Final Project

Classification of Big Cats from Google Image Dataset

DATA 603 Platforms for Big Data Processing

University of Maryland – Baltimore County

Prof. James Klucar

Team 5

Kumra, Shouray (skumra1)

Mironov, Sanaa (has1)

Purushothaman, Govind Ram (gx54176)

Thakkar, Mahima Ashokbhai (db59735)

The Data Set

The Google Open Images V5 features segmentation masks for 2.8 million object instances in 350 categories. Unlike bounding-boxes, which only identify regions in which an object is located, segmentation masks mark objects' outlines, characterizing their spatial extent to a much higher level of detail. These masks cover a broader range of object categories and a more significant total number of instances than any previous dataset.

The segmentation masks of the training set (2.68M) have been produced by an interactive segmentation process, where professional human annotators iteratively correct the output of a segmentation neural network. This is considered more efficient than manual drawing alone, while at the same time delivering accurate masks. In addition to the masks, 6.4M new human-verified image-level labels are present, reaching a total of 36.5M over nearly 20,000 categories.

Part 1: Preprocessing the data

The search for labels within the Google open image dataset was performed using google OID. In order to find a class of images for extraction, we search for the OID labels by examining the OID label hierarchy. Labels are in a hierarchy meaning that a Lion or Cheetah is a sub-label of Big Cats. To view the set of boxable labels(

https://storage.googleapis.com/openimages/2017_07/bbox_labels_vis/bbox_labels_vis.html)

Keras Models use ResNet image labels. ResNet has 1000 labels. It is important to note the difference of format between the two. ResNet labels are lower case, while Google OID are not. To start the preprocessing of the files, a Spark configuration must be made, and Spark must be launched to work on UMBC cluster ecosystem.

Using Spark syntax, we read the labels in the CSV file ‘class_description_boxable.csv’. It is necessary to create the appropriate schema to read in the file with a specific desired format. We created a simple schema with a struct type LabelName and LabelText. The CSV file imported all labels within the Google Image Dataset. Our goal is to only select and retain Big Cats. To remove any unwanted labels, we call a filter function on the dataframe and set LabelText to Tiger, Lion, Cheetah, Leopard, Jaguar (Animal), and Lynx. As it is a good practice, we check if new dataframe removed all unwanted labels and decreased the overall dataframe size.

Next, we download Image ID files into a new dataframe. Then, we separate the data into three new DataFrames, with each image id in a separate dataframe. In the process, we again defined a schema determining the desired format for viewing and reading the Image ID data. For example, Confidence should be a double, not a string, this can be defined within our schema. Once the files have been read in, they are joined together to create one large dataframe called image labels. The resulting image label dataframe holds all of the image labels form our original data set. The size of the dataframe is over 9 million images. This size is due to the hierarchical structure of the labels, which caused multiple labels for a single image. A big cat can be labeled as a Mammal and a Tiger, as well as other specific designations. To combat this problem, we group the image label by 'Source' and then use distanced 'ImageID' to filter out multiple labels from the same source.

With a dataframe so large, we use one dataframe to filter another. This is done by joining a smaller data frame with a large one. Since we relatively easily filtered the image labels dataframe, we can join the image labels with image IDs. In the process, the image IDs without corresponding image labels are eliminated. To make this dataframe even more robust we add in filter of confidence greater than 99%. This will filter out the large dataframe, image ID, with the desired labels and labels that have a high confidence interval.

To make sure that each image ID within the dataframe is distinct we run another filter on it with ImageID label confidence greater than 99%. The result is a smaller dataframe that went from a size of over 9 million to a little over 3500 labels.

Parquet File Read

The Google Open Image dataset is too large to load into the HDFS ecosystem. To solve this problem, the file was first written to an AVRO file, which was then written to a parquet file. The parquet format contains broken down data to make it easier to read in and to enable management of the large dataset.

When reading in the image parquet file, we extract only the columns we are interested in, such as ImageID, Subset the image belongs to, and the raw data of the image itself. We format these columns to match our main dataframe we have built so far.

We repeat the process of removing unwanted fields by joining a smaller dataframe, image IDs, with our current large dataframe, image parquet.

Bounding Box

At this point the data within our dataframes are not human-readable language and we do not have the full picture. We can address this by reading in the bounding boxes of the whole dataset. We join three dataframes into one bounding box dataframe. Again, this will have all the bounding boxes of all the images within the original dataset. When reading in the three CSV files, a schema was not created, so the data read in is not the format we want. This will be corrected later on. Repeating the process of removing unwanted fields by joining a smaller dataframe, image IDs, with our current large dataframe, bounding box. This process removes any image ID that is not within the dataframe image ID. The bounding box dataframe now has only the images we are interested in.

