Final Project
CS634
Fall 2021

Option-1
Supervised Data Mining (Classification)

Prepared by:

First Name:	Md Rakibul
Last Name:	Hasan
NJIT UCID:	mh629
Email:	mh629@njit.edu
Date:	11/20/2021

Table of Contents

Project Proposal Brief:

Project option number: Option 1
Project option name: Supervised Data Mining (Classification)
Algorithms to be used: Category 1 (Support Vector Machines) and Category 5 (Naive Bayes)
Programming Language: Category 10 (Python)
Library Tool: Scikit-learn
IDE: Google Colab
Data to be used in the project: UCI Machine Learning Repository: Census Income Data Set (Links to an external site.)
OS: macOS Monterey
Hardware: MacBook Pro (13-inch, Apple M1 chip, 2020)

Dataset Description:

This dataset has been extracted from 1994 Census dataset by Barry Becker. A reasonably clean records were extracted from the original Census data, and it is known as 'Adult' dataset. I have to predict whether a person makes over 50K a year or not using this dataset.

Title: Census Income Dataset
Data Set Characteristics: Multivariate
Number of Instances: 32561
Attribute Characteristics: Categorical, Integer
Number of feature Attributes: 15
Classes: 2

Attribute types and details are given below:

age: continuous.
workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
fnlwgt: continuous.
education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
education-num: continuous.
marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
sex: Female, Male.
capital-gain: continuous.

capital-loss: continuous.
hours-per-week: continuous.
native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands.

Reading and Pre-processing Dataset:

I have used Pandas library to read data from the URL ([UCI Machine Learning Repository: Census Income Data Set (Links to an external site.)]()). I have checked the total number of samples and attributes from the dataset.

```
[1] import pandas as pd
```

```
[3] data=pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data',
                     names=['age','workclass','fnlwgt','education','education-num','marital-status',
                            'occupation','relationship','race','sex','capital-gain','capital-loss',
                            'hours-per-week','native-country','income'])
    data.head()
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

```
[4] print("The Census Income Data Set has {0[0]} samples and {0[1]} feature attributes".format(data.shape))

The Census Income Data Set has 32561 samples and 15 feature attributes
```

Later, I have used LabelEncoder to convert string attributes to the numerical attributes. Then, I have separated the feature attributes and label attribute. ‘income’ is the label attribute, and I must predict that.

Since, I must use this dataset for training as well as testing, I have taken 75% of the dataset to train our classifier model and rest 25% will be used to test our model.

```python
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import svm
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.metrics import roc_curve, auc
from sklearn.datasets import make_multilabel_classification
from sklearn.multioutput import MultiOutputClassifier
```

```python
[7] le=LabelEncoder()
    attributes=['workclass','education','marital-status','occupation','relationship','race','sex','native-country','income']
    data[attributes] = data[attributes].apply(le.fit_transform)
```

```python
[8] attr = data.iloc[:, :14]
    lebel= data.iloc[:, 14]
    lebel.head()
```

```
0    0
1    0
2    0
3    0
4    0
Name: income, dtype: int64
```

```python
[9] X_train,X_test, Y_train, Y_test = train_test_split(attr, lebel, test_size= 0.25,random_state=0)
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.fit_transform(X_test)
```

Implementing Support Vector Machine (SVM) Classifier [Category 1]:

SVM stands for Support Vector Machine. SVM is a supervised machine learning algorithm that is commonly used for classification and regression challenges. Common applications of the SVM algorithm are Intrusion Detection System, Handwriting Recognition, Protein Structure Prediction, Detecting Steganography in digital images, etc.

SVM works by mapping data to a high-dimensional feature space so that data points can be categorized, even when the data are not otherwise linearly separable. A separator between the categories is found, then the data are transformed in such a way that the separator could be drawn as a hyperplane.

Support Vector machines have some special data points which we call "Support Vectors" and a separating hyperplane which is known as "Support Vector Machine". So, essentially SVM is a frontier that best segregates the classes. Support Vectors are the data points nearest to the hyperplane, the points of our data set which if removed, would alter the position of the dividing hyperplane. As we can see that there can be many hyperplanes which can segregate the two classes, the hyperplane that we would choose is the one with the highest margin.

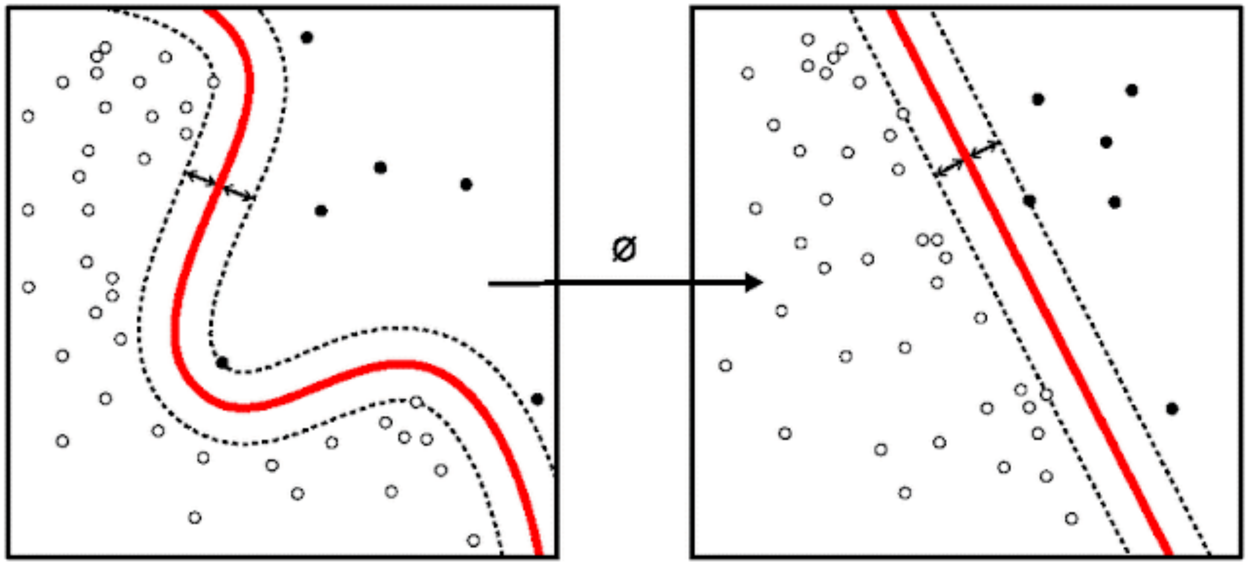


Figure: Different Hyperplanes

Image_source(https://medium.com/analytics-vidhya/implementing-svm-for-performing-classification-and-finding-accuracy-in-python-using-datasets-wine-e4fef8e804b4)

The mathematical function used for the transformation is known as the kernel function. SVM supports the following kernel types:

• Linear
• Polynomial
• Radial basis function (RBF)
• Sigmoid

Here, I have used all 4 kernels to train and test our dataset. Among these kernels, Radial basis function (RBF) could provide 80% accuracy after 10-fold cross-validation. With Sigmoid kernel, 10-fold cross-validation is 65%.

```
[13] clf_svm_rbf = svm.SVC(kernel='rbf')
     clf_svm_rbf.fit(X_train, Y_train)
     Y_prediction = clf_svm_rbf.predict(X_test)
     clf_svm_rbf = cross_val_score(clf_svm_rbf,attr,lebel,cv=10)
     print("%0.2f accuracy with a standard deviation of %0.2f" % (clf_svm_rbf.mean(), clf_svm_rbf.std()))

     0.80 accuracy with a standard deviation of 0.00
```

```
[9]  clf_svm_linear = svm.SVC(kernel='linear', C=1)
     clf_svm_linear.fit(X_train, Y_train)
     Y_prediction = clf_svm_linear.predict(X_test)
     result_svm_linear = cross_val_score(clf_svm_linear,attr,lebel,cv=10)
     print("%0.2f accuracy with a standard deviation of %0.2f" % (result_svm_linear.mean(), result_svm_linear.std()))
```

```
[10] clf_svm_poly = svm.SVC(kernel='poly')
     clf_svm_poly.fit(X_train, Y_train)
     Y_prediction = clf_svm_poly.predict(X_test)
     result_svm_poly = cross_val_score(clf_svm_poly,attr,lebel,cv=10)
     print("%0.2f accuracy with a standard deviation of %0.2f" % (result_svm_poly.mean(), result_svm_poly.std()))
```

```
[13] clf_svm_sig = svm.SVC(kernel='sigmoid')
     clf_svm_sig.fit(X_train, Y_train)
     Y_prediction = clf_svm_sig.predict(X_test)
     result_svm_sig = cross_val_score(clf_svm_sig,attr,lebel,cv=10)
     print("%0.2f accuracy with a standard deviation of %0.2f" % (result_svm_sig.mean(), result_svm_sig.std()))

     0.65 accuracy with a standard deviation of 0.01
```

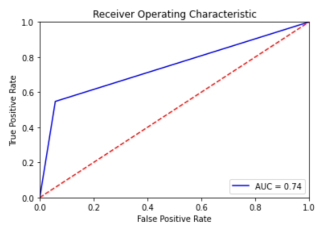
Here the confusion matrix has been shown as well as other parameters like precision, recall and f1-score. Area under curve is 74% for RBF kernel.

```
print(confusion_matrix(Y_test,Y_prediction))
print(classification_report(Y_test, Y_prediction))
fpr, tpr, threshold = metrics.roc_curve(Y_test, Y_prediction)
roc_auc = metrics.auc(fpr, tpr)
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
[[5804  355]
 [ 897 1085]]
              precision    recall  f1-score   support

           0       0.87      0.94      0.90      6159
           1       0.75      0.55      0.63      1982

    accuracy                           0.85      8141
   macro avg       0.81      0.74      0.77      8141
weighted avg       0.84      0.85      0.84      8141
```



Implementing Naïve Bayes Classifier [Category 5]:

Naive Bayes is a statistical classification technique based on Bayes Theorem. It is one of the simplest supervised learning algorithms. Naive Bayes classifier is a fast, accurate, and reliable algorithm. Naive Bayes classifiers have high accuracy and speed on large datasets. Naive Bayes classifier assumes that the effect of a particular feature in a class is independent of other features. For example, a loan applicant is desirable or not depending on his/her income, previous loan and transaction history, age, and location. Even if these features are interdependent, these features are still considered independently. This assumption simplifies computation, and that's why it is considered as naive. This assumption is called class conditional independence.

$$P(h|D) = \frac{\left(P(D|h) * P(h)\right)}{P(D)}$$

Here,

P(h): the probability of hypothesis h being true (regardless of the data). This is known as the prior probability of h.

P(D): the probability of the data (regardless of the hypothesis). This is known as the prior probability.

P(h|D): the probability of hypothesis h given the data D. This is known as posterior probability.

P(D|h): the probability of data d given that the hypothesis h was true. This is known as the posterior probability.

Here using sklearn library, Naïve Bayes algorithm has been implemented for classification for census dataset. The dataset was trained with 10-fold cross validation and the test dataset has achieved 80% accuracy.

Naive Bayes Classifier

```
[9] from sklearn.naive_bayes import GaussianNB
    clf_GNB = GaussianNB()
    clf_GNB.fit(X_train, Y_train)
    Y_prediction_GNB = clf_GNB.predict(X_test)
    result_clf_GNB = cross_val_score(clf_GNB,attr,lebel,cv=10)
    print("%0.2f accuracy with a standard deviation of %0.2f" % (result_clf_GNB.mean(), result_clf_GNB.std()))

    0.80 accuracy with a standard deviation of 0.01
```

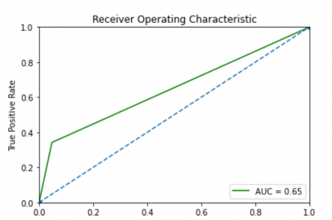
Here the confusion matrix has been shown as well as other parameters like precision, recall and f1-score. Area under curve is 65% for Naïve Bayes Classifier.

```
print(confusion_matrix(Y_test,Y_prediction_GNB))
print(classification_report(Y_test, Y_prediction_GNB))
fpr, tpr, threshold = metrics.roc_curve(Y_test, Y_prediction_GNB)
roc_auc = metrics.auc(fpr, tpr)
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'g', label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'o--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
[[5866  293]
 [1304  678]]
              precision    recall  f1-score   support

           0       0.82      0.95      0.88      6159
           1       0.70      0.34      0.46      1982

    accuracy                           0.80      8141
   macro avg       0.76      0.65      0.67      8141
weighted avg       0.79      0.80      0.78      8141
```



Conclusion:

For Census Income Dataset, SVM performed better than Naïve Bayes because the AUC of SVM with RBF kernel is greater than the AUC of Naïve Bayes. But training SVM with ‘Poly’ and ‘sigmod’ kernel takes longer time due to using too many training dataset

Appendix:

SVM Source Code:

URL: https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/svm/src/libsvm/svm.cpp

```
/*
Copyright (c) 2000-2009 Chih-Chung Chang and Chih-Jen Lin
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither name of copyright holders nor the names of its contributors
may be used to endorse or promote products derived from this software
without specific prior written permission.


THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/*
  Modified 2010:

  - Support for dense data by Ming-Fang Weng
```

```
   - Return indices for support vectors, Fabian Pedregosa
     <fabian.pedregosa@inria.fr>

   - Fixes to avoid name collision, Fabian Pedregosa

   - Add support for instance weights, Fabian Pedregosa based on work
     by Ming-Wei Chang, Hsuan-Tien Lin, Ming-Hen Tsai, Chia-Hua Ho and
     Hsiang-Fu Yu,
     <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/#weights_for_data_instances>.

   - Make labels sorted in svm_group_classes, Fabian Pedregosa.

   Modified 2020:

   - Improved random number generator by using a mersenne twister + tweaked
     lemire postprocessor. This fixed a convergence issue on windows targets.
     Sylvain Marie, Schneider Electric
     see <https://github.com/scikit-learn/scikit-learn/pull/13511#issuecomment-481729756>

 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <float.h>
#include <string.h>
#include <stdarg.h>
#include <climits>
#include <random>
#include "svm.h"
#include "_svm_cython_blas_helpers.h"
#include "../newrand/newrand.h"


#ifndef _LIBSVM_CPP
typedef float Qfloat;
typedef signed char schar;
#ifndef min
template <class T> static inline T min(T x,T y) { return (x<y)?x:y; }
#endif
#ifndef max
template <class T> static inline T max(T x,T y) { return (x>y)?x:y; }
#endif
```

```
template <class T> static inline void swap(T& x, T& y) { T t=x; x=y; y=t; }
template <class S, class T> static inline void clone(T*& dst, S* src, int n)
{
   dst = new T[n];
   memcpy((void *)dst,(void *)src,sizeof(T)*n);
}
static inline double powi(double base, int times)
{
   double tmp = base, ret = 1.0;

   for(int t=times; t>0; t/=2)
   {
      if(t%2==1) ret*=tmp;
      tmp = tmp * tmp;
   }
   return ret;
}
#define INF HUGE_VAL
#define TAU 1e-12
#define Malloc(type,n) (type *)malloc((n)*sizeof(type))

static void print_string_stdout(const char *s)
{
   fputs(s,stdout);
   fflush(stdout);
}
static void (*svm_print_string) (const char *) = &print_string_stdout;

static void info(const char *fmt,...)
{
   char buf[BUFSIZ];
   va_list ap;
   va_start(ap,fmt);
   vsprintf(buf,fmt,ap);
   va_end(ap);
   (*svm_print_string)(buf);
}
#endif
#define _LIBSVM_CPP


/* yeah, this is ugly.  It helps us to have unique names for both sparse
and dense versions of this library */
#ifdef _DENSE_REP
```

```
  #ifdef PREFIX
   #undef PREFIX
  #endif
  #ifdef NAMESPACE
   #undef NAMESPACE
  #endif
  #define PREFIX(name) svm_##name
  #define NAMESPACE svm
  namespace svm {
#else
  /* sparse representation */
  #ifdef PREFIX
   #undef PREFIX
  #endif
  #ifdef NAMESPACE
   #undef NAMESPACE
  #endif
  #define PREFIX(name) svm_csr_##name
  #define NAMESPACE svm_csr
  namespace svm_csr {
#endif


//
// Kernel Cache
//
// l is the number of total data items
// size is the cache size limit in bytes
//
class Cache
{
public:
   Cache(int l,long int size);
   ~Cache();

   // request data [0,len)
   // return some position p where [p,len) need to be filled
   // (p >= len if nothing needs to be filled)
   int get_data(const int index, Qfloat **data, int len);
   void swap_index(int i, int j);
private:
   int l;
   long int size;
   struct head_t
```

```
   {
      head_t *prev, *next;    // a circular list
      Qfloat *data;
      int len;        // data[0,len) is cached in this entry
   };

   head_t *head;
   head_t lru_head;
   void lru_delete(head_t *h);
   void lru_insert(head_t *h);
};

Cache::Cache(int l_,long int size_):l(l_),size(size_)
{
   head = (head_t *)calloc(l,sizeof(head_t));  // initialized to 0
   size /= sizeof(Qfloat);
   size -= l * sizeof(head_t) / sizeof(Qfloat);
   size = max(size, 2 * (long int) l); // cache must be large enough for two columns
   lru_head.next = lru_head.prev = &lru_head;
}

Cache::~Cache()
{
   for(head_t *h = lru_head.next; h != &lru_head; h=h->next)
      free(h->data);
   free(head);
}

void Cache::lru_delete(head_t *h)
{
   // delete from current location
   h->prev->next = h->next;
   h->next->prev = h->prev;
}

void Cache::lru_insert(head_t *h)
{
   // insert to last position
   h->next = &lru_head;
   h->prev = lru_head.prev;
   h->prev->next = h;
   h->next->prev = h;
}
```

```
int Cache::get_data(const int index, Qfloat **data, int len)
{
   head_t *h = &head[index];
   if(h->len) lru_delete(h);
   int more = len - h->len;

   if(more > 0)
   {
      // free old space
      while(size < more)
      {
         head_t *old = lru_head.next;
         lru_delete(old);
         free(old->data);
         size += old->len;
         old->data = 0;
         old->len = 0;
      }

