Visão por Computador - Reconhecimento de Objetos

Leonel Peixoto - 201204919
ei12178@fe.up.pt
Rodolfo Rodrigues - 200200428
ei12151@fe.up.pt

Faculdade de Engenharia
Universidade do Porto
Porto, Portugal

Resumo

Trabalhos recentes na área da visão por computador permitiram o desenvolvimento de novos métodos como redes neuronais pondo os métodos clássicos em segundo plano como as *Support Vector Machines* (SVN).

O objetivo principal deste projeto consiste em desenvolver um sistema que pode reconhecer e classificar um objeto presente numa imagem como sendo um objeto que pertence a uma de 10 classes diferentes. Para tal, usou-se e comparou-se dois métodos diferentes para reconhecer os objetos nas imagens. Utilizou-se uma rede neuronal convolucional e SVN usando *Bag of Words* (BoF) constituídos por descritores obtidos através do algoritmo SIFT.

Os dados processados foram obtidos do dataset do CIFAR-10 do Kaggle. Chegou-se a um resultado de 78.7% para a rede neural e 24.5% para a melhor variante do método SVM.

1 Introdução

Um dos principais propósitos da Visão por Computador é permitir que uma máquina seja capaz de compreender e obter informação útil de imagens e vídeo, estabelecendo algoritmos para o reconhecimento e classificação de objetos, pessoas, etc. A utilização de abordagens baseadas em *deep learning* é a mais recente tendência e por isso é interessante realizar um comparativo com a abordagem mais tradicional de *Support Vector Machines* (SVM).

O presente trabalho realiza um estudo comparativo entre uma *Convolution Neural Network* (CNN) e um classificador SVM associado a um extrator de features utilizando o algoritmo SIFT e um *clustering* em *Bag of Visual Words* (BoW), são avaliadas três variantes do classificador SVM: *Multiclass Ranking* SVM, *One Against All* SVM (OAA) e OAA SVM *Balanced*.

O *dataset* utilizando neste trabalho pertence à competição CIFAR-10 do Kaggle e é composto por 60.000 imagens (50.000 imagens de treino e 10.000 imagens de teste) distribuídas por 10 categorias com 6000 imagens por categoria.

2 Sistema Desenvolvido

Neste trabalho foram desenvolvidas duas abordagens distintas para o sistema de reconhecimento de objetos:

2.1 Rede Neuronal Convoluncional

O sistema de reconhecimento de objetos usando *deep learning* foi realizado com a construção de uma rede neuronal convolucional usando as bibliotecas Keras e Theano em Python.

A rede neuronal foi sujeita a uma fase de treino com o *dataset* de treino do CIFAR-10 para otimizar os pesos de ligação entre cada *perceptron* e uma fase de avaliação com o *dataset* de teste do CIFAR-10.

2.1.1 Arquitetura da rede

A rede neuronal é construída por várias camadas sucessivas. As quatro primeiras camadas são do tipo convolucionais seguida por duas camadas completamente conectadas. As camadas completamente conectadas também estão completamente conectadas as camadas anteriores. Desta forma, o conjunto permite minimizar a regressão logística da função objetivo. A saída de cada camada é aplicada uma função ReLU (*rectified linear unit*) para aumentar a velocidade de aprendizagem. Demais, as funções ReLU permitem uma aprendizagem mesmo quando os valores de entrada não foram redimensionados.

É importante salientar que a rede de aprendizagem não requer a extração de características (*features*) específicas. Em vez disso, pode-se estabelecer a analogia entre as camadas iniciais convolucionais da rede e o processo de extração e representação de características. O fato de este processo ser automatizado e otimizado como parte do modelo global é um ponto muito atraente para a abordagem de *deep learning*.

As imagens possuem um tamanho de 32 por 32 pixéis com 3 canais de cor. A primeira e segunda camada filtram as imagens com 32 *kernels* de tamanho 3x3 x 2. A quarta e quinta camada são constituídas por 64 *kernels* 3x3. A saída da segunda e quinta camada são normalizadas com uma função de agrupamento de máximos (*max pooling*).

2.1.2 Prevenção de *overfitting*

A rede neuronal criada é bastante grande, pois, o número total de parâmetros em todas as camadas é de vários milhões. Para o conjunto de treino de entrada dado, a rede foi treinada com 50 iterações. Há, portanto, um alto risco de *overfitting*. Para prevenir tal acontecimento, o algoritmo de treinamento emprega técnicas de "*dropout*", isto é, durante os passos de propagação de erro e atualização de peso, existe uma probabilidade de 0,5 que a saída de um neurônio em uma camada escondida seja definida como 0 (assim, efetivamente, impede-se de contribuir para aprendizagem nesse passo). Isso pode ser visto como treino de múltiplos métodos em paralelo (semelhante à validação cruzada para cada camada, onde um subconjunto dos dados é considerado de cada vez, para emular múltiplos conjuntos de dados com propriedades semelhantes, que de outra forma seriam excessivamente onerosos).

2.2 Support Vector Machines

O sistema de reconhecimento de objetos através da abordagem SVM é composto por três módulos: Extração de *Features*, Representação de Imagem, e Classificação. A deteção e o cálculo de descritores das *features* da imagem foi realizada utilizando o algoritmo SIFT.

A representação da imagem realizou-se através de "*visual words*" obtidas através do *clustering* (k-means) das *features* obtidas no passo anterior, posteriormente as imagens são representadas através de histogramas dessas "*visual words*". Finalmente, os histogramas são utilizados para o treino do classificador SVM. Os três módulos foram desenvolvidos utilizando a versão 2.4 da biblioteca OpenCV.

2.2.1 Extração de *Features*

O módulo de extração de *features* das imagens de teste é realizado utilizando o algoritmo SIFT para a deteção das *features* e cálculo dos descritores das *features*. O produto deste módulo é um conjunto de vetores normalizados que serve de base à definição do vocabulário de "*visual words*".

2.2.2 Representação de Imagem

O módulo anterior gera descritores das *features*das imagens do dataset que serão utilizados no processo de *clustering* do presente módulo, através do algoritmo k-means, os descritores são agrupados em *clusters* que representam uma "*visual word*" e conjunto dos *clusters* é o vocabulário do nosso modelo de *Bag of Words*.

A definição do vocabulário do nosso modelo de *Bag of Words* permite a representação das imagens de treino do *dataset* através de um histograma das "*visual words*", permitindo a aplicação da metodologia SVM a este espaço de *features* de dimensão igual à dimensão do vocabulário do modelo BoW.

Uma das principais variáveis desta abordagem é a dimensão do vocabulário (número de *clusters*) a usar, de forma a encontrar uma dimensão que produzisse resultados aceitáveis foram testados vários valores.