Still we are unable to read the data from the bounding box. To fix this problem, we join the bounding box dataframe with the label dataframe. This is joining the label code to the label name to make the information within the dataframe human-readable.

Extract image chips:

All the CSV files have been read, formatted to meet our criteria, and organized into a dataframe. Now we want to be able to extract an image chip, not just an image itself. The image chip will have information such as the label name, ID, bounding box size, and the confidence interval within each image.

To expand our bounding box dataframe to have all the information it needs, we join in the images parquet dataframe. This adds the raw ImageID of an image from the parquet data to the bounding box dataframe. To be able to extract each image chip, we define a UDF to extract the portion of the image defined within the bounding box.

UDF extract image chip uses the PIL library to read in one image at a time. When reading in the image it also extracts the image size. Then it starts converting the bounding box x and y directions to a float number. It calculates the size of the picture based on the bounding boxes. The bounding box coordinates are given as fractions of the number of pixels in the image. It takes the image and crops it to the size of the bounding box it just calculated. The image is saved to a byte-buffer by getting the raw bytes of the image before returning it and repeating the same process on the next image.

Parquet File Write

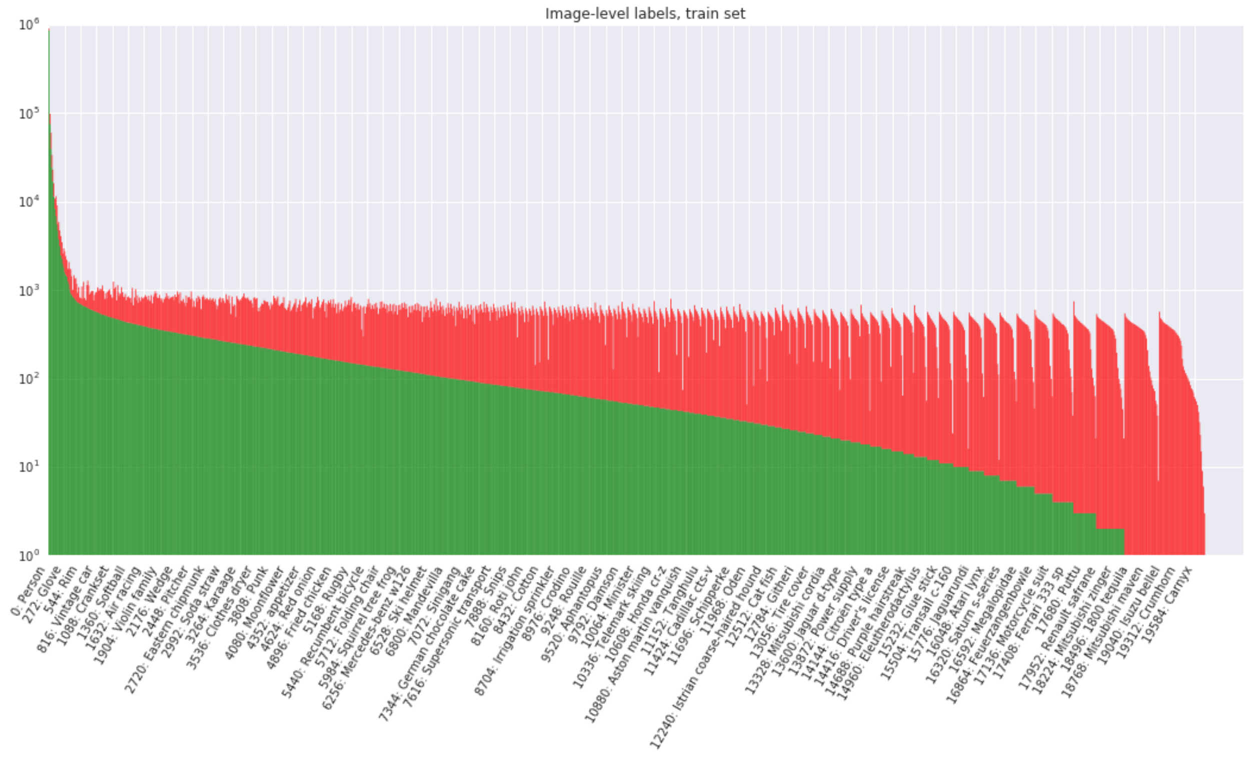
Now that all the data from the original Google Open Image dataset has been extracted and cleaned up to our desired format, we can write the dataframe to a parquet file. This is done by defining another UDF that writes each image chip file to the HDFS and evaluates it on the dataframe.
The UDF writes the image chips from every spark node in parallel across the cluster. This also avoids pulling every image chip back to the notebook later when needed.
Outside of the UDF function we write the file by using the desired dataframe to write.praquet file to the path. It is very important to consider that this action might result in running out of memory on the cluster. To address this potential problem, we increase capacity and, if necessary, use write.mode('overwrite').praquet to write over whatever was not saved.

