Capstone Project: Machine Learning Nanodegree

Xiaojian Deng
6/25/2016

Definition

Project Overview

The basic idea behind this project is to create an **image classifier**: a program which takes an image as input and returns a relevant string describing the image as output. Such a program can have powerful real-world application in many situations including:

- Unattended searching and scrapping of the web for pictures of certain objects.
- Automatically categorizing new goods for sale on an ecommerce website
- Intercepting and flagging inappropriate images that users uploaded on a forum, image board, or social network for moderators to review.
- Programming a robot to perform certain functions whenever it "sees" particular objects (i.e. a robot that identifies and picks apples from a tree.)

Ideally, we'll want to employ "incremental learning" in order to allow the classifier to rapidly incorporate information from new images that's presented to it on demand rather than retrain from scratch every time you want to add a new image to its knowledge base. More on incremental learning later in this write-up.

In the beginning of this study, I'll be focusing on images of 3 different types of "objects": Apples, Birds, and Cats

The "dataset" for this project consists of 100 images each of apples, birds, and cats taken from the free stock photo site Pixabay.com by searching and handpicking the top results for each keyword.

Note: This program further expands off of the "Bag of Words" Image Recognition program by Bikramjot Singh Hanzra (see: https://github.com/bikz05/bag-of-words). While the initial strategy for image processing and training remains similar at a higher level, my implementation and analysis goes much deeper, focusing on choosing optimal machine learning algorithms and parameters to further improve recall, accuracy, and/or speed as well as implementing incremental learning to incorporate new images on the fly rather than training a single static dataset.

Problem Statement

The purpose of this project is to train a classifier to identify and label images. For example, if you feed the classifier a picture of a cat, it should return the string "cat" (as long as it's properly trained.) Depending on the classifier used, the program will find the label that's most suitable for the image based on the "features" it observes in the image. More on features later.

Training the classifier involves the following stages:

1. Loading the images and their corresponding labels.
2. Extracting features from the images.
3. Simplifying the features through clustering and/or dimensionality reduction
4. Obtaining a "histogram", a distribution of image features among all known clusters found during K-means clustering.
5. Training and/or updating the classifier.

Similarly, predicting an image's label involves the following stages:

1. Loading the image
2. Extracting features from the image
3. Use the PCA and KMeans we trained back in the "training" steps to simplify this image
4. Obtaining a histogram of image features.
5. Predicting the label using the classifier we trained earlier by feeding it the transformed image.

Metrics

To measure the accuracy of our program, I tested it on out-of-sample images to find an F1 Score, which is the harmonic means of precision and recall. To fully understand F1 Scores, we'll discuss what Precision and Recall mean, especially in the context of image recognition.

Precision is defined as (True Positives) / (True Positives + False Positives.) Precision, True Positives, and False Positives only make sense in the context of a specific image class that's among all of the ones our classifier is identifying (for example, apples.) The True Positive measure for each class is the total number of images of that class that is correctly identified. The False Positives measure is the total number of images incorrectly labeled as that class. For example, a picture of a cat labeled as "apple" would be a False Positive with respect to apples.

Recall is defined as (True Positives) / (True Positives + False Negatives). False Negatives, with respect to a particular class, are the # of images not belonging to that class and are labeled as anything other than that class. With respect to the "apple" class, a cat labeled as a "cat" would be a False Negative as well as a bird labeled as a "cat" (even if that classification is incorrect.) Intuitively, Recall, with respect to a class, measures how well the classifier can identify images that do not belong in that class.

Since there are more than 2 different classes in our dataset, there are multiple ways to calculate an F1 Score, but I've chosen the Micro-average method which can be useful when the dataset varies in size. (In the real world, not all image classes will have roughly the same # of members in every data set.)

Thus, the Micro F1 Score can be expressed as $2*(\Sigma\text{Precision} * \Sigma\text{Recall})/(\Sigma\text{Precision} + \Sigma\text{Recall})$

Where $\Sigma\text{Precision} = \Sigma(\text{True Positives}) / \{\Sigma(\text{True Positives}) + \Sigma(\text{False Positives})\}$

And $\Sigma\text{Recall} = \Sigma(\text{True Positives}) / \{\Sigma(\text{True Positives}) + \Sigma(\text{False Negatives})\}$

In addition, training and testing time will be tracked though the main focus will be on improving the F1 scores with reducing training/testing time as a secondary objective.

Analysis

Data Exploration

At a higher level, the dataset consists of high quality photos in .JPG and .PNG format that focus on 1 or more objects of a particular category (in this specific case, apples, birds, or cats.) There are usually no other prominent objects located in the foreground of these pictures (say, a dog next to a cat when we're training a machine to recognize cats which might confuse the classifier.) I tried not to be too selective in picking the images for my dataset (unlike the experiments discussed in some other Computer Vision papers) since I'd rather have my algorithm to perform satisfactorily on real-world images rather than deliver superior performance in a controlled environment but poorly in the real world. (Later, I've also demonstrated the classifier's performance on more controlled images with well-defined images and little extraneous backgrounds.)

