PLS NIPALS C++

version 0.0.1

William Robson Schwartz
http://www.liv.ic.unicamp.br/~wschwartz/softwares.html

June 26, 2011

Contents

Chapter 1

Introduction

This library provides a C++ class to execute Partial Least Squares (PLS) NIPALS method for a scalar response variable for both dimension reduction or regression [1]. It provides a class composed of methods to build, load, and store a PLS model, project feature vectors onto the PLS model and retrieve its low dimensional representation.

The implementation of the NIPALS algorithm provided in this library is a translation from the MATLAB version of the NIPALS algorithm written by Dr. Hervé Abdi from The University of Texas at Dallas (http://www.utdallas.edu/~herve). This code requires OpenCV version 1.0 or superior (http://opencv.willowgarage.com/wiki/).

This code works either on Windows or on Linux. For Windows, a project for Visual Studio 2005 is provided. A Makefile can be used to compile all files and generate an executable **main** containing examples of usage. To incorporate this library in your project, copy every .cpp and .h file to your directory and compile them with your code. Then call the methods provided by the class Model.

PLS handles data in high dimensional feature spaces and can be employed as a dimensionality reduction technique. PLS is a powerful technique that provides dimensionality reduction for even hundreds of thousands of variables, considering the response variable in the process. The latter point is in contrast to traditional dimensionality reduction techniques such as Principal Component Analysis (PCA).

The difference between PLS and PCA is that the former creates orthogonal weight vectors by maximizing the covariance between set of variables: independent and response variables. Thus, PLS not only considers the variance of the samples but also considers the class labels in a classification task, for example. Fisher Discriminant Analysis (FDA) is, in this way, similar to PLS. However, FDA has the limitation that after dimensionality reduction, there are only c-1 meaningful latent variables, where c is the number of classes being considered.

Figure 1.1 shows projections of the feature vectors for the task of human detection onto the first two dimensions of the spaces estimated by PCA and PLS. PLS

clearly achieves better class separation than PCA.

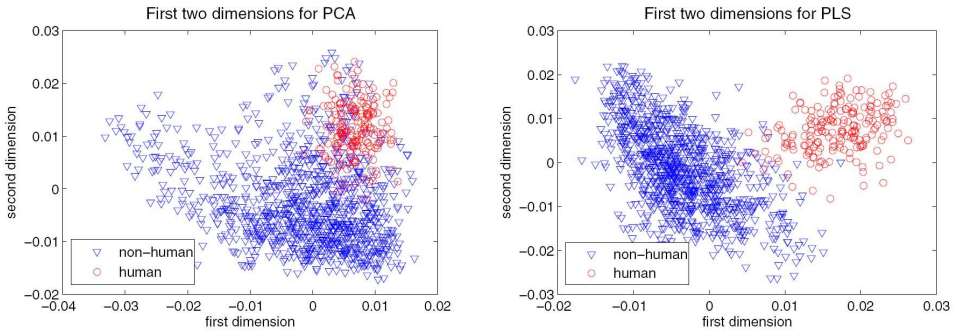


Figure 1.1: Comparison of PCA and PLS for dimensionality reduction. Projection of the first two dimensions of the training samples for one of the models learned in the cross-validation (figure extracted from [2]).

If you find bugs or problems in this software or you have suggestions to improve or make it more user friendly, please send an e-mail to williamrobschwartz@gmail.com.

This implementation has been used as part of the human detector approach developed by Schwartz et al. [2]. We kindly ask you to cite that reference upon the use of this code with the following bibtex entry.

```
@inproceedings{schwartz:ICCV:2009,
  author = {W. R. Schwartz and A. Kembhavi and D. Harwood and L.S. Davis},
  booktitle = {International Conference on Computer Vision},
  title = {{Human Detection Using Partial Least Squares Analysis}},
  pages = {24-31},
  year = {2009},
}
```

Chapter 2

Code Interface

2.1 Main Class

This library implements a C++ class called *Model* that provides a set of methods to build, save, and load PLS models as well as project either feature vector or matrix. Listing 2.1 displays available methods for this class.

Listing 2.1: class Model

```
1  class Model {
2      // constructor: either creates an empty model or load a stored PLS model
3      Model();
4      Model(string filename);
5
6      // build PLS model when response variable has only two values (for classification)
7      void CreatePLSModel(Matrix<float> *mPos, Matrix<float> *mNeg, int nfactors);
8
9      // build PLS model for any response variable
10     void CreatePLSModel(Matrix<float> *X, Vector<float> *Y, int nfactors);
11
12     // save model to a file
13     void SaveModel(string filename);
14
15     // project feature vector using this model
16     Vector<float> *ProjectFeatureVector(Vector<float> *feat);
17
18     // project feature matrix
19     Matrix<float> *ProjectFeatureMatrix(Matrix<float> *featMat);
20
21     // get number of features
22     int GetNumberFeatures();
23 };
```

Model The constructor *Model* either initialize an empty PLS model or load a previously stored model specified by string filename.

