TinyTags: Image Tagging and Indexing at Scale

Zach Beaver, Erin Boehmer, Nitin Kohli, Sharon Lin, Kunal Shah
UC Berkeley, Master of Information & Data Science
W251-2: Scaling Up! Really Big Data

What does the project do

Our project provides a means to automatically tag images with categorical labels and make the image dataset searchable through a front-facing search interface. Specifically, we use a distributed Random Forests machine learning algorithm to automatically tag a large dataset of images using a model built from a tagged subset of images. Our project leverages the data contained in the Tiny Image Dataset, which is a public dataset consisting of almost 80 million 32x32 images, amounting to ~250GB of data. To train the Random Forests classifier, available through Spark's MLlib, we use the tagged images included in the CIFAR-10 dataset.

Tiny Image Dataset: http://horatio.cs.nyu.edu/mit/tiny/data/index.html
CIFAR-10 Dataset: http://www.cs.toronto.edu/~kriz/cifar.html

The CIFAR-10 dataset contains 60,000 32x32 tagged color images, with 6000 images belonging to each of the 10 possible tag categories. Categories include: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Because the available tags are limited to 10 categories and used to train our classifier, we limit the image search to the 10 possible tag names.

In order to complete this project, we needed to transfer both the CIFAR-10 and Tiny Images datasets to an HDFS in the cloud, deployed via Cloudera. We then used Spark and its associated MLlib to train and test models that would tag the images using inherent features. After finding a suitable model to tag the remaining images in the dataset, we indexed the images and created a Solr search engine with a web UI front-end that allows users to query for images.

What Problem Is Being Solved?

While the problem of text organization and search is one that is fairly well-understood with mature solutions, automated organization and search over images is an area of active and aggressive research; entire labs are dedicated to the endeavor at some of the most well-known industry and academic institutions[1][2][3]. The primary dilemma is that image annotation is typically a tedious, manual, and subjective process. Current image search engines match user queries to user-annotated photos. This process is time-consuming and dependent upon the uploader with the shortcoming that two people could describe a photo in completely different ways; this

[1] http://vision.stanford.edu/
[2] http://research.microsoft.com/en-us/news/features/spp-102914.aspx
[3] http://googleresearch.blogspot.com/2014/09/building-deeper-understanding-of-images.html

significantly affects search results. In light of these shortcomings, automated image recognition (via machine learning) holds the promise of eliminating subjective, user-dependent annotation and ultimately saving people time with both image annotation and search.

Given that machine learning can eventually deliver on this promise of automated annotation, there must be a Big Data pipeline and ecosystem of tools that can effectively handle automated annotation (even retroactively) of large image corpuses. The goal of this project is to survey and utilize an appropriate set of Big Data tools necessary to ingest, automatically tag, index, and store large image corpuses. **As such, the value of this project is in creating an infrastructure necessary to support effective auto-annotation of images.** The applications of such a system include cloud-based organization of personal photo collections, standardizing image search criteria, reducing the amount of time needed to push image content via social media, and automating aerial recognition of disaster-related imagery. Image recognition research has advanced rapidly in recent years, and while wholesale image recognition is not yet mature, we want to develop the skills, strategy, and pipeline necessary to support such advances[4].

Big Data Tools

In order to complete the assignment, we decided upon a suite of Big Data tools, which includes:

- Apache Spark and Spark MLlib (deployed via Cloudera Manager)
- Python (PySolr, Pydoop, Flask, Scikit-Learn)
- IPython Notebook with PySpark support and YARN connection
- HDFS with Hue as UI (Deploy via Cloudera Manager)
- YARN as resource manager
- Softlayer (Deploy a YARN cluster using Cloudera Manager; host web application)
- Solr for index and search (Deploy via Cloudera Manager)

Building the Architecture

Setting up the Spark Cluster

We deploy a cluster that can handle both the size of the data and the speed required to process all the data. The cluster contains five nodes, one master and four workers, along with another node that runs Solr separately from the cluster. The cluster specs are:

Computing Instance: 4 x 2.0 GHz Cores
RAM: 8 GB (16 GB on node running IPython/PySpark)
Operating System: Ubuntu 14.04 - Minimal Install (64 bit)
First Disk: 25 GB (100 GB on node running IPython)
Second Disk: 200 GB (500 GB on node running Solr)
Uplink Port Speeds: 1 Gbps Public & Private Network Uplinks

[4] https://gigaom.com/2014/11/18/google-stanford-build-hybrid-neural-networks-that-can-explain-photos/

Once provisioned, the authorized_key file on each node is loaded with all group members' public ssh keys to enable password-less ssh. We also put all the nodes' IPs and hostnames into /etc/hosts to make sure each node can talk to one another. Next, we avoided network vulnerabilities by configuring a simple firewall. To better secure the cluster, we only opened ports specific to the services that are running on the cluster (for detailed steps on firewall setup, please reference the "Code" section at the end of this paper).

After setting up the cluster, we mounted the secondary disk in order to set up HDFS:

```
# You need to find out the name of your disk, e.g
fdisk -l |grep Disk |grep GB
# assuming your disk is called /dev/xvdc as it is for me,
mkdir -m 777 /hdfs
mkfs.ext4 /dev/xvdc
# add to to /etc/fstab
/dev/xvdc /hdfs                ext4    defaults,noatime        0 0
# now mount your disk
mount /hdfs
```

We decided to use Cloudera Manager to deploy services to our cluster given Cloudera conveniently supports most of the services we are interested in using:

```
$ wget http://archive.cloudera.com/cm5/installer/latest/cloudera-manager-installer.bin
$ chmod u+x cloudera-manager-installer.bin
$ sudo ./cloudera-manager-installer.bin
On master node, generate public/private keypair using: ssh-keygen
Copy the public key of the master node to all other nodes
Copy the private key of the master node to a file on your local machine, and upload to
the Cloudera UI when asked (use private key instead of password)
When setting up Cloudera for the first time, add only IPs (not hostnames) when asking
for hosts.
```

Using Cloudera Manager UI, the following services are deployed:

- HDFS: This is for all of our data storage. All output from Spark Machine Learning processing will be written to HDFS too.
- YARN: This is for Resource Management. Make sure all nodes have java installed. Master node installs java when Cloudera Manager runs but not worker nodes: apt-get install oracle-j2sdk1.7. Also adjust YARN's container memory parameter according to the RAM resource on the nodes.
- Spark: This is for our Machine Learning processing. This can also be done manually, but we deployed via Cloudera due to convenience.
- Hue: Interface for HDFS, requires installation of Hive and Oozie

Solr, the search engine that indexes tagged files, is installed on the node that will store the indexed data. This was installed on a node that is separate from the cluster, as it does not need any of the services provided through Cloudera. Solr was installed manually and comes with ZooKeeper (https://cwiki.apache.org/confluence/display/solr/Installing+Solr).

