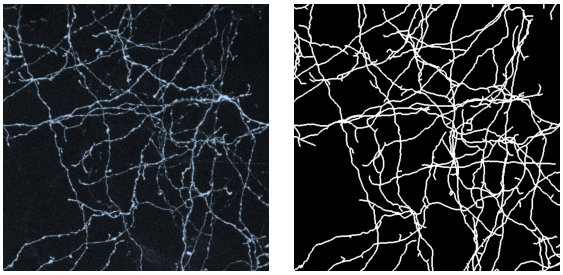
Northeastern University
EECE 5644 – Pattern Recognition and Machine Learning
April 2018, Asha Chen-Phang, Emily Dutile, Nate Otenti, Tristan Sweeney

Classifying High-Resolution Brain Scans using Apache Spark

1 Introduction

Being in the multi-core and machine learning era, it is important for our machine learning algorithms to take advantage of the potential speed up through the use of paralleling tasks on multi-core computers and distributed systems. With an interest in processing a massive image dataset, performing feature engineering and using well-known industry solution for faster computation, the group used Apache Spark [19], a processing model for analyzing big data, to analyze the speed up of machine learning algorithms for foreground-background classification in high-resolution brain scans. With the dataset, we performed preprocessing, feature extraction, model implementations in python using sklearn in a single-processor solution, and model implementations in scala on top of Apache Spark using MLlib to analyze time and processing efficiency in a parallel processing program upon distributed data along with accuracy. In a single machine environment, we experimented with Support Vector Machine (SVM), K-Nearest Neighbors (KNN), and Random Forest [17]. In our parallelized environment, Random Forest was implemented to analyze the model and training phase with parameter tuning. The Apache Spark environment was created on an Amazon EMR [8] cluster, leveraging the EMR file system (EMRFS) to access data in Amazon S3. The README file in the repository has details on environment setup.

The dataset used comes from a group of scientist that are interested in turning high-resolution brain scans, as seen in the left figure , into a graph representing nerve connections (indicated by the bright lines). The original image is 3-dimensional, but the 2-dimensional projection on the X-Y plane is shown below. The axons, which are the lines going across the image, are clearly visible, however, we do see a lot of noise in the image. Each pixel in an image was classified as either foreground (belongs to an axon) or background (does not belong to an axon) in order to improve the quality of algorithms that automatically trace these axons. The axons were manually traced by the group of scientist. The traced data looks like the figure on the right, where white indicates foreground and black indicates background.



The labeled data was generated as follows (using the manually traced image):

- In the image, select a pixel (i, j, k).
- Extract a neighborhood vector centered around (i, j, k).
- Extract the label of (i, j, k) from the trace. Here 1 indicates foreground; 0 indicates background.
- Save the record as the neighborhood vector, followed by the label.

As an example, assuming pixel (i=10, j=10, k=10) was selected and we are working with a neighborhood of size 3x3x3, these 27 pixels would be stored as a vector n with 27 elements, where n[0] stores the brightness value of pixel (9, 9, 9), n[1] the brightness of (9, 9, 10), n[2] of (9, 9, 11), n[3] of (9, 10, 9), and so on. The labeled data set contains neighborhoods of size 21x21x7, which was recommended by the domain experts.

1.1 The Data

The dataset[1] is composed of 6 labeled csv files as images, each roughly 6.5 GB. Each image (file) has rows that contain the input vector of 21x21x7 brightness values (intensity) from a 3D image, together with the center pixel's foreground-vs-background label. The last value is the label for the whole image. On each line there are 21*21*7+1 values. The labeled data from images 1, 2, 3, 4, and 6 are used to train the most accurate model for predicting the labels in image 5 of the datasets. In the final evaluation set, there are nearly 0.0057% foreground and 99.99943% background pixel. There is a csv file containing the true labels for image 5, in which the actual accuracy could be measured to verify if we beat the baseline of 98.6917962635579%. This results file was only used for final evaluation to verify if we achieved the goal of beating the baseline. For classification accuracy, a high value does not necessarily mean the model is performing well. In our case, the "dumb" model that always predicts label 0 for every input will have 99% accuracy, so any model achieving less than 99% would not be beating the dumb model.