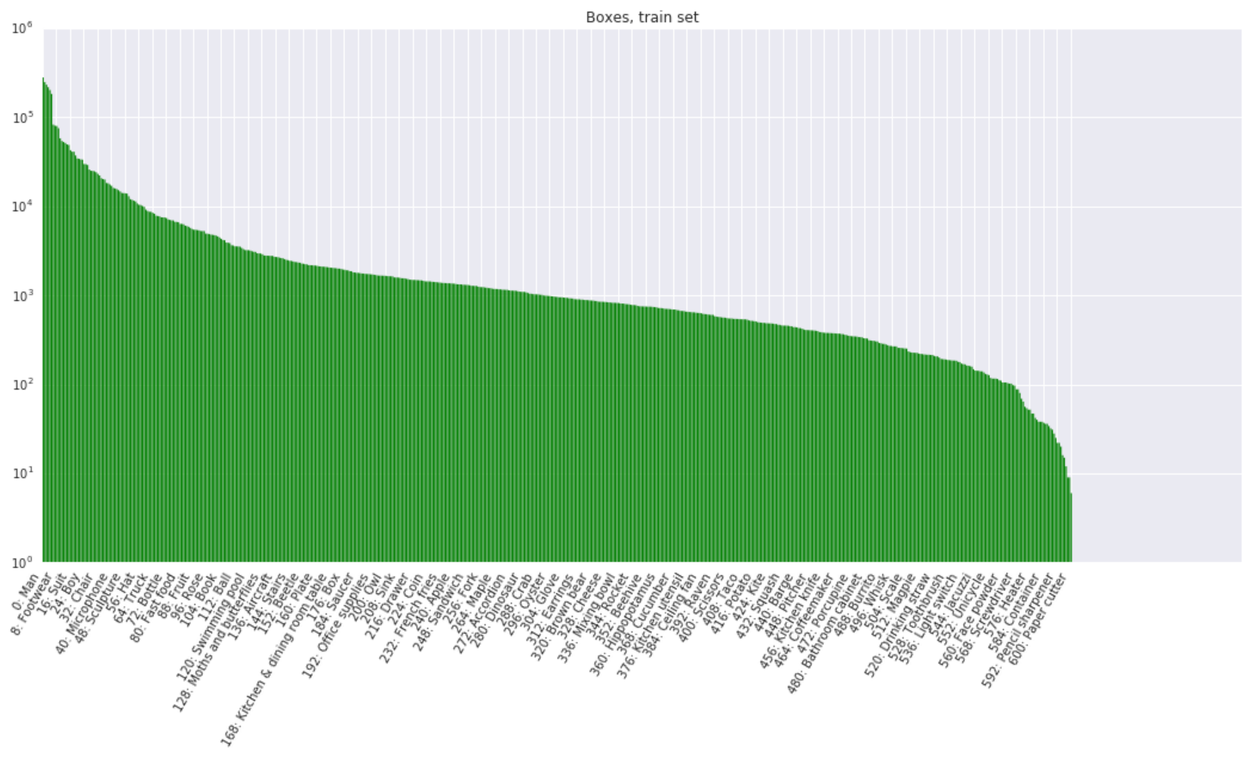
Basic Exploratory Analysis

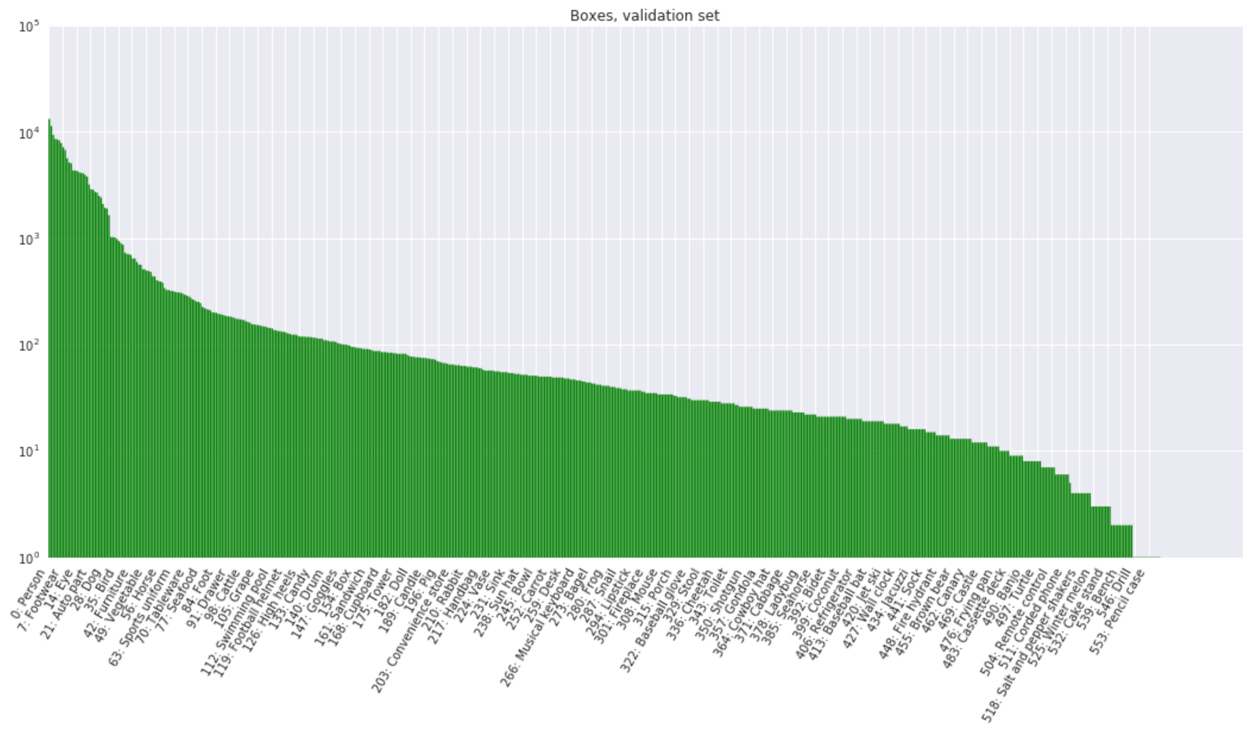
The dataset is not consistent. There are issues associated with different sizes, orientations, and occlusions possible in our target classes. In a few cases, we were not successful in downloading the actual image we wanted.

Label distribution

The following figures show the distribution of annotations across the dataset. The class distribution is heavily skewed. The number of positive examples in orders of labels, then by the number of negative examples. The different colors like green indicate positive examples, while red indicates negatives.




Boxes, train set
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0: Man
8: Footwear
16: Suit
24: Boy
32: Chair
40: Microphone
48: Sculpture
56: Hat
64: Truck
72: Bottle
80: Fast food
88: Fruit
96: Rose
104: Book
112: Ball
120: Swimming pool
128: Moths and butterflies
136: Aircraft
144: Stairs
152: Beetle
160: Plate
168: Kitchen & dining room table
176: Box
184: Saucer
192: Office supplies
200: Owl
208: Sink
216: Drawer
224: Coin
232: French fries
240: Apple
248: Sandwich
256: Fork
264: Maple
272: Accordion
280: Dinosaur
288: Crab
296: Oyster
304: Glove
312: Earrings
320: Brown bear
328: Cheese
336: Mixing bowl
344: Rocket
352: Beehive
360: Hippopotamus
368: Cucumber
376: Kitchen utensil
384: Ceiling fan
392: Raven
400: Scissors
408: Taco
416: Potato
424: Kite
432: Squash
440: Barge
448: Pitcher
456: Kitchen knife
464: Coffeemaker
472: Porcupine
480: Bathroom cabinet
488: Burrito
496: Whisk
504: Scale
512: Magpie
520: Drinking straw
528: Toothbrush
536: Light switch
544: Jacuzzi
552: Unicycle
560: Face powder
568: Screwdriver
576: Heater
584: Container
592: Pencil sharpener
600: Paper cutter

