CIFAR-10 Image Classification: Final Report

Shantanu Chhabra
Carnegie Mellon University
schhabra@andrew.cmu.edu

Srishti Srivastava
Carnegie Mellon University
srishtis@andrew.cmu.edu

1 Introduction

This paper consists of a description of the final project for the Fall 2015 instance of 10-601 : Introduction to Machine Learning, at Carnegie Mellon University. This project focuses on construction of a machine-learning training and classifying algorithm to detect features of an image and label the image with what it depicts. The team used the CIFAR-10 dataset and developed classifiers to classify a given subset of images as one of the ten classes - airplane, automobile, bird, cat, deer, dog, frog, horse, ship, or truck.

The team is pleased to share its Git repository for the project. All code in the repository can be accessed at `https://bitbucket.org/shantanuchhabra/10-601-final-project-cifar-10-image-classification`.

1.1 Motivation

Machine learning, the sub-discipline of computer science predicated on recognizing patterns and constructing predictive algorithms, often on large datasets, is quickly becoming one of the most powerful and valuable applications of computer science. The purpose of this project is to explore and optimize machine-learning techniques. In order to achieve this goal, an image-classifying algorithm that takes an image from a subset of the CIFAR-10 database and correctly identifies the object or animal depicted in it will be developed and optimized for accuracy and efficiency. This project will therefore allow the students to gain a greater practical and conceptual understanding of machine learning while working on a real-world image-processing challenge.

1.2 Background and Related Work

1.2.1 Dataset

The training algorithm for this project uses a subset of the CIFAR-10 dataset (`https://www.cs.toronto.edu/~kriz/cifar.html`). This dataset is a collection of 32x32 labeled color images that fall into one of ten categories (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, or truck). For this project, the algorithm was trained using 5,000 of these images.

1.2.2 Feature Extraction

Feature extraction was performed with the help of the open-source VLFeat package for MATLAB (`http://www.vlfeat.org/`). VLFeat consists of computer-vision algorithms used in image processing, specifically for feature extraction and pattern matching used in machine learning. The feature extraction algorithm used in this project was the histogram of oriented gradients method (HoG). HoG was chosen for its simplicity and proficiency in object and edge detection. The theory behind HoG is to split the image into small square cells, compute the histogram of oriented gradients in each cell which appear visually to ”converge” along edges, and returns a normalized descriptor of each cell, which is a real-valued number. These descriptors provide a set of ”features” for the image, which can then be used as an input to a standard classifying algorithm.

1.2.3 Training and Local Accuracy Checking

The classifier was trained using the entire CIFAR-10 subset provided (all 5,000 images), and tested locally on a randomly chosen 1,000-image subset. In order to check accuracy, the labels outputted by the classifier were compared against the labels provided by the dataset, and the number of matches were returned as a percentage out of 1,000.

2 Method

Before any classification algorithm could be utilized, features needed to be extracted from the given image. This project uses the HoG algorithm given in the VLFeat package to reduce the image to a matrix of normalized features whose size depends on the HoG cell-size used.

2.1 Classifier 1: Gaussian Naive Bayes Classifier

The first classifier used was a simple Naive Bayes Classifier. The premise of any Naive Bayes Classifier is to choose the label k with the maximum probability that the label of the testing example Y = k, and that

$$P(Y = y_k|X_1, ..., X_n) = \frac{P(Y = y_k)\prod_i P(X_i|Y = y_k)}{\sum_j P(Y = y_j)\prod_i P(X_i|Y = y_j)} \tag{1}$$

However, since the classifier considers images, the X_i are real-valued and their possible values are practically infinite. Therefore, we may assume that

$$P(X_i|Y = y_k) \tag{2}$$

follows the Gaussian distribution, and

$$P(X_i = x|Y = y_k) = \frac{1}{\sqrt{2\pi\sigma^2}_{ik}} e^{-\frac{1}{2}(\frac{x-\mu_{ik}}{\sigma_{ik}})^2} \tag{3}$$

where μ_{ik} and σ_{ik} (and subsequently X_i) are updated using a maximum likelihood estimate during training. Finally, in the classifying step, a label is determined for the given text example by choosing the label with the maximum probability of

$$P(Y = y_k)\prod_i P(X_i|Y = y_k) \tag{4}$$

This label is then assigned to the image.

2.2 Classifier 2: K-Nearest Neighbors Classifier

The second classifer used was a k-nearest neighbors algorithm, which represents the set of images as points on an n-dimensional hyperplane, then returns the k nearest neighbors by some metric of distance to the test example and assigns the test example a label based on the labels of its neighbors. For this project, the team used a k-value of 1, and used the Euclidean metric, as described below, to calculate distances between images and points on the hyperplane.

