Park or Bird? An XKCD Inspired Distributed Image Processing and Machine Learning Classifier using Spark

Evan Kepner, Joan Qiu, Simon Macarthur
UC Berkeley School of Information (iSchool)
Berkeley, CA, USA

August 15, 2015

Abstract

In computer science, it can be difficult to explain the difference between the easy and the virtually impossible[1]. We were inspired by an xkcd comic to take the "park or bird" challenge using Spark and MLlib. Our goal is to build a scalable system for image processing: ingesting raw images, converting images to machine learning features, training a classifier, and ultimately building a deployable scalable prediction engine based on Spark.

Figure 1: Our inspiration (we had 2 months).

1 Cluster Configuration

We have configured a 3 node cluster with each node having 10 cores and 8 GB of available memory. GPFS is used as a shared filesystem for holding raw image data and Spark analysis inputs. Spark 1.4.0 is deployed across the entire cluster to minimize network traffic for source files. Due to write affinity with GPFS, we execute all transformation and data preparation scripts on the appropriate node for the stored files. Birds were stored on a single node, Parks on a single node, and Other on a single node. Spark is configured to use 6 GB of memory per executor and driver.

2 Methods

2.1 Data Collection

Metadata were gathered from Flickr[2] using the API and stored as JSON documents in MongoDB. All images are licensed under creative commons. We gathered 128 GB of image metadata, totalling over 100 million individual records, for our initial search. Keywords were applied to the `user_tag` field such as "park" or "bird", including national park names and specific bird species, to filter records of potential interest. This subset was downloaded into the GPFS cluster. We had explored image downloads with the Google Image API and other robots.txt respectful web-scraping techniques but found the Flickr method to be sufficient. Of the 100 million records, approximately 2.5 million had a keyword for "bird". We used MongoImport after discovering unacceptable latency in our implementation with the Python client. Example image metadata stored in the MongoDB collection:


```
{
    "_id" : ObjectId("5589caf77ee04b7694d5cfb8"),
    "id" : NumberLong(5552808760),
    "User_NSID" : "57644250@N00",
    "User_nickname" : "welshmackem",
    "Date_taken" : "2011-03-05 16:02:35.0",
    "Date_uploaded" : 1300875447,
    "Capture_device" : "Canon+EOS+400D+DIGITAL",
    "Title" : "Goosander",
    "Description" : "",
    "User_tags" : "bird",
    "Machine_tags" : "",
    "Longitude" : -1.955995,
    "Latitude" : 55.153741,
    "Accuracy" : 13,
    "page_URL" : "http://www.flickr.com/photos/
        57644250@N00/5552808760/",
```

```
    "downloadURL" : "http://farm6.staticflickr.com/
        5135/5552808760_5a56121b83.jpg",
    "License_name" : "Attribution-NonCommercial-ShareAlike License",
    "License_URL" : "http://creativecommons.org/
        licenses/by-nc-sa/2.0/",
    "serveridentifier" : 5135,
    "farm_identifier" : 6,
    "secret" : "5a56121b83",
    "secret_original" : "d354bbe015",
    "Extension" : "jpg",
    "Photo_video" : 0
}
```


2.2 Search

Given we have metadata for 100 million images, viewing these images can be problematic, and searching them impossible! For this reason, we implemented Apache Solr[3] to enable fast indexing and searching. To enable easy user search and viewing of photographs, Blacklight[4] was installed and configured as the search interface. This provided the ability to search for terms in the metadata including title and user-tags. Out of the box functionality for Blacklight only provides textual information for the search, so customization was performed to enable Blacklight to return the images in the search results. The final Blacklight search interface, connecting to Solr, can be found at http://169.53.132.59:3000/

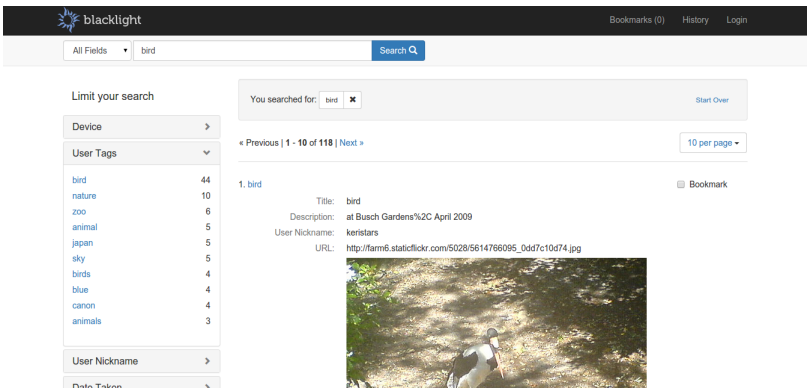

Figure 2: Blacklight search interface

2.3 Data Cleansing and Labeling

User tags provided a good first pass at labeling the data. We used the metadata from our search as the original designation. There was a high variance within the collected

