```
#ifndef __OPENCV_ML_HPP__
#define __OPENCV_ML_HPP__

#ifdef __cplusplus
#  include "opencv2/core.hpp"
#endif

#include "opencv2/core/core_c.h"
#include <limits.h>

#ifdef __cplusplus

#include <map>
#include <iostream>

// Apple defines a check() macro somewhere in the debug headers
// that interferes with a method definiton in this header
#undef check

/****************************************************************************************\
*                                 Main struct definitions                                *
\****************************************************************************************/

/* log(2*PI) */
#define CV_LOG2PI (1.8378770664093454835606594728112)

/* columns of <trainData> matrix are training samples */
#define CV_COL_SAMPLE 0

/* rows of <trainData> matrix are training samples */
#define CV_ROW_SAMPLE 1

#define CV_IS_ROW_SAMPLE(flags) ((flags) & CV_ROW_SAMPLE)

struct CvVectors
{
    int type;
    int dims, count;
    CvVectors* next;
    union
    {
        uchar** ptr;
        float** fl;
        double** db;
    } data;
};

#if 0
/* A structure, representing the lattice range of statmodel parameters.
   It is used for optimizing statmodel parameters by cross-validation method.
   The lattice is logarithmic, so <step> must be greater then 1. */
typedef struct CvParamLattice
{
    double min_val;
    double max_val;
    double step;
}
CvParamLattice;

CV_INLINE CvParamLattice cvParamLattice( double min_val, double max_val,
                                         double log_step )
{
    CvParamLattice pl;
    pl.min_val = MIN( min_val, max_val );
    pl.max_val = MAX( min_val, max_val );
    pl.step = MAX( log_step, 1. );
    return pl;
}

CV_INLINE CvParamLattice cvDefaultParamLattice( void )
{
    CvParamLattice pl = {0,0,0};
    return pl;
}
#endif

/* Variable type */
#define CV_VAR_NUMERICAL    0
#define CV_VAR_ORDERED      0
#define CV_VAR_CATEGORICAL  1

#define CV_TYPE_NAME_ML_SVM         "opencv-ml-svm"
#define CV_TYPE_NAME_ML_KNN         "opencv-ml-knn"
#define CV_TYPE_NAME_ML_NBAYES      "opencv-ml-bayesian"
#define CV_TYPE_NAME_ML_EM          "opencv-ml-em"
#define CV_TYPE_NAME_ML_BOOSTING    "opencv-ml-boost-tree"
#define CV_TYPE_NAME_ML_TREE        "opencv-ml-tree"
#define CV_TYPE_NAME_ML_ANN_MLP     "opencv-ml-ann-mlp"
#define CV_TYPE_NAME_ML_CNN         "opencv-ml-cnn"
#define CV_TYPE_NAME_ML_RTREES      "opencv-ml-random-trees"
```

```
#define CV_TYPE_NAME_ML_ERTREES     "opencv-ml-extremely-randomized-trees"
#define CV_TYPE_NAME_ML_GBT         "opencv-ml-gradient-boosting-trees"

#define CV_TRAIN_ERROR  0
#define CV_TEST_ERROR   1

class CV_EXPORTS_W CvStatModel
{
public:
    CvStatModel();
    virtual ~CvStatModel();

    virtual void clear();

    CV_WRAP virtual void save( const char* filename, const char* name=0 ) const;
    CV_WRAP virtual void load( const char* filename, const char* name=0 );

    virtual void write( CvFileStorage* storage, const char* name ) const;
    virtual void read( CvFileStorage* storage, CvFileNode* node );

protected:
    const char* default_model_name;
};

/****************************************************************************************\
*                                  Normal Bayes Classifier                               *
\****************************************************************************************/

/* The structure, representing the grid range of statmodel parameters.
   It is used for optimizing statmodel accuracy by varying model parameters,
   the accuracy estimate being computed by cross-validation.
   The grid is logarithmic, so <step> must be greater then 1. */

class CvMLData;

struct CV_EXPORTS_W_MAP CvParamGrid
{
    // SVM params type
    enum { SVM_C=0, SVM_GAMMA=1, SVM_P=2, SVM_NU=3, SVM_COEF=4, SVM_DEGREE=5 };

    CvParamGrid()
    {
        min_val = max_val = step = 0;
    }

    CvParamGrid( double min_val, double max_val, double log_step );
    //CvParamGrid( int param_id );
    bool check() const;

    CV_PROP_RW double min_val;
    CV_PROP_RW double max_val;
    CV_PROP_RW double step;
};

inline CvParamGrid::CvParamGrid( double _min_val, double _max_val, double _log_step )
{
    min_val = _min_val;
    max_val = _max_val;
    step = _log_step;
}

class CV_EXPORTS_W CvNormalBayesClassifier : public CvStatModel
{
public:
    CV_WRAP CvNormalBayesClassifier();
    virtual ~CvNormalBayesClassifier();

    CvNormalBayesClassifier( const CvMat* trainData, const CvMat* responses,
        const CvMat* varIdx=0, const CvMat* sampleIdx=0 );

    virtual bool train( const CvMat* trainData, const CvMat* responses,
        const CvMat* varIdx = 0, const CvMat* sampleIdx=0, bool update=false );

    virtual float predict( const CvMat* samples, CV_OUT CvMat* results=0, CV_OUT CvMat* r
esults_prob=0 ) const;
    CV_WRAP virtual void clear();

    CV_WRAP CvNormalBayesClassifier( const cv::Mat& trainData, const cv::Mat& responses,
                             const cv::Mat& varIdx=cv::Mat(), const cv::Mat& sampleIdx=cv:
:Mat() );
    CV_WRAP virtual bool train( const cv::Mat& trainData, const cv::Mat& responses,
                        const cv::Mat& varIdx = cv::Mat(), const cv::Mat& sampleIdx=cv::Ma
t(),
                        bool update=false );
    CV_WRAP virtual float predict( const cv::Mat& samples, CV_OUT cv::Mat* results=0, CV_
OUT cv::Mat* results_prob=0 ) const;

    virtual void write( CvFileStorage* storage, const char* name ) const;
    virtual void read( CvFileStorage* storage, CvFileNode* node );

protected:
    int     var_count, var_all;
    CvMat*  var_idx;
    CvMat*  cls_labels;
    CvMat** count;
    CvMat** sum;
    CvMat** productsum;
    CvMat** avg;
    CvMat** inv_eigen_values;
    CvMat** cov_rotate_mats;
    CvMat*  c;
};


/****************************************************************************************\
*                           K-Nearest Neighbour Classifier                               *
\****************************************************************************************/

// k Nearest Neighbors
class CV_EXPORTS_W CvKNearest : public CvStatModel
{
public:

    CV_WRAP CvKNearest();
    virtual ~CvKNearest();

    CvKNearest( const CvMat* trainData, const CvMat* responses,
                const CvMat* sampleIdx=0, bool isRegression=false, int max_k=32 );

    virtual bool train( const CvMat* trainData, const CvMat* responses,
                        const CvMat* sampleIdx=0, bool is_regression=false,
                        int maxK=32, bool updateBase=false );

    virtual float find_nearest( const CvMat* samples, int k, CV_OUT CvMat* results=0,
        const float** neighbors=0, CV_OUT CvMat* neighborResponses=0, CV_OUT CvMat* dist=
0 ) const;

    CV_WRAP CvKNearest( const cv::Mat& trainData, const cv::Mat& responses,
```

```
                const cv::Mat& sampleIdx=cv::Mat(), bool isRegression=false, int max_k=32
);

    CV_WRAP virtual bool train( const cv::Mat& trainData, const cv::Mat& responses,
                        const cv::Mat& sampleIdx=cv::Mat(), bool isRegression=false,
                        int maxK=32, bool updateBase=false );

    virtual float find_nearest( const cv::Mat& samples, int k, cv::Mat* results=0,
                                const float** neighbors=0, cv::Mat* neighborResponses=0,
                                cv::Mat* dist=0 ) const;
    CV_WRAP virtual float find_nearest( const cv::Mat& samples, int k, CV_OUT cv::Mat& re
sults,
                                       CV_OUT cv::Mat& neighborResponses, CV_OUT cv::Mat
& dists) const;

    virtual void clear();
    int get_max_k() const;
    int get_var_count() const;
    int get_sample_count() const;
    bool is_regression() const;

    virtual float write_results( int k, int k1, int start, int end,
        const float* neighbor_responses, const float* dist, CvMat* _results,
        CvMat* _neighbor_responses, CvMat* _dist, Cv32suf* sort_buf ) const;

    virtual void find_neighbors_direct( const CvMat* _samples, int k, int start, int end,
        float* neighbor_responses, const float** neighbors, float* dist ) const;

protected:

    int max_k, var_count;
    int total;
    bool regression;
    CvVectors* samples;
};

/****************************************************************************************
\
*                                  Support Vector Machines
*
\****************************************************************************************
/

// SVM training parameters
struct CV_EXPORTS_W_MAP CvSVMParams
{
    CvSVMParams();
    CvSVMParams( int svm_type, int kernel_type,
                 double degree, double gamma, double coef0,
                 double Cvalue, double nu, double p,
                 CvMat* class_weights, CvTermCriteria term_crit );

    CV_PROP_RW int         svm_type;
    CV_PROP_RW int         kernel_type;
    CV_PROP_RW double      degree; // for poly
    CV_PROP_RW double      gamma;  // for poly/rbf/sigmoid/chi2
    CV_PROP_RW double      coef0;  // for poly/sigmoid

    CV_PROP_RW double      C;  // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVR
    CV_PROP_RW double      nu; // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_NU_SVR
    CV_PROP_RW double      p; // for CV_SVM_EPS_SVR
    CvMat*      class_weights; // for CV_SVM_C_SVC
    CV_PROP_RW CvTermCriteria term_crit; // termination criteria
};


struct CV_EXPORTS CvSVMKernel
{
    typedef void (CvSVMKernel::*Calc)( int vec_count, int vec_size, const float** vecs,
