Statistical Machine Learning with OpenCV

Philipp Wagner*

January 10, 2011

Abstract

This document is an introduction to Machine Learning with OpenCV. Artificial Neural Networks and Support Vector Machines are briefly explained and examples in OpenCV are given.

1 Introduction

Machine Learning is a branch of Artificial Intelligence and concerned with the question how to make machines able to learn from data. The core idea is to enable a machine to make intelligent decisions and predictions based on experiences from the past. Algorithms of Machine Learning require interdisciplinary knowledge and often intersect with topics of statistics, mathematics, physics, pattern recognition and more.

OpenCV (Open Source Computer Vision) is a library for computer vision and comes with a machine learning library for:

- Decision Trees
- Boosting
- Support Vector Machines
- Expectation Maximization
- Neural Networks

This document helps with:

- Setting up a development environment in Windows and Linux
- Getting Started with CMake and OpenCV
- Using the Machine Learning Library of OpenCV

A brief introduction to Support Vector Machines and Neural Networks are given.

2 Setup

Follow this guide to have a nice Development Environment for Linux and Windows. You have to download Eclipse and CMake. Windows users can use MinGW as a C/C++ compiler.

2.1 Eclipse with CDT (Windows/Linux)

Eclipse is an Integrated Development Environment (IDE). It is possibly best known as a Java development platform, but can be extended for languages such as C/C++, PHP, Python and more. The easiest way to get Eclipse with C/C++ support is to download *Eclipse for C++ Developers* for your operating system at: http://www.eclipse.org/downloads/. The download already includes the CDT Plugin (C/C++ Development Toolkit).

There is no installation. Just unpack and start the eclipse executable.

*E-Mail: bytefish–at–gmx–dot–de

2.2 MinGW (Windows only)

This step is Windows only. MinGW (Minimalist GNU for Windows) is a port of the GNU Compiler Collection (GCC) and can be used for development of native Microsoft Windows applications. Download the automated mingw-get-installer from sourceforge (called *mingw-get-inst-20101030.exe* at time of writing this). If the path to the download changes, navigate there from the MinGW project page at: `http://www.mingw.org`. Be sure to select *"C++ Compiler"* in the *Compiler Suite* dialog during setup.

MinGW doesn't add its binaries to the global Windows PATH environment. The MinGW Page says: *Add* `C:\MinGW\bin;` *to the PATH environment variable by opening the System control panel, going to the Advanced tab, and clicking the Environment Variables button. If you currently have a Command Prompt window open, it will not recognize the change to the environment variables; you will need to open a new Command Prompt window to get the new PATH.*

2.3 CMake (Windows/Linux)

CMake is an open-source, cross-platform build system used by many opensource projects including: OpenCV, MiKTeX or Blender 3D. It works on Linux, Windows, Mac OS X and can generate (among many others) Eclipse CDT project files and that's why it is suggested as a build system for OpenCV projects.

2.4 Installing CMake

2.4.1 Windows

Download the installer from the Resources Page at `http://www.cmake.org` (called cmake-2.8.3-win32-x86.exe at time of writing). This will install cmake, ccmake and the cmake-gui.

Make sure to select *"Add CMake to the system PATH for all users"* during setup.

2.4.2 Linux

CMake is often available from a distributions repository or can be downloaded as an installer from the Resources Page at `http://www.cmake.org`. Installing the generic Linux binaries with the is done by typing:

```
sudo sh cmake-2.8.3-Linux-i386.sh --prefix=/usr/local
```

In Debian or Ubuntu cmake can be installed with apt:

```
sudo apt-get install cmake cmake-gui
```

2.5 OpenCV

OpenCV is a Computer Vision Library started by Intel in 1999 and now actively developed by Willow Garage. It includes advanced computer vision algorithms and sets the focus on real-time image processing. In 2009 OpenCV2 got a C++ interface and integrates with Python. If you encounter any problems with this install guide consult the OpenCV Install Guide aswell.

Windows

Mircosoft Windows users download the Windows installer from the sourceforge repository of OpenCV at `http://sourceforge.net/projects/opencvlibrary/`. (called OpenCV-2.2.0-win32-vs2010.exe at time of writing this).

Make sure to select *"Add OpenCV to the system PATH for all users"* during setup.

Linux

Linux users should inspect their repositories, because most distributions have pre-packaged binaries. In Debian/Ubuntu you simply type:

```
sudo apt-get install opencv opencv-doc
```

Build OpenCV from sources

OpenCV is open source and comes with CMake as build system. Sometimes you need to compile OpenCV yourself, for example if the pre-packaged binaries use SSE2 and your CPU doesn't support SSE2. If so, download the OpenCV sources, unpack them and cd to the directory.

For building OpenCV from source type:

```
cmake .
make
sudo make install
```

Building OpenCV without SSE2

If you are working on a CPU without SSE2 you will need to deselect the SSE2 build option before compiling it, otherwise OpenCV won't work as expected. To see which flags your CPU supports (assuming Linux), type `cat /proc/cpuinfo | grep flags` in a terminal.

On my workstation:

```
philipp@banana:~$ cat /proc/cpuinfo | grep flags
flags : fpu vme de pse tsc msr pae mce cx8 sep mtrr pge mca cmov pat pse36 mmx fxsr sse syscall mp mmxext 3dnowext 3dnow up ts fid vid
```

It's an AMD Athlon XP and does not support SSE2. In order to turn off SSE2 support for OpenCV with the cmake-gui:

- Start `cmake-gui` either from prompt or menu.
- Set the OpenCV source folder as *Source Directory* and *Build Directory*
- Click *configure* and choose: `Unix Makefiles`.
- Deselect *ENABLE_SSE2* support from the list of flags.
- Click *configure* again to accept the setup.
- Click *generate* and proceed with the normal compiling procedure as described above.