Euclidean distance in an n-dimensional plane:

$$d(p, q) = d(q, p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2} \tag{5}$$

The team's initial attempt at implementing this algorithm consisted of measuring the Euclidean distance between the test example and each of the 5000 training examples, and returning the label of the 1 training example closest to the testing example. While effective, however, this algorithm was extremely slow.

The team then implemented an algorithm that trains by calculating the distance of each training example from the origin and sorting them into an ordered list. Then, the distance from the origin of the test example was computed. The ordered list of training examples was searched using an efficient binary search to return the closest training example whose distance from the origin was less than or equal to the distance from the origin of the training example. To obtain the other points on the same hypersphere of distance from the origin, the nearest λ training examples in the ordered list both less than and greater than the selected training example were considered. Then, the absolute distance between the testing example and each of the selected training examples was computed. The label of the training example with the least Euclidean distance from the testing example was returned.

The majority of the distance calculations and the ordering of the training example list was performed in the training stage, and a much more targeted version of the initial kNN algorithm, limited to a smaller set of computations, was performed. Therefore, the algorithm was made much more efficient without significantly sacrificing accuracy.

2.3 Classifier 3: K-means Classifier

The basic premise of a k-means classifier is to cluster the members of each class together in n-dimensional space, and update the centroid of each cluster until all of the clusters have converged. In order to implement this algorithm as part of this project, the team trained the classifier by initializing the clusters based on the training data. The ith coordinate of the centroid of each cluster was calculated by the following equation:

$$\mu_i = \frac{\sum_{k=1}^{n} x_k}{n} \tag{6}$$

where n is the number of points in the cluster. Then, the testing examples were added into the representation of n-dimensional space. The distance from the center of each cluster to the testing example was calculated, and the example was assigned to the nearest cluster. The centroids were recalculated, and each of the testing examples were assigned again.

In theory, this algorithm would be repeated until none of the testing examples changed their membership in a cluster. However, in implementing this algorithm, the team had to be mindful of the efficiency of the code, as relying on convergence of the algorithm in every case might take too long. Therefore, the team decided to implement a cap on the number of iterations of the algorithm. At the end of the iterations, the testing examples were assigned the label of the cluster that they were in.

3 Experiments and Results

3.1 Classifier 1: Gaussian Naive Bayes Classifier

Experiment 1: Out of curiosity and to have a benchmark, the team experimented with the Naive Bayes Classifier built into MATLAB, using the same training data and running tests using the same testing data for both the built-in classifier and the implemented classifier.
First of all, the team extracted the features using `extract_feature` as provided to us in `extract_feature.m`.
Next, they used `fitNaiveBayes` to train their classifier. `fitNaiveBayes` is a built-in function in `NaiveBayes` class in MATLAB.
`fitNaiveBayes(features, Y)` returns a Naive Bayes model with 10 classes with a normal feature distribution.
Finally, in `classify.m`, we call Naive Bayes' built-in function, `predict` on the Naive Bayes model we found in `train.m` and the testing data to get the prediction.

Result: Interestingly, the team found that their classifier had equivalent or superior accuracy to the built-in classifier. However, their classifier was shown to be slower than the built-in Naive Bayes implementation. The team noted its observations in Table 1

Experiment 2: This experiment was performed with the goal of figuring out the HoG cell-size that would most improve the classifier. The team approached the problem by changing the values of cell-size and documenting how accurate the predictions were. The cell-sizes that the team tested were 8, 12, 14, 16, and 32. The team trained their classifier using the 5000 images in `small_data_batch_N` where $1 \leq N \leq 5$ and used `small_data_batch_N` as 5 different testing datasets. They tested percentage accuracy by running `sum(predicted_labels) == provided_labels`.

Table 1: HoG Average %age Classified by Cell Size

CELL SIZE	Built-in GNB	Self-Implemented GNB
8	49.9	50.0
12	45.8	46.1
14	40.9	41.1
16	41.0	41.1
32	32.6	33.0

Table 2: HoG Cell Size vs Classifier Accuracy

CELL SIZE	AVERAGE % CLASSIFIED
8	50.0
12	46.1
14	41.1
16	41.1
32	33.0

Result: The results of varying the cell-size concurred with the teams' intuition. Larger cell sizes although fast, reduced the accuracy of the classifier. For example, a cell-size of 32 took about a tenth of the time that a cell-size of 8 took, but suffered a 36% loss in accuracy (Accuracy with HoG cell-size = 8 was around 50% whereas that with cell-size = 32 was about 32%). After carefully considering the merits and drawbacks of all possible cell-sizes, the team decided that the most optimal cell-size for the purpose of the classifier would be 8. The team documented its observations in Table 2.