2 Related Works

Being in the big data age, datasets are rapidly growing in size and complexity. Due to this, cloud computing architectures and solutions are becoming more pervasive and machine learning is also becoming a vital component of these large-scale data processing pipelines. Under this umbrella, exploratory analysis, feature engineering, machine learning, and model evaluation are all critical components to the distribute machine learning solution. Since many machine learning techniques are computationally expensive, this makes them ideal candidates for parallelization. However, these methods are usually complex, so implementing a parallelized solution is often challenging.

Apache Spark is a well known open source framework that is becoming more readily used within industry as a cluster computer system that excels in scaling machine learning tasks, including its open-source distributed machine learning library as a standard component, MLlib [20]. MLlib contains algorithms for classification, regression, collaborative filtering, clustering, and decomposition. Apache Spark is known for its speed it comparison to Hadoop MapReduce, running programs up to 100x faster in memory or 10x faster on disk. For programmers, it can be used to quickly write applications in either Scala, Java, or Python. Spark SQL, another Spark component, is also available to easily extract data from sources, such as Apache Hive. Since Spark has named its as a general purpose big data platform, its easy to run in standalone mode locally, on YARN or Amazon's EC2, and it also reads from HDFS, Amazon's S3, and HBase. Industry technology leaders are leaning towards Spark MLlib due to its scalability, performance, user friendly APIs, and its other components [13].

The dataset itself is very interesting, since we are dealing with class imbalance and image data. Machine learning algorithms usually assume that the number of items per class is roughly similar, so approaches were researched in order to deal with this along with preprocessing the image data. As Bartosz Krawczyk explains in *Learning from Imbalanced Data: Open Challenges and Future Directions* [2], our systems that learn from imbalanced data often have a hard time overcoming these biases without an overly complex system. Over the past several years, ensemble methods are widely used in order to handle class imbalance due to its successful approaches of Boosting and Bagging. Since Random Forests train a set of decision trees, the model training phase and model use (prediction) phase of decision trees are possible targets for parallelization. Decision trees are "easier" to parallelize in comparison to many other classification models because of their tree structure [5].

Although accuracy is a very common metric for many traditional machine learning applications, in practice it is often observed that 0 is the value of the recall for the minority class. Due to this, other metrics need to be considered, such as a confusion matrix in order to see better performance on the minority class. Since accuracy may not be the best metric to determine the quality of the model, precision and recall were focused on in order to assess our models ability to beat the true baseline [3].

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

[1] `https://drive.google.com/drive/u/0/folders/1EJBgJFmp-FQf2czw9LGImoOhEO2OvOoo`

In order to improve accuracy and also receive the best sensitivity to the foreground and background classification, feature extraction was researched on image processing data as suggested by many researchers [6].

3 Methods

Various models were evaluated in order to receive the highest accuracy of predictions on the high resolution brain scans[2] such as Linear SVM, Nearest Neighbor, Decision Tree, Neural Net and AdaBoost (see figure in Appendices C). Different models were trained on a subset of the data: 100 random background samples and 100 random foreground samples. We observed that the decision trees, and the two ensemble methods were the ones that performed the best, so Random Forest become the focus since it is also available in MLlib. Random Forests is a well-known ensemble method that's used to build predictive classification models. The model creates an entire forest of random uncorrelated decision trees to arrive at the best possible answer. Random Forest looks to reduce a correlation issue, a limitation to bagging trees, through choosing a subsample of the feature space at each split by using a stopping criteria for node splits to prune the trees. Exploration of multiple parameter combinations were explored to achieve the best possible accuracy.

3.1 Data Analysis and Feature Extraction

To begin visualizing the data, we performed our analysis using Jupyter Notebooks, numpy, and matplotlib. To do so, we took one of the csv files, found the rows of the file that are labeled as 1, found the rows of the file that has 0 as the label, and generated a file with 100 "foreground" images and 100 "background" images (see Appendix A). Visualizing this subset of data increased our intuition on what were the important features to segregating samples from different classes.

