ILLINOIS DATA SCIENCE INITIATIVE

TECHNICAL REPORT

A Comparison between Apache Spark's MLlib vs. Python's Scikit-learn

Author:
Ema Milojkovic
David Wang
Professor Robert J. Brunner

June 2, 2017

A Comparison Between Apache Spark's MLlib vs. Python's Scikit-learn

EMA MILOJKOVIC[1], DAVID WANG[1], AND PROFESSOR ROBERT J. BRUNNER[1,2,3]

[1] *University of Illinois at Urbana-Champaign*
[2] *National Center for Supercomputing Applications*
[3] *Laboratory for Computation, Data, and Machine Learning*

Compiled June 2, 2017

This paper aims to identify the ideal data environments for optimal performance by Apache Spark's MLlib and Python's Scikit-learn by comparing run time and outputs of classification, regression, and clustering machine learning algorithms. This paper is intended for beginner data scientists who are uncertain about which machine learning library is best for their data and hardware conditions.

https://github.com/lcdm-uiuc

1. INTRODUCTION

Every day, approximately 2.5 million terabytes of data is created. What makes this statistic significant is that 90% of the world's data has been created in the past two years alone.[5] The recent surge in data quantity has opened a new area of computer science known as "Big Data." In efforts to analyze this vast volume of information, a new division of computer scientists, data scientists, have turned to machine learning libraries to efficiently carry out calculations. Among the most common machine learning libraries are MLlib and Scikit-learn.

Both libraries have algorithms for different means of data analysis including classification, regression, and clustering. Acknowledging the fact that both MLlib and Scikit-Learn offer almost identical services, it can be difficult for beginning data scientists to distinguish when one should be used over the other. This report will show various trials and tests to determine when to use MLlib over Scikit-learn.

The rest of this paper will be as follows: first, a list of assumptions and a general description of MLlib and Scikit-Learn will be provided. Second, the datasets used in the various trials will be identified, including how the data was mapped for the specific algorithms. Next, an explanation of each machine learning algorithm tested will be provided along with an explanation of its parameters. Then, the outputs of the algorithms in different execution conditions will be compared. Finally, a conclusion of optimal situations for both machine learning libraries will be identified.

2. ASSUMPTIONS AND SETUP

A. Assumptions

This paper assumes that the reader has access to a Hadoop cluster with Hadoop Distributed File System, Spark, and Python 3 installed.

B. Software

MLlib is a library that is available when Spark is downloaded. Spark can be installed and run locally on a multi-core machine or deployed on a Hadoop 2 Cluster.[3] Scikit-learn and Natural Language Toolkit (NLTK) must be installed on the cluster.

C. Setup to Execution

To maximize similarities in MLlib and Scikit-learn's code execution the data sets were parsed and cleaned prior to script execution. Algorithms from both libraries were given the same cleaned data sets to analyze, so the only metric to test was their runtime and performance

The MLlib programs read in the data from files uploaded to HDFS and were executed using spark-submit with the following command:

```
spark-submit --master yarn-client --num-
    ↪ executors 10 --executor-cores 5 --
    ↪ executor-memory 4G --driver-memory 4G [
    ↪ MLlib script name].py
```

The Scikit-learn programs read in the data from text files that were copied from HDFS to local and were executed in regular Python fashion:

```
python3 [Scikit-learn script name].py
```

Although Scikit-learn offers functionality to parallelize data analysis in certain cases, it is not advertised as a key aspect of the library. For this reason, the Scikit-learn scripts will be executed with the standard process for comparison. MLlib will use parallelized execution, a core feature of the library, for comparison of their optimal use cases. Since Scikit-learn is severely limited by the quantity of data that can be allocated within RAM, scripts for both libraries were run on a 128 GB server on the Nebula cluster provided by the National Center for Supercomputing Applications.

3. THE MACHINE LEARNING LIBRARIES

A. MLlib

MLlib is Spark's distributed machine learning library used to process large datasets. The library targets large-scale learning settings that benefit from data-parallelism or model-parallelism to store and operate on data or models.

MLlib consists of fast and scalable implementations of common machine learning algorithms including classification, regression, collaborative filtering, clustering, and dimensionality reduction. MLlib was made to inter-operate with NumPy in Python, R libraries, and any Hadoop data source (such as HDFS or HBase). [1]

B. Scikit-Learn

Scikit-learn is an open source, commercially usable machine learning library for Python. Traditionally it is used on the scale of one hundred thousand data points and does not utilize data-parallelism.

It supports both supervised and unsupervised learning through various classification, regression and clustering algorithms, which include support vector machines, random forests, gradient boosting, k-means and DBSCAN. However, Scikit-learn does not support deep or reinforcement learning. Scikit-Learn is designed to inter-operate with the Python numerical and scientific libraries NumPy, SciPy, and matplotlib. [8]

C. Note

All of the algorithm and data mapping code in this report will be written in PySpark[1] to be used with Spark and MLlib. This was chosen because Scikit-learn has very thorough documentation and a larger online community of users. MLlib is harder to use and therefore the code included was intended to aid beginners using Spark and MLlib.