The command line for doing the same would be (assuming you are in the OpenCV folder):

```
cmake -DENABLE_SSE2=OFF .
```

followed by:

```
make
sudo make install
```

3 Getting started with CMake

This chapter serves as an introduction to CMake, because it is used for the OpenCV project. Instead of making up a simple Hello World, it will show how to write, test and build a library with CMake and the Boost Framework.

Problem

Imagine your are working on a project with GPS data and you are told to write and test a library, let's call it *geolib*, to calculate the distance between two positions on earth (also called great-circle distance or orthodromic distance). Points on earth are given in latitude (ϕ) and longitude (λ) and for the distance calculation one can use the haversine formula:

$$\operatorname{haversin}\left(\frac{d}{R}\right) = \operatorname{haversin}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{haversin}(\Delta\lambda). \tag{1}$$

Where $haversin$ is:

$$haversin(\theta) = \sin^2(\theta/2)$$

And applying the inverse haversine gives the distance d:

$$d = R\operatorname{haversin}^{-1}(h) = 2R\arcsin\left(\sqrt{h}\right)$$

Where h is:

$$h = \operatorname{haversin}(\frac{d}{R})$$

Files

The final project has a structure shown in Listing 1. The source files are in the appendix or can be downloaded from: http://bytefish.de/lib/exe/fetch.php?media=blog:cmake_example.zip.

Listing 1: Project structure.

```
philipp@banana:~/git/cmake$ tree src
src/
|-- CMakeLists.txt
|-- geolib
| |-- geolib.cc
| |-- geolib.h
| `-- test.cc
`-- main.cc

1 directory, 5 files
```

Implementation

You start by implementing the library. The header file in Listing 2 is using Doxygen annotations to document the code. Doxygen is not introduced here, but it is strongly advised to document your work (because someone wants to *use* and *understand* it aswell). The corresponding source file in Listing **??** implements the haversine formula given in equation 1.

Listing 2: lst:geolib.h

```
#ifndef _geolib_h
#define _geolib_h
#include <math.h>

/**
* @author philipp
* @date 2010
*/
namespace geolib {

    const double DEG2RAD = M_PI / 180.0;
    const double RAD2DEG = 180.0 / M_PI;
    const double EARTH_RADIUS = 6371.01;

    /**
     * @brief great-circle distance (shortest distance between two points on earth)
     * @param[in] latitude point 1
     * @param[in] longitude point 1
     * @param[in] latitude point 2
     * @param[in] longitude point 2
     * @return great-circle distance
    */
    double great_circle_distance(double lat1, double long1, double lat2, double long2);

}

#endif
```

The corresponding source file in geolib.cc implements the haversine formula given in the projects description:

Listing 3: geometry.cc

```
#include "geolib.h"

namespace geolib {

    double great_circle_distance(double lat1, double long1, double lat2, double long2) {
        lat1 *= DEG2RAD;
        long1 *= DEG2RAD;
        lat2 *= DEG2RAD;
        long2 *= DEG2RAD;

        double deltaLat = lat1-lat2;
        double deltaLong = long1-long2;
        double a = pow(sin(deltaLat/2.0),2.0) + cos(lat1) * cos(lat2) * pow(sin(deltaLong/2.0),2.0);
        double c = 2 * asin(sqrt(a));

        return EARTH_RADIUS * c;
    }

}
```

3.1 Tests

Software has bugs. So tests are important in software development to ensure that components are working correctly. They are a sort of living documentation and should reduce defects and failures, especially when refactoring software. If you are used to JUnit you will find the Boost unit testing framework for C++ pleasing. Boost (http://www.boost.org) is a great library that adds enormous extra power to C++.

The Boost testing framework is a set of macros to validate expressions and each macro belongs to one of the three warning levels:

- **WARN** logs a warning if the test fails
- **CHECK** continues the test on failure
- **REQUIRE** stops the test on failure

The best introduction is probably given by IBM developerWorks: Get to know the Boost unit test framework. A list of available macros is given by the Boost documentation on testing tools Listing 4 shows how to test the great-circle distance of the geolib with the Boost unit testing framework. As test data I use the airport example from the Wikipedia Page on Great-circle distance:

Listing 4: test.cc

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE "GeoLib_Tests"
#include <boost/test/unit_test.hpp>
#include "geolib.h"

BOOST_AUTO_TEST_CASE(gcircle_test) {

        double expected = 2886.448973436;

        double bna_lat = 36.12;
        double bna_long = -86.67;
        double lax_lat = 33.94;
        double lax_long = -118.40;

        double actual = geolib::great_circle_distance(bna_lat, bna_long, lax_lat, lax_long);
        BOOST_CHECK_CLOSE(expected, actual, 1e-5);

}
```

3.2 CMakeLists.txt: Putting it all together

CMake searches for a file called *CMakeLists.txt* in a given directory. The CMakeLists.txt for this project is:

Listing 5: CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED( VERSION 2.6 )
PROJECT(HELLOWORLD_PROJECT)
INCLUDE_DIRECTORIES("${CMAKE_SOURCE_DIR}/geolib")
ADD_LIBRARY(geolib geolib/geolib.cc geolib/geolib.h)
ADD_EXECUTABLE(helloworld main.cc)
TARGET_LINK_LIBRARIES(helloworld geolib)

# Tests
FIND_PACKAGE( Boost REQUIRED COMPONENTS unit_test_framework)
INCLUDE_DIRECTORIES(${Boost_INCLUDE_DIRS})
ADD_EXECUTABLE(test_geolib geolib/test.cc)
TARGET_LINK_LIBRARIES(test_geolib ${Boost_LIBRARIES} geolib)
```

3.3 Build

Building the project can be done by an out of source build:

```
philipp@banana:~/git/cmake$ mkdir build && cd build
philipp@banana:~/git/cmake/build$ cmake ../src/
philipp@banana:~/git/cmake/build$ make
```

Running the executable *helloworld* calculates the distance from Berlin to Munich (in km):

```
philipp@banana:~/git/cmake/build$ ./helloworld
Distance is 508.978.
```

Running the test returns no errors:

```
philipp@banana:~/git/cmake/build$ ./test_geolib
Running 1 test case...

*** No errors detected
```

4 Getting Started with OpenCV

Now that Eclipse, MinGW (Windows), CMake and OpenCV are installed it is time to create the first OpenCV project. It is called *hello_opencv*. Files for this example are put in a folder at: `C:\workspace\hello_opencv`. Of course any valid path can be chosen and kept consistent with the example.