      // allocate new space
      h->data = (Qfloat *)realloc(h->data,sizeof(Qfloat)*len);
      size -= more;
      swap(h->len,len);
   }

   lru_insert(h);
   *data = h->data;
   return len;
}

void Cache::swap_index(int i, int j)
{
   if(i==j) return;

   if(head[i].len) lru_delete(&head[i]);
   if(head[j].len) lru_delete(&head[j]);
   swap(head[i].data,head[j].data);
   swap(head[i].len,head[j].len);
   if(head[i].len) lru_insert(&head[i]);
   if(head[j].len) lru_insert(&head[j]);

   if(i>j) swap(i,j);
   for(head_t *h = lru_head.next; h!=&lru_head; h=h->next)
   {
```

```
        if(h->len > i)
        {
            if(h->len > j)
                swap(h->data[i],h->data[j]);
            else
            {
                // give up
                lru_delete(h);
                free(h->data);
                size += h->len;
                h->data = 0;
                h->len = 0;
            }
        }
    }
}

//
// Kernel evaluation
//
// the static method k_function is for doing single kernel evaluation
// the constructor of Kernel prepares to calculate the l*l kernel matrix
// the member function get_Q is for getting one column from the Q Matrix
//
class QMatrix {
public:
    virtual Qfloat *get_Q(int column, int len) const = 0;
    virtual double *get_QD() const = 0;
    virtual void swap_index(int i, int j) const = 0;
    virtual ~QMatrix() {}
};

class Kernel: public QMatrix {
public:
#ifdef _DENSE_REP
    Kernel(int l, PREFIX(node) * x, const svm_parameter& param, BlasFunctions
*blas_functions);
#else
    Kernel(int l, PREFIX(node) * const * x, const svm_parameter& param, BlasFunctions
*blas_functions);
#endif
    virtual ~Kernel();

    static double k_function(const PREFIX(node) *x, const PREFIX(node) *y,
```

```
            const svm_parameter& param, BlasFunctions *blas_functions);
    virtual Qfloat *get_Q(int column, int len) const = 0;
    virtual double *get_QD() const = 0;
    virtual void swap_index(int i, int j) const // no so const...
    {
        swap(x[i],x[j]);
        if(x_square) swap(x_square[i],x_square[j]);
    }
protected:

    double (Kernel::*kernel_function)(int i, int j) const;

private:
#ifdef _DENSE_REP
    PREFIX(node) *x;
#else
    const PREFIX(node) **x;
#endif
    double *x_square;
    // scipy blas pointer
    BlasFunctions *m_blas;

    // svm_parameter
    const int kernel_type;
    const int degree;
    const double gamma;
    const double coef0;

    static double dot(const PREFIX(node) *px, const PREFIX(node) *py, BlasFunctions
*blas_functions);
#ifdef _DENSE_REP
    static double dot(const PREFIX(node) &px, const PREFIX(node) &py, BlasFunctions
*blas_functions);
#endif

    double kernel_linear(int i, int j) const
    {
        return dot(x[i],x[j],m_blas);
    }
    double kernel_poly(int i, int j) const
    {
        return powi(gamma*dot(x[i],x[j],m_blas)+coef0,degree);
    }
    double kernel_rbf(int i, int j) const
```

```
  {
    return exp(-gamma*(x_square[i]+x_square[j]-2*dot(x[i],x[j],m_blas)));
  }
  double kernel_sigmoid(int i, int j) const
  {
    return tanh(gamma*dot(x[i],x[j],m_blas)+coef0);
  }
  double kernel_precomputed(int i, int j) const
  {
#ifdef _DENSE_REP
    return (x+i)->values[x[j].ind];
#else
    return x[i][(int)(x[j][0].value)].value;
#endif
  }
};

#ifdef _DENSE_REP
Kernel::Kernel(int l, PREFIX(node) * x_, const svm_parameter& param, BlasFunctions
*blas_functions)
#else
Kernel::Kernel(int l, PREFIX(node) * const * x_, const svm_parameter& param, BlasFunctions
*blas_functions)
#endif
:kernel_type(param.kernel_type), degree(param.degree),
 gamma(param.gamma), coef0(param.coef0)
{
  m_blas = blas_functions;
  switch(kernel_type)
  {
    case LINEAR:
      kernel_function = &Kernel::kernel_linear;
      break;
    case POLY:
      kernel_function = &Kernel::kernel_poly;
      break;
    case RBF:
      kernel_function = &Kernel::kernel_rbf;
      break;
    case SIGMOID:
      kernel_function = &Kernel::kernel_sigmoid;
      break;
    case PRECOMPUTED:
      kernel_function = &Kernel::kernel_precomputed;
```

```
        break;
    }

    clone(x,x_,l);

    if(kernel_type == RBF)
    {
        x_square = new double[l];
        for(int i=0;i<l;i++)
            x_square[i] = dot(x[i],x[i],blas_functions);
    }
    else
        x_square = 0;
}

Kernel::~Kernel()
{
    delete[] x;
    delete[] x_square;
}

#ifdef _DENSE_REP
double Kernel::dot(const PREFIX(node) *px, const PREFIX(node) *py, BlasFunctions
*blas_functions)
{
    double sum = 0;

    int dim = min(px->dim, py->dim);
    sum = blas_functions->dot(dim, px->values, 1, py->values, 1);
    return sum;
}

double Kernel::dot(const PREFIX(node) &px, const PREFIX(node) &py, BlasFunctions
*blas_functions)
{
    double sum = 0;

    int dim = min(px.dim, py.dim);
    sum = blas_functions->dot(dim, px.values, 1, py.values, 1);
    return sum;
}
#else
double Kernel::dot(const PREFIX(node) *px, const PREFIX(node) *py, BlasFunctions
*blas_functions)
```

```
{
   double sum = 0;
   while(px->index != -1 && py->index != -1)
   {
      if(px->index == py->index)
      {
         sum += px->value * py->value;
         ++px;
         ++py;
      }
      else
      {
         if(px->index > py->index)
            ++py;
         else
            ++px;
      }
   }
   return sum;
}
#endif

double Kernel::k_function(const PREFIX(node) *x, const PREFIX(node) *y,
             const svm_parameter& param, BlasFunctions *blas_functions)
{
   switch(param.kernel_type)
   {
      case LINEAR:
         return dot(x,y,blas_functions);
      case POLY:
         return powi(param.gamma*dot(x,y,blas_functions)+param.coef0,param.degree);
      case RBF:
      {
         double sum = 0;
#ifdef _DENSE_REP
         int dim = min(x->dim, y->dim), i;
         double* m_array = (double*)malloc(sizeof(double)*dim);
         for (i = 0; i < dim; i++)
         {
            m_array[i] = x->values[i] - y->values[i];
         }
         sum = blas_functions->dot(dim, m_array, 1, m_array, 1);
         free(m_array);
         for (; i < x->dim; i++)
```

```
            sum += x->values[i] * x->values[i];
        for (; i < y->dim; i++)
            sum += y->values[i] * y->values[i];
#else
        while(x->index != -1 && y->index !=-1)
        {
            if(x->index == y->index)
            {
                double d = x->value - y->value;
                sum += d*d;
                ++x;
                ++y;
            }
            else
            {
                if(x->index > y->index)
                {
                    sum += y->value * y->value;
                    ++y;
                }
                else
                {
                    sum += x->value * x->value;
                    ++x;
                }
            }
        }

        while(x->index != -1)
        {
            sum += x->value * x->value;
            ++x;
        }

        while(y->index != -1)
        {
            sum += y->value * y->value;
            ++y;
        }
#endif
        return exp(-param.gamma*sum);
    }
    case SIGMOID:
        return tanh(param.gamma*dot(x,y,blas_functions)+param.coef0);
```

```
      case PRECOMPUTED:  //x: test (validation), y: SV
            {
#ifdef _DENSE_REP
        return x->values[y->ind];
#else
        return x[(int)(y->value)].value;
#endif
            }
      default:
        return 0;  // Unreachable
    }
}
// An SMO algorithm in Fan et al., JMLR 6(2005), p. 1889--1918
// Solves:
//
//  min 0.5(\alpha^T Q \alpha) + p^T \alpha
//
//     y^T \alpha = \delta
//     y_i = +1 or -1
//     0 <= alpha_i <= Cp for y_i = 1
//     0 <= alpha_i <= Cn for y_i = -1
//
// Given:
//
//  Q, p, y, Cp, Cn, and an initial feasible point \alpha
//  l is the size of vectors and matrices
//  eps is the stopping tolerance
//
// solution will be put in \alpha, objective value will be put in obj
//

class Solver {
public:
   Solver() {};
   virtual ~Solver() {};

   struct SolutionInfo {
      double obj;
      double rho;
            double *upper_bound;
      double r;   // for Solver_NU
            bool solve_timed_out;
   };
```

```
   void Solve(int l, const QMatrix& Q, const double *p_, const schar *y_,
        double *alpha_, const double *C_, double eps,
        SolutionInfo* si, int shrinking, int max_iter);
protected:
   int active_size;
   schar *y;
   double *G;     // gradient of objective function
   enum { LOWER_BOUND, UPPER_BOUND, FREE };
   char *alpha_status; // LOWER_BOUND, UPPER_BOUND, FREE
   double *alpha;
   const QMatrix *Q;
   const double *QD;
   double eps;
   double Cp,Cn;
      double *C;
   double *p;
   int *active_set;
   double *G_bar;     // gradient, if we treat free variables as 0
   int l;
   bool unshrink;  // XXX

   double get_C(int i)
   {
      return C[i];
   }
   void update_alpha_status(int i)
   {
      if(alpha[i] >= get_C(i))
         alpha_status[i] = UPPER_BOUND;
      else if(alpha[i] <= 0)
         alpha_status[i] = LOWER_BOUND;
      else alpha_status[i] = FREE;
   }
   bool is_upper_bound(int i) { return alpha_status[i] == UPPER_BOUND; }
   bool is_lower_bound(int i) { return alpha_status[i] == LOWER_BOUND; }
   bool is_free(int i) { return alpha_status[i] == FREE; }
   void swap_index(int i, int j);
   void reconstruct_gradient();
   virtual int select_working_set(int &i, int &j);
   virtual double calculate_rho();
   virtual void do_shrinking();
private:
   bool be_shrunk(int i, double Gmax1, double Gmax2);
};
```

```
void Solver::swap_index(int i, int j)
{
   Q->swap_index(i,j);
   swap(y[i],y[j]);
   swap(G[i],G[j]);
   swap(alpha_status[i],alpha_status[j]);
   swap(alpha[i],alpha[j]);
   swap(p[i],p[j]);
   swap(active_set[i],active_set[j]);
   swap(G_bar[i],G_bar[j]);
      swap(C[i], C[j]);
}

void Solver::reconstruct_gradient()
{
   // reconstruct inactive elements of G from G_bar and free variables

   if(active_size == l) return;

   int i,j;
   int nr_free = 0;

   for(j=active_size;j<l;j++)
      G[j] = G_bar[j] + p[j];

   for(j=0;j<active_size;j++)
      if(is_free(j))
         nr_free++;

   if(2*nr_free < active_size)
      info("\nWarning: using -h 0 may be faster\n");

   if (nr_free*l > 2*active_size*(l-active_size))
   {
      for(i=active_size;i<l;i++)
      {
         const Qfloat *Q_i = Q->get_Q(i,active_size);
         for(j=0;j<active_size;j++)
            if(is_free(j))
               G[i] += alpha[j] * Q_i[j];
      }
   }
   else
```

```
	{
		for(i=0;i<active_size;i++)
			if(is_free(i))
			{
				const Qfloat *Q_i = Q->get_Q(i,l);
				double alpha_i = alpha[i];
				for(j=active_size;j<l;j++)
					G[j] += alpha_i * Q_i[j];
			}
	}
}

void Solver::Solve(int l, const QMatrix& Q, const double *p_, const schar *y_,
		double *alpha_, const double *C_, double eps,
		SolutionInfo* si, int shrinking, int max_iter)
{
	this->l = l;
	this->Q = &Q;
	QD=Q.get_QD();
	clone(p, p_,l);
	clone(y, y_,l);
	clone(alpha,alpha_,l);
		clone(C, C_, l);
	this->eps = eps;
	unshrink = false;
		si->solve_timed_out = false;

	// initialize alpha_status
	{
		alpha_status = new char[l];
		for(int i=0;i<l;i++)
			update_alpha_status(i);
	}

	// initialize active set (for shrinking)
	{
		active_set = new int[l];
		for(int i=0;i<l;i++)
			active_set[i] = i;
		active_size = l;
	}

	// initialize gradient
	{
```

```
    G = new double[l];
    G_bar = new double[l];
    int i;
    for(i=0;i<l;i++)
    {
        G[i] = p[i];
        G_bar[i] = 0;
    }
    for(i=0;i<l;i++)
        if(!is_lower_bound(i))
        {
            const Qfloat *Q_i = Q.get_Q(i,l);
            double alpha_i = alpha[i];
            int j;
            for(j=0;j<l;j++)
                G[j] += alpha_i*Q_i[j];
            if(is_upper_bound(i))
                for(j=0;j<l;j++)
                    G_bar[j] += get_C(i) * Q_i[j];
        }
}

// optimization step

int iter = 0;
int counter = min(l,1000)+1;

while(1)
{
            // set max_iter to -1 to disable the mechanism
            if ((max_iter != -1) && (iter >= max_iter)) {
                info("WARN: libsvm Solver reached max_iter");
                si->solve_timed_out = true;
                break;
            }

    // show progress and do shrinking

    if(--counter == 0)
    {
        counter = min(l,1000);
        if(shrinking) do_shrinking();
        info(".");
    }
```

```
int i,j;
if(select_working_set(i,j)!=0)
{
   // reconstruct the whole gradient
   reconstruct_gradient();
   // reset active set size and check
   active_size = l;
   info("*");
   if(select_working_set(i,j)!=0)
      break;
   else
      counter = 1;    // do shrinking next iteration
}

++iter;

// update alpha[i] and alpha[j], handle bounds carefully

const Qfloat *Q_i = Q.get_Q(i,active_size);
const Qfloat *Q_j = Q.get_Q(j,active_size);

double C_i = get_C(i);
double C_j = get_C(j);

double old_alpha_i = alpha[i];
double old_alpha_j = alpha[j];

if(y[i]!=y[j])
{
   double quad_coef = QD[i]+QD[j]+2*Q_i[j];
   if (quad_coef <= 0)
      quad_coef = TAU;
   double delta = (-G[i]-G[j])/quad_coef;
   double diff = alpha[i] - alpha[j];
   alpha[i] += delta;
   alpha[j] += delta;

   if(diff > 0)
   {
      if(alpha[j] < 0)
      {
         alpha[j] = 0;
         alpha[i] = diff;
```

```
            }
        }
        else
        {
            if(alpha[i] < 0)
            {
                alpha[i] = 0;
                alpha[j] = -diff;
            }
        }
        if(diff > C_i - C_j)
        {
            if(alpha[i] > C_i)
            {
                alpha[i] = C_i;
                alpha[j] = C_i - diff;
            }
        }
        else
        {
            if(alpha[j] > C_j)
            {
                alpha[j] = C_j;
                alpha[i] = C_j + diff;
            }
        }
    }
    else
    {
        double quad_coef = QD[i]+QD[j]-2*Q_i[j];
        if (quad_coef <= 0)
            quad_coef = TAU;
        double delta = (G[i]-G[j])/quad_coef;
        double sum = alpha[i] + alpha[j];
        alpha[i] -= delta;
        alpha[j] += delta;

        if(sum > C_i)
        {
            if(alpha[i] > C_i)
            {
                alpha[i] = C_i;
                alpha[j] = sum - C_i;
            }
```

```
    }
    else
    {
        if(alpha[j] < 0)
        {
            alpha[j] = 0;
            alpha[i] = sum;
        }
    }
    if(sum > C_j)
    {
        if(alpha[j] > C_j)
        {
            alpha[j] = C_j;
            alpha[i] = sum - C_j;
        }
    }
    else
    {
        if(alpha[i] < 0)
        {
            alpha[i] = 0;
            alpha[j] = sum;
        }
    }
}

// update G

double delta_alpha_i = alpha[i] - old_alpha_i;
double delta_alpha_j = alpha[j] - old_alpha_j;

for(int k=0;k<active_size;k++)
{
    G[k] += Q_i[k]*delta_alpha_i + Q_j[k]*delta_alpha_j;
}

// update alpha_status and G_bar

{
    bool ui = is_upper_bound(i);
    bool uj = is_upper_bound(j);
    update_alpha_status(i);
    update_alpha_status(j);
```

```
        int k;
        if(ui != is_upper_bound(i))
        {
           Q_i = Q.get_Q(i,l);
           if(ui)
              for(k=0;k<l;k++)
                 G_bar[k] -= C_i * Q_i[k];
           else
              for(k=0;k<l;k++)
                 G_bar[k] += C_i * Q_i[k];
        }

        if(uj != is_upper_bound(j))
        {
           Q_j = Q.get_Q(j,l);
           if(uj)
              for(k=0;k<l;k++)
                 G_bar[k] -= C_j * Q_j[k];
           else
              for(k=0;k<l;k++)
                 G_bar[k] += C_j * Q_j[k];
        }
     }
  }

  // calculate rho

  si->rho = calculate_rho();

  // calculate objective value
  {
     double v = 0;
     int i;
     for(i=0;i<l;i++)
        v += alpha[i] * (G[i] + p[i]);

     si->obj = v/2;
  }

  // put back the solution
  {
     for(int i=0;i<l;i++)
        alpha_[active_set[i]] = alpha[i];
  }
```

```
  // juggle everything back
  /*{
    for(int i=0;i<l;i++)
      while(active_set[i] != i)
        swap_index(i,active_set[i]);
        // or Q.swap_index(i,active_set[i]);
  }*/

  for(int i=0;i<l;i++)
    si->upper_bound[i] = C[i];

  info("\noptimization finished, #iter = %d\n",iter);

  delete[] p;
  delete[] y;
  delete[] alpha;
  delete[] alpha_status;
  delete[] active_set;
  delete[] G;
  delete[] G_bar;
  delete[] C;
}