Image pre-processing and feature extraction was performed with the hopes to reduce the sample size and increase accuracy of the image data as suggested. Consequently, this would also allow us to try more parameters as the training will be speed up (see Appendix A for the comparison of images of foreground sample and background sample). We looked at the distribution of the training data since we knew we were dealing with class imbalance on the validation/prediction data set (image 5) and found thresholds in order to avoid misleading accuracy metrics. The statistics on all of planes(xy, xz, yz) slices for the two images led us to identifying important features along with a threshold of 50 to tell the difference between a foreground and background image. By looking at the images for foreground and background, we noticed that there were some obvious features that were highly correlated to the label such as the center pixel value - the center pixel value is usually high in a foreground image, and usually low in a background image. From this we decided not to train complicated models on the full image but to use the following features and consequently the feature vector for all 3 planes XY, XZ, YZ:

- center pixel value
- average of a window of pixels around the center pixel (FFT slice mean)
- number of pixels with intensity greater than a threshold (50)

In the application pipeline (see Appendix D) *Feature Extraction* is our first job. It is in charge of reducing the dataset size from 6 GB per image to 65 MB per image. This is one job per image. This piece has been implemented as a map only job and has to be executed in every sample data (train, test, validation).

From our data analysis we performed a classification comparison [11] of several classifiers on a smaller sample of the dataset through cross validation to gain a better sense of the nature of the decision boundaries of different classifiers with respect to the dataset. In Appendix D, the classifier comparison can be viewed along with the classification accuracy on the test set in the lower right. The test set contained a balanced scenario with 100/100 samples, proving that the Random Forest had the best accuracy with a 60/40 train/test sample using the feature vector without parameter tuning. According to our previous research on dealing with imbalanced data, these results aligned.

[2] `https://drive.google.com/drive/u/0/folders/1EJBgJFmp-FQf2czw9LGImoOhEO2OvOoo`

3.2 Sampling

Using the standard way of training and testing a classification model, the labeled data is partitioned into 3 separate sets: training, validation and test data. Although uniform random partitioning into training and validation data often works well, for this particular data set it is not sufficient [4]. Assume we have two labeled records with centers (x,y,z) and (x+1, y+1, and z+1), in this case, uniform sampling could assign (x,y,z) to training and (x+1, y+1, and z+1) to the validation data. The two neighborhoods and labels of the two pixels are highly correlated. The model would overfit to the training data and give overly optimistic validation accuracy from the correlations. To ensure independent training and validation, partitioning by image is performed on images 1, 2, 3, 4, and 6.

3.3 Parameter Tuning

Hyperparameter optimization was performed in order to enhance our model in Spark through the use of ParamGridBuilder() in MLlib. Changes of parameters controlling partitioning affected performance and accuracy. The parameters that we tweaked and set in our model were the following:

- Maximum depth of tree: splits for all trees in the forest. Higher values can lead to overfitting which decreases accuracy and increases run time of the phases. We observed the number of tasks increasing when increasing the depth.
- Number of tress: automatically train trees until performance is maximized or specify the number of trees. We saw a correlation between increasing the number of trees and our performance in scaling.
- Maximum bins: increasing this allows the algorithm to consider more split candidates and make fine-grained split decisions but it increases computation and communication. We saw better results but longer run times by increasing. We observed an increase in metadata shuffle when increasing bins during the phase of Random Forest training.
- Impurity: gini was used since it does not require computing logarithmic functions like entropy (less expensive) and online reports had shown that this measure has a small effect on performance.

3.4 Parallel Processing

Our distributed parallel application pipeline (see Appendix D) runs on AWS EMR. First, preprocessing was 1 job per image. Next, pre-processing transforms 6GB images on 65MB image. The training images are fed into the Random Forest Model to train. The image is persisted, which is a job that shuffles the data depending on the parameters. The classification job has a persisted model and it receives only the image that we want to classify. Since we intentionally made our feature vector small to help with class imbalance and performance, we didnt see a lot of scaling during this phase (firing up more than 4 worker machines was not necessary).

4 Results

As we learned through our readings and research, accuracy is not the best metric to select the best model, as it does not attribute the right importance to the minority class with respect to the dataset. However, we did use it as a scoring metric since we had an available file with the true results of image 5 to compare to in order to see if we beat the baseline after complete analysis, exploration and implementation. We also used confusion matrices to better analysis the results as recommended [3].