                                       const float* another, float* results );
    CvSVMKernel();
    CvSVMKernel( const CvSVMParams* params, Calc _calc_func );
    virtual bool create( const CvSVMParams* params, Calc _calc_func );
    virtual ~CvSVMKernel();

    virtual void clear();
    virtual void calc( int vcount, int n, const float** vecs, const float* another, float
* results );

    const CvSVMParams* params;
    Calc calc_func;

    virtual void calc_non_rbf_base( int vec_count, int vec_size, const float** vecs,
                                    const float* another, float* results,
                                    double alpha, double beta );
    virtual void calc_intersec( int vcount, int var_count, const float** vecs,
                                const float* another, float* results );
    virtual void calc_chi2( int vec_count, int vec_size, const float** vecs,
                            const float* another, float* results );
    virtual void calc_linear( int vec_count, int vec_size, const float** vecs,
                              const float* another, float* results );
    virtual void calc_rbf( int vec_count, int vec_size, const float** vecs,
                           const float* another, float* results );
    virtual void calc_poly( int vec_count, int vec_size, const float** vecs,
                            const float* another, float* results );
    virtual void calc_sigmoid( int vec_count, int vec_size, const float** vecs,
                               const float* another, float* results );
};


struct CvSVMKernelRow
{
    CvSVMKernelRow* prev;
    CvSVMKernelRow* next;
    float* data;
};


struct CvSVMSolutionInfo
{
    double obj;
    double rho;
    double upper_bound_p;
    double upper_bound_n;
    double r;   // for Solver_NU
};

class CV_EXPORTS CvSVMSolver
{
public:
    typedef bool (CvSVMSolver::*SelectWorkingSet)( int& i, int& j );
    typedef float* (CvSVMSolver::*GetRow)( int i, float* row, float* dst, bool existed );
    typedef void (CvSVMSolver::*CalcRho)( double& rho, double& r );

    CvSVMSolver();

    CvSVMSolver( int count, int var_count, const float** samples, schar* y,
                 int alpha_count, double* alpha, double Cp, double Cn,
                 CvMemStorage* storage, CvSVMKernel* kernel, GetRow get_row,
                 SelectWorkingSet select_working_set, CalcRho calc_rho );
    virtual bool create( int count, int var_count, const float** samples, schar* y,
                 int alpha_count, double* alpha, double Cp, double Cn,
                 CvMemStorage* storage, CvSVMKernel* kernel, GetRow get_row,
                 SelectWorkingSet select_working_set, CalcRho calc_rho );
```

```
    virtual ~CvSVMSolver();

    virtual void clear();
    virtual bool solve_generic( CvSVMSolutionInfo& si );

    virtual bool solve_c_svc( int count, int var_count, const float** samples, schar* y,
                              double Cp, double Cn, CvMemStorage* storage,
                              CvSVMKernel* kernel, double* alpha, CvSVMSolutionInfo& si )
;
    virtual bool solve_nu_svc( int count, int var_count, const float** samples, schar* y,
                               CvMemStorage* storage, CvSVMKernel* kernel,
                               double* alpha, CvSVMSolutionInfo& si );
    virtual bool solve_one_class( int count, int var_count, const float** samples,
                                  CvMemStorage* storage, CvSVMKernel* kernel,
                                  double* alpha, CvSVMSolutionInfo& si );

    virtual bool solve_eps_svr( int count, int var_count, const float** samples, const fl
oat* y,
                                CvMemStorage* storage, CvSVMKernel* kernel,
                                double* alpha, CvSVMSolutionInfo& si );

    virtual bool solve_nu_svr( int count, int var_count, const float** samples, const flo
at* y,
                               CvMemStorage* storage, CvSVMKernel* kernel,
                               double* alpha, CvSVMSolutionInfo& si );

    virtual float* get_row_base( int i, bool* _existed );
    virtual float* get_row( int i, float* dst );

    int sample_count;
    int var_count;
    int cache_size;
    int cache_line_size;
    const float** samples;
    const CvSVMParams* params;
    CvMemStorage* storage;
    CvSVMKernelRow lru_list;
    CvSVMKernelRow* rows;

    int alpha_count;

    double* G;
    double* alpha;

    // -1 - lower bound, 0 - free, 1 - upper bound
    schar* alpha_status;

    schar* y;
    double* b;
    float* buf[2];
    double eps;
    int max_iter;
    double C[2];  // C[0] == Cn, C[1] == Cp
    CvSVMKernel* kernel;

    SelectWorkingSet select_working_set_func;
    CalcRho calc_rho_func;
    GetRow get_row_func;

    virtual bool select_working_set( int& i, int& j );
    virtual bool select_working_set_nu_svm( int& i, int& j );
    virtual void calc_rho( double& rho, double& r );
    virtual void calc_rho_nu_svm( double& rho, double& r );

    virtual float* get_row_svc( int i, float* row, float* dst, bool existed );
    virtual float* get_row_one_class( int i, float* row, float* dst, bool existed );
    virtual float* get_row_svr( int i, float* row, float* dst, bool existed );
};


struct CvSVMDecisionFunc
{
    double rho;
    int sv_count;
    double* alpha;
    int* sv_index;
};


// SVM model
class CV_EXPORTS_W CvSVM : public CvStatModel
{
public:
    // SVM type
    enum { C_SVC=100, NU_SVC=101, ONE_CLASS=102, EPS_SVR=103, NU_SVR=104 };

    // SVM kernel type
    enum { LINEAR=0, POLY=1, RBF=2, SIGMOID=3, CHI2=4, INTER=5 };

    // SVM params type
    enum { C=0, GAMMA=1, P=2, NU=3, COEF=4, DEGREE=5 };

    CV_WRAP CvSVM();
    virtual ~CvSVM();

    CvSVM( const CvMat* trainData, const CvMat* responses,
           const CvMat* varIdx=0, const CvMat* sampleIdx=0,
           CvSVMParams params=CvSVMParams() );

    virtual bool train( const CvMat* trainData, const CvMat* responses,
                        const CvMat* varIdx=0, const CvMat* sampleIdx=0,
                        CvSVMParams params=CvSVMParams() );

    virtual bool train_auto( const CvMat* trainData, const CvMat* responses,
        const CvMat* varIdx, const CvMat* sampleIdx, CvSVMParams params,
        int kfold = 10,
        CvParamGrid Cgrid      = get_default_grid(CvSVM::C),
        CvParamGrid gammaGrid  = get_default_grid(CvSVM::GAMMA),
        CvParamGrid pGrid      = get_default_grid(CvSVM::P),
        CvParamGrid nuGrid     = get_default_grid(CvSVM::NU),
        CvParamGrid coeffGrid  = get_default_grid(CvSVM::COEF),
        CvParamGrid degreeGrid = get_default_grid(CvSVM::DEGREE),
        bool balanced=false );

    virtual float predict( const CvMat* sample, bool returnDFVal=false ) const;
    virtual float predict( const CvMat* samples, CV_OUT CvMat* results, bool returnDFVal=
false ) const;

    CV_WRAP CvSVM( const cv::Mat& trainData, const cv::Mat& responses,
          const cv::Mat& varIdx=cv::Mat(), const cv::Mat& sampleIdx=cv::Mat(),
          CvSVMParams params=CvSVMParams() );

    CV_WRAP virtual bool train( const cv::Mat& trainData, const cv::Mat& responses,
                        const cv::Mat& varIdx=cv::Mat(), const cv::Mat& sampleIdx=cv::Mat(
),
                        CvSVMParams params=CvSVMParams() );

    CV_WRAP virtual bool train_auto( const cv::Mat& trainData, const cv::Mat& responses,
                             const cv::Mat& varIdx, const cv::Mat& sampleIdx, CvSVMParams
params,
                             int k_fold = 10,
                             CvParamGrid Cgrid      = CvSVM::get_default_grid(CvSVM::C),
                             CvParamGrid gammaGrid  = CvSVM::get_default_grid(CvSVM::GAMMA
),
```

```
                            CvParamGrid pGrid      = CvSVM::get_default_grid(CvSVM::P),
                            CvParamGrid nuGrid     = CvSVM::get_default_grid(CvSVM::NU),
                            CvParamGrid coeffGrid  = CvSVM::get_default_grid(CvSVM::COEF)
,
                            CvParamGrid degreeGrid = CvSVM::get_default_grid(CvSVM::DEGRE
E),
                            bool balanced=false);
    CV_WRAP virtual float predict( const cv::Mat& sample, bool returnDFVal=false ) const;
    CV_WRAP_AS(predict_all) virtual void predict( cv::InputArray samples, cv::OutputArray
 results ) const;

    CV_WRAP virtual int get_support_vector_count() const;
    virtual const float* get_support_vector(int i) const;
    virtual CvSVMParams get_params() const { return params; }
    CV_WRAP virtual void clear();

    virtual const CvSVMDecisionFunc* get_decision_function() const { return decision_func
; }

    static CvParamGrid get_default_grid( int param_id );

    virtual void write( CvFileStorage* storage, const char* name ) const;
    virtual void read( CvFileStorage* storage, CvFileNode* node );
    CV_WRAP int get_var_count() const { return var_idx ? var_idx->cols : var_all; }

protected:

    virtual bool set_params( const CvSVMParams& params );
    virtual bool train1( int sample_count, int var_count, const float** samples,
                         const void* responses, double Cp, double Cn,
                         CvMemStorage* _storage, double* alpha, double& rho );
    virtual bool do_train( int svm_type, int sample_count, int var_count, const float** s
amples,
                         const CvMat* responses, CvMemStorage* _storage, double* alpha );
    virtual void create_kernel();
    virtual void create_solver();

    virtual float predict( const float* row_sample, int row_len, bool returnDFVal=false )