At a lower level, a computer perceives an image as a matrix of tuples. Each entry in the matrix represents the color of a point in the image, and each pixel is denoted by a tuple representing the intensities of the Red, Green, and Blue pixels mixed together to create the appropriate hue that pixel is supposed to take. The number of ways a computer can accurately represent an image of a particular object, like an apple, is extremely vast.

Since this is essentially a supervised learning problem for training a computer to recognize certain patterns: pictures of objects and to associate words with these objects, it's imperative that the data is transformed optimally. Even a tiny 100x100 image with 255 possible values for each of its Red, Green, and Blue pixels that make up every one of its points will have 100*100*255*255*255 or over 165 billion possible images. Reducing the dimensionality of a picture's pixel data and/or using clustering to find similarly colored pixels adjacent to each other will make the problem of identifying objects more computationally realistic than feeding a machine learning algorithm raw pixel data.

One of the algorithms I'll be using to extract Features from images is SIFT (Scale Invariant Feature Transform.) Each photo will typically contain hundreds or even thousands of potential Features and each Feature will be represented as a 128-dimensional vector. One major advantage of SIFT is that it can still detect Features that have been rotated or scaled bigger or smaller though the tradeoff is a longer processing time. (In the Exploratory Visualization section, I'll demonstrate the effect of SIFT on my dataset; other algorithms for extracting features will be explored further.)

Here's an overview of the statistics of the features that the SIFT algorithm extracted from my dataset:

CATEGORY	**# Images**	**Min**	**Max**	**Median**	**Mean**	**SD**
Apples	100	41	6072	424	600.90	705.08
Birds	100	76	3951	866.5	1098.96	923.26
Cats	100	129	4266	1221	1396.27	954.28
COMBINED	300	41	6072	755	1032	927.94

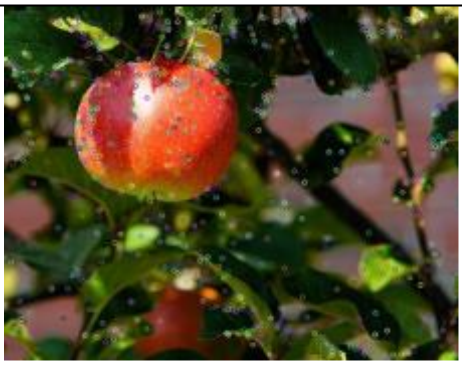
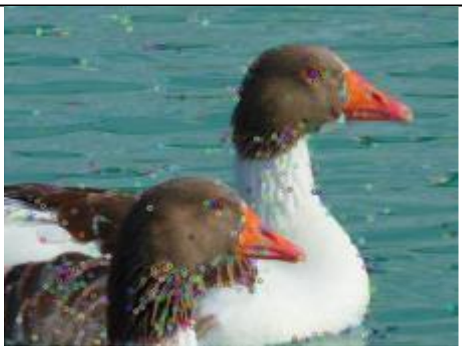
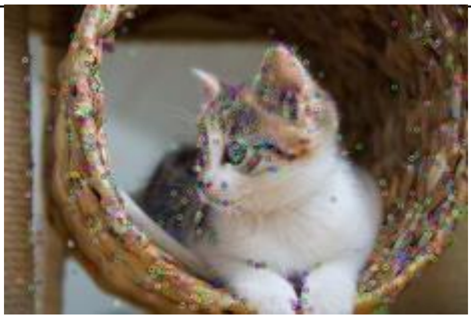
Though it'll be more apparent in the graph shown in the next section, the number of features extracted from each image in the data set is nowhere near a normal distribution. Most images have a relative small number of features but there are a few outliers with many thousands of features. The mean and even the SD (Standard Deviation) are much higher than the median which suggests the disproportionate effects of these outliers on the dataset.

Later on, it's important to consider the effects that these outlier images with thousands of features (usually an image with a large and complex background) will have on our Incremental Learning algorithms. Incrementally training a classifier on a small batch of images with a single outlier especially early on will likely create a bias and decrease its accuracy. The side effects from outliers will likely be less pronounced when they're part of a larger batch, but larger batches also have their drawbacks and a tradeoff will be further discussed further in this report.