2.2.3 Classificação

A representação em histograma das imagens do *dataset* foram utilizadas no treino do classificador SVM, dado que neste trabalho de reconhecimento de objetos existem 10 classificações distintas foram elaboradas três variantes do classificador SVM: *Multiclass ranking* SVM, *One Against All* SVM (OAA) e OAA SVM *Balanced*.

A variante *Multiclass Ranking* SVM utiliza a função de decisão para tentar classificar todas as 10 classes existentes, a OAA SVM possui um classificador binário SVM para cada classe para separar os membros dessa classe de todas as outras classes. Por fim, como no *dataset* de 60.000 imagens existem 6.000 imagens por classe, um *subset* de treino terá em média apenas 10 % de imagens de cada classe, o que no caso da variante OAA SVM se traduz num set de treino desequilibrado para cada um dos classificadores binários. Assim, elaborou-se uma terceira variante OAA SVM *Balanced* onde se garante que 50 % do *subset* de treino são imagens da classe que o classificador binário pretende identificar.

Na variante *Multiclass Ranking* SVM, o classificador SVM atribui directamente uma classe às imagens que processa, mas no caso das variantes OAA SVM e OAA SVM *Balanced* esta classificação não é direta porque é necessário conjugar os resultados dos vários classificadores binários, bem como, resolver situações em que vários classificadores aceitam ou todos rejeitam uma determinada imagem. Nestas duas variantes, a classificação atribuída à imagem coincide com a classe do classificador binário com o maior valor da função de decisão (maior distância à fronteira do classificador SVM).

3 Resultados

3.1 Rede Neuronal Convoluncional

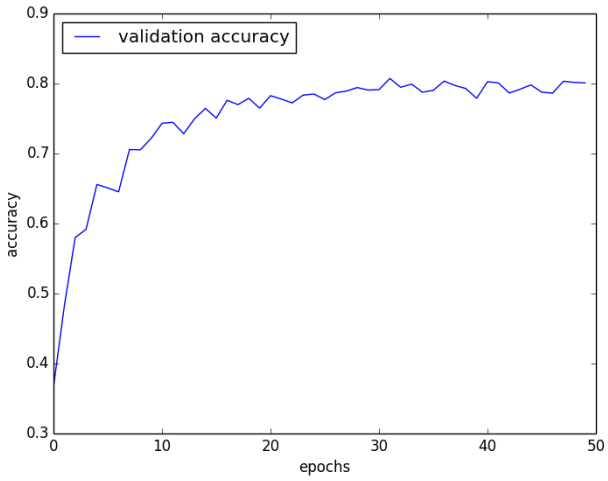


Figure 1: Variação da precisão da CNN ao longo dos ciclos de treino

O treino da rede neuronal convoluncional permitiu obter uma precisão de 78.7%, um valor substancialmente superior ao obtido na abordagem SVM. Foram testadas algumas configurações alternativas da rede neuronal, mas este processo não foi exaustivo porque o processo de treino da rede é extremamente demorado, especialmente quando não é possível utilizar o GPU. Por outro lado, o processo de classificação das imagens é bastante rápido. O tempo necessário para o treino da rede é a sua principal desvantagem, sendo mesmo mais longo do que o já demorado SVM.

3.2 *Support Vector Machines*

A variante *Multiclass Ranking* SVM permitiu obter o valor mais elevado de precisão 24.5% com um vocabulário de 3500 palavras.

Na variante *One Against All* SVM obteve-se 16.9% com um vocabulário de 3000 palavras e com um *dataset* de treino equilibrado conseguese uma melhoria de precisão para 19.9% com um vocabulário de 2000 palavras.

A variante Multiclass ranking SVM apesar de só possuir um classificador SVM tem um processo de treino mais lento do que as outras variantes constituídas por dez classificadores binários SVM.

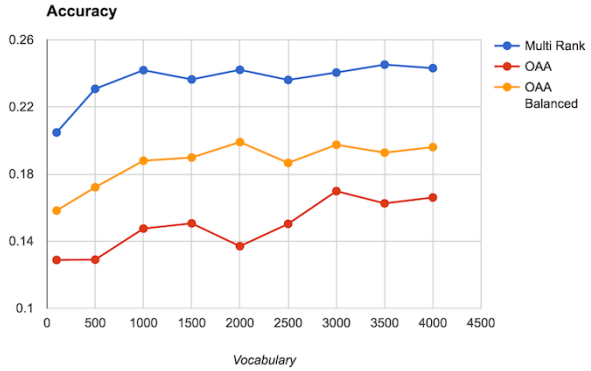


Figure 2: Variação da precisão das abordagens SVM para diferentes dimensões de vocabulário

4 Conclusões

A rede neuronal convoluncional teve uma prestação superior às variantes da abordagem SVM, o que está de acordo com o que era esperado apesar de a diferença entre as prestações ser superior às nossas expectativas. O SVM tem dificuldade em processar *datasets* com um elevado número de dimensões porque a maior parte desses espaços de *features* são constituídos por dados que não são separáveis. Como o fundamento do modelo SVM é a procura de um hiperplano que separe os dados desse espaço de features, se os dados não são separáveis é geometricamente impossível encontrar uma solução.

A rede neuronal convoluncional não tem estas limitações apesar de que o treino de uma rede é uma tarefa bastante dispendiosa ao nível do tempo e dos recursos. É importante referir que é difícil antever os resultados de uma determinada rede neuronal com base na sua estrutura e nas características das suas camadas.

Concluímos que, nos problemas de reconhecimento de objetos, as redes neuronais convoluncionais têm uma clara vantagem sobre as abordagens mais clássicas mesmo considerando o maior custo do seu processo de treino.

5 Melhoramentos Futuros

Relativamente às variantes da abordagem SVM seria interessante realizar o processo de treino dos classificadores SVM de forma a obter os parâmetros ótimos com base na estimativa do erro da validação cruzada. Outra melhoria do trabalho realizado seria a utilização de um *dataset* maior na variante OAA SVM *Balanced* porque o *dataset* utilizado para cada um dos classificadores SVM desta variante é um quinto do dataset utilizado nas outras duas variantes.

No caso da rede neuronal convoluncional, como já foi referido, devido à dificuldade em prever o impacto da estrutura e da configuração da rede nos resultados seria importante despender mais algum tempo na sua otimização.

Referências

[1] I. Sutskever A. Krizhevsky and G. E. Hinton. *Imagenet classification with deep convolutional neural networks*. Advances in neural information processing systems, 2012.

[2] Ben Aisen. *A Comparison of Multiclass SVM Methods*. http://courses.media.mit.edu/2006fall/mas622j/Projects/aisen-project/, 2006.

[3] Jorge A. Silva and Luís A. Teixeira. *Slides das Aulas*. Visão por Computador, 2016.