 const;

    virtual void write_params( CvFileStorage* fs ) const;
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    void optimize_linear_svm();

    CvSVMParams params;
    CvMat* class_labels;
    int var_all;
    float** sv;
    int sv_total;
    CvMat* var_idx;
    CvMat* class_weights;
    CvSVMDecisionFunc* decision_func;
    CvMemStorage* storage;

    CvSVMSolver* solver;
    CvSVMKernel* kernel;

private:
    CvSVM(const CvSVM&);
    CvSVM& operator = (const CvSVM&);
};

/****************************************************************************************
\
*                              Expectation - Maximization
*
 \****************************************************************************************
/
namespace cv
{
class CV_EXPORTS_W EM : public Algorithm
{
public:
    // Type of covariation matrices
    enum {COV_MAT_SPHERICAL=0, COV_MAT_DIAGONAL=1, COV_MAT_GENERIC=2, COV_MAT_DEFAULT=COV
_MAT_DIAGONAL};

    // Default parameters
    enum {DEFAULT_NCLUSTERS=5, DEFAULT_MAX_ITERS=100};

    // The initial step
    enum {START_E_STEP=1, START_M_STEP=2, START_AUTO_STEP=0};

    CV_WRAP EM(int nclusters=EM::DEFAULT_NCLUSTERS, int covMatType=EM::COV_MAT_DIAGONAL,
       const TermCriteria& termCrit=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS,
                                                 EM::DEFAULT_MAX_ITERS, FLT_EPSILON));

    virtual ~EM();
    CV_WRAP virtual void clear();

    CV_WRAP virtual bool train(InputArray samples,
                       OutputArray logLikelihoods=noArray(),
                       OutputArray labels=noArray(),
                       OutputArray probs=noArray());

    CV_WRAP virtual bool trainE(InputArray samples,
                        InputArray means0,
                        InputArray covs0=noArray(),
                        InputArray weights0=noArray(),
                        OutputArray logLikelihoods=noArray(),
                        OutputArray labels=noArray(),
                        OutputArray probs=noArray());

    CV_WRAP virtual bool trainM(InputArray samples,
                        InputArray probs0,
                        OutputArray logLikelihoods=noArray(),
                        OutputArray labels=noArray(),
                        OutputArray probs=noArray());

    CV_WRAP Vec2d predict(InputArray sample,
                OutputArray probs=noArray()) const;

    CV_WRAP bool isTrained() const;

    AlgorithmInfo* info() const;
    virtual void read(const FileNode& fn);

protected:

    virtual void setTrainData(int startStep, const Mat& samples,
                              const Mat* probs0,
                              const Mat* means0,
                              const std::vector<Mat>* covs0,
                              const Mat* weights0);

    bool doTrain(int startStep,
                 OutputArray logLikelihoods,
                 OutputArray labels,
                 OutputArray probs);
    virtual void eStep();
    virtual void mStep();

    void clusterTrainSamples();
```

```
    void decomposeCovs();
    void computeLogWeightDivDet();

    Vec2d computeProbabilities(const Mat& sample, Mat* probs) const;

    // all inner matrices have type CV_64FC1
    CV_PROP_RW int nclusters;
    CV_PROP_RW int covMatType;
    CV_PROP_RW int maxIters;
    CV_PROP_RW double epsilon;

    Mat trainSamples;
    Mat trainProbs;
    Mat trainLogLikelihoods;
    Mat trainLabels;

    CV_PROP Mat weights;
    CV_PROP Mat means;
    CV_PROP std::vector<Mat> covs;

    std::vector<Mat> covsEigenValues;
    std::vector<Mat> covsRotateMats;
    std::vector<Mat> invCovsEigenValues;
    Mat logWeightDivDet;
};
} // namespace cv

/////////////////////////////////////////////////////////////////////////////////////////
\
*                                       Decision Tree                                 *
*
\/////////////////////////////////////////////////////////////////////////////////////////
/\
struct CvPair16u32s
{
    unsigned short* u;
    int* i;
};


#define CV_DTREE_CAT_DIR(idx,subset) \
    (2*((subset[(idx)>>5]&(1 << ((idx) & 31)))==0)-1)

struct CvDTreeSplit
{
    int var_idx;
    int condensed_idx;
    int inversed;
    float quality;
    CvDTreeSplit* next;
    union
    {
        int subset[2];
        struct
        {
            float c;
            int split_point;
        }
        ord;
    };
};

struct CvDTreeNode
{
    int class_idx;
    int Tn;
    double value;

    CvDTreeNode* parent;
    CvDTreeNode* left;
    CvDTreeNode* right;

    CvDTreeSplit* split;

    int sample_count;
    int depth;
    int* num_valid;
    int offset;
    int buf_idx;
    double maxlr;

    // global pruning data
    int complexity;
    double alpha;
    double node_risk, tree_risk, tree_error;

    // cross-validation pruning data
    int* cv_Tn;
    double* cv_node_risk;
    double* cv_node_error;

    int get_num_valid(int vi) { return num_valid ? num_valid[vi] : sample_count; }
    void set_num_valid(int vi, int n) { if( num_valid ) num_valid[vi] = n; }
};


struct CV_EXPORTS_W_MAP CvDTreeParams
{
    CV_PROP_RW int   max_categories;
    CV_PROP_RW int   max_depth;
    CV_PROP_RW int   min_sample_count;
    CV_PROP_RW int   cv_folds;
    CV_PROP_RW bool  use_surrogates;
    CV_PROP_RW bool  use_1se_rule;
    CV_PROP_RW bool  truncate_pruned_tree;
    CV_PROP_RW float regression_accuracy;
    const float* priors;

    CvDTreeParams();
    CvDTreeParams( int max_depth, int min_sample_count,
                   float regression_accuracy, bool use_surrogates,
                   int max_categories, int cv_folds,
                   bool use_1se_rule, bool truncate_pruned_tree,
                   const float* priors );
};


struct CV_EXPORTS CvDTreeTrainData
{
    CvDTreeTrainData();
    CvDTreeTrainData( const CvMat* trainData, int tflag,
                      const CvMat* responses, const CvMat* varIdx=0,
                      const CvMat* sampleIdx=0, const CvMat* varType=0,
                      const CvMat* missingDataMask=0,
                      const CvDTreeParams& params=CvDTreeParams(),
                      bool _shared=false, bool _add_labels=false );
    virtual ~CvDTreeTrainData();

    virtual void set_data( const CvMat* trainData, int tflag,
                           const CvMat* responses, const CvMat* varIdx=0,
                           const CvMat* sampleIdx=0, const CvMat* varType=0,
                           const CvMat* missingDataMask=0,
                           const CvDTreeParams& params=CvDTreeParams(),
                           bool _shared=false, bool _add_labels=false,
```

```
                        bool _update_data=false );
    virtual void do_responses_copy();

    virtual void get_vectors( const CvMat* _subsample_idx,
         float* values, uchar* missing, float* responses, bool get_class_idx=false );

    virtual CvDTreeNode* subsample_data( const CvMat* _subsample_idx );

    virtual void write_params( CvFileStorage* fs ) const;
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    // release all the data
    virtual void clear();

    int get_num_classes() const;
    int get_var_type(int vi) const;
    int get_work_var_count() const {return work_var_count;}

    virtual const float* get_ord_responses( CvDTreeNode* n, float* values_buf, int* sampl
e_indices_buf );
    virtual const int* get_class_labels( CvDTreeNode* n, int* labels_buf );
    virtual const int* get_cv_labels( CvDTreeNode* n, int* labels_buf );
    virtual const int* get_sample_indices( CvDTreeNode* n, int* indices_buf );
    virtual const int* get_cat_var_data( CvDTreeNode* n, int vi, int* cat_values_buf );
    virtual void get_ord_var_data( CvDTreeNode* n, int vi, float* ord_values_buf, int* so
rted_indices_buf,
                                   const float** ord_values, const int** sorted_indices,
int* sample_indices_buf );
    virtual int get_child_buf_idx( CvDTreeNode* n );

    ////////////////////////////////////

    virtual bool set_params( const CvDTreeParams& params );
    virtual CvDTreeNode* new_node( CvDTreeNode* parent, int count,
                                   int storage_idx, int offset );

    virtual CvDTreeSplit* new_split_ord( int vi, float cmp_val,
                int split_point, int inversed, float quality );
    virtual CvDTreeSplit* new_split_cat( int vi, float quality );
    virtual void free_node_data( CvDTreeNode* node );
    virtual void free_train_data();
    virtual void free_node( CvDTreeNode* node );

    int sample_count, var_all, var_count, max_c_count;
    int ord_var_count, cat_var_count, work_var_count;
    bool have_labels, have_priors;
    bool is_classifier;
    int tflag;

    const CvMat* train_data;
    const CvMat* responses;
    CvMat* responses_copy; // used in Boosting

    int buf_count, buf_size; // buf_size is obsolete, please do not use it, use expressio
n ((int64)buf->rows * (int64)buf->cols / buf_count) instead
    bool shared;
    int is_buf_16u;

    CvMat* cat_count;
    CvMat* cat_ofs;
    CvMat* cat_map;

    CvMat* counts;
    CvMat* buf;
    inline size_t get_length_subbuf() const
    {
        size_t res = (size_t)(work_var_count + 1) * (size_t)sample_count;
        return res;
    }

    CvMat* direction;
    CvMat* split_buf;

    CvMat* var_idx;
    CvMat* var_type; // i-th element =
                     //   k<0  - ordered
                     //   k>=0 - categorical, see k-th element of cat_* arrays
    CvMat* priors;
    CvMat* priors_mult;

    CvDTreeParams params;

