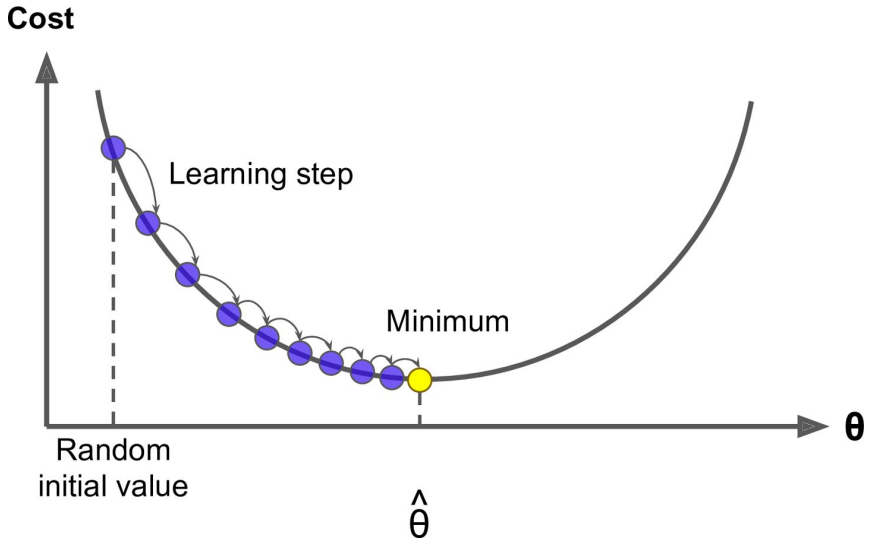

Fig.- 3b- Descents in gradient descent

At the end of the gradient descent algorithm, we aspire to get the values of Θ for which the cost function is the least. This would directly mean that the model has achieved a best fit for the training set.

Effect of regularization on gradient descent:
Since the cost function definition gets changed slightly by the introduction of regularization term, the gradient of the regularized cost function looks like,

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \quad \text{for } j \geq 1$$

This formulation of the gradient descent algorithm will be used in order to compute the update-values for the parameters.

Model to be applied:

Logistic Regression Model

In statistics, the logistic model (or logit model) is a widely used statistical model that, in its basic form, uses a logistic function to model a binary dependent variable; many more complex extensions exist. In regression analysis, logistic regression (or logit regression) is estimating the parameters of a logistic model; it is a form of binomial regression which corresponds to binary classification (fig 4). Mathematically, a binary logistic model has a dependent variable with two possible values, such as pass/fail, win/lose, alive/dead or healthy/sick; these are represented by an indicator variable, where the two values are labeled "0" and "1". In the logistic model, the log-odds (the logarithm of the odds) for the value labeled "1" is a linear combination of one or more independent variables ("predictors"); the independent variables can each be a binary variable (two classes, coded by an indicator variable) or a continuous variable (any real value). The corresponding probability of the value labeled "1" can vary between 0 (certainly the value "0") and 1 (certainly the value "1"), hence the labeling; the function that converts log-odds to probability is the logistic function, hence the name.

The binary logistic regression model has extensions to more than two levels of the dependent variable: categorical outputs with more than two values are modelled by multinomial logistic regression which corresponds to a solution of a multi-class classification problem (fig- 4 and fig- 5). This is often achieved by the idea of one-vs-all classification. The model itself simply models probability of output in terms of input, and does not perform statistical classification (it is not a classifier), though it can be used to make a classifier, for instance by choosing a cutoff value and classifying inputs with probability greater than the cutoff as one class, below the cutoff as the other; this is a common way to make a binary classifier.
Some other key discussion about logistic regression, including the objective function,the method employed to optimize and decision boundary, have already been discussed in the previous section during the discussion of module 1.