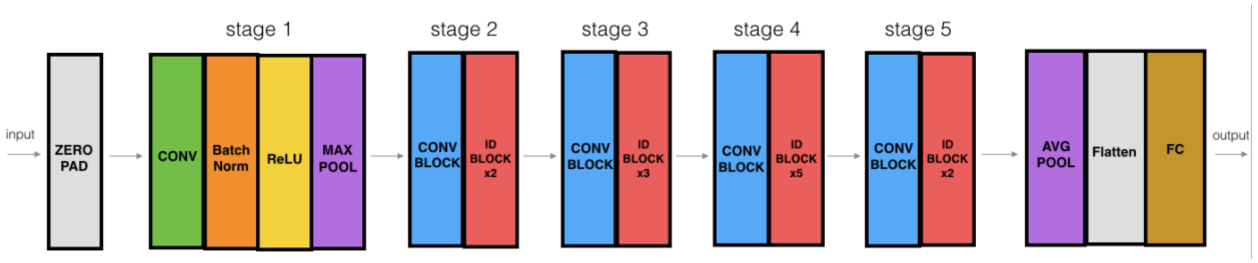

Boxes, validation set
10^5
10^4
10^3
10^2
10^1
10^0
0: Person
7: Footwear
14: Eye
21: Auto part
28: Dog
35: Bird
42: Furniture
49: Vegetable
56: Horse
63: Sports uniform
70: Tableware
77: Seafood
84: Foot
91: Drawer
98: Cattle
105: Grape
112: Swimming pool
119: Football helmet
126: High heels
133: Candy
140: Drum
147: Goggles
154: Box
161: Sandwich
168: Cupboard
175: Tower
182: Doll
189: Candle
196: Pig
203: Convenience store
210: Rabbit
217: Handbag
224: Vase
231: Sink
238: Sun hat
245: Bowl
252: Carrot
259: Desk
266: Musical keyboard
273: Bagel
280: Frog
287: Snail
294: Lipstick
301: Fireplace
308: Mouse
315: Porch
322: Baseball glove
329: Stool
336: Cheetah
343: Toilet
350: Shotgun
357: Gondola
364: Cowboy hat
371: Cabbage
378: Ladybug
385: Seahorse
392: Bidet
399: Coconut
406: Refrigerator
413: Baseball bat
420: Jet ski
427: Wall clock
434: Jacuzzi
441: Sock
448: Fire hydrant
455: Brown bear
462: Canary
469: Castle
476: Frying pan
483: Cassette deck
490: Banjo
497: Turtle
504: Remote control
511: Corded phone
518: Salt and pepper shakers
525: Winter melon
532: Cake stand
539: Bench
546: Drill
553: Pencil case

Model Evaluation

Why ResNet50?

ResNet50 is a 50-layer deep, pre-trained Deep Learning model for image classification of the Convolutional Neural Network, mostly used to analyze visual imagery. A pre-trained model is a more practical approach than collecting sufficient amount of data and training the model from the start. It is trained on a million images of 1000 categories from the ImageNet database. The model has over 23 million trainable parameters, which indicates a deep architecture that makes it better for image recognition and classification. Also, ResNet50 has excellent generalization performance with smaller error rates as compared to other pre-trained models like AlexNet, GoogleNet, or VGG19. The network has an image input size of 224-by-224.

Further, in a deep convolutional neural network like VGG16, several layers are stacked and trained to the task, and the network learns several low/mid/high-level features at the end of its layers. Whereas in residual learning, instead of learning some features, we try to learn some residual by subtracting the feature learned from the input of the layer. This is done by directly connecting the nth layer's input to some (n+x) layer. Training these kinds of networks is more accessible than training simple deep convolutional neural networks, and the problem of degrading accuracy is also resolved.



Why MOBILENET?

MobileNet is a separable convolution (Conv) model that is composed of depth-wise and pointwise Conv. Depth-wise Conv is performed independently for every input channel of the image; this reduces the computational cost by omitting conversation in the channel.

depthwise conv

MobileNet independently performs Conv. MobileNet is efficiently used to predict the category we want if available in ImageNet.

Advantages of using MobileNet:

- Easy to apply as it is lightweight;
- Uses depth wise separable conv, which is a streamlined and human-understandable architecture;
- Reduces the size of the network;
- Reduces the number of parameters, resulting in greater space availability;
- A small Convolutional Neural Network does not forget the weights at a large scale and gives reliable predictions;
- Fast in performance as compared to a few other pre-trained models.

Disadvantages:

- There is a slight reduction in accuracy when compared with other pre-trained models.