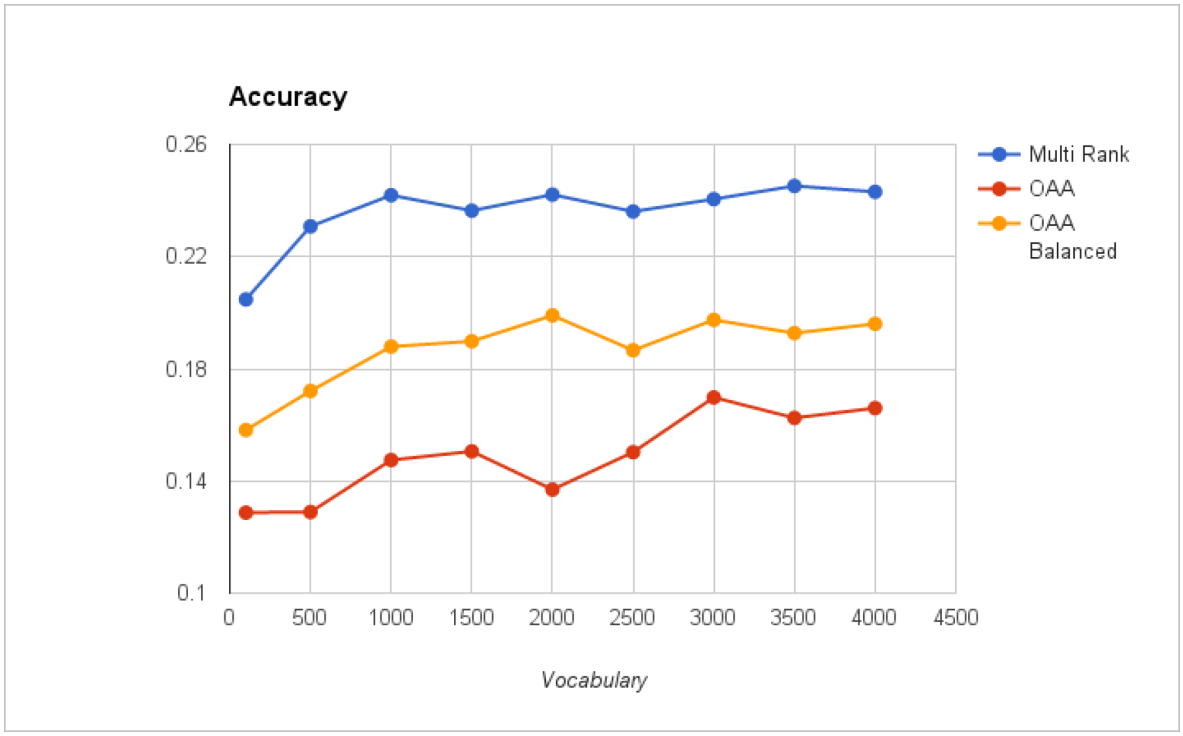
[4] J. Sivic and A. Zisserman. *Video Google: A Text Retrieval Approach to Object Matching in Videos*. ICCV, 2003.

Anexo

Listagem do código

```
vcom_proj_2/
|
|--- cnn/
|     |--- src/
|          |--- obj_rec.py
|          |--- obj_rec_model.h5
|          |--- obj_rec_model.50.h5
|
|--- svm_oaa/
|     |--- img/
|     |--- src/
|     |     |--- main.cpp
|     |     |--- svm.cpp
|     |     |--- svm.hpp
|     |--- Makefile
|
|--- svm_rank/
      |--- img/
      |--- src/
      |     |--- main.cpp
      |     |--- svm.cpp
      |     |--- svm.hpp
      |--- Makefile
```