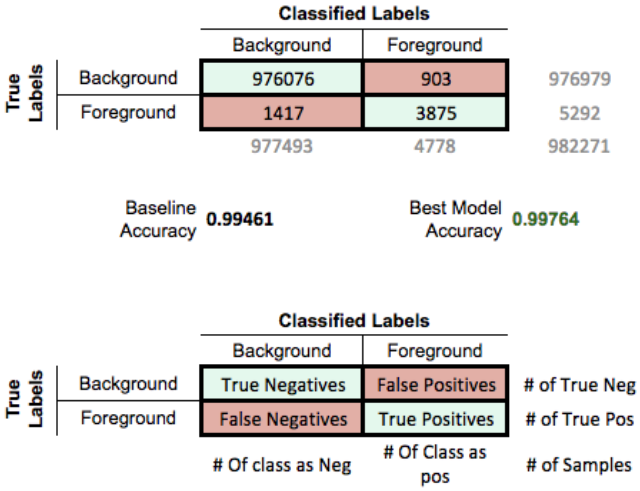
4.1 Parallel System

Table 1 shows the results of different Random Forest parameters on model accuracy. In our best model run on AWS, 10 machines with 40 partitions were used on the data in the training phase. In the prediction phase, only 4 machines were used. As we can see in the table below, increasing the depth, number of trees, and max bins increased the run time but also improved accuracy leading us to select depth of 15 and number of bins (discretization of the continuous features) as 512. When taking into account the number of trees, there was no gain beyond the point of 50 trees, for this reason, 50 was the selected metric.

Run time (mins)	Depth	Bins	Trees	Accuracy
8.47	3	256	50	0.99718
11.67	5	256	50	0.99730
30.50	10	256	50	0.99759
69.80	15	256	50	0.99761
27.77	10	32	50	0.99703
27.83	10	64	50	0.99731
29.50	10	128	50	0.99747
33.87	10	512	50	0.99760
5.80	4	256	25	0.99729
8.47	4	256	50	0.99725
30.23	4	256	100	0.99724
22.8	15	512	50	0.99768

Table 1: Parameters Explored for Random Forest

The Confusion matrix shows comparison between proposed baseline and best model accuracy. As it can be observed by the bold numbers in black and green, for the current testing set (the entire image 2 sample data), our model beats the baseline.



The running time and speed up results for the model training and prediction phase on a single machine and Apache Spark are show below. The number of trees and max depth are kept consistent, 50 and 15 respectively, seeing that these numbers resulted in the best accuracy out of our runs.

4.1.1 Model Training

The following table shows that there is a a good speedup on running time when we scaled from 2 workers to 10 worker machines. Taking in consideration that 50 decision trees are being trained in the selected model, and that MLLib implements parallel tasks based on the number of trees and splits of data, it makes sense that the training job scales well with the increase of the number of available cores (workers* core/worker) up to the number of trees. But, beyond this point, the speedup gain stagnates which can be observed by the running time values of 10 (40 cores) and 19 workers (76 cores).

$$Speedup(x, y) = \text{running time on } x \text{ machines / running time on } y \text{ machines}$$
$$Speedup(3, 10) = 5.02$$
$$Speedup(3, 19) = 5.58$$

Workers	Running Time (seconds)
2	2924
10	582
19	524

4.1.2 Model Prediction

The table bellow show the running time for classification of the validation dataset with respect to the number of machines. During this job we have set the number of partitions to be equal to the number of available cores on our system. It is interesting to note that this job scales well until we reach 4 worker machines (16 cores). After this point, the application does not scale well and plateaus at roughly 2.8 speedup if compared to the running time of 1 machine. Inspecting the possible causes indicate that there is a non-negligible overhead when splitting and and using chunks of data smaller then $65/(4 \times 4) = 4$MB, which corresponds to:

$$\text{ChunkSize} = \text{ValidationDataSize}/(\text{Number Of Workers} \times \text{Number Of Cores})$$

Workers	Running Time (seconds)
1	240
2	160
4	88
8	92
10	82

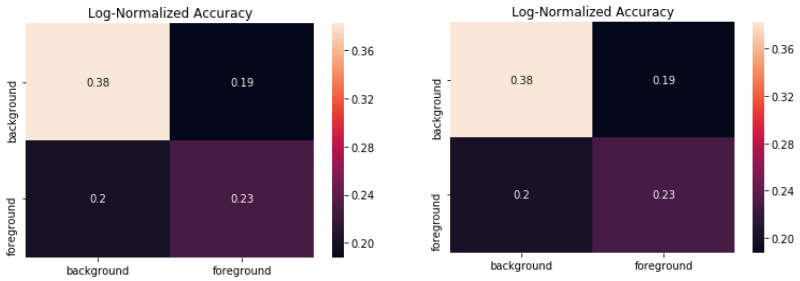
$$Speedup(x, y) = \text{running time on } x \text{ machines / running time on } y \text{ machines}$$
$$Speedup(1, 4) = 2.727$$
$$Speedup(1, 10) = 2.92$$