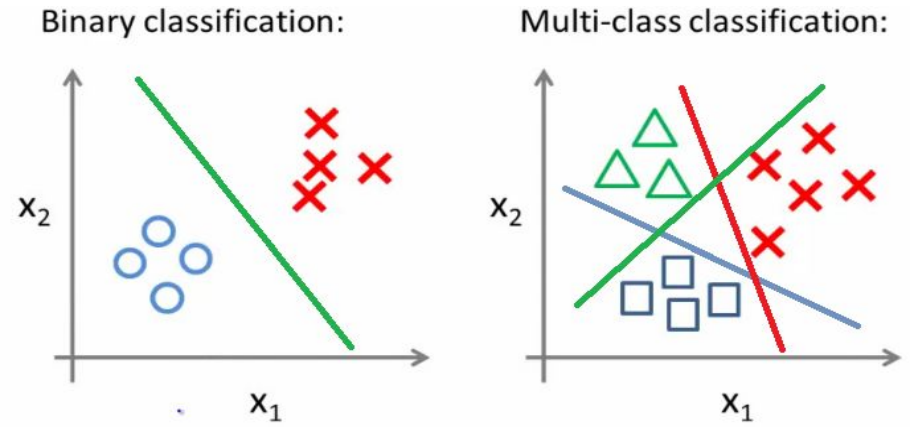

Fig.- 4- Binary and multi-class classification

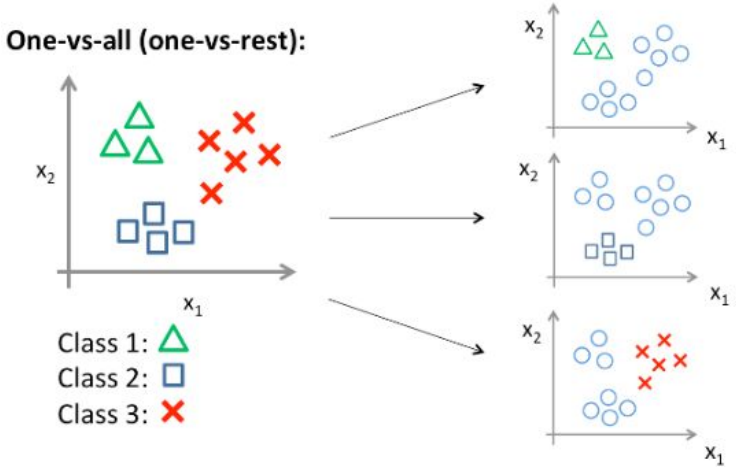

Fig. - 5- One-vs-all multiclass classification

Pseudo code:

Below is a generalised pseudo code for logistic regression.

```
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

#Predicting the test set result
y_pred = classifier.predict(X_test)

#Making the confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

On Hadoop and Apache Spark, the algorithm needs to be made more specific. The train and test data will be moved to the HDFS prior to working on the implementation of the algorithm. For the project, LogisticRegression() function will be attempted to be implemented from scratch. From what we already discussed in the previous sections, the ingredients to make a programmed version of this model are quite apparent. Consequently, the fit(), pred() and confusion_matrix() methods will also have to be implemented from scratch. Confusion matrix is a statistical metric which is generally applied to test the performance of classification algorithms. The Spark API for Python (PySpark) has been decided to be the preferred framework to work with.

Phase 4: Model Building

- **Method, library functions and technology used:**
    - Implementation without HDFS/Spark on local machine

The ultimate aim of doing this project comes down to seeing how HDFS, alongwith supporting functional environments can achieve what system without HDFS and related processing tools cannot. The model developed is a Logistic Regression model, one written in Python from scratch and the other written using PySpark. The first code will be

run on a local system in an attempt to train the MNIST handwritten dataset, and the second code will be run on the same dataset, stored on a single node HDFS.

- Library functions and methods used to accomplish the Python code shown in code snippet 1:

The following packages were used to handle the data efficiently on the local machine file system:

I. Pandas
II. Numpy
III. Scipy

Pandas is a python package to handle the csv files efficiently. In this implementation code, it was used to read the csv files (train and test sets). The csv files are read as Pandas DataFrame objects.
In order to handle the dataset and perform mathematical operation on it, Numpy is used extensively. *Numpy* stores the dataset as Numpy Array objects. These objects may be one-dimensional or two-dimensional based on the number of columns in the original dataset. *Scipy* package contains many built in functions to perform scientific calculations. This package was specifically used to optimize the objective function of this problem.

- Technologies used:

I. Anaconda
II. Jupyter notebook

Anaconda is much like an augmented version of Python. Alongwith all the libraries and features that a normal Python distribution comes with, it has many other libraries that are most used in data science. Packages such as numpy, pandas, etc. which were otherwise had to be installed manually, come already installed with the necessary dependencies.
Jupyter notebook is a web-based version of IPython (Interactive Python). This provides a convenient way to write the code, analyse outputs and maintain.

- Library functions used to accomplish the Python code shown in code snippet 2:

The second implementation of logistic regression was done using the scikit-learn library class LogisticRegression, along with the packages that were used in code snippet-1. Scikit-learn functions are optimised to use the hardware in the best way possible. The performance of this code is used as a reference to measure how the other two algorithms perform.

- **Python code written for local system:**