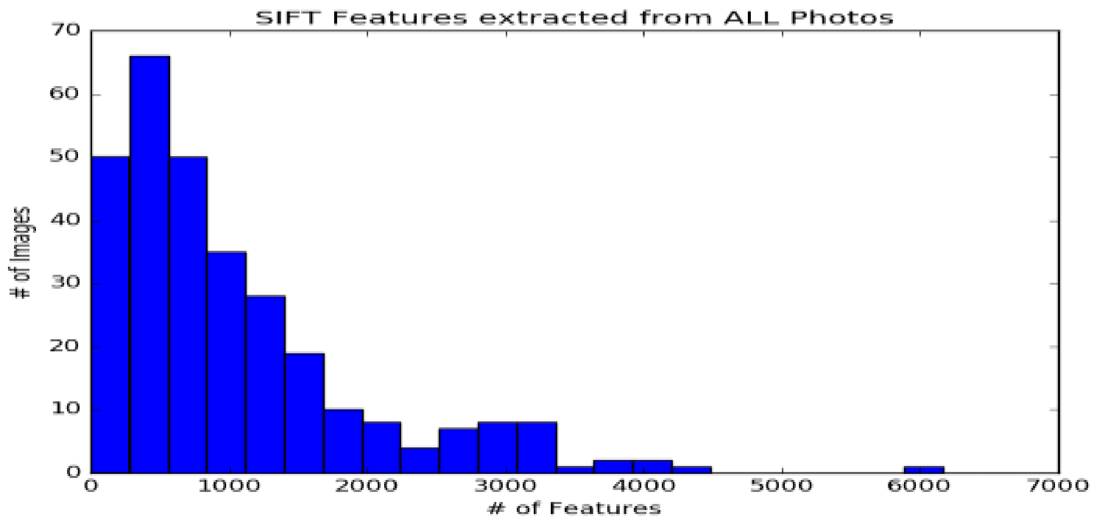
Exploratory Visualization

Here are examples of images from my dataset (on the left) and the various points on them where the SIFT algorithm has extracted Features from. The colored circles represent micro regions of the picture containing key Features, as deemed by the SIFT algorithm:

Original Image	**SIFT Features Circled**

As you can see from the sample photos above, the feature extraction algorithm did not distinguish between the objects of interest in the photos and the background "noise". This is expected and not a problem for now since it's my job to teach the computer to recognize the distinguishing features of each type of object. In order to do this, we'll need to perform clustering on the features to create a feature histogram but before we can choose the # of clusters, we'll need to look at the distribution of the # of Features that are extracted from each image in the chart below.



From the chart and the stats calculated earlier, each image yields an average of about 1000 Features with the vast majority yielding below 1000 Features and a few outliers yielding 3000+. Since speed is also a priority in addition to accuracy and the vast majority of the data lies in the lower end of the range, it makes more sense to pick algorithms and their respective parameters to best deal with images with around 500-1500 features.

Algorithms and Techniques

The entire process can be divided into several stages. Parameters that'll be tested at each stage will be noted:

- **Stage 1: Getting the Data**
  - Load the images from file.
  - Randomly divide the images into a training and a testing set.
  - Choose whether to convert the image to Greyscale or not.
  - Extract features from each image with a feature extraction algorithm.
    - SIFT: 128-dimensional feature vector. A method of generating features from an image that's invariant to translation, scaling, and rotation. Created by David Lowe in 1999.
    - SURF: 64-dimensional feature vector. An alternative and faster method of finding features based off of SIFT. Created by Herbert Bay in 2006.
    - ORB: 32-dimensional feature vector. A faster but less accurate feature generator which looks for corners in an image. Created by Ethan Rublee in 2011.
    - Choose whether or not to apply the RootSIFT extension.
- **Stage 2: Preprocessing the Data**
  - We'll divide the data into clusters using K-Means clustering and to generate a histogram for each image showing the distribution of feature vectors residing in

