University of Information Technology

Computer Vision project

Objects Classification on Caltech 101

Truong Phuc Anh

Lam Han Vuong

School of Electrical Engineering

Supervised by

Ngo Duc Thanh Dr

1 January 2018

Contents

List of Figures

List of Tables

1 Introduction

Object recognition is a common task of computer vision for identifying a specific object in a digital image or video. Humans can recognize multiple objects in images with little effort, but this task is still a big challenge for computer vision systems due to the fact that the objects may vary somewhat in different view points, in many different sizes, scales, types or even when they are translated or rotated (in video).

Figure 1: Chairs with different view points, sizes, materials make the recognition task more challenging.

Object recognition is useful in applications such as video stabilization [1], Manufacturing Quality Control [2], Automated vehicle parking systems [3], etc.

Many approaches to the task have been implemented over the years. Common techniques include deep learning based approaches such as convolutional neural networks, and feature-based approaches using edges, gradients, histogram of oriented gradients (HOG), Haar wavelets, and linear binary patterns.

In this project, we end up with a simple feature-based approache using Bag of words (BOW) with two different learning algorithms K-Nearest Neighbor (KNN)

and Support Vector Machine (SVM) on Caltech101 just for practising what we have learn from the class.

2 Dataset

We use Caltech 101 for this project. Pictures of objects belonging to 101 categories. About 40 to 800 images per category. Most categories have about 50 images. Collected in September 2003 by Fei-Fei Li, Marco Andreetto, and Marc 'Aurelio Ranzato. The size of each image is roughly 300 x 200 pixels.

For more infomation about the dataset, please visit *http://www.vision.caltech.edu/Image_Datasets/Caltech101/Caltech101.html*.

3 Methods

3.1 Flow

We used the generic flow for this recognition task with *Bag of visual words* as feature extractor and *KNN, SVM* as learning algorithms.