pictures for what represented a bird or park based on these criteria. As a cleansing attempt, we loaded the images onto a separate server and provided a viewable portal using Piwigo[5]. Piwigo provides 15 images per page for viewing and we were able to label images by hand. To expedite the labelling process we built a Python web scraper using Requests[6] and BeautifulSoup[7] to generate a csv of image IDs. This way, on each page we could quickly go through the pre-populated list and tag as either valid or invalid for each category.

We manually labelled 1000 parks and 1000 birds in this manner, and found that approximately 63% of our sample for each would provide valid clean images. We attempted to train a basic classifier on these 1000 samples to classify the remaining images, but found that the accuracy was too low to be practically useful. Therefore, we have moved forward without fully cleansed data and understand the accuracy of our model will suffer. Our goal is to build the framework for scalable processing and our next iteration will deal directly with data labelling. Possible solutions include using Mechanical Turk or other crowdsourcing, as well as building a small labeling app. Alternatively, we could use pre-cleaned data sources besides Flickr as our image providers. Our feature extraction, model building, and cluster configuration would work with any image set of appropriate scale.

2.4 Feature Extraction

Different features extraction techniques were used for testing multiple models. Features extraction methods included:

1. Full RGB pixel arrays from normalized and scaled images.
2. Size invariant vector extraction by pixel ratio.
3. Size invariant key point and descriptor extraction using computer vision algorithms.

Our feature data were structured as key-value pairs. Given the volume, multiprocessing and compression were required. Numpy[8] and PIL[9] are used with Python to create text files for consumption in Spark. Python Multiprocessing[10] handles the parallelization. Images are stored with a unique numeric ID e.g. 1234567890.jpg. Our multiprocessing script took advantage of this nomenclature. The last digit of the image ID was used for CPU assignment (0-9) and compressed files were created for each CPU set e.g. `bird_mp_0.txt.gz` through `bird_mp_9.txt.gz` in the GPFS cluster.

Each data row in the compressed file is a key-value pair of pixel values from the numpy transformation with the image ID. In Python, the array representation of an RGB image is in a 3-D array. For our full feature transformation we flattened this to a 1-D array. To ease splitting the key-value data in Scala, we are separating with two

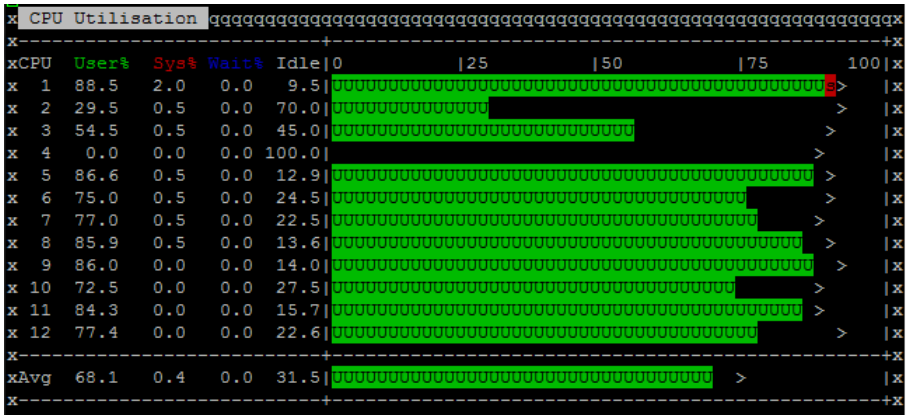

Figure 3: Example CPU utilization for one node running the image pre-processing scripts in parallel.

delimiters: a comma between the key and value, and spaces between each number in the value array e.g.

128346178290.jpg, 0 25 255 6 9 1 .... 9 82 254

This convention is maintained for all feature extraction methods, and lends itself to consistent key-value extraction in Scala and PySpark e.g.