each cluster. PCA will be used to reduce the dimensionality of the data here. Note that another parameter is whether we'll do K-means or PCA first.
        - PCA first then K-Means
        - K-Means first then PCA
    - K-Means Clusters = 25, 50, 100, 200
    - sklearn.cluster.MiniBatchKMeans: This is the variation of the K-Means clustering algorithm we'll use because it can adjust its cluster centers with the addition of new data (aka incremental learning.) This is achieved through stochastic gradient descent.
    - PCA Parameters: N-components = X*(# of dimensions) where X = 0.25, 0.5, 0.75, and 1.0. We'll use sklearn.decomposition.IncrementalPCA which also supports incremental learning by only storing estimates of its component and noise variances when updating the explained_variance_ratio_. Compared to PCA, it's loss in accuracy is minimal.
    - Perform TF-IDF Vectorization after generating the histogram. TF-IDF was originally designed for classifying and querying text documents more efficiently by lowering the weights of the "filler" words. We'll use TF-IDF to lower the weights of the "filler" clusters in the histogram generated from K-Means.
- **Stage 3: Training the Data:** Start out by using the default settings in a classifier that supports Incremental Learning such as:
    - **sklearn.naive_bayes.BernoulliNB**: A class of Naïve Bayes classification algorithms which converts the histogram to a multivariate binary distribution (of 0 and 1's.) This may speed up the training and reduce the possibility of overfitting but may also over-simplify the data.
    - **sklearn.linear_model.SGDClassifier**: This is an implementation of a Linear Support Vector Machine Classifier (LinearSVC) which uses stochastic gradient descent and supports incremental learning. Its learning rate decreases over time and places less importance on later training examples. This can be useful if this classifier pretrained and packaged with an app for identifying very specific classes of images.
    - **sklearn.linear_model.Perceptron**: A special version of the SGDClassifier with a constant learning rate (which places the same importance on earlier and later training examples.)
    - **sklearn.linear_model.PassiveAggressiveClassifier**: This incremental learning classifier works much like a SVM but with 2 differences. First, before it trains on new data, it predicts it and if the answer is correct, it won't train on that data. Second, if it's incorrect, then tries to make minimal adjustments to the hyperplanes until the classifier gets the correct answer (unlike a SVM/SVC which seeks a maximum-margin hyperplane.) This may limit overfitting in some datasets but will also put a higher bias on the first few training examples.
- **Stage 4: Test/Predict/Train Again until the model is optimal**
    - Test several combinations of the above parameters on the SAME train/test split.
    - Then generate a new train/test split and test the SAME combinations of parameters that were tested in the previous set.

    - This will ensure that the results in each set are comparable (by fixing the test/train split.)
    - And averages, ranges, etc. for the performance of each set of parameters can be determined by varying the train/test split used to find them.

<u>Summary of Parameters:</u>

- #1: Which feature extraction algorithm to use (SIFT, SURF, or ORB)
- #2: Convert the image to Grey before extracting features? (Y/N)?
- #3: Use the RootSIFT extension (Y/N)?
- #4: Perform PCA before or after K-Means (Y/N)?
- #5: # of PCA components (25%, 50%, 75%, or 100% of original dimensions)
- #6: # of K-means clusters (25, 50, 100, 200)
- #7: TF-IDF Vectorization? (Y/N)?
- #8: Which Classifier to use? (4 different ones)

Benchmark

Since this is a classification problem and the success/failure of each classification attempt is binary (either it got the right classification or it didn't), the F1 score provides a reasonable metric for a benchmark. Keep in mind that with 3 classes, you can achieve a precision and recall of 1/3 each by random guessing which amounts to an F1 score of 1/3. Now a classifier that just doesn't beat or even just barely beats random guessing isn't very impressive so let's set the bar higher and shoot for an F1 score that's twice of that: 0.67.

The F1 score will be calculated by training a classifier (with specific parameters as outlined in the previous section) and then benchmarking it by running it on an out of sample testing set.

Here's a similar study done on 20 different categories of food images(with 100 samples in each.) F1 scores by category range from 0.11 to 0.64 with an average F1 score of about 0.3.

Methodology

Data Preprocessing

As previously mentioned, many of the hundreds or even thousands of features extracted from each picture are irrelevant to the object in interest so it's important to perform clustering to see if certain image classes contain large numbers of features aggregated in a few key clusters. The only clustering algorithm in Sklearn that supports Incremental Learning is MiniBatchKMeans (a variation of K-Means) so we'll use that.

Similarly, the # of dimensions in the feature vectors extracted is quite vast, from 32-dimensions for an ORB vector to 128-dimensions for a SIFT vector so it's worth running PCA to see if we can reduce some of the unnecessary dimensions and to minimize the prospect of overfitting on the noise from the less relevant dimensions.

See "Stage 2: Preprocessing the Data" for details on the various algorithms used here.

Implementation

(Note: please follow along in the iPython Notebook and/or source code for examples pertaining to this section.) For explanations for the various algorithms used here, please refer back to Stages 3 and 4 in the "Algorithms and Techniques" section.

The **classifier** class has a constructor that sets the parameters (like classifier, feature-finding algorithm, # of K-means clusters, etc.) It loads a set of images with the getFeatures() member function, extracts their features, and preprocesses those features using KMeans and PCA before storing the processed features as a member variable. At this point, you can either run train() which will train the classifier to recognize the new images or test() which will get the classifier to predict the classes which those images belong to.

The **image.py** file is pretty self-explanatory: an **image** class which contains the image's pixel data and the corresponding category that the user assigned to the image. When benchmarking the classifier, the CLF will try to predict what category the image belongs to based on its pixels and then check to see if it matches the category specified in the member variable.

The **tools.py** file contains an **environment** class which handles the loading of the images/data as well as creating the training/test split via rand_train_test(). The **images.csv** file contains the relative paths to all of the image files in the dataset along with their respective categories. The constructor for the environment class takes such a CSV file as input and returns an environment which contains a list of all image objects in the dataset as well as a list of unique categories those images belong in.