4. DATASETS

A. Movie Reviews Dataset

For the purpose of the Naive Bayes comparison, a pre-classified movie review dataset (provided by researcher Dai Quoc Nguyen)[7] was chosen. It holds 233,600 labeled movie reviews in a very simple format that follows the pattern

```
(review text, review score)
```

This data was first cleaned and mapped using NLTK [2] to remove unnecessary words and punctuation.

```
reviews = reviews.mapPartitions(lambda x: csv.
    ↪ reader(x, delimiter=','))
words_score = reviews.map(map_portable_format)
words_score = words_score.filter(lambda x: x)
```

The code above maps the reviews using a function defined as *map_portable_format*. This function will take in lines from the RDD one at a time. After ensuring that a given line has a text and a score, the following Spark code, using NLTK, was used to map and return the data in a more usable format:

```
tokenizer = RegexpTokenizer(r'\w+')
text = tokenizer.tokenize(text)
stop = set(nltk.corpus.stopwords.words('
    ↪ english'))
text = [word.lower() for word in text if word.
    ↪ lower() not in stop]
return score + "\t" + "␣".join(text)
```

The first two lines of code tokenizes the inputted text line. That is, it splits the line by spaces making each word a "token." Next, all stop words[3] are filtered out of the text. The remaining important words are then combined into a list and returned with their overall score.

The idea here is that each word in the review is now associated with the review's overall score. The algorithm can then identify similar words and predict similar ratings.

B. Financial Stock Dataset

A dataset containing the value of stocks at different times and dates was provided through a corporate sponsor. The data is split between approximately one thousand files, where each file contains one stock. The file sizes range from a few megabytes to a few gigabytes, containing lines formatted in the following pattern:

```
(PERMNO, SYMBOL, DATE, itime, value)
```

The PERMNO value is a unique numeric identifier for the current business, similar to a stock symbol. The SYMBOL value is the universal stock exchange symbol for the current business. DATE represents the current date and itime represents the time at which the stock had a value represented by the value parameter.

This data was only slightly reformatted by converting the DATE and itime values into a single datetime value which represents a date, plus a fraction of the time that would have passed in the day according to the itime parameter. The following python function was used to perform this operation:

```
#Maps data from csv to a (stock,time) point
def map_to_point(row):
    date = row[2]
    time = row[3]
    value = row[4]
    try:
        date = float(date)
        value = float(value)
        h, m, s = time.split(':')
        offset = (int(h) * 60 + int(m))/(float
            ↪ (1440))
        date += offset
        time_vector = []
        time_vector.append(round(date, 1))
        return (value, time_vector)
    except ValueError:
        return None
```

This function takes in a row of stock data and grabs the DATE, itime, and value. Using itime, a fraction of a day is calculated and appended to the current date value. Finally, this data is returned as a tuple with the feature vector being the datetime and the label being the value of the stock at that given time.

[1] Since PySpark is Python based, some code will be identical when translating for use with Scikit-learn.

[2] NLTK is a Python library used for Natural Language Processing. It offers features for classification, tokenization, parsing, and semantic reasoning among others.

[3] Stop words are a group of the most common words in a language that add no new information to a text. Examples of stop words in English are "the," "do," "for," and so on.

C. Artificially Clustered Dataset

For the KMeans clustering algorithm, no official dataset was used. Instead, a python script was written and executed to create an "artificially clustered" [4] set of data points. The dataset was generated using 7 cluster centers, containing 1,000,000 data points. However, these values were picked arbitrarily as any number of clusters and data points would work.

The location of each data point was determined by using a Gaussian distribution to randomly select a distance from the cluster center in the X and Y directions. The specified parameters for the dataset generation process are as follows:

- **num_clusters = 7** – the number of clusters that points were centered around
- **max_data_points = 1,000,000** – the number of data points that were generated
- **mean_of_sd = 6,000,000** – the average standard deviation of the Gaussian distribution that determined each cluster's standard deviation
- **sd_of_sd = 60,000** – the standard deviation of Gaussian distribution that determined each cluster's standard deviation
- **max_x = 100,000,000** – the maximum X coordinate value
- **max_y = 100,000,000** – the maximum Y coordinate value

Using these values for the parameters generated the graph of clusters see in Figure 1:

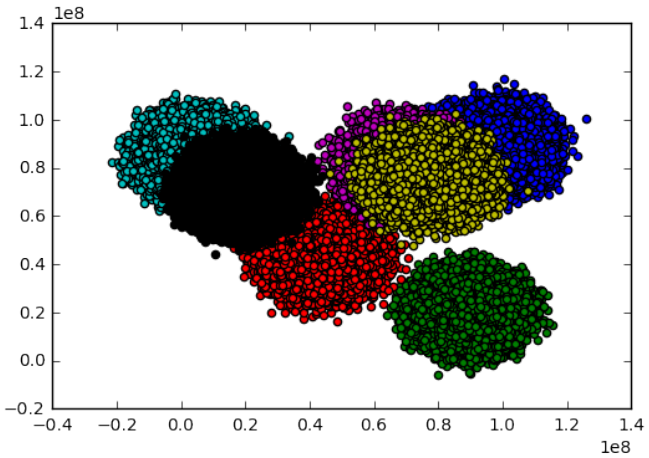

Fig. 1. Artificial clustered generated using a Gaussian distribution. Each color denotes one of the 7 clusters.