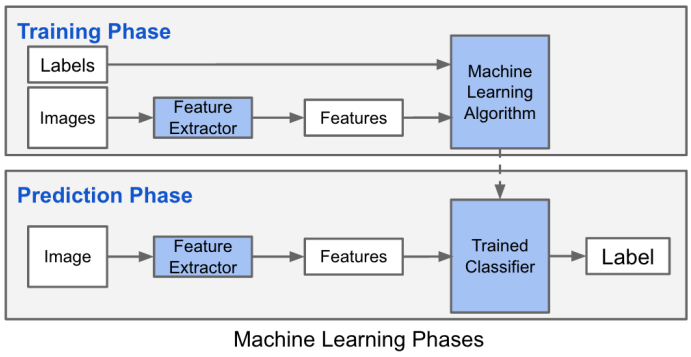

Figure 2: Generic flow for object recognition

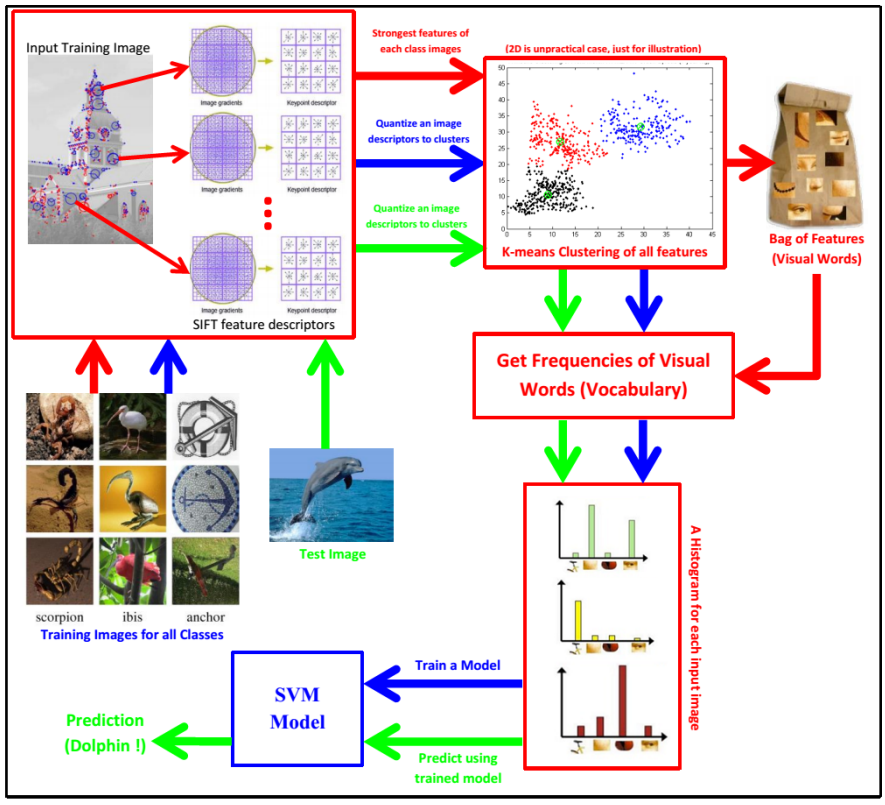

Figure 3: The detail process of recognition with BOW and SVM

3.2 Bag Of Visual Words (BOV)

Over view

The bag-of-words model is a vector space model representing text data when modeling text with machine learning algorithms. In practice, this model is mainly used as a tool of feature generation.

Basic idea

In doucment classification after transforming the text into a "bag of words", we can calculate various measures (also call features) to characterize the text. The most common type of characteristic is the histogram of words (based on the number of times a word appears in the document).

In oder to use bag of words (from text) in computer vision (for images), we can treat image features as words. Thus, we have *Bag Of Visual Words*. The vocabulary can be built up by clustering all feature points collected from a suitable source (Ex. the training set) with each cluster represents for a word. After we have the vocabulary, the histogram of words will be used as feature vector of an image.

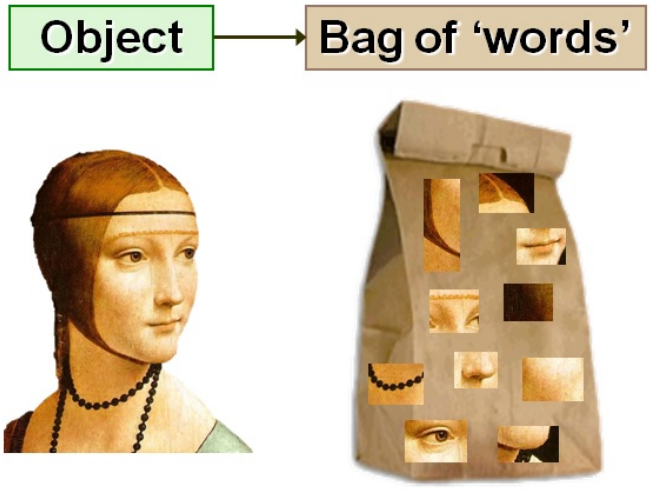

Figure 4: An example for BOV

In this project, we use SIFT as feature extractor and K-Means as clustering algorithm to build the vocabulary.

Pseudo-code

```
# Create a vocabulary from a set of images
def create_vocabulary(images):
    # 1. Get key points (interested points) from all images
    key_points = get_key_points(images)

    # 2. Calculate descriptors for those key points.
    descriptors = get_descriptors(images, key_points)

```

```
    # 3. Run K-Means clustering on those descriptor
    cluster_centers = run_kmeans_on(descriptors)

    # 4. Each cluster center is a word in vocabulary
    return cluster_centers as vocabulary
```

```
# Calculate histogram of visual words (as a feature vector) for an image
def cal_histogram(image, vocabulary):
    # 1. Create an empty histogram
    n_words = len(vocabulary)
    histogram = np.zero(n_words) # zeros array with length = n_words

    # 2. Calculate all descriptors in this image
    key_points = get_key_points(image)
    descriptors = get_descriptors(key_points)

    # 3. For each descriptor, find out which word it is (in vocabulary).
    for descriptor in descriptors:
        word_index = which_word(descriptor, vocabulary)
        histogram[word_index]++ # increase histogram bin

    return histogram
```

```
# Find out which word an descriptor is (in vocabulary)
def which_word(descriptor, vocabulary):
    n_words = len(vocabulary)
    dist = np.zero(n_words) # zeros array with length = n_words
    for i in range(n_words)
        dist[i] = L1_distance(vocabulary[i], descriptor)
    return min_index_of(dist)
```

3.3 K-Neareast Neighbor (KNN)

Over view

KNN is a type of instance-based learning, or lazy learning which means we don't need to train a real model for redicting.

Basic idea

The basic idea is with any unknown sample x (or a query point), finding k *nearest samples* in training set and assigning the label which is most frequent among *k-nearest samples* to that query point.