Scala Example:

```
val parks = sc.textFile("/gpfs/gpfsfpo/park_raw_ml")
val p = parks.map(x => x.split(',')(1))
        .map(_.split(" "))
        .map(x=>x.map(_.toDouble))
```

PySpark Example:

```
parks = sc.textFile("/gpfs/gpfsfpo/park_raw_ml")
p = parks.map(lambda x: x.split(',')[1])
        .map(lambda x: x.split(' '))
        .map(lambda x: [float(y) for y in x])
        .map(lambda x: Vectors.dense(x))
```

2.4.1 Full RGB Pixel Arrays

Images come in different sizes. Our first step was to normalize the sizes for consistent analysis. This involved running a pre-processing script to gather the dimensions of every image with Python PIL, and determine the average image size for the entire collection. We used this average as our baseline resizing scale. From the 400,000 images we gathered for parks and birds our average image dimensions were: 478 x 398. Images were resized

to 20% of the average dimension, 96px by 80px. Raw data were not modified. This resizing was done at the time of array creation for our analysis matrix. This produced a 23,040 feature space in the flattened array representation since each image is RGB (96 x 80 x 3).

This is the largest feature vector representation. 391,000 bird images require 8.3 GB of storage space represented in compressed form using this vector structure. An individual image averages 21 KB. Our multiprocessing scripts chunk output into 10 gzip files. For birds, this method yields a single gzip file per CPU averaging 800 MB each, representing approximately 37,000 individual images per compressed file. For parks, each compressed file is approximate 300 MB representing 14,000 individual images per compressed file. The total compressed size for park images is 3.1 GB, bringing the total analysis file size to 11.4 GB for the entire image data set.

2.4.2 Size Invariant Pixel Ratios

The RGB color space has $(2^8)^3$ elements (over 16 million), which caused our large feature space using pixel arrays on scaled images. Our goal was to reduce the overall feature size by mapping each of the possible 256 possible values to a ratio of pixels for each color based on the total number of pixels[11]. We set a block size of 4, giving a $4^3 = 64$ possible features in our vector.

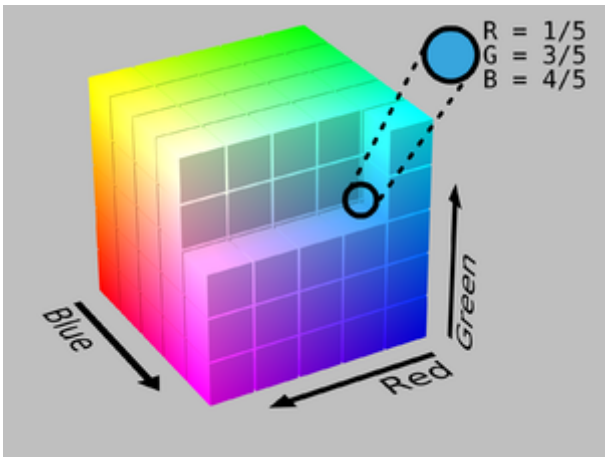

Figure 4: Illustration showing the RGB ratio block reduction[12].

This smaller feature space greatly reduced the storage requirements for analysis files, and improved model training time. Our total parks storage requirement was 14 MB over 10 1.4 MB compressed files, still averaging 14,000 parks per file. Birds required 70 MB over 10 5.5 MB compressed files, still averaging 37,000 birds per file. We sampled 1000 park and bird images for initial RGB reduction analysis to see if there were trends in the ratios.