1. CMakeLists.txt

From the CMake introduction you should be familiar with CMake already. The OpenCV library is linked against the executable. Save CMakeLists.txt from Listing 6 to `C:\workspace\hello_opencv`.

Listing 6: CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6)
PROJECT(hello_opencv_proj)
FIND_PACKAGE( OpenCV REQUIRED )
ADD_EXECUTABLE( hello_opencv main.cpp)
TARGET_LINK_LIBRARIES( hello_opencv ${OpenCV_LIBS} )
```

2. main.cpp

Now the OpenCV C++ API is introduced.

```
#include "cv.h"
#include "ml.h"

using namespace std;

int main() {
        cv::Mat plot(240,320,CV_8UC3);
        plot.setTo(cv::Scalar(255,255,255));
        cv::putText(plot,"Hello_OpenCV",Point(1,100), FONT_HERSHEY_SIMPLEX, 1.0 ,Scalar(0,0,0), 1, 8, false);
        cv::namedWindow("Hello_OpenCV", CV_WINDOW_AUTOSIZE);
        cv::imshow("Hello_OpenCV", plot);
        cv::waitKey();
}
```

Save main.cpp from Listing 10 to `C:\workspace\hello_opencv`.

3. Generate project and makefiles

We will need to import the project to eclipse, so we generate the CDT specific Makefiles:

- Start the *cmake-gui*
- Browse to `C:\workspace\hello_opencv` as *Source Folder* and *Build Folder*
- Click *configure* and select "Eclipse CDT4 - Unix Makefiles" from the Dialog
- Click *configure* again to accept the configuration
- Click *generate* to generate the Makefiles and Eclipse related project files

Your folder will now look like this:

```
C:\workspace\hello_opencv>dir

11.12.2010 14:33 <DIR> .
11.12.2010 14:33 <DIR> ..
11.12.2010 14:33 22.202 .cproject
11.12.2010 14:33 3.591 .project
11.12.2010 14:28 29.062 CMakeCache.txt
11.12.2010 14:33 <DIR> CMakeFiles
11.12.2010 14:18 190 CMakeLists.txt
11.12.2010 14:28 1.470 cmake_install.cmake
11.12.2010 14:14 556 main.cpp
11.12.2010 14:33 5.179 Makefile
```

Or if you feel more familiar with a terminal type:

```
C:\> cd C:\workspace\hello_opencv
C:\> cmake -G"Eclipse CDT4 - Unix Makefiles" .
```

4. Import project and makefiles into Eclipse

1. `File -> Import...`
2. `General -> Existing Projects into Workspace`
3. Select `C:\workspace\hello_opencv` as root directory
4. Click `Finish`

5. Build and run the project

To build the project:

- `Project -> Build All`

To run the project:

- `Run -> Run`

And a window similar to Figure 4 pops up.

Figure 1: OpenCV: Hello World

5 Random Numbers in OpenCV

Each thread in OpenCV has access to a default random number generator cv::theRNG(). For a single pass the seed for the random number generator doesn't have to be set, but if many random numbers have to be generated (for example in a loop) the seed has to be set.

This can be done by assigning the local time to the random number generator:

```
cv::theRNG() = cv::RNG(time(0))
```

5.1 Uniform Distribution

Uniform Distributions can be generated in OpenCV with the help of *cv::randu*. The signature of *cv::randu* is:

```
void randu(Mat& mtx, const Scalar& low, const Scalar& high);
```

Where

- **mtx** is the Matrix to be filled with uniform distributed random numbers
- **low** is the inclusive lower boundary of generated random numbers
- **high** is the exclusive upper boundary of generated random numbers

5.2 Normal Distribution

Normal Distribution can be generated in OpenCv with the help of *cv::randn*.

```
void randn(Mat& mtx, const Scalar& mean, const Scalar& stddev);
```

Where

- **mtx** is the Matrix to be filled with normal distributed random numbers
- **mean** the mean value of the generated random numbers
- **stddev** the standard deviation of the generated random numbers

5.3 Bivariate Gaussian Distribution

Multivariate normal distributions can be generated with the C API by using *cvRandMVNormal* from the Machine Learning library. However this function seems to have no C++ equivalent yet, so in order to generate random numbers for the bivariate case one can also use the function given in 7.

Listing 7: bivariate_gaussian dstribution

```
void bivariate_gaussian(float mu_x, float sigma_x, float mu_y, float sigma_y, float rho, cv::Mat &x, cv::Mat &y) {
  assert(x.rows == y.rows && x.rows > 0);

  int n = x.rows;

  randn(x, mu_x, sigma_x);

  cv::Mat r(n,1, CV_32FC1);
  cv::theRNG() = cv::RNG(time(0));
```

```
  randn(r, 0, 1);
  for(int i = 0; i < n; i++) {
    y.at<float>(i, 0) = sqrt(sigma_y*sigma_y - (rho * sigma_y)*(rho * sigma_y)) * r.at<float>(i, 0) + mu_y;
    y.at<float>(i, 0) = y.at<float>(i,0) + rho * sigma_y/sigma_x * (x.at<float>(i,0) - mu_x);
}
```

6 Preparing the Training and Test Dataset for OpenCV ML

In the C++ Machine Learning API of OpenCV training and test data is given as a cv::Mat matrix. The constructor of cv::Mat is defined as:

```
Mat::Mat(int rows, int cols, int type);
```

Where

- rows is the number of samples (for all classes!)
- columns is the number of dimensions
- type is the image type

In the machine learning library of OpenCV each row or column in the training data is a n-dimensional sample. The default ordering is row sampling and class labels are given in a matrix with equal length (one column only, of course).