// return 1 if already optimal, return 0 otherwise
int Solver::select_working_set(int &out_i, int &out_j)
{
  // return i,j such that
  // i: maximizes -y_i * grad(f)_i, i in I_up(\alpha)
  // j: minimizes the decrease of obj value
  //    (if quadratic coefficient <= 0, replace it with tau)
  //    -y_j*grad(f)_j < -y_i*grad(f)_i, j in I_low(\alpha)

  double Gmax = -INF;
  double Gmax2 = -INF;
  int Gmax_idx = -1;
  int Gmin_idx = -1;
  double obj_diff_min = INF;

  for(int t=0;t<active_size;t++)
    if(y[t]==+1)
    {
      if(!is_upper_bound(t))
        if(-G[t] >= Gmax)
```

```
            {
               Gmax = -G[t];
               Gmax_idx = t;
            }
      }
      else
      {
         if(!is_lower_bound(t))
            if(G[t] >= Gmax)
            {
               Gmax = G[t];
               Gmax_idx = t;
            }
      }

   int i = Gmax_idx;
   const Qfloat *Q_i = NULL;
   if(i != -1) // NULL Q_i not accessed: Gmax=-INF if i=-1
      Q_i = Q->get_Q(i,active_size);

   for(int j=0;j<active_size;j++)
   {
      if(y[j]==+1)
      {
         if (!is_lower_bound(j))
         {
            double grad_diff=Gmax+G[j];
            if (G[j] >= Gmax2)
               Gmax2 = G[j];
            if (grad_diff > 0)
            {
               double obj_diff;
               double quad_coef = QD[i]+QD[j]-2.0*y[i]*Q_i[j];
               if (quad_coef > 0)
                  obj_diff = -(grad_diff*grad_diff)/quad_coef;
               else
                  obj_diff = -(grad_diff*grad_diff)/TAU;

               if (obj_diff <= obj_diff_min)
               {
                  Gmin_idx=j;
                  obj_diff_min = obj_diff;
               }
            }
```

```
          }
       }
       else
       {
          if (!is_upper_bound(j))
          {
             double grad_diff= Gmax-G[j];
             if (-G[j] >= Gmax2)
                Gmax2 = -G[j];
             if (grad_diff > 0)
             {
                double obj_diff;
                double quad_coef = QD[i]+QD[j]+2.0*y[i]*Q_i[j];
                if (quad_coef > 0)
                   obj_diff = -(grad_diff*grad_diff)/quad_coef;
                else
                   obj_diff = -(grad_diff*grad_diff)/TAU;

                if (obj_diff <= obj_diff_min)
                {
                   Gmin_idx=j;
                   obj_diff_min = obj_diff;
                }
             }
          }
       }
    }

    if(Gmax+Gmax2 < eps || Gmin_idx == -1)
       return 1;

    out_i = Gmax_idx;
    out_j = Gmin_idx;
    return 0;
 }

 bool Solver::be_shrunk(int i, double Gmax1, double Gmax2)
 {
    if(is_upper_bound(i))
    {
       if(y[i]==+1)
          return(-G[i] > Gmax1);
       else
          return(-G[i] > Gmax2);
```

```
   }
   else if(is_lower_bound(i))
   {
      if(y[i]==+1)
         return(G[i] > Gmax2);
      else
         return(G[i] > Gmax1);
   }
   else
      return(false);
}

void Solver::do_shrinking()
{
   int i;
   double Gmax1 = -INF;      // max { -y_i * grad(f)_i | i in I_up(\alpha) }
   double Gmax2 = -INF;      // max { y_i * grad(f)_i | i in I_low(\alpha) }

   // find maximal violating pair first
   for(i=0;i<active_size;i++)
   {
      if(y[i]==+1)
      {
         if(!is_upper_bound(i))
         {
            if(-G[i] >= Gmax1)
               Gmax1 = -G[i];
         }
         if(!is_lower_bound(i))
         {
            if(G[i] >= Gmax2)
               Gmax2 = G[i];
         }
      }
      else
      {
         if(!is_upper_bound(i))
         {
            if(-G[i] >= Gmax2)
               Gmax2 = -G[i];
         }
         if(!is_lower_bound(i))
         {
            if(G[i] >= Gmax1)
```

```
                Gmax1 = G[i];
            }
        }
    }

    if(unshrink == false && Gmax1 + Gmax2 <= eps*10)
    {
        unshrink = true;
        reconstruct_gradient();
        active_size = l;
        info("*");
    }

    for(i=0;i<active_size;i++)
        if (be_shrunk(i, Gmax1, Gmax2))
        {
            active_size--;
            while (active_size > i)
            {
                if (!be_shrunk(active_size, Gmax1, Gmax2))
                {
                    swap_index(i,active_size);
                    break;
                }
                active_size--;
            }
        }
}

double Solver::calculate_rho()
{
    double r;
    int nr_free = 0;
    double ub = INF, lb = -INF, sum_free = 0;
    for(int i=0;i<active_size;i++)
    {
        double yG = y[i]*G[i];

        if(is_upper_bound(i))
        {
            if(y[i]==-1)
                ub = min(ub,yG);
            else
                lb = max(lb,yG);
```

```
        }
        else if(is_lower_bound(i))
        {
            if(y[i]==+1)
                ub = min(ub,yG);
            else
                lb = max(lb,yG);
        }
        else
        {
            ++nr_free;
            sum_free += yG;
        }
    }

    if(nr_free>0)
        r = sum_free/nr_free;
    else
        r = (ub+lb)/2;

    return r;
}

//
// Solver for nu-svm classification and regression
//
// additional constraint: e^T \alpha = constant
//
class Solver_NU : public Solver
{
public:
    Solver_NU() {}
    void Solve(int l, const QMatrix& Q, const double *p, const schar *y,
           double *alpha, const double *C_, double eps,
           SolutionInfo* si, int shrinking, int max_iter)
    {
        this->si = si;
        Solver::Solve(l,Q,p,y,alpha,C_,eps,si,shrinking,max_iter);
    }
private:
    SolutionInfo *si;
    int select_working_set(int &i, int &j);
    double calculate_rho();
    bool be_shrunk(int i, double Gmax1, double Gmax2, double Gmax3, double Gmax4);
```

```
    void do_shrinking();
};

// return 1 if already optimal, return 0 otherwise
int Solver_NU::select_working_set(int &out_i, int &out_j)
{
    // return i,j such that y_i = y_j and
    // i: maximizes -y_i * grad(f)_i, i in I_up(\alpha)
    // j: minimizes the decrease of obj value
    //    (if quadratic coefficient <= 0, replace it with tau)
    //    -y_j*grad(f)_j < -y_i*grad(f)_i, j in I_low(\alpha)

    double Gmaxp = -INF;
    double Gmaxp2 = -INF;
    int Gmaxp_idx = -1;

    double Gmaxn = -INF;
    double Gmaxn2 = -INF;
    int Gmaxn_idx = -1;

    int Gmin_idx = -1;
    double obj_diff_min = INF;

    for(int t=0;t<active_size;t++)
        if(y[t]==+1)
        {
            if(!is_upper_bound(t))
                if(-G[t] >= Gmaxp)
                {
                    Gmaxp = -G[t];
                    Gmaxp_idx = t;
                }
        }
        else
        {
            if(!is_lower_bound(t))
                if(G[t] >= Gmaxn)
                {
                    Gmaxn = G[t];
                    Gmaxn_idx = t;
                }
        }

    int ip = Gmaxp_idx;
```

```
int in = Gmaxn_idx;
const Qfloat *Q_ip = NULL;
const Qfloat *Q_in = NULL;
if(ip != -1) // NULL Q_ip not accessed: Gmaxp=-INF if ip=-1
   Q_ip = Q->get_Q(ip,active_size);
if(in != -1)
   Q_in = Q->get_Q(in,active_size);

for(int j=0;j<active_size;j++)
{
   if(y[j]==+1)
   {
      if (!is_lower_bound(j))
      {
         double grad_diff=Gmaxp+G[j];
         if (G[j] >= Gmaxp2)
            Gmaxp2 = G[j];
         if (grad_diff > 0)
         {
            double obj_diff;
            double quad_coef = QD[ip]+QD[j]-2*Q_ip[j];
            if (quad_coef > 0)
               obj_diff = -(grad_diff*grad_diff)/quad_coef;
            else
               obj_diff = -(grad_diff*grad_diff)/TAU;

            if (obj_diff <= obj_diff_min)
            {
               Gmin_idx=j;
               obj_diff_min = obj_diff;
            }
         }
      }
   }
   else
   {
      if (!is_upper_bound(j))
      {
         double grad_diff=Gmaxn-G[j];
         if (-G[j] >= Gmaxn2)
            Gmaxn2 = -G[j];
         if (grad_diff > 0)
         {
            double obj_diff;
```

```
                double quad_coef = QD[in]+QD[j]-2*Q_in[j];
                if (quad_coef > 0)
                    obj_diff = -(grad_diff*grad_diff)/quad_coef;
                else
                    obj_diff = -(grad_diff*grad_diff)/TAU;

                if (obj_diff <= obj_diff_min)
                {
                    Gmin_idx=j;
                    obj_diff_min = obj_diff;
                }
            }
        }
    }
}

if(max(Gmaxp+Gmaxp2,Gmaxn+Gmaxn2) < eps || Gmin_idx == -1)
    return 1;

if (y[Gmin_idx] == +1)
    out_i = Gmaxp_idx;
else
    out_i = Gmaxn_idx;
out_j = Gmin_idx;

return 0;
}

bool Solver_NU::be_shrunk(int i, double Gmax1, double Gmax2, double Gmax3, double
Gmax4)
{
    if(is_upper_bound(i))
    {
        if(y[i]==+1)
            return(-G[i] > Gmax1);
        else
            return(-G[i] > Gmax4);
    }
    else if(is_lower_bound(i))
    {
        if(y[i]==+1)
            return(G[i] > Gmax2);
        else
            return(G[i] > Gmax3);
```

```
	}
	else
		return(false);
}

void Solver_NU::do_shrinking()
{
	double Gmax1 = -INF;	// max { -y_i * grad(f)_i | y_i = +1, i in I_up(\alpha) }
	double Gmax2 = -INF;	// max { y_i * grad(f)_i | y_i = +1, i in I_low(\alpha) }
	double Gmax3 = -INF;	// max { -y_i * grad(f)_i | y_i = -1, i in I_up(\alpha) }
	double Gmax4 = -INF;	// max { y_i * grad(f)_i | y_i = -1, i in I_low(\alpha) }

	// find maximal violating pair first
	int i;
	for(i=0;i<active_size;i++)
	{
		if(!is_upper_bound(i))
		{
			if(y[i]==+1)
			{
				if(-G[i] > Gmax1) Gmax1 = -G[i];
			}
			else	if(-G[i] > Gmax4) Gmax4 = -G[i];
		}
		if(!is_lower_bound(i))
		{
			if(y[i]==+1)
			{
				if(G[i] > Gmax2) Gmax2 = G[i];
			}
			else	if(G[i] > Gmax3) Gmax3 = G[i];
		}
	}

	if(unshrink == false && max(Gmax1+Gmax2,Gmax3+Gmax4) <= eps*10)
	{
		unshrink = true;
		reconstruct_gradient();
		active_size = l;
	}

	for(i=0;i<active_size;i++)
		if (be_shrunk(i, Gmax1, Gmax2, Gmax3, Gmax4))
		{
```

```
        active_size--;
        while (active_size > i)
        {
          if (!be_shrunk(active_size, Gmax1, Gmax2, Gmax3, Gmax4))
          {
            swap_index(i,active_size);
            break;
          }
          active_size--;
        }
      }
}

double Solver_NU::calculate_rho()
{
  int nr_free1 = 0,nr_free2 = 0;
  double ub1 = INF, ub2 = INF;
  double lb1 = -INF, lb2 = -INF;
  double sum_free1 = 0, sum_free2 = 0;

  for(int i=0;i<active_size;i++)
  {
    if(y[i]==+1)
    {
      if(is_upper_bound(i))
        lb1 = max(lb1,G[i]);
      else if(is_lower_bound(i))
        ub1 = min(ub1,G[i]);
      else
      {
        ++nr_free1;
        sum_free1 += G[i];
      }
    }
    else
    {
      if(is_upper_bound(i))
        lb2 = max(lb2,G[i]);
      else if(is_lower_bound(i))
        ub2 = min(ub2,G[i]);
      else
      {
        ++nr_free2;
        sum_free2 += G[i];
```

```
        }
      }
    }

    double r1,r2;
    if(nr_free1 > 0)
      r1 = sum_free1/nr_free1;
    else
      r1 = (ub1+lb1)/2;

    if(nr_free2 > 0)
      r2 = sum_free2/nr_free2;
    else
      r2 = (ub2+lb2)/2;

    si->r = (r1+r2)/2;
    return (r1-r2)/2;
}

//
// Q matrices for various formulations
//
class SVC_Q: public Kernel
{
public:
    SVC_Q(const PREFIX(problem)& prob, const svm_parameter& param, const schar *y_,
BlasFunctions *blas_functions)
    :Kernel(prob.l, prob.x, param, blas_functions)
    {
      clone(y,y_,prob.l);
      cache = new Cache(prob.l,(long int)(param.cache_size*(1<<20)));
      QD = new double[prob.l];
      for(int i=0;i<prob.l;i++)
        QD[i] = (this->*kernel_function)(i,i);
    }

    Qfloat *get_Q(int i, int len) const
    {
      Qfloat *data;
      int start, j;
      if((start = cache->get_data(i,&data,len)) < len)
      {
        for(j=start;j<len;j++)
          data[j] = (Qfloat)(y[i]*y[j]*(this->*kernel_function)(i,j));
```

```
    }
    return data;
  }

  double *get_QD() const
  {
    return QD;
  }

  void swap_index(int i, int j) const
  {
    cache->swap_index(i,j);
    Kernel::swap_index(i,j);
    swap(y[i],y[j]);
    swap(QD[i],QD[j]);
  }

  ~SVC_Q()
  {
    delete[] y;
    delete cache;
    delete[] QD;
  }
private:
  schar *y;
  Cache *cache;
  double *QD;
};

class ONE_CLASS_Q: public Kernel
{
public:
  ONE_CLASS_Q(const PREFIX(problem)& prob, const svm_parameter& param,
BlasFunctions *blas_functions)
  :Kernel(prob.l, prob.x, param, blas_functions)
  {
    cache = new Cache(prob.l,(long int)(param.cache_size*(1<<20)));
    QD = new double[prob.l];
    for(int i=0;i<prob.l;i++)
      QD[i] = (this->*kernel_function)(i,i);
  }

  Qfloat *get_Q(int i, int len) const
  {
```

```
      Qfloat *data;
      int start, j;
      if((start = cache->get_data(i,&data,len)) < len)
      {
         for(j=start;j<len;j++)
            data[j] = (Qfloat)(this->*kernel_function)(i,j);
      }
      return data;
   }

   double *get_QD() const
   {
      return QD;
   }

   void swap_index(int i, int j) const
   {
      cache->swap_index(i,j);
      Kernel::swap_index(i,j);
      swap(QD[i],QD[j]);
   }

   ~ONE_CLASS_Q()
   {
      delete cache;
      delete[] QD;
   }
private:
   Cache *cache;
   double *QD;
};

class SVR_Q: public Kernel
{
public:
   SVR_Q(const PREFIX(problem)& prob, const svm_parameter& param, BlasFunctions
*blas_functions)
   :Kernel(prob.l, prob.x, param, blas_functions)
   {
      l = prob.l;
      cache = new Cache(l,(long int)(param.cache_size*(1<<20)));
      QD = new double[2*l];
      sign = new schar[2*l];
      index = new int[2*l];
```

```
    for(int k=0;k<l;k++)
    {
        sign[k] = 1;
        sign[k+l] = -1;
        index[k] = k;
        index[k+l] = k;
        QD[k] = (this->*kernel_function)(k,k);
        QD[k+l] = QD[k];
    }
    buffer[0] = new Qfloat[2*l];
    buffer[1] = new Qfloat[2*l];
    next_buffer = 0;
}

void swap_index(int i, int j) const
{
    swap(sign[i],sign[j]);
    swap(index[i],index[j]);
    swap(QD[i],QD[j]);
}

Qfloat *get_Q(int i, int len) const
{
    Qfloat *data;
    int j, real_i = index[i];
    if(cache->get_data(real_i,&data,l) < l)
    {
        for(j=0;j<l;j++)
            data[j] = (Qfloat)(this->*kernel_function)(real_i,j);
    }

    // reorder and copy
    Qfloat *buf = buffer[next_buffer];
    next_buffer = 1 - next_buffer;
    schar si = sign[i];
    for(j=0;j<len;j++)
        buf[j] = (Qfloat) si * (Qfloat) sign[j] * data[index[j]];
    return buf;
}

double *get_QD() const
{
    return QD;
}
```

```
  ~SVR_Q()
  {
    delete cache;
    delete[] sign;
    delete[] index;
    delete[] buffer[0];
    delete[] buffer[1];
    delete[] QD;
  }
private:
  int l;
  Cache *cache;
  schar *sign;
  int *index;
  mutable int next_buffer;
  Qfloat *buffer[2];
  double *QD;
};

//
// construct and solve various formulations
//
static void solve_c_svc(
  const PREFIX(problem) *prob, const svm_parameter* param,
  double *alpha, Solver::SolutionInfo* si, double Cp, double Cn, BlasFunctions
*blas_functions)
{
  int l = prob->l;
  double *minus_ones = new double[l];
  schar *y = new schar[l];
    double *C = new double[l];

  int i;

  for(i=0;i<l;i++)
  {
    alpha[i] = 0;
    minus_ones[i] = -1;
    if(prob->y[i] > 0)
    {
      y[i] = +1;
      C[i] = prob->W[i]*Cp;
    }
```

```
        else
        {
          y[i] = -1;
          C[i] = prob->W[i]*Cn;
        }
    }

    Solver s;
    s.Solve(l, SVC_Q(*prob,*param,y, blas_functions), minus_ones, y,
        alpha, C, param->eps, si, param->shrinking,
                param->max_iter);

