Chest X-Ray Analysis: CSE6250 - Fall 2019

Kelly "Scott" Sims, Dane Smith, Joshua Harris

Demo

https://www.youtube.com/watch?v=cThZBXj18D8

Code

https://harrisjosh-project-data.s3.amazonaws.com/cse6250/CSE6250.tar.gz **(public through Dec 16, 2019)**

https://github.com/mooseburger1/cse6250 **(private, request access from Scott Sims)**

Keys - *The following keys give access to the S3 buckets mentioned where referenced augmented datasets, tfrecords, models, and outputs are stored. Please email Scott Sims (ksims35@gatech.edu) or Joshua Harris (joshuaharris@gatech.edu) with questions. Instructions provided in READMEs within code zip file above.*

AWS ID: `AKIAWVFQFSQH7O5PHHMJ` **AWS Secret:** `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`

Abstract

Using our proposed "Code of Conduct" model, we attempt to show utilizing transfer learning can perform better on automated chest X-Ray interpretation, facilitated by 14 Individual autoencoders. The autoencoders aid the model in "cheating" to improved results.

Introduction

As healthcare advances and begins to adopt new technologies to improve patient care, following best practices in big data analytics and machine learning becomes more important. Applying deep learning and computer vision techniques to create classification systems for analysis of medical records, specifically medical images like X-Ray, promise to increase in efficiency for medical systems. Augmenting expert analysis with automating suggested classification and labeling of medical images can increase the throughput of radiologists, reduce errors, and provide additional support to medical staff.

Hospitals, healthcare professionals, insurance agencies, and researchers are all driving adoption of big data analytics and its application in the American healthcare system .[8]

Approach and Metrics

CheXpert is a large public dataset for chest radiograph interpretation, consisting of 224,316 chest radiographs of 65,240 patients labeled for the presence of 14 observations as positive, negative, or uncertain .[1] The CheXpert data set will be our primary dataset for this experiment. The CheXpert team provides a full dataset of two hundred thousand radiographs but also provides a sample dataset with a smaller number of images, which we are utilizing for our initial experiments and metrics.

We chose to use Amazon Web Services (AWS) for the project due to overall team experience on the platform, its strength and relative dominance in private industry, and interest in learning more about the technologies available to us via AWS. The original CheXpert dataset was downloaded to Simple Storage Service (S3) and we deployed our data pipeline on an Elastic Map Reduce (EMR) cluster with one master node and five worker nodes running Hadoop and Spark.

Following best practices for big data projects, we performed some exploratory data analysis on the limited dataset prior to writing or deploying any code. Table 1 shows the results of our data analysis on the limited dataset, and this information provided us with some valuable estimates for the resources required for the processing pipeline. Specifically, the initial analysis allowed us to see how to split the data when doing augmentation. We chose to

process and augment all images at once in a distributed process, but collect and write processed data on a per classification basis. This decision was made to limit the amount of resources we needed to provision on Elastic Compute Cloud (EC2), AWS' compute service.

Classification	Only Occurrence	Total Occurrences	Unsure Occurrences
No Finding	6.03% (13453)	10.02% (22381)	0.00% (0)
Enlarged Cardiomediastinum	0.44% (979)	4.83% (10798)	5.55% (12403)
Cardiomegaly	0.66% (1464)	12.09% (27000)	3.62% (8087)
Lung Opacity	1.08% (2420)	47.26% (105581)	2.51% (5598)
Lung Lesion	0.33% (738)	4.11% (9186)	0.67% (1488)
Edema	1.09% (2440)	23.39% (52246)	5.81% (12984)
Consolidation	0.30% (663)	6.62% (14783)	12.42% (27742)
Pneumonia	0.20% (455)	2.70% (6039)	8.40% (18770)
Atelectasis	0.62% (1375)	14.94% (33376)	15.10% (33739)
Pneumothorax	1.08% (2424)	8.70% (19448)	1.41% (3145)
Pleural Effusion	1.53% (3411)	38.58% (86187)	5.20% (11628)
Pleural Other	0.13% (286)	1.58% (3523)	1.19% (2653)
Fracture	0.67% (1492)	4.05% (9040)	0.29% (642)
Support Devices	0.81% (1817)	51.92% (116001)	0.48% (1079)

The images in the dataset varied in size, with the heights of all images ranging from 280 to 350 pixels, and the width of all images ranging from 270 to 320 pixels. This required us to add a standardization step to the processing pipeline.