IPython Notebook Setup with PySpark and YARN Support

The instruction for setting up IPython Notebook with PySpark Support along with notes on setting up YARN connection:
http://ramhiser.com/2015/02/01/configuring-ipython-notebook-support-for-pyspark/
In addition, install pip and ipython: apt-get install python-pip, ipython, ipython-notebook

The PySpark support requires several paths added to the bashrc. Please reference the "Code" section for the specific path lines.

Potential YARN/HDFS Configuration Issue:
There may be a permission issue upon launching PySpark with YARN connection, when YARN tries to access HDFS. This is because when launching Spark, HDFS is used to sync code with YARN. If this is an issue, an error message will appear when you try to run Spark context commands in IPython Notebook. If so, change user to "hdfs" and create a user directory "root" using mkdir, and chown root:root. This will bypass the permission issue.

Access Cluster's IPython Notebook

In order to start IPython notebook, in one terminal, run: "ipython notebook --profile=pyspark"
To open the IPython Notebook UI in local web browser and bypass the firewall, we used the ssh tunneling command: "ssh -L port:localhost:port root@IP"

This allows access to the IPython Notebook on your local machine. On your local browser, go to: http://localhost:42424/

Spark is not necessarily faster when running multiple computations, but is faster when running *massive* computations. Any time an action is run on an RDD, the action has to be transmitted to all the executors (in this case, all five nodes), which can take some time. In order not to overcommit, we used the combination of YARN UI and htop to make sure processes are accepted and running.

From there, we built our algorithm using IPython Notebook and ran our jobs in IPython. We used byobu to run processes in the background, which avoids process termination when ssh loses connection.

Image Recognition with Random Forests in Spark

The primary goal of this project is to automatically create tags (or labels) for images. Creating an accurate classifier is a crucial piece of this project, but also a very time-consuming one to do well. Achieving near state-of-the art performance in image recognition can take months, even

for experienced researchers, as demonstrated by recent image recognition competitions on Kaggle[5]. At the beginning of the project, our ambitious goal was to try to automatically tag all types of images in the Tiny Images data set; we eventually narrowed our scope such that (1) we would attempt to only provide tags to images whose labels are present in the pre-tagged CIFAR-10 dataset and (2) we would only tag Tiny Images and make them available for search when the algorithm predicted a tag above a certain accuracy threshold.

Having thus narrowed the scope, our first task was to choose a classifier that was (1) supported by Spark and (2) provided a modicum of accuracy. Training a ML algorithm is an extremely iterative process, marked by trial and error over the course of many experiments. To facilitate this effort, we decided to separate our development and production environments. We made algorithm decisions as a team (in parallel through a common GitHub repository) by using Scikit-Learn on our local machines; this was our development environment before transferring the final algorithm decisions to Spark. We chose this 'local dev environment' approach for a few reasons:

1. It allowed us to begin developing the algorithm while the Spark environment was being readied by another team member.
2. The labeled training data is actually small (50,000 images), which can be further divided and does not necessitate distributed computing. The real need for distributed computing comes with trying to automatically tag the 80 million Tiny Images data set.
3. In development, iteration time is key. Working with a library we were familiar with (Scikit-Learn) allowed us test multiple algorithms rapidly.
4. We could work in parallel without worrying about resource constraints of running multiple Spark jobs at once.

After local development, we translated the hyper-parameters (with some tweaks due to memory limitations and hyper-parameter support in Spark) to Spark's MLlib for implementation and trained on the entire CIFAR-10 data set (see Appendix "Local Development in Scikit-Learn).

We then completed the final code development in two phases

1) Learning a model:
   We imported train data (CIFAR-10) into Spark as RDDs. After the import, we run the RandomForestClassifier training step on the RDD. We trained it on 10 trees in the forest.
2) Predicting using the model:
   a) The entire dataset was a ~250GB bin file. And even after splitting the file into 80 smaller bin files to transfer into HDFS, each individual bin file was still too big for the driver memory to process both transformation into two-dimensional array and RDD conversion. Therefore, we further split each bin file into 50 RDDs, which were much smaller. This solved our input data problem.

[5] http://benanne.github.io/2015/03/17/plankton.html

b) We wanted to predict the accuracy of our results as well. i.e. how many of the trained trees agreed on the output label. This was difficult in Spark, especially in PySpark, as there is no currently no library supporting accuracy output. Therefore, we have to create a function for outputting accuracy.
c) We want to only keep output with certain levels of accuracy. Therefore, from the local ML development, we determined that 40% is a good cutoff threshold for accuracy. When predicting on the test dataset, we chose not to write out an output if the accuracy falls below 40% (i.e. less than 40% of the trees in the Random Forest "voted" for the particular class). This reduces the amount of "contentious," and therefore likely inaccurate, data we have to write out.
d) We still had to choose a good output format for our results that would make it easily indexable. (NOTE: We could have sent the content of an RDD directly to Solr, but this would not be a good approach in case the process fails. Therefore, we separated indexing from output production). We decided to output a JSON dictionary with the RDD block number, feature vector, label, and the accuracy.

After these steps, we have an indexable JSON output file for each input RDD file.

Indexing the Processed Data Set

Once the RDDs have gone through the algorithm, it outputs a tuple of four fields for each image: the tag, which ranges from 1 to 10, accuracy, which ranges from 4 to 10, the RDD block number, which ranges from 0 to 3999, and the raw bytes. If the accuracy is less than 4, the image will be discarded and will not be written out. We then sent the data to Solr for indexing. We first created a new schema for the raw data as we don't want it to be indexed. A new schema called "data" is created as a field that's not indexed but is stored. Aside from the raw data that is stored as a non-indexed JSON field, we indexed three total fields: the tag, the accuracy, and the number of the RDD it comes from. The last one is mainly for if indexing runs into any errors. In such cases, we can tell which RDD it stops at, delete all the data in Solr from that RDD and restart from that RDD. Otherwise, because Solr randomly assigns index ID, if indexing runs into any error, restarting without knowing the stop point will be extremely difficult.

Because of the large amount of data that needs to be indexed, we ended up provisioning an additional node with a 750GB secondary disk. Non-indexed fields require a constant amount of drive space, but indexed fields require about 2x as much space on the hard drive during Solr processing. Therefore, a large amount of disk space was needed to process the image JSON. The Solr node was not provisioned as part of the cluster as it does not use the services from Cloudera, but rather communicates only with the node from the cluster that is sending the processed JSON image data.

