Pattern Classification and Machine Learning

Benoit Rat[1], Jean-Christophe Fillion-Robin[2]
Communication Systems Faculty
EPFL Lausanne
CH-1015 Lausanne, Switzerland

***Abstract* — Using the third party library Torch specialized in solving data mining problems, we will compare the back propagation, the Support Vector Machine (SVM) and the Gaussian Mixture Model (GMM) classification methods.**

I. Introduction & motivation

The aim of our research is to apply different classification methods on a realistic database and discover what are the difficulties linked to the use of artificial neural networks. To help us solving such a problem, it exists a lot of third party library implementing all the basic tools of classification such as Weka [1] or Torch [2]. We chose to use the second one because, first of all, it's a C++-based tools and it's more efficient computationally wide. It was not the only reason since the size of the database used were small. An other reason concerned the possibility to get accurate help and fast response concerning our questions about the implementation of the different classification solutions. Indeed, Torch is developed by people from the IDIAP, a research institute in close relation with EPFL. This paper will present our results concerning the classification of the "Wisconsin Diagnostic Breast Cancer data base" using the back propagation method, the Support Vector Machine (SVM) method and the Gaussian Mixture Model method based on Expectation-Maximization algorithm (initialized with K-Means).

II. Database

For this work we have started using the segmentation images database. As it was really difficult to observe relevant overfitting for the back-propagation algorithm even using few samples in the training, we decided to use "Wisconsin Diagnostic Breast Cancer data base" (WDBC) which gives more interesting result for a "theoretical flavor" study.

The WDBC data base contains 569 samples. Each of them has an ID, a class type tagged with some medical expertise and 30 different features. Our goal is to know if the subject is malignant (M) or benign (B) using the 30 preprocessed features. We are not going to describe the meaning of all the 30 features since they are available in the references [3].

To perform the classification we separated our database in 3 sets: Training, Validation, Testing. We first processed each line to fit the *MatDatSet* structure: (removing ID field and comma, changing the class type to {0,1} and moving it to the end of the row). Then we took one third of the database to built the testing set which is not changed after. Finally, we designed and implemented a C++ program generating new validation and training set by randomly permuting the two third of samples that left. We also have been cautious to respect more or less the proportion of benign and malignant: $|B|/|M| \approx 1.7$. This random permutation is, in some extent, a sort of cross validation between training and testing when we use more than one iteration over this two sets randomly picked.

III. Back-Propagation

A. Description & Structure.

Back-Propagation in an algorithm which corresponds to a gradient descent on a multi-layer perceptron [4]. As we use the structure of an MLP, we still want to minimize the error function, which is in our case the mean square error (MSE) between the target and the output.

We first focus on the structure of the MLP we should use: This can be done by setting the number of hidden layers and the number of hidden neurones in each layers. In order to connect the different neurones from a layer to another we use a linear weighted function. Then the hidden neurones, and outputs neurones have a transfer function (step, linear, sigmoid or tanh). As we want our output to be smooth and continuous values between 0 and 1, we use the Sigmoid transfer function $g(h)$.

Moreover, the BackProp algorithm has a learning rate parameter which corresponds to the step of a gradient descent. Finally the criterion to stop the algorithm is given by the final MSE to reach (aka. *end_accuracy*) or in the worst case a maximum number of iterations.

B. Playing with the different parameters.

During the first part of our research on the MLP, we tried different parameters against each others. This is a difficult task to do due to their important numbers. We first set a very low *end_accuracy*: 10^{-6} and the maximum number of iterations to 10000. Then we try with coarse values for the number of hidden units and the learning rate in order to observe the influence of using 1,2 or 3 hidden layers. No significant improvements were observed whereas the computational time has been increased, so we decide to use only one hidden layer.

Afterwards, to obtain the overfitting plot we run our algorithm with different number of hidden neurones

Report written June 13th 2007.