The original data from CheXpert was structured in two directories, a training directory and a valid directory. We needed a different structuring system to train our model. Our desired format is Train -> Classification -> Images & Valid -> Classification -> Images. But Train and Valid directories are contained in a parent directory "Data" on S3. Making this change in storage structure required adding logic in our preprocessing to correctly identify images to their respective classes, preprocess them, and write them to our desired directory structure.

Table 1. Analysis of Classifications of Small CheXpert Dataset

Our Code of Conduct Model. Transfer learning is the entrypoint of our model. Due to the specificity of the target images, only the lower layers of a pretrained Inception-ResNet-v2 model are needed and all weights will be frozen. Our own architecture will be built and trained to succeed the pretrained model. In conjunction with the pretrained model, semi-unsupervised Autoencoders will be trained; one for each classification.

Each image is labeled in the dataset, and individual Autoencoders(AE) can be trained with respect to each class and used as an anomaly detector. For example, training an AE solely on pneumonia cases creates a model that has learned the identity function for pneumonia X-rays. When a non-pneumonia image is fed through the model, it should fail to respectably recreate the image, meaning the identity function has failed, and anomaly is detected. By creating an AE for each classification, we can create a vector of the output from every AE for a given image. This vector will be used in later layers of the ultimate network.

Finally, a hybrid RNN will be trained as the final layers of the pretrained Inception-ResNet model. The high level idea of a Neural Network is that it learns underlying patterns and features represented in the data by itself. Our model will violate this "code of conduct" by utilizing outside sources of information during its training process (the vectorized output from the Anomaly detection AutoEncoders). In a traditional RNN, the output of a given cell at time (h-1) is fed back as an input to the cell with the incoming feed forward data at time (h). As a consequence, it is learning features and patterns from the data with respect to information learned from the data at time (h-1). Our model aims to augment this workflow, and help the model cheat towards the proper classification. Instead of creating this recurrent feedback loop, we will feed the AutoEncoder output into the layers of the NN and train a second set of weights within a given cell. We hope that his "Information Doping" facilitates the model becoming a great classifier that generalizes perfectly to unseen data.

Our model will be gauged on AUC for overall performance, with an emphasis on the recall metric. One could argue that the cost of missing a diagnosis is more detrimental than the cost of incorrectly diagnosing a patient. By improving recall, we are improving outcome on the question "What proportion of actual positives were identified correctly?" As far as we can tell in initial research, the attempt of model cheating with information doping has not been attempted, and our model is unique in this regard.

Implementation

The final architecture of the project includes four distinct phases: data preprocessing and augmentation, data pipelining, creation of autoencoder models, and the creation of our Code of Conduct model. We explore each step of the process below.

Preprocessing and Data Augmentation. Prior to completing the code of conduct model using Inception-ResNet, we had to complete our data processing pipeline to properly train the model. In Appendix 1, we show a diagram of the data processing pipeline architecture, and discuss the steps involved in running it.

The data processing pipeline uses PySpark and OpenCV to complete its work. After initially filtering the data on S3 to a specific classification, each image undergoes contrast limited adaptive histogram equalization, standardization, and finally data augmentation prior to being saved back to S3.

The first step of the process takes an individual image from the filtered dataset, and passes the image as a byte array to a function that performs an adaptive histogram equalization, specifically Contrast Limited AHE (CLAHE), using OpenCV.

After the CLAHE completes, we augment the image by flipping horizontally, vertically, horizontally & vertically, and standardize the size of the images at 299x299 for use with Inception-ResNet-v2. These images are then re-encoded to a JPEG byte string and saved back to S3 using the boto library from Python. We explain the reason for using the boto library below. After the process completes for one classification we use the results in Table 1 to check the size of our expected augmented data set against the results of the pipeline. After augmenting the original dataset, the total size increases from 439GB to 1.6TB.