        /*
    double sum_alpha=0;
    for(i=0;i<l;i++)
        sum_alpha += alpha[i];

    if (Cp==Cn)
        info("nu = %f\n", sum_alpha/(Cp*prob->l));
        */

    for(i=0;i<l;i++)
        alpha[i] *= y[i];

        delete[] C;
    delete[] minus_ones;
    delete[] y;
}

static void solve_nu_svc(
    const PREFIX(problem) *prob, const svm_parameter *param,
    double *alpha, Solver::SolutionInfo* si, BlasFunctions *blas_functions)
{
    int i;
    int l = prob->l;
    double nu = param->nu;

    schar *y = new schar[l];
        double *C = new double[l];

    for(i=0;i<l;i++)
        {
        if(prob->y[i]>0)
          y[i] = +1;
```

```
      else
         y[i] = -1;

      C[i] = prob->W[i];
   }

   double nu_l = 0;
   for(i=0;i<l;i++) nu_l += nu*C[i];
   double sum_pos = nu_l/2;
   double sum_neg = nu_l/2;

   for(i=0;i<l;i++)
      if(y[i] == +1)
      {
         alpha[i] = min(C[i],sum_pos);
         sum_pos -= alpha[i];
      }
      else
      {
         alpha[i] = min(C[i],sum_neg);
         sum_neg -= alpha[i];
      }

   double *zeros = new double[l];

   for(i=0;i<l;i++)
      zeros[i] = 0;

   Solver_NU s;
   s.Solve(l, SVC_Q(*prob,*param,y,blas_functions), zeros, y,
      alpha, C, param->eps, si,  param->shrinking, param->max_iter);
   double r = si->r;

   info("C = %f\n",1/r);

   for(i=0;i<l;i++)
      {
      alpha[i] *= y[i]/r;
      si->upper_bound[i] /= r;
      }

   si->rho /= r;
   si->obj /= (r*r);
```

```
      delete[] C;
   delete[] y;
   delete[] zeros;
}

static void solve_one_class(
   const PREFIX(problem) *prob, const svm_parameter *param,
   double *alpha, Solver::SolutionInfo* si, BlasFunctions *blas_functions)
{
   int l = prob->l;
   double *zeros = new double[l];
   schar *ones = new schar[l];
   double *C = new double[l];
   int i;

   double nu_l = 0;

   for(i=0;i<l;i++)
   {
      C[i] = prob->W[i];
      nu_l += C[i] * param->nu;
   }

   i = 0;
   while(nu_l > 0)
   {
      alpha[i] = min(C[i],nu_l);
      nu_l -= alpha[i];
      ++i;
   }
   for(;i<l;i++)
      alpha[i] = 0;

   for(i=0;i<l;i++)
   {
      zeros[i] = 0;
      ones[i] = 1;
   }

   Solver s;
   s.Solve(l, ONE_CLASS_Q(*prob,*param,blas_functions), zeros, ones,
      alpha, C, param->eps, si, param->shrinking, param->max_iter);

      delete[] C;
```

```
    delete[] zeros;
    delete[] ones;
}

static void solve_epsilon_svr(
    const PREFIX(problem) *prob, const svm_parameter *param,
    double *alpha, Solver::SolutionInfo* si, BlasFunctions *blas_functions)
{
    int l = prob->l;
    double *alpha2 = new double[2*l];
    double *linear_term = new double[2*l];
    schar *y = new schar[2*l];
        double *C = new double[2*l];
        int i;

    for(i=0;i<l;i++)
    {
        alpha2[i] = 0;
        linear_term[i] = param->p - prob->y[i];
        y[i] = 1;
                C[i] = prob->W[i]*param->C;

        alpha2[i+l] = 0;
        linear_term[i+l] = param->p + prob->y[i];
        y[i+l] = -1;
                C[i+l] = prob->W[i]*param->C;
    }

    Solver s;
    s.Solve(2*l, SVR_Q(*prob,*param,blas_functions), linear_term, y,
        alpha2, C, param->eps, si, param->shrinking, param->max_iter);

    double sum_alpha = 0;
    for(i=0;i<l;i++)
    {
        alpha[i] = alpha2[i] - alpha2[i+l];
        sum_alpha += fabs(alpha[i]);
    }


    delete[] alpha2;
    delete[] linear_term;
        delete[] C;
    delete[] y;
```

```
}

static void solve_nu_svr(
	const PREFIX(problem) *prob, const svm_parameter *param,
	double *alpha, Solver::SolutionInfo* si, BlasFunctions *blas_functions)
{
	int l = prob->l;
	double *C = new double[2*l];
	double *alpha2 = new double[2*l];
	double *linear_term = new double[2*l];
	schar *y = new schar[2*l];
	int i;

	double sum = 0;
	for(i=0;i<l;i++)
	{
		C[i] = C[i+l] = prob->W[i]*param->C;
		sum += C[i] * param->nu;
	}
	sum /= 2;

	for(i=0;i<l;i++)
	{
		alpha2[i] = alpha2[i+l] = min(sum,C[i]);
		sum -= alpha2[i];

		linear_term[i] = - prob->y[i];
		y[i] = 1;

		linear_term[i+l] = prob->y[i];
		y[i+l] = -1;
	}

	Solver_NU s;
	s.Solve(2*l, SVR_Q(*prob,*param,blas_functions), linear_term, y,
		alpha2, C, param->eps, si, param->shrinking, param->max_iter);

	info("epsilon = %f\n",-si->r);

	for(i=0;i<l;i++)
		alpha[i] = alpha2[i] - alpha2[i+l];

	delete[] alpha2;
	delete[] linear_term;
```

```
      delete[] C;
   delete[] y;
}

//
// decision_function
//
struct decision_function
{
   double *alpha;
   double rho;
};

static decision_function svm_train_one(
   const PREFIX(problem) *prob, const svm_parameter *param,
   double Cp, double Cn, int *status, BlasFunctions *blas_functions)
{
   double *alpha = Malloc(double,prob->l);
   Solver::SolutionInfo si;
   switch(param->svm_type)
   {
      case C_SVC:
         si.upper_bound = Malloc(double,prob->l);
         solve_c_svc(prob,param,alpha,&si,Cp,Cn,blas_functions);
         break;
      case NU_SVC:
         si.upper_bound = Malloc(double,prob->l);
         solve_nu_svc(prob,param,alpha,&si,blas_functions);
         break;
      case ONE_CLASS:
         si.upper_bound = Malloc(double,prob->l);
         solve_one_class(prob,param,alpha,&si,blas_functions);
         break;
      case EPSILON_SVR:
         si.upper_bound = Malloc(double,2*prob->l);
         solve_epsilon_svr(prob,param,alpha,&si,blas_functions);
         break;
      case NU_SVR:
         si.upper_bound = Malloc(double,2*prob->l);
         solve_nu_svr(prob,param,alpha,&si,blas_functions);
         break;
   }

      *status |= si.solve_timed_out;
```

```
	info("obj = %f, rho = %f\n",si.obj,si.rho);

	// output SVs

	int nSV = 0;
	int nBSV = 0;
	for(int i=0;i<prob->l;i++)
	{
		if(fabs(alpha[i]) > 0)
		{
			++nSV;
			if(prob->y[i] > 0)
			{
				if(fabs(alpha[i]) >= si.upper_bound[i])
					++nBSV;
			}
			else
			{
				if(fabs(alpha[i]) >= si.upper_bound[i])
					++nBSV;
			}
		}
	}

		free(si.upper_bound);

	info("nSV = %d, nBSV = %d\n",nSV,nBSV);

	decision_function f;
	f.alpha = alpha;
	f.rho = si.rho;
	return f;
}

// Platt's binary SVM Probabilistic Output: an improvement from Lin et al.
static void sigmoid_train(
	int l, const double *dec_values, const double *labels,
	double& A, double& B)
{
	double prior1=0, prior0 = 0;
	int i;

	for (i=0;i<l;i++)
```

```
    if (labels[i] > 0) prior1+=1;
    else prior0+=1;

  int max_iter=100;   // Maximal number of iterations
  double min_step=1e-10;  // Minimal step taken in line search
  double sigma=1e-12; // For numerically strict PD of Hessian
  double eps=1e-5;
  double hiTarget=(prior1+1.0)/(prior1+2.0);
  double loTarget=1/(prior0+2.0);
  double *t=Malloc(double,l);
  double fApB,p,q,h11,h22,h21,g1,g2,det,dA,dB,gd,stepsize;
  double newA,newB,newf,d1,d2;
  int iter;

  // Initial Point and Initial Fun Value
  A=0.0; B=log((prior0+1.0)/(prior1+1.0));
  double fval = 0.0;

  for (i=0;i<l;i++)
  {
    if (labels[i]>0) t[i]=hiTarget;
    else t[i]=loTarget;
    fApB = dec_values[i]*A+B;
    if (fApB>=0)
      fval += t[i]*fApB + log(1+exp(-fApB));
    else
      fval += (t[i] - 1)*fApB +log(1+exp(fApB));
  }
  for (iter=0;iter<max_iter;iter++)
  {
    // Update Gradient and Hessian (use H' = H + sigma I)
    h11=sigma; // numerically ensures strict PD
    h22=sigma;
    h21=0.0;g1=0.0;g2=0.0;
    for (i=0;i<l;i++)
    {
      fApB = dec_values[i]*A+B;
      if (fApB >= 0)
      {
        p=exp(-fApB)/(1.0+exp(-fApB));
        q=1.0/(1.0+exp(-fApB));
      }
      else
      {
```

```
      p=1.0/(1.0+exp(fApB));
      q=exp(fApB)/(1.0+exp(fApB));
    }
    d2=p*q;
    h11+=dec_values[i]*dec_values[i]*d2;
    h22+=d2;
    h21+=dec_values[i]*d2;
    d1=t[i]-p;
    g1+=dec_values[i]*d1;
    g2+=d1;
  }

  // Stopping Criteria
  if (fabs(g1)<eps && fabs(g2)<eps)
    break;

  // Finding Newton direction: -inv(H') * g
  det=h11*h22-h21*h21;
  dA=-(h22*g1 - h21 * g2) / det;
  dB=-(-h21*g1+ h11 * g2) / det;
  gd=g1*dA+g2*dB;


  stepsize = 1;      // Line Search
  while (stepsize >= min_step)
  {
    newA = A + stepsize * dA;
    newB = B + stepsize * dB;

    // New function value
    newf = 0.0;
    for (i=0;i<l;i++)
    {
      fApB = dec_values[i]*newA+newB;
      if (fApB >= 0)
        newf += t[i]*fApB + log(1+exp(-fApB));
      else
        newf += (t[i] - 1)*fApB +log(1+exp(fApB));
    }
    // Check sufficient decrease
    if (newf<fval+0.0001*stepsize*gd)
    {
      A=newA;B=newB;fval=newf;
      break;
```

```
      }
      else
        stepsize = stepsize / 2.0;
    }

    if (stepsize < min_step)
    {
      info("Line search fails in two-class probability estimates\n");
      break;
    }
  }

  if (iter>=max_iter)
    info("Reaching maximal iterations in two-class probability estimates\n");
  free(t);
}

static double sigmoid_predict(double decision_value, double A, double B)
{
  double fApB = decision_value*A+B;
  // 1-p used later; avoid catastrophic cancellation
  if (fApB >= 0)
    return exp(-fApB)/(1.0+exp(-fApB));
  else
    return 1.0/(1+exp(fApB)) ;
}

// Method 2 from the multiclass_prob paper by Wu, Lin, and Weng
static void multiclass_probability(int k, double **r, double *p)
{
  int t,j;
  int iter = 0, max_iter=max(100,k);
  double **Q=Malloc(double *,k);
  double *Qp=Malloc(double,k);
  double pQp, eps=0.005/k;

  for (t=0;t<k;t++)
  {
    p[t]=1.0/k;  // Valid if k = 1
    Q[t]=Malloc(double,k);
    Q[t][t]=0;
    for (j=0;j<t;j++)
    {
      Q[t][t]+=r[j][t]*r[j][t];
```

```
      Q[t][j]=Q[j][t];
    }
    for (j=t+1;j<k;j++)
    {
      Q[t][t]+=r[j][t]*r[j][t];
      Q[t][j]=-r[j][t]*r[t][j];
    }
  }
  for (iter=0;iter<max_iter;iter++)
  {
    // stopping condition, recalculate QP,pQP for numerical accuracy
    pQp=0;
    for (t=0;t<k;t++)
    {
      Qp[t]=0;
      for (j=0;j<k;j++)
        Qp[t]+=Q[t][j]*p[j];
      pQp+=p[t]*Qp[t];
    }
    double max_error=0;
    for (t=0;t<k;t++)
    {
      double error=fabs(Qp[t]-pQp);
      if (error>max_error)
        max_error=error;
    }
    if (max_error<eps) break;

    for (t=0;t<k;t++)
    {
      double diff=(-Qp[t]+pQp)/Q[t][t];
      p[t]+=diff;
      pQp=(pQp+diff*(diff*Q[t][t]+2*Qp[t]))/(1+diff)/(1+diff);
      for (j=0;j<k;j++)
      {
        Qp[j]=(Qp[j]+diff*Q[t][j])/(1+diff);
        p[j]/=(1+diff);
      }
    }
  }
  if (iter>=max_iter)
    info("Exceeds max_iter in multiclass_prob\n");
  for(t=0;t<k;t++) free(Q[t]);
  free(Q);
```

```
  free(Qp);
}

// Cross-validation decision values for probability estimates
static void svm_binary_svc_probability(
  const PREFIX(problem) *prob, const svm_parameter *param,
  double Cp, double Cn, double& probA, double& probB, int * status, BlasFunctions
*blas_functions)
{
  int i;
  int nr_fold = 5;
  int *perm = Malloc(int,prob->l);
  double *dec_values = Malloc(double,prob->l);

  // random shuffle
  for(i=0;i<prob->l;i++) perm[i]=i;
  for(i=0;i<prob->l;i++)
  {
    int j = i+bounded_rand_int(prob->l-i);
    swap(perm[i],perm[j]);
  }
  for(i=0;i<nr_fold;i++)
  {
    int begin = i*prob->l/nr_fold;
    int end = (i+1)*prob->l/nr_fold;
    int j,k;
    struct PREFIX(problem) subprob;

    subprob.l = prob->l-(end-begin);
#ifdef _DENSE_REP
    subprob.x = Malloc(struct PREFIX(node),subprob.l);
#else
    subprob.x = Malloc(struct PREFIX(node)*,subprob.l);
#endif
    subprob.y = Malloc(double,subprob.l);
        subprob.W = Malloc(double,subprob.l);

    k=0;
    for(j=0;j<begin;j++)
    {
      subprob.x[k] = prob->x[perm[j]];
      subprob.y[k] = prob->y[perm[j]];
      subprob.W[k] = prob->W[perm[j]];
      ++k;
```

```
    }
    for(j=end;j<prob->l;j++)
    {
        subprob.x[k] = prob->x[perm[j]];
        subprob.y[k] = prob->y[perm[j]];
        subprob.W[k] = prob->W[perm[j]];
        ++k;
    }
    int p_count=0,n_count=0;
    for(j=0;j<k;j++)
        if(subprob.y[j]>0)
            p_count++;
        else
            n_count++;

    if(p_count==0 && n_count==0)
        for(j=begin;j<end;j++)
            dec_values[perm[j]] = 0;
    else if(p_count > 0 && n_count == 0)
        for(j=begin;j<end;j++)
            dec_values[perm[j]] = 1;
    else if(p_count == 0 && n_count > 0)
        for(j=begin;j<end;j++)
            dec_values[perm[j]] = -1;
    else
    {
        svm_parameter subparam = *param;
        subparam.probability=0;
        subparam.C=1.0;
        subparam.nr_weight=2;
        subparam.weight_label = Malloc(int,2);
        subparam.weight = Malloc(double,2);
        subparam.weight_label[0]=+1;
        subparam.weight_label[1]=-1;
        subparam.weight[0]=Cp;
        subparam.weight[1]=Cn;
        struct PREFIX(model) *submodel = PREFIX(train)(&subprob,&subparam, status,
blas_functions);
        for(j=begin;j<end;j++)
        {
#ifdef _DENSE_REP
                PREFIX(predict_values)(submodel,(prob-
>x+perm[j]),&(dec_values[perm[j]]), blas_functions);
#else
```

```
            PREFIX(predict_values)(submodel,prob->x[perm[j]],&(dec_values[perm[j]]),
blas_functions);
#endif
            // ensure +1 -1 order; reason not using CV subroutine
            dec_values[perm[j]] *= submodel->label[0];
         }
         PREFIX(free_and_destroy_model)(&submodel);
         PREFIX(destroy_param)(&subparam);
      }
      free(subprob.x);
      free(subprob.y);
            free(subprob.W);
   }
   sigmoid_train(prob->l,dec_values,prob->y,probA,probB);
   free(dec_values);
   free(perm);
}

// Return parameter of a Laplace distribution
static double svm_svr_probability(
   const PREFIX(problem) *prob, const svm_parameter *param, BlasFunctions
*blas_functions)
{
   int i;
   int nr_fold = 5;
   double *ymv = Malloc(double,prob->l);
   double mae = 0;

   svm_parameter newparam = *param;
   newparam.probability = 0;
   newparam.random_seed = -1; // This is called from train, which already sets
                         // the seed.
   PREFIX(cross_validation)(prob,&newparam,nr_fold,ymv, blas_functions);
   for(i=0;i<prob->l;i++)
   {
      ymv[i]=prob->y[i]-ymv[i];
      mae += fabs(ymv[i]);
   }
   mae /= prob->l;
   double std=sqrt(2*mae*mae);
   int count=0;
   mae=0;
   for(i=0;i<prob->l;i++)
      if (fabs(ymv[i]) > 5*std)
```

```
        count=count+1;
      else
        mae+=fabs(ymv[i]);
  mae /= (prob->l-count);
  info("Prob. model for test data: target value = predicted value + z,\nz: Laplace distribution
e^(-|z|/sigma)/(2sigma),sigma= %g\n",mae);
  free(ymv);
  return mae;
}