	Accuracy		
Vocabulary Size	Multi Rank	OAA	OAA Balanced
100	0.204748	0.128719	0.158169
500	0.230792	0.128919	0.172093
1000	0.241911	0.147451	0.187919
1500	0.236402	0.150556	0.189823
2000	0.242112	0.136933	0.199038
2500	0.236101	0.150255	0.186617
3000	0.240509	0.169789	0.197436
3500	0.245217	0.162476	0.192728
4000	0.243113	0.165982	0.196033



```
// ============================
// Convolutional Neural Network
// ============================

import sys
import argparse
import matplotlib.pyplot as plt
import pandas as pd

from keras.models import Sequential
from keras.models import load_model
from keras.layers import Convolution2D
from keras.layers import Activation
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.utils import np_utils
from keras import backend as K

def main():
    """Main Function"""

    # Usage: obj_rec.py [-h] [-l FILENAME] [-e]
    parser = argparse.ArgumentParser(
        description='Object Recognition - Deep Learning')
    parser.add_argument(
        '-l', '--load', type=str, help='load model file', \
        metavar='FILENAME')
    parser.add_argument(
        '-e', '--evaluate', help='evaluate model with test images', \
        action='store_true')
    parser.add_argument(
        '-s', '--showerrors', help='show classification errors', \
        action='store_true')
    args = parser.parse_args()

    # load data set
    x_train, y_train, x_test, y_test, input_shape = load_cifar10_dataset()

    if args.load:
        print "\nLoading Model"
        model = load_model(args.load)
        model.summary()
        print "Model successfully loaded"

    if args.evaluate:
        score = model.evaluate(x_test, y_test)
        print "\nTest accuracy: %0.05f" % score[1]

    if args.showerrors:
        show_class_errors(model, x_test, y_test)

    if args.evaluate or args.showerrors:
        sys.exit()

    if not args.load:

        # create model
        nb_filters = 32
```

```
pool_size = (2, 2)
kernel_size = (3, 3)

# the sequential model is a linear stack of layers
model = Sequential()

# Convolution operator for filtering windows of two-dimensional
    inputs
model.add(Convolution2D(
    nb_filters, \
    kernel_size[0], \
    kernel_size[1], \
    border_mode='valid', \
    input_shape=input_shape))

# Activation layer with relu
model.add(Activation('relu'))

model.add(Convolution2D(
    nb_filters, \
    kernel_size[0], \
    kernel_size[1]))

model.add(Activation('relu'))

# Max Pooling
model.add(MaxPooling2D(pool_size=pool_size))

# Dropout consists in randomly setting a fraction p
# of input units to 0 at each update during training time,
# which helps prevent overfitting.
model.add(Dropout(0.25))

model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))

model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.25))

# Flattens the input. Does not affect the batch size.
model.add(Flatten())

# Regular fully connected NN layer
model.add(Dense(512))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(10))

model.add(Activation('softmax'))

# model summary
model.summary()

# Configures the learning process
```

```
        model.compile(
            loss='categorical_crossentropy', \
            optimizer='adadelta', \
            metrics=['accuracy'])

    # train the model
    history = model.fit(
        x_train, \
        y_train, \
        batch_size=32, \
        nb_epoch=5, \
        verbose=1, \
        validation_split=0.1)

    # test the model
    score = model.evaluate(x_test, y_test)
    print "\nTest accuracy: %0.05f" % score[1]

    # Graph with training accuracy
    plot_training_history(history)

    # Saves model in a HDF5 file, includes:
    # Architecture of the model
    # Weights of the model
    # Training config
    # State of the optimizer
    model.save('obj_rec_model.h5')


def load_cifar10_dataset():
    """Loads CIFAR10 data set"""
    from keras.datasets import cifar10

    ## 32x32 color images: shape -> (nb_samples, 3, 32, 32)
    img_w = img_h = 32
    img_channels = 3

    ## Load cifar10 data set
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()

    ## th -> (samples, channels, height, width)
    if K.image_dim_ordering() == 'th':
        x_train = x_train.reshape(
            x_train.shape[0], img_channels, img_w, img_h)
        x_test = x_test.reshape(
            x_test.shape[0], img_channels, img_w, img_h)
        input_shape = (
            img_channels, img_w, img_h)
    ## tf -> (samples, height, width, channels)
    else:
        x_train = x_train.reshape(
            x_train.shape[0], img_w, img_h, img_channels)
        x_test = x_test.reshape(
            x_test.shape[0], img_w, img_h, img_channels)
        input_shape = (
            img_w, img_h, img_channels)

    x_train = x_train.astype('float32')/255
    x_test = x_test.astype('float32')/255
    y_train = np_utils.to_categorical(y_train, 10)
```

```
    y_test = np_utils.to_categorical(y_test, 10)
    return (x_train, y_train, x_test, y_test, input_shape)

def plot_training_history(history):
    """Draws a graph with the trainning history"""
    plt.plot(history.history['val_acc'])
    plt.ylabel('accuracy')
    plt.xlabel('epochs')
    plt.legend(['validation accuracy'], loc='upper left')
    plt.show()

def show_class_errors(model, x_test, y_test):
    """Shows classification errors"""
    # Predict test images classes
    img_classes = (
        "airplane", \
        "automobile", \
        "bird", \
        "cat", \
        "deer", \
        "dog", \
        "frog", \
        "horse", \
        "ship", \
        "truck")

    y_hat = model.predict_classes(x_test)
    y_test_array = y_test.argmax(1)
    pd.crosstab(y_hat, y_test_array)
    test_wrong = [im for im in zip(x_test, y_hat, y_test_array) if im[1] !=
        im[2]]
    plt.figure(figsize=(15, 15))
    for ind, val in enumerate(test_wrong[:20]):
        plt.subplot(10, 10, ind + 1)
        im = val[0]
        plt.axis("off")
        plt.text(0, 0, img_classes[val[2]], fontsize=14, color='green') #
            correct
        plt.text(0, 32, img_classes[val[1]], fontsize=14, color='red')  #
            predicted
        plt.imshow(im, cmap='gray')
    plt.show()

if __name__ == '__main__':
    main()
```

```
// ============================
// One Against All SVM
// ============================
// ============================
// MAIN.CPP
// ============================

#include "svm.hpp"

// Main Function
int main(int argc, const char* argv[]) {

    std::cout << "OpenCV Version: " << CV_VERSION << std::endl;

    if (argc > 4) {
        std::cout << "Usage: ./obj_rec_svm nWords nTrainImg [balanced]" <<
            std::endl;
        exit(-1);
    }

    cv::initModule_nonfree();

    int nWords = atoi(std::string(argv[1]).c_str());
    int nTrainImg = atoi(std::string(argv[2]).c_str());
    int balanced = false;

    if((argc == 4) && (std::string(argv[3]).compare("balanced") == 0))
    {
        balanced = true;
    }

    std::cout << "Num Words = " << nWords << std::endl;
    std::cout << "Num Img   = " << nTrainImg << std::endl;
    std::cout << "Balanced  = " << balanced << std::endl;

    ObjRec obj_rec(nWords,nTrainImg);
    cv::Mat descriptors = obj_rec.getDescriptors();
    cv::Mat vocabulary = obj_rec.getVocabulary(descriptors);

    // Support Vector Machines - One vs. All

    cv::Mat trainData;
    cv::Mat trainLabels;
    std::vector<cv::SVM*> svmVec;

    // airplane
    cv::SVM airplaneSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels,