1. e-mail: benoit.rat@epfl.ch
2. e-mail: jean-christophe.fillion-robin@epfl.ch

{1,5,10,15,25,35,50,100} and different learning rate {0.1,0.01,0.001,0.0001}. We first denote that a learning rate of 0.1 is too high because the learning curbs oscillates until converging to a high MSE value for both training and validation set. We also denote that taking a really small learning rate increases the convergence speed such as the presence of a high number of hidden neurones. Moreover the minimum value of the MSE on the validation is not better than when we use an higher learning rate. Therefore we decided to focus our experiments using the learning rates 0.01 and 0.001. We noticed that adding more hidden units give us better overfitting. Finally, after 20 experiments with randomly built set both training and validation, the best compromise between good overfitting curbs and lowest MSE is obtained using the a learning rate = 0.001 and a number of hidden neurones equal to 50.

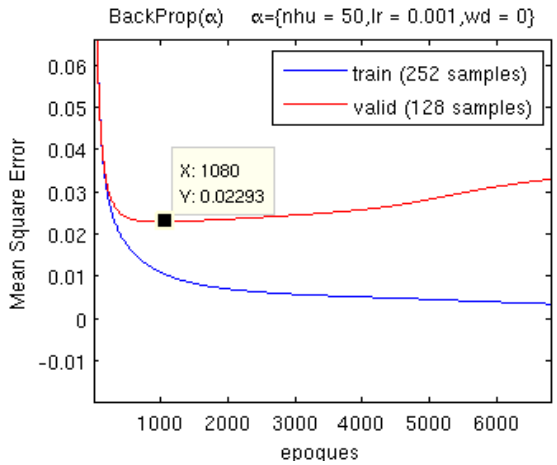
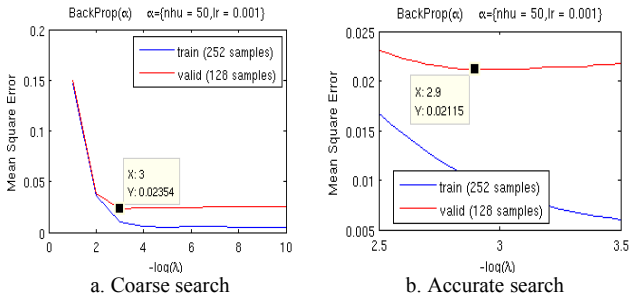


Figure 1: Overfitting.

C. Regularization using weight decay.

Once we have set these different parameters, we notice that, due to the overfitting, the final MSE for validation is higher than the MSE error at epoch 1080 (See Figure 1). In order to compensate overfitting we use a regularization method to control the flexibility of the network.

We test different values of the weight decay λ using a log step in order to find the optimal one. 20 different iterations with random initialization has been used to obtain smooth curbs. The search of the optimal value λ has been started using coarse step and has given its best result around 10^{-3}. Then using a fine step we obtain the optimal value $\lambda = 10^{-2.90} = 0.0013$.



a. Coarse search b. Accurate search

Figure 2: Regularization graph (optimal weight decay)

The final classification curve are showed in Figure 3 :

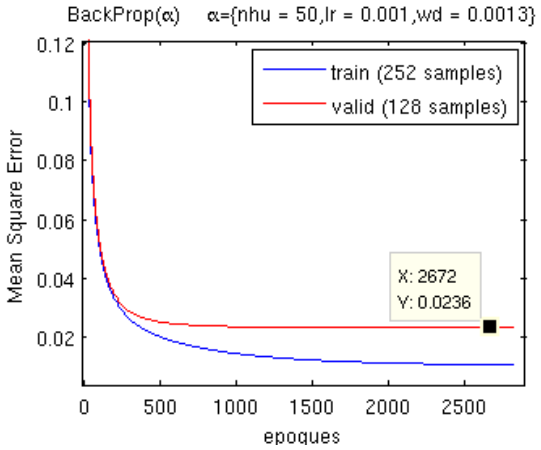


Figure 3: Final learning curves of Back-Propagation

	Training	Validation	Testing
MSE error	0.00552671	0.0236324	0.0238956
Classification error	0.3967%	2,3438%	2.6455%