Building a Web-based User Interface

With the indexed and stored JSON data residing on the Solr VM, our last step was to build an interface providing users with dynamic access to the tagged image data through search. To do this, we use the Flask microframework which provides a convenient MVC framework with simple

decorators for web traffic routing. The app is located on the Solr server, for faster query response time, and can be accessed at http://192.155.209.244:5000/ This framework also allows for simple integration with Solrpy, which is a python library made specifically to facilitate solr integration through the following commands:

```
import solr
s = solr.SolrConnection('http://localhost:8983/solr/TinyImages_shard2_replica1')
...
response = s.query('*:*', fq=fq_str, rows=30)
for hit in response.results:
        # do something
```

For the complete code base of the Flask webapp, please navigate to the public github repo for this project: https://github.com/Erin-Boehmer/MIDS_tinytags

The app consists of two main pages: an index page where users can search for tagged images and an about page (with information about our team). The index page contains a query panel where users can select [1, 10] category tags and an accuracy range from [40%, 100%]. These search options are based on the CIFAR-10 data which contained 10 possible tag labels and the accuracy derivation strategy for our classification algorithm. An image that was tagged with 4 trees agreeing on the label will receive an accuracy of 40%. An image with a tag agreed upon by all 10 trees will have an accuracy of 100%. These accuracies do not, by any means, imply that the tag is 100% accurate, but instead reflect the degree of agreement among trees.

When the user hits “Search,” the form is submitted using an AJAX post to the Flask @query route. The query method within the python controller then passes a request via Solrpy to the solr instance that contains the image JSON data in the general form of:

```
response = s.query('*:*', fq="label_i:(1 OR 2 OR 8) AND num_tree_i:[8 TO 10]", rows=30)
```

The response is limited to 30 results rather than the default of 10 (for pagination purposes, given the small size of the images). Solr returns the 30 JSON records as unicode, which the controller cleans into a python dictionary structure (using numpy) and then passes as JSON back to the AJAX request via python’s jsonify method.

On success, the Javascript code uses the JSON to dynamically create HTML/Twitter Bootstrap thumbnails for each image. The thumbnails are also parsed to display the accuracy and label associated with each image. We had originally planned to use matplotlib to translate the pixel data into visible imagery. However, the images were more easily transferred from controller to view using JSON. Within the view, the Javascript uses HTML canvas elements to convert the pixels into actual images. Each pixel array contained 3,072 elements, where each consecutive triplet contained a red, green, and blue coding of the pixels true color. The arrays had already been reshaped using numpy in the controller, but for display in HTML canvas, an alpha pixel was added and the data was transferred to a Uint32Array for faster renderings speeds.

Sample Data and Output

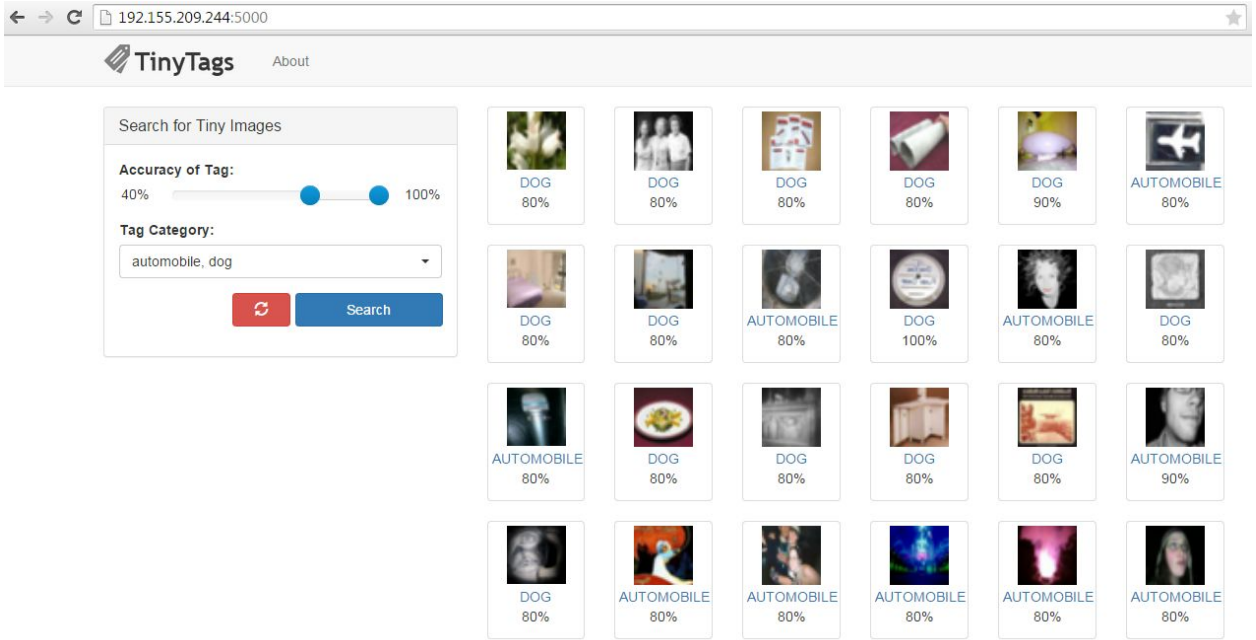
Solr Query Response

NOTE: This is a truncated sample of the data response (the full response has a total of 3072 pixel values in the “data” array)


```
{
 "responseHeader": {
  "status": 0,
  "QTime": 886,
  "params": {
   "indent": "true",
   "q": "label_i:1\n",
   "_": "1430274706983",
   "wt": "json"
  }
 },
 "response": {
  "numFound": 1442966,
  "start": 0,
  "maxScore": 4.2996345,
  "docs": [
   {
    "rdd_num_i": 47,
    "data":
"[42,41,42,45,44,50,53,48,49,66,73,76,80,76,61,40,48,71,64,58,63,73,73,48,40,58,61,60,60,49,50,
47,51,52,44,42,42,46,53,54,53,59,61,65,70,78,74,52,47,63,57,47,46,55,62,57,45,59,72,77,67,57,54
,48,54,54,47,42,41,42,50,55,73,69,57,57,58,69,79,78,73,69,63,49,41,46,52,60,57,49,56,80,75,59,5
7,58,53,54,52,50,44,45,50,51,83,85,67,61,54,60,74,78,78,67,62,53,44,40,35,43,66,69,59,69,81,62,
52,50,57,56,55,56,49,47,52,59,78,74,60,60,59,59,73,84,82,71,62,55,45,40,29,26,41,60,64,62,67,63
,63,59,69,56,54,56,55,53,58,62,70,53, ...]",
    "label_i": 1,
    "num_tree_i": 4,
    "id": "a626f718-c00e-4dc3-8b81-0733df30f61b",
    "_version_": 1498863366282674200
   },
   {
    "rdd_num_i": 47,
    "data":
"[135,134,129,115,97,70,33,19,11,58,89,51,33,33,33,42,60,74,80,83,84,74,62,66,63,73,79,85,88,8
6,86,81,105,110,102,88,73,47,36,31,36,68,76,61,54,52,50,54,64,75,81,82,84,67,53,58,59,67,69,82,
87,88,88,88,74,106,107,90,69,58,52,66,90,89,79,68,61,61,57,57,67,77,83,84,85,69,55,61,69,77,82,
88,90,90,87,88,65,90,105,85,86,98,90,82,82,81,73,64,59,62,61,65,72,78,86,87,84,76,66,70,82,85,8
7,88,89,89,88,88,59,93,103,86,102,107,110,102,93,87,78,69,59,59,62,60,55,75,83,87,87,87,86,85,
86,87,87,86,88,89,87,87,72,107,97,117,123, ...]",
    "label_i": 1,
    "num_tree_i": 4,
    "id": "f8a38aea-d8b9-488e-b91e-5fe47f9ef7be",
```

```
      "_version_": 1498863366355026000
    }
  ]
 }
}
```