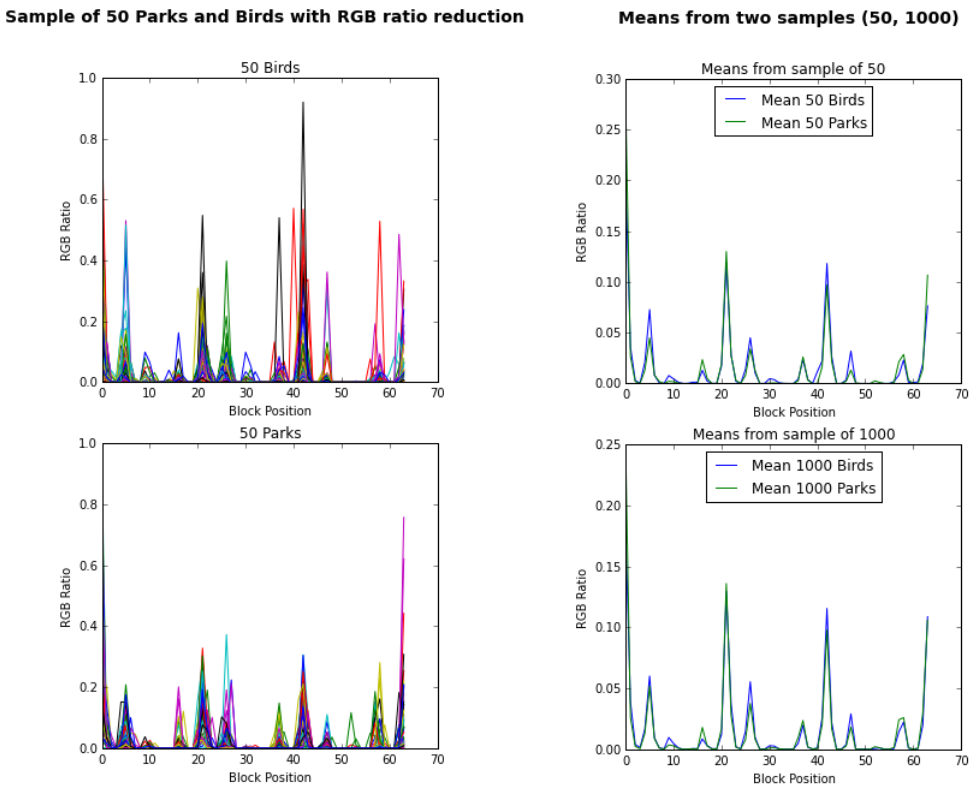

Figure 5: 50 bird and park images plotted with RGB block ratios and the means from all samples.

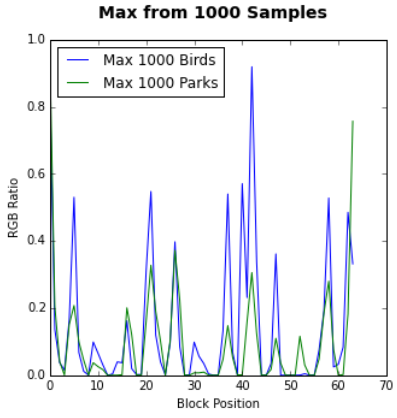

Figure 6: Max values from 1000 samples in parks and birds RGB block ratios.

2.4.3 OpenCV Algorithms

OpenCV[13] provides different algorithms for size invariant feature detection. Using SIFT, SURF, and ORB feature arrays can be extracted from images and processed with our system. Given our known training data quality we did not implement this for full processing; however, our analysis shows that ORB would provide a viable method for feature extraction. ORB[14] uses FAST (Features from Accelerated Segment Test) in pyramids to detect stable keypoints, selects the strongest features using FAST or Harris response, and finds their orientation using first-order moments and computes the descriptors using BRIEF (Binary Robust Independent Elementary Features)[15]. Using ORB we limit the representation to an n x 32 matrix. With n=100 key points our feature space would be 3200 descriptors per image. This feature space would be completely manageable in our system, and we have built in the ability to reduce dimensionality with PCA if needed.

Figure 7: Image processed with the SIFT algorithm plotting key points.

Figure 8: Image processed with the SURF algorithm plotting key points.

Figure 9: Image processed with the BRIEF algorithm plotting key points.

Figure 10: Image processed with the ORB algorithm plotting key points.

2.5 MLlib Model Training

Using MLlib and Scala we tested two forms of logistic regression modelling: logistic regression with SGD and logistic regression with LBFGS. Our code runs with both models, though for implementation we restrict the code on one model per run. In general, we have focused on LBFGS. The Scala code executes the following routine:

1. The birds and parks directories of compressed files are set as new RDDs. Appropriate selections are made to keep an approximately even count of parks and birds.
2. A new labeled point and vector RDD is created for each:
   (a) The key-value pair is split on the comma, keeping the value array.
   (b) The value array is split on the space and converted from string to double type.
   (c) The value array is mapped to a dense vector representation and associated with a label: 1 for bird, and 0 for park.
3. The park and bird vector RDDs are unioned as a new data RDD.
4. The data RDD is randomly split into training and test arrays (70%, 30% respectively).
5. The model is trained on the training array.
6. The model is evaluated using MulticlassMetrics precision against the test array.
7. The model is saved to the cluster for the prediction engine.