Table 1: Final result of Back-Propagation

IV. Support Vector Machine

A. Description & Structure.

In order to deal with the fact that the perceptron is not linearly separable we introduce the Support Vector Machine (SVM). The SVM is an optimal perceptron with a kernel that project the input space into a linear separable space maximizing the security margin. The main difficulty in the SVM is to find the best kernel to perform this "linearization".[5]

In this work, we are asked to use a Gaussian kernel to ease the task of selecting the best structure of SVM. In this sense the standard deviation (stdv) of the Gaussian is the parameter to optimize. We could also look at the C parameter which deals with the security margin and the error, but at it is not asked we let the default value $C = 100$.

B. Regularization using standard deviation.

To build this regularization plot we have trained the SVM changing the value of the stdv from 1 to 200 with unit step. Doing this operation we denote that the SVM is really fast to learn (maybe due to the purpose of Torch when it was created).

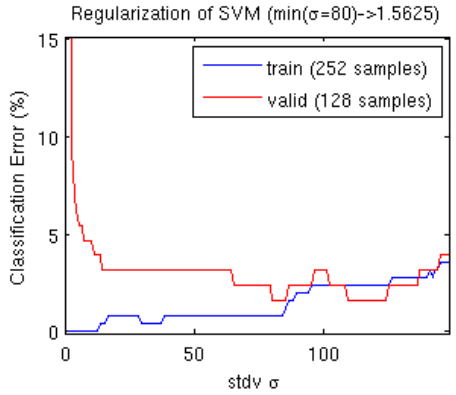


Figure 4: Regularization graph (optimal std deviation)

We finally obtain the regularization graph in Figure 4, and we take the standard deviation equal to $\sigma=80$.

We also notice, that has the set has a small size we have our curves with big step. To compensate this effect we have performed this operation on different random permutations of both training and validation sets. Applying this method we have noticed that the optimal σ highly depend on how the two set are distributed.

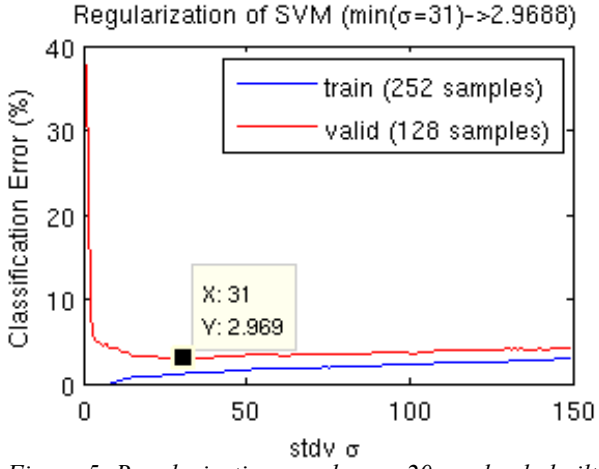


Figure 5: Regularization graph over 20 randomly built set

As Figure 5 shows, the optimal mean value of standard deviation other 20 iterations is $\sigma=31$.

	Training	Validation	Testing
Error with $\sigma=80$ (%)	0.7956	1,5625	3.1752
Error with $\sigma=31$ (%)	0.3968	2.4617	3.1658

Table 2: Classification error with different value of stdv.

The Table 2 shows that using $\sigma=31$ may be a better idea to perform better classification on unknown data because it is more general (as we use a sort of cross validation).

V. Gaussian Mixture Model

A.Description & Structure.

Since the distribution of the point in our database can not be easily represented by only one probability function, the use of Gaussian Mixture Model (GMM) which is a linear combination of gaussian probability distributions (also know as normal probability distributions) reveals to be an interesting option. The key problem is to determine the number M of probability distributions allowing to elegantly discriminate the inputs and match them to the different classes.