Data Output from User Interface



Challenges

Breaking Down a Large Single File

One of the biggest issues we spotted right after determining the dataset is that the data is stored in a single bin file of ~80 million image raw data at around ~250GB. In order to pipe the data into HDFS, we decided to break the bin files into 80 smaller bins of ~1 million image raw data each. Once the bin files are in HDFS, our next challenge is to convert the files into RDD objects that are compatible with SparkContext. Because bin files lack data structure, before converting the file into RDD, we would have to first give it structure. Because each bin file is around 3GB, we find that our next challenge is we would run out of driver memory because IPython has to hold each bin file in memory when converting to array before writing the entire set out as RDD. To bypass this problem, we decide to further break each bin files into 50 RDD-compatible pickle files, thus creating 4000 RDD-compatible objects.

Accuracy Challenges

As this is a “Big Data” class, our primary focus was to create an image ingestion and search platform at scale. Although accurate Machine Learning is important for image tagging, state-of-the-art image recognition normally takes months of research and experiments.

Recognizing this, we decided to focus upon the platform's infrastructure while performing enough machine learning experiments to obtain a minimally viable product. We attempted several image feature extraction techniques but did not see major improvement. Please reference the "Improve Machine Learning Algorithm Accuracy" section in the Appendix for detail.

Alternative Strategies

Before developing the aforementioned pipeline, we considered a few other strategies, most of which contained subtle variations on the process we implemented. However, one of the approaches we discussed was a more drastic departure from the path we pursued; this alternative strategy was to use a small GPU cluster, Python's Theano library, and Convolutional Neural Networks to perform image recognition.

Convolutional Neural Networks are state-of-the-art and have yielded the highest accuracies in many image recognition tasks[6]. However, they are almost prohibitively expensive (computationally) to train. Since the backpropagation algorithm of neural nets can be expressed as a series of matrix multiplications, this training time can reduced by orders of magnitude by utilizing GPUs. Python's Theano library has been developed to utilize GPUs in training Neural Networks and would thus be well-suited to the task.

The benefits of a GPU-based approach are:

- State of the art classification rates
- Ability to test ML algorithms that Spark does not currently support (e.g. Neural Network variations)
- Experience using GPUs, which are becoming increasingly utilized in industry

The drawbacks are:

- Overcoming a steep learning curve in understanding how to properly train a Convolutional Neural Network. Neural Nets notoriously have exorbitant numbers of hyper-parameters requiring a tacit and nuanced understanding of their architecture. Because this is a "Big Data" class, we thought the majority of our time for this project should not be spent trying to understand a specific machine learning algorithm.
- Inability to use Spark. In choosing a problem to solve, our group resolved to find a task that allowed us to use Spark, since that skill set is increasingly in demand for data scientists.

Our group ultimately decided that these drawbacks outweighed the benefits, but we are still curious as to how a GPU-based approach would fare in relation to our realized strategy. Some open questions that group members may later consider include:

- What is the accuracy trade-off between "out-of-the-box" random forest and convolutional neural net implementations?

[6] http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf

- What is the monetary difference between using GPU and CPU-based clusters?
- What are the impacts on development time for switching from random forests to convolutional neural nets?
- Is there a way to maximize the cost/benefit of accuracy vs. training time for a GPU-based approach a priori?

Appendix

Improving Machine Learning Algorithm Accuracy

Thus, our team wanted to optimize the accuracy of predictions generated by a model from Spark's limited Machine Learning algorithm base using some feature engineering techniques that could be accomplished within a limited time frame.

To begin, we divided the CIFAR-10 data into a smaller subset to locally run and test classification accuracy results after feature engineering. Any resulting accuracy improvements would then be translated into the SparkContext and deployed with our system.

We tried several techniques for image feature extraction. First, we applied edge detection algorithms (Sobel and Roberts) to understand whether edge detection would help the machine distinguish object shapes and boundaries. Using the skimage library, we found that this did not in fact improve accuracy. When combining both the colored dataset and edge detection data (effectively doubling the size of the training and testing sets for the Roberts and Sobel algorithms separately), the accuracy fell from 38.4% to 37.9% (Sobel) and 36.6% (Roberts).

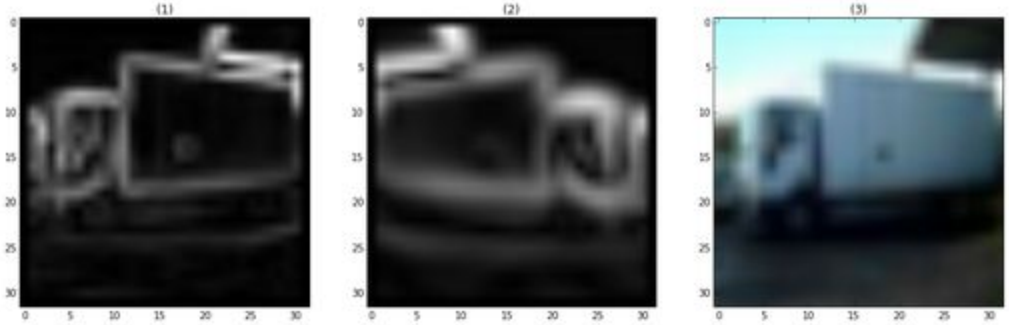


Figure: (From left to right) Roberts Edge Detection, Sobel Edge Detection - rotated with Gaussian blur applied, and original Tiny Image

We then decided that perhaps flipping the images to portray both a left-facing and right-facing object would improve algorithm accuracy. But again, upon applying this method to the image dataset, we saw the Random Forests classifier accuracy fall from 38.4% to 34.7%. This seemed counter-intuitive, since we were effectively doubling the size of our training data and flipping images to account for possible object orientations, but the fall in accuracy showed this transformation to actually confound patterns rather than clarify them.

We then tried to use the Harris corner detection algorithm to account for some unexplained image variance, thinking that the edge detection may help to distinguish meaningful features such as cat ears or vehicle boundaries. Alas - this also led to a drop in accuracy by about 9.1%.