```
cv::Mat trainingData(numTrainingPoints, 2, CV_32FC1);
cv::Mat testData(numTestPoints, 2, CV_32FC1);

cv::randu(trainingData,0,1);
cv::randu(testData,0,1);

cv::Mat trainingClasses = labelData(trainingData, equation);
cv::Mat testClasses = labelData(testData, equation);
```

Since only binary classification problems are considered the function f returns the classes -1 and 1 for a given two-dimensional data point:

```
// function to learn
int f(float x, float y, int equation) {
  switch(equation) {
  case 0:
    return y > sin(x*10) ? -1 : 1;
    break;
  case 1:
    return y > cos(x * 10) ? -1 : 1;
    break;
  case 2:
    return y > 2*x ? -1 : 1;
    break;
  case 3:
    return y > tan(x*10) ? -1 : 1;
    break;
  default:
    return y > cos(x*10) ? -1 : 1;
  }
}
```

And to label data one can use the function labelData:

```
cv::Mat labelData(cv::Mat points, int equation) {
  cv::Mat labels(points.rows, 1, CV_32FC1);
  for(int i = 0; i < points.rows; i++) {
    float x = points.at<float>(i,0);
    float y = points.at<float>(i,1);
    labels.at<float>(i, 0) = f(x, y, equation);
  }
  return labels;
}
```

7 Support Vector Machines (SVM)

Support Vector Machines were first introduced by Vapnik and Chervonenkis in their paper *"Theory of Pattern Recognition"* [VC74]. The core idea is to find the optimal hyperplane to seperate a dataset, while there are theoretically infinite hyperplanes to seperate the dataset. A hyperplane is chosen, so that the distance to the nearest datapoint of both classes is maximized (Listing 2). The points spanning the hyperplane are the *Support Vectors*, hence the name *Support Vector Machines*.[CV95]

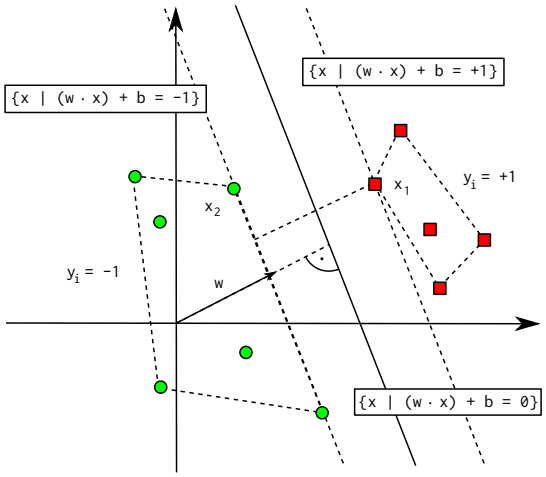


Figure 2: Maxmimum Margin Classifier.

7.1 Definition

Given a Set of Datapoints $\mathcal{D}$:

$$\mathcal{D} = \{(x_i, y_i) | x_i \in \mathbb{R}^p, y_i \in \{-1, 1\}\}_{i=1}^n$$

where

- x_i is a point in p-dimensional vector
- y_i is the corresponding class label

We search for $\omega \in \mathbb{R}^n$ and bias b, forming the Hyperplane H:

$$\omega^T x + b = 0$$

that seperates both classes so that:

$$\omega^T x + b = 1, \text{if } y = 1$$
$$\omega^T x + b = -1, \text{if } y = -1$$

The distance from a point to the Hyperplane is then given as:

$$r = \frac{\omega^T + b}{\|\omega\|}$$

This leads to the optimization problem:

$$\min \frac{1}{2}\omega^T \omega$$

subject to:

$$\omega^T x + b \geq 1, y = 1$$
$$\omega^T x + b \leq 1, y = -1$$

Such quadratic optimization problems can be solved with standard solvers, such as GNU Octave or Matlab.

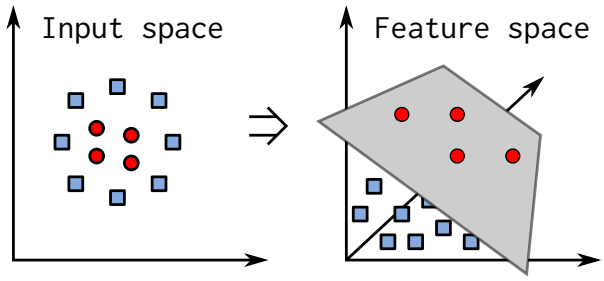


Figure 3: Kernel Trick

7.1.1 Non-linear SVM

The kernel trick is used for classifying non-linear datasets. It works by transforming data points into a higher dimensional feature space with a *kernel function*, where the dataset can be seperated again (see Figure 3).

Commonly used kernel functions are *RBF kernels*:

$$k(x, x') = exp(\frac{\|x - x'\|^2}{\sigma^2})$$

or *polynomial kernels*:

$$k(x, x') = (x \cdot x')^d$$

.

7.2 SVM in OpenCV

Parameters for a SVM have to be defined in the structure *CvSVMParams*.

Parameters

Listing 8: Example CvSVMParams