All values were decided on through trial and error. It was found that these values created clusters that were relatively evenly distributed throughout the X and Y coordinates so that there wasn't one huge cluster and made clusters that weren't absurdly dense with points. The following code snippets will be specific to X values, however identical code was used for Y values.

First, the cluster centers must be determined. Using the Python library NumPy, random integers can be generated for the X and Y coordinate values of the center of each cluster.

```
x_centers = np.random.randint(max_x, size =
    ↪ num_clusters)
```

Next, each cluster must have its own standard deviation value which determines how far each generated point can be from the center.

```
x_sd = 1 + np.abs( np.random.normal(mean_of_sd
    ↪ , sd_of_sd, size = num_clusters) )
```

The x values for each cluster can be represented as a list of lists where the parent list contains lists of X coordinates for each cluster. Using this schema, the data points can be randomly generated and recorded as follows

```
for i in range(max_data_points):
    cluster = np.random.randint(num_clusters)
    x_i = np.random.normal(x_centers[cluster],
        ↪ x_sd[cluster])
    X[cluster].append(x_i)
```

This will generate the maximum number data points by randomly selecting one of the cluster centers to be it's center and then using the cluster's standard deviation to plot a point. Using the Gaussian distribution ensures that the points will be somewhat clustered instead of uniformly distributed. The X and Y values were then matched up and written to an output data file as :

```
(x_value, y_value)
```

The final dataset contained 1,000,000 data points and was 43 MB in size. The text file containing the data was directly accessible to the Scikit-learn script. However, the file was also uploaded to HDFS for MLlib to have access. Since the points were already mapped, they only needed to be parsed from the file and stored to their library's respective data structures – a list for Scikit-learn and an RDD for MLlib.

5. MACHINE LEARNING ALGORITHM

The Scikit-learn community has created a flow chart for selecting machine learning algorithms, which is shown in Figure 2 [2] Since MLlib and Scikit-learn provide many of the same machine learning algorithms, the flow chart's structure holds for both libraries for deciding appropriate algorithms. However, since MLlib can run efficiently on much larger datasets, the numbers in the chart are specific to Scikit-learn and are not true for MLlib. This flowchart was used to determine the algorithms for the datasets specified above.

A. Classifying Movie Reviews: Naive Bayes

Scikit-learn and MLlib both have functionality for classification using Multinomial Naive Bayes and term frequency - inverse document frequency (TF-IDF). TF-IDF is used to determine how important a word is in a given text. Naive Bayes is a supervised learning algorithm optimized to run analysis on labeled text data points to predict labels on new data points.

Using the movie review dataset, the Naive Bayes algorithm can predict whether a movie had a positive or negative rating from the words within its review. This can be achieved by labeling every word with the review's overall rating out of 5 stars with the term frequency - inverse document frequency feature to create a training set of data. To extract the TF-IDF feature from the movie reviews, the following Spark MLlib code was used.

```
tf = HashingTF().transform(labeled_data.map(
    ↪ lambda x: x[1]))
```

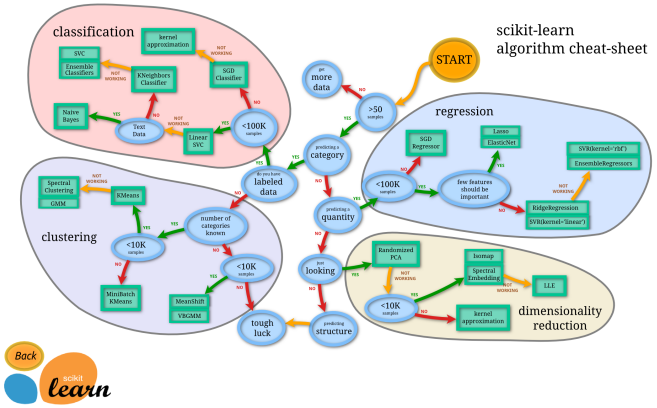

Fig. 2. As provided by the Scikit-learn community, the flowchart shows the different machine learning algorithms and what data situations are ideal for each one.

```
idf = IDF(minDocFreq=50).fit(tf)
tfidf = idf.transform(tf)
```

This newly parsed TF-IDF data is then split into a training and test data set. A model can then be trained using the Naive Bayes algorithm on the training data.

```
# Reassemble the data into (label, feature)
    ↪ key-value pairs
zipped_data =
        (labels.zip(tfidf).map(lambda x:
            ↪ LabeledPoint(x[0], x[1])).cache
            ↪ ())
training, test = zipped_data.randomSplit([0.7,
    ↪  0.3])
model = NaiveBayes.train(training)
```

The Naive Bayes algorithm was run with default parameters because the only parameter that could have been changed was lambda, a value which controls additive smoothing. This was deemed unnecessary in the context of the movie dataset that was used.