```
import pandas as pd
import numpy as np

from sklearn.linear_model import LogisticRegression
from scipy.optimize import minimize

data = pd.read_csv('dataset/train.csv')
print(data.columns)

X = data.iloc[:, 1:].values
y = data['label'].values

m = X.shape[0]
n = X.shape[1]
ones = np.ones(shape = (m, 1))

# Adding bias to the training dataset
X_biased = np.concatenate([ones, X], axis = 1)
print(m, n)

def sigmoid(z):
    return(1 / (1 + np.exp(-z)))

def lrcostFunctionReg(theta, reg, X, y):
    m = y.size
    h = sigmoid(X.dot(theta))
```

```
    J = -1*(1/m)*(np.log(h).T.dot(y)+np.log(1-h).T.dot(1-y)) +
(reg/(2*m))*np.sum(np.square(theta[1:]))

    if np.isnan(J[0]):
        return(np.inf)
    return(J[0])

def lrgradientReg(theta, reg, X,y):
    m = y.size
    h = sigmoid(X.dot(theta.reshape(-1,1)))

    grad = (1/m)*X.T.dot(h-y) +
(reg/m)*np.r_[[[0]],theta[1:].reshape(-1,1)]

    return(grad.flatten())


def oneVsAll(features, classes, n_labels, reg):
    initial_theta = np.zeros((X.shape[1],1))  # 401x1
    all_theta = np.zeros((n_labels, X.shape[1])) #10x401

    for c in np.arange(1, n_labels+1):
        res = minimize(lrcostFunctionReg, initial_theta,
args=(reg, features, (classes == c)*1), method=None,
                       jac=lrgradientReg,
options={'maxiter':50})
        all_theta[c-1] = res.x
    return(all_theta)

theta = oneVsAll(X_biased, y, 10, 0.1)

def predictOneVsAll(all_theta, features):
    probs = sigmoid(X.dot(all_theta.T))

    # Adding one because Python uses zero based indexing for the
```

```
10 columns (0-9),
    # while the 10 classes are numbered from 1 to 10.
    return(np.argmax(probs, axis=1)+1)

pred = predictOneVsAll(theta, X)
print('Training set accuracy: {} %'.format(np.mean(pred ==
y.ravel())*100))
```

Code snippet 1: Logistic Regression from scratch

- Python code using scikit-learn

```
clf = LogisticRegression(C=10, penalty='l2', solver='liblinear')
# Scikit-learn fits intercept automatically, so we exclude first
column with 'ones' from X when fitting.
print(X.shape, y.shape)
clf.fit(X, y)