There's a key member function in the environment class called **benchmark**() which takes 12 parameters. Parameter 1 specifies the # of Training Images for each category. Parameter 2 specifies the # of Testing images for each category. Parameter 3 specifies the total # of training and testing runs to make before calculating averaged stats. **Each "run" will generate a unique random training-testing split!** Parameters 4 to 12 are the specific classifier parameters to use. The benchmark() function will print the average # of seconds spent on Training and Testing as well as the F1 score (averaged among the # of runs specified in parameter 3.)

Complications

One complication I encountered was that some images have very few features, fewer than the # of K-Means clusters. The probability is small but if we happen to pick one of these images first while training a new classifier (one by one via Incremental Learning), then the program will crash. To remedy this, if we find that the first image that we try to train has too few features, the Mini Batch K-Means classifier will temporarily reduce the number of clusters and then ramp it back up to the intended # again in subsequent calls of partial_fit(().

Also, partial_fit() often refuses to run whenever the sample you're fitting has only 1 class so it's not possible to immediately fit every image. (Imagine we're implanting this on a mobile device and app is trying to fit images on the fly whenever the user takes a picture.) To get around this, I've implemented a queue local variable (train_queue) and partial_fit() gets called on this queue whenever images of 2 or more classes are added. **However, even then, the resulting F1 scores are pretty poor when calling partial_fit() on small queues so I've introduced another parameter: minimum queue size (threshold) before calling partial_fit() when doing incremental training.** This parameter can't be too large either or else the batch will take too long to train, resulting in a poor user experience. Values for minimum queue size I'll test are between 3 and 25.

Another complication I encountered was calculating the F1 score. Since I had more than 2 classes, it threw a warning asking for a specific type of F1 score. I chose "micro" (see http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html) which counts the global false positives, true negatives, etc.

Finally, the classifier often erroneous results when given a single image to classify at a time (for example, labeling all pictures as "cat".) In order to solve this, at least 1 "dummy" image must be inserted into the list if the classifier is receiving only 1 image to classify. The predict_and_visualize() member function in the classifier class relies on this since it takes a single image as an input and outputs a new image with the predicted class written on the image.

Refinement

I created the function rand_grid_search() as a member function of the environment class for testing combinations of parameters. It creates a number of training-testing splits specified by the **sets** variable where each split contains a number of samples from every class as specified by the **trainsize** and **testsize** variables. The variable **tries** specifies how many combinations of parameters the algorithm should randomly choose to test.

I decided to run a Grid Search with the following parameters which took roughly ~12 hours to complete:

- **trainsize**: 50 from each class
- **testsize**: 50 from each class

- **sets**: 5 training/testing splits
- **tries**: 1000 (out of a total of 2560 possible combinations)
- **Clf**: SGDClassifier, BernoulliNB, Perceptron, or PassiveAggressiveClassifier
- **Feature_finder**: ORB only (to make searching faster, we can then try switching to SURF or SIFT on the most optimal parameters.)
- **Convert_grey**: True or False
- **Rootsift**: True or False
- **Pca_before_kmeans**: only True
- **Kclusters**: 25, 50, 100, or 200
- **Pca_ratio**: 0.25, 0.5, 0.75, or 1.0
- **Tfidf**: True or False
- **Incremental_threshold**: 5, 10, 15, 20, or 25

The results are output into a Pandas dataframe sorted by F1 score in descending order. I've outputted the results into the file **results.csv**.

I've also compared the parameters in the Top 50 F1 scores with the entire population of 1000 parameters to get a sense which parameters lead to better results.

CLASSIFIER	% Occurrence in Top 100	% Total Occurance
SGDClassifier	17%	25.9%
BernoulliNB	34%	25.3%
Perceptron	19%	25.7%
PassiveAggressiveClassifier	30%	23.1%

Verdict: Either BernoulliNB or PassiveAggressiveClassifier. As explained previously, BernoulliNB uses binarization of the input data which may lead to loss of information and result in inconsistency, so I'd go with **PassiveAggressiveClassifier** with its comparable performance.

Next, let's take a look at the K-Means clusters, PCA Ratio, and Incremental Threshold which take discrete values and compare their means in the Top 100 vs the entire population:

PARAMETER	Mean in Top 100	Mean in Total Population
Kclusters	138.75	90.25
Pca_ratio	0.672	0.625
Incremental_threshold	20.35	15.125

Verdicts:

- **Kclusters** = 200. Since our choices are 25, 50, 100, and 200 and the mean is 138.75 in the Top 100, 200 is likely overrepresented in the top results.
- **Pca_ratio** = 0.5 or 0.75. There's little difference in the two means so we'll pick 0.5 in favor of slightly reducing computation time.
- **Incremental_threshold** = 25. Since the maximum value tested was 25 and the mean in the top 100 is over 20, 25 is likely overrepresented in the top results.