```
CvSVMParams param = CvSVMParams();

param.svm_type = CvSVM::C_SVC;
param.kernel_type = CvSVM::LINEAR;

param.degree = 0; // for poly
param.gamma = 20; // for poly/rbf/sigmoid
param.coef0 = 0; // for poly/sigmoid

param.C = 7; // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVR
param.nu = 0.0; // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_NU_SVR
param.p = 0.0; // for CV_SVM_EPS_SVR

param.class_weights = NULL; // for CV_SVM_C_SVC
param.term_crit.type = CV_TERMCRIT_ITER | CV_TERMCRIT_EPS;
param.term_crit.max_iter = 1000;
param.term_crit.epsilon = 1e-6;
```

Where the parameters are (taken from the OpenCV documentation):

- svm_type
  - CvSVM::C_SVC n-class classification ($n \geq 2$), allows imperfect separation of classes with penalty multiplier C for outliers.
  - CvSVM::NU_SVC n-class classification with possible imperfect separation. Parameter nu (in the range $0 \ldots 1$, the larger the value, the smoother the decision boundary) is used instead of C.
  - CvSVM::ONE_CLASS one-class SVM. All the training data are from the same class, SVM builds a boundary that separates the class from the rest of the feature space.
  - CvSVM::EPS_SVR regression. The distance between feature vectors from the training set and the fitting hyper-plane must be less than p. For outliers the penalty multiplier C is used.
  - CvSVM::NU_SVR regression; nu is used instead of p.

- kernel_type
  - CvSVM::LINEAR no mapping is done, linear discrimination (or regression) is done in the original feature space. It is the fastest option. $d(x, y) = x \cdot y == (x, y)$.
  - CvSVM::POLY polynomial kernel: $d(x, y) = (gamma * (x \cdot y) + coef0)^{degree}$.
  - CvSVM::RBF radial-basis-function kernel; a good choice in most cases: $d(x, y) = exp(-gamma * |x - y|^2)$.
  - CvSVM::SIGMOID sigmoid function is used as a kernel: $d(x, y) = tanh(gamma * (x \cdot y) + coef0)$.
- C, nu, p Parameters in the generalized SVM optimization problem.
- class_weights Optional weights, assigned to particular classes. They are multiplied by C and thus affect the misclassification penalty for different classes. The larger weight, the larger penalty on misclassification of data from the corresponding class.
- term_criteria Termination procedure for iterative SVM training procedure (which solves a partial case of constrained quadratic optimization problem)
  - type is either CV_TERMCRIT_ITER or CV_TERMCRIT_ITER
  - max_iter is the maximum number of iterations in training.
  - epsilon is the error to stop training.

Training

Training can either be done by passing the vector with the training data and vector with the corresponding class labels to the constructor or the train method.

```
CvSVM(const CvMat* _train_data,
      const CvMat* _responses,
      const CvMat* _var_idx=0,
      const CvMat* _sample_idx=0,
      CvSVMParams _params=CvSVMParams());
```

where

- **_train_data** is a Matrix with the n-dimensional feature vectors
- **_responses** is a vector with the class for the corresponding feature vector
- **_var_idx** identifies features of interest (can be left empty for this example, in code: `cv::Mat()`)
- **_sample_idx** identifies samples of interest (can be left empty for this example, in code: `cv::Mat()`)
- **_params** Parameter for the SVM from Listing 8

This applies to the train method aswell:

```
virtual bool train(const CvMat* _train_data,
  const CvMat* _responses,
  const CvMat* _var_idx=0,
  const CvMat* _sample_idx=0,
  CvSVMParams _params=CvSVMParams() );
```

The *train* methods of the SVM has some limitations:

- Only `CV_ROW_SAMPLE` is supported
- Missing measurements are not supported

The *train_auto* method finds the best parameters with a Gridsearch and a k-fold cross validation. This method is only available for OpenCV Versions ≥ 2.0.

Prediction

Self explaining code.

```
for(int i = 0; i < testData.rows; i++) {
      cv::Mat sample = testData.row(i);
      float result = svm.predict(sample);
}
```

Support Vectors

The support vectors of a SVM can be obtained using the get_support_vector function of the API:

```
int svec_count = svm.get_support_vector_count();
for(int vecNum = 0; vecNum < svec_count; vecNum++) {
        const float* vec = svm.get_support_vector(vecNum);
}
```

A complete example for Support Vector Machines in OpenCV is given in the Appendix.

8 Multi Layer Perceptron

An Artificial Neural Network is a biological inspired computational model. *Inputs* multiplied by *weights* result in an *activation* and form the *output* of a network.

Research in Artificial Neural Networks (ANN) began in 1943, when McCulloch and Pitts gave a definition of a formal neuron in their paper *"A Logical Calculus of the Ideas Immanent in Nervous Activity"* [MP43]. In 1958 Rosenblatt invented the perceptron, which is a simple feedforward neural network. The downfall of the perceptron algorithm is that it only converges on lineary seperable datasets and is not able to solve non-lineary problems such as the XOR problem. This was proven by Minsky and Papert in their monograph *"Perceptrons"*, but they showed that a two-layer feedforward architecture can overcome this limitation. It was until 1986 when Rumelhart, Hinton and Williams presented a learning rule for Aritificial Neural Networks with hidden units in their paper *"Learning Internal Representations by Error Propagation"*. The original discovery of backpropagation is actually credited to Werbos who described the algorithm in his 1974 Ph.D. thesis at Havard University, see [Wer94] for the roots of backpropagation.

A detailed introduction to Pattern Recognition with Neural Networks is given by [Bis95].

8.1 Backpropagation

1. Initilaize weights with random values
2. Present the input vector to the network
3. Evaluate the output of the network after a forward propagation of the signal
4. Calculate $\delta_j = (y_j - d_j)$ where d_j is the target output of neuron j and y_j is the actual output $y_j = g(\sum_i w_{ij} x_i) = (1 + e^{-\sum_i w_{ij} x_i})^{-1}$, (when the activation function is of a sigmoid type).
5. For all other neurons (from the first to the last layer) calculate $\delta_j = \sum_k w_{jk} g'(x) \delta_k$, where δ_k is δ_j of the succeeding layer and $g'(x) = y_k(1 - y_k)$
6. Update weights with $w_{ij}(t+1) = w_{ij}(t) - \eta y_i y_j (1 - y_j) \delta_j$, where η is the learning rate.
7. Termination Criteria. Goto Step 2 for a fixed number of iterations or an error.

The network error is defined as:

$$E = \frac{1}{2} \sum_{j=1}^{m} (d_j - y_j)^2$$

8.2 MLP in OpenCV

A Multilayer Perceptron in OpenCV is an instance of CvANN_MLP.