Our learnings and the observations below came from overcoming challenges to achieve our desired results while creating the preprocessing pipeline. PySpark's implementation of image handling and the image format is comparatively new to the Spark ecosystem, so the current documentation covers only part of what we need. However, even with its limitations for image processing, in order to efficiently process a large number of images as we need to for the CheXpert dataset, we decided to pursue a distributed processing model in Spark.

The ImageSchema produced by Spark produces a structure similar to OpenCV's approach to processing an image, however the byte array data from the image also needed to be converted to a numpy array in order to perform our CLAHE (histogram equalization) and image standardization. Because of the lack of documentation around Spark's image handling and ImageSchema specifically, performing this conversion required a few days of research and experimentation, but eventually we came to an acceptable solution.

After successfully converting the images from ImageSchema data byte array format to a numpy array, there were fewer issues implementing the CLAHE histogram equalization and image standardization.

However, after augmenting and standardizing the image data, saving the images to S3 after processing completed posed a problem. Once again, the lack of documentation around Spark's image handling, saving the images the way we wanted to required a few days of research and experimentation, including reaching out to Spark's open source support team as well as Stack Overflow to eventually come to the solution we implemented in our processing pipeline.

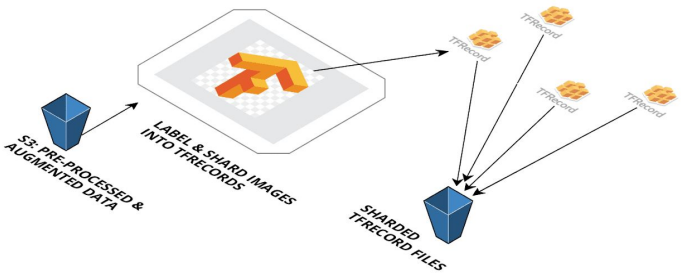
Our initial approach, using *spark.write.format("")* to save the data to S3, did not work because Spark does not provide this method in the image processing API. Eventually we explored using the intended class to save images, *org.apache.spark.ml.source.image.PatchedImage*, to pass the augmented and standardized images and save to S3, but this also did not work. We raised the issue with Spark's open source team and are waiting to see if we need to open an issue on the open source library itself. At this time, we had to compromise and pursue a different path forward to complete the processing pipeline on schedule. Making this compromise allowed us to begin running the pipeline on the limited dataset to explore options for how best to process the entire dataset in Spark.

Because of the challenges we faced, and the current limitations of Spark's distributed processing of images, our final implementation used the well documented boto library for Python and AWS. This method requires collecting the images to driver first, serializing them as an OpenCV encoded byte stream in jpg format, and finally writing the file to S3. Because the full CheXpert dataset is ~500 GBs of images, this approach requires collecting a multiple of 500GBs to the drive due to data augmentation. While this is not ideal, it was a necessary compromise we accepted due to the time limitations on the project.

The final implementation of the processing pipeline filters the full dataset down to a single classification in order to limit the amount of data necessary to store on the drive. Spark collects all images associated with a unique classification, processes them using the methods described above, and uploads the resulting images to S3, cleaning up memory after all the images complete processing. The pipeline then collects all images associated with the next unique classification. This process is iteratively executed until all images are written back to S3 after preprocessing. Again, this was necessary due to the experimental status of Spark image handling.

Processing the smaller dataset in this way took longer than anticipated, so for the full dataset we'll be adding more worker nodes to our EMR cluster. The architecture of the processing pipeline will remain the same.

Data Pipeline and TfRecords. It is necessary to increase the efficiency and processing of the images in the dataset due to the size of the augmented dataset after processing. We use Tensorflow's binary data format, TensorRecords, to increase efficiency. TFRecords increase efficiency because of their binary data format, which is smaller than other formats. These records can also be read more efficiently by our models, and take less time to copy than other formats. We can also utilize Google's protocol buffer when using TFRecord format, another boost to efficiency in training our autoencoders.