Finally, we'll analyze the binary (True/False) parameters by comparing the % True in the two different samples:

PARAMETER	% True in Top 100	% True Total
Convert_grey	46%	50.1%
Rootsift	50%	49.4%
Tfidf	35%	51.5%

Verdict: **Covert_grey** and **Rootsift** can be set to either True or False according to this data. In principle, they should both be set to **True** (this will be explained in detail in the next section.) TF-IDF measurably decreases F1 scores as evidenced by its lower occurrence in the Top 100 so that should also be set to **False**.

Final Parameters:

- **Clf**: PassiveAggressiveClassifier
- **Feature_finder**: SIFT (though we used ORB for fast testing)
- **Convert_grey**: True
- **Rootsift**: True
- **Pca_before_kmeans**: True
- **Kclusters**: 200
- **Pca_ratio**: 0.5
- **Tfidf**: False
- **Incremental_threshold**: 25

Let's compare the performance between SIFT and ORB using this final set of parameters:

Algorithm	Train Time	Test Time	F1 Score
SIFT	61.89	60.30	0.58
ORB	5.21	4.01	0.47

Results

Model Evaluation and Validation

While we didn't achieve our target F1 score of 0.67, I believe the parameters chosen are in line with my expectations. The F1 score is a bit low but that could be due to our initial choice of a data set. More on that later.

Further analysis on the parameters chosen:

- **Algorithm: SIFT**. SIFT is by far the most sophisticated candidate algorithm for extracting features from images, creating 128-dimensional feature vectors compared to 64-D and 32-D vectors from SURF and ORB respectively.

- **Convert to Grayscale? True**. The ORB and SIFT algorithms are designed to work only on greyscale images so it makes sense that performance and/or accuracy will take a hit if they are fed color images (if an error isn't raised.)
- **RootSIFT adjustment? True**. The SIFT algorithm traditionally uses Euclidian distance instead of Hellinger (Chi-Squared) distance to measure similarity between images, but according to the blog post introducing RootSIFT and a 2012 paper, the Hellinger distance is more reliable for image classification.
- **PCA before K-Means? True**. Setting this parameter to False threw an error which was justified after further investigation - since K-means is used in the histogram generation and not as a preprocessing step, it makes sense that PCA should be performed prior to it in order to reduce the dimensionality of the data.
- **K-Means Clusters: 200**: Images are very sophisticated so it makes sense to cluster their features into a large number of categories. But having too many clusters will impact performance so a tradeoff has to be made.
- **# of PCA vectors: 50% of original:** Since the # of dimensions depends on the algorithm chosen (SIFT, ORB, etc.), it makes sense to set the PCA components to a % of the original # of dimensions. Picking 50% of the PCA vectors with the highest explained variance ratios should result in less than 50% of the original variance being lost in the new dimensional space, resulting in greater efficiency.
- **TF-IDF vectorization? False**: A small but measurable difference in F1 Scores has been observed when TF-IDF is turned off. A possible explanation might be that due to the use of incremental learning: the cluster centers are constantly changing as new data is dynamically added while TF-IDF relies on the cluster centers being constant over time.
- **Threshold in incremental training? 25**: It makes sense that incremental training works better when it's fed more samples in a single batch. Queuing up images and training them in larger batches seems to give greater F1 scores but there's also a tradeoff. End users would much rather deal with a small delay after submitting every 5, 10, or 20 photos than a long delay after submitting, say, every 100 photos as the program processes them all for incremental learning. According to our stats, it took about 62 seconds to train 150 images which translates to a 10 second delay on every 25th image.
- **Classifier? sklearn.linear_model.PassiveAggressiveClassifier**: This classifier, at a higher level, learns from its mistakes instead of trying to fit everything. This resembles how intelligent animals (including humans) learn to interact with the world including how to identify and interpret images.

Perturbations and Unseen Data

In order to test the effectiveness of the algorithm on different datasets, we'll use the Caltech101 Data Set:
http://www.vision.caltech.edu/Image_Datasets/Caltech101/101_ObjectCategories.tar.gz

We'll pick the following 3 categories: airplanes, brain, and crab.

Each test run will take 20 random images from each category (60 total) in the training set and another 20 remaining images from each category (60 total) in the testing set.

The results, averaged over 10 runs each:

Algorithm	Train Time	Test Time	F1 Score
SIFT	8.21	7.87	0.75
ORB	1.87	0.91	0.58

These F1 scores are much higher but keep in mind that the images in this dataset do not contain as much background noise as the images in the original dataset.