// label: label name, start: begin of each class, count: #data of classes, perm: indices to the
original data
// perm, length l, must be allocated before calling this subroutine
static void svm_group_classes(const PREFIX(problem) *prob, int *nr_class_ret, int
**label_ret, int **start_ret, int **count_ret, int *perm)
{
  int l = prob->l;
  int max_nr_class = 16;
  int nr_class = 0;
  int *label = Malloc(int,max_nr_class);
  int *count = Malloc(int,max_nr_class);
  int *data_label = Malloc(int,l);
  int i, j, this_label, this_count;

  for(i=0;i<l;i++)
  {
    this_label = (int)prob->y[i];
    for(j=0;j<nr_class;j++)
    {
      if(this_label == label[j])
      {
        ++count[j];
        break;
      }
    }
    if(j == nr_class)
    {
      if(nr_class == max_nr_class)
      {
        max_nr_class *= 2;
        label = (int *)realloc(label,max_nr_class*sizeof(int));
        count = (int *)realloc(count,max_nr_class*sizeof(int));
```

```
        }
        label[nr_class] = this_label;
        count[nr_class] = 1;
        ++nr_class;
    }
}

    /*
     * Sort labels by straight insertion and apply the same
     * transformation to array count.
     */
    for(j=1; j<nr_class; j++)
    {
        i = j-1;
        this_label = label[j];
        this_count = count[j];
        while(i>=0 && label[i] > this_label)
        {
            label[i+1] = label[i];
            count[i+1] = count[i];
            i--;
        }
        label[i+1] = this_label;
        count[i+1] = this_count;
    }

    for (i=0; i<l; i++)
    {
        j = 0;
        this_label = (int)prob->y[i];
        while(this_label != label[j]){
            j ++;
        }
        data_label[i] = j;
    }

int *start = Malloc(int,nr_class);
start[0] = 0;
for(i=1;i<nr_class;i++)
    start[i] = start[i-1]+count[i-1];
for(i=0;i<l;i++)
{
    perm[start[data_label[i]]] = i;
    ++start[data_label[i]];
```

```
    }

    start[0] = 0;
    for(i=1;i<nr_class;i++)
        start[i] = start[i-1]+count[i-1];

    *nr_class_ret = nr_class;
    *label_ret = label;
    *start_ret = start;
    *count_ret = count;
    free(data_label);
}

} /* end namespace */

// Remove zero weighed data as libsvm and some liblinear solvers require C > 0.
//
static void remove_zero_weight(PREFIX(problem) *newprob, const PREFIX(problem) *prob)
{
    int i;
    int l = 0;
    for(i=0;i<prob->l;i++)
        if(prob->W[i] > 0) l++;
    *newprob = *prob;
    newprob->l = l;
#ifdef _DENSE_REP
    newprob->x = Malloc(PREFIX(node),l);
#else
        newprob->x = Malloc(PREFIX(node) *,l);
#endif
    newprob->y = Malloc(double,l);
    newprob->W = Malloc(double,l);

    int j = 0;
    for(i=0;i<prob->l;i++)
        if(prob->W[i] > 0)
        {
            newprob->x[j] = prob->x[i];
            newprob->y[j] = prob->y[i];
            newprob->W[j] = prob->W[i];
            j++;
        }
}
```

```
//
// Interface functions
//
PREFIX(model) *PREFIX(train)(const PREFIX(problem) *prob, const svm_parameter *param,
	int *status, BlasFunctions *blas_functions)
{
   PREFIX(problem) newprob;
   remove_zero_weight(&newprob, prob);
   prob = &newprob;

   PREFIX(model) *model = Malloc(PREFIX(model),1);
   model->param = *param;
   model->free_sv = 0; // XXX

   if(param->random_seed >= 0)
   {
      set_seed(param->random_seed);
   }

   if(param->svm_type == ONE_CLASS ||
     param->svm_type == EPSILON_SVR ||
     param->svm_type == NU_SVR)
   {
      // regression or one-class-svm
      model->nr_class = 2;
      model->label = NULL;
      model->nSV = NULL;
      model->probA = NULL; model->probB = NULL;
      model->sv_coef = Malloc(double *,1);

      if(param->probability &&
        (param->svm_type == EPSILON_SVR ||
         param->svm_type == NU_SVR))
      {
         model->probA = Malloc(double,1);
         model->probA[0] = NAMESPACE::svm_svr_probability(prob,param,blas_functions);
      }

            NAMESPACE::decision_function f = NAMESPACE::svm_train_one(prob,param,0,0,
status,blas_functions);
      model->rho = Malloc(double,1);
      model->rho[0] = f.rho;

      int nSV = 0;
```

```
		int i;
		for(i=0;i<prob->l;i++)
			if(fabs(f.alpha[i]) > 0) ++nSV;
		model->l = nSV;
#ifdef _DENSE_REP
		model->SV = Malloc(PREFIX(node),nSV);
#else
		model->SV = Malloc(PREFIX(node) *,nSV);
#endif
				model->sv_ind = Malloc(int, nSV);
		model->sv_coef[0] = Malloc(double, nSV);
		int j = 0;
		for(i=0;i<prob->l;i++)
			if(fabs(f.alpha[i]) > 0)
			{
				model->SV[j] = prob->x[i];
						model->sv_ind[j] = i;
				model->sv_coef[0][j] = f.alpha[i];
				++j;
			}

		free(f.alpha);
	}
	else
	{
		// classification
		int l = prob->l;
		int nr_class;
		int *label = NULL;
		int *start = NULL;
		int *count = NULL;
		int *perm = Malloc(int,l);

		// group training data of the same class
				NAMESPACE::svm_group_classes(prob,&nr_class,&label,&start,&count,perm);
#ifdef _DENSE_REP
		PREFIX(node) *x = Malloc(PREFIX(node),l);
#else
		PREFIX(node) **x = Malloc(PREFIX(node) *,l);
#endif
				double *W = Malloc(double, l);

		int i;
		for(i=0;i<l;i++)
```

```
			{
		x[i] = prob->x[perm[i]];
		W[i] = prob->W[perm[i]];
			}

	// calculate weighted C

	double *weighted_C = Malloc(double, nr_class);
	for(i=0;i<nr_class;i++)
		weighted_C[i] = param->C;
	for(i=0;i<param->nr_weight;i++)
	{
		int j;
		for(j=0;j<nr_class;j++)
			if(param->weight_label[i] == label[j])
				break;
		if(j == nr_class)
			fprintf(stderr,"warning: class label %d specified in weight is not found\n", param-
>weight_label[i]);
		else
			weighted_C[j] *= param->weight[i];
	}

	// train k*(k-1)/2 models

	bool *nonzero = Malloc(bool,l);
	for(i=0;i<l;i++)
		nonzero[i] = false;
			NAMESPACE::decision_function *f =
Malloc(NAMESPACE::decision_function,nr_class*(nr_class-1)/2);

	double *probA=NULL,*probB=NULL;
	if (param->probability)
	{
		probA=Malloc(double,nr_class*(nr_class-1)/2);
		probB=Malloc(double,nr_class*(nr_class-1)/2);
	}

	int p = 0;
	for(i=0;i<nr_class;i++)
		for(int j=i+1;j<nr_class;j++)
		{
			PREFIX(problem) sub_prob;
			int si = start[i], sj = start[j];
```

```
            int ci = count[i], cj = count[j];
            sub_prob.l = ci+cj;
#ifdef _DENSE_REP
            sub_prob.x = Malloc(PREFIX(node),sub_prob.l);
#else
            sub_prob.x = Malloc(PREFIX(node) *,sub_prob.l);
#endif
            sub_prob.W = Malloc(double,sub_prob.l);
            sub_prob.y = Malloc(double,sub_prob.l);
            int k;
            for(k=0;k<ci;k++)
            {
               sub_prob.x[k] = x[si+k];
               sub_prob.y[k] = +1;
               sub_prob.W[k] = W[si+k];
            }
            for(k=0;k<cj;k++)
            {
               sub_prob.x[ci+k] = x[sj+k];
               sub_prob.y[ci+k] = -1;
               sub_prob.W[ci+k] = W[sj+k];
            }

            if(param->probability)

NAMESPACE::svm_binary_svc_probability(&sub_prob,param,weighted_C[i],weighted_C[j],pr
obA[p],probB[p], status, blas_functions);

            f[p] = NAMESPACE::svm_train_one(&sub_prob,param,weighted_C[i],weighted_C[j],
status, blas_functions);
            for(k=0;k<ci;k++)
               if(!nonzero[si+k] && fabs(f[p].alpha[k]) > 0)
                  nonzero[si+k] = true;
            for(k=0;k<cj;k++)
               if(!nonzero[sj+k] && fabs(f[p].alpha[ci+k]) > 0)
                  nonzero[sj+k] = true;
            free(sub_prob.x);
            free(sub_prob.y);
                        free(sub_prob.W);
            ++p;
         }

      // build output
```

```
model->nr_class = nr_class;

model->label = Malloc(int,nr_class);
for(i=0;i<nr_class;i++)
   model->label[i] = label[i];

model->rho = Malloc(double,nr_class*(nr_class-1)/2);
for(i=0;i<nr_class*(nr_class-1)/2;i++)
   model->rho[i] = f[i].rho;

if(param->probability)
{
   model->probA = Malloc(double,nr_class*(nr_class-1)/2);
   model->probB = Malloc(double,nr_class*(nr_class-1)/2);
   for(i=0;i<nr_class*(nr_class-1)/2;i++)
   {
      model->probA[i] = probA[i];
      model->probB[i] = probB[i];
   }
}
else
{
   model->probA=NULL;
   model->probB=NULL;
}

int total_sv = 0;
int *nz_count = Malloc(int,nr_class);
model->nSV = Malloc(int,nr_class);
for(i=0;i<nr_class;i++)
{
   int nSV = 0;
   for(int j=0;j<count[i];j++)
      if(nonzero[start[i]+j])
      {
         ++nSV;
         ++total_sv;
      }
   model->nSV[i] = nSV;
   nz_count[i] = nSV;
}

      info("Total nSV = %d\n",total_sv);
```

```
	model->l = total_sv;
		model->sv_ind = Malloc(int, total_sv);
#ifdef _DENSE_REP
	model->SV = Malloc(PREFIX(node),total_sv);
#else
	model->SV = Malloc(PREFIX(node) *,total_sv);
#endif
	p = 0;
	for(i=0;i<l;i++) {
	   if(nonzero[i]) {
				model->SV[p] = x[i];
				model->sv_ind[p] = perm[i];
				++p;
			}
		}

	int *nz_start = Malloc(int,nr_class);
	nz_start[0] = 0;
	for(i=1;i<nr_class;i++)
	   nz_start[i] = nz_start[i-1]+nz_count[i-1];

	model->sv_coef = Malloc(double *,nr_class-1);
	for(i=0;i<nr_class-1;i++)
	   model->sv_coef[i] = Malloc(double,total_sv);

	p = 0;
	for(i=0;i<nr_class;i++)
	   for(int j=i+1;j<nr_class;j++)
	   {
	      // classifier (i,j): coefficients with
	      // i are in sv_coef[j-1][nz_start[i]...],
	      // j are in sv_coef[i][nz_start[j]...]

	      int si = start[i];
	      int sj = start[j];
	      int ci = count[i];
	      int cj = count[j];

	      int q = nz_start[i];
	      int k;
	      for(k=0;k<ci;k++)
	         if(nonzero[si+k])
	            model->sv_coef[j-1][q++] = f[p].alpha[k];
	      q = nz_start[j];
```

```
            for(k=0;k<cj;k++)
              if(nonzero[sj+k])
                model->sv_coef[i][q++] = f[p].alpha[ci+k];
            ++p;
          }

        free(label);
        free(probA);
        free(probB);
        free(count);
        free(perm);
        free(start);
            free(W);
        free(x);
        free(weighted_C);
        free(nonzero);
        for(i=0;i<nr_class*(nr_class-1)/2;i++)
          free(f[i].alpha);
        free(f);
        free(nz_count);
        free(nz_start);
    }
    free(newprob.x);
    free(newprob.y);
    free(newprob.W);
    return model;
}

// Stratified cross validation
void PREFIX(cross_validation)(const PREFIX(problem) *prob, const svm_parameter *param,
int nr_fold, double *target, BlasFunctions *blas_functions)
{
    int i;
    int *fold_start = Malloc(int,nr_fold+1);
    int l = prob->l;
    int *perm = Malloc(int,l);
    int nr_class;
    if(param->random_seed >= 0)
    {
        set_seed(param->random_seed);
    }

    // stratified cv may not give leave-one-out rate
    // Each class to l folds -> some folds may have zero elements
```

```
if((param->svm_type == C_SVC ||
   param->svm_type == NU_SVC) && nr_fold < l)
{
   int *start = NULL;
   int *label = NULL;
   int *count = NULL;
      NAMESPACE::svm_group_classes(prob,&nr_class,&label,&start,&count,perm);

   // random shuffle and then data grouped by fold using the array perm
   int *fold_count = Malloc(int,nr_fold);
   int c;
   int *index = Malloc(int,l);
   for(i=0;i<l;i++)
      index[i]=perm[i];
   for (c=0; c<nr_class; c++)
      for(i=0;i<count[c];i++)
      {
         int j = i+bounded_rand_int(count[c]-i);
         swap(index[start[c]+j],index[start[c]+i]);
      }
   for(i=0;i<nr_fold;i++)
   {
      fold_count[i] = 0;
      for (c=0; c<nr_class;c++)
         fold_count[i]+=(i+1)*count[c]/nr_fold-i*count[c]/nr_fold;
   }
   fold_start[0]=0;
   for (i=1;i<=nr_fold;i++)
      fold_start[i] = fold_start[i-1]+fold_count[i-1];
   for (c=0; c<nr_class;c++)
      for(i=0;i<nr_fold;i++)
      {
         int begin = start[c]+i*count[c]/nr_fold;
         int end = start[c]+(i+1)*count[c]/nr_fold;
         for(int j=begin;j<end;j++)
         {
            perm[fold_start[i]] = index[j];
            fold_start[i]++;
         }
      }
   fold_start[0]=0;
   for (i=1;i<=nr_fold;i++)
      fold_start[i] = fold_start[i-1]+fold_count[i-1];
   free(start);
```

```
      free(label);
      free(count);
      free(index);
      free(fold_count);
   }
   else
   {
      for(i=0;i<l;i++) perm[i]=i;
      for(i=0;i<l;i++)
      {
         int j = i+bounded_rand_int(l-i);
         swap(perm[i],perm[j]);
      }
      for(i=0;i<=nr_fold;i++)
         fold_start[i]=i*l/nr_fold;
   }

   for(i=0;i<nr_fold;i++)
   {
      int begin = fold_start[i];
      int end = fold_start[i+1];
      int j,k;
      struct PREFIX(problem) subprob;

      subprob.l = l-(end-begin);
#ifdef _DENSE_REP
      subprob.x = Malloc(struct PREFIX(node),subprob.l);
#else
      subprob.x = Malloc(struct PREFIX(node)*,subprob.l);
#endif
      subprob.y = Malloc(double,subprob.l);
      subprob.W = Malloc(double,subprob.l);

      k=0;
      for(j=0;j<begin;j++)
      {
         subprob.x[k] = prob->x[perm[j]];
         subprob.y[k] = prob->y[perm[j]];
         subprob.W[k] = prob->W[perm[j]];
         ++k;
      }
      for(j=end;j<l;j++)
      {
         subprob.x[k] = prob->x[perm[j]];
```

```
         subprob.y[k] = prob->y[perm[j]];
         subprob.W[k] = prob->W[perm[j]];
         ++k;
      }
            int dummy_status = 0; // IGNORES TIMEOUT ERRORS
      struct PREFIX(model) *submodel = PREFIX(train)(&subprob,param, &dummy_status,
blas_functions);
      if(param->probability &&
        (param->svm_type == C_SVC || param->svm_type == NU_SVC))
      {
         double *prob_estimates=Malloc(double, PREFIX(get_nr_class)(submodel));
         for(j=begin;j<end;j++)
#ifdef _DENSE_REP
            target[perm[j]] = PREFIX(predict_probability)(submodel,(prob->x +
perm[j]),prob_estimates, blas_functions);
#else
                        target[perm[j]] = PREFIX(predict_probability)(submodel,prob-
>x[perm[j]],prob_estimates, blas_functions);
#endif
         free(prob_estimates);
      }
      else
         for(j=begin;j<end;j++)
#ifdef _DENSE_REP
            target[perm[j]] = PREFIX(predict)(submodel,prob->x+perm[j],blas_functions);
#else
            target[perm[j]] = PREFIX(predict)(submodel,prob->x[perm[j]],blas_functions);
#endif
      PREFIX(free_and_destroy_model)(&submodel);
      free(subprob.x);
      free(subprob.y);
            free(subprob.W);
   }
   free(fold_start);
   free(perm);
}


int PREFIX(get_svm_type)(const PREFIX(model) *model)
{
   return model->param.svm_type;
}

int PREFIX(get_nr_class)(const PREFIX(model) *model)
```

```
{
   return model->nr_class;
}

void PREFIX(get_labels)(const PREFIX(model) *model, int* label)
{
   if (model->label != NULL)
      for(int i=0;i<model->nr_class;i++)
         label[i] = model->label[i];
}

double PREFIX(get_svr_probability)(const PREFIX(model) *model)
{
   if ((model->param.svm_type == EPSILON_SVR || model->param.svm_type == NU_SVR) &&
      model->probA!=NULL)
      return model->probA[0];
   else
   {
      fprintf(stderr,"Model doesn't contain information for SVR probability inference\n");
      return 0;
   }
}

double PREFIX(predict_values)(const PREFIX(model) *model, const PREFIX(node) *x, double*
dec_values, BlasFunctions *blas_functions)
{
   int i;
   if(model->param.svm_type == ONE_CLASS ||
     model->param.svm_type == EPSILON_SVR ||
     model->param.svm_type == NU_SVR)
   {
      double *sv_coef = model->sv_coef[0];
      double sum = 0;

      for(i=0;i<model->l;i++)
#ifdef _DENSE_REP
               sum += sv_coef[i] * NAMESPACE::Kernel::k_function(x,model->SV+i,model-
>param,blas_functions);
#else
            sum += sv_coef[i] * NAMESPACE::Kernel::k_function(x,model->SV[i],model-
>param,blas_functions);
#endif
      sum -= model->rho[0];
      *dec_values = sum;
```