Finally, the trained model's accuracy was judged using Spark MLlib's built in confusion matrix and precision metrics.

```
# Use the test data and get predicted labels
    ↪ from our model
test_preds = (test.map(lambda x: x.label)
                  .zip(model.predict(test.map(
                      ↪ lambda x: x.features)
                      ↪ )))
test_metrics = MulticlassMetrics(test_preds.
    ↪ map(lambda x: (x[0], float(x[1]))))
print(test_metrics.confusionMatrix().toArray()
    ↪ )
print(test_metrics.precision())
```

The method being used to analyze the results and compare accuracies from Naive Bayes is a confusion matrix, also known as an error matrix. Simply put, a confusion matrix shows actual v.s. predicted values for every classification label. In this case, the confusion matrix has the ratings (negative, positive) on left and bottom sides. The columns represent Naive Bayes' predicted rating and the rows represent the actual ratings. Figure 3 is an example of a confusion matrix with a description of how to interpret it.

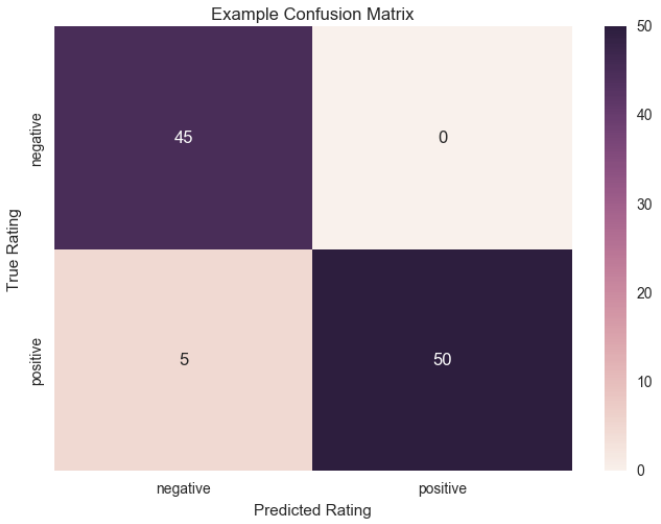

Fig. 3. The numbers in this example were arbitrarily chosen. The confusion matrix would be interpreted as follows: 45 negative reviews were successfully identified as negative reviews, 0 negative reviews were identified as positive, 5 positive reviews were mistakenly identified as negative, and 50 positive reviews were successfully identified as positive reviews.

The darkness of a square in the confusion matrix is determined by the frequency of that square's classification. If a certain prediction occurs more frequently, the square will appear darker. A perfect model would produce a confusion matrix with dark colors exclusively on the major matrix diagonal, implying that every prediction of class "X" correctly predicted class "X".

B. Regression on Stock Data

A linear regression algorithm takes in data points (as a two dimensional coordinate) and creates a model to predict a y value for every x value such that the predicted y values are linear when plotted. This line of y values is also known as the "best fit line" since it minimizes the squared summed differences in distance between a data point's y value and its predicted value. With regards to stocks, a linear regression model can be used to predict a stock's value at any given time in the dataset range. The label will be the actual value of each stock at a certain point in time. Once the data is correctly formatted and mapped the data is split and fed into the Linear Regression algorithm for training.

```
model = LinearRegressionWithSGD.train(
    ↪ training, iterations=1000, step
    ↪ =0.00000001, intercept=True)
```

To train the Linear Regression model the follow parameters were used:

- iterations (default 100) - 1000 was used to increase the accuracy of the model
- step (default 1.0) - 0.00000001, The step value was chosen by running the algorithm with differing step values on a sample of the stock dataset. Table 1 shows these results.

- intercept (default false) - True, This parameter allows the linear regression to intercept the y-axis instead of being forced to pass through the origin. Because the stock prices do not always pass through the origin, intercept is set to true.

- other parameters - Other parameters were unchanged from their default values because changing them would not result in significant differences in percent error

Table 1. Percent Error For Differing Step Values

Step Value	*PercentError*
0.000001	*NaN*[4]
0.0000001	233517.25%
0.00000001	1.39%
0.000000001	66.63%
0.0000000001	90.71%

Once the model has finished training, error values can be extracted in the following manner.

```
# function to calculate percent error
def map_percent_error(data_point):
    y_true = data_point[0]
    y_pred = data_point[1]
    if y_true != 0:
        return (abs((y_true - y_pred) / y_true
            ↪ ) * 100, 1)
    return None

# run predictions on the trained model
test_features = test.map(lambda x: x.features)
predictions = model.predict(test.map(lambda x:
    ↪  x.features))
test_preds = test.map(lambda x: x.label).zip(
    ↪ predictions)

# calculate mean absolute percent error
total_percent = test_preds.map(
    ↪ map_percent_error)
total_percent = total_percent.filter(lambda x:
    ↪  x)
average_percent = total_percent.reduce(lambda
    ↪ x, y: (x[0] + y[0], x[1] + y[1]))
average_percent = average_percent[0] /
    ↪ average_percent[1]
```

Using the percent error that was calculated from the map_percent_error function, the mean absolute percentage error was found for each linear regression that was executed on every stock file.