    CvMemStorage* tree_storage;
    CvMemStorage* temp_storage;

    CvDTreeNode* data_root;

    CvSet* node_heap;
    CvSet* split_heap;
    CvSet* cv_heap;
    CvSet* nv_heap;

    cv::RNG* rng;
};

class CvDTree;
class CvForestTree;

namespace cv
{
    struct DTreeBestSplitFinder;
    struct ForestTreeBestSplitFinder;
}

class CV_EXPORTS_W CvDTree : public CvStatModel
{
public:
    CV_WRAP CvDTree();
    virtual ~CvDTree();

    virtual bool train( const CvMat* trainData, int tflag,
                        const CvMat* responses, const CvMat* varIdx=0,
                        const CvMat* sampleIdx=0, const CvMat* varType=0,
                        const CvMat* missingDataMask=0,
                        CvDTreeParams params=CvDTreeParams() );

    virtual bool train( CvMLData* trainData, CvDTreeParams params=CvDTreeParams() );

    // type in {CV_TRAIN_ERROR, CV_TEST_ERROR}
    virtual float calc_error( CvMLData* trainData, int type, std::vector<float> *resp = 0
 );

    virtual bool train( CvDTreeTrainData* trainData, const CvMat* subsampleIdx );

    virtual CvDTreeNode* predict( const CvMat* sample, const CvMat* missingDataMask=0,
                                  bool preprocessedInput=false ) const;

    CV_WRAP virtual bool train( const cv::Mat& trainData, int tflag,
                        const cv::Mat& responses, const cv::Mat& varIdx=cv::Mat(),
                        const cv::Mat& sampleIdx=cv::Mat(), const cv::Mat& varType=cv::Mat
(),
                        const cv::Mat& missingDataMask=cv::Mat(),
                        CvDTreeParams params=CvDTreeParams() );
```

```
    CV_WRAP virtual CvDTreeNode* predict( const cv::Mat& sample, const cv::Mat& missingDa
taMask=cv::Mat(),
                                bool preprocessedInput=false ) const;
    CV_WRAP virtual cv::Mat getVarImportance();

    virtual const CvMat* get_var_importance();
    CV_WRAP virtual void clear();

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name ) const;

    // special read & write methods for trees in the tree ensembles
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                       CvDTreeTrainData* data );
    virtual void write( CvFileStorage* fs ) const;

    const CvDTreeNode* get_root() const;
    int get_pruned_tree_idx() const;
    CvDTreeTrainData* get_data();

protected:
    friend struct cv::DTreeBestSplitFinder;

    virtual bool do_train( const CvMat* _subsample_idx );

    virtual void try_split_node( CvDTreeNode* n );
    virtual void split_node_data( CvDTreeNode* n );
    virtual CvDTreeSplit* find_best_split( CvDTreeNode* n );
    virtual CvDTreeSplit* find_split_ord_class( CvDTreeNode* n, int vi,
                            float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_
buf = 0 );
    virtual CvDTreeSplit* find_split_cat_class( CvDTreeNode* n, int vi,
                            float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_
buf = 0 );
    virtual CvDTreeSplit* find_split_ord_reg( CvDTreeNode* n, int vi,
                            float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_
buf = 0 );
    virtual CvDTreeSplit* find_split_cat_reg( CvDTreeNode* n, int vi,
                            float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_
buf = 0 );
    virtual CvDTreeSplit* find_surrogate_split_ord( CvDTreeNode* n, int vi, uchar* ext_bu
f = 0 );
    virtual CvDTreeSplit* find_surrogate_split_cat( CvDTreeNode* n, int vi, uchar* ext_bu
f = 0 );
    virtual double calc_node_dir( CvDTreeNode* node );
    virtual void complete_node_dir( CvDTreeNode* node );
    virtual void cluster_categories( const int* vectors, int vector_count,
        int var_count, int* sums, int k, int* cluster_labels );

    virtual void calc_node_value( CvDTreeNode* node );

    virtual void prune_cv();
    virtual double update_tree_rnc( int T, int fold );
    virtual int cut_tree( int T, int fold, double min_alpha );
    virtual void free_prune_data(bool cut_tree);
    virtual void free_tree();

    virtual void write_node( CvFileStorage* fs, CvDTreeNode* node ) const;
    virtual void write_split( CvFileStorage* fs, CvDTreeSplit* split ) const;
    virtual CvDTreeNode* read_node( CvFileStorage* fs, CvFileNode* node, CvDTreeNode* par
ent );
    virtual CvDTreeSplit* read_split( CvFileStorage* fs, CvFileNode* node );
    virtual void write_tree_nodes( CvFileStorage* fs ) const;
    virtual void read_tree_nodes( CvFileStorage* fs, CvFileNode* node );

    CvDTreeNode* root;
    CvMat* var_importance;
    CvDTreeTrainData* data;
    CvMat train_data_hdr, responses_hdr;
    cv::Mat train_data_mat, responses_mat;

public:
    int pruned_tree_idx;
};


/****************************************************************************************
\
*                                    Random Trees Classifier                            *
*
\****************************************************************************************
/

class CvRTrees;

class CV_EXPORTS CvForestTree: public CvDTree
{
public:
    CvForestTree();
    virtual ~CvForestTree();

    virtual bool train( CvDTreeTrainData* trainData, const CvMat* _subsample_idx, CvRTree
s* forest );

    virtual int get_var_count() const {return data ? data->var_count : 0;}
    virtual void read( CvFileStorage* fs, CvFileNode* node, CvRTrees* forest, CvDTreeTrai
nData* _data );

    /* dummy methods to avoid warnings: BEGIN */
    virtual bool train( const CvMat* trainData, int tflag,
                        const CvMat* responses, const CvMat* varIdx=0,
                        const CvMat* sampleIdx=0, const CvMat* varType=0,
                        const CvMat* missingDataMask=0,
                        CvDTreeParams params=CvDTreeParams() );

    virtual bool train( CvDTreeTrainData* trainData, const CvMat* _subsample_idx );
    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                       CvDTreeTrainData* data );
    /* dummy methods to avoid warnings: END */

protected:
    friend struct cv::ForestTreeBestSplitFinder;

    virtual CvDTreeSplit* find_best_split( CvDTreeNode* n );
    CvRTrees* forest;
};


struct CV_EXPORTS_W_MAP CvRTParams : public CvDTreeParams
{
    //Parameters for the forest
    CV_PROP_RW bool calc_var_importance; // true <=> RF processes variable importance
    CV_PROP_RW int nactive_vars;
    CV_PROP_RW CvTermCriteria term_crit;

    CvRTParams();
    CvRTParams( int max_depth, int min_sample_count,
                float regression_accuracy, bool use_surrogates,
                int max_categories, const float* priors, bool calc_var_importance,
                int nactive_vars, int max_num_of_trees_in_the_forest,
                float forest_accuracy, int termcrit_type );
};
```

```
class CV_EXPORTS_W CvRTrees : public CvStatModel
{
public:
    CV_WRAP CvRTrees();
    virtual ~CvRTrees();
    virtual bool train( const CvMat* trainData, int tflag,
                        const CvMat* responses, const CvMat* varIdx=0,
                        const CvMat* sampleIdx=0, const CvMat* varType=0,
                        const CvMat* missingDataMask=0,
                        CvRTParams params=CvRTParams() );

    virtual bool train( CvMLData* data, CvRTParams params=CvRTParams() );
    virtual float predict( const CvMat* sample, const CvMat* missing = 0 ) const;
    virtual float predict_prob( const CvMat* sample, const CvMat* missing = 0 ) const;

    CV_WRAP virtual bool train( const cv::Mat& trainData, int tflag,
                        const cv::Mat& responses, const cv::Mat& varIdx=cv::Mat(),
                        const cv::Mat& sampleIdx=cv::Mat(), const cv::Mat& varType=cv::Mat
(),
                        const cv::Mat& missingDataMask=cv::Mat(),
                        CvRTParams params=CvRTParams() );
    CV_WRAP virtual float predict( const cv::Mat& sample, const cv::Mat& missing = cv::Ma
t() ) const;
    CV_WRAP virtual float predict_prob( const cv::Mat& sample, const cv::Mat& missing = c
v::Mat() ) const;
    CV_WRAP virtual cv::Mat getVarImportance();

    CV_WRAP virtual void clear();

    virtual const CvMat* get_var_importance();
    virtual float get_proximity( const CvMat* sample1, const CvMat* sample2,
        const CvMat* missing1 = 0, const CvMat* missing2 = 0 ) const;

    virtual float calc_error( CvMLData* data, int type , std::vector<float>* resp = 0 );
// type in {CV_TRAIN_ERROR, CV_TEST_ERROR}

    virtual float get_train_error();

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name ) const;

    CvMat* get_active_var_mask();
    CvRNG* get_rng();

    int get_tree_count() const;
    CvForestTree* get_tree(int i) const;

protected:
    virtual cv::String getName() const;

    virtual bool grow_forest( const CvTermCriteria term_crit );

    // array of the trees of the forest
    CvForestTree** trees;
    CvDTreeTrainData* data;
    CvMat train_data_hdr, responses_hdr;
    cv::Mat train_data_mat, responses_mat;
    int ntrees;
    int nclasses;
    double oob_error;
    CvMat* var_importance;
    int nsamples;

    cv::RNG* rng;
    CvMat* active_var_mask;
};

/****************************************************************************************\
*                           Extremely randomized trees Classifier                        *
\****************************************************************************************/
struct CV_EXPORTS CvERTreeTrainData : public CvDTreeTrainData
{
    virtual void set_data( const CvMat* trainData, int tflag,