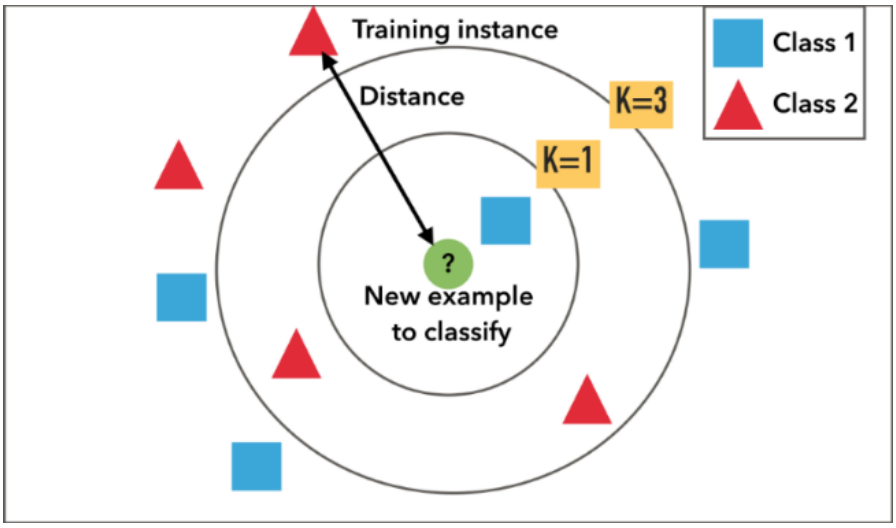

Figure 5: An example of KNN with different K

A common method used for calculate the distance metric between samples is Euclidean distance. We also use this model in this project for a simple implementation.

Pseudo-code

```
# Find the label of an unknown sample x
# x: unknown sample
# X: training data
# Y: labels of training data
# k: parameter k
def knn(x, X, Y, k):
    n_samples = length(X)

    # 1. Calculate the distance between x and all training samples in X
    for i in range(0, n_samples):
        distances[i] = Euclidean_distance(x, X[i])

    # 2. Get index of k-th minimum distances
    k_indeces = get_minimum_indeces(k, distances)

    # 3. Get voted labels from k_indeces
    voted_labels = labels[k_indeces]

    # 4. return the most frequent among voted_labels as the label of x
    return most_frequent(voted_labels)
```

3.4 Support Vector Machines (SVMs)

Over view

SVMs are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. [4]. In this project we only consider about classification.

In 1957, a simple linear model called the Perceptron was invented by Frank Rosenblatt to do classification.

A few years later, Vapnik and Chervonenkis, proposed another model called the "Maximal Margin Classifier", the SVM was born.

Then, in 1992, Vapnik et al. had the idea to apply what is called the Kernel Trick, which allow to use the SVM to classify linearly nonseparable data.

Eventually, in 1995, Cortes and Vapnik introduced the Soft Margin Classifier which allows us to accept some misclassifications when using a SVM.

So just when we talk about classification there is already four different Support Vector Machines:

1. The original one : the Maximal Margin Classifiers
2. The kernelized version using the Kernel Trick
3. The soft-margin version
4. The soft-margin kernelized version (which combine 1, 2 and 3)

For more detail about the history, you can read here.

In this project, we implement the last version for multi-classification with two approaches *one vs one* and *one vs rest*

Basic idea

In the case of SVM, it learns a linear model (a hyperplane) to seperate data. Let's take a quick look at how the first SVM (also call hard margin) is built up in simple case (linear sepratable data).

Before SVM, Perceptron Learning Algorithm is a simple model for finding a hyperplane to separate data. However, its biggest weakness is that it will not find the same hyperplane every time but not the best one (because of the randomness, you can find the detail here). SVMs solve this problem by finding the *best hyperplane* (we call this the **optimal hyperplane**).

