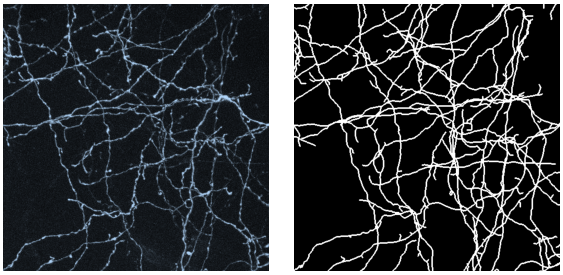
Northeastern University
EECE 5644 – Pattern Recognition and Machine Learning
April 2018, Asha Chen-Phang, Emily Dutile, Nate Otenti, Tristan Sweeney

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 and using well-known industry solution for faster computation, the group used Apache Spark, 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, implemented Random Forest in python using sklearn in a single-processor solution, and implemented Random Forest in scala on top of Apache Spark using MLlib to analyze time and processing efficiency in a parallel processing program upon distributed data. The Apache Spark environment was created on an Amazon EMR cluster, leveraging the EMR file system (EMRFS) to access data in Amazon S3.

We worked with a dataset with the interest of 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. For better intuition the 2-dimensional projection on the X-Y plane is shown. As you can see, the image is noisy but the axons, which are the lines going across the image, are clearly visible. To improve the quality of algorithms that automatically trace these axons in an image, we classified each pixel as foreground (belongs to an axon) or background (does not belong to an axon). The traced data looks like the figure on the right, where white indicates foreground and black indicates background.



Using the manually traced image, the labeled data was generated as follows:

- Select a pixel (i, j, k) in the image.
- 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.

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), n[4] of (9, 10, 10), and so on. Neighborhoods of size 21x21x7 were recommended by domain experts, which is what the labeled data sets contain.

1.1 The Data

The dataset[1] is composed of several labeled csv files, each consisting of rows that contain an input vector of 21x21x7 brightness values (intensity) from a 3D image, together with the center pixel's foreground-vs-background label. The

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

last value is the label for the whole image. On each line there are 21*21*7+1 values. Each image, which is a csv file, is roughly 6.5 GB. 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 99.99% background pixel. 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.

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. Uniform random partitioning into training and validation data often works well, but there are datasets, such as this one, where it is not sufficient. Assume we have two labeled records with centers (x,y,z) and (x+1, y+1, and z+1). 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 needed.

2 Related Works

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. The model that gave us the best results was Random Forest. 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 was explored to achieve the best possible accuracy.

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.

3.1 Preprocessing and Feature Extraction

Image pre-processing and feature extraction was performed (see Appendix A for the comparison of images of foreground sample and background sample). We looked at the distribution of the data since we knew we were dealing with class imbalance 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 to tell the difference between a foreground and 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:

- center pixel value
- average of a window of pixels around the center pixel
- number of pixels with intensity greater than a threshold

In the application Pipeline (see Appendix C) *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 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 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. The feature vector was made up of 10 features and the label.

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

```
# Feature Vector
CenterPixel,...
  xyImgSliceMean,xyFFTSliceMean,xyCntOverThres,...
  xzImgSliceMean,xzFFTSliceMean,xzCntOverThres,...
  yzImgSliceMean,yzFFTSliceMean,yzCntOverThres,...
  label
```

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.

3.2 Parameter Tuning

Hyperparameter optimization was performed. 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.

3.3 Computation and Partitioning

4 Results

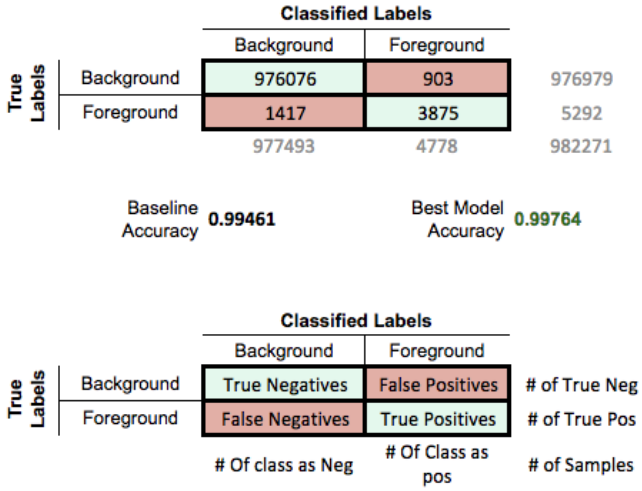
Accuracy did not seem to be the best metric to select the best model, as it does not attribute the right importance to the minority class with respect to the dataset, but we did use it as a scoring metric. Table 1 shows the results of different Random Forest parameters on model accuracy. In our best 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.

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

As we can see in the table above, 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.



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.

Model	Run time (mins)	Accuracy
Random Forest	22.8	.99767
SVM	0	0
KNN	0	0

Table 2: Accuracy Comparison of Sequential vs. Parallel Environment

4 Performance

The running time and speed up results for the model training and prediction phase on a single machine and Apache Spark are show below:

4.1 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).