The parameter M fixed, there is an other important task to achieve. Recalling a gaussian probability distribution is managed by two important parameters, the mean μ and the variance σ^2 , the use of the K-Means algorithm [6] allows to discover the parameters for each one of the M gaussian distributions.

Technically, this last operation is done inside the Torch framework through the initialization of *DiagonalGMM object* using a *Kmeans object* preliminary trained to discover the initial gaussian parameters.

B.Expectation-Maximization algorithm

The GMM is trained using the Expectation-Maximization (EM) algorithm [7]. The concept behind the EM strategy is a 2-step mechanism iteratively repeated until convergence. The stopping criterion is defined by the *end_accuracy*. It means the process is stopped when the difference between the iteration n and the iteration $n+1$ is smaller than the *end_accuracy* value. After some trials, a value of 10^{-3} for *end_accuracy* seems to minimize the Negative Log-Likelihood (NLL) the best. The value gathered at each EM iteration can be described as the NLL.

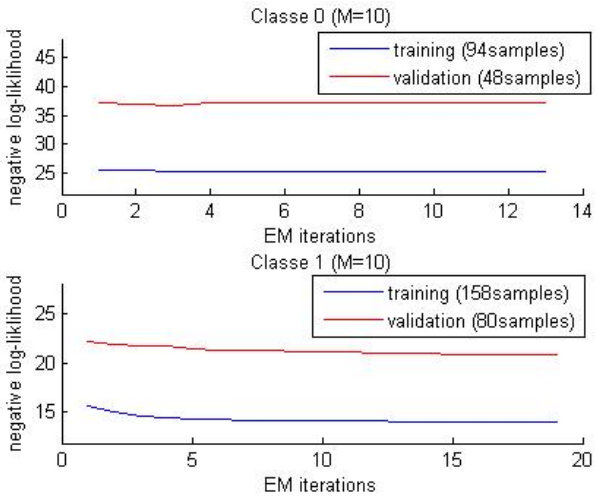


Figure 6: Negative log-likelihood = function(EM iteration) for M = 10 gaussians

Looking at Figure 6, we can not observe any significant overfitting for any of the classes.

C.Regularization using the number of gaussian.

Next, we wanted to understand how the number of gaussians influences the classification error. Before going further in our analysis, let's recall how are generated the different model. First, each dataset (training, validation and testing sets) are split by classes. Then, using the training data, the GMM model is initialized with the K-Means algorithm and is trained (with an *EMTrainer object*) on each classes independently. Since the database used has 2 classes (C_0 and C_1), we obtained two models gmm^0 and gmm^1 for each dataset.

Later on, we wrote a *classification error function* able to compute the number of misclassified *examples*. An *example* (or sample) being a set of X inputs. The core of this function is based on the simple comparison of the probability returned by each of the models for a given *example*. For each example, we obtain the probabilities of being in C_0 and in C_1 , if the *example* should be in C_0 and the gmm^1 returns the highest probability, it's a misclassified one. We iterate this comparison for all the examples and we obtain the percentage of classification error. Applying this method for each of the set, we can visualize the efficiency of the GMM model compared to the number of gaussians used (see Figure 7).

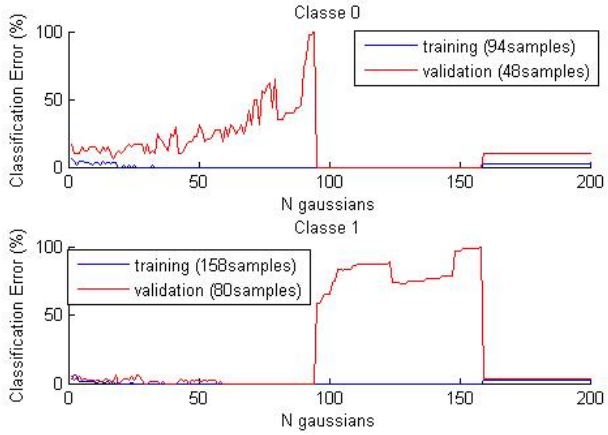


Figure 7: Regularization graph for GMM

Analyzing Figure 7, we can extract the number of gaussians minimizing the classification error for both classes C_0 and C_1 . Setting the parameter $n_{gaussians}$ to 17 seems to be a good compromise (see Table 3). We can also notice that when the number M of gaussians is closer to the number of examples, the classification of the validation set increase drastically.