4.2 Single Machine

Small scale machine learning proved to be possible on a commodity laptop, given preprocessed data that had already undergone dimensionality reduction. Utilizing SciKit-Learn, a python framework, we were able to train a random forest model on this preprocessed dataset with comparable accuracy. Such results can be seen tabulated below. The two confusion matrices are the local runs of Random Forest using the impurity gini and entropy respectively. We were anticipating the run with gini impurity to be faster due to the lower computation cost, but our results showed nothing overly interesting. The accuracy was very slightly worst (.001%), and this could be due to not having a maxbins parameter like there is in MLlib.

Model	Run time (mins)	Accuracy
Random Forest	9.8	.99767

Table 2: Comparison of Models on Single Machine using Preprocessed Dataset



When attempting to perform fitting on the full dataset using sklearn the Jupyter notebook (the environment we were running in) consistently crashed, and never could make forward progress on training. It's easy to speculate that the crashing was due to us trying to load an excessive amount of data into memory, which caused Jupyter to crash. Running natively in a single python process, we found still that it was impractical to load the entire dataset at once, as by the time it was loaded a significant amount of time had passed and in that time there was a nonzero chance that something could interrupt training (laptop dying, hibernating, or crashing).

To be able to train a large classifier without blowing out the RAM capacity of our machine, we leveraged the ensemble nature of the random forest classifier and developed a tool on top of sklearn to partition the dataset and create separate files for each tree in the forest, trained each tree separately, then assembled them into an ensemble classifier by manipulating some python internals of sklearn. This approach differed from the intrinsic behavior of sklearn by constraining the size of the active set of RAM on the machine, having only one active dataset and tree at any time. Our approach vigorously checkpoints, leaving little to be lost in case of a fault, but we don't believe that this approach is generalizable to all approaches and algorithms (and requires annoying manual instrumentation) so your mileage may vary.

4.3 Parallel Environment vs. Single Machine

When comparing the run times of the parallel environment vs the single machine environment on the preprocessed dataset that was only 65MB per image, the single machine that had 32GB of RAM performed nearly twice as fast as the parallelized Spark environment as seen in the prior tables. In this case, we can see that utilizing a more complex environment on a smaller dataset can cause unnecessary overhead and data shuffling which leads to longer execution times.

Model	Run time (mins)	# Trees	Max Depth
Random Forest (Single Machine)	75.8	50	15
Random Forest (Parallel - 19 Machines)	33.4	50	15

Table 3: Full Dataset Execution Comparison (Model Training Phase)

Shown in the table above, when utilizing the entire data set that contained over 3000 features, the parallelized environment significantly out performs the single machine run when increasing the number of machines to 19. With the ability to run tasks in parallel, this is an invaluable benefit of our parallel environment with massive datasets.

5 Conclusion

The purpose and motivation of our project was to take an interesting and complex image data set to perform the following: data analysis, feature extraction/pre-processing, parameter tuning, running and configuring an Apache Spark application in scala on AWS EMR, selecting and implementing an appropriate model, and analyzing the execution times between single machine runs and parallel runs. Datasets that suffer from class imbalance often result in bias or overly complex models, or made it nearly impossible to beat baseline accuracies of over 99%. Our approach followed a mixture of recommended methods and ultimately beat the baseline of the validation data (image 5).

Our experiments in training on a laptop locally found that for small datasets (in both number of samples and features per sample) it can be more practical to train locally for 5 10 minutes than to offload to a EMR cluster, wait for it to come online, transfer over data, compute, and download results. However, dimensionality reduction may not always be as accurate as when applied to this particular dataset and may produce suboptimal results, so millage may vary on this approach. However, if it can be created having a feature reduced dataset available for quick iteration on approaches and algorithm tuning can be monumentally useful for tuning an algorithm in the field, and for demonstrating approaches to others.