CreatePLSModel Method *CreatePLSModel* builds a PLS model for a number of factors (dimensionality of the low dimensional space, please refer to [2] for more details). There are two ways of building a model:

1. passing matrices, with feature vectors, representing the negative (mNeg) and the positive classes (mPos). **Note**: when this method is used, negative class presents label -1 and positive class $+1$.

2. passing a matrix with the feature vectors (X) and a vector (Y) with the class labels or response values, in case of regression.

SaveModel Method *SaveModel* stores the built PLS model in a yml file with name defined by the string filename.

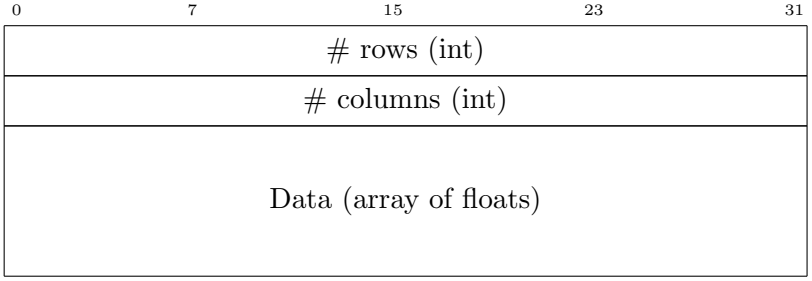
ProjectFeatureVector Method *ProjectFeatureVector* projects a feature vector onto the built PLS model and returns the latent variables in the low dimensional space.

ProjectFeatureMatrix Method *ProjectFeatureMatrix* projects multiple feature vectors, stored in the matrix featMat row-by-row, onto the built PLS model and returns a matrix in which the latent variables in the low dimensional space and also stored row-by-row.

GetNumberFeatures Method *GetNumberFeatures* retrieves the number of feature descriptors considered by the PLS model.

2.2 Auxiliary Classes

This library also implements some auxiliary classes: *Vector* and *Matrix*. Matrices and vectors can be either loaded or saved in files with format *feat*. This format only supports 32-bits float data type and is defined as follows (matrices are stored column-wise). To facilitate the use, two MATLAB functions (load_matrix.m and write_matrix.m) are provided to read and write matrices and vectors in format *feat*.



Listing 2.2: class Vector

```
template <class T>
class Vector {

    // create a vector with elements
    Vector(int n);

    // load a vector from a file
    Vector(string filename);

    // retrieve the element at position x
    T GetElement(int x);

    // set element value at position x
    void SetElement(int x, T value);

    // retrieve number of elements in the vector
    int GetNElements();

    // write vector to file
    void Write(string filename);

    // copy vector
    Vector<T> *Copy();
}
```

Vector Constructor *Vector* either creates a vector with n elements or loads a stored *feat* file defined by string filename.

GetElement Method *GetElement* access the element at position x. **Note**: the first element of the vector is at position 0.

SetElement Method *SetElement* attributes value to position x of the vector.

GetNElements Method *GetNElements* retrieves the number of elements contained in the vector.

Write Method *Write* saves the vector in a file defined by string filename in the format *feat*. **Note**: to use this function, the type of class Vector must be float.

Copy Method *Copy* duplicates vector and returns a pointer to the new vector.

Listing 2.3: class Matrix

```
template <class T>
class Vector {
```

```
4       // create a matrix with r rows and c columns
5       Matrix(int r, int c);
6
7       // load a matrix from a file
8       Matrix(string filename);
9
10      // retrieve the element at row y and column x
11      T GetElement(int x, int y);
12
13      // set element value at row y and column x
14      void SetValue(int x, int y, T value);
15
16      // get number of rows
17      int GetNRows();
18
19      // get number of columns
20      int GetNCols();
21
22      // contatenate rows of matrices m1 and m2, return a new matrix
23      Matrix<T> *ConcatenateMatricesRows(Matrix<T> *m1, Matrix<T> *m2);
24
25      // copy matrix
26      Matrix<T> *Copy();
27
28      // retrieve row i of the matrix, return a new vector
29      Vector<T> *GetRow(int i);
30
31      // set row i of the matrix with the vector InVector
32      void SetRow(Vector<T> *InVector, int r);
33
34      // write matrix to a file
35      void Write(string filename);
36  }
```

Matrix Constructor *Matrix* either loads a matrix with format *feat* from file defined by string filename or creates a matrix with r rows and c columns.