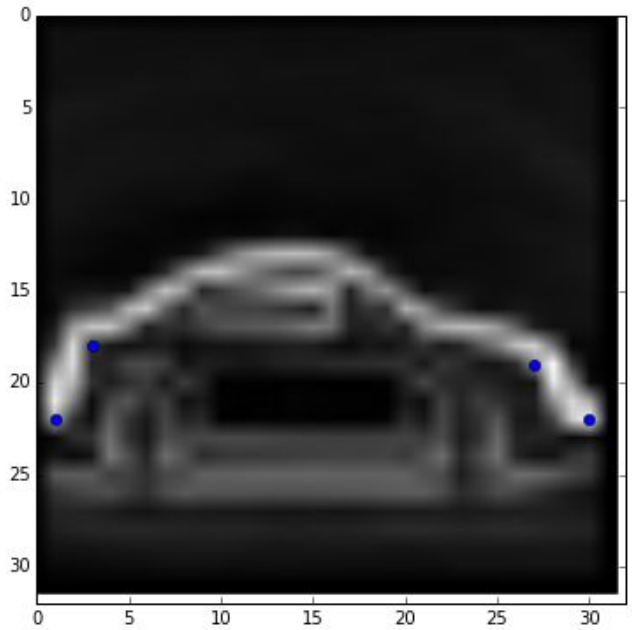


Figure: Harris corner detection showing the object boundaries in a Tiny Image of a car.

Furthermore, we then decided to use Principal Component Analysis (PCA) to see if we could reduce the dimensions of our inputs variables. Given the 32 x 32 x 3 input data, our features alone lived in 3,072 dimensional space. If PCA were to prove to be successful, our run time would decrease as we would not need to explore all thousands of features, but a significantly smaller set of linear combinations of them.

Running PCA on a subset of the training data locally we found that the first 100 principal components explained ~90% of the variance in the data. However, reducing the problem from 3,072 space to 100 space led to a decrease in accuracy, despite having found a lower dimensional subspace that explained 90% of the variability. One possible explanation for this may be the image data violates the assumption that it resides in a linear subspace. Hence, by imposing PCA we are further distorting the true nature of the data.

After several rounds of feature engineering that led to declines in classifier accuracy or modest improvements that could easily be attributed to random variance in model performance, we decided to forego these algorithms and opt for processing without preprocessing. This does have the added benefit of avoiding added processing time using our Spark cluster, but results in a raw classifier model that will likely be inaccurate for the majority of Tiny Images.

Local Development in Scikit-Learn

In testing subsets of the CIFAR-10 data we received the following accuracy results on hold-out test sets of images:

- Support Vector Machines: 9.8 %
- K-Nearest Neighbors: 29.2 %
- Random Forests: 38.4 %

- Boosted Decision Trees: 28.35 %
- Logistic Regression with L1 Regularization: 27.75 %
- Logistic Regression with L2 Regularization: 36.65 %
- Multi-class Adaboost: 29.75 %

Random Forests provided the highest percent accuracy. We subsequently used grid search, cross-validation to find optimal hyper-parameters for the model. The hyper-parameters yielding the highest accuracies were:

- Minimum Number of Observations per Leaf: 1
- Minimum Number of Observations required for a tree to split: 2
- Number of estimators (trees): 500

However, after this hyper-parameter tuning, our accuracy remained around 50%. Wanting to be more certain of our tagging accuracy, we delved into how trees in the Random Forest 'voted.' We discovered that our model was well calibrated and that when at least 30% of trees agreed on the same class, the accuracy of the model was 80% for this subset of observations. Having made these decisions, we then transferred our findings to Spark, where we trained a Random Forest model on the entire CIFAR-10 data set and then predicted tags for the entire Tiny Image data set.

Code

All codes can also be found in our Github repository:
https://github.com/Erin-Boehmer/MIDS_tinytags

Firewall Setup - Run in Terminal

```
$ sudo ufw allow 22  # ssh
$ sudo ufw allow 8080  # running web server
$ sudo ufw allow 7077  # Spark port for service to listen on
$ sudo ufw allow 8081  # running web server
$ sudo ufw allow 8088  # YARN Webapp, only need to do this on master
$ sudo ufw allow 7180  # Cloudera Manager Server Web UI
$ sudo ufw allow 7182  # Cloudera Manager Server RPC, only need to do this on master
$ sudo ufw allow 8888  # Hue Server, only need to do this on Hue node
$ sudo ufw allow 5000  # Port for web UI; only need to do this on Solr node
$ sudo ufw allow from 192.155.209.243 to any port 8983 #only need to do this on Solr node
$ sudo ufw allow from 192.155.209.243 to any port 7574 #only need to do this on Solr node
$ sudo ufw allow proto tcp to any port 48000:48020  # For Softlayer monitoring
$ sudo ufw allow from 192.155.209.242
$ sudo ufw allow from 198.23.92.100
$ sudo ufw allow from 23.246.214.6
$ sudo ufw allow from 23.246.250.162
$ sudo ufw allow from 192.155.209.243
$ sudo ufw allow from 127.0.0.1
$ sudo ufw enable
```

Bashrc Paths

```
export YARN_CONF_DIR="/etc/hadoop/conf.cloudera.yarn/"
export SPARK_HOME="/opt/cloudera/parcels/CDH/lib/spark/"
export PYSPARK_SUBMIT_ARGS="--master yarn-client --executor-memory 6g --driver-memory 14g
--conf spark.driver.maxResultSize=2g --conf spark.akka.frameSize=200 --conf
spark.storage.memoryFraction=1 --conf spark.core.connection.ack.wait.timeout=600 --conf
spark.rdd.compress=true"
export JAVA_HOME="/usr/lib/jvm/java-7-oracle-cloudera/"
export PATH=$PATH:/usr/lib/jvm/java-7-oracle-cloudera/bin/
export HADOOP_HOME=”/opt/cloudera/parcels/CDH/”
```

Bin Splitting

```
infile = open('/Users/Username/Desktop/tiny_images.bin', 'rb')
byte_count = 32 * 32 * 3

image_count = 79302017
for i in range(0, 80):
    outfile_name = "/Volumes/My Book 3.0/images/%sm_%sm.bin" % (str(i), str(i + 1))
    outfile = open(outfile_name, 'wb')
    for j in range(i*1000000, min((i+1)*1000000, image_count)):
        data = infile.read(byte_count)
        outfile.write(data)
    outfile.close()
```

```
    print("wrote out %s" % outfile_name)
```

RDD Splitting

```
import os
import numpy
from pyspark import SparkConf, SparkContext
import pydoop.hdfs as hdfs

spark_home = os.environ.get('SPARK_HOME', None)

for i in range(0, 80):
    infile = '/user/admin/%sm_%sm.bin' % (str(i), str(i + 1))
    f = hdfs.open(infile)
    byte_count = 32 * 32 * 3
    total_bytes = f.size
    for j in range(0,50):
        images = []
        for k in range(0,total_bytes/byte_count/50):
            data = f.read(byte_count)
            images.append(data)
        rdd = sc.parallelize(images)
        outfile = '/user/admin/RDD/rdd%s' % str(i*50+j)
        print 'Writing %s' % outfile
        rdd.saveAsPickleFile(outfile)
        print 'Wrote %s' % outfile
```