Following best practices for data pipelining, we avoided sequential processing of the next batch before loading onto the GPU, we also wanted the CPU to be processing in parallel with the GPU so when the next batch was ready to be loaded on the GPU, there was no idle time.

Figure 1. Architecture Diagram of our data pipeline and TFRecord creation process

During the creation of our TFRecords, the images get labeled with their classification, image data, and the class name of the record. After combining multiple images into sharded TFRecord files, we saved the processed records back to S3 to feed into the autoencoders for processing and training in the next step of the process.

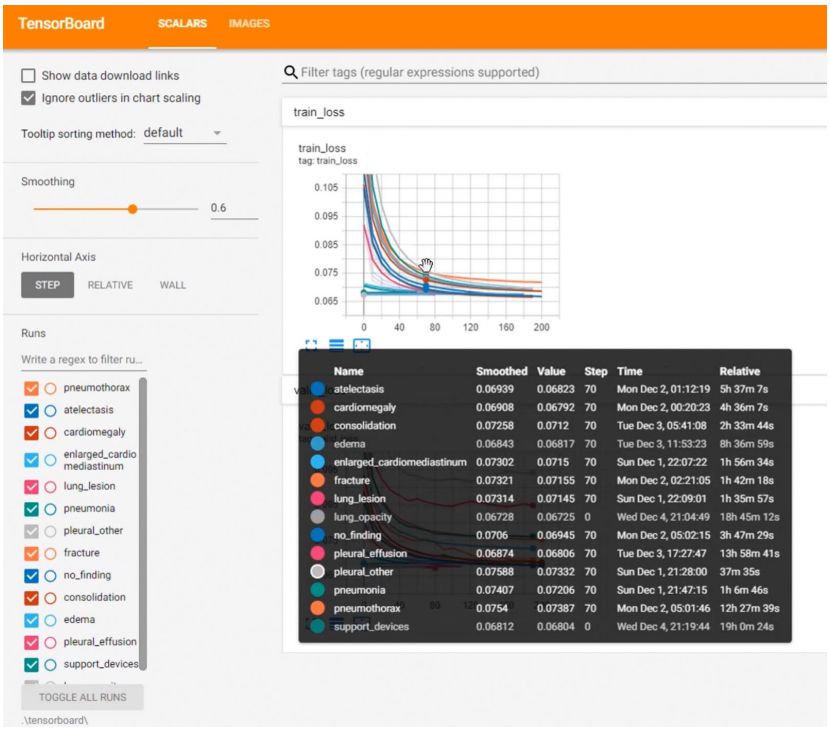
We experienced fewer challenges implementing the data pipeline as best practices are well established and the libraries are well documented. Our final implementation included a command line interface which allowed execution of the tfrecord creation with configuration options for input and output directories on S3, classification names to be processed and sharded, and number of shards for the dataset. Larger datasets like edema required more shards to ensure the resilience of the process in case of failure.

AutoEncoders. An autoencoder is a model that attempts to learn the identity function of an input image by extracting compressed features of that image and from those compressed features the image is attempted to be reconstructed.

Creating a Tensorflow model for each autoencoder, we trained and validated performance over two hundred epochs per classification, saving resulting models for use in our final Code of Conduct model.

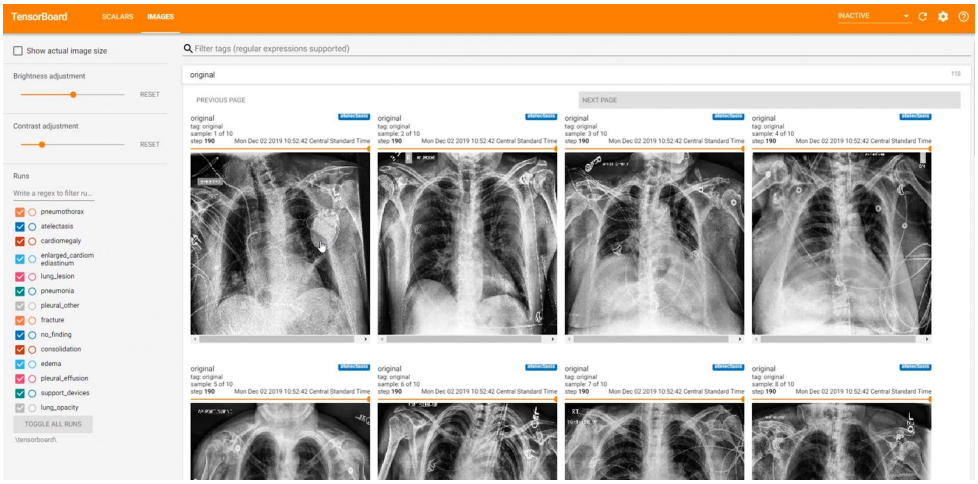
Visualizing the progress of the models via Tensorboard allowed our team to monitor the autoencoder recreation of the image (Figure 3) as epochs progressed.