2.5.1 Library Requirements for Spark 1.4.0

1. org.apache.spark.{SparkContext, SparkConf}
2. org.apache.spark.mllib.classification.{LogisticRegressionWithSGD, LogisticRegressionWithLBFGS, LogisticRegressionModel}
3. org.apache.spark.mllib.evaluation.MulticlassMetrics
4. org.apache.spark.mllib.regression.LabeledPoint
5. org.apache.spark.mllib.linalg.Vectors

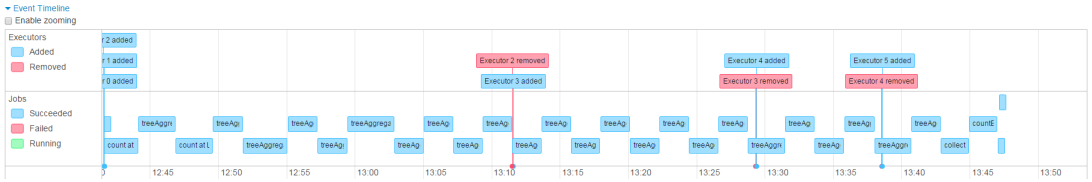

Figure 11: Logistic regression with LBFGS training event timeline in Spark using the full RGB feature arrays.

2.6 MLlib Principal Component Analysis

Using MLlib and Scala we created a model building routine with dimensionality reduction using the standard scaler and principal component analysis. With this method we can project the full feature space onto a lower dimensional representation. The routine executes as follows:

1. The birds and parks directories of compressed files are set as new RDDs. Appropriate selections are made to keep an approximately even count of parks and birds.
2. A new labeled point and vector RDD is created for each:
   (a) The key-value pair is split on the comma, keeping the value array.
   (b) The value array is split on the space and converted from string to double type.
   (c) The value array is mapped to a dense vector representation and associated with a label: 1 for bird, and 0 for park.
3. The park and bird vector RDDs are unioned as a new data RDD.
4. The data RDD is randomly split into training and test arrays (70%, 30% respectively).
5. A standard scaler is fit to the training data to normalize values with the training mean and variance.
6. The standard scaler is applied to both the training and test arrays creating RDDs of labeled points.
7. A principal component analysis object is created with a set number of components based on the scaled training data.
8. The scaled training and scaled test RDDs are projected onto the lower dimensional subspace using the fit principal component analysis object as new RDDs.
9. The model is trained on the projected training array.

10. The model is evaluated using MulticlassMetrics precision and Mean Squared Error against the projected test array.

11. The model is saved to the cluster.

At this time, the objects for the fit standard scaler and principal component analysis could not be exported to disk for use in the prediction engine, and do not exist in PySpark for pickling[16]. The cluster must be scaled appropriately depending on the feature space. Our current cluster configuration will only support principal component analysis on the RGB ratio (64 features) vectors and not the full RGB feature set of (20,340 features). The following results are based on logistic regression with LBFGS. There were 109,754 birds and 101,025 parks used in the total data set (uncleaned, as previously specified). For a sample from the full RGB feature space we found that the first 150 principal components accounted for 90% of the variance in the data.

PCA Components	Precision	MSE
10	0.80	0.198
20	0.79	0.21
30	0.81	0.187

Table 1: Principal component analysis precision and mean squared error.

2.6.1 Library Requirements for Spark 1.4.0

1. org.apache.spark.{SparkContext, SparkConf}

2. org.apache.spark.mllib.classification.{LogisticRegressionWithSGD, LogisticRegressionWithLBFGS, LogisticRegressionModel}

3. org.apache.spark.mllib.evaluation.MulticlassMetrics

4. org.apache.spark.mllib.regression.LabeledPoint

5. org.apache.spark.mllib.linalg.Vectors

6. org.apached.spark.features.{PCA, StandardScaler}

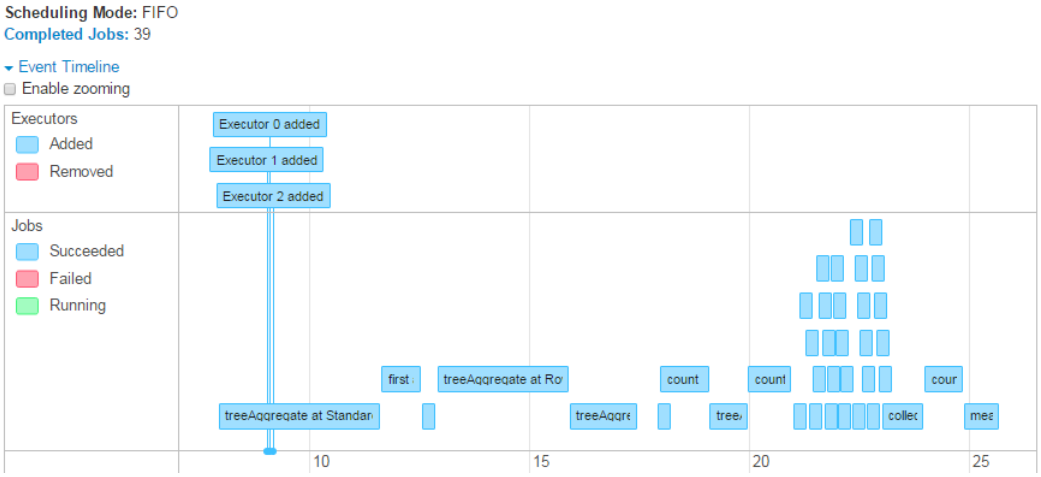

Figure 12: PCA model training event timeline in Spark using logistic regression with LBFGS on the RGB ratio reduced feature vectors.