	Training	Validation	Testing
Classification error C_0 (%)	2.1276	6.2500	8.5714
Classification error C_1 (%)	0.6329	1.2500	3.3613
Avg. Classification error (%)	1.3803	3.7500	5.9664

Table 3: Classification error of C_0 and C_1 for M = 17 gaussians

VI. Discussion

The 3 methods applied to build an efficient classifier give an average classification error smaller then 10%. To obtain acceptable results, a variable number of parameters need to be tuned.

The back propagation method required greater attention regarding the number of parameters (number of hidden layer and hidden neurones, learning rate, weight decay) as the complexity of the model. After the model is settled and well-tuned, looking at Figure 8, we can observe the back propagation method presents the best performance in term of minimization of the classification error. In same way, concerning the classification error on the testing set, the SVM method presents decent results diverging only from half a unit. Moreover we have seen that this method is easy and fast to tune. The GMM method, compared to the two others, presents the worst results. A advantages of the GMM could be the low number of iterations to obtain the model. Between 4 and 20 iterations of the EM algorithm.

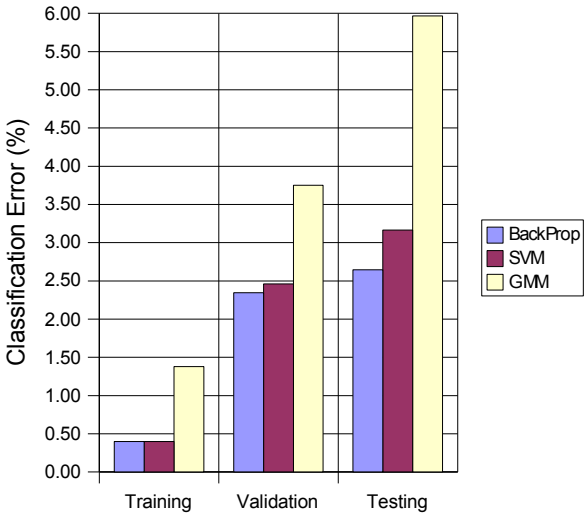


Figure 8: Comparison of different classification methods

VII. References

[1] R. Collobert, S. Bengio, and J. Mariéthoz. *Torch: a modular machine learning software library*. Technical Report IDIAP-RR 02-46, IDIAP, 2002.

[2] *Weka: Java-Library for Data Mining* - http://www.cs.waikato.ac.nz/~ml/weka/index.html

[3] *WDBC database details* - http://mlearn.ics.uci.edu/databases/breast-cancer-wisconsin/breast-cancer-wisconsin.names

[4] Morgan Kaufmann,.*Data Mining - Practical Machine Learning Tools and Techniques*, 2nd edition.[2005] - Chap 6 – p224-234

[5] R.Duda, P.Hart, D.Stork, *Pattern Classification*, 2nd edition.[2001] - Chap 5 - p259

[6] *Kmeans algorithm* - http://en.wikipedia.org/wiki/K-means_algorithm

[7] *EM algorithm* - http://en.wikipedia.org/wiki/Expectation-maximization_algorithm

VIII. Appendix

The code for loading the data and setting the measurer is not relevant, so we decide to insert it only for the back-propagation in the appendix. We also use a structure *MlpParam* to ease the use of function having a lot of parameters. This structure may differ between the back propagation, the SVM and the GMM implementation.

Part B,C,D of the appendix only contains relevant code.