C. Clustering Artificially Clustered Data

Clustering is a form of unsupervised learning where the model is provided a data set and number of clusters (centroids), k. The algorithm divides the data set points into k groups of similar data points. Similarity between points is determined by their numerical similarities. For example, given a data set of coordinates, the KMeans algorithm clusters points by their x and y values. To select the ideal number of clusters, the data must first be trained with different k values. On each trial, the WSSSE (within set sum of squares error) is recorded and plotted. When plotted, the k value that creates the "elbow" in the graph is the point of diminishing returns [6] and is the k value that is the optimal number of clusters for the dataset. Due to the ambiguity of the exact point of diminishing returns, the exact k picked can, and often does, differ between people by a small margin.

The algorithm used to compute the WSSSE score is as follows:

```
errors = []
# Build the model (cluster the data)
for i in range(1, 15):
    clusters = KMeans.train(data, i,
        ↪ maxIterations=300, runs=10,
        ↪ initializationMode="k-means")
    WSSSE = data.map(lambda point: error(point
        ↪ )).reduce(lambda x, y: x + y)
    errors.append((i, str(WSSSE)))
```

where error for a given point is defined as the sum of the squared error. This mathematically defined as : $\sum_{i=1}^{K} \sum_{x \in c_i} dist(x, c_i)^2$ where K is the number of clusters, x is a point inside the cluster, and c_i is the cluster's center. This formula can be represented in python as

```
def error(point):
    center = clusters.centers[clusters.
        ↪ predict(point)]
    return sqrt(sum([x**2 for x in (point -
        ↪ center)]))
```

For the artificially clustered dataset, the plotted WSSSE values appear as follows in Figure 4 :

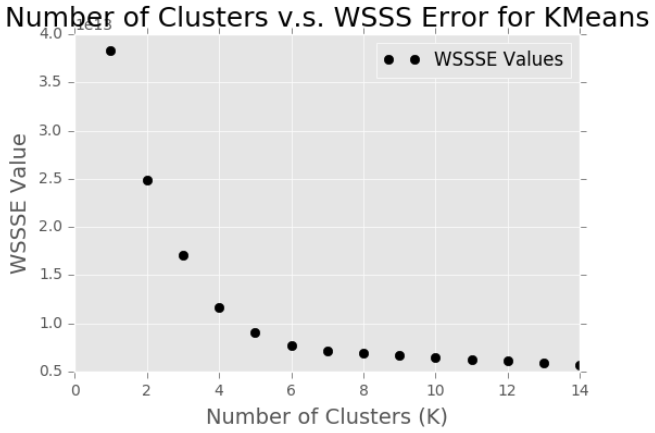

Fig. 4. As more clusters are used to classify the data, the WSSS Error decreases. Notice how the points 5, 6, and 7 could all easily be seen as the point of diminishing return. This is due to the inherent ambiguity of exactly when the point of diminishing returns is reached.

Given the graph above, the point of diminishing returns occurs when K is approximately 7, which will be the value used.

Because Scikit-learn is a very mature library, the MLlib clustering parameters were chosen to very closely match to the Scikit-

[4]This means that the percent error was too large to be understood by Python

learn parameters. This resulted in the following parameters to train the KMeans model:

- k - 7, as selected to be the desired number of clusters
- max iterations - 300 was used to match the Scikit-learn default for maximum number of iterations to run. This value was appropriate for the dataset.
- runs (default 10) - 10, equivalent to Scikit-learn's n_init parameter whose default is also 10. This parameter determines how many times the KMeans algorithm will be run with different centroid seeds.
- initializationMode - 'kmeans' was used to match Scikit-learn's optimized parameter 'k-means++' for selecting centroid seeds to reach convergence more quickly. MLlib does not have a 'k-means++' option.
- other parameters - Other parameters were unchanged from their default values because changing them would not result in significant differences in output.

The code for clustering the data in PySpark is as follows:

```
clusters = KMeans.train(data, 7, maxIterations
    ↪ =300, runs=10, initializationMode="k–
    ↪ means")
point = data.map(lambda x: (x, clusters.
    ↪ predict(x)))
```

The first line uses the KMeans algorithm, as imported from MLlib, to train a model on the current data RDD. Then, a final map command on the RDD matches each coordinate to its cluster number (between 1 and 7) and mapped to a unique color.