```
        if(model->param.svm_type == ONE_CLASS)
            return (sum>0)?1:-1;
        else
            return sum;
    }
    else
    {
        int nr_class = model->nr_class;
        int l = model->l;

        double *kvalue = Malloc(double,l);
        for(i=0;i<l;i++)
#ifdef _DENSE_REP
                kvalue[i] = NAMESPACE::Kernel::k_function(x,model->SV+i,model-
>param,blas_functions);
#else
            kvalue[i] = NAMESPACE::Kernel::k_function(x,model->SV[i],model-
>param,blas_functions);
#endif

        int *start = Malloc(int,nr_class);
        start[0] = 0;
        for(i=1;i<nr_class;i++)
            start[i] = start[i-1]+model->nSV[i-1];

        int *vote = Malloc(int,nr_class);
        for(i=0;i<nr_class;i++)
            vote[i] = 0;

        int p=0;
        for(i=0;i<nr_class;i++)
            for(int j=i+1;j<nr_class;j++)
            {
                double sum = 0;
                int si = start[i];
                int sj = start[j];
                int ci = model->nSV[i];
                int cj = model->nSV[j];

                int k;
                double *coef1 = model->sv_coef[j-1];
                double *coef2 = model->sv_coef[i];
                for(k=0;k<ci;k++)
```

```
            sum += coef1[si+k] * kvalue[si+k];
          for(k=0;k<cj;k++)
            sum += coef2[sj+k] * kvalue[sj+k];
          sum -= model->rho[p];
          dec_values[p] = sum;

          if(dec_values[p] > 0)
            ++vote[i];
          else
            ++vote[j];
          p++;
        }

      int vote_max_idx = 0;
      for(i=1;i<nr_class;i++)
        if(vote[i] > vote[vote_max_idx])
          vote_max_idx = i;

      free(kvalue);
      free(start);
      free(vote);
      return model->label[vote_max_idx];
    }
}

double PREFIX(predict)(const PREFIX(model) *model, const PREFIX(node) *x, BlasFunctions
*blas_functions)
{
    int nr_class = model->nr_class;
    double *dec_values;
    if(model->param.svm_type == ONE_CLASS ||
      model->param.svm_type == EPSILON_SVR ||
      model->param.svm_type == NU_SVR)
       dec_values = Malloc(double, 1);
    else
       dec_values = Malloc(double, nr_class*(nr_class-1)/2);
    double pred_result = PREFIX(predict_values)(model, x, dec_values, blas_functions);
    free(dec_values);
    return pred_result;
}

double PREFIX(predict_probability)(
    const PREFIX(model) *model, const PREFIX(node) *x, double *prob_estimates,
BlasFunctions *blas_functions)
```

```
{
   if ((model->param.svm_type == C_SVC || model->param.svm_type == NU_SVC) &&
      model->probA!=NULL && model->probB!=NULL)
   {
      int i;
      int nr_class = model->nr_class;
      double *dec_values = Malloc(double, nr_class*(nr_class-1)/2);
      PREFIX(predict_values)(model, x, dec_values, blas_functions);

      double min_prob=1e-7;
      double **pairwise_prob=Malloc(double *,nr_class);
      for(i=0;i<nr_class;i++)
         pairwise_prob[i]=Malloc(double,nr_class);
      int k=0;
      for(i=0;i<nr_class;i++)
         for(int j=i+1;j<nr_class;j++)
         {

pairwise_prob[i][j]=min(max(NAMESPACE::sigmoid_predict(dec_values[k],model-
>probA[k],model->probB[k]),min_prob),1-min_prob);
            pairwise_prob[j][i]=1-pairwise_prob[i][j];
            k++;
         }
            NAMESPACE::multiclass_probability(nr_class,pairwise_prob,prob_estimates);

      int prob_max_idx = 0;
      for(i=1;i<nr_class;i++)
         if(prob_estimates[i] > prob_estimates[prob_max_idx])
            prob_max_idx = i;
      for(i=0;i<nr_class;i++)
         free(pairwise_prob[i]);
      free(dec_values);
      free(pairwise_prob);
      return model->label[prob_max_idx];
   }
   else
      return PREFIX(predict)(model, x, blas_functions);
}


void PREFIX(free_model_content)(PREFIX(model)* model_ptr)
{
   if(model_ptr->free_sv && model_ptr->l > 0 && model_ptr->SV != NULL)
#ifdef _DENSE_REP
```

```
    for (int i = 0; i < model_ptr->l; i++)
      free(model_ptr->SV[i].values);
#else
    free((void *)(model_ptr->SV[0]));
#endif

  if(model_ptr->sv_coef)
  {
    for(int i=0;i<model_ptr->nr_class-1;i++)
      free(model_ptr->sv_coef[i]);
  }

  free(model_ptr->SV);
  model_ptr->SV = NULL;

  free(model_ptr->sv_coef);
  model_ptr->sv_coef = NULL;

  free(model_ptr->sv_ind);
  model_ptr->sv_ind = NULL;

  free(model_ptr->rho);
  model_ptr->rho = NULL;

  free(model_ptr->label);
  model_ptr->label= NULL;

  free(model_ptr->probA);
  model_ptr->probA = NULL;

  free(model_ptr->probB);
  model_ptr->probB= NULL;

  free(model_ptr->nSV);
  model_ptr->nSV = NULL;
}

void PREFIX(free_and_destroy_model)(PREFIX(model)** model_ptr_ptr)
{
  if(model_ptr_ptr != NULL && *model_ptr_ptr != NULL)
  {
    PREFIX(free_model_content)(*model_ptr_ptr);
    free(*model_ptr_ptr);
    *model_ptr_ptr = NULL;
```

```
  }
}

void PREFIX(destroy_param)(svm_parameter* param)
{
  free(param->weight_label);
  free(param->weight);
}

const char *PREFIX(check_parameter)(const PREFIX(problem) *prob, const svm_parameter
*param)
{
  // svm_type

  int svm_type = param->svm_type;
  if(svm_type != C_SVC &&
    svm_type != NU_SVC &&
    svm_type != ONE_CLASS &&
    svm_type != EPSILON_SVR &&
    svm_type != NU_SVR)
     return "unknown svm type";

  // kernel_type, degree

  int kernel_type = param->kernel_type;
  if(kernel_type != LINEAR &&
    kernel_type != POLY &&
    kernel_type != RBF &&
    kernel_type != SIGMOID &&
    kernel_type != PRECOMPUTED)
     return "unknown kernel type";

  if(param->gamma < 0)
     return "gamma < 0";

  if(param->degree < 0)
     return "degree of polynomial kernel < 0";

  // cache_size,eps,C,nu,p,shrinking

  if(param->cache_size <= 0)
     return "cache_size <= 0";

  if(param->eps <= 0)
```

```
        return "eps <= 0";

    if(svm_type == C_SVC ||
       svm_type == EPSILON_SVR ||
       svm_type == NU_SVR)
        if(param->C <= 0)
            return "C <= 0";

    if(svm_type == NU_SVC ||
       svm_type == ONE_CLASS ||
       svm_type == NU_SVR)
        if(param->nu <= 0 || param->nu > 1)
            return "nu <= 0 or nu > 1";

    if(svm_type == EPSILON_SVR)
        if(param->p < 0)
            return "p < 0";

    if(param->shrinking != 0 &&
       param->shrinking != 1)
        return "shrinking != 0 and shrinking != 1";

    if(param->probability != 0 &&
       param->probability != 1)
        return "probability != 0 and probability != 1";

    if(param->probability == 1 &&
       svm_type == ONE_CLASS)
        return "one-class SVM probability output not supported yet";


    // check whether nu-svc is feasible

    if(svm_type == NU_SVC)
    {
        int l = prob->l;
        int max_nr_class = 16;
        int nr_class = 0;
        int *label = Malloc(int,max_nr_class);
        double *count = Malloc(double,max_nr_class);

        int i;
        for(i=0;i<l;i++)
        {
```

```
		int this_label = (int)prob->y[i];
		int j;
		for(j=0;j<nr_class;j++)
			if(this_label == label[j])
			{
				count[j] += prob->W[i];
				break;
			}
		if(j == nr_class)
		{
			if(nr_class == max_nr_class)
			{
				max_nr_class *= 2;
				label = (int *)realloc(label,max_nr_class*sizeof(int));
				count = (double *)realloc(count,max_nr_class*sizeof(double));

			}
			label[nr_class] = this_label;
			count[nr_class] = prob->W[i];
			++nr_class;
		}
	}

	for(i=0;i<nr_class;i++)
	{
		double n1 = count[i];
		for(int j=i+1;j<nr_class;j++)
		{
			double n2 = count[j];
			if(param->nu*(n1+n2)/2 > min(n1,n2))
			{
				free(label);
				free(count);
				return "specified nu is infeasible";
			}
		}
	}
	free(label);
	free(count);
}

if(svm_type == C_SVC ||
   svm_type == EPSILON_SVR ||
   svm_type == NU_SVR ||
```

```
    svm_type == ONE_CLASS)
  {
    PREFIX(problem) newprob;
    // filter samples with negative and null weights
    remove_zero_weight(&newprob, prob);

    char* msg = NULL;
    // all samples were removed
    if(newprob.l == 0)
      msg =  "Invalid input - all samples have zero or negative weights.";
    else if(prob->l != newprob.l &&
        svm_type == C_SVC)
    {
      bool only_one_label = true;
      int first_label = newprob.y[0];
      for(int i=1;i<newprob.l;i++)
      {
        if(newprob.y[i] != first_label)
        {
          only_one_label = false;
          break;
        }
      }
      if(only_one_label == true)
        msg = "Invalid input - all samples with positive weights have the same label.";
    }

    free(newprob.x);
    free(newprob.y);
    free(newprob.W);
    if(msg != NULL)
      return msg;
  }
  return NULL;
}

void PREFIX(set_print_string_function)(void (*print_func)(const char *))
{
  if(print_func == NULL)
    svm_print_string = &print_string_stdout;
  else
    svm_print_string = print_func;
}
```

Naïve Bayes source code:

URL: https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/naive_bayes.py

```
# -*- coding: utf-8 -*-

"""
The :mod:`sklearn.naive_bayes` module implements Naive Bayes algorithms. These
are supervised learning methods based on applying Bayes' theorem with strong
(naive) feature independence assumptions.
"""

# Author: Vincent Michel <vincent.michel@inria.fr>
#         Minor fixes by Fabian Pedregosa
#         Amit Aides <amitibo@tx.technion.ac.il>
#         Yehuda Finkelstein <yehudaf@tx.technion.ac.il>
#         Lars Buitinck
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#         (parts based on earlier work by Mathieu Blondel)
#
# License: BSD 3 clause
import warnings

from abc import ABCMeta, abstractmethod


import numpy as np
from scipy.special import logsumexp

from .base import BaseEstimator, ClassifierMixin
from .preprocessing import binarize
from .preprocessing import LabelBinarizer
from .preprocessing import label_binarize
from .utils import deprecated
from .utils.extmath import safe_sparse_dot
from .utils.multiclass import _check_partial_fit_first_call
from .utils.validation import check_is_fitted, check_non_negative
from .utils.validation import _check_sample_weight


__all__ = [
    "BernoulliNB",
    "GaussianNB",
    "MultinomialNB",
```

```
    "ComplementNB",
    "CategoricalNB",
]


class _BaseNB(ClassifierMixin, BaseEstimator, metaclass=ABCMeta):
    """Abstract base class for naive Bayes estimators"""

    @abstractmethod
    def _joint_log_likelihood(self, X):
        """Compute the unnormalized posterior log probability of X

        I.e. ``log P(c) + log P(x|c)`` for all rows x of X, as an array-like of
        shape (n_classes, n_samples).

        Input is passed to _joint_log_likelihood as-is by predict,
        predict_proba and predict_log_proba.
        """

    @abstractmethod
    def _check_X(self, X):
        """To be overridden in subclasses with the actual checks.

        Only used in predict* methods.
        """

    def predict(self, X):
        """
        Perform classification on an array of test vectors X.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            The input samples.

        Returns
        -------
        C : ndarray of shape (n_samples,)
            Predicted target values for X.
        """
        check_is_fitted(self)
        X = self._check_X(X)
        jll = self._joint_log_likelihood(X)
        return self.classes_[np.argmax(jll, axis=1)]
```

```
    def predict_log_proba(self, X):
        """
        Return log-probability estimates for the test vector X.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            The input samples.

        Returns
        -------
        C : array-like of shape (n_samples, n_classes)
            Returns the log-probability of the samples for each class in
            the model. The columns correspond to the classes in sorted
            order, as they appear in the attribute :term:`classes_`.
        """
        check_is_fitted(self)
        X = self._check_X(X)
        jll = self._joint_log_likelihood(X)
        # normalize by P(x) = P(f_1, ..., f_n)
        log_prob_x = logsumexp(jll, axis=1)
        return jll - np.atleast_2d(log_prob_x).T

    def predict_proba(self, X):
        """
        Return probability estimates for the test vector X.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            The input samples.

        Returns
        -------
        C : array-like of shape (n_samples, n_classes)
            Returns the probability of the samples for each class in
            the model. The columns correspond to the classes in sorted
            order, as they appear in the attribute :term:`classes_`.
        """
        return np.exp(self.predict_log_proba(X))


class GaussianNB(_BaseNB):
```

```
"""
Gaussian Naive Bayes (GaussianNB).

Can perform online updates to model parameters via :meth:`partial_fit`.
For details on algorithm used to update feature means and variance online,
see Stanford CS tech report STAN-CS-79-773 by Chan, Golub, and LeVeque:

    http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf

Read more in the :ref:`User Guide <gaussian_naive_bayes>`.

Parameters
----------
priors : array-like of shape (n_classes,)
    Prior probabilities of the classes. If specified the priors are not
    adjusted according to the data.

var_smoothing : float, default=1e-9
    Portion of the largest variance of all features that is added to
    variances for calculation stability.

    .. versionadded:: 0.20

Attributes
----------
class_count_ : ndarray of shape (n_classes,)
    number of training samples observed in each class.

class_prior_ : ndarray of shape (n_classes,)
    probability of each class.

classes_ : ndarray of shape (n_classes,)
    class labels known to the classifier.

epsilon_ : float
    absolute additive value to variances.

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
```

```
    has feature names that are all strings.

    .. versionadded:: 1.0

sigma_ : ndarray of shape (n_classes, n_features)
    Variance of each feature per class.

    .. deprecated:: 1.0
       `sigma_` is deprecated in 1.0 and will be removed in 1.2.
       Use `var_` instead.

var_ : ndarray of shape (n_classes, n_features)
    Variance of each feature per class.

    .. versionadded:: 1.0

theta_ : ndarray of shape (n_classes, n_features)
    mean of each feature per class.

See Also
--------
BernoulliNB : Naive Bayes classifier for multivariate Bernoulli models.
CategoricalNB : Naive Bayes classifier for categorical features.
ComplementNB : Complement Naive Bayes classifier.
MultinomialNB : Naive Bayes classifier for multinomial models.

Examples
--------
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> Y = np.array([1, 1, 1, 2, 2, 2])
>>> from sklearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(X, Y)
GaussianNB()
>>> print(clf.predict([[-0.8, -1]]))
[1]
>>> clf_pf = GaussianNB()
>>> clf_pf.partial_fit(X, Y, np.unique(Y))
GaussianNB()
>>> print(clf_pf.predict([[-0.8, -1]]))
[1]
"""
```

```
def __init__(self, *, priors=None, var_smoothing=1e-9):
    self.priors = priors
    self.var_smoothing = var_smoothing

def fit(self, X, y, sample_weight=None):
    """Fit Gaussian Naive Bayes according to X, y.

    Parameters
    ----------
    X : array-like of shape (n_samples, n_features)
        Training vectors, where `n_samples` is the number of samples
        and `n_features` is the number of features.

    y : array-like of shape (n_samples,)
        Target values.

    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).

        .. versionadded:: 0.17
           Gaussian Naive Bayes supports fitting with *sample_weight*.

    Returns
    -------
    self : object
        Returns the instance itself.
    """
    y = self._validate_data(y=y)
    return self._partial_fit(
        X, y, np.unique(y), _refit=True, sample_weight=sample_weight
    )

def _check_X(self, X):
    """Validate X, used only in predict* methods."""
    return self._validate_data(X, reset=False)

@staticmethod
def _update_mean_variance(n_past, mu, var, X, sample_weight=None):
    """Compute online update of Gaussian mean and variance.

    Given starting sample count, mean, and variance, a new set of
    points X, and optionally sample weights, return the updated mean and
    variance. (NB - each dimension (column) in X is treated as independent
    -- you get variance, not covariance).
```

```
Can take scalar mean and variance, or vector mean and variance to
simultaneously update a number of independent Gaussians.

See Stanford CS tech report STAN-CS-79-773 by Chan, Golub, and LeVeque:

http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf

Parameters
----------
n_past : int
    Number of samples represented in old mean and variance. If sample
    weights were given, this should contain the sum of sample
    weights represented in old mean and variance.

mu : array-like of shape (number of Gaussians,)
    Means for Gaussians in original set.

var : array-like of shape (number of Gaussians,)
    Variances for Gaussians in original set.

sample_weight : array-like of shape (n_samples,), default=None
    Weights applied to individual samples (1. for unweighted).

Returns
-------
total_mu : array-like of shape (number of Gaussians,)
    Updated mean for each Gaussian over the combined set.

total_var : array-like of shape (number of Gaussians,)
    Updated variance for each Gaussian over the combined set.
"""
if X.shape[0] == 0:
    return mu, var

# Compute (potentially weighted) mean and variance of new datapoints
if sample_weight is not None:
    n_new = float(sample_weight.sum())
    new_mu = np.average(X, axis=0, weights=sample_weight)
    new_var = np.average((X - new_mu) ** 2, axis=0, weights=sample_weight)
else:
    n_new = X.shape[0]
    new_var = np.var(X, axis=0)
    new_mu = np.mean(X, axis=0)
```

```
    if n_past == 0:
        return new_mu, new_var

    n_total = float(n_past + n_new)

    # Combine mean of old and new data, taking into consideration
    # (weighted) number of observations
    total_mu = (n_new * new_mu + n_past * mu) / n_total

    # Combine variance of old and new data, taking into consideration
    # (weighted) number of observations. This is achieved by combining
    # the sum-of-squared-differences (ssd)
    old_ssd = n_past * var
    new_ssd = n_new * new_var
    total_ssd = old_ssd + new_ssd + (n_new * n_past / n_total) * (mu - new_mu) ** 2
    total_var = total_ssd / n_total

    return total_mu, total_var

def partial_fit(self, X, y, classes=None, sample_weight=None):
    """Incremental fit on a batch of samples.