The autoencoder models trained on AWS EC2 instances with CUDA enabled GPU acceleration. Instances in this p-series from AWS are specifically designed for this kind of model creation in production machine learning environments.



Because of time constraints on the project, we faced difficult choices balancing resource and time constraints. Training individual autoencoders were often long running processes (Table 2 below) requiring up to a day to complete each. In order to complete autoencoders on time we spun up multiple VMs simultaneously to train in parallel, but the result was increased cost due to our increased utilization of AWS resources. At the end of the process, training autoencoders took about two weeks and cost just over $500 in compute costs and data transfer fees.

As a result, we trained autoencoders for only 200 epochs before moving on to the implementation of our final code of conduct model.



Figures 2-3. AutoEncoder training progress was tracked via logs and Tensorboard

The Code of Conduct Model. While waiting for the autoencoders to complete training, we downloaded the Inception-ResNet-v2 model began working on our Code of Conduct model utilizing transfer learning and the final autoencoders produced in our previous step. By chopping off the final layer of the Inception-ResNet-v2, we added in our autoencoders, providing a "cheat sheet" for our final model to reference as the input came into the model. The image input passes into all fourteen autoencoders as well as Inception-Resnet-v2 and comes out with fifteen results, the lowest error autoencoder providing a hint to the model as to which classification might be its best final guess.

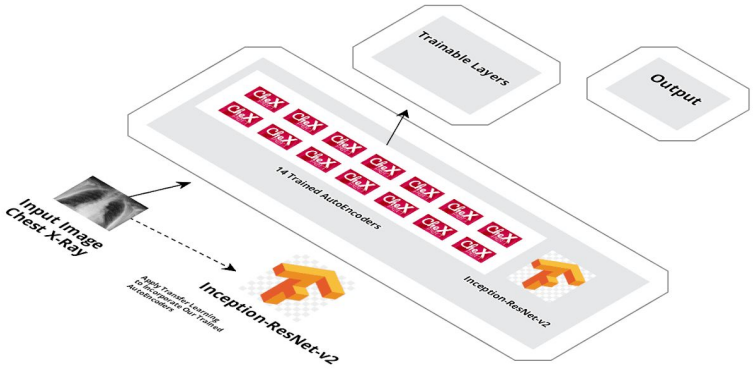
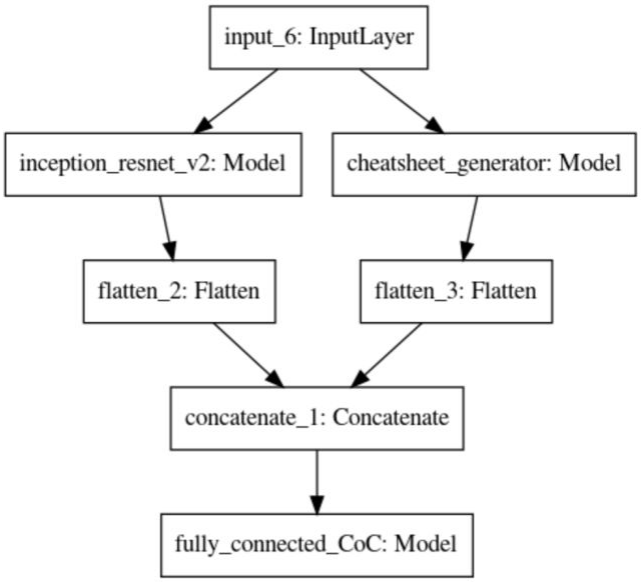


Figure 4. Our Code of Conduct model architecture

In our first iteration of the Code of Conduct model, epochs take between twelve and fourteen hours to run.