        "airplane", balanced);
    obj_rec.trainSVM(trainData, trainLabels, airplaneSVM);
    svmVec.push_back(&airplaneSVM);
    trainData.release();
    trainLabels.release();

    // automobile
    cv::SVM automobileSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels,
        "automobile", balanced);
    obj_rec.trainSVM(trainData, trainLabels, automobileSVM);
    svmVec.push_back(&automobileSVM);
```

```
    trainData.release();
    trainLabels.release();

    // bird
    cv::SVM birdSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels, "bird",
        balanced);
    obj_rec.trainSVM(trainData, trainLabels, birdSVM);
    svmVec.push_back(&birdSVM);
    trainData.release();
    trainLabels.release();

    // cat
    cv::SVM catSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels, "cat",
        balanced);
    obj_rec.trainSVM(trainData, trainLabels, catSVM);
    svmVec.push_back(&catSVM);
    trainData.release();
    trainLabels.release();

    // deer
    cv::SVM deerSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels, "deer",
        balanced);
    obj_rec.trainSVM(trainData, trainLabels, deerSVM);
    svmVec.push_back(&deerSVM);
    trainData.release();
    trainLabels.release();

    // dog
    cv::SVM dogSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels, "dog",
        balanced);
    obj_rec.trainSVM(trainData, trainLabels, dogSVM);
    svmVec.push_back(&dogSVM);
    trainData.release();
    trainLabels.release();

    // frog
    cv::SVM frogSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels, "frog",
        balanced);
    obj_rec.trainSVM(trainData, trainLabels, frogSVM);
    svmVec.push_back(&frogSVM);
    trainData.release();
    trainLabels.release();

    // horse
    cv::SVM horseSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels, "horse",
        balanced);
    obj_rec.trainSVM(trainData, trainLabels, horseSVM);
    svmVec.push_back(&horseSVM);
    trainData.release();
    trainLabels.release();

    // ship
    cv::SVM shipSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels, "ship",
```

```
        balanced);
    obj_rec.trainSVM(trainData, trainLabels, shipSVM);
    svmVec.push_back(&shipSVM);
    trainData.release();
    trainLabels.release();

    // truck
    cv::SVM truckSVM;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels, "truck",
        balanced);
    obj_rec.trainSVM(trainData, trainLabels, truckSVM);
    svmVec.push_back(&truckSVM);
    trainData.release();
    trainLabels.release();

    // Test SVM
    obj_rec.testSVM(vocabulary, svmVec);

    return 0;
}
```

```
// ============================
// One Against All SVM
// ============================
// ============================
// SVM.HPP
// ============================
#ifndef svm_hpp
#define svm_hpp

// #include <fstream>
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <limits>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/nonfree.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/ml/ml.hpp>

class ObjRec
{
public:
    cv::Mat dictionary;
    int nWords;
    int nTrainImg;

    ObjRec(int nWords, int nTrainImg);

    // Get image features descriptors for each training images
    cv::Mat getDescriptors();

    // kmeans -based class to train visual vocabulary using the bag of
        visual words approach
    // Clusters train descriptors
    // The vocabulary consists of cluster centers.
    cv::Mat getVocabulary(const cv::Mat& descriptors);

    // Get BoW histogram for each training images
    void prepareSVMtrainData(const cv::Mat& vocabulary, cv::Mat& trainData,
        cv::Mat& trainLabels, std::string classLabel, bool balanced);

    // Get int value for image label
    int getLabelVal(std::string label);

    // Train SVM classifier
    int trainSVM(const cv::Mat& trainData, const cv::Mat& trainLabels, cv::
        SVM& svm);

    // Test SVM classifier
    int testSVM(const cv::Mat& vocabulary, const std::vector<cv::SVM*>
        svmVec);

};

#endif /* svm_hpp */
```

```
// ============================
// One Against All SVM
// ============================
// ============================
// SVM.CPP
// ============================
#include "svm.hpp"

ObjRec::ObjRec(int nWords, int nTrainImg)
{
    this->nWords = nWords;
    this->nTrainImg = nTrainImg;
}

cv::Mat ObjRec::getDescriptors()
{
    std::ostringstream oss;

    cv::Mat image;
    cv::Mat descriptors;
    cv::Mat allDescriptors;
    std::vector<cv::KeyPoint> keypoints;

    std::cout << "[Object Recognition] Features Extraction" << std::endl;

    cv::Ptr<cv::FeatureDetector> detector =
        cv::FeatureDetector::create("SIFT");
    cv::Ptr<cv::DescriptorExtractor> extractor = cv::DescriptorExtractor::
        create("SIFT");

    int n_img = (this->nTrainImg * 0.8);
    for (int i = 1; i <= n_img; i++)
    {
        std::cout << i << "/" << this->nTrainImg << std::endl;

        // Clear string stream
        oss.str(std::string());

        // Build filename
        oss << "img/" << i << ".png";
        std::string fileName = oss.str();

        image = cv::imread(fileName, CV_LOAD_IMAGE_GRAYSCALE);

        if (!image.data)
        {
            std::cout << "Could not open or find the image" << std::endl;
            continue;
        }

        // Detects features in an image
        detector->detect(image, keypoints);

        if(keypoints.empty())
        {
            continue;
        }

        // Computes the descriptors for a set of keypoints detected in an
            image
```

```
            extractor->compute(image, keypoints, descriptors);

            // Store features descritors
            allDescriptors.push_back(descriptors);
        }

        return allDescriptors;
    }

    cv::Mat ObjRec::getVocabulary(const cv::Mat &descriptors)
    {
        cv::Mat vocabulary;

        std::cout << "[Object Recognition] BoW - training visual vocabulary" <<
            std::endl;
        std::cout << "[Object Recognition] BoW - Clustering " << descriptors.
            rows << " features" << std::endl;

        // number of words
        int clusterCount = this->nWords;

        // the number of times the algorithm is executed using different initial
            labellings
        int attempts = 1;

        // Use kmeans++ center initialization
        int flags = cv::KMEANS_PP_CENTERS;

        // kmeans -based class to train visual vocabulary using the bag of
            visual words approach
        cv::BOWKMeansTrainer bowTrainer(clusterCount, cv::TermCriteria(),
            attempts, flags);

        // input descriptors are clustered, returns the vocabulary
        vocabulary = bowTrainer.cluster(descriptors);

        return vocabulary;
    }

    void ObjRec::prepareSVMtrainData(const cv::Mat &vocabulary, cv::Mat &
        trainData, cv::Mat &trainLabels, std::string classLabel, bool balanced)
    {
        std::cout << "[Object Recognition] Train data for SVM: " << classLabel <
            < std::endl;

        // Detects keypoints in an image
        cv::Ptr<cv::FeatureDetector> detector =
            cv::FeatureDetector::create("SIFT");

        // Descriptor extractor that is used to compute descriptors for an input
            image and its keypoints
        cv::Ptr<cv::DescriptorExtractor> extractor = cv::DescriptorExtractor::
            create("SIFT");

        // Descriptor matcher that is used to find the nearest word of the
            trained vocabulary