References

[1] Map-Reduce for Machine Learning on Multicore,
http://www.andrewng.org/portfolio/map-reduce-for-machine-learning-on-multicore/

[2] Bartosz Krawczyk. *Learning from imbalanced data: open challenges and future directions*. November 2016, Volume 5, Issue 4, pp 221232

[3] On the Class Imbalance Problem,
http://sci2s.ugr.es/keel/pdf/specific/congreso/guo_on_2008.pdf

[4] Learning from Imbalanced Data,
https://www.cs.utah.edu/ piyush/teaching/ImbalancedLearning.pdf

[5] A Parallel Random Forest Algorithm for Big Data in a Spark Cloud Computing Environment,
https://ieeexplore.ieee.org/document/7557062/

[6] A Detailed Review of Feature Extraction in Image Processing Systems,
https://ieeexplore.ieee.org/document/6783417/

[7] Riedewald, Mirek. *Ensemble Models*. April 2018, Northeastern University.

[8] Apache Spark - Amazon EMR,
https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark.html

[9] Apache Spark Developer Cheat Sheet,
https://mapr.com/ebooks/spark/apache-spark-cheat-sheet.html

[10] Cross Validator Model,
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-mllib/spark-mllib-CrossValidato

[11] Classifier Comparison,
http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html

[12] Classification by using Ensembles of Classifiers,
https://grzegorzgajda.gitbooks.io/spark-examples/content/classification/rf-classification.html

[13] MLlib: Scalable Machine Learning on Spark
https://web.stanford.edu/ rezab/sparkworkshop/slides/xiangrui.pdf

[14] Ensembles - RDD-based API
https://spark.apache.org/docs/latest/mllib-ensembles.html

[15] EMR Add Steps
https://docs.aws.amazon.com/cli/latest/reference/emr/add-steps.html

[16] Decision Tree
https://spark.apache.org/docs/2.2.0/mllib-decision-tree.html

[17] Random Forest Classifier
http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[18] Fast Fourier Transform
https://en.m.wikipedia.org/wiki/Fast_Fourier_transform

[19] Spark
http://spark.apache.org/

[20] MLlib
https://spark.apache.org/mllib/

[21] Matei Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. Association for Computing Machinery., 2014

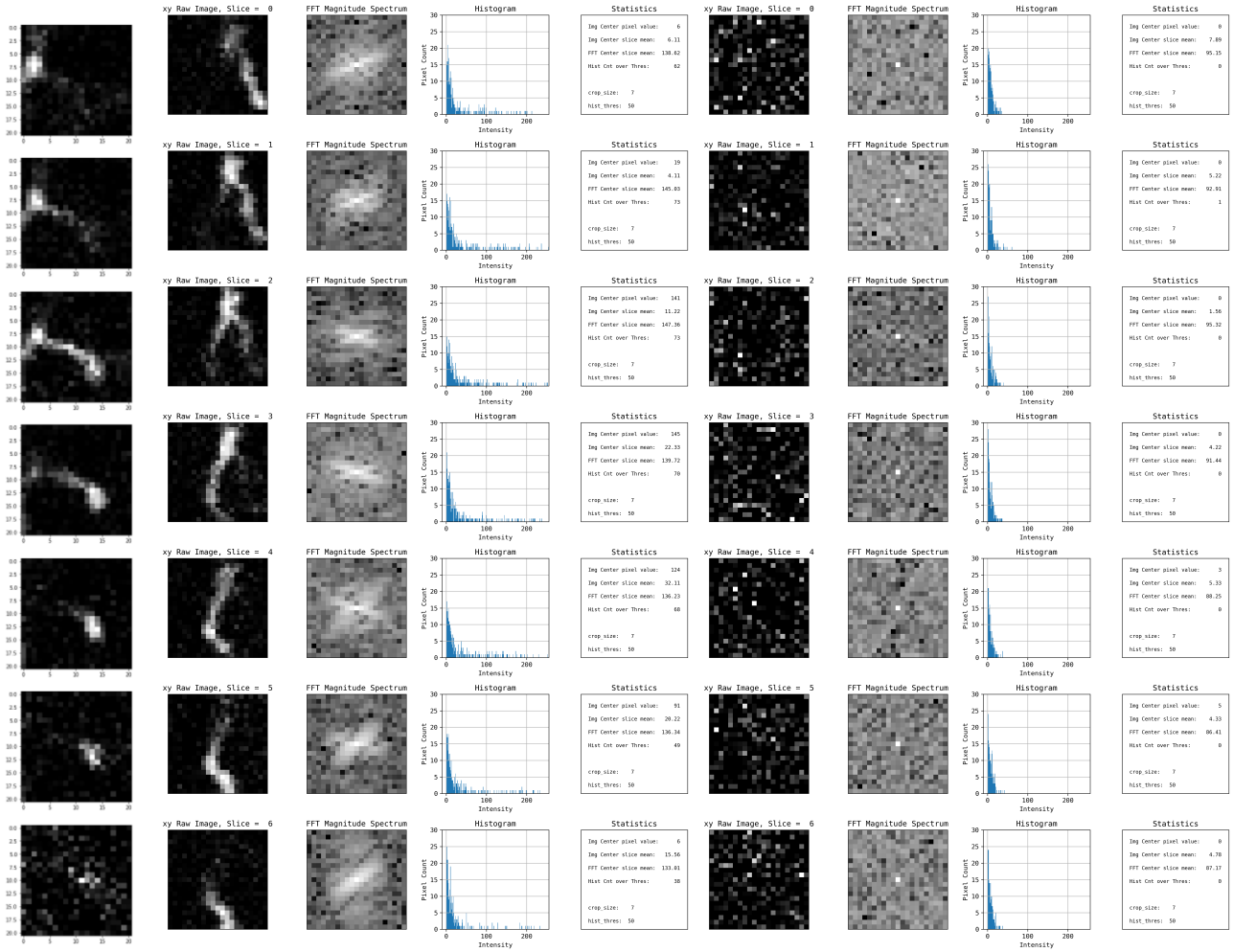
[22] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media Inc., 2015