    This method is expected to be called several times consecutively
    on different chunks of a dataset so as to implement out-of-core
    or online learning.

    This is especially useful when the whole dataset is too big to fit in
    memory at once.

    This method has some performance and numerical stability overhead,
    hence it is better to call partial_fit on chunks of data that are
    as large as possible (as long as fitting in the memory budget) to
    hide the overhead.

    Parameters
    ----------
    X : array-like of shape (n_samples, n_features)
        Training vectors, where `n_samples` is the number of samples and
        `n_features` is the number of features.

    y : array-like of shape (n_samples,)
        Target values.
```

```
    classes : array-like of shape (n_classes,), default=None
        List of all the classes that can possibly appear in the y vector.

        Must be provided at the first call to partial_fit, can be omitted
        in subsequent calls.

    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).

        .. versionadded:: 0.17

    Returns
    -------
    self : object
        Returns the instance itself.
    """
    return self._partial_fit(
        X, y, classes, _refit=False, sample_weight=sample_weight
    )

def _partial_fit(self, X, y, classes=None, _refit=False, sample_weight=None):
    """Actual implementation of Gaussian NB fitting.

    Parameters
    ----------
    X : array-like of shape (n_samples, n_features)
        Training vectors, where `n_samples` is the number of samples and
        `n_features` is the number of features.

    y : array-like of shape (n_samples,)
        Target values.

    classes : array-like of shape (n_classes,), default=None
        List of all the classes that can possibly appear in the y vector.

        Must be provided at the first call to partial_fit, can be omitted
        in subsequent calls.

    _refit : bool, default=False
        If true, act as though this were the first time we called
        _partial_fit (ie, throw away any past fitting and start over).

    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).
```

```
Returns
-------
self : object
"""
if _refit:
    self.classes_ = None

first_call = _check_partial_fit_first_call(self, classes)
X, y = self._validate_data(X, y, reset=first_call)
if sample_weight is not None:
    sample_weight = _check_sample_weight(sample_weight, X)

# If the ratio of data variance between dimensions is too small, it
# will cause numerical errors. To address this, we artificially
# boost the variance by epsilon, a small fraction of the standard
# deviation of the largest dimension.
self.epsilon_ = self.var_smoothing * np.var(X, axis=0).max()

if first_call:
    # This is the first call to partial_fit:
    # initialize various cumulative counters
    n_features = X.shape[1]
    n_classes = len(self.classes_)
    self.theta_ = np.zeros((n_classes, n_features))
    self.var_ = np.zeros((n_classes, n_features))

    self.class_count_ = np.zeros(n_classes, dtype=np.float64)

    # Initialise the class prior
    # Take into account the priors
    if self.priors is not None:
        priors = np.asarray(self.priors)
        # Check that the provide prior match the number of classes
        if len(priors) != n_classes:
            raise ValueError("Number of priors must match number of classes.")
        # Check that the sum is 1
        if not np.isclose(priors.sum(), 1.0):
            raise ValueError("The sum of the priors should be 1.")
        # Check that the prior are non-negative
        if (priors < 0).any():
            raise ValueError("Priors must be non-negative.")
        self.class_prior_ = priors
    else:
```

```
            # Initialize the priors to zeros for each class
            self.class_prior_ = np.zeros(len(self.classes_), dtype=np.float64)
    else:
        if X.shape[1] != self.theta_.shape[1]:
            msg = "Number of features %d does not match previous data %d."
            raise ValueError(msg % (X.shape[1], self.theta_.shape[1]))
        # Put epsilon back in each time
        self.var_[:, :] -= self.epsilon_

    classes = self.classes_

    unique_y = np.unique(y)
    unique_y_in_classes = np.in1d(unique_y, classes)

    if not np.all(unique_y_in_classes):
        raise ValueError(
            "The target label(s) %s in y do not exist in the initial classes %s"
            % (unique_y[~unique_y_in_classes], classes)
        )

    for y_i in unique_y:
        i = classes.searchsorted(y_i)
        X_i = X[y == y_i, :]

        if sample_weight is not None:
            sw_i = sample_weight[y == y_i]
            N_i = sw_i.sum()
        else:
            sw_i = None
            N_i = X_i.shape[0]

        new_theta, new_sigma = self._update_mean_variance(
            self.class_count_[i], self.theta_[i, :], self.var_[i, :], X_i, sw_i
        )

        self.theta_[i, :] = new_theta
        self.var_[i, :] = new_sigma
        self.class_count_[i] += N_i

    self.var_[:, :] += self.epsilon_

    # Update if only no priors is provided
    if self.priors is None:
        # Empirical prior, with sample_weight taken into account
```

```
        self.class_prior_ = self.class_count_ / self.class_count_.sum()

    return self

  def _joint_log_likelihood(self, X):
    joint_log_likelihood = []
    for i in range(np.size(self.classes_)):
      jointi = np.log(self.class_prior_[i])
      n_ij = -0.5 * np.sum(np.log(2.0 * np.pi * self.var_[i, :]))
      n_ij -= 0.5 * np.sum(((X - self.theta_[i, :]) ** 2) / (self.var_[i, :]), 1)
      joint_log_likelihood.append(jointi + n_ij)

    joint_log_likelihood = np.array(joint_log_likelihood).T
    return joint_log_likelihood

  @deprecated(  # type: ignore
    "Attribute `sigma_` was deprecated in 1.0 and will be removed in"
    "1.2. Use `var_` instead."
  )
  @property
  def sigma_(self):
    return self.var_


_ALPHA_MIN = 1e-10


class _BaseDiscreteNB(_BaseNB):
  """Abstract base class for naive Bayes on discrete/categorical data

  Any estimator based on this class should provide:

  __init__
  _joint_log_likelihood(X) as per _BaseNB
  """

  def _check_X(self, X):
    """Validate X, used only in predict* methods."""
    return self._validate_data(X, accept_sparse="csr", reset=False)

  def _check_X_y(self, X, y, reset=True):
    """Validate X and y in fit methods."""
    return self._validate_data(X, y, accept_sparse="csr", reset=reset)
```

```
def _update_class_log_prior(self, class_prior=None):
    n_classes = len(self.classes_)
    if class_prior is not None:
        if len(class_prior) != n_classes:
            raise ValueError("Number of priors must match number of classes.")
        self.class_log_prior_ = np.log(class_prior)
    elif self.fit_prior:
        with warnings.catch_warnings():
            # silence the warning when count is 0 because class was not yet
            # observed
            warnings.simplefilter("ignore", RuntimeWarning)
            log_class_count = np.log(self.class_count_)

        # empirical prior, with sample_weight taken into account
        self.class_log_prior_ = log_class_count - np.log(self.class_count_.sum())
    else:
        self.class_log_prior_ = np.full(n_classes, -np.log(n_classes))

def _check_alpha(self):
    if np.min(self.alpha) < 0:
        raise ValueError(
            "Smoothing parameter alpha = %.1e. alpha should be > 0."
            % np.min(self.alpha)
        )
    if isinstance(self.alpha, np.ndarray):
        if not self.alpha.shape[0] == self.n_features_in_:
            raise ValueError(
                "alpha should be a scalar or a numpy array with shape [n_features]"
            )
    if np.min(self.alpha) < _ALPHA_MIN:
        warnings.warn(
            "alpha too small will result in numeric errors, setting alpha = %.1e"
            % _ALPHA_MIN
        )
        return np.maximum(self.alpha, _ALPHA_MIN)
    return self.alpha

def partial_fit(self, X, y, classes=None, sample_weight=None):
    """Incremental fit on a batch of samples.

    This method is expected to be called several times consecutively
    on different chunks of a dataset so as to implement out-of-core
    or online learning.
```

```
This is especially useful when the whole dataset is too big to fit in
memory at once.

This method has some performance overhead hence it is better to call
partial_fit on chunks of data that are as large as possible
(as long as fitting in the memory budget) to hide the overhead.

Parameters
----------
X : {array-like, sparse matrix} of shape (n_samples, n_features)
    Training vectors, where `n_samples` is the number of samples and
    `n_features` is the number of features.

y : array-like of shape (n_samples,)
    Target values.

classes : array-like of shape (n_classes,), default=None
    List of all the classes that can possibly appear in the y vector.

    Must be provided at the first call to partial_fit, can be omitted
    in subsequent calls.

sample_weight : array-like of shape (n_samples,), default=None
    Weights applied to individual samples (1. for unweighted).

Returns
-------
self : object
    Returns the instance itself.
"""
first_call = not hasattr(self, "classes_")
X, y = self._check_X_y(X, y, reset=first_call)
_, n_features = X.shape

if _check_partial_fit_first_call(self, classes):
    # This is the first call to partial_fit:
    # initialize various cumulative counters
    n_classes = len(classes)
    self._init_counters(n_classes, n_features)

Y = label_binarize(y, classes=self.classes_)
if Y.shape[1] == 1:
    if len(self.classes_) == 2:
        Y = np.concatenate((1 - Y, Y), axis=1)
```

```
        else:  # degenerate case: just one class
            Y = np.ones_like(Y)

    if X.shape[0] != Y.shape[0]:
        msg = "X.shape[0]=%d and y.shape[0]=%d are incompatible."
        raise ValueError(msg % (X.shape[0], y.shape[0]))

    # label_binarize() returns arrays with dtype=np.int64.
    # We convert it to np.float64 to support sample_weight consistently
    Y = Y.astype(np.float64, copy=False)
    if sample_weight is not None:
        sample_weight = _check_sample_weight(sample_weight, X)
        sample_weight = np.atleast_2d(sample_weight)
        Y *= sample_weight.T

    class_prior = self.class_prior

    # Count raw events from data before updating the class log prior
    # and feature log probas
    self._count(X, Y)

    # XXX: OPTIM: we could introduce a public finalization method to
    # be called by the user explicitly just once after several consecutive
    # calls to partial_fit and prior any call to predict[_[log_]proba]
    # to avoid computing the smooth log probas at each call to partial fit
    alpha = self._check_alpha()
    self._update_feature_log_prob(alpha)
    self._update_class_log_prior(class_prior=class_prior)
    return self

def fit(self, X, y, sample_weight=None):
    """Fit Naive Bayes classifier according to X, y.

    Parameters
    ----------
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        Training vectors, where `n_samples` is the number of samples and
        `n_features` is the number of features.

    y : array-like of shape (n_samples,)
        Target values.

    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).
```

```
        Returns
        -------
        self : object
            Returns the instance itself.
        """
        X, y = self._check_X_y(X, y)
        _, n_features = X.shape

        labelbin = LabelBinarizer()
        Y = labelbin.fit_transform(y)
        self.classes_ = labelbin.classes_
        if Y.shape[1] == 1:
            if len(self.classes_) == 2:
                Y = np.concatenate((1 - Y, Y), axis=1)
            else:  # degenerate case: just one class
                Y = np.ones_like(Y)

        # LabelBinarizer().fit_transform() returns arrays with dtype=np.int64.
        # We convert it to np.float64 to support sample_weight consistently;
        # this means we also don't have to cast X to floating point
        if sample_weight is not None:
            Y = Y.astype(np.float64, copy=False)
            sample_weight = _check_sample_weight(sample_weight, X)
            sample_weight = np.atleast_2d(sample_weight)
            Y *= sample_weight.T

        class_prior = self.class_prior

        # Count raw events from data before updating the class log prior
        # and feature log probas
        n_classes = Y.shape[1]
        self._init_counters(n_classes, n_features)
        self._count(X, Y)
        alpha = self._check_alpha()
        self._update_feature_log_prob(alpha)
        self._update_class_log_prior(class_prior=class_prior)
        return self

    def _init_counters(self, n_classes, n_features):
        self.class_count_ = np.zeros(n_classes, dtype=np.float64)
        self.feature_count_ = np.zeros((n_classes, n_features), dtype=np.float64)

    # mypy error: Decorated property not supported
```

```
    @deprecated(  # type: ignore
        "Attribute `coef_` was deprecated in "
        "version 0.24 and will be removed in 1.1 (renaming of 0.26)."
    )
    @property
    def coef_(self):
        return (
            self.feature_log_prob_[1:]
            if len(self.classes_) == 2
            else self.feature_log_prob_
        )

    # mypy error: Decorated property not supported
    @deprecated(  # type: ignore
        "Attribute `intercept_` was deprecated in "
        "version 0.24 and will be removed in 1.1 (renaming of 0.26)."
    )
    @property
    def intercept_(self):
        return (
            self.class_log_prior_[1:]
            if len(self.classes_) == 2
            else self.class_log_prior_
        )

    def _more_tags(self):
        return {"poor_score": True}

    # TODO: Remove in 1.2
    # mypy error: Decorated property not supported
    @deprecated(  # type: ignore
        "Attribute `n_features_` was deprecated in version 1.0 and will be "
        "removed in 1.2. Use `n_features_in_` instead."
    )
    @property
    def n_features_(self):
        return self.n_features_in_


class MultinomialNB(_BaseDiscreteNB):
    """
    Naive Bayes classifier for multinomial models.

    The multinomial Naive Bayes classifier is suitable for classification with
```

```
discrete features (e.g., word counts for text classification). The
multinomial distribution normally requires integer feature counts. However,
in practice, fractional counts such as tf-idf may also work.

Read more in the :ref:`User Guide <multinomial_naive_bayes>`.

Parameters
----------
alpha : float, default=1.0
    Additive (Laplace/Lidstone) smoothing parameter
    (0 for no smoothing).

fit_prior : bool, default=True
    Whether to learn class prior probabilities or not.
    If false, a uniform prior will be used.

class_prior : array-like of shape (n_classes,), default=None
    Prior probabilities of the classes. If specified the priors are not
    adjusted according to the data.

Attributes
----------
class_count_ : ndarray of shape (n_classes,)
    Number of samples encountered for each class during fitting. This
    value is weighted by the sample weight when provided.

class_log_prior_ : ndarray of shape (n_classes,)
    Smoothed empirical log probability for each class.

classes_ : ndarray of shape (n_classes,)
    Class labels known to the classifier

coef_ : ndarray of shape (n_classes, n_features)
    Mirrors ``feature_log_prob_`` for interpreting `MultinomialNB`
    as a linear model.

    .. deprecated:: 0.24
        ``coef_`` is deprecated in 0.24 and will be removed in 1.1
        (renaming of 0.26).

feature_count_ : ndarray of shape (n_classes, n_features)
    Number of samples encountered for each (class, feature)
    during fitting. This value is weighted by the sample weight when
    provided.
```

```
feature_log_prob_ : ndarray of shape (n_classes, n_features)
    Empirical log probability of features
    given a class, ``P(x_i|y)``.

intercept_ : ndarray of shape (n_classes,)
    Mirrors ``class_log_prior_`` for interpreting `MultinomialNB`
    as a linear model.

    .. deprecated:: 0.24
        ``intercept_`` is deprecated in 0.24 and will be removed in 1.1
        (renaming of 0.26).

n_features_ : int
    Number of features of each sample.

    .. deprecated:: 1.0
        Attribute `n_features_` was deprecated in version 1.0 and will be
        removed in 1.2. Use `n_features_in_` instead.

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

See Also
--------
BernoulliNB : Naive Bayes classifier for multivariate Bernoulli models.
CategoricalNB : Naive Bayes classifier for categorical features.
ComplementNB : Complement Naive Bayes classifier.
GaussianNB : Gaussian Naive Bayes.

Notes
-----
For the rationale behind the names `coef_` and `intercept_`, i.e.
naive Bayes as a linear classifier, see J. Rennie et al. (2003),
Tackling the poor assumptions of naive Bayes text classifiers, ICML.
```

```
    References
    ----------
    C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to
    Information Retrieval. Cambridge University Press, pp. 234-265.
    https://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html

    Examples
    --------
    >>> import numpy as np
    >>> rng = np.random.RandomState(1)
    >>> X = rng.randint(5, size=(6, 100))
    >>> y = np.array([1, 2, 3, 4, 5, 6])
    >>> from sklearn.naive_bayes import MultinomialNB
    >>> clf = MultinomialNB()
    >>> clf.fit(X, y)
    MultinomialNB()
    >>> print(clf.predict(X[2:3]))
    [3]
    """

    def __init__(self, *, alpha=1.0, fit_prior=True, class_prior=None):
        self.alpha = alpha
        self.fit_prior = fit_prior
        self.class_prior = class_prior

    def _more_tags(self):
        return {"requires_positive_X": True}

    def _count(self, X, Y):
        """Count and smooth feature occurrences."""
        check_non_negative(X, "MultinomialNB (input X)")
        self.feature_count_ += safe_sparse_dot(Y.T, X)
        self.class_count_ += Y.sum(axis=0)

    def _update_feature_log_prob(self, alpha):
        """Apply smoothing to raw counts and recompute log probabilities"""
        smoothed_fc = self.feature_count_ + alpha
        smoothed_cc = smoothed_fc.sum(axis=1)

        self.feature_log_prob_ = np.log(smoothed_fc) - np.log(
            smoothed_cc.reshape(-1, 1)
        )

    def _joint_log_likelihood(self, X):
```

```
        """Calculate the posterior log probability of the samples X"""
        return safe_sparse_dot(X, self.feature_log_prob_.T) + self.class_log_prior_


class ComplementNB(_BaseDiscreteNB):
    """The Complement Naive Bayes classifier described in Rennie et al. (2003).

    The Complement Naive Bayes classifier was designed to correct the "severe
    assumptions" made by the standard Multinomial Naive Bayes classifier. It is
    particularly suited for imbalanced data sets.

    Read more in the :ref:`User Guide <complement_naive_bayes>`.