6. EXPERIMENTAL PROCEDURE

To conduct this experiment and gather results for comparison against the two machine learning libraries the following procedure was conducted:

Firstly, each algorithm was executed on the same hardware as outlined above in the Hardware Specifications section. Each MLlib algorithm was executed with the parameters specified in the Assumptions section. The scripts were also ran one after the other for MLlib and Scikit-learn to ensure the conditions and server load of the execution environment were kept as close as possible.

7. RESULTS & ANALYSIS

The output from all of the trials were documented and compared, using each algorithm's respective accuracy metric.

A. Naive Bayes

When running training and testing through a Naive Bayes model, the model's accuracy is seen through a confusion matrix.

Figures 5 and 6 shows the confusion matrix outputs from each library's Naive Bayes model and Table 2 compares the precision and runtime of the two libaries' Naive Bayes.

When comparing the results of Naive Bayes run using both machine learning libraries, one can see that the precision does not differ significantly between the two libraries. MLlib pulls ahead slightly with 86.29% vs 85.73%, but this difference is negligible. However, observing the two confusion matrices shows

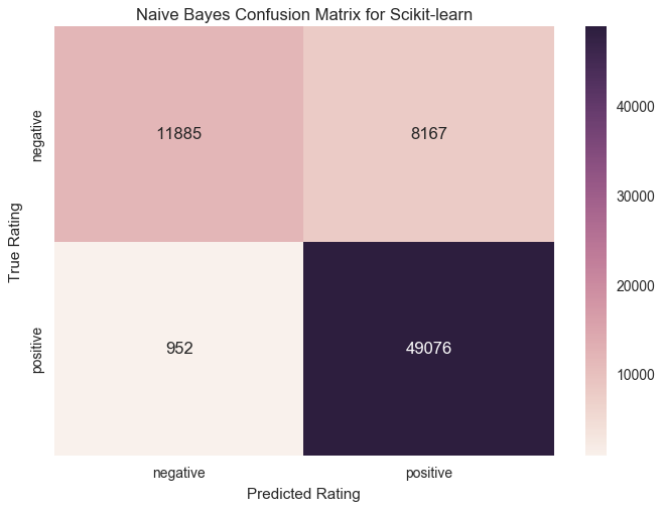

Fig. 5. The Scikit-learn Naive Bayes algorithm had a 98% success rate for identifying positive reviews as positive and a 59% success rate for identifying a negative review as negative.

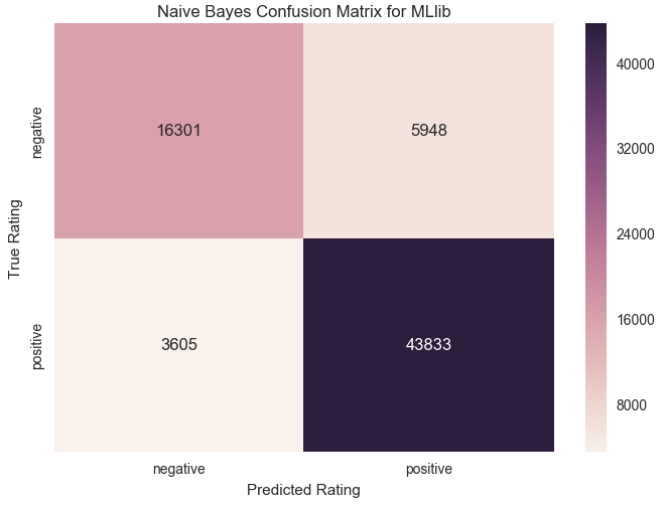

Fig. 6. The MLlib Naive Bayes algorithm had a 92% success rate for identifying positive reviews as positive and a 77% success rate for identifying a negative review as negative.

Table 2. Precision Values for Naive Bayes Algorithm

Library	Precision	Time (Minutes)
MLlib	86.29%	4.9666
Scikit − learn	85.73%	1.1026

that Scikit-learn had a higher positive-positive classification success, but MLlib had a higher negative-negative success. Neither library seemed to have an overall "better" Naive Bayes algorithm in regards to accuracy.

What is more interesting is the difference in run times between the two libraries. Scikit-learn was able to train the Naive Bayes model around 5x faster (1 minute vs 5 minutes). This is believed to be mainly due to that fact that MLlib was designed to handle significantly larger datasets by parallelizing and distributing data and data operations over many nodes. This, however creates some overhead that will make performance relatively slower for smaller datasets. This same conclusion becomes even more apparent when comparing the results of linear regression ran on MLlib and Scikit-learn.

B. Linear Regression

As mentioned before, a linear regression analysis was run on a collection of stocks. Since the dataset was large enough, the data was sampled into 1GB, 15GB, and 30GB samples. The linear regression algorithm was run on each dataset size.