2.7 Prediction Engine

We have created a prediction engine using PySpark that combines our feature processing technique with the trained model. The script takes either a single image, or directory of images, as the command line argument input and outputs predicted labels with the file name. We chose PySpark instead of Scala as our build language to maintain consistency with the PIL processing requirements. The prediction engine runs the following routine:

1. Based on the input command line argument, process either a single image or group of images using the feature extraction technique for the specified model.
2. Processed image files are saved to the cluster GPFS directory in compressed form for access to all Spark nodes.
3. An RDD is created from the processed inputs.
4. The RDD is split by commas to create an RDD for keys, and an RDD for values.
5. The values RDD is split on the white space, and mapped from string to double type.
6. A vector RDD is generated from the values RDD.
7. Predictions are generated for each input case based on the vector RDD.
8. The RDD of keys is zipped with the RDD of output predictions specifying “Bird” (1) or “Park” (0) for each processed image file.

The prediction engine takes two arguments: –i for the input file or folder, and –m for the model method (1 for full RGB arrays, 2 for RGB ratio reduction method).

2.7.1 Prediction Engine Running Example

```
# /usr/local/spark/bin/spark-submit spark-Predictor.py --help
usage: spark-Predictor.py [-h] --i I --m M

Park or Bird Prediction Engine

optional arguments:
  -h, --help          show this help message and exit
  --i I, --input I    Input file or directory of jpg images
  --m M, --method M   Model method, 1 or 2

# /usr/local/spark/bin/spark-submit spark-Predictor.py
--i /gpfs/gpfsfpo/pred_test/ --m 1

/gpfs/gpfsfpo/pred_test/8216542107.jpg
/gpfs/gpfsfpo/pred_test/503215945.jpg
/gpfs/gpfsfpo/pred_test/2792971789.jpg
/gpfs/gpfsfpo/pred_test/3655965983.jpg
/gpfs/gpfsfpo/pred_test/2792971788.jpg
/gpfs/gpfsfpo/pred_test/p_14296153.jpg
/gpfs/gpfsfpo/pred_test/8216542102.jpg
/gpfs/gpfsfpo/pred_test/5412441551.jpg
/gpfs/gpfsfpo/pred_test/p_2511878805.jpg
/gpfs/gpfsfpo/pred_test/5412441554.jpg
/gpfs/gpfsfpo/pred_test/p_534861364.jpg
/gpfs/gpfsfpo/pred_test/503215946.jpg
/gpfs/gpfsfpo/pred_test/8216542100.jpg

********** RESULTS ******************
[(u'8216542107.jpg', "IT'S A BIRD!"), (u'503215945.jpg', "IT'S A BIRD!"),
(u'2792971789.jpg', "IT'S A BIRD!"), (u'3655965983.jpg', "IT'S A BIRD!"),
(u'2792971788.jpg', "IT'S A BIRD!"), (u'p_14296153.jpg', "IT'S A BIRD!"),
(u'8216542102.jpg', "IT'S A BIRD!"), (u'5412441551.jpg', "IT'S A BIRD!"),
(u'p_2511878805.jpg', "IT'S A BIRD!"), (u'5412441554.jpg', "IT'S A BIRD!"),
(u'p_534861364.jpg', "IT'S A BIRD!"), (u'503215946.jpg', "IT'S A BIRD!"),
(u'8216542100.jpg', "IT'S A BIRD!")]
```

2.7.2 Library Requirements for Spark 1.4.0

1. pyspark.mllib.regression.LabeledPoint
2. pyspark.mllib.classification.LogisticRegressionModel
3. pyspark.mllib.linalg import.Vectors

4. pyspark.SparkContext
5. Python specific: os, glob, PIL, numpy, sys, gzip, argparse

3 Agile Development

Our team used the Agile methodology to organize the project. We had sprints, at 2 weeks each, with varying intensity for tasks. Sprints are chronological but stories are listed in no particular order. Tasks were created in a backlog which fed sprints based on new discoveries during development. All code is kept in a public repository: https://github.com/EvanKepner/Park-or-bird

3.1 Sprint: AC/DC

1. We need a MongoDB to store the Flickr files.
2. We need a proof of concept for Spark MLlib for image classification.
3. We need a park photo scraping process for non-Flickr images.
4. How can we contribute servers to a cluster across accounts in a single VLAN?
5. Create Python program to download Flickr data.
6. Create cluster and SSH keys.