ML Processing

```
import os
import numpy
import cPickle
import sys
import operator
import json

from pyspark import SparkConf, SparkContext
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import RandomForest, DecisionTreeModel
from pyspark.mllib.classification import SVMWithSGD

import pydoop.hdfs as hdfs

spark_home = os.environ.get('SPARK_HOME', None)

def _unpickle(file):
        fo = open(file, 'rb')
        dict = cPickle.load(fo)
        fo.close()
        return dict

def _loadtraindata():
        data = [_unpickle('/root/cifar-10-batches-py/data_batch_1'),
        _unpickle('/root/cifar-10-batches-py/data_batch_2'),
        _unpickle('/root/cifar-10-batches-py/data_batch_3'),
        _unpickle('/root/cifar-10-batches-py/data_batch_4'),
        _unpickle('/root/cifar-10-batches-py/data_batch_5')]

        trainingdata = []
```

```
        traininglabels = []

        for data_i in data:
        trainingdata = trainingdata + data_i['data'].tolist()
        traininglabels = traininglabels + data_i['labels']
        return trainingdata, traininglabels

def createTrainRDD():
        (trainingdata, traininglabels) = _loadtraindata()
        labeled = [(x, traininglabels[i]) for i,x in enumerate(trainingdata)]
        trainrdd = sc.parallelize(labeled)
        trainlprdd = trainrdd.map(lambda x: LabeledPoint(x[1], x[0]))
        return trainlprdd

def createRFModel():
        trainlprdd = createTrainRDD()
        model = RandomForest.trainClassifier(trainlprdd, numClasses=10,
categoricalFeaturesInfo={},
                                                numTrees=10, featureSubsetStrategy="auto",
                                                impurity='gini', maxDepth=10, maxBins=50)
        return model

model = createRFModel()

def predict_proba(rf_model, testRDD):

        trees = rf_model._java_model.trees()
        ntrees = rf_model.numTrees()
        scores_dict = {i: 0 for i in range(0,10)}
        scoresRDD = testRDD.map(lambda x: scores_dict.copy())

        for tree in trees:
                dtm = DecisionTreeModel(tree)
                currentScoreRDD = dtm.predict(testRDD)
                scoresRDD = scoresRDD.zip(currentScoreRDD)

                def reduceTuple(x):
                        x[0][int(x[1])] += 1
                        return x[0]

                scoresRDD = scoresRDD.map(reduceTuple)
        return scoresRDD

def getClassifiedRDD(rdd_file):
        testdataRDD = sc.pickleFile(rdd_file)
        testimageRDD = testdataRDD.map(lambda x: numpy.fromstring(x, dtype='uint8').tolist())
        scoresRDD = predict_proba(model, testimageRDD)

        finalRDD = testimageRDD.zip(scoresRDD)

        def finalizeRDD(x):
                max_label = -1
                max_value = -1
                for k in x[1]:
                        if x[1][k] > max_value:
                                max_value = x[1][k]
                                max_label = k
                return {'data': x[0], 'label': max_label, 'num_trees': max_value}

        finalRDD = finalRDD.map(finalizeRDD)
        finalRDD = finalRDD.filter(lambda x: x['num_trees'] >= 4)
```

```
        return finalRDD

def saveOutputRDDs(start, end):
        for i in range(start, end):
                infile = '/user/admin/RDD/rdd%d' % (i)
                outfile = '/user/admin/outputRDD/rdd%d' % (i)
                outputRDD = getClassifiedRDD(infile)
                outputRDD = outputRDD.map(lambda x: json.dumps(x))
                outputRDD.saveAsTextFile(outfile)
                print 'Wrote %s' % outfile

saveOutputRDDs(0, 4000)
```

Solr Indexing

```
import os
import subprocess
import pydoop.hdfs as hdfs
import json
import requests

def solr_post(json_line, rdd_num):
  image = json.loads(json_line)
  json_dict = {'add':{'doc':{'label_i':image['label'],
                             'rdd_num_i':rdd_num,
                             'data':json.dumps(image['data'], separators=(',',':')),
                             'num_tree_i':image['num_trees']}, 'commitWithin':1000}}
  r =
requests.post("http://192.155.209.244:8983/solr/TinyImages_shard1_replica1/update?wt=json",
    headers={'Content-type':'application/json'},
    data=json.dumps(json_dict))

for i in range(0, 4000):
  infile = sc.textFile('/user/admin/output10RDD/rdd%d' % (i))
  infile.foreach(lambda l: solr_post(l,i))
  print 'Completed file rdd%d' % (i)
```

Solr Query

// From Flask app.py

```
@app.route('/query', methods=['POST'])
def query():
    # create the query string for tags
    tag_lookup = {
        "airplane":"0",
        "automobile":"1",
        "bird":"2",
        "cat":"3",
        "deer":"4",
        "dog":"5",
        "frog":"6",
        "horse":"7",
        "ship":"8",
        "truck":"9" }
    tag_str = ""
```

```
    if request.json['tags']:
        for tag in request.json['tags']:
            if(tag_str == ""):
                tag_str = tag_lookup[str(tag)]
            else:
                tag_str = "%s OR %s" % (tag_str, tag_lookup[str(tag)])
    # if tag_str is not blank, finalize for addition to the query
    if tag_str is not "":
        tag_str = "label_i:(%s) AND" % (tag_str)


    # create the query string for accuracy
    low_acc = int(request.json['accuracies'][0])/10
    high_acc = int(request.json['accuracies'][1])/10
    accuracy_str = 'num_tree_i:[%d TO %d]' % (low_acc, high_acc)

    # create the filter query and submit to solr
    fq_str = "%s %s" % (tag_str, accuracy_str)
    response = s.query('*:*', fq=fq_str, rows=30)

    # create the image json response
    data = {"images": []}
    for hit in response.results:
        # getting the json response data in the right format
        pixel_data = str(hit['data'])
        pixel_data = pixel_data[1:-1].split(",")
        clean_data = np.asarray([(int(x)) for x in pixel_data])
        html_ready_pixels = clean_data.reshape(3,1024).swapaxes(0,1)
        html_ready_pixels = np.c_[html_ready_pixels,
np.zeros(html_ready_pixels.shape[0])].flatten().tolist()

        # create the json return object
        image = {"data":html_ready_pixels, "label_i":hit['label_i'],
"num_tree_i":hit['num_tree_i']}
        data["images"].append(image)

    return(jsonify(data))
```

Image Conversion

For complete code, please see the github repo at:
https://github.com/Erin-Boehmer/MIDS_tinytags