                           const CvMat* responses, const CvMat* varIdx=0,
                           const CvMat* sampleIdx=0, const CvMat* varType=0,
                           const CvMat* missingDataMask=0,
                           const CvDTreeParams& params=CvDTreeParams(),
                           bool _shared=false, bool _add_labels=false,
                           bool _update_data=false );
    virtual void get_ord_var_data( CvDTreeNode* n, int vi, float* ord_values_buf, int* mi
ssing_buf,
                                   const float** ord_values, const int** missing, int* sa
mple_buf = 0 );
    virtual const int* get_sample_indices( CvDTreeNode* n, int* indices_buf );
    virtual const int* get_cv_labels( CvDTreeNode* n, int* labels_buf );
    virtual const int* get_cat_var_data( CvDTreeNode* n, int vi, int* cat_values_buf );
    virtual void get_vectors( const CvMat* _subsample_idx, float* values, uchar* missing,
                              float* responses, bool get_class_idx=false );
    virtual CvDTreeNode* subsample_data( const CvMat* _subsample_idx );
    const CvMat* missing_mask;
};

class CV_EXPORTS CvForestERTree : public CvForestTree
{
protected:
    virtual double calc_node_dir( CvDTreeNode* node );
    virtual CvDTreeSplit* find_split_ord_class( CvDTreeNode* n, int vi,
        float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_buf = 0 );
    virtual CvDTreeSplit* find_split_cat_class( CvDTreeNode* n, int vi,
        float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_buf = 0 );
    virtual CvDTreeSplit* find_split_ord_reg( CvDTreeNode* n, int vi,
        float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_buf = 0 );
    virtual CvDTreeSplit* find_split_cat_reg( CvDTreeNode* n, int vi,
        float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_buf = 0 );
    virtual void split_node_data( CvDTreeNode* n );
};

class CV_EXPORTS_W CvERTrees : public CvRTrees
{
public:
    CV_WRAP CvERTrees();
    virtual ~CvERTrees();
    virtual bool train( const CvMat* trainData, int tflag,
                        const CvMat* responses, const CvMat* varIdx=0,
                        const CvMat* sampleIdx=0, const CvMat* varType=0,
                        const CvMat* missingDataMask=0,
                        CvRTParams params=CvRTParams());
    CV_WRAP virtual bool train( const cv::Mat& trainData, int tflag,
                        const cv::Mat& responses, const cv::Mat& varIdx=cv::Mat(),
                        const cv::Mat& sampleIdx=cv::Mat(), const cv::Mat& varType=cv::Mat
(),
                        const cv::Mat& missingDataMask=cv::Mat(),
                        CvRTParams params=CvRTParams());
    virtual bool train( CvMLData* data, CvRTParams params=CvRTParams() );
protected:
    virtual cv::String getName() const;
    virtual bool grow_forest( const CvTermCriteria term_crit );
};
```

```
/****************************************************************************************\
\
*                                 Boosted tree classifier
*
\****************************************************************************************/

struct CV_EXPORTS_W_MAP CvBoostParams : public CvDTreeParams
{
    CV_PROP_RW int boost_type;
    CV_PROP_RW int weak_count;
    CV_PROP_RW int split_criteria;
    CV_PROP_RW double weight_trim_rate;

    CvBoostParams();
    CvBoostParams( int boost_type, int weak_count, double weight_trim_rate,
                   int max_depth, bool use_surrogates, const float* priors );
};


class CvBoost;

class CV_EXPORTS CvBoostTree: public CvDTree
{
public:
    CvBoostTree();
    virtual ~CvBoostTree();

    virtual bool train( CvDTreeTrainData* trainData,
                        const CvMat* subsample_idx, CvBoost* ensemble );

    virtual void scale( double s );
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                       CvBoost* ensemble, CvDTreeTrainData* _data );
    virtual void clear();

    /* dummy methods to avoid warnings: BEGIN */
    virtual bool train( const CvMat* trainData, int tflag,
                        const CvMat* responses, const CvMat* varIdx=0,
                        const CvMat* sampleIdx=0, const CvMat* varType=0,
                        const CvMat* missingDataMask=0,
                        CvDTreeParams params=CvDTreeParams() );
    virtual bool train( CvDTreeTrainData* trainData, const CvMat* _subsample_idx );

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                       CvDTreeTrainData* data );
    /* dummy methods to avoid warnings: END */

protected:

    virtual void try_split_node( CvDTreeNode* n );
    virtual CvDTreeSplit* find_surrogate_split_ord( CvDTreeNode* n, int vi, uchar* ext_bu
f = 0 );
    virtual CvDTreeSplit* find_surrogate_split_cat( CvDTreeNode* n, int vi, uchar* ext_bu
f = 0 );
    virtual CvDTreeSplit* find_split_ord_class( CvDTreeNode* n, int vi,
        float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_buf = 0 );
    virtual CvDTreeSplit* find_split_cat_class( CvDTreeNode* n, int vi,
        float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_buf = 0 );
    virtual CvDTreeSplit* find_split_ord_reg( CvDTreeNode* n, int vi,
        float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_buf = 0 );
    virtual CvDTreeSplit* find_split_cat_reg( CvDTreeNode* n, int vi,
        float init_quality = 0, CvDTreeSplit* _split = 0, uchar* ext_buf = 0 );
    virtual void calc_node_value( CvDTreeNode* n );
    virtual double calc_node_dir( CvDTreeNode* n );

    CvBoost* ensemble;
};


class CV_EXPORTS_W CvBoost : public CvStatModel
{
public:
    // Boosting type
    enum { DISCRETE=0, REAL=1, LOGIT=2, GENTLE=3 };

    // Splitting criteria
    enum { DEFAULT=0, GINI=1, MISCLASS=3, SQERR=4 };

    CV_WRAP CvBoost();
    virtual ~CvBoost();

    CvBoost( const CvMat* trainData, int tflag,
             const CvMat* responses, const CvMat* varIdx=0,
             const CvMat* sampleIdx=0, const CvMat* varType=0,
             const CvMat* missingDataMask=0,
             CvBoostParams params=CvBoostParams() );

    virtual bool train( const CvMat* trainData, int tflag,
             const CvMat* responses, const CvMat* varIdx=0,
             const CvMat* sampleIdx=0, const CvMat* varType=0,
             const CvMat* missingDataMask=0,
             CvBoostParams params=CvBoostParams(),
             bool update=false );

    virtual bool train( CvMLData* data,
             CvBoostParams params=CvBoostParams(),
             bool update=false );

    virtual float predict( const CvMat* sample, const CvMat* missing=0,
                           CvMat* weak_responses=0, CvSlice slice=CV_WHOLE_SEQ,
                           bool raw_mode=false, bool return_sum=false ) const;

    CV_WRAP CvBoost( const cv::Mat& trainData, int tflag,
            const cv::Mat& responses, const cv::Mat& varIdx=cv::Mat(),
            const cv::Mat& sampleIdx=cv::Mat(), const cv::Mat& varType=cv::Mat(),
            const cv::Mat& missingDataMask=cv::Mat(),
            CvBoostParams params=CvBoostParams() );

    CV_WRAP virtual bool train( const cv::Mat& trainData, int tflag,
                        const cv::Mat& responses, const cv::Mat& varIdx=cv::Mat(),
                        const cv::Mat& sampleIdx=cv::Mat(), const cv::Mat& varType=cv::Mat
(),
                        const cv::Mat& missingDataMask=cv::Mat(),
                        CvBoostParams params=CvBoostParams(),
                        bool update=false );

    CV_WRAP virtual float predict( const cv::Mat& sample, const cv::Mat& missing=cv::Mat(
),
                                   const cv::Range& slice=cv::Range::all(), bool rawMode=
false,
                                   bool returnSum=false ) const;

    virtual float calc_error( CvMLData* _data, int type , std::vector<float> *resp = 0 );
 // type in {CV_TRAIN_ERROR, CV_TEST_ERROR}

    CV_WRAP virtual void prune( CvSlice slice );

    CV_WRAP virtual void clear();

    virtual void write( CvFileStorage* storage, const char* name ) const;
    virtual void read( CvFileStorage* storage, CvFileNode* node );
    virtual const CvMat* get_active_vars(bool absolute_idx=true);
```

```
    CvSeq* get_weak_predictors();

    CvMat* get_weights();
    CvMat* get_subtree_weights();
    CvMat* get_weak_response();
    const CvBoostParams& get_params() const;
    const CvDTreeTrainData* get_data() const;

protected:

    virtual bool set_params( const CvBoostParams& params );
    virtual void update_weights( CvBoostTree* tree );
    virtual void trim_weights();
    virtual void write_params( CvFileStorage* fs ) const;
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    virtual void initialize_weights(double (&p)[2]);

    CvDTreeTrainData* data;
    CvMat train_data_hdr, responses_hdr;
    cv::Mat train_data_mat, responses_mat;
    CvBoostParams params;
    CvSeq* weak;

    CvMat* active_vars;
    CvMat* active_vars_abs;
    bool have_active_cat_vars;

    CvMat* orig_response;
    CvMat* sum_response;
    CvMat* weak_eval;
    CvMat* subsample_mask;
    CvMat* weights;
    CvMat* subtree_weights;
    bool have_subsample;
};


/****************************************************************************************\
*                                  Gradient Boosted Trees                                *
\****************************************************************************************/

// DataType: STRUCT CvGBTreesParams
// Parameters of GBT (Gradient Boosted trees model), including single
// tree settings and ensemble parameters.