3.2 Sprint: Black Sabbath

1. Install GPFS Cluster (to replace HDFS)
2. How do we handle multiple sizes/quality of photos?
3. Can we read ndarrays into RDDs or is a text file necessary?
4. Install Spark on the Hadoop and GPFS clusters.
5. Provision server with photo album to view photos.
6. Get average image dimensions from collection.

3.3 Sprint: Cake

1. Create a model for predicting clean data.
2. Identify clean data from gathered samples.
3. Run multiprocessing for images and create analysis files for Spark.
4. Train baseline Scala Spark model.
5. Get more parks!

6. Classification for images from MongoDB.
7. Holy compression Batman!
8. Multithreaded processing for images.
9. Can we set a pixel threshold to store more zeros in the analysis file for greater compression?
10. Modify image processing to reduce to RGB ratio cube.
11. Train more models.

3.4 Sprint: Deftones

1. Create a prediction engine to take raw images or directories and produce a label.
2. Implement PCA in Scala model.
3. How do we use PCA and Standard Scaler in the prediction engine?
4. Investigate algorithms for pre-processing with OpenCV.

4 Challenges

1. Originally we went with HDFS for shared storage, but the block size was problematic for storing multiple small files. We moved to GPFS and set a small block size.

2. Using the PyMongo API for insertion of records was too slow for our data volume. MongoImport was significantly faster and imported 100 million records in a matter of hours.

3. Our original image transformation code used Wand[17] and ImageMagick[18], which later had a release causing conflicts on our Centos 7 systems. For this reason we moved away from Wand and ImageMagick to PIL.

4. Initially resizing to the average image size produced analysis files that were too large. We scaled the average dimensions to 20% and implemented compression storage using the Python gzip library. This may have been too extreme for final quality; however, given the data quality constraints we proceeded at this scale for more rapid testing.

5. Our first pass at processing produced four output files: raw, raw blur, grayscale, grayscale blur. With the discovery of latency and storage requirements we trimmed to only raw output for the full feature set. With all four outputs on full features we estimated our volume would exceed 1 TB.

6. Processing images to create the Spark analysis file was prohibitively slow. Taking advantage of the integer IDs for our image storage, we increased the cluster node CPU capacity to 10 CPUs per node and assigned processing to each CPU based on the last digit of the image ID. Combined with the reduction to 20% of the average dimensions this dramatically increased processing speed.

7. Data from Flickr was not always good quality for park and bird categories e.g. family photos, park entrance signs, etc. We created a web portal where the photos could be viewed to assess quality, both in searchable form with Solr and as paged displays with Piwigo. We identified photos in each category as good or bad and trained a separate algorithm to prune the data set accordingly. With 1000 images of each we did not have sufficient labels to train for quality within each subset. A more robust quality mechanism is required.

5 Conclusion

We have demonstrated that Spark 1.4.0 can be used for image classification with logistic regression models in a scalable fashion. Our end-to-end solution provides a framework for:

1. Gathering image metadata in a searchable form stored in MongoDB.
2. Collecting images based on metadata criteria and storing them in a scalable cluster.
3. Examining and cleaning the data through a viewable portal.
4. Index images and metadata for further investigation and faceted search.
5. Training a machine learning model to further classify cleaned data given enough samples.
6. Determining the average dimensions of all gathered data for potential resizing.
7. Applying multiple image feature extraction techniques to the data based on resizing and scale invariant features such as the RGB block ratio or ORB algorithm.
8. Applying parallelization techniques to image processing, and storing all outputs in chunked and compressed form for analysis.
9. Training multiple machine learning models on diverse feature spaces, varying from 64 to 20,340 features across 400,000 samples.
10. Applying standard scaler methods and principal component analysis for dimensionality reduction.
11. Using the trained models in a deployable prediction engine built on PySpark which can take multiple model options, an input file, or an input directory, and generate classification predictions.