A. Back propagation

```
/* A structure is used to ease the use of parameters*/
struct MlpParam{
    int n_inputs;
    int n_outputs;
    int n_hu;
    int max_iter;
    real accuracy;
    real learning_rate;
    real weight_decay;
    char *file;
    char *valid_file;
    char *suffix;
    char *model_file;
};


//=================== Create the MLP... =========
ConnectedMachine *createMachine(Allocator *allocator, MlpParam
*param) {
    ConnectedMachine *mlp = new(allocator) ConnectedMachine();
    if(param->n_hu > 0)  {

        //Set the first layer (input -> hidden units)
        Linear *c1 = new(allocator) Linear( param->n_inputs, param-
>n_hu);
        c1->setROption("weight decay", param->weight_decay);
        mlp->addFCL(c1);

        //Set the second layer (threshold in hidden units)
        Tanh *c2 = new(allocator) Tanh(param->n_hu);
        mlp->addFCL(c2);

        //Set the third layer (Output value)
        Linear *c3 = new(allocator) Linear(param->n_hu, param-
>n_outputs);
        c3->setROption("weight decay", param->weight_decay);
        mlp->addFCL(c3);

        //Put the last value to binary
        Sigmoid *c4 = new(allocator) Sigmoid(param->n_outputs);
            mlp->addFCL(c4);
        }

    // Initialize the MLP
    mlp->build();
    mlp->setPartialBackprop();

    return mlp;
}


//=================== The Trainer ================
StochasticGradient *createTrainer(Allocator *allocator,
ConnectedMachine *mlp, MlpParam *param) {


    // The criterion for the StochasticGradient (MSE criterion)
    Criterion *criterion = NULL;
    criterion = new(allocator) MSECriterion(mlp->n_outputs);

    // The Gradient Machine Trainer
    StochasticGradient *trainer = new(allocator) StochasticGradient(mlp,
criterion);

    if(param !=NULL) {
        trainer->setIOption("max iter",param->max_iter);
        trainer->setROption("end accuracy", param->accuracy);
        trainer->setROption("learning rate", param->learning_rate);
    }
    return trainer;
}
...

    //=========== Loading & Normalize Data  ========
    MatDataSet *mat_vdata = new(allocator) MatDataSet(param-
>valid_file, param->n_inputs, param->n_outputs);
    MatDataSet *mat_data = new(allocator) MatDataSet(param->file,
param->n_inputs, param->n_outputs);
    MeanVarNorm *mv_norm = new(allocator)
MeanVarNorm(mat_data);
        mat_data->preProcess(mv_norm);
    mat_vdata->preProcess(mv_norm);

    //Setting the class label type for ClassMeasurer
    Sequence *class_labels = new(allocator) Sequence(2,1);
    class_labels->frames[0][0] =  0;
    class_labels->frames[0][1] =  1;
    DataSet *data = new(allocator) ClassFormatDataSet(mat_data,
class_labels);
    DataSet *vdata = new(allocator) ClassFormatDataSet(mat_vdata,
class_labels);
    TwoClassFormat *class_format = new(allocator)
TwoClassFormat(data);

    //=================== Measurer  ===================
    char mse_train_fname[256] = "MSE_train";
    strcat(mse_train_fname,param->suffix);
    DiskXFile *mse_train_file = new(allocator)
DiskXFile(mse_train_fname, "w");
    MSEMeasurer *mse_meas = new(allocator) MSEMeasurer(mlp-
>outputs, data, mse_train_file);
    measurers.addNode(mse_meas);
    ClassMeasurer *class_meas = new(allocator) ClassMeasurer(mlp-
>outputs, data, class_format, cmd->getXFile("class_err"));
    measurers.addNode(class_meas);

[...]

trainer->train(data, &measurers);
```