3.2 Classifier 2: K-Nearest Neighbor Classifier

Experiment 1: This experiment was performed to determine the best implementation of kNN in terms of accuracy and efficiency. Three methods were tested.

1. The naive implementation of kNN, in which the testing example is compared against every training example to find its nearest neighbor. This algorithm would run in $O(n)$ time.
2. An implementation that sorts the training examples by distance from the origin and then chooses the neighbor with the closest distance from the origin to the testing example's distance from the origin. The sorting would only need to be done once during training, and using an efficient binary search would reduce the time complexity from $O(n)$ in the size of the training examples to $O(\log(n))$ for a binary search.
3. An implementation similar to the second that instead of searching for only one training example with a comparable distance from the origin, searches for a pre-defined number of them, and then compares each one directly to the testing example. This algorithm runs in $O(\log(n))$ for binary search + $O(m)$ for m direct comparisons, where $m < n$.

Result: All three implementations were found to have a sufficiently efficient runtime. However, their accuracy varied, as can be seen in Table 3. Empirically, the team discovered the following about each algorithm:

1. The naive algorithm was very slow and scaled poorly as the number of testing examples increased. However, it was also relatively accurate because the nearest neighbor would inevitably be found.
2. This implementation was extremely efficient. However, it was also extremely inaccurate. The team determined this to be because, while the testing example and the selected training example were guaranteed to lie on (or near) the same hypersphere, it was possible that another point/training example on the same hypersphere was actually directly closer to the

Table 3: kNN Implementation vs Accuracy

METHOD	AVERAGE % CLASSIFIED
1	44.6
2	10.8
3	46.7

Table 4: λ vs Accuracy

λ VALUE	AVERAGE % CLASSIFIED
25	21.3
250	28.5
2500	46.7

testing example. This could not be accounted for by a list sorted by distance from the origin. The team was therefore inspired to try the third method, a hybrid of the first two.

3. The final method isolated the correct hypersphere as described in method 2, but then attempted to isolate a pre-defined number of other points on the same hypersphere and compare the training example directly to each of them in order to find the nearest neighbor. This algorithm achieved a good balance of both accuracy and speed.

Experiment 2: The team then proceeded to test the value of λ, or the number of additional points on the hypersphere to be compared against the testing example. Three possible λ values were chosen, and were tested for their accuracy, with the expectation that a higher λ would result in more accurate but slower results.

Result: The results of the accuracy of different λ values can be seen in Table 4. Since the team found the greatest lambda to still be sufficiently efficient, they decided to use 2500 in order to achieve the greatest possible accuracy.

Experiment 3: The team then decided to test out different metrics to determine the distances between the points/images, and their effect on the accuracy of the classifier. The metrics tested were the usual Euclidean, the Manhattan distance, the cosine distance, and two additional metrics designed by the team. In order to ensure an unbiased experiment, the classifer was tested on data included in the training data, implying that the classifer should work with perfect accuracy. All metrics are described below:

Euclidean distance in an n-dimensional plane:

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{7}$$

Manhattan distance in an n-dimensional plane:

$$d(p,q) = \sum_{i=1}^{n} |q_i - p_i| \tag{8}$$

Cosine distance:

$$d(p,q) = \frac{\sum_{i=1}^{n} (p_i * q_i)}{\sqrt{\sum_{i=1}^{n} p_i^2}\sqrt{\sum_{i=1}^{n} q_i^2}} \tag{9}$$

Team-developed metric 1:

$$d(p,q) = \sum_{i=1}^{n} (q_i - p_i)^3 \tag{10}$$

Table 5: Effect of Metric on Accuracy

METRIC	AVERAGE % CLASSIFIED
Euclidean	100.0
Manhattan	66.2
Cosine	5.9
Team 1	75.4
Team 2	71.2

Table 6: K-Means: Effect of Number of Iterations on Accuracy

ITERATIONS	PERCENTAGE CORRECTNESS
1	45.7
10	45.4
20	44.5
100	42.4
200	39.9
500	36.2
1000	33.3

Team-developed metric 2:

$$d(p,q) = \sum_{i=1}^{n} (q_i - p_i)^2 \tag{11}$$

Result: The results obtained can be seen in Table 5. The team discovered that only the Euclidean metric preserved 100% accuracy, with all others performing below that. The team was interested to see that cosine distance had a particularly poor performance, possibly suggesting that the distance values were relatively small, and normalization of those values to the interval [0,1] resulted in information loss. The team finally chose to use the Euclidean distance in the kNN algorithm.

3.3 Classifier 3: K-means Classifier

Experiment: The major experiment in the k-means classifier was to vary the number of iterations that the algorithm made before the labels were assigned. The team had to be careful to prevent the algorithm from exceeding the allotted time, while still ensuring enough iterations to generate meaningful results. The results of the varying iteration caps are show in Table 6.