```
CvANN_MLP mlp;
```

Parameters

Parameters

```
CvANN_MLP_TrainParams params;
CvTermCriteria criteria;
criteria.max_iter = 100;
criteria.epsilon = 0.00001f;
criteria.type = CV_TERMCRIT_ITER | CV_TERMCRIT_EPS;
params.train_method = CvANN_MLP_TrainParams::BACKPROP;
params.bp_dw_scale = 0.05f;
params.bp_moment_scale = 0.05f;
params.term_crit = criteria;
```

Layers

The number of neurons per layer is stored in a row-ordered cv::Mat.

```
cv::Mat layers = cv::Mat(4, 1, CV_32SC1);

layers.row(0) = cv::Scalar(2);
layers.row(1) = cv::Scalar(10);
layers.row(2) = cv::Scalar(15);
layers.row(3) = cv::Scalar(1);

mlp.create(layers);
```

Training

The API for the MLP training is similiar to the SVM training.

```
mlp.train(trainingData, trainingClasses, cv::Mat(), cv::Mat(), params);
```

Prediction

The API for the prediction is different from the SVM API. Activations of the output layer are stored in the cv::Mat response. Since there is only one neuron in the output layer for a binary classification problem, it is the only activation to check.

```
mlp.predict(sample, response);
float result = response.at<float>(0,0);
```

9 Evaluation

To evaluate a predictor it is possible to calculate the True Positive Rate. For two classes it is:

$$TPR = \frac{TP}{TP + FP}$$

The performance of Support Vector Machines and especially Neural Networks depend on the parameters chosen. In case of the Neural Network it is difficult to find the appropriate parameters and designing an Artifical Neural Network is often more or less a rule of thumb. Parameters for a Support Vector Machine can be estimated using Cross Validation and Grid Search (both can be used as train_auto in OpenCV $\geq$ 2.0).

9.1 Training- and Testdata

We assume the function to be learned is a sine:

$$y = sin(10x)$$

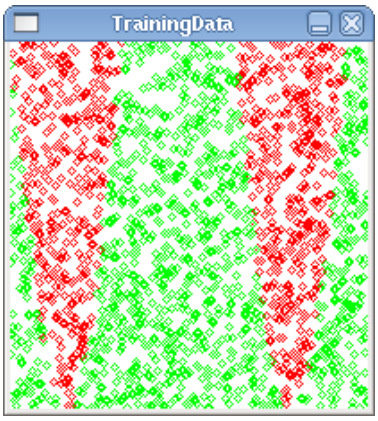
The size of the Trainingset is set to:

- 2000 Datapoints

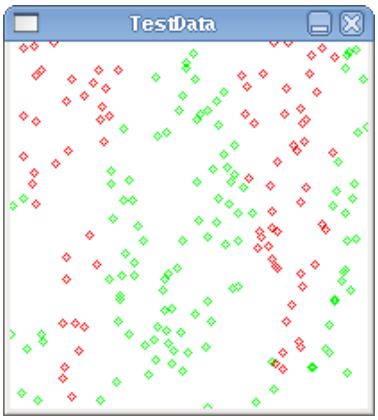
The Size of the Testset is:

- 200 Datapoints

Trainingdata



Testdata



9.2 SVM

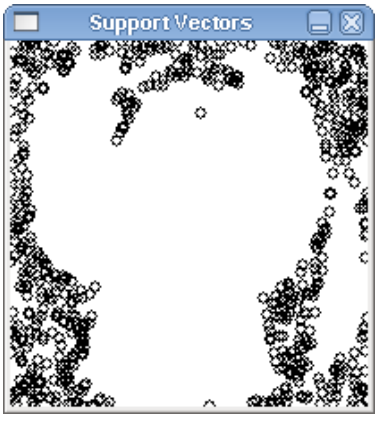
The classic SVM can optimally decide linear seperable data. If non-linear data is to be classified a Kernel trick is applied.

9.2.1 Linear SVM

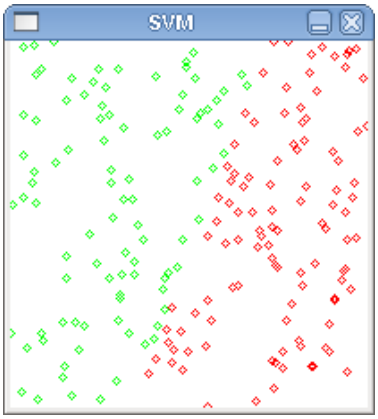
The linear SVM is chosen by setting the parameter to param.kernel_type = CvSVM::LINEAR;. But the predictor has a poor performance on the non-linear dataset:

$$TPR_{SVMLinear} = 0.515$$

Support Vectors



Test Data Classification

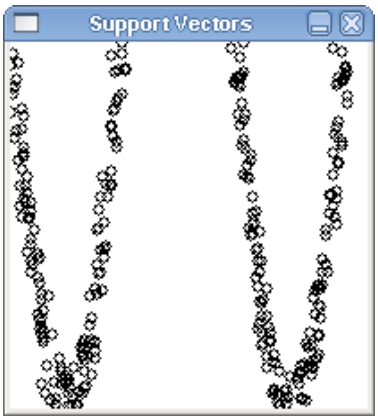


9.2.2 RBF SVM

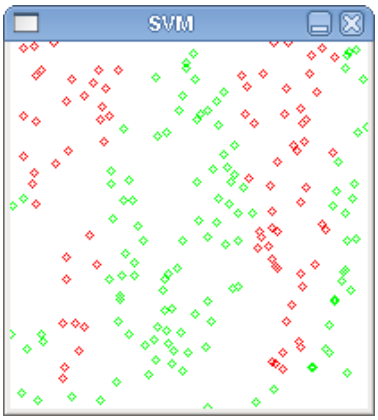
The Radial Basis Functions (RBF) is a commonly chosen function for transforming low-dimensional data into a higher dimension, where it can be seperated. The parameter is: param.kernel_type = CvSVM::RBF;. By transforming the data it is linearly seperable in a higher dimension and yields a good performance:

$$TPR_{SVMRBF} = 0.99$$

Support Vectors



Test Data Classification



9.3 MLP

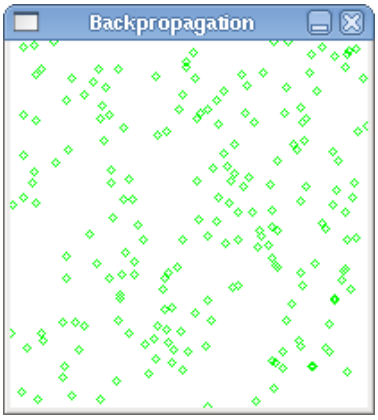