Image Conversion Using Matplotlib/Numpy

```
##################################################
### --- Assumes [rrr...ggg...bbb] format --- ###
##################################################

### --- Necessary Setup --- ###
```

```python
import json
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline

### --- Read in the JSON --- ###
with open("imageSample.json") as json_file:
    json_data = json.load(json_file)

### --- Pick out the relevant field and convert from Unicode to Numpy Array--- ###
image = str(json_data['data']) #extract relevant info from the json and parse as string
image = image[1:-1].split(",") #drop off the front and back brackets from string list and split into an array
image_clean = np.asarray([(255-int(x)) for x in image]) #reverse the scale and parse as nparray
proper_image = image_clean.reshape(3,1024).swapaxes(0,1).reshape(32,32,3) #reshape data to plot with plt

### --- Display--- ###
plt.imshow(proper_image)
plt.show()
```

Image Conversion Using Javascript and HTML Canvas

NOTE: This is a snippet of the full code, specifically the “success” function of an AJAX request for solr response data

```
...
success: function(msg) {
                                tag_lookup = {
                                    "0":"airplane",
                                    "1":"automobile",
                                    "2":"bird",
                                    "3":"cat",
                                    "4":"deer",
                                    "5":"dog",
                                    "6":"frog",
                                    "7":"horse",
                                    "8":"ship",
                                    "9":"truck"
                                }

                            var html_response = "";
                            for (image_index in msg.images) {
                                image = msg.images[image_index]
                                html_response = html_response +
                                '<div class="col-md-2">' +
                                    '<div class="text-center thumbnail">' +
                                        '<canvas id="' + image_index + '" width="32"
height="32" class="rotate90 col-md-12">Your browser does not support the HTML5 canvas
tag.</canvas>' +
```

```
                                        '<center><h5
class="text-primary">'+tag_lookup[image.label_i].toUpperCase() + '</h5></center>' +
                                        '<center><h5 class="move-up">'+ (image.num_tree_i*10)
+ '%</h5></center>' +
                                    '</div>' +
                                '</div>'
                            }
                            $('span#image_results').html(html_response);

                            for(var i=0; i < msg.images.length; i++) {
                                pixel_data = msg.images[i].data;

                                var canvas = document.getElementById(i);
                                var ctx = canvas.getContext('2d');
                                var imageData = ctx.getImageData(0, 0, canvas.width,
canvas.height);

                                var buf = new ArrayBuffer(imageData.data.length);
                                var buf8 = new Uint8ClampedArray(buf);
                                var data = new Uint32Array(buf);

                                for (var y = 0; y < canvas.height; ++y) {
                                    for (var x = 0; x < canvas.width; ++x) {
                                        img_index = y*canvas.width + x;
                                        pixel_index = img_index*4;
                                        data[img_index] =
                                            (255   << 24) |                       // alpha
                                            (pixel_data[pixel_index+2] << 16) |    // blue
                                            (pixel_data[pixel_index+1] <<  8) |    // green
                                             pixel_data[pixel_index];              // red
                                    }
                                }

                                imageData.data.set(buf8);

                                ctx.putImageData(imageData, 0, 0);
                            }
                        },
                    error: … // continue with AJAX request
```

Web UI

// Partial view from templates/index.html

```
{% block content %}
    <div class="row">
        <div class="col-md-4">
            <div id="searchpanel" class="panel panel-default">
                <div class="panel-heading"><h3 class="panel-title">Search for Tiny
Images</h3></div>
                <div class="panel-body">
```

```
                    <form id="query_form" class="form-horizontal">
                        <div class="form-group">
                            <label class="col-md-12" for="slider">Accuracy of Tag:</label>
                            <div class="col-md-2" style="margin-left: 2px">40%</div>
                            <div class="col-md-8" style="margin-left: -7px">
                                <input id="slider" type="text" class="span2" value=""
data-slider-min="40" data-slider-max="100" data-slider-step="10" data-slider-value="[50,90]"/>
                            </div>
                            <div class="col-md-2" style="margin-left: -3px">100%</div>
                        </div>

                        <div class="form-group">
                            <label class="col-md-12" for="multiselect">Tag Category:</label>
                            <select id="multiselect" class="selectpicker col-md-12" multiple>
                                <option>airplane</option>
                                <option>automobile</option>
                                <option>bird</option>
                                <option>cat</option>
                                <option>deer</option>
                                <option>dog</option>
                                <option>frog</option>
                                <option>horse</option>
                                <option>ship</option>
                                <option>truck</option>
                            </select>
                        </div>

                        <div class="form-group">
                            <button id="query_button" type="button" style="margin-right: 14px"
class="btn btn-primary col-md-5 pull-right">Search</button>
                            <button id="refresh" type="button" style="margin-right:5px"
class="btn btn-danger pull-right col-md-2" aria-label="Reset">
                                <span class="glyphicon glyphicon-refresh"
aria-hidden="true"></span>
                            </button>
                        </div>
                    </form>

                </div>
            </div>
        </div>
        <div class="col-md-8">
            <div class="row">
            <span id="image_results">
                <div class="jumbotron">
                    <h1>Welcome to TinyTags!</h1>
                    <div class="row">
                        <div style="margin-right: 5px; margin-left: 20px;" class="thumbnail
col-md-1"><img src="{{ url_for('static', filename='img/airplane.png') }}"
```

```
alt="airplane"></div>
                        <div style="margin-right: 5px;" class="thumbnail col-md-1"><img
src="{{ url_for('static', filename='img/automobile.png') }}" alt="automobile"></div>
                        <div style="margin-right: 5px;" class="thumbnail col-md-1"><img
src="{{ url_for('static', filename='img/bird.png') }}" alt="bird"></div>
                        <div style="margin-right: 5px;" class="thumbnail col-md-1"><img
src="{{ url_for('static', filename='img/cat.png') }}" alt="cat"></div>
                        <div style="margin-right: 5px;" class="thumbnail col-md-1"><img
src="{{ url_for('static', filename='img/deer.png') }}" alt="deer"></div>
                        <div style="margin-right: 5px;" class="thumbnail col-md-1"><img
src="{{ url_for('static', filename='img/dog.png') }}" alt="dog"></div>
                        <div style="margin-right: 5px;" class="thumbnail col-md-1"><img
src="{{ url_for('static', filename='img/frog.png') }}" alt="frog"></div>
                        <div style="margin-right: 5px;" class="thumbnail col-md-1"><img
src="{{ url_for('static', filename='img/horse.png') }}" alt="horse"></div>
                        <div style="margin-right: 5px;" class="thumbnail col-md-1"><img
src="{{ url_for('static', filename='img/ship.png') }}" alt="ship"></div>
                        <div class="thumbnail col-md-1"><img src="{{ url_for('static',
filename='img/truck.png') }}" alt="truck"></div>
                    </div><br>
                    <p>Use the search panel to get started searching for tagged images within
the MIT datastore of 80 million tiny images.</p>
                </div>
            </span>
            </div> <!-- END INDIVIDUAL IMAGE ROW -->
        </div> <!-- END IMAGE RESULTS COLUMN -->
    </div> <!-- END MAIN PAGE ROW -->

{% endblock %}
```