Result: The team was surprised to discover that the most optimal number of iterations was 1, as the nature of the algorithm would seem to suggest that a higher number of iterations would improve accuracy. However, since 1 appeared to result in both the highest accuracy and efficiency, the team decided to use it in their final implementation.

3.4 Miscellaneous Experiments

The team also experimented with a few other classifiers, as well as alternative methods of feature extraction. Some of the classifying algorithms attempted included logistic regression and Gaussian mixture models. After implementing both algorithms, the team discovered the following.

Experiment 1: The team attempted to use logistic regression for classification, modifying an existing binary logistic regression classifier (used on Homework 1 for the class) to a one-against-many approach. This approach essentially classified a testing example as in class 0 or not, then tested any examples that weren't assigned to class 0 in the same way against class 1, and so on.

Results: The team discovered that a simple logistic regression was not adept at handling so many classes, and had a very poor accuracy. While modifications could theoretically be made to logistic regression for a categorical Y variable, the team felt that their time was better spent on a different approach.

Experiment 2: The team was initially interested in Gaussian mixture models due to their efficient handling of approximately normal, real-valued data, and attempted to implement the GMM model based on the implementation from the homework.

Results: Similar to the results of the logistic regression, the team found it difficult to handle a Y variable that was categorical. Furthermore, while Gaussian mixture models were easy to train, the team's attempts to modify the Naive Bayes algorithm to classify the GMM results were not very effective. Despite the promise of an accurate model, the team decided to instead pursue the k-nearest neighbors classifier for its relative simplicity.

The team also experimented with an alternative algorithm for feature extraction.

Experiment 3: The team experimented with the Dense Scale Invariant Feature Transform (DSIFT) package found in VLFeat to extract features instead of HoG. DSIFT provides similar advantages to HoG, but is a different descriptor, and the team felt that its object-recognition capabilities could potentially provide a better fit for the dataset.

Results: After implementing a rudimentary DSIFT algorithm, the team was disappointed to discover that the implementation did not provide a better input to the classification algorithms, resulting in either equal or lower accuracy. Therefore, the team decided to abandon their attempt to use DSIFT in favor of spending more time improving their classifiers.

4 Conclusion

At the completion of the project, the team had achieved their greatest success with their Naive Bayes Classifier, which provided 48.6% accuracy on the testing data set. In addition, the team's other classifiers, the K-nearest neighbor and K-means, reached 46.7% and 45.1%, respectively. After working hard on optimizing and limiting the error in their models and classifiers, the team felt that their classifiers were able to meaningfully address the task, despite the challenges of trading off accuracy and efficiency. Alongside the three submitted classifiers, the team also learned a lot from the project through exploring a wide variety of commonly-used Machine Learning algorithms such as logistic regression and Gaussian mixture models, and working with feature-extraction algorithms like Histogram of Oriented Gradients (HoG) and Dense Scale Invariant Feature Transform (DSIFT). The team also learned how to address the real-world challenges of inconsistent runtimes and accuracy on different machines.

In regards to improving their existing submission, the team feels as though their k-means submission could have been made more efficient, and would have liked to integrate the Fisher Method into their Naive Bayes classifier and a k-D tree into the k-nearest neighbor algorithm as a means to improve accuracy. They would also have liked to pay greater attention to performance to address issues such as bias and overfitting in their models. If asked to do the project again, the team would have liked to explore more complex algorithms like neural nets, and composite algorithms such as a Gaussian mixture model with a support vector machine. They would also pay greater attention to code design, limiting the amount of needed optimization and improvement.

Despite the limitations of their solution, the team felt that the project was a valuable addition to the class, and that they were successful in applying the theoretical concepts learned in the class to a real-world situation. From the project, the team felt that they had gained far greater insight into the field of Machine Learning and its practical and theoretical challenges, and were excited to see that their algorithms could successfully classify the images from the CIFAR-10 dataset.

References

[1] Forsyth, D.A. *Edges, Orientation, HOG and SIFT*. UIUC, 2012.

[2] Mitchell, T.M. ”Generitive and Discriminative Classifiers: Naive Bayes and Logistic Regression.” *Machine Learning.* McGraw Hill, 1997.

[3] Bishop, Christopher M. *Pattern Recognition and Machine Learning.* New York: Springer, 2006.

[4] Nguyen, Thinh. Lecture 13, Oregon State University. *Nearest Neighbor Search*

[5] Soder, Ola. University of Virginia. *Improving the classification accuracy of kNN classifiers*

[6] MathWorks Documentation for MATLAB

[7] GNU Octave Documentation

[8] VLFeat documentation for Dense SIFT

[9] VLFeat documentation for Histogram of Oriented Gradients