The performance of the MLP depends on the architecture of the network (and the learning rule). The title of the next section indicates the layers in the net architecture, e.g. $2-5-5-1$ is a network with 2 input neurons, two hidden layers with 5 neurons and 1 output neuron. Since it is only a binary classification problem, there is only 1 output neuron and its activation indicates the predicted class.

9.3.1 MLP 2-5-5-1

Two hidden layers with only 5 neurons are not an adequate structure for the given data. The TPR is only slightly above 50%, which is close to decide randomly:

$$TPR_{MLP2-5-5-1} = 0.585$$

Test Data Classification

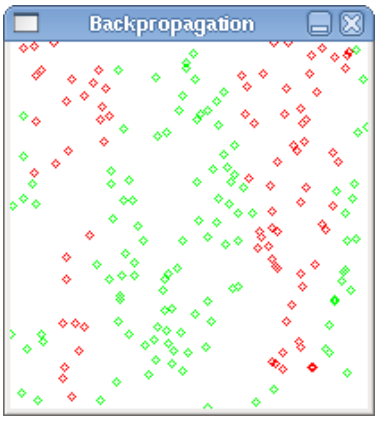


9.3.2 MLP 2-10-15-1

By using a better suited network architecture the classification rate progresses to:

$$TPR_{MLP2-10-15-1} = 0.93$$

Test Data Classification



A Sourcecode

A.1 cmake.example

Listing 9: Introdcution Code to CMake

```
#include <iostream>
#include "geometry.h"
```

```
using namespace std;

struct point {
        double latitude;
        double longitude;
};

int main() {
        // Berlin: 52.614723,13.373108
        // Munich: 48.189895,11.560364
        struct point p1 = {52.62, 13.37};
        struct point p2 = {48.19, 11.56};

        double d = geometry::great_circle_distance(p1.latitude, p1.longitude, p2.latitude, p2.longitude);

        cout << "Distance is " << d << "." << endl;

        return 0;
}
```

B main.cpp

Listing 10: Complete Sourcecode example for SVM and MLP

```
#include <iostream>
#include <math.h>
#include <string>
#include "cv.h"
#include "ml.h"
#include "highgui.h"

using namespace cv;
using namespace std;

#define numTrainingPoints 200
#define numTestPoints 200
#define size 200
#define eq 2

// tpr
float evaluate(cv::Mat& predicted, cv::Mat& actual) {
        assert(predicted.rows == actual.rows);
        int t = 0;
        int f = 0;
        for(int i = 0; i < actual.rows; i++) {
                float p = predicted.at<float>(i,0);
                float a = actual.at<float>(i,0);
                if((p >= 0.0 && a >= 0.0) || (p <= 0.0 && a <= 0.0)) {
                        t++;
} else {
                        f++;
}
}
        return (t * 1.0) / (t + f);
}

// plot data and class
void plot_binary(cv::Mat& data, cv::Mat& classes, string name) {
        cv::Mat plot(size, size, CV_8UC3);
        plot.setTo(CV_RGB(255,255,255));
        for(int i = 0; i < data.rows; i++) {

                float x = data.at<float>(i,0) * size;
                float y = data.at<float>(i,1) * size;

                if(classes.at<float>(i, 0) > 0) {
                        cv::circle(plot, Point(x,y), 2, CV_RGB(255,0,0),1);
} else {
                        cv::circle(plot, Point(x,y), 2, CV_RGB(0,255,0),1);
}
}
        cv::imshow(name, plot);
}

// function to learn
int f(float x, float y, int equation) {
        switch(equation) {
        case 0:
                return y > sin(x*10) ? -1 : 1;
                break;
        case 1:
                return y > cos(x * 10) ? -1 : 1;
                break;
        case 2:
                return y > 2*x ? -1 : 1;
                break;
        case 3:
                return y > tan(x*10) ? -1 : 1;
                break;
        default:
                return y > cos(x*10) ? -1 : 1;
}
}