    .. versionadded:: 0.20

    Parameters
    ----------
    alpha : float, default=1.0
        Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

    fit_prior : bool, default=True
        Only used in edge case with a single class in the training set.

    class_prior : array-like of shape (n_classes,), default=None
        Prior probabilities of the classes. Not used.

    norm : bool, default=False
        Whether or not a second normalization of the weights is performed. The
        default behavior mirrors the implementations found in Mahout and Weka,
        which do not follow the full algorithm described in Table 9 of the
        paper.

    Attributes
    ----------
    class_count_ : ndarray of shape (n_classes,)
        Number of samples encountered for each class during fitting. This
        value is weighted by the sample weight when provided.

    class_log_prior_ : ndarray of shape (n_classes,)
        Smoothed empirical log probability for each class. Only used in edge
        case with a single class in the training set.

    classes_ : ndarray of shape (n_classes,)
        Class labels known to the classifier
```

```
coef_ : ndarray of shape (n_classes, n_features)
   Mirrors ``feature_log_prob_`` for interpreting `ComplementNB`
   as a linear model.

   .. deprecated:: 0.24
      ``coef_`` is deprecated in 0.24 and will be removed in 1.1
      (renaming of 0.26).

feature_all_ : ndarray of shape (n_features,)
   Number of samples encountered for each feature during fitting. This
   value is weighted by the sample weight when provided.

feature_count_ : ndarray of shape (n_classes, n_features)
   Number of samples encountered for each (class, feature) during fitting.
   This value is weighted by the sample weight when provided.

feature_log_prob_ : ndarray of shape (n_classes, n_features)
   Empirical weights for class complements.

intercept_ : ndarray of shape (n_classes,)
   Mirrors ``class_log_prior_`` for interpreting `ComplementNB`
   as a linear model.

   .. deprecated:: 0.24
      ``coef_`` is deprecated in 0.24 and will be removed in 1.1
      (renaming of 0.26).

n_features_ : int
   Number of features of each sample.

   .. deprecated:: 1.0
      Attribute `n_features_` was deprecated in version 1.0 and will be
      removed in 1.2. Use `n_features_in_` instead.

n_features_in_ : int
   Number of features seen during :term:`fit`.

   .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
   Names of features seen during :term:`fit`. Defined only when `X`
   has feature names that are all strings.
```

```
    .. versionadded:: 1.0

    See Also
    --------
    BernoulliNB : Naive Bayes classifier for multivariate Bernoulli models.
    CategoricalNB : Naive Bayes classifier for categorical features.
    GaussianNB : Gaussian Naive Bayes.
    MultinomialNB : Naive Bayes classifier for multinomial models.

    References
    ----------
    Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003).
    Tackling the poor assumptions of naive bayes text classifiers. In ICML
    (Vol. 3, pp. 616-623).
    https://people.csail.mit.edu/jrennie/papers/icml03-nb.pdf

    Examples
    --------
    >>> import numpy as np
    >>> rng = np.random.RandomState(1)
    >>> X = rng.randint(5, size=(6, 100))
    >>> y = np.array([1, 2, 3, 4, 5, 6])
    >>> from sklearn.naive_bayes import ComplementNB
    >>> clf = ComplementNB()
    >>> clf.fit(X, y)
    ComplementNB()
    >>> print(clf.predict(X[2:3]))
    [3]
    """

    def __init__(self, *, alpha=1.0, fit_prior=True, class_prior=None, norm=False):
        self.alpha = alpha
        self.fit_prior = fit_prior
        self.class_prior = class_prior
        self.norm = norm

    def _more_tags(self):
        return {"requires_positive_X": True}

    def _count(self, X, Y):
        """Count feature occurrences."""
        check_non_negative(X, "ComplementNB (input X)")
        self.feature_count_ += safe_sparse_dot(Y.T, X)
        self.class_count_ += Y.sum(axis=0)
```

```
        self.feature_all_ = self.feature_count_.sum(axis=0)

    def _update_feature_log_prob(self, alpha):
        """Apply smoothing to raw counts and compute the weights."""
        comp_count = self.feature_all_ + alpha - self.feature_count_
        logged = np.log(comp_count / comp_count.sum(axis=1, keepdims=True))
        # _BaseNB.predict uses argmax, but ComplementNB operates with argmin.
        if self.norm:
            summed = logged.sum(axis=1, keepdims=True)
            feature_log_prob = logged / summed
        else:
            feature_log_prob = -logged
        self.feature_log_prob_ = feature_log_prob

    def _joint_log_likelihood(self, X):
        """Calculate the class scores for the samples in X."""
        jll = safe_sparse_dot(X, self.feature_log_prob_.T)
        if len(self.classes_) == 1:
            jll += self.class_log_prior_
        return jll


class BernoulliNB(_BaseDiscreteNB):
    """Naive Bayes classifier for multivariate Bernoulli models.

    Like MultinomialNB, this classifier is suitable for discrete data. The
    difference is that while MultinomialNB works with occurrence counts,
    BernoulliNB is designed for binary/boolean features.

    Read more in the :ref:`User Guide <bernoulli_naive_bayes>`.

    Parameters
    ----------
    alpha : float, default=1.0
        Additive (Laplace/Lidstone) smoothing parameter
        (0 for no smoothing).

    binarize : float or None, default=0.0
        Threshold for binarizing (mapping to booleans) of sample features.
        If None, input is presumed to already consist of binary vectors.

    fit_prior : bool, default=True
        Whether to learn class prior probabilities or not.
        If false, a uniform prior will be used.
```

```
class_prior : array-like of shape (n_classes,), default=None
    Prior probabilities of the classes. If specified the priors are not
    adjusted according to the data.

Attributes
----------
class_count_ : ndarray of shape (n_classes,)
    Number of samples encountered for each class during fitting. This
    value is weighted by the sample weight when provided.

class_log_prior_ : ndarray of shape (n_classes,)
    Log probability of each class (smoothed).

classes_ : ndarray of shape (n_classes,)
    Class labels known to the classifier

coef_ : ndarray of shape (n_classes, n_features)
    Mirrors ``feature_log_prob_`` for interpreting `BernoulliNB`
    as a linear model.

feature_count_ : ndarray of shape (n_classes, n_features)
    Number of samples encountered for each (class, feature)
    during fitting. This value is weighted by the sample weight when
    provided.

feature_log_prob_ : ndarray of shape (n_classes, n_features)
    Empirical log probability of features given a class, P(x_i|y).

intercept_ : ndarray of shape (n_classes,)
    Mirrors ``class_log_prior_`` for interpreting `BernoulliNB`
    as a linear model.

n_features_ : int
    Number of features of each sample.

    .. deprecated:: 1.0
        Attribute `n_features_` was deprecated in version 1.0 and will be
        removed in 1.2. Use `n_features_in_` instead.

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24
```

```
feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

See Also
--------
CategoricalNB : Naive Bayes classifier for categorical features.
ComplementNB : The Complement Naive Bayes classifier
    described in Rennie et al. (2003).
GaussianNB : Gaussian Naive Bayes (GaussianNB).
MultinomialNB : Naive Bayes classifier for multinomial models.

References
----------
C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to
Information Retrieval. Cambridge University Press, pp. 234-265.
https://nlp.stanford.edu/IR-book/html/htmledition/the-bernoulli-model-1.html

A. McCallum and K. Nigam (1998). A comparison of event models for naive
Bayes text classification. Proc. AAAI/ICML-98 Workshop on Learning for
Text Categorization, pp. 41-48.

V. Metsis, I. Androutsopoulos and G. Paliouras (2006). Spam filtering with
naive Bayes -- Which naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

Examples
--------
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> Y = np.array([1, 2, 3, 4, 4, 5])
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(X, Y)
BernoulliNB()
>>> print(clf.predict(X[2:3]))
[3]
"""

def __init__(self, *, alpha=1.0, binarize=0.0, fit_prior=True, class_prior=None):
    self.alpha = alpha
```

```python
        self.binarize = binarize
        self.fit_prior = fit_prior
        self.class_prior = class_prior

    def _check_X(self, X):
        """Validate X, used only in predict* methods."""
        X = super()._check_X(X)
        if self.binarize is not None:
            X = binarize(X, threshold=self.binarize)
        return X

    def _check_X_y(self, X, y, reset=True):
        X, y = super()._check_X_y(X, y, reset=reset)
        if self.binarize is not None:
            X = binarize(X, threshold=self.binarize)
        return X, y

    def _count(self, X, Y):
        """Count and smooth feature occurrences."""
        self.feature_count_ += safe_sparse_dot(Y.T, X)
        self.class_count_ += Y.sum(axis=0)

    def _update_feature_log_prob(self, alpha):
        """Apply smoothing to raw counts and recompute log probabilities"""
        smoothed_fc = self.feature_count_ + alpha
        smoothed_cc = self.class_count_ + alpha * 2

        self.feature_log_prob_ = np.log(smoothed_fc) - np.log(
            smoothed_cc.reshape(-1, 1)
        )

    def _joint_log_likelihood(self, X):
        """Calculate the posterior log probability of the samples X"""
        n_features = self.feature_log_prob_.shape[1]
        n_features_X = X.shape[1]

        if n_features_X != n_features:
            raise ValueError(
                "Expected input with %d features, got %d instead"
                % (n_features, n_features_X)
            )

        neg_prob = np.log(1 - np.exp(self.feature_log_prob_))
        # Compute  neg_prob · (1 - X).T  as  ∑neg_prob - X · neg_prob
```

```
        jll = safe_sparse_dot(X, (self.feature_log_prob_ - neg_prob).T)
        jll += self.class_log_prior_ + neg_prob.sum(axis=1)

        return jll


class CategoricalNB(_BaseDiscreteNB):
    """Naive Bayes classifier for categorical features.

    The categorical Naive Bayes classifier is suitable for classification with
    discrete features that are categorically distributed. The categories of
    each feature are drawn from a categorical distribution.

    Read more in the :ref:`User Guide <categorical_naive_bayes>`.

    Parameters
    ----------
    alpha : float, default=1.0
        Additive (Laplace/Lidstone) smoothing parameter
        (0 for no smoothing).

    fit_prior : bool, default=True
        Whether to learn class prior probabilities or not.
        If false, a uniform prior will be used.

    class_prior : array-like of shape (n_classes,), default=None
        Prior probabilities of the classes. If specified the priors are not
        adjusted according to the data.

    min_categories : int or array-like of shape (n_features,), default=None
        Minimum number of categories per feature.

        - integer: Sets the minimum number of categories per feature to
          `n_categories` for each features.
        - array-like: shape (n_features,) where `n_categories[i]` holds the
          minimum number of categories for the ith column of the input.
        - None (default): Determines the number of categories automatically
          from the training data.

        .. versionadded:: 0.24

    Attributes
    ----------
    category_count_ : list of arrays of shape (n_features,)
```

Holds arrays of shape (n_classes, n_categories of respective feature)
    for each feature. Each array provides the number of samples
    encountered for each class and category of the specific feature.

class_count_ : ndarray of shape (n_classes,)
    Number of samples encountered for each class during fitting. This
    value is weighted by the sample weight when provided.

class_log_prior_ : ndarray of shape (n_classes,)
    Smoothed empirical log probability for each class.

classes_ : ndarray of shape (n_classes,)
    Class labels known to the classifier

feature_log_prob_ : list of arrays of shape (n_features,)
    Holds arrays of shape (n_classes, n_categories of respective feature)
    for each feature. Each array provides the empirical log probability
    of categories given the respective feature and class, ``P(x_i|y)``.

n_features_ : int
    Number of features of each sample.

    .. deprecated:: 1.0
        Attribute `n_features_` was deprecated in version 1.0 and will be
        removed in 1.2. Use `n_features_in_` instead.

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

n_categories_ : ndarray of shape (n_features,), dtype=np.int64
    Number of categories for each feature. This value is
    inferred from the data or set by the minimum number of categories.

    .. versionadded:: 0.24

See Also

```
    --------
    BernoulliNB : Naive Bayes classifier for multivariate Bernoulli models.
    ComplementNB : Complement Naive Bayes classifier.
    GaussianNB : Gaussian Naive Bayes.
    MultinomialNB : Naive Bayes classifier for multinomial models.

    Examples
    --------
    >>> import numpy as np
    >>> rng = np.random.RandomState(1)
    >>> X = rng.randint(5, size=(6, 100))
    >>> y = np.array([1, 2, 3, 4, 5, 6])
    >>> from sklearn.naive_bayes import CategoricalNB
    >>> clf = CategoricalNB()
    >>> clf.fit(X, y)
    CategoricalNB()
    >>> print(clf.predict(X[2:3]))
    [3]
    """

    def __init__(
        self, *, alpha=1.0, fit_prior=True, class_prior=None, min_categories=None
    ):
        self.alpha = alpha
        self.fit_prior = fit_prior
        self.class_prior = class_prior
        self.min_categories = min_categories

    def fit(self, X, y, sample_weight=None):
        """Fit Naive Bayes classifier according to X, y.

        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            Training vectors, where `n_samples` is the number of samples and
            `n_features` is the number of features. Here, each feature of X is
            assumed to be from a different categorical distribution.
            It is further assumed that all categories of each feature are
            represented by the numbers 0, ..., n - 1, where n refers to the
            total number of categories for the given feature. This can, for
            instance, be achieved with the help of OrdinalEncoder.

        y : array-like of shape (n_samples,)
            Target values.
```

```
        sample_weight : array-like of shape (n_samples,), default=None
            Weights applied to individual samples (1. for unweighted).

        Returns
        -------
        self : object
            Returns the instance itself.
        """
        return super().fit(X, y, sample_weight=sample_weight)

    def partial_fit(self, X, y, classes=None, sample_weight=None):
        """Incremental fit on a batch of samples.

        This method is expected to be called several times consecutively
        on different chunks of a dataset so as to implement out-of-core
        or online learning.

        This is especially useful when the whole dataset is too big to fit in
        memory at once.

        This method has some performance overhead hence it is better to call
        partial_fit on chunks of data that are as large as possible
        (as long as fitting in the memory budget) to hide the overhead.

        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            Training vectors, where `n_samples` is the number of samples and
            `n_features` is the number of features. Here, each feature of X is
            assumed to be from a different categorical distribution.
            It is further assumed that all categories of each feature are
            represented by the numbers 0, ..., n - 1, where n refers to the
            total number of categories for the given feature. This can, for
            instance, be achieved with the help of OrdinalEncoder.

        y : array-like of shape (n_samples,)
            Target values.

        classes : array-like of shape (n_classes,), default=None
            List of all the classes that can possibly appear in the y vector.

            Must be provided at the first call to partial_fit, can be omitted
            in subsequent calls.
```

```
    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).

    Returns
    -------
    self : object
        Returns the instance itself.
    """
    return super().partial_fit(X, y, classes, sample_weight=sample_weight)

def _more_tags(self):
    return {"requires_positive_X": True}

def _check_X(self, X):
    """Validate X, used only in predict* methods."""
    X = self._validate_data(
        X, dtype="int", accept_sparse=False, force_all_finite=True, reset=False
    )
    check_non_negative(X, "CategoricalNB (input X)")
    return X

def _check_X_y(self, X, y, reset=True):
    X, y = self._validate_data(
        X, y, dtype="int", accept_sparse=False, force_all_finite=True, reset=reset
    )
    check_non_negative(X, "CategoricalNB (input X)")
    return X, y

def _init_counters(self, n_classes, n_features):
    self.class_count_ = np.zeros(n_classes, dtype=np.float64)
    self.category_count_ = [np.zeros((n_classes, 0)) for _ in range(n_features)]

@staticmethod
def _validate_n_categories(X, min_categories):
    # rely on max for n_categories categories are encoded between 0...n-1
    n_categories_X = X.max(axis=0) + 1
    min_categories_ = np.array(min_categories)
    if min_categories is not None:
        if not np.issubdtype(min_categories_.dtype, np.signedinteger):
            raise ValueError(
                "'min_categories' should have integral type. Got "
                f"{min_categories_.dtype} instead."
            )
```

```
            n_categories_ = np.maximum(n_categories_X, min_categories_, dtype=np.int64)
            if n_categories_.shape != n_categories_X.shape:
                raise ValueError(
                    f"'min_categories' should have shape ({X.shape[1]},"
                    ") when an array-like is provided. Got"
                    f" {min_categories_.shape} instead."
                )
            return n_categories_
        else:
            return n_categories_X

    def _count(self, X, Y):
        def _update_cat_count_dims(cat_count, highest_feature):
            diff = highest_feature + 1 - cat_count.shape[1]
            if diff > 0:
                # we append a column full of zeros for each new category
                return np.pad(cat_count, [(0, 0), (0, diff)], "constant")
            return cat_count

        def _update_cat_count(X_feature, Y, cat_count, n_classes):
            for j in range(n_classes):
                mask = Y[:, j].astype(bool)
                if Y.dtype.type == np.int64:
                    weights = None
                else:
                    weights = Y[mask, j]
                counts = np.bincount(X_feature[mask], weights=weights)
                indices = np.nonzero(counts)[0]
                cat_count[j, indices] += counts[indices]

        self.class_count_ += Y.sum(axis=0)
        self.n_categories_ = self._validate_n_categories(X, self.min_categories)
        for i in range(self.n_features_in_):
            X_feature = X[:, i]
            self.category_count_[i] = _update_cat_count_dims(
                self.category_count_[i], self.n_categories_[i] - 1
            )
            _update_cat_count(
                X_feature, Y, self.category_count_[i], self.class_count_.shape[0]
            )

    def _update_feature_log_prob(self, alpha):
        feature_log_prob = []
        for i in range(self.n_features_in_):
```

```
            smoothed_cat_count = self.category_count_[i] + alpha
            smoothed_class_count = smoothed_cat_count.sum(axis=1)
            feature_log_prob.append(
                np.log(smoothed_cat_count) - np.log(smoothed_class_count.reshape(-1, 1))
            )
        self.feature_log_prob_ = feature_log_prob

    def _joint_log_likelihood(self, X):
        self._check_n_features(X, reset=False)
        jll = np.zeros((X.shape[0], self.class_count_.shape[0]))
        for i in range(self.n_features_in_):
            indices = X[:, i]
            jll += self.feature_log_prob_[i][:, indices].T
        total_ll = jll + self.class_log_prior_
        return total_ll
```