pred2 = clf.predict(X[:,1:])
print('Training set accuracy: {} %'.format(np.mean(pred2 ==
y.ravel())*100))
```

Code snippet 2: Logistic Regression using scikit-learn library class

- Implementation with HDFS and Spark on single node Hadoop Cluster:

Following packages were used to implement the code on HDFS:

I. PySpark
II. MLlib (PySpark package)

MapReduce is not very efficient when it comes to processing large amounts of data because it does everything on the disk which is the slowest storage device on any system. The strongest feature of Spark is its in-memory computation, i.e, it brings data onto the memory for processing. If hardware limits the amount of data that can be brought processed on the main memory, then it takes a part of the data that can be worked upon in the RAM.
Spark can work on top of Apache YARN cluster manager (figure - 6). Spark is a single package that can handle a wide range of works- batch processing, machine learning, streaming data processing etc. This was not natively supported in vanilla Hadoop, for eg- in order to process structured data we had to install HIVE on top of Hadoop. Similarly, ML jobs require Mahout installation. This implies that all the components of Spark work in coherence, and are optimised to give the best performance possible.

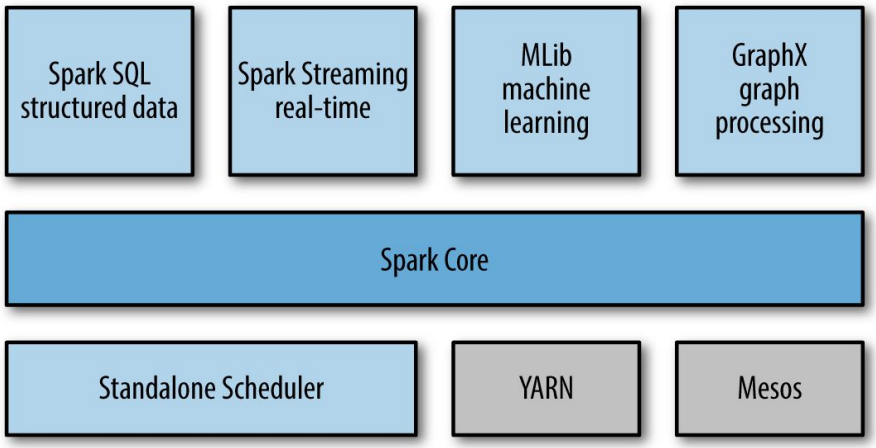

Fig- 6: Spark Components (unified into Spark)

- Spark Operations:

Resilient Distributed Datasets (RDDs) are what provide Spark the ability to do what traditional MapReduce cannot. Features of RDDs:

I. Fault tolerant distributed datasets

II. Lazy evaluation (forms a pipeline of transformations on RDDs, and does no action until an action is explicitly issued; these pipelines are conceptually represented as Directed Acyclic Graphs (DAGs).

III. Caching (frequently used data is cached)

IV. In-memory computation

V. Immutability (no transformation affects the original RDD, each transformation forms a new RDD).

VI. Partitioning (each RDD is stored in partitions, each partition is assigned to a task)

Figure- 7 shows the two operations in Spark- Transformation and Action. Note that both are performed on RDDs.

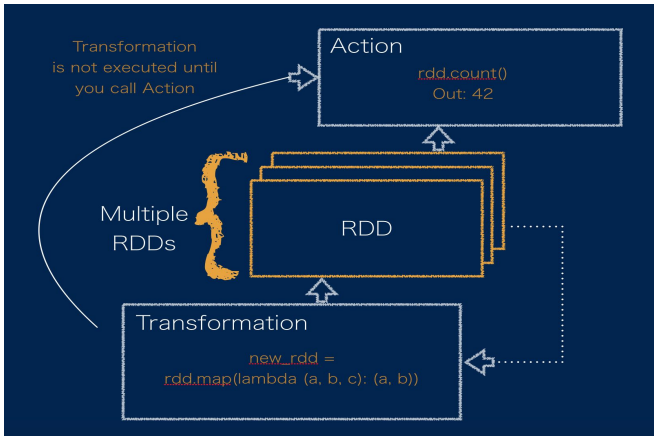

Fig- 7: Spark operations

MLlib is a Spark module, which comes with it by default. So it leverages all the advantages of RDD datasets in order to train any ML model. The library functions built into MLlib are optimised to use Spark's features along with the hardware in the best way possible.

Following (code snippet- 3) is the implementation of logistic regression using PySpark MLlib.