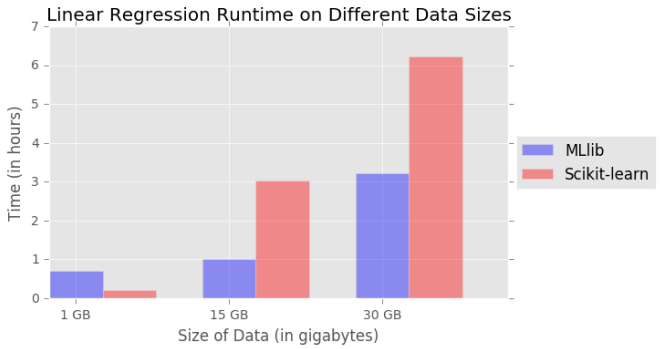

Fig. 7. As the size of the data increased, there is a shift in which library was faster. For smaller datasets, Scikit-learn finished computations much quicker, but for lager datasets MLlib finished quicker.

Figure 7 shows the time it took the linear regression algorithm to run on a 1 GB, 15 GB, and 30 GB dataset. As one can see from the graph of run times, Scikit-learn will initially perform quite well on small datasets that comfortably fit on one machine (1 GB). However, as the datasets got larger, Scikit-learn failed to scale. As noted in the Naive Bayes section, MLlib runs slower at small dataset sizes; the process of parallelization of the data at this size exceeds the time for the algorithm to run, thus causing a larger runtime. For larger datasets, MLlib performed as expected. MLlib completed its linear regression algorithm significantly before Scikit-learn, often finishing in about 10% of the time it took Scikit-learn. More notably, MLlib scaled more efficiently as the dataset size grew. The differences in time to finish between the two libraries grew as MLlib was able to keep up with the data size and Scikit-learn struggled.

The percent errors for the library's linear regression algorithms tell an interesting story: as the data size increased, there appeared to be an insignificant change in percent error. This means that the algorithms were able to maintain their accuracy of predictions, regardless of what level of accuracy it was. Figure 8 shows that across the board, Scikit-learn had a lower percent error. In other words, Scikit-learn's linear regression algorithm tends to be more accurate at any given data size.

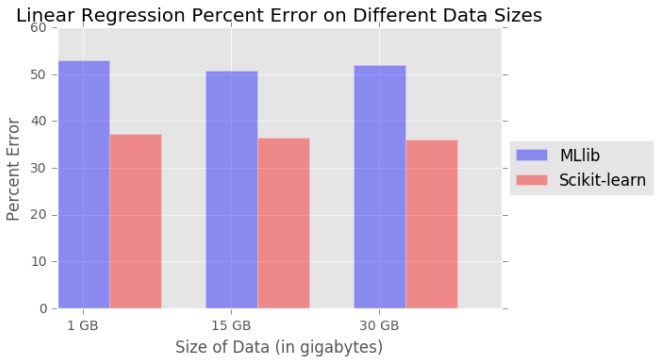

Fig. 8. With the various dataset sizes above, Scikit-learn has shown to consistently have a lower percent error on its computations.

When combining the information from runtime and from percent error, it seems that MLlib compromises speed for accuracy as it consistently has a higher percent error, but is able to analyze large datasets more efficiently than does Scikit-learn. Scikit-learn is on the other end of this spectrum where speed is sacrificed to produce more accurate results. This is most likely due to changes in how each algorithm was trained with different model parameters. Scikit-learn does not give the user any control over the number of iterations or step value when doing linear regression. These values are chosen independently by the library according to attributes of the dataset. This allows for Scikit-learn to pick parameters that probably allow for such higher accuracy compared to MLlib.

C. K Means

Of the three machine learning algorithms explored in this report, KMeans seemed to have the least difference between the two libraries. Figures 9 and 10 show the clusters generated by the two algorithms for the same dataset. Both Graphs are different from the original set of clusters, as the original set and three overlapping clusters in the top right corner and two overlapping clusters in the top left. Interestingly, both libraries split up the overlapping clusters almost identically. Instead of overlaying the clusters, the one large cluster created by the actual clusters was split up into adjacent clusters.

There appeared to be little difference in the run times of both libraries' algorithms since they both finished within a minute of each other. More specifically, Scikit-learn finished running in about 4 minutes and MLlib took about 5 minutes.

Something to note is that KMeans is a well established and practiced machine learning algorithm based off of Lloyd's algorithm [4]. Both libraries seem to have similar if not identical implementation of this algorithm since they produced seemingly identical output when run with the same parameters. This speaks to the similarities between MLlib and Scikit-learn in the sense that their some of algorithms have the same implementation.