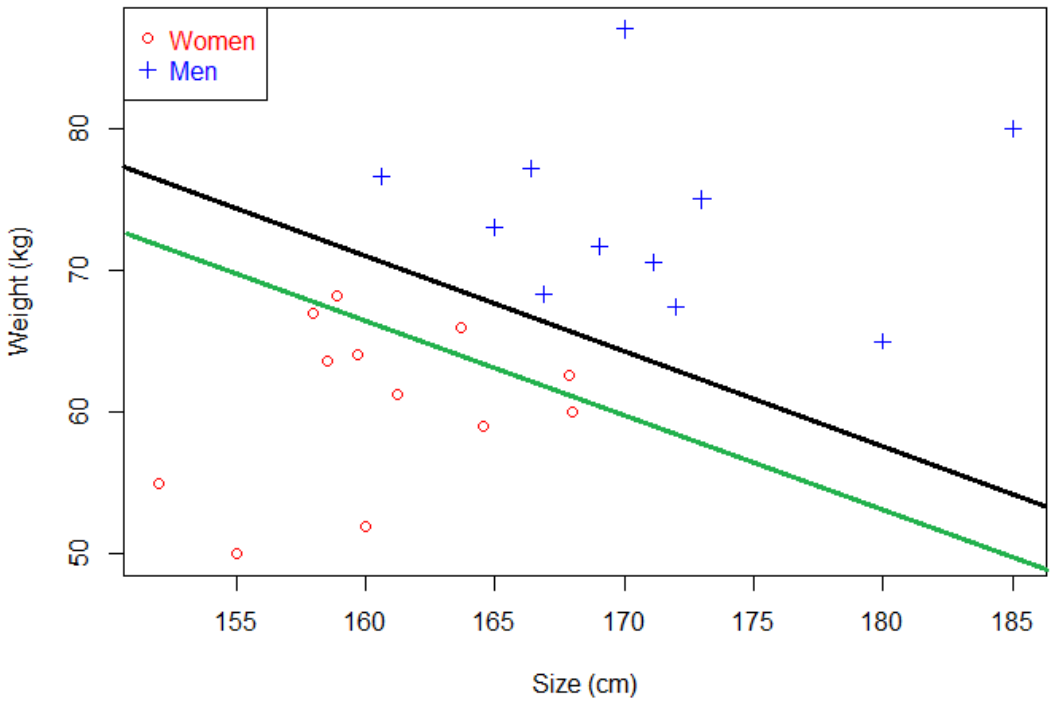

Figure 6: The black hyperplane classifies more accurately than the green one

SVMs compare two hyperplanes (which one is better for separate data) base on **the margin**.Basically the margin is a no man's land. There will never be any data point inside the margin.

We can make the following observations:

- If an hyperplane is very close to a data point, its margin will be small.

- The further an hyperplane is from a data point, the larger its margin will be.

This means that the optimal hyperplane will be the one with the biggest margin.

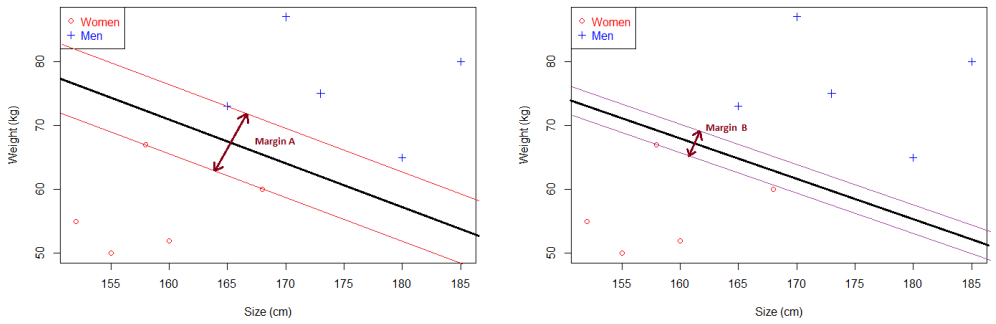

Figure 7: Margin B is smaller than Margin A, so Margin A would be the better one for sperating data.

In practice, we can compute the distance between the hyperplane and the closest data point. Once we have this value, if we double it we will the value of margin. After determine what the margin is and how to calculate it, we will solve the optimal problem (quadratic problem) to find the *support vectors* and compute w, b parameter for the optiaml hyperplan from those *support vector*.

We will not go on the mathematics behind SVMs, but you can read this book.

Pseudo-code

For hard margin classification, all we need to do is let mathematis do the job (solve the quadratic problem). The quadratic problem solver we used is available here scikit-learn libary. Below is an example code for using it.

```
1 import cvxopt.solvers
2 X, y = get_dataset(ls.get_training_examples)
3 m = X.shape[0]
```

```
# Gram matrix - The matrix of all possible inner products of X.
K = np.array([np.dot(X[i], X[j])
for j in range(m)
for i in range(m)]).reshape((m, m))
P = cvxopt.matrix(np.outer(y, y) * K)
q = cvxopt.matrix(-1 * np.ones(m))
# Equality constraints
A = cvxopt.matrix(y, (1, m))
b = cvxopt.matrix(0.0)
# Inequality constraints
G = cvxopt.matrix(np.diag(-1 * np.ones(m)))
h = cvxopt.matrix(np.zeros(m))
# Solve the problem
solution = cvxopt.solvers.qp(P, q, G, h, A, b)
# Lagrange multipliers
multipliers = np.ravel(solution['x'])
# Support vectors have positive multipliers.
has_positive_multiplier = multipliers > 1e-7
sv_multipliers = multipliers[has_positive_multiplier]
support_vectors = X[has_positive_multiplier]
support_vectors_y = y[has_positive_multiplier]
```

For computing w

```
def compute_w(multipliers, X, y):
    return np.sum(multipliers[i] * y[i] * X[i]
        for i in range(len(y)))
```

For computing b

```
def compute_b(w, X, y):
    return np.sum([y[i] - np.dot(w, X[i])
        for i in range(len(X))])/len(X)
```

4 Experiments

4.1 Data preprocessing

We randomly split data into train (80) and test (20) subsets using sklearn.model_ selection.train_ test_ split. The ground-truth labels of data will be get from the folder from original dataset and then it will be stored in **features/y_ train.pkl** and **features/y_ test.pkl** for using in python code.