```
from pyspark.sql import SparkSession

spark =
SparkSession.builder.appName('logisticsRegression').getOrCreate(
)

from pyspark.ml.classification import LogisticRegression
```

```
import time

start_loading_data_time = time.time()

training_set =
spark.read.csv('hdfs://localhost:9000/BigDataProject/dataset/tra
in.csv', header = True, inferSchema = True)

print('Data loading time: ' + str(time.time() -
start_loading_data_time))


# Training

# Transforming the dataset- creating an array for each row in
the dataset. Each such array will be assigned to a new column
names 'features'

from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler


feature_columns = training_set.columns[1:]
label_column = training_set.columns[0]
# print(feature_columns)
# print(label_column)

print(label_column)

vector_assembler = VectorAssembler(inputCols = feature_columns,
outputCol = 'features')


final_train_set = vector_assembler.transform(training_set)
```

```
final_train_set.count()

logisticRegressor = LogisticRegression(featuresCol = 'features',
labelCol = 'label')

start_training_time = time.time()


fitted_logisticRegressor =
logisticRegressor.fit(final_train_set)

print('training time: ' + str(time.time() -
start_training_time))


test_data =
spark.read.csv('hdfs://localhost:9000/BigDataProject/dataset/tes
t.csv', header = True, inferSchema = True)


# test_data.dtypes


# #### Transforming the test dataset in the same way as we did
for the training set

# test_data.columns

feature_columns = test_data.columns[1:]
label_column = test_data.columns[0]


# feature_columns

vector_assembler_2 = VectorAssembler(inputCols =
```

```
feature_columns, outputCol= 'features')


final_test_data = vector_assembler_2.transform(test_data)

final_test_data.select("features").show()


prediction_and_labels =
fitted_logisticRegressor.evaluate(final_test_data)

prediction_and_labels.predictions.select(['label','prediction'])
.show()

type(prediction_and_labels.predictions.select('label'))

list_of_labels =
prediction_and_labels.predictions.select('label').collect()

num_rows_in_list =
prediction_and_labels.predictions.select('label').count()
num_rows_in_list

list_of_labels = [list_of_labels[i].label for i in
range(num_rows_in_list)]

# list_of_prediction_and_labels


list_of_predictions =
prediction_and_labels.predictions.select('prediction').collect()
list_of_predictions = [int(list_of_predictions[i].prediction)
for i in range(num_rows_in_list)]

# list_of_predictions
```

```
list_of_labels_prediction_pairs = zip(list_of_labels,
list_of_predictions)


list_of_labels_prediction_pairs =
list(list_of_labels_prediction_pairs)

# list_of_labels_prediction_pairs


# Evaluating the classifer

from pyspark.mllib.evaluation import MulticlassMetrics

predictions_and_labels_parallelized =
sc.parallelize(list_of_labels_prediction_pairs)

metrics = MulticlassMetrics(predictions_and_labels_parallelized)

correct_predictions = 0;

for i in range(num_rows_in_list):
    if (list_of_labels_prediction_pairs[i][0] ==
list_of_labels_prediction_pairs[i][1]):
        correct_predictions += 1

print(correct_predictions)

testset_num_rows = num_rows_in_list
test_accuracy = correct_predictions/testset_num_rows
print(test_accuracy)

import pandas as pd
```

```
labels_pd = pd.DataFrame(list_of_labels)
predictions_pd = pd.DataFrame(list_of_predictions)

from sklearn.metrics import confusion_matrix

conf_mat = confusion_matrix(predictions_pd, labels_pd)
conf_mat
sum = 0
for i in range(10):
    sum += conf_mat[i, i]