Our Experimental Results section is based on the first two epochs run against our Code of Conduct model, and we hope to see increased accuracy over time as we get more training epochs completed. Our primary limitation for initial findings is time, and we will continue running training epochs after the project deadline is complete to attempt to push for better results.



Model: "Code of Conduct Model"

Layer (type)	Output Shape	Param #	Connected to
input_6 (InputLayer)	[(None, 299, 299, 3)	0	
inception_resnet_v2 (Model)	(None, 8, 8, 1536)	54336736	input_6[0][0]
cheatsheet_generator (Model)	(None, 299, 299, 42)	195193642	input_6[0][0]
flatten_2 (Flatten)	(None, 98304)	0	inception_resnet_v2[1][0]
flatten_3 (Flatten)	(None, 3754842)	0	cheatsheet_generator[1][0]
concatenate_1 (Concatenate)	(None, 3853146)	0	flatten_2[0][0] flatten_3[0][0]
fully_connected_CoC (Model)	(None, 14)	192668214	concatenate_1[0][0]

Total params: 442,198,592
Trainable params: 0
Non-trainable params: 442,198,592

Figures 5-6. Code of Conduct model diagram

Experimental Results

The initial goals set our in our project proposal timeline for this point of the project included procuring the dataset, performing exploratory data analysis on the dataset, completing the data pipeline, creating autoencoders, and creating our Code of Conduct model. We hit these milestones, but because of time constraints feel we could easily improve the performance in each area given more time.

Our experimental results are focused on two areas: the autoencoder results and the final Code of Conduct model results. As mentioned above, the autoencoder results are based on 200 epochs and the primary limitation on the performance of both was time. Money was also a consideration but our team will continue to pursue improvements at additional cost after the project deadline. The final model performance is limited to the two training epochs of we were able to complete at this time.

Results of the Autoencoders. The Autoencoders took between two and thirty nine hours to complete two hundred epochs. The results below show each classification's time to complete 200 epochs, and resulting MSE after 200 epochs. We also show the result on the valid training set for each AE.

Classification	Time to Train 200 Epochs	MSE	**Valid Training Results (MSE)**
Atelectasis	15 hours 17 minutes	0.06697	**0.06802**
Cardiomegaly	12 hours 30 minutes	0.06653	**0.06812**
Consolidation	7 hours 22 minutes	0.06864	**0.06875**
Edema	23 hours 3 minutes	0.06759	**0.06853**
Enlarged Cardiomediastinum	5 hours 13 minutes	0.06859	**0.07279**
Fracture	4 hours 31 minutes	0.06885	**0.06939**
Lung Lesion	4 hours 20 minutes	0.0688	**0.0849**
Lung Opacity	18 hours 45 minutes	0.06728	**0.06742**
No Finding	10 hours 50 minutes	0.06674	**0.06792**
Pleural Effusion	1 day 14 hours 56 minutes	0.06739	**0.06729**
Pleural Other	1 hour 41 minutes	0.06965	**0.0942**
Pneumonia	3 hours 2 minutes	0.06901	**0.0761**
Pneumothorax	18 hours 41 minutes	0.07175	**0.07536**
Support Devices	19 hours	0.06812	**0.06843**

Our final Code of Conduct model completed two epochs over the final 48 hour window for the project. At the time of this writing, after two epochs we have a loss of 1.26 (categorical cross entropy loss) and an accuracy of 74%. While this is somewhat disappointing, it is still very early in the training process.

Conclusion

The project's scope was large and ambitious, and we completed our initial goals for completing the original proposed architecture code and executing our proposed model, if only for a limited amount of training. However, we did not accomplish all we wanted to regarding accuracy, especially compared with CheXpert's v1 results. We

believe it is too early to decisively say if our model is a success or failure, and we will continue to iterate and improve on it over time before submitting to the CheXpert leaderboard for external evaluation.