GetElement Method *GetElement* access matrix element at column x and row y. **Note**: the matrix indexation starts at position $(0, 0)$.

SetValue Method *SetValue* set value to element at column x and row y.

GetNRows Method *GetNRows* retrieves the number of rows in the matrix.

GetNCols Method *GetNCols* retrieves the number of columns in the matrix.

ConcatenateMatricesRows Method *ConcatenateMatricesRows* concatenates two matrices: m1 and m2, returning a third matrix with the same number of columns and m1.GetNRows() + m2.GetNRows() rows.

Copy Method *Copy* duplicates the matrix returning a pointer to the newly created matrix.

GetRow Method *GetRow* retrieves the row r of the matrix and return a pointer to the created vector.

SetRow Method *SetRow* sets the r row of the matrix with the vector InVector, which has the same number of elements as number of columns of the matrix.

Write Method *Write* saves the matrix to a file defined by string filename in format *feat*.

Chapter 3

Examples

Inside file main.cpp there are three examples to use the PLS NIPALS that are described as follows.

First Example The first example, shown in Listing 3.1, builds a PLS model where the response variable presents only two values. This setup is usually used in classification. Lines 8 and 9 load training samples for both classes (negative and positive), stored in .feat files. The PLS model is built in line 13, using 4 factors (the resulting sub-space presents dimensionality 4). Finally, line 16 saves the model in a yml file.

Listing 3.1: Example to build a PLS model for a binary response variable.

```
1  #include "model.h"
2
3  void Example1() {
4  Model *model;
5  Matrix<float> *mpos, *mneg;
6
7    // load matrices with training samples
8    mpos = new Matrix<float>("data/PosSamplesTraining.feat");
9    mneg = new Matrix<float>("data/NegSamplesTraining.feat");
10
11   // create PLS model with 4 factors
12   model = new Model();
13   model->CreatePLSModel(mpos, mneg, 4);
14
15   // save PLS model created
16   model->SaveModel("data/Model.yml");
17 }
```

Second Example The second example, shown in Listing 3.2, builds a PLS model where Y represents a scalar response variable. Lines 9-10 load the features and the response variables to matrix X and vector Y. Lines 13-14 build the PLS model with

4 factors (the resulting sub-space presents dimensionality 4). Then it is saved in file Model2.yml.

Listing 3.2: Example to build a PLS model for a scalar response variable.

```
1  #include "model.h"
2
3  void Example2() {
4  Model *model;
5  Matrix<float> *X;
6  Vector<float> *Y;
7
8    // load matrices with training samples
9    X = new Matrix<float>("data/X.feat");
10   Y = new Vector<float>("data/Y.feat");
11
12   // create PLS model with 4 factors
13   model = new Model();
14   model->CreatePLSModel(X, Y, 4);
15
16   // save PLS model created
17   model->SaveModel("data/Model2.yml");
18 }
```

Third Example The example shown in Listing 3.3, load a PLS model from a file (Line 8), and testing feature vectors (Lines 11-12) and extract their low dimensional representation. Lines 15-18 project samples from matrix mpos onto the PLS model and save their low dimensional representation in file LowDPosSamples.feat. Similar process is executed for matrix mneg in Lines 19-20.

Listing 3.3: Example to load a stored PLS model and project features.

```
1  #include "model.h"
2
3  void Example3() {
4  Model *model;
5  Matrix<float> *mlowD;
6
7    // load PLS model saved previously
8    model = new Model("data/Model.yml");
9
10   // load matrix with testing samples for each class
11   mpos = new Matrix<float>("data/PosSamplesTesting.feat");
12   mneg = new Matrix<float>("data/NegSamplesTesting.feat");
13
14   // project the positive feature vectors onto the PLS model
15   mlowD = model->ProjectFeatureMatrix(mpos);
16   mlowD->Write("data/LowDPosSamples.feat");
17
18   // project the negative feature vectors onto the PLS model
19   mlowD = model->ProjectFeatureMatrix(mneg);
```

```
20  mlowD->Write("data/LowDNegSamples.feat");
```

Bibliography

[1] H. Wold, "Partial least squares," in *Encyclopedia of Statistical Sciences*, S. Kotz and N. Johnson, Eds. New York: Wiley, 1985, vol. 6, pp. 581–591.

[2] W. R. Schwartz, A. Kembhavi, D. Harwood, and L. S. Davis, "Human detection using partial least squares analysis," in *IEEE International Conference on Computer Vision*, 2009.