sum # total rows in test set = 10k
```

Code snipped-3: PySpark MLlib Logistic Regression

MLlib library functions internally convert the data to be processed into RDDs. For the MNIST dataset, the results were very significant.

Phase 5: Results and Conclusion

- Outputs

1. Vanilla implementation output

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-8-88298ee8d878> in <module>()
      1 start_train_time = time.time()
----> 2 theta = oneVsAll(X_biased, y, 10, 0.1)
      3 print('Training time: ' + str(time.time() - start_train_time))

<ipython-input-7-5439890aaca2> in oneVsAll(features, classes, n_labels, reg)
      5     for c in np.arange(1, n_labels+1):
      6         res = minimize(lrcostFunctionReg, initial_theta, args=(reg, features, (classes == c)*1), method=Non
e,
----> 7                        jac=lrgradientReg, options={'maxiter':50})
      8         all_theta[c-1] = res.x
      9     return(all_theta)

~/anaconda3/lib/python3.6/site-packages/scipy/optimize/_minimize.py in minimize(fun, x0, args, method, jac, hess, he
ssp, bounds, constraints, tol, callback, options)
    595         return _minimize_cg(fun, x0, args, jac, callback, **options)
    596     elif meth == 'bfgs':
--> 597         return _minimize_bfgs(fun, x0, args, jac, callback, **options)
    598     elif meth == 'newton-cg':
    599         return _minimize_newtoncg(fun, x0, args, jac, hess, hessp, callback,

~/anaconda3/lib/python3.6/site-packages/scipy/optimize/optimize.py in _minimize_bfgs(fun, x0, args, jac, callback, g
tol, norm, eps, maxiter, disp, return_all, **unknown_options)
    961     else:
    962         grad_calls, myfprime = wrap_function(fprime, args)
--> 963     gfk = myfprime(x0)
    964     k = 0
    965     N = len(x0)

~/anaconda3/lib/python3.6/site-packages/scipy/optimize/optimize.py in function_wrapper(*wrapper_args)
    291     def function_wrapper(*wrapper_args):
    292         ncalls[0] += 1
--> 293         return function(*(wrapper_args + args))
    294
    295     return ncalls, function_wrapper

<ipython-input-6-a0ffc6e086f9> in lrgradientReg(theta, reg, X, y)
      3     h = sigmoid(X.dot(theta.reshape(-1,1)))
      4
----> 5     grad = (1/m)*X.T.dot(h-y) + (reg/m)*np.r_[[[0]],theta[1:].reshape(-1,1)]
      6
      7     return(grad.flatten())

MemoryError:
```

Figure 8: Output of vanilla implementation

2. Scikit-learn implementation trained successfully in 28 minutes.
3. PySpark MLlib implemenation trained in 25 seconds.

The observations have been categorised based different measures of comparison.
Measure 1: Data load time:

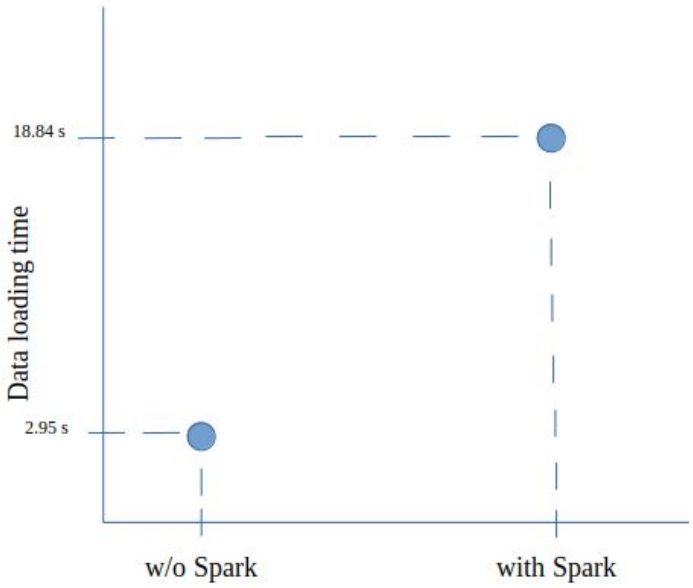

Fig. 9: Data loading time

Fig. 9 illustrates the observation that Spark took comparatively more time to load the same dataset. Since the data was stored on HDFS, it makes sense that the data loading time took more time on Spark-based implementation.