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

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

4.2 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 plateau at 2.80 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 $65/(4 \times 4) = 4MB$, which corresponds to:

$$ChunkSize = ValidationDataSize/(NofWorkers \times NofCores)$$

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

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

5 Conclusion

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

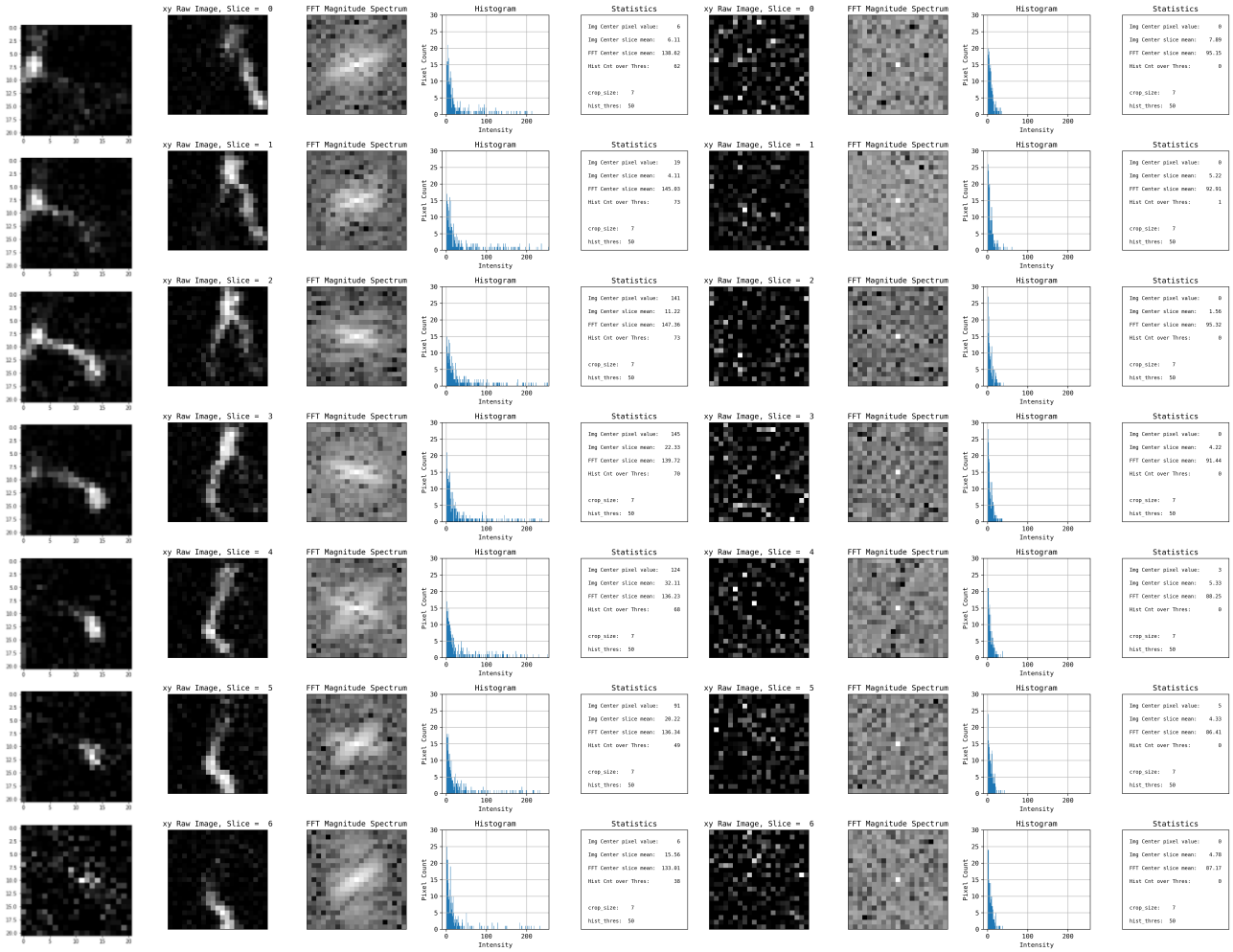
[15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media Inc., 2015

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

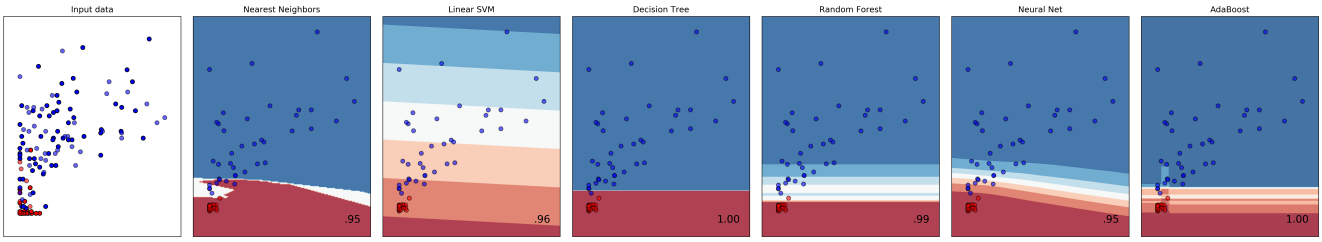
[17] Spark Programming Guide
http://spark.apache.org/docs/latest/programming-guide.html

Appendices

A Visualizing the Input, Foreground And Background Analysis



B Classification Comparison test



C Application Pipeline