//
// weak_count          - count of trees in the ensemble
// loss_function_type  - loss function used for ensemble training
// subsample_portion   - portion of whole training set used for
//                       every single tree training.
//                       subsample_portion value is in (0.0, 1.0].
//                       subsample_portion == 1.0 when whole dataset is
//                       used on each step. Count of sample used on each
//                       step is computed as
//                       int(total_samples_count * subsample_portion).
// shrinkage           - regularization parameter.
//                       Each tree prediction is multiplied on shrinkage value.


struct CV_EXPORTS_W_MAP CvGBTreesParams : public CvDTreeParams
{
    CV_PROP_RW int weak_count;
    CV_PROP_RW int loss_function_type;
    CV_PROP_RW float subsample_portion;
    CV_PROP_RW float shrinkage;

    CvGBTreesParams();
    CvGBTreesParams( int loss_function_type, int weak_count, float shrinkage,
        float subsample_portion, int max_depth, bool use_surrogates );
};

// DataType: CLASS CvGBTrees
// Gradient Boosting Trees (GBT) algorithm implementation.
//
// data             - training dataset
// params           - parameters of the CvGBTrees
// weak             - array[0..(class_count-1)] of CvSeq
//                    for storing tree ensembles
// orig_response    - original responses of the training set samples
// sum_response     - predicitons of the current model on the training dataset.
//                    this matrix is updated on every iteration.
// sum_response_tmp - predicitons of the model on the training set on the next
//                    step. On every iteration values of sum_responses_tmp are
//                    computed via sum_responses values. When the current
//                    step is complete sum_response values become equal to
//                    sum_responses_tmp.
// sampleIdx       - indices of samples used for training the ensemble.
//                    CvGBTrees training procedure takes a set of samples
//                    (train_data) and a set of responses (responses).
//                    Only pairs (train_data[i], responses[i]), where i is
//                    in sample_idx are used for training the ensemble.
// subsample_train  - indices of samples used for training a single decision
//                    tree on the current step. This indices are countered
//                    relatively to the sample_idx, so that pairs
//                    (train_data[sample_idx[i]], responses[sample_idx[i]])
//                    are used for training a decision tree.
//                    Training set is randomly splited
//                    in two parts (subsample_train and subsample_test)
//                    on every iteration accordingly to the portion parameter.
// subsample_test   - relative indices of samples from the training set,
//                    which are not used for training a tree on the current
//                    step.
// missing          - mask of the missing values in the training set. This
//                    matrix has the same size as train_data. 1 - missing
//                    value, 0 - not a missing value.
// class_labels     - output class labels map.
// rng              - random number generator. Used for spliting the
//                    training set.
// class_count      - count of output classes.
//                    class_count == 1 in the case of regression,
//                    and > 1 in the case of classification.
// delta            - Huber loss function parameter.
// base_value       - start point of the gradient descent procedure.
//                    model prediction is
//                    f(x) = f_0 + sum_{i=1..weak_count-1}(f_i(x)), where
//                    f_0 is the base value.



class CV_EXPORTS_W CvGBTrees : public CvStatModel
{
public:

    /*
    // DataType: ENUM
    // Loss functions implemented in CvGBTrees.
    //
    // SQUARED_LOSS
    // problem: regression
    // loss = (x - x')^2
```

```
    //
    // ABSOLUTE_LOSS
    // problem: regression
    // loss = abs(x - x')
    //
    // HUBER_LOSS
    // problem: regression
    // loss = delta*( abs(x - x') - delta/2), if abs(x - x') > delta
    //           1/2*(x - x')^2, if abs(x - x') <= delta,
    //           where delta is the alpha-quantile of pseudo responses from
    //           the training set.
    //
    // DEVIANCE_LOSS
    // problem: classification
    //
    */
    enum {SQUARED_LOSS=0, ABSOLUTE_LOSS, HUBER_LOSS=3, DEVIANCE_LOSS};


    /*
    // Default constructor. Creates a model only (without training).
    // Should be followed by one form of the train(...) function.
    //
    // API
    // CvGBTrees();

    // INPUT
    // OUTPUT
    // RESULT
    */
    CV_WRAP CvGBTrees();


    /*
    // Full form constructor. Creates a gradient boosting model and does the
    // train.
    //
    // API
    // CvGBTrees( const CvMat* trainData, int tflag,
            const CvMat* responses, const CvMat* varIdx=0,
            const CvMat* sampleIdx=0, const CvMat* varType=0,
            const CvMat* missingDataMask=0,
            CvGBTreesParams params=CvGBTreesParams() );

    // INPUT
    // trainData    - a set of input feature vectors.
    //                size of matrix is
    //                <count of samples> x <variables count>
    //                or <variables count> x <count of samples>
    //                depending on the tflag parameter.
    //                matrix values are float.
    // tflag         - a flag showing how do samples stored in the
    //                trainData matrix row by row (tflag=CV_ROW_SAMPLE)
    //                or column by column (tflag=CV_COL_SAMPLE).
    // responses     - a vector of responses corresponding to the samples
    //                in trainData.
    // varIdx       - indices of used variables. zero value means that all
    //                variables are active.
    // sampleIdx    - indices of used samples. zero value means that all
    //                samples from trainData are in the training set.
    // varType      - vector of <variables count> length. gives every
    //                variable type CV_VAR_CATEGORICAL or CV_VAR_ORDERED.
    //                varType = 0 means all variables are numerical.
    // missingDataMask  - a mask of misiing values in trainData.
    //                missingDataMask = 0 means that there are no missing
    //                values.
    // params         - parameters of GTB algorithm.
    // OUTPUT
    // RESULT
    */
    CvGBTrees( const CvMat* trainData, int tflag,
            const CvMat* responses, const CvMat* varIdx=0,
            const CvMat* sampleIdx=0, const CvMat* varType=0,
            const CvMat* missingDataMask=0,
            CvGBTreesParams params=CvGBTreesParams() );


    /*
    // Destructor.
    */
    virtual ~CvGBTrees();


    /*
    // Gradient tree boosting model training
    //
    // API
    // virtual bool train( const CvMat* trainData, int tflag,
            const CvMat* responses, const CvMat* varIdx=0,
            const CvMat* sampleIdx=0, const CvMat* varType=0,
            const CvMat* missingDataMask=0,
            CvGBTreesParams params=CvGBTreesParams(),
            bool update=false );

    // INPUT
    // trainData    - a set of input feature vectors.
    //                size of matrix is
    //                <count of samples> x <variables count>
    //                or <variables count> x <count of samples>
    //                depending on the tflag parameter.
    //                matrix values are float.
    // tflag         - a flag showing how do samples stored in the
    //                trainData matrix row by row (tflag=CV_ROW_SAMPLE)
    //                or column by column (tflag=CV_COL_SAMPLE).
    // responses     - a vector of responses corresponding to the samples
    //                in trainData.
    // varIdx       - indices of used variables. zero value means that all
    //                variables are active.
    // sampleIdx    - indices of used samples. zero value means that all
    //                samples from trainData are in the training set.
    // varType      - vector of <variables count> length. gives every
    //                variable type CV_VAR_CATEGORICAL or CV_VAR_ORDERED.
    //                varType = 0 means all variables are numerical.
    // missingDataMask  - a mask of misiing values in trainData.
    //                missingDataMask = 0 means that there are no missing
    //                values.
    // params         - parameters of GTB algorithm.
    // update         - is not supported now. (!)
    // OUTPUT
    // RESULT
    // Error state.
    */
    virtual bool train( const CvMat* trainData, int tflag,
            const CvMat* responses, const CvMat* varIdx=0,
            const CvMat* sampleIdx=0, const CvMat* varType=0,
            const CvMat* missingDataMask=0,
            CvGBTreesParams params=CvGBTreesParams(),
            bool update=false );


    /*
    // Gradient tree boosting model training
    //
    // API
```

```
    // virtual bool train( CvMLData* data,
            CvGBTreesParams params=CvGBTreesParams(),
            bool update=false ) {return false;}

    // INPUT
    // data          - training set.
    // params        - parameters of GTB algorithm.
    // update        - is not supported now. (!)
    // OUTPUT
    // RESULT
    // Error state.
    */
    virtual bool train( CvMLData* data,
            CvGBTreesParams params=CvGBTreesParams(),
            bool update=false );


    /*
    // Response value prediction
    //
    // API
    // virtual float predict_serial( const CvMat* sample, const CvMat* missing=0,
            CvMat* weak_responses=0, CvSlice slice = CV_WHOLE_SEQ,
            int k=-1 ) const;

    // INPUT
    // sample         - input sample of the same type as in the training set.
    // missing        - missing values mask. missing=0 if there are no
    //                  missing values in sample vector.
    // weak_responses  - predictions of all of the trees.
    //                  not implemented (!)