Let's try adding a 4th category to the above set: pizza (making it airplanes, brain, crab, and pizza):

Algorithm	Train Time	Test Time	F1 Score
SIFT	13.31	11.68	0.62
ORB	2.58	1.31	0.48

It seems that adding a category of images like pizzas which has very dense and varied features greatly reduces the reliability of the algorithm.

Let's try changing pizzas to stop signs which has less features than pizzas (making our categories = airplanes, brain, crab, and stop_sign):

Algorithm	Train Time	Test Time	F1 Score
SIFT	11.17	9.79	0.75
ORB	2.31	1.26	0.57

Using the avg_features() function in display_features.py, we'll calculate the avg # of features per image in every category we've explored so far:

Category	Avg SIFT Features
Apple	601
Bird	1099
Cat	1396
Airplanes	286
Brain	569
Crab	697
Pizza	892
Stop Sign	299

It seems that the Pizzas have far more SIFT features than Stop Signs, which may explain the lower F1 scores when categorizing a set of objects that includes Pizzas.

Although the F1 Scores still decreased, adding Stop Signs to our dataset didn't change our results as dramatically as adding the more feature-rich photos of Pizzas. All else being equal, the F1 score of our classifier is expected to decrease with the addition of more classes of objects to our dataset, owing to the fact that a naive classifier that randomly guesses the class can still achieve an F1 score of 1/N, where N is the # of categories. More on this in the next section.

Justification

Before we move on to the statistical analysis, I'd like to point out a drawback in the F1 score metric. Suppose we have 2 classifiers which both achieved an F1 score of 0.5. One of them achieved it on a dataset with only 2 classes. One of them achieved it on a dataset with 100 classes. It's clear that the classifier that achieved an F1 of 0.5 on 100 classes is superior to the one that got the same score on only 2 classes (which can be achieved even by, say, flipping a coin if the distribution is roughly even.)

As we discussed in earlier sections, the benchmark F1 score is 1/N, where N is the # of classes since random guessing would yield a F1 score of about 1/N assuming pictures are distributed roughly equally among all classes. To measure the robustness of the model, we define an "Error" measure as **1 – F1** and take the ratio of the expected "Error" of a naive random classifier vs the "Error" of the actual classifier we trained to measure the robustness of the model:

$$\text{Robustness} = \frac{1-\frac{1}{N}}{1-F1}$$

Group	**# Classes**	**Original F1 (SIFT)**	**Robustness**
Apples, Birds, Cats	3	0.58	2.38
Airplanes, Brains	2	0.92	12.5
Airplanes, Brains, Crabs	3	0.75	4.0
Airplanes, Brains, Crabs, Pizzas	4	0.62	2.63
Airplanes, Brains, Crabs, Stop Signs	4	0.75	4.0
Food Classification Benchmark	20	0.3	1.43

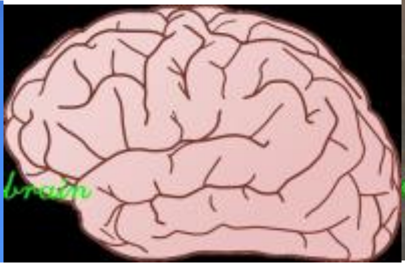
As expected, our classifier is least robust when classifying complex photos of Apples, Birds, and Cats and most robust when classifying simple photos such as airplanes, brains, and crabs. The food classification study also used fairly complex images which may explain the lower robustness.

Does this final solution "solve" the original problem of image classification? With respect to outperforming randomly guessing the classes, most definitely. But are the results good enough for practical use? Probably not, unless we're doing binary classification with just 2 classes (i.e. predicting images you "like" vs. images you don't "like".)

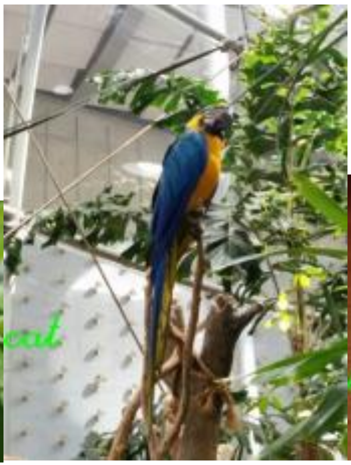
Conclusion

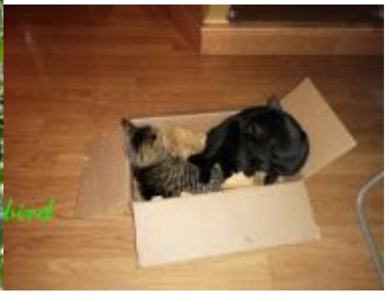
Free Form Visualization

As we discussed previously, images (like the following) without much background noise are relatively easy to identify using the method outlined.