        // for each keypoint descriptor of the image
        cv::Ptr<cv::DescriptorMatcher> matcher =
            cv::DescriptorMatcher::create("FlannBased");
```

```
// compute an image descriptor using the bag of visual words
// 1 - Compute descriptors for a given image and its keypoints set.
// 2 - Find the nearest visual words from the vocabulary for each
    keypoint descriptor.
// 3 - Compute the bag-of-words image descriptor as is
// a normalized histogram of vocabulary words encountered in the image
cv::BOWImgDescriptorExtractor bowDE(extractor, matcher);

// Sets a visual vocabulary
// Each row of the vocabulary is a visual word (cluster center)
bowDE.setVocabulary(vocabulary);

std::ostringstream oss;
cv::Mat image;
cv::Mat bowDescriptor;
std::vector<cv::KeyPoint> keypoints;

// Open csv file with labels for each train image
std::ifstream infile("img/trainLabels.csv");

// Remove header from csv file
std::string s;
std::getline(infile, s);

int counterPos = 0;
int counterNeg = 0;

int n_img = (this->nTrainImg * 0.8);
for (int i = 1; i <= n_img; i++)
{
    // TODO: CHANGE THIS!!!!
    //std::cout << i << "/" << this->nTrainImg << std::endl;

    // Clear string stream
    oss.str(std::string());

    // Build filename
    oss << "img/" << i << ".png";
    std::string fileName = oss.str();

    image = cv::imread(fileName, CV_LOAD_IMAGE_GRAYSCALE);

    if (!image.data)
    {
        std::cout << "Could not open or find the image" << std::endl;
        continue;
    }

    // Detects features in an image
    detector->detect(image, keypoints);

    if (keypoints.empty())
    {
        std::getline(infile, s);
        //std::cout << "Could not find keypoints in the image" <<
            std::endl;
        continue;
    }

    // Computes an image descriptor using the set visual vocabulary.
```

```
            bowDE.compute(image, keypoints, bowDescriptor);

            // Store train labels
            if (!std::getline(infile, s))
            {
                std::cout << "Unable to get label line" << std::endl;
                continue;
            }

            std::istringstream ss(s);
            std::getline(ss, s, ',');
            std::getline(ss, s, ',');

            if (balanced)
            {
                if (s.compare(classLabel) == 0)
                {
                    trainData.push_back(bowDescriptor);
                    trainLabels.push_back(1);
                    counterPos++;
                }
                else
                {
                    if (counterNeg < counterPos)
                    {
                        trainData.push_back(bowDescriptor);
                        trainLabels.push_back(-1);
                        counterNeg++;
                    }
                }
            }
            else
            {
                if (s.compare(classLabel) == 0)
                {
                    trainData.push_back(bowDescriptor);
                    trainLabels.push_back(1);
                    counterPos++;
                }
                else
                {
                    trainData.push_back(bowDescriptor);
                    trainLabels.push_back(-1);
                    counterNeg++;
                }
            }
        }

        std::cout << "Pos Samples = " << counterPos << std::endl;
        std::cout << "Neg Samples = " << counterNeg << std::endl;

    }

    int ObjRec::getLabelVal(std::string label)
    {
        int labelVal = -1;

        if (label.compare("airplane") == 0)
        {
            labelVal = 0;
```

```
    }

    if (label.compare("automobile") == 0)
    {
        labelVal = 1;
    }

    if (label.compare("bird") == 0)
    {
        labelVal = 2;
    }

    if (label.compare("cat") == 0)
    {
        labelVal = 3;
    }

    if (label.compare("deer") == 0)
    {
        labelVal = 4;
    }

    if (label.compare("dog") == 0)
    {
        labelVal = 5;
    }

    if (label.compare("frog") == 0)
    {
        labelVal = 6;
    }

    if (label.compare("horse") == 0)
    {
        labelVal = 7;
    }

    if (label.compare("ship") == 0)
    {
        labelVal = 8;
    }

    if (label.compare("truck") == 0)
    {
        labelVal = 9;
    }

    return labelVal;
}

int ObjRec::trainSVM(const cv::Mat &trainData, const cv::Mat &trainLabels,
    cv::SVM &svm)
{
    std::cout << "[Object Recognition] Training SVM" << std::endl;

    // SVM Parameters
    CvSVMParams params;

    // Train SVM
    svm.train(trainData, trainLabels, cv::Mat(), cv::Mat(), params);
```

```
    return 0;
}

int ObjRec::testSVM(const cv::Mat& vocabulary, const std::vector<cv::SVM*>
    svmVec)
{
    std::cout << "[Object Recognition] Testing SVM" << std::endl;

    cv::Mat testData;
    cv::Mat testLabels;
    cv::Mat testClass;
    int n_class = 10;
    double distances[n_class];

    // Detects keypoints in an image
    cv::Ptr<cv::FeatureDetector> detector =
        cv::FeatureDetector::create("SIFT");

    // Descriptor extractor that is used to compute descriptors for an input
        image and its keypoints
    cv::Ptr<cv::DescriptorExtractor> extractor = cv::DescriptorExtractor::
        create("SIFT");

    // Descriptor matcher that is used to find the nearest word of the
        trained vocabulary
    // for each keypoint descriptor of the image
    cv::Ptr<cv::DescriptorMatcher> matcher =
        cv::DescriptorMatcher::create("FlannBased");

    // compute an image descriptor using the bag of visual words
    // 1 - Compute descriptors for a given image and its keypoints set.
    // 2 - Find the nearest visual words from the vocabulary for each
        keypoint descriptor.
    // 3 - Compute the bag-of-words image descriptor as is
    // a normalized histogram of vocabulary words encountered in the image
    cv::BOWImgDescriptorExtractor bowDE(extractor, matcher);

    // Sets a visual vocabulary
    // Each row of the vocabulary is a visual word (cluster center)
    bowDE.setVocabulary(vocabulary);

    std::ostringstream oss;
    cv::Mat image;
    cv::Mat bowDescriptor;
    std::vector<cv::KeyPoint> keypoints;

    // Open csv file with labels for each train image
    std::ifstream infile("img/trainLabels.csv");

    // Remove header from csv file
    std::string s;
    std::getline(infile, s);

    int n_img = (this->nTrainImg * 0.8);
    for (int i = 1; i <= n_img; i++)
    {
        std::getline(infile, s);
    }
```

```
    for (int i = n_img + 1; i <= n_img + (this->nTrainImg * 0.2); i++)
    {
        std::cout << i << "/" << this->nTrainImg << std::endl;

        // Clear string stream
        oss.str(std::string());

        // Build filename
        oss << "img/" << i << ".png";
        std::string fileName = oss.str();

        image = cv::imread(fileName, CV_LOAD_IMAGE_GRAYSCALE);

        if (!image.data)
        {
            std::cout << "Could not open or find the image" << std::endl;
            continue;
        }

        // Detects features in an image
        detector->detect(image, keypoints);

        if (keypoints.empty())
        {
            std::getline(infile, s);
            //std::cout << "Could not find keypoints in the image" <<
                std::endl;
            continue;
        }

        // Computes an image descriptor using the set visual vocabulary.
        bowDE.compute(image, keypoints, bowDescriptor);
        testData.push_back(bowDescriptor);

        // Store train labels