    // slice           - part of the ensemble used for prediction.
    //                  slice = CV_WHOLE_SEQ when all trees are used.
    // k               - number of ensemble used.
    //                  k is in {-1,0,1,..,<count of output classes-1>}.
    //                  in the case of classification problem
    //                  <count of output classes-1> ensembles are built.
    //                  If k = -1 ordinary prediction is the result,
    //                  otherwise function gives the prediction of the
    //                  k-th ensemble only.
    // OUTPUT
    // RESULT
    // Predicted value.
    */
    virtual float predict_serial( const CvMat* sample, const CvMat* missing=0,
            CvMat* weakResponses=0, CvSlice slice = CV_WHOLE_SEQ,
            int k=-1 ) const;

    /*
    // Response value prediction.
    // Parallel version (in the case of TBB existence)
    //
    // API
    // virtual float predict( const CvMat* sample, const CvMat* missing=0,
            CvMat* weak_responses=0, CvSlice slice = CV_WHOLE_SEQ,
            int k=-1 ) const;

    // INPUT
    // sample         - input sample of the same type as in the training set.
    // missing        - missing values mask. missing=0 if there are no
    //                  missing values in sample vector.
    // weak_responses  - predictions of all of the trees.
    //                  not implemented (!)
    // slice           - part of the ensemble used for prediction.
    //                  slice = CV_WHOLE_SEQ when all trees are used.
    // k               - number of ensemble used.
    //                  k is in {-1,0,1,..,<count of output classes-1>}.
    //                  in the case of classification problem
    //                  <count of output classes-1> ensembles are built.
    //                  If k = -1 ordinary prediction is the result,
    //                  otherwise function gives the prediction of the
    //                  k-th ensemble only.
    // OUTPUT
    // RESULT
    // Predicted value.
    */
    virtual float predict( const CvMat* sample, const CvMat* missing=0,
            CvMat* weakResponses=0, CvSlice slice = CV_WHOLE_SEQ,
            int k=-1 ) const;

    /*
    // Deletes all the data.
    //
    // API
    // virtual void clear();

    // INPUT
    // OUTPUT
    // delete data, weak, orig_response, sum_response,
    //        weak_eval, subsample_train, subsample_test,
    //        sample_idx, missing, lass_labels
    // delta = 0.0
    // RESULT
    */
    CV_WRAP virtual void clear();

    /*
    // Compute error on the train/test set.
    //
    // API
    // virtual float calc_error( CvMLData* _data, int type,
    //        std::vector<float> *resp = 0 );
    //
    // INPUT
    // data  - dataset
    // type  - defines which error is to compute: train (CV_TRAIN_ERROR) or
    //         test (CV_TEST_ERROR).
    // OUTPUT
    // resp  - vector of predicitons
    // RESULT
    // Error value.
    */
    virtual float calc_error( CvMLData* _data, int type,
            std::vector<float> *resp = 0 );

    /*
    //
    // Write parameters of the gtb model and data. Write learned model.
    //
    // API
    // virtual void write( CvFileStorage* fs, const char* name ) const;
    //
    // INPUT
    // fs     - file storage to read parameters from.
    // name   - model name.
    // OUTPUT
    // RESULT
    */
    virtual void write( CvFileStorage* fs, const char* name ) const;


    /*
    //
    // Read parameters of the gtb model and data. Read learned model.
```

```
    //
    // API
    // virtual void read( CvFileStorage* fs, CvFileNode* node );
    //
    // INPUT
    // fs     - file storage to read parameters from.
    // node   - file node.
    // OUTPUT
    // RESULT
    */
    virtual void read( CvFileStorage* fs, CvFileNode* node );


    // new-style C++ interface
    CV_WRAP CvGBTrees( const cv::Mat& trainData, int tflag,
              const cv::Mat& responses, const cv::Mat& varIdx=cv::Mat(),
              const cv::Mat& sampleIdx=cv::Mat(), const cv::Mat& varType=cv::Mat(),
              const cv::Mat& missingDataMask=cv::Mat(),
              CvGBTreesParams params=CvGBTreesParams() );

    CV_WRAP virtual bool train( const cv::Mat& trainData, int tflag,
                        const cv::Mat& responses, const cv::Mat& varIdx=cv::Mat(),
                        const cv::Mat& sampleIdx=cv::Mat(), const cv::Mat& varType=cv::Mat
(),
                        const cv::Mat& missingDataMask=cv::Mat(),
                        CvGBTreesParams params=CvGBTreesParams(),
                        bool update=false );

    CV_WRAP virtual float predict( const cv::Mat& sample, const cv::Mat& missing=cv::Mat(
),
                            const cv::Range& slice = cv::Range::all(),
                            int k=-1 ) const;

protected:

    /*
    // Compute the gradient vector components.
    //
    // API
    // virtual void find_gradient( const int k = 0);

    // INPUT
    // k        - used for classification problem, determining current
    //            tree ensemble.
    // OUTPUT
    // changes components of data->responses
    // which correspond to samples used for training
    // on the current step.
    // RESULT
    */
    virtual void find_gradient( const int k = 0);


    /*
    //
    // Change values in tree leaves according to the used loss function.
    //
    // API
    // virtual void change_values(CvDTree* tree, const int k = 0);
    //
    // INPUT
    // tree      - decision tree to change.
    // k         - used for classification problem, determining current
    //             tree ensemble.
    // OUTPUT
    // changes 'value' fields of the trees' leaves.
    // changes sum_response_tmp.
    // RESULT
    */
    virtual void change_values(CvDTree* tree, const int k = 0);


    /*
    //
    // Find optimal constant prediction value according to the used loss
    // function.
    // The goal is to find a constant which gives the minimal summary loss
    // on the _Idx samples.
    //
    // API
    // virtual float find_optimal_value( const CvMat* _Idx );
    //
    // INPUT
    // _Idx        - indices of the samples from the training set.
    // OUTPUT
    // RESULT
    // optimal constant value.
    */
    virtual float find_optimal_value( const CvMat* _Idx );


    /*
    //
    // Randomly split the whole training set in two parts according
    // to params.portion.
    //
    // API
    // virtual void do_subsample();
    //
    // INPUT
    // OUTPUT
    // subsample_train - indices of samples used for training
    // subsample_test  - indices of samples used for test
    // RESULT
    */
    virtual void do_subsample();


    /*
    //
    // Internal recursive function giving an array of subtree tree leaves.
    //
    // API
    // void leaves_get( CvDTreeNode** leaves, int& count, CvDTreeNode* node );
    //
    // INPUT
    // node         - current leaf.
    // OUTPUT
    // count        - count of leaves in the subtree.
    // leaves       - array of pointers to leaves.
    // RESULT
    */
    void leaves_get( CvDTreeNode** leaves, int& count, CvDTreeNode* node );


    /*
    //
    // Get leaves of the tree.
    //
    // API
    // CvDTreeNode** GetLeaves( const CvDTree* dtree, int& len );
    //
    // INPUT
    // dtree            - decision tree.
```

```
    // OUTPUT
    // len              - count of the leaves.
    // RESULT
    // CvDTreeNode**    - array of pointers to leaves.
    */
    CvDTreeNode** GetLeaves( const CvDTree* dtree, int& len );


    /*
    //
    // Is it a regression or a classification.
    //
    // API
    // bool problem_type();
    //
    // INPUT
    // OUTPUT
    // RESULT
    // false if it is a classification problem,
    // true - if regression.
    */
    virtual bool problem_type() const;


    /*
    //
    // Write parameters of the gtb model.
    //
    // API
    // virtual void write_params( CvFileStorage* fs ) const;
    //
    // INPUT
    // fs           - file storage to write parameters to.
    // OUTPUT
    // RESULT
    */
    virtual void write_params( CvFileStorage* fs ) const;


    /*
    //
    // Read parameters of the gtb model and data.
    //
    // API
    // virtual void read_params( CvFileStorage* fs );
    //
    // INPUT
    // fs           - file storage to read parameters from.
    // OUTPUT
    // params       - parameters of the gtb model.
    // data         - contains information about the structure
    //                of the data set (count of variables,
    //                their types, etc.).
    // class_labels - output class labels map.