// label data with equation
cv::Mat labelData(cv::Mat points, int equation) {
        cv::Mat labels(points.rows, 1, CV_32FC1);
        for(int i = 0; i < points.rows; i++) {
                        float x = points.at<float>(i,0);
                        float y = points.at<float>(i,1);
                        labels.at<float>(i, 0) = f(x, y, equation);
}
        return labels;
}
```

```
void svm(cv::Mat& trainingData, cv::Mat& trainingClasses, cv::Mat& testData, cv::Mat& testClasses) {
        CvSVMParams param = CvSVMParams();

        param.svm_type = CvSVM::C_SVC;
        param.kernel_type = CvSVM::RBF; //CvSVM::RBF, CvSVM::LINEAR ...
        param.degree = 0; // for poly
        param.gamma = 20; // for poly/rbf/sigmoid
        param.coef0 = 0; // for poly/sigmoid

        param.C = 7; // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVR
        param.nu = 0.0; // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_NU_SVR
        param.p = 0.0; // for CV_SVM_EPS_SVR

        param.class_weights = NULL; // for CV_SVM_C_SVC
        param.term_crit.type = CV_TERMCRIT_ITER +CV_TERMCRIT_EPS;
        param.term_crit.max_iter = 1000;
        param.term_crit.epsilon = 1e-6;

        // SVM training (use train auto for OpenCV>=2.0)
        CvSVM svm(trainingData, trainingClasses, cv::Mat(), cv::Mat(), param);

        cv::Mat predicted(testClasses.rows, 1, CV_32F);

        for(int i = 0; i < testData.rows; i++) {
                cv::Mat sample = testData.row(i);

                float x = sample.at<float>(0,0);
                float y = sample.at<float>(0,1);

                predicted.at<float>(i, 0) = svm.predict(sample);
}

        cout << "TPR_{SVM} = " << evaluate(predicted, testClasses) << endl;
        plot_binary(testData, predicted, "Predictions_SVM");

        // plot support vectors
        cv::Mat plot_sv(size, size, CV_8UC3);
        plot_sv.setTo(CV_RGB(255,255,255));

        int svec_count = svm.get_support_vector_count();
        for(int vecNum = 0; vecNum < svec_count; vecNum++) {
                const float* vec = svm.get_support_vector(vecNum);
                cv::circle(plot_sv, Point(vec[0]*size, vec[1]*size), 3 , CV_RGB(0, 0, 0));
}
        cv::imshow("Support_Vectors", plot_sv);

}

void mlp(cv::Mat& trainingData, cv::Mat& trainingClasses, cv::Mat& testData, cv::Mat& testClasses) {

        cv::Mat layers = cv::Mat(4, 1, CV_32SC1);

        layers.row(0) = cv::Scalar(2);
        layers.row(1) = cv::Scalar(10);
        layers.row(2) = cv::Scalar(15);
        layers.row(3) = cv::Scalar(1);

        CvANN_MLP mlp;
        CvANN_MLP_TrainParams params;
        CvTermCriteria criteria;
        criteria.max_iter = 100;
        criteria.epsilon = 0.00001f;
        criteria.type = CV_TERMCRIT_ITER | CV_TERMCRIT_EPS;
        params.train_method = CvANN_MLP_TrainParams::BACKPROP;
        params.bp_dw_scale = 0.05f;
        params.bp_moment_scale = 0.05f;
        params.term_crit = criteria;

        mlp.create(layers);

        // train
        mlp.train(trainingData, trainingClasses, cv::Mat(), cv::Mat(), params);

        cv::Mat response(1, 1, CV_32FC1);
        cv::Mat predicted(testClasses.rows, 1, CV_32F);
        for(int i = 0; i < testData.rows; i++) {
                cv::Mat response(1, 1, CV_32FC1);
                cv::Mat sample = testData.row(i);

                mlp.predict(sample, response);
                predicted.at<float>(i,0) = response.at<float>(0,0);

}

        cout << "TPR_{MLP} = " << evaluate(predicted, testClasses) << endl;
        plot_binary(testData, predicted, "Predictions_Backpropagation");
}

int main() {

        cv::Mat trainingData(numTrainingPoints, 2, CV_32FC1);
        cv::Mat testData(numTestPoints, 2, CV_32FC1);

        cv::randu(trainingData,0,1);
        cv::randu(testData,0,1);

        cv::Mat trainingClasses = labelData(trainingData, eq);
        cv::Mat testClasses = labelData(testData, eq);

        plot_binary(trainingData, trainingClasses, "Training_Data");
        plot_binary(testData, testClasses, "Test_Data");

        svm(trainingData, trainingClasses, testData, testClasses);
        mlp(trainingData, trainingClasses, testData, testClasses);

        cv::waitKey();

        return 0;
}
```

References

[Bis95] Bishop, C. M.: *Neural Networks for Pattern Recognition*. Oxford : Oxford University Press, 1995

[CV95] Cortes, Corinna ; Vapnik, Vladimir: Support-Vector Networks. In: *Machine Learning* Bd. 20, 1995, S. 273–297

[MP43] McCulloch, W. ; Pitts, W.: A logical calculus of the ideas immanent in nervous activity. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 115–133

[VC74] Vapnik, V. N. ; Chervonenkis, A. Y.: *Theory of Pattern Recognition [in Russian]*. USSR : Nauka, 1974

[Wer94] Werbos, P.J.: *The Roots of Backpropagation: From ordered derivatives to Neural Networks and Political Forecasting*. New York : John Wiley and Sons, 1994