        if (!std::getline(infile, s))
        {
            continue;
        }

        std::istringstream ss(s);
        std::getline(ss, s, ',');
        std::getline(ss, s, ',');

        // std::cout << s << " = " << getLabelVal(s) << std::endl;
        testLabels.push_back(getLabelVal(s));

        // airplane
        distances[0] = svmVec[0]->predict(bowDescriptor, true);

        // automobile
        distances[1] = svmVec[1]->predict(bowDescriptor, true);

        // bird
        distances[2] = svmVec[2]->predict(bowDescriptor, true);

        // cat
        distances[3] = svmVec[3]->predict(bowDescriptor, true);

        // deer
```

```
            distances[4] = svmVec[4]->predict(bowDescriptor, true);

            // dog
            distances[5] = svmVec[5]->predict(bowDescriptor, true);

            // frog
            distances[6] = svmVec[6]->predict(bowDescriptor, true);

            // horse
            distances[7] = svmVec[7]->predict(bowDescriptor, true);

            // ship
            distances[8] = svmVec[8]->predict(bowDescriptor, true);

            // truck
            distances[9] = svmVec[9]->predict(bowDescriptor, true);

            // Get SVM classification
            int classification = -1;
            double dist = std::numeric_limits<double>::max();

            for(int z=0; z<n_class; z++)
            {
                if(distances[z] < dist)
                {
                    classification = z;
                    dist = distances[z];
                }
            }

            testClass.push_back(classification);

            // Calculate classification rate
            double rateIt = 1 - ((double)cv::countNonZero(testLabels - testClass
                ) / testData.rows);
            std::cout << "Iteration Rate = " << rateIt << std::endl;

        }

        std::cout << "[Object Recognition] Final Results " << std::endl;

        // Calculate classification rate
        double rate = 1 - ((double)cv::countNonZero(testLabels - testClass) /
            testData.rows);
        std::cout << "Classification Rate = " << rate << std::endl;

        return 0;
    }
```

```cpp
// ============================
// Multiclass Ranking SVM
// ============================
// ============================
// MAIN.CPP
// ============================

#include "svm.hpp"

// Main Function
int main(int argc, const char* argv[]) {

    std::cout << "OpenCV Version: " << CV_VERSION << std::endl;

    if (argc != 3) {
        std::cout << "Usage: ./obj_rec_svm nWords nTrainImg" << std::endl;
        exit(-1);
    }

    cv::initModule_nonfree();

    int nWords = atoi(std::string(argv[1]).c_str());
    int nTrainImg = atoi(std::string(argv[2]).c_str());

    ObjRec obj_rec(nWords,nTrainImg);
    cv::Mat descriptors = obj_rec.getDescriptors();
    cv::Mat vocabulary = obj_rec.getVocabulary(descriptors);

    cv::Mat trainData;
    cv::Mat trainLabels;
    obj_rec.prepareSVMtrainData(vocabulary, trainData, trainLabels);

    //std::cout << trainData << std::endl;
    //std::cout << trainLabels << std::endl;

    // Support Vector Machines
    cv::SVM svm;

    // Train SVM
    obj_rec.trainSVM(trainData, trainLabels, svm);

    // Test SVM
    obj_rec.testSVM(vocabulary, svm);

    return 0;
}
```

```
// ============================
// Multiclass Ranking SVM
// ============================
// ============================
// SVM.HPP
// ============================
#ifndef svm_hpp
#define svm_hpp

// #include <fstream>
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <map>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/nonfree.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/ml/ml.hpp>

class ObjRec
{
public:
    cv::Mat dictionary;
    int nWords;
    int nTrainImg;

    ObjRec(int nWords, int nTrainImg);

    // Get image features descriptors for each training images
    cv::Mat getDescriptors();

    // kmeans -based class to train visual vocabulary using the bag of
        visual words approach
    // Clusters train descriptors
    // The vocabulary consists of cluster centers.
    cv::Mat getVocabulary(const cv::Mat& descriptors);

    // Get BoW histogram for each training images
    void prepareSVMtrainData(const cv::Mat& vocabulary, cv::Mat& trainData,
        cv::Mat& trainLabels);

    // Get int value for image label
    int getLabelVal(std::string label);

    // Train SVM classifier
    int trainSVM(const cv::Mat& trainData, const cv::Mat& trainLabels, cv::
        SVM& svm);

    // Test SVM classifier
    int testSVM(const cv::Mat& vocabulary, const cv::SVM& svm);

};

#endif /* svm_hpp */
```

```
// ============================
// Multiclass Ranking SVM
// ============================
// ============================
// SVM.CPP
// ============================
#include "svm.hpp"

ObjRec::ObjRec(int nWords, int nTrainImg)
{
    this->nWords = nWords;
    this->nTrainImg = nTrainImg;

}

cv::Mat ObjRec::getDescriptors()
{
    std::ostringstream oss;

    cv::Mat image;
    cv::Mat descriptors;
    cv::Mat allDescriptors;
    std::vector<cv::KeyPoint> keypoints;

    std::cout << "[Object Recognition] Features Extraction" << std::endl;

    cv::Ptr<cv::FeatureDetector> detector =
        cv::FeatureDetector::create("SIFT");
    cv::Ptr<cv::DescriptorExtractor> extractor = cv::DescriptorExtractor::
        create("SIFT");

    int n_img = (this->nTrainImg * 0.8);
    for(int i=1; i<=n_img; i++)
    {
        std::cout << i << "/" << this->nTrainImg << std::endl;

        // Clear string stream
        oss.str(std::string());

        // Build filename
        oss << "img/" << i << ".png";
        std::string fileName = oss.str();

        image = cv::imread(fileName, CV_LOAD_IMAGE_GRAYSCALE);

        if(!image.data)
        {
            std::cout <<  "Could not open or find the image" << std::endl;
            continue;
        }

        // Detects features in an image
        detector->detect(image, keypoints);

        // Computes the descriptors for a set of keypoints detected in an
            image
        extractor->compute(image, keypoints, descriptors);

        // Store features descritors
        allDescriptors.push_back(descriptors);
```

```
    }

    return allDescriptors;

}

cv::Mat ObjRec::getVocabulary(const cv::Mat& descriptors)
{
    cv::Mat vocabulary;

    std::cout << "[Object Recognition] BoW - training visual vocabulary" <<
        std::endl;
    std::cout << "[Object Recognition] BoW - Clustering " << descriptors.
        rows << " features" << std::endl;

    // number of words
    int clusterCount = this->nWords;

    // the number of times the algorithm is executed using different initial
        labellings
    int attempts = 1;

    // Use kmeans++ center initialization
    int flags = cv::KMEANS_PP_CENTERS;

    // kmeans -based class to train visual vocabulary using the bag of
        visual words approach
    cv::BOWKMeansTrainer bowTrainer(clusterCount, cv::TermCriteria(),
        attempts, flags);

    // input descriptors are clustered, returns the vocabulary
    vocabulary = bowTrainer.cluster(descriptors);

    return vocabulary;
}

void ObjRec::prepareSVMtrainData(const cv::Mat& vocabulary, cv::Mat&
    trainData, cv::Mat& trainLabels)
{
    std::cout << "[Object Recognition] BoW - Getting BoW histogram for each
        training images" << std::endl;

    // Detects keypoints in an image
    cv::Ptr<cv::FeatureDetector> detector =
        cv::FeatureDetector::create("SIFT");