    // RESULT
    */
    virtual void read_params( CvFileStorage* fs, CvFileNode* fnode );
    int get_len(const CvMat* mat) const;


    CvDTreeTrainData* data;
    CvGBTreesParams params;

    CvSeq** weak;
    CvMat* orig_response;
    CvMat* sum_response;
    CvMat* sum_response_tmp;
    CvMat* sample_idx;
    CvMat* subsample_train;
    CvMat* subsample_test;
    CvMat* missing;
    CvMat* class_labels;

    cv::RNG* rng;

    int class_count;
    float delta;
    float base_value;

};



/****************************************************************************************\
*                                Artificial Neural Networks (ANN)                        *
\****************************************************************************************/

/////////////////////////////////// Multi-Layer Perceptrons //////////////////////////////

struct CV_EXPORTS_W_MAP CvANN_MLP_TrainParams
{
    CvANN_MLP_TrainParams();
    CvANN_MLP_TrainParams( CvTermCriteria term_crit, int train_method,
                           double param1, double param2=0 );
    ~CvANN_MLP_TrainParams();

    enum { BACKPROP=0, RPROP=1 };

    CV_PROP_RW CvTermCriteria term_crit;
    CV_PROP_RW int train_method;

    // backpropagation parameters
    CV_PROP_RW double bp_dw_scale, bp_moment_scale;

    // rprop parameters
    CV_PROP_RW double rp_dw0, rp_dw_plus, rp_dw_minus, rp_dw_min, rp_dw_max;
};


class CV_EXPORTS_W CvANN_MLP : public CvStatModel
{
public:
    CV_WRAP CvANN_MLP();
    CvANN_MLP( const CvMat* layerSizes,
               int activateFunc=CvANN_MLP::SIGMOID_SYM,
               double fparam1=0, double fparam2=0 );

    virtual ~CvANN_MLP();

    virtual void create( const CvMat* layerSizes,
                         int activateFunc=CvANN_MLP::SIGMOID_SYM,
                         double fparam1=0, double fparam2=0 );

    virtual int train( const CvMat* inputs, const CvMat* outputs,
                       const CvMat* sampleWeights, const CvMat* sampleIdx=0,
                       CvANN_MLP_TrainParams params = CvANN_MLP_TrainParams(),
                       int flags=0 );
    virtual float predict( const CvMat* inputs, CV_OUT CvMat* outputs ) const;

    CV_WRAP CvANN_MLP( const cv::Mat& layerSizes,
```

```
            int activateFunc=CvANN_MLP::SIGMOID_SYM,
            double fparam1=0, double fparam2=0 );

    CV_WRAP virtual void create( const cv::Mat& layerSizes,
                     int activateFunc=CvANN_MLP::SIGMOID_SYM,
                     double fparam1=0, double fparam2=0 );

    CV_WRAP virtual int train( const cv::Mat& inputs, const cv::Mat& outputs,
                    const cv::Mat& sampleWeights, const cv::Mat& sampleIdx=cv::Mat(),
                    CvANN_MLP_TrainParams params = CvANN_MLP_TrainParams(),
                    int flags=0 );

    CV_WRAP virtual float predict( const cv::Mat& inputs, CV_OUT cv::Mat& outputs ) const
;

    CV_WRAP virtual void clear();

    // possible activation functions
    enum { IDENTITY = 0, SIGMOID_SYM = 1, GAUSSIAN = 2 };

    // available training flags
    enum { UPDATE_WEIGHTS = 1, NO_INPUT_SCALE = 2, NO_OUTPUT_SCALE = 4 };

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* storage, const char* name ) const;

    int get_layer_count() { return layer_sizes ? layer_sizes->cols : 0; }
    const CvMat* get_layer_sizes() { return layer_sizes; }
    double* get_weights(int layer)
    {
        return layer_sizes && weights &&
            (unsigned)layer <= (unsigned)layer_sizes->cols ? weights[layer] : 0;
    }

    virtual void calc_activ_func_deriv( CvMat* xf, CvMat* deriv, const double* bias ) con
st;

protected:

    virtual bool prepare_to_train( const CvMat* _inputs, const CvMat* _outputs,
            const CvMat* _sample_weights, const CvMat* sampleIdx,
            CvVectors* _ivecs, CvVectors* _ovecs, double** _sw, int _flags );

    // sequential random backpropagation
    virtual int train_backprop( CvVectors _ivecs, CvVectors _ovecs, const double* _sw );

    // RPROP algorithm
    virtual int train_rprop( CvVectors _ivecs, CvVectors _ovecs, const double* _sw );

    virtual void calc_activ_func( CvMat* xf, const double* bias ) const;
    virtual void set_activ_func( int _activ_func=SIGMOID_SYM,
                                 double _f_param1=0, double _f_param2=0 );
    virtual void init_weights();
    virtual void scale_input( const CvMat* _src, CvMat* _dst ) const;
    virtual void scale_output( const CvMat* _src, CvMat* _dst ) const;
    virtual void calc_input_scale( const CvVectors* vecs, int flags );
    virtual void calc_output_scale( const CvVectors* vecs, int flags );

    virtual void write_params( CvFileStorage* fs ) const;
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    CvMat* layer_sizes;
    CvMat* wbuf;
    CvMat* sample_weights;
    double** weights;
    double f_param1, f_param2;
    double min_val, max_val, min_val1, max_val1;
    int activ_func;
    int max_count, max_buf_sz;
    CvANN_MLP_TrainParams params;
    cv::RNG* rng;
};

/****************************************************************************************\
*                            Auxilary functions declarations                             *
\****************************************************************************************/

/* Generates <sample> from multivariate normal distribution, where <mean> - is an
   average row vector, <cov> - symmetric covariation matrix */
CVAPI(void) cvRandMVNormal( CvMat* mean, CvMat* cov, CvMat* sample,
                            CvRNG* rng CV_DEFAULT(0) );

/* Generates sample from gaussian mixture distribution */
CVAPI(void) cvRandGaussMixture( CvMat* means[],
                                CvMat* covs[],
                                float weights[],
                                int clsnum,
                                CvMat* sample,
                                CvMat* sampClasses CV_DEFAULT(0) );

#define CV_TS_CONCENTRIC_SPHERES 0

/* creates test set */
CVAPI(void) cvCreateTestSet( int type, CvMat** samples,
                 int num_samples,
                 int num_features,
                 CvMat** responses,
                 int num_classes, ... );

/****************************************************************************************\
*                                        Data                                          *
\****************************************************************************************/

#define CV_COUNT     0
#define CV_PORTION   1

struct CV_EXPORTS CvTrainTestSplit
{
    CvTrainTestSplit();
    CvTrainTestSplit( int train_sample_count, bool mix = true);
    CvTrainTestSplit( float train_sample_portion, bool mix = true);

    union
    {
        int count;
        float portion;
    } train_sample_part;
    int train_sample_part_mode;

    bool mix;
};

class CV_EXPORTS CvMLData
{
public:
    CvMLData();
    virtual ~CvMLData();

    // returns:
```

```
    // 0 - OK
    // -1 - file can not be opened or is not correct
    int read_csv( const char* filename );

    const CvMat* get_values() const;
    const CvMat* get_responses();
    const CvMat* get_missing() const;

    void set_header_lines_number( int n );
    int get_header_lines_number() const;

    void set_response_idx( int idx ); // old response become predictors, new response_idx
 = idx
                                      // if idx < 0 there will be no response
    int get_response_idx() const;

    void set_train_test_split( const CvTrainTestSplit * spl );
    const CvMat* get_train_sample_idx() const;
    const CvMat* get_test_sample_idx() const;
    void mix_train_and_test_idx();

    const CvMat* get_var_idx();
    void chahge_var_idx( int vi, bool state ); // misspelled (saved for back compitabilit
y),
                                               // use change_var_idx
    void change_var_idx( int vi, bool state ); // state == true to set vi-variable as pre
dictor

    const CvMat* get_var_types();
    int get_var_type( int var_idx ) const;
    // following 2 methods enable to change vars type
    // use these methods to assign CV_VAR_CATEGORICAL type for categorical variable
    // with numerical labels; in the other cases var types are correctly determined autom
atically
    void set_var_types( const char* str );  // str examples:
                                            // "ord[0-17],cat[18]", "ord[0,2,4,10-12], ca
t[1,3,5-9,13,14]",
                                            // "cat", "ord" (all vars are categorical/ord
ered)
    void change_var_type( int var_idx, int type); // type in { CV_VAR_ORDERED, CV_VAR_CAT
EGORICAL }

    void set_delimiter( char ch );
    char get_delimiter() const;

    void set_miss_ch( char ch );
    char get_miss_ch() const;

    const std::map<cv::String, int>& get_class_labels_map() const;

protected:
    virtual void clear();

    void str_to_flt_elem( const char* token, float& flt_elem, int& type);
    void free_train_test_idx();

    char delimiter;
    char miss_ch;
    //char flt_separator;

    CvMat* values;
    CvMat* missing;
    CvMat* var_types;
    CvMat* var_idx_mask;

    CvMat* response_out; // header
    CvMat* var_idx_out; // mat
    CvMat* var_types_out; // mat

    int header_lines_number;

    int response_idx;

    int train_sample_count;
    bool mix;

    int total_class_count;
    std::map<cv::String, int> class_map;

    CvMat* train_sample_idx;
    CvMat* test_sample_idx;
    int* sample_idx; // data of train_sample_idx and test_sample_idx

    cv::RNG* rng;
};


namespace cv
{

typedef CvStatModel StatModel;
typedef CvParamGrid ParamGrid;
typedef CvNormalBayesClassifier NormalBayesClassifier;
typedef CvKNearest KNearest;
typedef CvSVMParams SVMParams;
typedef CvSVMKernel SVMKernel;
typedef CvSVMSolver SVMSolver;
typedef CvSVM SVM;
typedef CvDTreeParams DTreeParams;
typedef CvMLData TrainData;
typedef CvDTree DecisionTree;
typedef CvForestTree ForestTree;
typedef CvRTParams RandomTreeParams;
typedef CvRTrees RandomTrees;
typedef CvERTreeTrainData ERTreeTRainData;
typedef CvForestERTree ERTree;
typedef CvERTrees ERTrees;
typedef CvBoostParams BoostParams;
typedef CvBoostTree BoostTree;
typedef CvBoost Boost;
typedef CvANN_MLP_TrainParams ANN_MLP_TrainParams;
typedef CvANN_MLP NeuralNet_MLP;
typedef CvGBTreesParams GradientBoostingTreeParams;
typedef CvGBTrees GradientBoostingTrees;

template<> CV_EXPORTS void DefaultDeleter<CvDTreeSplit>::operator ()(CvDTreeSplit* obj) c
onst;

CV_EXPORTS bool initModule_ml(void);
}

#endif // __cplusplus
#endif // __OPENCV_ML_HPP__

/* End of file. */
```