Local Dev for Random Forests

Note: Code originally ran in IPython Notebook

```
# coding: utf-8

# In[1]:

#Import the necessary libraries
import cPickle
from sklearn.ensemble import RandomForestClassifier
from sklearn import grid_search
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
#%matplotlib inline


# In[2]:
```

```
#Function to unpickle the image files and return the CIFAR dictionary containing the data and
the labels
def unpickle(file):
    fo = open(file, 'rb')
    dict = cPickle.load(fo)
    fo.close()
    return dict


### Reading in the First batch

# In[3]:

#Read in the first batch of data (the batch must reside in a folder called 'data' that resides
in the parent folder of the current directory)
#The pickle file contains two elements --> 'data' (numpy array) and 'labels' (numeric list)
batch1 = unpickle("../data/data_batch_1")

#Read in the label names for these images
label_names = unpickle("../data/batches.meta")


### Split into test and train and run first pass on RF

# In[4]:

def split_test_train(cifar_data, seed_val):
    '''This function takes in one of the cifar datasets and randomly splits the data into test
(20%) and train (80%).
    It returns a dictionary with four data structures:
    1. train_data (numpy array)
    2. train_labels (list)
    3. test_data (numpy array)
    4. test_labels (list)'''

    from numpy.random import seed

    #set a seed and randomly split into training and test
    seed(seed_val)

    #Get the data and labels
    all_data = cifar_data['data']
    all_labels = cifar_data['labels']

    #Randomly sample indexes from the data
    indexes = np.random.choice(len(all_data), size = len(all_data), replace = False)

    #Split into Test and Train
    train_amount = int(round(0.8 * len(indexes)))
    train_data = all_data[indexes[:train_amount]]
```

```
    train_labels = [all_labels[i] for i in indexes[:train_amount]]
    test_data = all_data[indexes[train_amount:]]
    test_labels = [all_labels[i] for i in indexes[train_amount:]]

    #Put it all in a dictionary and return the dictionary
    split_data = {"train_data": train_data, "train_labels": train_labels, "test_data":
test_data, "test_labels": test_labels}
    return split_data


### The split function creates a dictionary with 'train_data', 'train_labels', 'test_data',
'test_labels'

# In[5]:

#split the data into test and train
split_data = split_test_train(batch1, 4)


# In[ ]:

###Run First pass of Random Forest###
rf_baseline = RandomForestClassifier(n_estimators=500)
rf_baseline.fit(split_data['train_data'], split_data['train_labels'])
print 'baseline accurracy is', round(rf_baseline.score(split_data['test_data'],
split_data['test_labels']), 3) * 100


# In[8]:

###Add More trees### --> no help
rf_baseline = RandomForestClassifier(n_estimators=1000)
rf_baseline.fit(split_data['train_data'], split_data['train_labels'])
print 'increased tree accurracy is', round(rf_baseline.score(split_data['test_data'],
split_data['test_labels']), 3) * 100


# In[19]:

#Lets use some cv and check a couple other parameters
rf_params = {'n_estimators': [500], 'min_samples_split': [1, 2, 3, 5, 7, 10],
'min_samples_leaf': [1, 2, 3, 5, 7, 10]}
rf = RandomForestClassifier()
clf = grid_search.GridSearchCV(rf, rf_params)
clf.fit(split_data['train_data'], split_data['train_labels'])
print 'The best accuracy is', round(clf.score(split_data['test_data'],
split_data['test_labels']), 3) * 100


# In[20]:
```

```
print clf.best_params_


# In[23]:

#Lets use some cv and check a couple other parameters
rf_params = {'n_estimators': [10,25,50,100,200,300,400,500,600,700]}
clf = grid_search.GridSearchCV(rf, rf_params)
clf.fit(split_data['train_data'], split_data['train_labels'])
print 'The best accuracy is', round(clf.score(split_data['test_data'],
split_data['test_labels']), 3) * 100


# In[24]:

print clf.best_params_


### add more data and see how the accuracy increases

# In[ ]:

batch2 = unpickle("../data/data_batch_2")


# In[ ]:

split_data2 = split_test_train(batch2, 5)


# In[ ]:

#Combine the data
all_train_data = np.concatenate((split_data['train_data'], split_data2['train_data']), axis =
0)
all_train_labels = split_data['train_labels'] + split_data2['train_labels']
all_test_data = np.concatenate((split_data['test_data'], split_data2['test_data']), axis = 0)
all_test_labels = split_data['test_labels'] + split_data2['test_labels']


# In[ ]:

#Quick sanity check --> dimensions look good
print all_train_data.shape, len(all_train_labels), all_test_data.shape, len(all_test_labels)


# In[21]:

new_rf = RandomForestClassifier(n_estimators=500, min_samples_leaf=3, min_samples_split=5)
```

```
new_rf.fit(all_train_data, all_train_labels)
print 'increased tree accurracy is', round(new_rf.score(all_test_data, all_test_labels), 3) *
100


# In[ ]:

new_rf2 = RandomForestClassifier(n_estimators=500, min_samples_leaf=1, min_samples_split=2)
new_rf2.fit(all_train_data, all_train_labels)
print 'increased tree accurracy is', round(new_rf2.score(all_test_data, all_test_labels), 3) *
100


### Checking Calibration of Model --> Seeing if Tree Voting is Overconfident

# In[ ]:

prob_list = list()
all_probs = new_rf2.predict_proba(all_test_data)


#Get a list of tuples showing the probability and predicted labels for the most likely class
for i in range(len(all_test_labels)):
    for prob in all_probs[i]:
        if int(all_test_labels[i]) == list(all_probs[i]).index(prob):
            prob_list.append((prob, True))
        else:
            prob_list.append((prob, False))



#Make the histogram
true_vals = list()
false_vals = list()

#Separate the true and false values into separate lists
for item, value in prob_list:
    if value:
        true_vals.append(item)
    else:
        false_vals.append(item)

#Grab a list with all digit probs >=  0.9
upper_cal_true = float(len([x for x in true_vals if x >= 0.9]))
upper_cal_false = float(len([x for x in false_vals if x >= 0.9]))
print type(upper_cal_false)
print "0.9 - 1.0 Model calibration score: ", upper_cal_true / (upper_cal_true +
upper_cal_false)

#Grab a list with all digit probs <=  0.1
```

```
lower_cal_true = float(len([x for x in true_vals if x <= 0.1]))
lower_cal_false = float(len([x for x in false_vals if x <= 0.1]))
print "0.0 - 0.1 Model calibration score: ", lower_cal_true / (lower_cal_true +
lower_cal_false)

#plot the histogram
plt.figure()
plt.hist([np.array(true_vals), np.array(false_vals)], 10, stacked = True)
plt.xlabel("Model probabilities")
plt.ylabel("Count of probabilities")
plt.show


# In[ ]:

plt.figure()
plt.hist([max(i) for i in all_probs], 10)
plt.xlabel("Model probabilities")
plt.ylabel("Count of probabilities")
plt.show
```