4.2 Feature extraction

The BOV vocabulary is built up by using all images in dataset. We get about 40 - 50 key points and its descriptors in every image, then run K-Means with **k** = _ on this key points space to get the vocabulary with _ visual words. The vocabulary is stored in **bin/vocabulary.pkl**

After having the vocabulary, we calculate the histogram of visual words for all images in both training set and test set. The result (feature vectors of training and test set) is stored in **features/X_ train.pkl** and **features/X_ test.pkl**

4.3 KNN test

We simply run KNN on test set with diffrent k (from 1 to 100) and end up with result below.

4.4 SVMs test

We try to train SVM model with different parameter values, different kernels and aproachs for multi-classification. All models are saved in **bin/svm-models** with format "**multi-class_ kernel_ c_ degree_ gamma**". The value of parameter

is changing following the conclusion from theory we've learn in the class and our experiences in the testing process.

Here is some results (for more detail, you can read the log files from **log/svm**).

method	C	$training - time$	$training - accuracy$	$test - accuracy$
ovo	0.001	39.568	0.126	0.044
	0.01	36.864	0.326	0.151
	0.1	41.069	0.998	0.142
	0.2	44.101	1	0.142
	0.3	41.413	1	0.142
	100	40.766	1	0.142
	1000	44.251	1	0.142
ovr	0.001	2.713	0.406	0.060
	0.01	3.411	0.840	0.101
	0.1	5.355	0.977	0.104
	0.2	5.956	0.986	0.085
	0.3	6.241	0.988	0.079
	100	7.320	0.975	0.038
	1000	7.654	0.974	0.041

Table 1: SVM result with linear kernel.

method	*C*	*degree*	$training - time$	$training - accuracy$	$test - accuracy$
ovo	0.001	1	39.568	0.126	0.044
	0.01	2	36.864	0.326	0.151
	0.1	3	41.069	0.998	0.142
	0.2	4	44.101	1	0.142
	0.3	5	41.413	1	0.142
	100	6	40.766	1	0.142
	1000	7	44.251	1	0.142
ovr	0.001	8	2.713	0.406	0.060
	0.01	9	3.411	0.840	0.101
	0.1	10	5.355	0.977	0.104
	0.2	11	5.956	0.986	0.085
	0.3	12	6.241	0.988	0.079
	100	13	7.320	0.975	0.038
	1000	14	7.654	0.974	0.041

Table 2: SVM result with poly kernel.

method	*C*	*gamma*	$training - time$	$training - accuracy$	$test - accuracy$
ovo	0.001	1	39.568	0.126	0.044
	0.01	2	36.864	0.326	0.151
	0.1	3	41.069	0.998	0.142
	0.2	4	44.101	1	0.142
	0.3	5	41.413	1	0.142
	100	6	40.766	1	0.142
	1000	7	44.251	1	0.142
ovr	0.001	8	2.713	0.406	0.060
	0.01	9	3.411	0.840	0.101
	0.1	10	5.355	0.977	0.104
	0.2	11	5.956	0.986	0.085
	0.3	12	6.241	0.988	0.079
	100	13	7.320	0.975	0.038
	1000	14	7.654	0.974	0.041

Table 3: SVM result with rbf kernel.

5 Conclusions

6 References

[1] F. Liu, M. Gleicher, H. Jin and A. Agarwala, 'Content-preserving warps for 3d video stabilization', *ACM Transactions on Graphics*, 2009.

[2] C. Demant, B. Streicher-Abel and P. Waszkewitz, *Industrial image processing: Visual quality control in manufacturing*. Springer, 2013.

[3] H. G. Jung, D. S. Kim, P. J. Yoon and J. Kim, *Structural, Syntactic and Statistical Pattern Recognition*. Springer Berlin / Heidelberg, 2006.

[4] Wikipedia, *Support vector machine*, 2004. [Online]. Available: https://en.wikipedia.org/wiki/Support_vector_machine.

7 Appendices