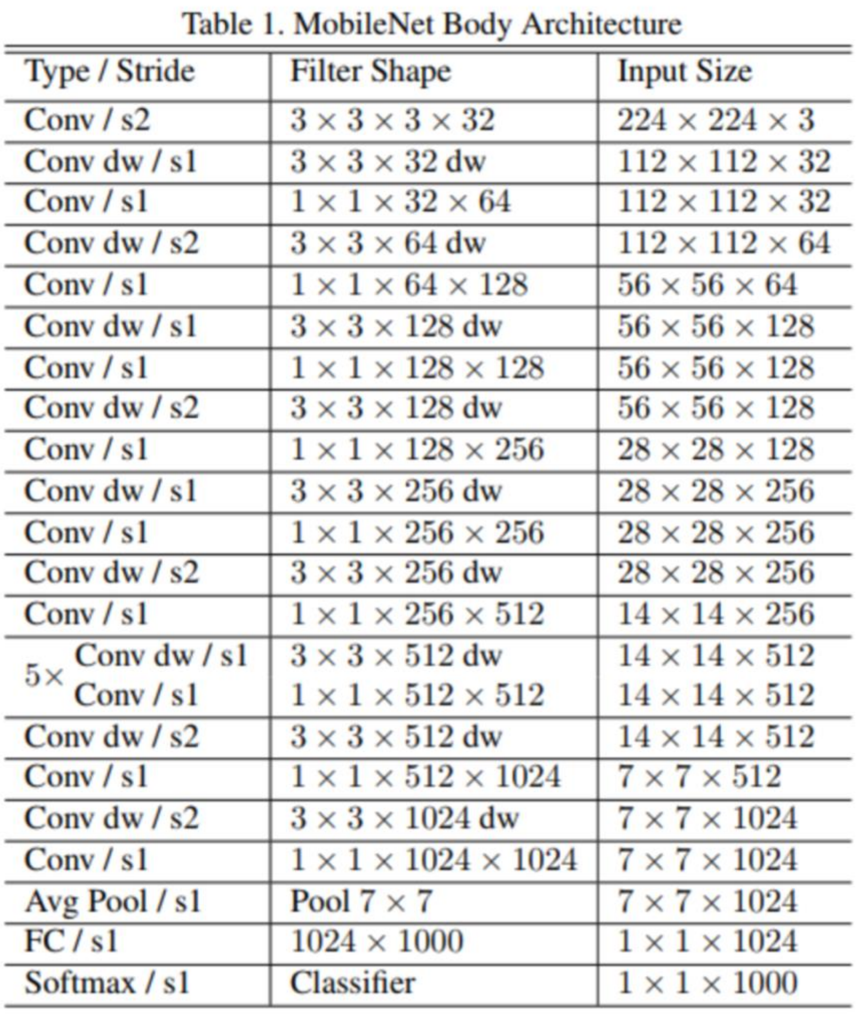


Table 1. MobileNet Body Architecture

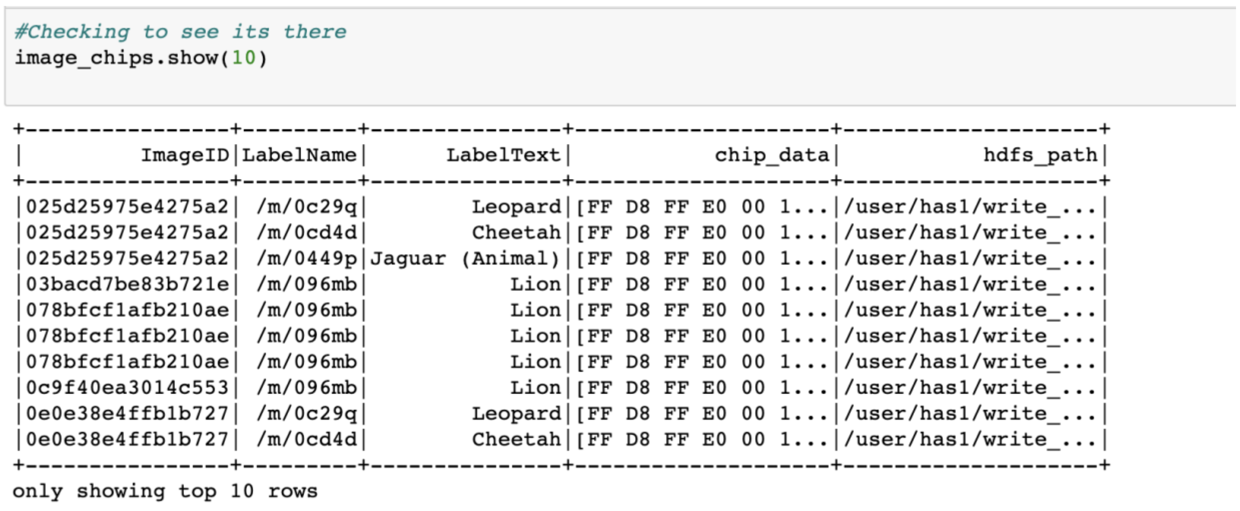
Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5× Conv dw / s1 Conv / s1	$3 \times 3 \times 512$ dw $1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$ $14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Running a Model