Team Contributions

Each team member made significant contributions to the project.

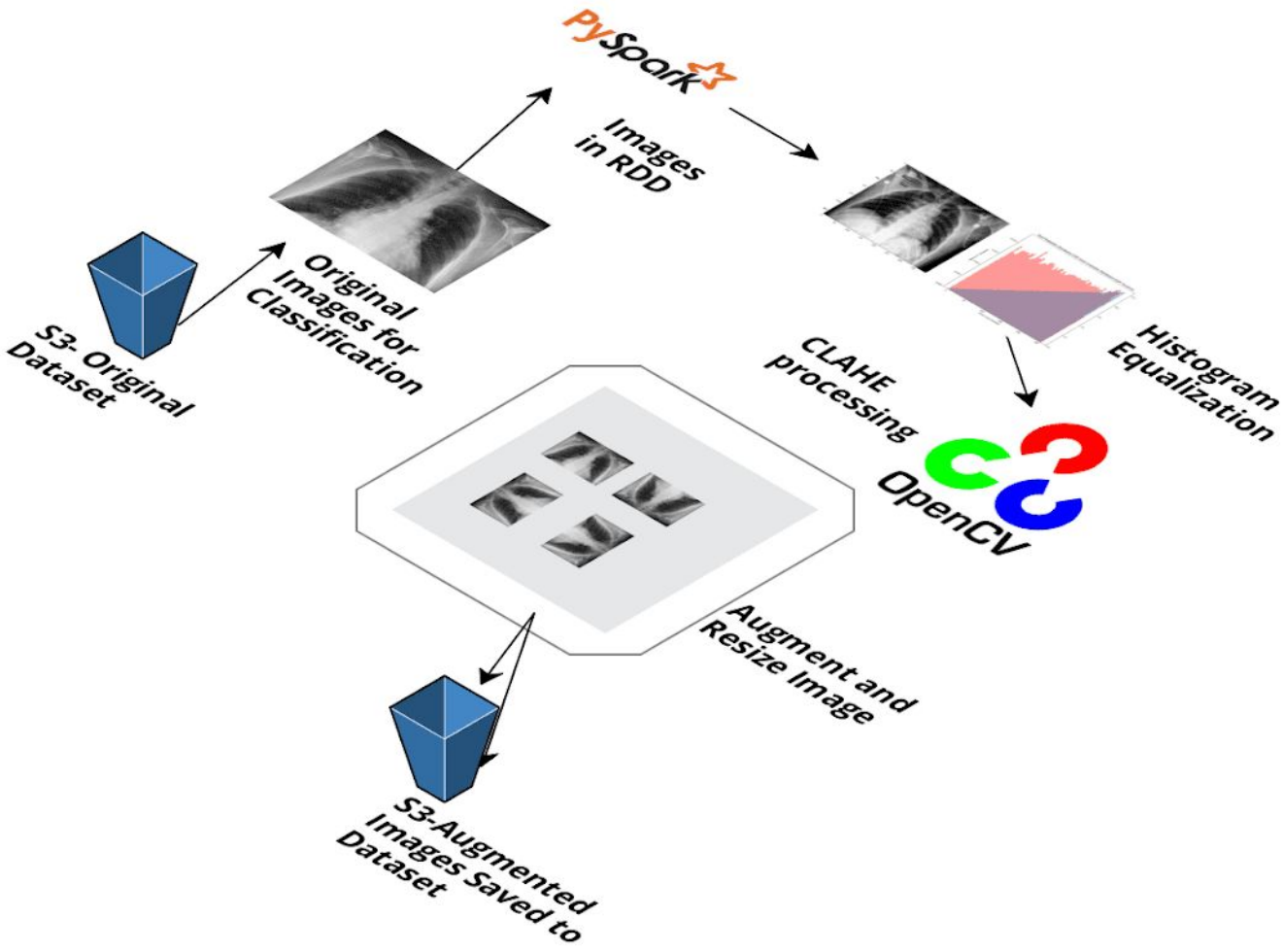
Scott Sims proposed the original ideas presented in this project. He wrote much of the code for the data preprocessing pipeline as well as the autoencoders. Scott also contributed ideas for the architecture of our final model. Joshua Harris downloaded and prepared the raw dataset for the preprocessing pipeline, moving images on AWS S3 to prepare them for the pipeline, and executed the code to prepare images for the autoencoders. Joshua also contributed significant efforts towards writing the proposal, draft, and final paper as well as the video demonstration. Dane Smith provided statistical analysis on the training dataset to assist with our initial run of the preprocessing pipeline, and contributed code for the final model.