Measure 2: Training time:

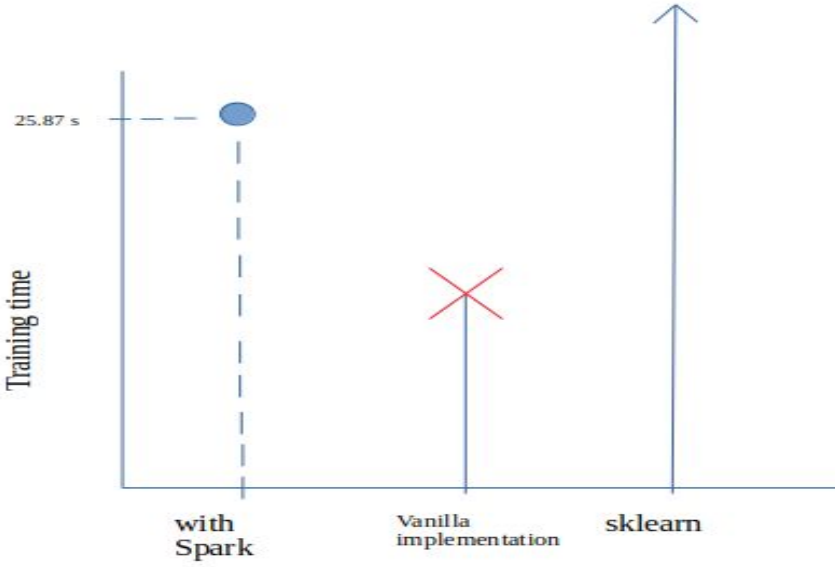

Fig.- 10: Training time

While training with Spark, MLlib Logistic Regression trains successfully, while the implementation done from scratch (vanilla implementation) failed with a *MemoryError* (figure 8). Scikit-learn *LogisticRegression* took 1721.102 seconds (approx. 28 minutes) to finish training .

Measure 3: Accuracy:

As the vanilla implementation failed to train, we could compare the accuracy on test set for only sklearn and pyspark implementations.
PySpark LogisticRegression performed better than Scikit-learn LogisticRegression. Figures 11 & 12 show the confusion matrix for PySpark's performance on the test data and Scikit-learn's performance on the test data respectively.

```
array([[ 963,    0,    5,    3,    1,    8,    5,    0,    6,    7],
       [   0, 1114,    9,    1,    1,    2,    3,    8,    8,    6],
       [   1,    4,  939,   14,    4,    3,    8,   24,    3,    3],
       [   1,    1,   16,  940,    4,   30,    1,    3,   22,   10],
       [   0,    0,   11,    3,  918,    9,    6,    3,    6,   19],
       [   5,    2,    6,   22,    0,  796,   12,    0,   21,    6],
       [   5,    3,    9,    0,   10,   11,  922,    0,    6,    0],
       [   3,    1,    6,   11,    6,    6,    0,  960,    7,   16],
       [   1,   10,   28,   11,    8,   24,    1,    2,  883,    6],
       [   1,    0,    3,    5,   30,    3,    0,   28,   12,  936]])
```

Figure- 11: Confusion matrix for PySpark implementation performance on test set

```
array([[ 963,    0,    2,    1,    1,    2,    3,    3,    4,    1],
       [   0, 1115,    2,    1,    1,    1,    4,    1,    9,    1],
       [   4,    9,  922,   15,    7,    6,   10,   11,   46,    2],
       [   3,    2,   15,  934,    2,   14,    3,   10,   19,    8],
       [   2,    3,    4,    3,  914,    0,    9,    5,    5,   37],
       [   9,    2,    0,   29,    6,  788,   14,    6,   29,    9],
       [   7,    3,    4,    0,    5,   14,  918,    0,    7,    0],
       [   2,    7,   22,    4,    3,    0,    1,  954,    5,   30],
       [  10,   10,    8,   18,   10,   18,    7,   10,  873,   10],
       [   9,    8,    0,   13,   24,    9,    0,   26,   12,  908]])
```

Figure 12- Confusion matrix for Scikit-learn performance on test set

Accuracy of PySpark: 93.72%
Accuracy of sklearn: 92.89%

Following table summarizes the observed values for comparision measures:

	Vanilla implementation	Scikit-learn	PySpark
Data loading time	2.95s	2.95s	18.84
Model training time	X	1721.102s	25.87s
Accuracy	X	92.89%	93.72

Table 1: Comparison measure values

Conclusion: Spark, alongwith HDFS, can make computations very fast. For this specific problem, PySpark was able to train about 67 times faster than the Scikit-learn implementation. On the other hand, the vanilla implementation did not finish training. Therefore, Spark could be a promising option for processing big datasets.