[23] Petar Zecevic and Marko Bonaci. *Spark in Action*. Manning Publications., 2016

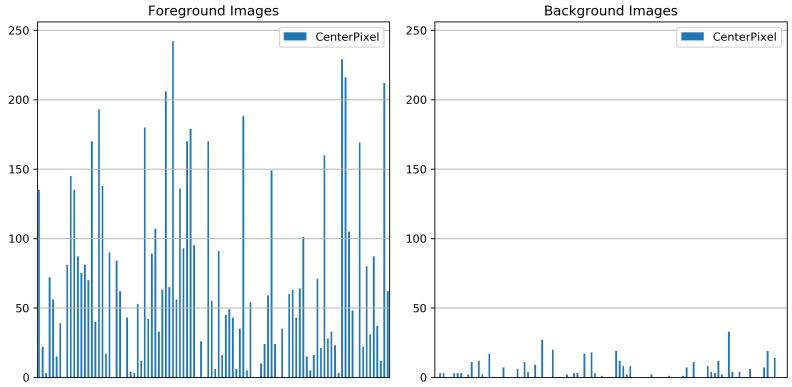
[24] Spark Programming Guide
`http://spark.apache.org/docs/latest/programming-guide.html`

Appendices

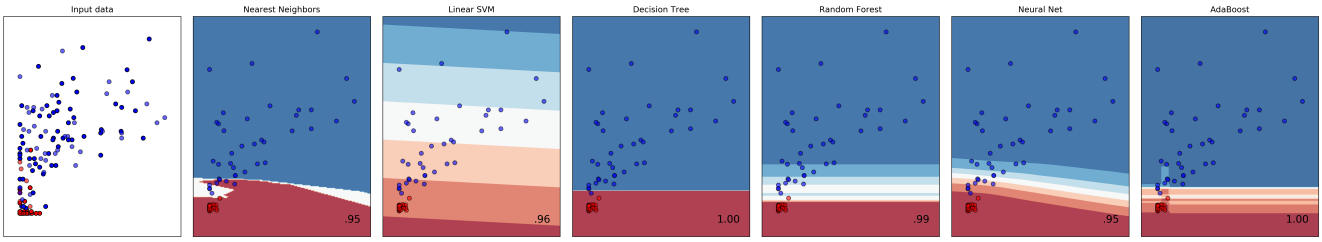
A Visualizing the Input, Foreground And Background Analysis



B Center Pixel - foreground and background



C Classification Comparison test



D Application Pipeline