References

1. Irvin, J., Rajpurkar, P., Ko, M., Yu, Y., Ciurea-Ilcus, S., Chute, C., ... & Seekins, J. (2019). Chexpert: A large chest radiograph dataset with uncertainty labels and expert comparison. arXiv preprint arXiv:1901.07031.
2. Chalapathy, R., & Chawla, S. (2019). Deep learning for anomaly detection: A survey. arXiv preprint arXiv:1901.03407.
3. Hum, Y. C., Lai, K. W., & Mohamad Salim, M. I. (2014). Multiobjectives bihistogram equalization for image contrast enhancement. Complexity, 20(2), 22-36.
4. Andrews, J., Tanay, T., Morton, E. J., & Griffin, L. D. (2016). Transfer representation-learning for anomaly detection. JMLR.
5. Rajpurkar, P., Irvin, J., Zhu, K., Yang, B., Mehta, H., Duan, T., ... & Lungren, M. P. (2017). Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. arXiv preprint arXiv:1711.05225.
6. Q. Guan and Y. Huang. Multi-label chest x-ray image classification via category-wise residual attention learning. Pattern Recognition Letters, 2018.
7. Wang, X., Peng, Y., Lu, L., Lu, Z., Bagheri, M., & Summers, R. M. (2017). Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2097-2106).
8. Raghupathi, W., & Raghupathi, V. (2014). Big data analytics in healthcare: promise and potential. Health information science and systems, 2(1), 3.
9. Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A., "Inception-v4, Inception-ResNet and the impact of residual connections on learning," in [AAAI Conference on Artificial Intelligence], 4278–4284 (2017). [7] He, K., Zhang, X., Ren, S., and Sun, J., "Deep residual learning for image recognition," in [IEEE Conference on Computer Vision and Pattern Recognition], 770–778 (2016).
10. J. Alex Stark (2000). Adaptive Image Contrast Enhancement Using Generalizations of Histogram Equalization. IEEE Transactions on Image Processing, Vol. 9, No. 5

Appendix 1. Data Processing Architecture Diagram

Below is a high level diagram of our working image processing pipeline used to train our model. Finalizing the architecture of this preprocessing pipeline on the smaller CheXpert dataset allows us to focus the rest of our allotted time on training our final model with a solid foundational dataset properly augmented to tackle the task at hand.



From our proposal, we discussed the first phase of the experiment involves preparing the images for training via a preprocessing pipeline. First, each image in the dataset has an arbitrary width and height in pixels; there is no standard image size. Because of this, images will be resized to a canonical form of 299x299. This size is necessary in order to meet the input requirements for the Inception-ResNet.

Second, the images will undergo adaptive histogram equalization. Histogram equalization is a process in computer vision used to enhance contrast in images by spreading out the most frequent pixel intensities across the pixel spectrum. One possible complication is over amplification of pixel intensities with respect to the global pixel

spectrum. In order to combat this potential side effect, Adaptive Histogram Equalization computes several histograms with respect to several locales in the image. The distinct histograms are then utilized to spread out pixel intensities for their respective regions of the image. This improves local contrast as opposed to global.

The last step of the preprocessing pipeline is an exercise of training sample generation. Each image will undergo an augmentation process of rotation, shifting or scaling to create a new training example. This reduces the probability of a high variance model. Because each image is independent and identically distributed, the preprocessing pipeline can be conducted in parallel using HDFS and Spark.

We've also included a link to our Github repository containing the code for this pipeline here:

https://github.com/Mooseburger1/CSE6250/blob/scott_dev/Pyspark_Pipeline_updated.ipynb