In a separate notebook we run a Keras model in parallel with Spark on all the image chips that were extracted. Since this is a new notebook, Spark must be configured and launched with the model file uploaded.

The process is the same for both models when it comes to reading the data.

We start out by reading in the data. This is the data from the parquet file that was created in the pre-processing section. Once the data has been read in and saved to a model, we verify that the amount of data that is within the parquet file matches the parquet file made in the preprocess section. Then you must create a grouping column. This dataframe contains 10k images to run predictions on. This section adds in a new column to the dataframe because we don't have something to group by. Arrow is a shared data structure, the new process cannot just add to the memory space of the originating process. To overcome this, we must create empty columns to be populated by the UDF. I do this using the `lit` function to populate columns with literal values of an empty string and a 0.0.



```
#Checking to see its there
image_chips.show(10)
```

```
+----------------+---------+---------------+--------------------+--------------------+
|         ImageID|LabelName|      LabelText|           chip_data|           hdfs_path|
+----------------+---------+---------------+--------------------+--------------------+
|025d25975e4275a2| /m/0c29q|        Leopard|[FF D8 FF E0 00 1...|/user/has1/write_...|
|025d25975e4275a2| /m/0cd4d|        Cheetah|[FF D8 FF E0 00 1...|/user/has1/write_...|
|025d25975e4275a2| /m/0449p|Jaguar (Animal)|[FF D8 FF E0 00 1...|/user/has1/write_...|
|03bacd7be83b721e| /m/096mb|           Lion|[FF D8 FF E0 00 1...|/user/has1/write_...|
|078bfcf1afb210ae| /m/096mb|           Lion|[FF D8 FF E0 00 1...|/user/has1/write_...|
|078bfcf1afb210ae| /m/096mb|           Lion|[FF D8 FF E0 00 1...|/user/has1/write_...|
|078bfcf1afb210ae| /m/096mb|           Lion|[FF D8 FF E0 00 1...|/user/has1/write_...|
|0c9f40ea3014c553| /m/096mb|           Lion|[FF D8 FF E0 00 1...|/user/has1/write_...|
|0e0e38e4ffb1b727| /m/0c29q|        Leopard|[FF D8 FF E0 00 1...|/user/has1/write_...|
|0e0e38e4ffb1b727| /m/0cd4d|        Cheetah|[FF D8 FF E0 00 1...|/user/has1/write_...|
+----------------+---------+---------------+--------------------+--------------------+
only showing top 10 rows
```

Pandas UDF For the Models

I create the Pandas UDF using the `pandas_udf` decorator. The schema of the dataframe is passed into the UDF decorator.