B. SVM

```
//======= Create the MLP... ===

SVM * createMachine(Allocator *allocator, SvmParam *param) {

    //Create the kernel type
    Kernel *kernel = new(allocator) GaussianKernel(1./(param-
>stdv*param->stdv));

    //Create SVM
    SVM *svm = new(allocator) SVMClassification(kernel);


    if(param->mode == TRAIN)   {
        svm->setROption("C", param->c_cst);
        svm->setROption("cache size", param->cache_size);
    }

    return svm;
}
...

    //Use to format the matdatset class label from 0,1 to -1,1
    Sequence *class_labels = new(allocator) Sequence(2, 1);
    class_labels->frames[0][0] = -1;
    class_labels->frames[1][0] = 1;
    DataSet *data = new(allocator) ClassFormatDataSet(mat_data,
class_labels);
...

    //We use a classmeasurer to get the classification error.
    MeasurerList measurers;
    TwoClassFormat *class_format = new(allocator)
TwoClassFormat(data);
    char class_train_fname[256] = "Class_train";
    strcat(class_train_fname,param->suffix);
    DiskXFile *class_train_file = new(allocator)
DiskXFile(class_train_fname, "a");
    ClassMeasurer *class_meas = new(allocator) ClassMeasurer(svm-
>outputs, data, class_format, class_train_file);
    measurers.addNode(class_meas);

...
    trainer.train(data, NULL);
    message("%d SV with %d at bounds", svm->n_support_vectors, svm-
>n_support_vectors_bound);
    trainer.test(&measurers);
```

C. GMM – Classification error function

```
real error(DiagonalGMM ** gmms, int n_classes, DataSet * data, int
classe_id){
    real * gmm_outputs = new real[n_classes];
    int misclassified = 0;

    //loop though all examples of the data set
    for(int i= 0; i < data->n_examples; i++){
        //select example 'i'
        data->setExample(i);

        for (int j=0; j < n_classes; j++){
            //update output using EM
            gmms[j]->forward(data->inputs);

            //read output
            gmm_outputs[j] = (real)gmms[j]->outputs->frames[0][0];
        }
        //compare gmm_outputs
        int greater = 0;
        for (int k=1; k<n_classes; k++){
            if ( gmm_outputs[k] > gmm_outputs[greater]) greater = k;
        }

        //check classification
        if (classe_id != greater) misclassified++;
    }
    return ((real)misclassified / (real)data->n_examples)*100;
}
```

D. Format the database

```
#define NOF_PATTERNS 569
#define NOF_DIMS 30
#define NOF_CLASSES 1
#define RATIO_TRAIN 0.02
#define RATIO_VALID 0.6
#define ID_VS_OTHERS 0

using namespace std;

/* Class that is use to organize the patterns */
class Pattern {

public:
    int id;
    string code;
    string values;

    //Constructor by default
    Pattern():
        id(0), code("0"), values("")
    {}

    //Destructor
    ~Pattern() {}

    void setCode(string name) {
        if(name.compare("M") == 0) {
            id=1;
            code="1"
        }
        else {
            id=1;
            code="1"
        }
    }

    void setValues(string str) {
        for(int i(0); i<str.length(); i++) {
            if(str[i] == ',') str[i]=' ';
        }
        values = str;
    }

    friend ostream& operator<<(ostream& out, const Pattern&  p);
};

// opérateur pour l'affichage d'un pattern
ostream& operator<<(ostream& out, const Pattern&  p)
{
    out << p.values << " " << p.code;
    return out;
}

struct SortByCode
{
   // Object version
    bool operator ()(const Pattern & p1,const Pattern & p2 ) const
```

```
    {
        return p1.id < p2.id;
    }

};

class iotaGen
{
public:
  iotaGen (int start = 0) : current(start) { }
  int operator() () { return current++; }
private:
  int current;
};

// do it again, with explicit random number generator
struct RandomInteger {
      int operator() (int m) { return rand() % m; }
} randomize;

//Randomly shuffle a vector starting from start to end.
vector<int> randperm(int start,int end)
{
  // first make the vector containing 1 2 3 ... 10
  std::vector<int> numbers(end-start);
  std::generate(numbers.begin(), numbers.end(), iotaGen(start));

  // then randomly shuffle the elements
  std::random_shuffle(numbers.begin(), numbers.end(),randomize);

    return numbers;
 }
```