8. CONCLUSION

When choosing which library to use, one of the biggest factors is the size of the dataset which is being analyzed. From the results gathered, if your dataset is large, then MLlib would be the best option. This was shown most confidently in the linear regression tests that were run on the stock dataset. As outlined previously,

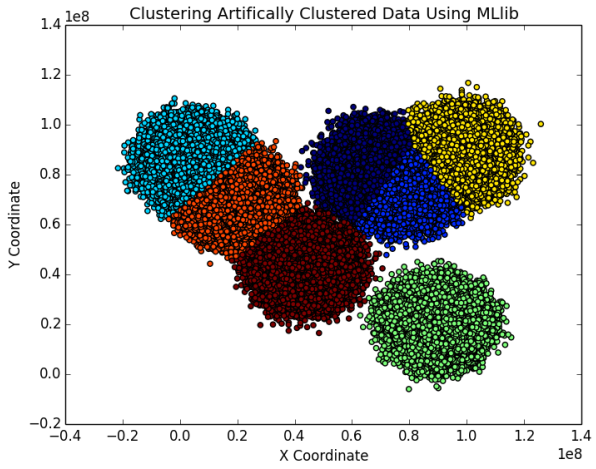

Fig. 9. The graph used MLlib's KMeans algorithm to plot coordinates clustered by their x and y values. There are 7 groups, each of which is denoted by a different color.

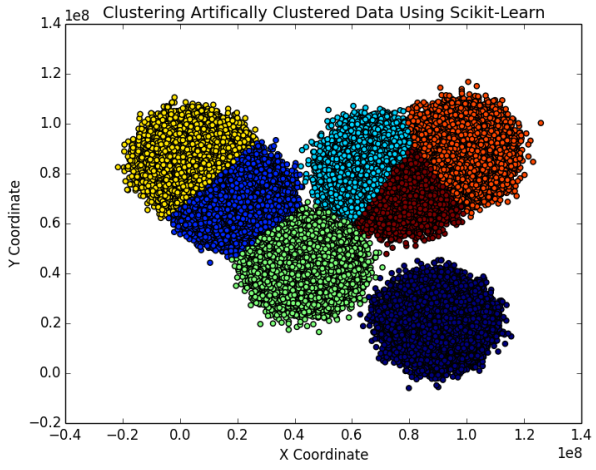

Fig. 10. The graph used Scikit's KMeans algorithm to plot coordinates clustered by their x and y values. There are 7 groups, each of which is denoted by a different color.

Scikit-learn has trouble keeping up with MLlib's distributed data operations and quickly fell behind in run time even though it had a large advantage for smaller datasets such as the 1GB stock dataset.

Second, it is important to consider the trade-off between run time and precision when choosing between Scikit-learn and MLlib. For most cases, such as using Naive Bayes or KMeans, the output from both libraries were very similar in output/precision values. However in the example of Linear Regression, MLlib produced data with a considerably larger percent error (50% error for MLlib vs Scikit's 35% error) as can be seen in Figure 8. When precision must be preserved on datasets that can fit on one machine's memory, Scikit-learn would be the better suited library.

Finally, one should examine the ease of use of both libraries when choosing between them. Scikit-learn has very easy integration with popular Python libraries such as NumPy and Matplotlib. This allows users to very quickly train machine learning models and visualize output with little difficulty. When using MLlib, output must usually be written to HDFS and must be properly mapped in order to be used in conjunction with data visualization libraries such as Matplotlib. Also, Scikit-learn is far richer in terms of implementations of a larger number of commonly used algorithms as compared to Spark MLlib. These algorithms are also extremely well documented due to having a large community of users compared to the less popular MLlib.

Scikit-learn is a very well established machine learning library with strong implementations of many algorithms and thorough documentation. MLlib has coverage for many common machine learning methods such as ones used in this paper (Naive Bayes, Linear Regression, KMeans, etc...). The library, however, is still not as fully-featured as Scikit-learn and, as stated previously, MLlib's only real advantage over Scikit is allowing for a smaller run time on large datasets - usually spanning many gigabytes - due to its ability to parallelize data. Ultimately the decision to use MLlib or Scikit-learn comes down to the priorities of the project and the available resources.

SUPPLEMENTAL DOCUMENTS

For details on the two machine learning libraries, see scikit-learn and MLlib.

ACKNOWLEDGMENTS

The authors thank Sameet Sapra and Quinn Jarrell for technical support and NCSA for access to the Nebula Cluster, without which this project wouldn't be possible. The authors would also like to thank Tyson Trauger, Teja Ambati, and Josh Dunigan for edits and feedback.

REFERENCES

1. Mllib | apache spark. URL http://spark.apache.org/mllib/.
2. Choosing the right estimator¶. URL http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html.
3. Spark standalone mode. URL http://spark.apache.org/docs/latest/spark-standalone.html.
4. Clustering with k-means in python, Apr 2014. URL https://datasciencelab.wordpress.com/2013/12/12/clustering-with-k-means-in-python/.
5. Bringing big data to the enterprise. Mar 2017. URL https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html.

6. Daniel Martin. How can we choose a "good" k for k-means clustering?, Dec 2013. URL https://www.quora.com/How-can-we-choose-a-good-K-for-K-means-clustering.
7. Dai Quoc Nguyen, Dat Quoc Nguyen, Thanh Vu, and Son Bao Pham. Sentiment Classification on Polarity Reviews: An Empirical Study Using Rating-based Features. In *Proceedings of the 5th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis*, pages 128–135, 2014.
8. Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.