We have successfully applied the Agile development methodology to this data science initiative. Our work demonstrates how data science can be managed with an Agile project structure. Our framework will work on increasingly large systems. With GPFS, parallel image processing, and Spark, the system is fully elastic in its capability to scale.

There is the potential for continuation with this project:

1. Implement a data cleansing and labelling solution using our framework with Mechanical Turk or other crowdsourced application.
2. Run ORB feature extraction across the collection for analysis.
3. Test alternate models, such as a neural net, for image classification.

6 Architecture Diagrams

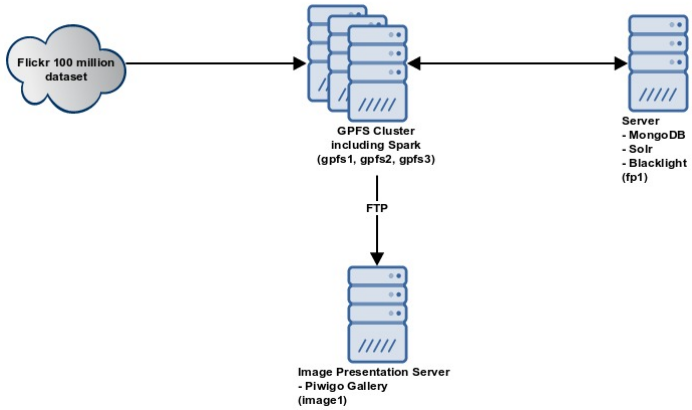

Figure 13: Cluster and systems infrastructure diagram.

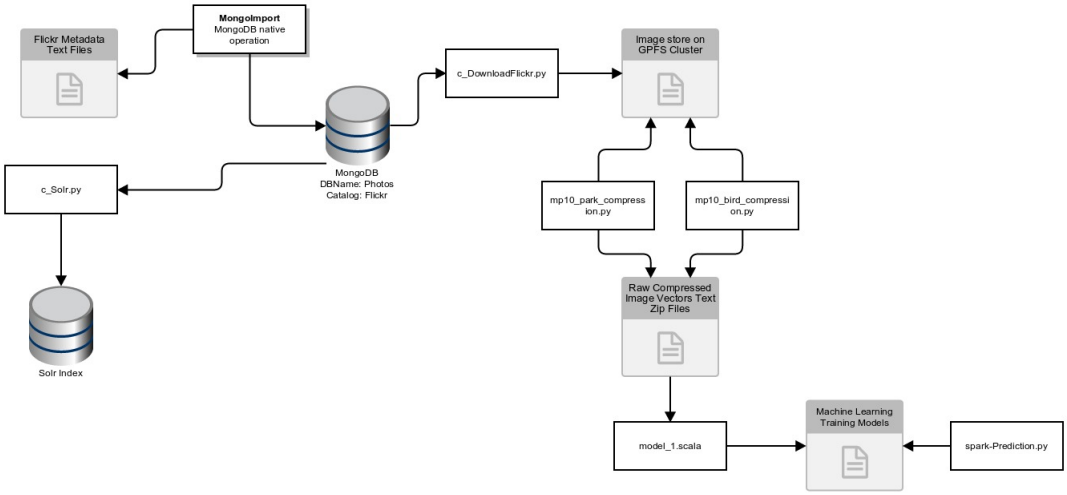

Figure 14: Software architecture diagram.

References

[1] http://xkcd.com/1425/

[2] http://labs.yahoo.com/news/yfcc100m/

[3] http://lucene.apache.org/solr/

[4] http://projectblacklight.org/

[5] http://piwigo.org/

[6] http://docs.python-requests.org/en/latest/

[7] http://www.crummy.com/software/BeautifulSoup/

[8] http://www.numpy.org/

[9] https://pypi.python.org/pypi/PIL/

[10] https://docs.python.org/2/library/multiprocessing.html

[11] http://www.ippatsuman.com/2014/08/13/day-and-night-an-image-classifier-with-scikit-learn/

[12] https://en.wikipedia.org/wiki/RGB_color_space

[13] http://opencv.org/

[14] Ethan Rublee, Vincent Rabaud, Kurt Konolige, Gary R. Bradski: ORB: An efficient alternative to SIFT or SURF. ICCV 2011: 2564-2571.

[15] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua, "BRIEF: Binary Robust Independent Elementary Features", 11th European Conference on Computer Vision (ECCV), Heraklion, Crete. LNCS Springer, September 2010.

[16] https://issues.apache.org/jira/browse/SPARK-6227

[17] http://docs.wand-py.org/en/0.4.0/

[18] http://www.imagemagick.org/script/index.php