    // Descriptor extractor that is used to compute descriptors for an input
        image and its keypoints
    cv::Ptr<cv::DescriptorExtractor> extractor = cv::DescriptorExtractor::
        create("SIFT");

    // Descriptor matcher that is used to find the nearest word of the
        trained vocabulary
    // for each keypoint descriptor of the image
    cv::Ptr<cv::DescriptorMatcher> matcher =
        cv::DescriptorMatcher::create("FlannBased");

    // compute an image descriptor using the bag of visual words
    // 1 - Compute descriptors for a given image and its keypoints set.
    // 2 - Find the nearest visual words from the vocabulary for each
```

```
    keypoint descriptor.
// 3 - Compute the bag-of-words image descriptor as is
// a normalized histogram of vocabulary words encountered in the image
cv::BOWImgDescriptorExtractor bowDE(extractor, matcher);

// Sets a visual vocabulary
// Each row of the vocabulary is a visual word (cluster center)
bowDE.setVocabulary(vocabulary);

std::ostringstream oss;
cv::Mat image;
cv::Mat bowDescriptor;
std::vector<cv::KeyPoint> keypoints;

// Open csv file with labels for each train image
std::ifstream infile("img/trainLabels.csv");

// Remove header from csv file
std::string s;
std::getline(infile, s);

int n_img = (this->nTrainImg * 0.8);
for(int i=1; i<=n_img; i++)
{
    std::cout << i << "/" << this->nTrainImg << std::endl;

    // Clear string stream
    oss.str(std::string());

    // Build filename
    oss << "img/" << i << ".png";
    std::string fileName = oss.str();

    image = cv::imread(fileName, CV_LOAD_IMAGE_GRAYSCALE);

    if(!image.data)
    {
        std::cout <<  "Could not open or find the image" << std::endl;
        continue;
    }

    // Detects features in an image
    detector->detect(image, keypoints);

    // TODO: CHANGE THIS!!!!!!!
    if(keypoints.empty())
    {
        std::getline(infile, s);
        std::cout <<  "Could not find keypoints in the image" << std::
            endl;
        continue;
    }

    // Computes an image descriptor using the set visual vocabulary.
    bowDE.compute(image, keypoints, bowDescriptor);
    trainData.push_back(bowDescriptor);

    // Store train labels
    if(!std::getline(infile, s))
    {
```

```
            std::cout << "Unable to get label line" << std::endl;
            continue;
        }

        std::istringstream ss(s);
        std::getline(ss, s, ',');
        std::getline(ss, s, ',');

        //std::cout << s << " = " << getLabelVal(s) << std::endl;
        trainLabels.push_back((float) getLabelVal(s));

    }

}

int ObjRec::getLabelVal(std::string label)
{
    int labelVal = -1;

    if (label.compare("airplane") == 0)
    {
        labelVal = 0;
    }

    if (label.compare("automobile") == 0)
    {
        labelVal = 1;
    }

    if (label.compare("bird") == 0)
    {
        labelVal = 2;
    }

    if (label.compare("cat") == 0)
    {
        labelVal = 3;
    }

    if (label.compare("deer") == 0)
    {
        labelVal = 4;
    }

    if (label.compare("dog") == 0)
    {
        labelVal = 5;
    }

    if (label.compare("frog") == 0)
    {
        labelVal = 6;
    }

    if (label.compare("horse") == 0)
    {
        labelVal = 7;
    }

    if (label.compare("ship") == 0)
```

```
        {
            labelVal = 8;
        }

        if (label.compare("truck") == 0)
        {
            labelVal = 9;
        }

        return labelVal;

    }

    int ObjRec::trainSVM(const cv::Mat& trainData, const cv::Mat& trainLabels,
        cv::SVM& svm)
    {
        std::cout << "[Object Recognition] Training SVM" << std::endl;

        // SVM Parameters
        CvSVMParams params;

        // Train SVM
        svm.train(trainData, trainLabels, cv::Mat(), cv::Mat(), params);

        // Store trained SVM
        // TODO: CHANGE THIS!!!!!
        // svm.save("train_model.svm");

        return 0;
    }

    int ObjRec::testSVM(const cv::Mat& vocabulary, const cv::SVM& svm)
    {
        std::cout << "[Object Recognition] Testing SVM" << std::endl;

        cv::Mat testData;
        cv::Mat testLabels;
        cv::Mat testClass;

        // Detects keypoints in an image
        cv::Ptr<cv::FeatureDetector> detector =
            cv::FeatureDetector::create("SIFT");

        // Descriptor extractor that is used to compute descriptors for an input
            image and its keypoints
        cv::Ptr<cv::DescriptorExtractor> extractor = cv::DescriptorExtractor::
            create("SIFT");

        // Descriptor matcher that is used to find the nearest word of the
            trained vocabulary
        // for each keypoint descriptor of the image
        cv::Ptr<cv::DescriptorMatcher> matcher =
            cv::DescriptorMatcher::create("FlannBased");

        // compute an image descriptor using the bag of visual words
        // 1 - Compute descriptors for a given image and its keypoints set.
        // 2 - Find the nearest visual words from the vocabulary for each
            keypoint descriptor.
        // 3 - Compute the bag-of-words image descriptor as is
        // a normalized histogram of vocabulary words encountered in the image
```

```
cv::BOWImgDescriptorExtractor bowDE(extractor, matcher);

// Sets a visual vocabulary
// Each row of the vocabulary is a visual word (cluster center)
bowDE.setVocabulary(vocabulary);

std::ostringstream oss;
cv::Mat image;
cv::Mat bowDescriptor;
std::vector<cv::KeyPoint> keypoints;

// Open csv file with labels for each train image
std::ifstream infile("img/trainLabels.csv");

// Remove header from csv file
std::string s;
std::getline(infile, s);

// TODO: CHANGE THIS!!!!!!!
int n_img = (this->nTrainImg * 0.8);

for(int i=1; i<=n_img; i++)
{
    std::getline(infile, s);
}

for(int i=n_img+1; i<=n_img+(this->nTrainImg * 0.2); i++)
{
    std::cout << i << "/" << this->nTrainImg << std::endl;

    // Clear string stream
    oss.str(std::string());

    // Build filename
    oss << "img/" << i << ".png";
    std::string fileName = oss.str();

    image = cv::imread(fileName, CV_LOAD_IMAGE_GRAYSCALE);

    if(!image.data)
    {
        std::cout <<  "Could not open or find the image" << std::endl;
        continue;
    }

    // Detects features in an image
    detector->detect(image, keypoints);

    // TODO: CHANGE THIS!!!!!!!
    if(keypoints.empty())
    {
        std::getline(infile, s);
        std::cout <<  "Could not find keypoints in the image" << std::
            endl;
        continue;
    }

    // Computes an image descriptor using the set visual vocabulary.
    bowDE.compute(image, keypoints, bowDescriptor);
    testData.push_back(bowDescriptor);
```

```
            // Store train labels
            if(!std::getline(infile, s))
            {
                continue;
            }

            std::istringstream ss(s);
            std::getline(ss, s, ',');
            std::getline(ss, s, ',');

            // std::cout << s << " = " << getLabelVal(s) << std::endl;
            testLabels.push_back((float) getLabelVal(s));

            // Test SVM
            float classification = svm.predict(bowDescriptor);
            testClass.push_back(classification);

        }

        // Calculate classification rate
        double rate = 1 - ((double) cv::countNonZero(testLabels - testClass) /
            testData.rows);
        std::cout << "[Object Recognition] Classification Rate = " << rate <<
            std::endl;

        return 0;

    }
```