But images that have far more features and background objects are more difficult to identify accurately, as shown below:





It's worth noting that the similar object background can often confuse the classifier such as the image of apples in a tree below to the left and images of a bird in a tree to the right, both of which are in the dataset:

While humans have years of experience backing their ability to identify objects like apples and birds independent of the setting where they're found (i.e. on a tree), computers do not have that capability (until we train them sufficiently via supervised learning.) In other words, a computer "sees" the entire image on the left including the green leaves and the tree branches in addition to the round red objects and is taught that it is an "apple", not the round objects by themselves. And when it tries to decipher the image to the right, it sees the green leaves in the background taking up a significant portion of the image and will likely be classified similarly as the image to the left, ignoring the bird taking up just a small fraction of the image in the foreground.

Finally, here's a sample Confusion Matrix on a random Training-Testing split using our original data. The corresponding F1 score is 0.57:

	Predicted bird	**Predicted apple**	**Predicted cat**
Actual bird	29	4	17
Actual apple	19	25	6
Actual cat	13	5	32

It seems the classifier incorrectly classified quite a few "apples" as "birds" this time around (as expected by the logic explained previously) but not very much so with respect to the opposite case of misclassifying "birds" as "apples. Here, cats and birds are also mixed up often, possibly due to the fact that they're both animals with similar body parts like eyes, faces, etc.

Reflection

The goal of this project was to build an image classification engine that a general user without knowledge of machine learning, computer vision, or programming can use to teach a computer or mobile device to recognize certain classes of images (i.e. apples, cats, birds, airplanes, etc.) The classifier would be designed to incrementally train and update itself on demand whenever new images along with their correct classifications are presented to it without having to retrain it from scratch every time the end user decides to add a new image to train the classifier on.

At a more technical level, the image classification engine uses the OpenCV library to read images and extract key features from them using algorithms like SIFT, ORB, etc. PCA is then used to reduce the dimensionality of these complex feature vectors while minimizing information loss (so the classifier can be trained faster and without overfitting on the noise.) K-means clustering is then used to group together similar feature vectors. A histogram is then generated which represents the % distribution of different feature vectors among all the clusters found by K-Means. These histograms for every image in the training set are then used to train a Stochastic Gradient Descent Classifier. When trying to predict the class of an unknown image, it'll need to be converted into a histogram just like when training the classifier except now the classifier will then predict the class of the image using its histogram.

Difficulties

There were a couple of difficulties encountered when building this image classifier. First, the problem was defined in very general terms: to train the classifier to identify any type of object. Many unrelated objects have similar features and/or exist in similar settings which end up confusing the classifier unless more examples are added to the training set. But large sets also require a ton of computation time to process and train so a tradeoff had to be made.

Second, image complexity was something that was difficult to quantify and control in the project even though we found out the hard way that it played a major role in the success of the classifier. It's not practical to search the web for images based on "complexity". So gathering a suitable dataset ended up posing a greater challenge than expected. Cropping the images in the first dataset was considered but later decided against due to the time involved and the fact that most of the images in the second dataset investigated (from Caltech) were already cropped.

Overall, this project has demonstrated that it is possible to train a computer to recognize images but it still requires more work before it's ready to be deployed in a general setting especially for a non-technical end user.

Improvement

It's been shown that tens, hundreds, or even more training images are required before the classifier is able to classify those images somewhat reliably. If this classifier were to run on someone's mobile device, it would require the end user to snap countless example pictures of a category of object before it's ready to identify it with some accuracy. A potential solution would be to connect the classifier to the APIs of popular image sharing sites like Pinterest, Imgur, Flickr, etc. and have it gather additional example images to train on (with supervision and approval from the end user so that erroneous and misleading images will be excluded from the training queue.) For example, if the end user wanted to add a new category called "computers", he/she may start off by photographing his own computer and sending it to the classifier but the program will then search the various image databases for a dozen or more pictures of computers and ask the user to select the ones it should use to learn this new class.

Another direction for improvement would be to swap the Sklearn classifiers for deep learning neural networks instead (such as those in Tensorflow or Keras.) These neural networks can be trained using the histogram data collected after running K-means or directly on the SIFT/ORB features or even the raw image data itself. If your computer has a decent GPU, these deep learning algorithms will also likely be much faster since Sklearn doesn't support GPUs (at least right now.)

Finally, another area for improvement may be to branch off and create specialized classifiers for identifying very specific types of objects. Instead of detecting "cats" as a possible object, we could design a classifier to identify specific breeds of cats. Or instead of detecting "cars" as one class of object, we can build a classifier that'll tell you the make/model/year of a car by its image. One major advantage of going this route is that the program can be packaged with a classifier that's already pre-trained on some relevant images, unlike a general classifier which is expected to identify every possible image category under the sun.