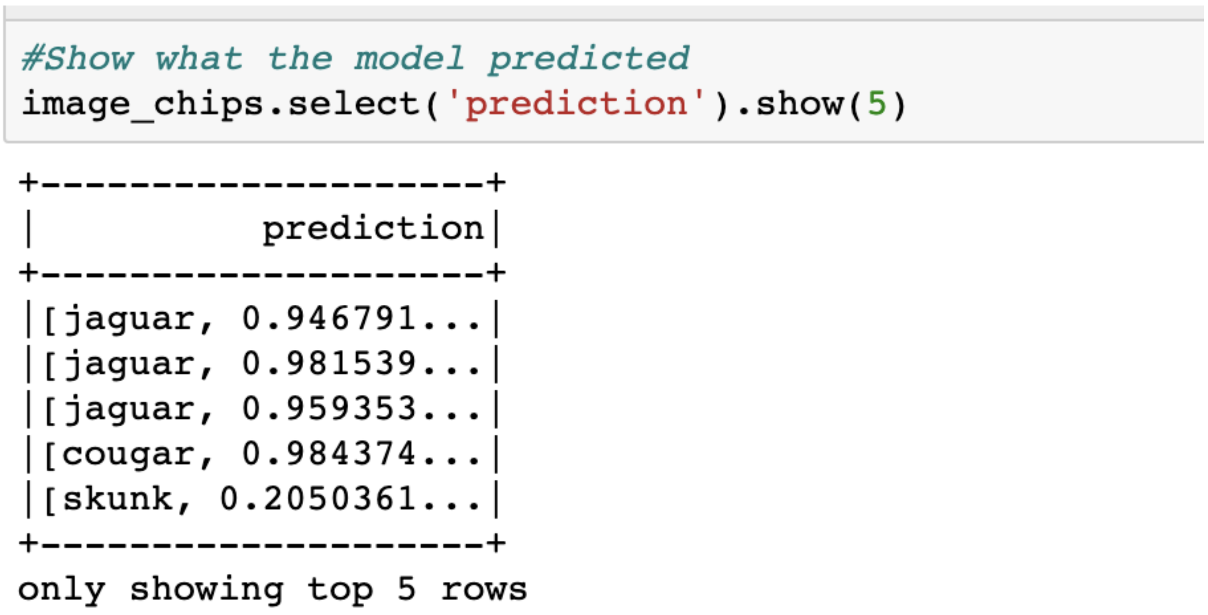
The input to the function (called `pdf` here) is a Pandas dataframe. This is simply a Pandas dataframe that is pointing to a __portion__ of the Spark dataframe. It is created __without copying__ any data from the Spark memory to the Python process memory so it's fast.

The output of the function is the _same_ Pandas dataframe. The function simply modifies the dataframe as it wants. Doing this is actually modifying the Spark in-memory data structure! This is why new empty columns had to be pre-allocated in the spark dataframe so the UDF could put new data in an existing empty memory location. The function uses Keras to generate a ResNet50 prediction model. It is also applied to MobileNet predication model.

Once the evaluation is complete, we can extract the predication label and confidence score by creating another UDF to wrap the Keras evaluation that is defined as a MapType. MapType is similar to a Python dictionary, but Spark sticks to a strong data type, thus allowing for setting a hard datatype.

MobileNet Model Prediction:

MobileNets trade off between latency, size and accuracy while comparing favorably with popular models from the literature. Though the pretrained network can classify images into 1000 object categories. However, MobileNet prediction are not as accurate as ResNet prediction would have been because of the model were built. Though we have high confidence score with the MobileNet prediction, another model would have been better.



```
#Show what the model predicted
image_chips.select('prediction').show(5)
```

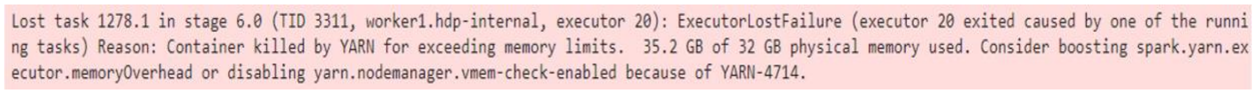
```
+--------------------+
|          prediction|
+--------------------+
|[jaguar, 0.946791...|
|[jaguar, 0.981539...|
|[jaguar, 0.959353...|
|[cougar, 0.984374...|
|[skunk, 0.2050361...|
+--------------------+
only showing top 5 rows
```

ResNet Model Prediction:

We were unsuccessful in compiling the model on our dataset. However, reading in the data and running it thought an image chip evaluate is the same. Instead of running the images on MobileNet prediction model the images would have been compiled with ResNet50 predication model.

Challenges and Errors faced:

The main issue we faced at the start of the project was memory issues. Whenever we wrote to the parquet file, it would exceed the physical memory of 20GB. This was later fixed by increasing capacity to 50GB



```
Lost task 1278.1 in stage 6.0 (TID 3311, worker1.hdp-internal, executor 20): ExecutorLostFailure (executor 20 exited caused by one of the runni
ng tasks) Reason: Container killed by YARN for exceeding memory limits.  35.2 GB of 32 GB physical memory used. Consider boosting spark.yarn.ex
ecutor.memoryOverhead or disabling yarn.nodemanager.vmem-check-enabled because of YARN-4714.
```

the second major issue we faced is running the ResNet Model on our dataset. We continued to have issues with the file format that was being read in by the UDF. Because of this we were not able to run any prediction that would have been better than the MobileNet prediction on our dataset.

Reference

https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96

https://medium.com/@nina95dan/simple-image-classification-with-resnet-50-334366e7311a

All notebooks were adopted from Professor James Klucar