--- Python interface of LIBSVM ---

Table of Contents

=====

Introduction

====

Python (http://www.python.org/) is a programming language suitable for rapid development. This tool provides a simple Python interface to LIBSVM, a library for support vector machines (http://www.csie.ntu.edu.tw/~cjlin/libsvm). The interface is very easy to use as the usage is the same as that of LIBSVM. The interface is developed with the built-in Python library "ctypes."

Installation

====

On Unix systems, type

> make

The interface needs only LIBSVM shared library, which is generated by
the above command. We assume that the shared library is on the LIBSVM
main directory or in the system path.

For windows, the shared library libsvm.dll for 64-bit python is ready
in the directory `..\windows'. To regenerate the shared library,
please follow the instruction of building windows binaries in LIBSVM
README.

Quick Start

===

"Quick Start with Scipy" is in the next section.

There are two levels of usage. The high-level one uses utility functions
in svmutil.py and the usage is the same as the LIBSVM MATLAB interface.

```
>>> from svmutil import *
# Read data in LIBSVM format
>>> y, x = svm_read_problem('../heart_scale')
>>> m = svm_train(y[:200], x[:200], '-c 4')
>>> p_label, p_acc, p_val = svm_predict(y[200:], x[200:], m)

# Construct problem in python format
# Dense data
>>> y, x = [1,-1], [[1,0,1], [-1,0,-1]]
# Sparse data
>>> y, x = [1,-1], [{1:1, 3:1}, {1:-1,3:-1}]
>>> prob = svm_problem(y, x)
>>> param = svm_parameter('-t 0 -c 4 -b 1')
>>> m = svm_train(prob, param)
```

```
# Precomputed kernel data (-t 4)
# Dense data
>>> y, x = [1,-1], [[1, 2, -2], [2, -2, 2]]
# Sparse data
>>> y, x = [1,-1], [{0:1, 1:2, 2:-2}, {0:2, 1:-2, 2:2}]
# isKernel=True must be set for precomputed kernel
>>> prob = svm_problem(y, x, isKernel=True)
>>> param = svm_parameter('-t 4 -c 4 -b 1')
>>> m = svm_train(prob, param)
# For the format of precomputed kernel, please read LIBSVM README.


# Other utility functions
>>> svm_save_model('heart_scale.model', m)
>>> m = svm_load_model('heart_scale.model')
>>> p_label, p_acc, p_val = svm_predict(y, x, m, '-b 1')
>>> ACC, MSE, SCC = evaluations(y, p_label)

# Getting online help
>>> help(svm_train)

The low-level use directly calls C interfaces imported by svm.py. Note that
all arguments and return values are in ctypes format. You need to handle them
carefully.

>>> from svm import *
>>> prob = svm_problem([1,-1], [{1:1, 3:1}, {1:-1,3:-1}])
>>> param = svm_parameter('-c 4')
>>> m = libsvm.svm_train(prob, param) # m is a ctype pointer to an svm_model
# Convert a Python-format instance to svm_nodearray, a ctypes structure
>>> x0, max_idx = gen_svm_nodearray({1:1, 3:1})
```

```
>>> label = libsvm.svm_predict(m, x0)
```

Quick Start with Scipy

======

Make sure you have Scipy installed to proceed in this section.

If numba (http://numba.pydata.org) is installed, some operations will be much faster.

There are two levels of usage. The high-level one uses utility functions

in svmutil.py and the usage is the same as the LIBSVM MATLAB interface.

```
>>> import scipy
>>> from svmutil import *
# Read data in LIBSVM format
>>> y, x = svm_read_problem('../heart_scale', return_scipy = True) # y: ndarray, x: csr_matrix
>>> m = svm_train(y[:200], x[:200, :], '-c 4')
>>> p_label, p_acc, p_val = svm_predict(y[200:], x[200:, :], m)

# Construct problem in Scipy format
# Dense data: numpy ndarray
>>> y, x = scipy.asarray([1,-1]), scipy.asarray([[1,0,1], [-1,0,-1]])
# Sparse data: scipy csr_matrix((data, (row_ind, col_ind))
>>> y, x = scipy.asarray([1,-1]), scipy.sparse.csr_matrix(([1, 1, -1, -1], ([0, 0, 1, 1], [0, 2, 0, 2])))
>>> prob = svm_problem(y, x)
>>> param = svm_parameter('-t 0 -c 4 -b 1')
>>> m = svm_train(prob, param)

# Precomputed kernel data (-t 4)
# Dense data: numpy ndarray
>>> y, x = scipy.asarray([1,-1]), scipy.asarray([[1,2,-2], [2,-2,2]])
# Sparse data: scipy csr_matrix((data, (row_ind, col_ind))
>>> y, x = scipy.asarray([1,-1]), scipy.sparse.csr_matrix(([1, 2, -2, 2, -2, 2], ([0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2])))
```

```
# isKernel=True must be set for precomputed kernel
>>> prob = svm_problem(y, x, isKernel=True)
>>> param = svm_parameter('-t 4 -c 4 -b 1')
>>> m = svm_train(prob, param)
# For the format of precomputed kernel, please read LIBSVM README.

# Apply data scaling in Scipy format
>>> y, x = svm_read_problem('../heart_scale', return_scipy=True)
>>> scale_param = csr_find_scale_param(x, lower=0)
>>> scaled_x = csr_scale(x, scale_param)

# Other utility functions
>>> svm_save_model('heart_scale.model', m)
>>> m = svm_load_model('heart_scale.model')
>>> p_label, p_acc, p_val = svm_predict(y, x, m, '-b 1')
>>> ACC, MSE, SCC = evaluations(y, p_label)

# Getting online help
>>> help(svm_train)

The low-level use directly calls C interfaces imported by svm.py. Note that
all arguments and return values are in ctypes format. You need to handle them
carefully.

>>> from svm import *
>>> prob = svm_problem(scipy.asarray([1,-1]), scipy.sparse.csr_matrix(([1, 1, -1, -1], ([0, 0, 1, 1], [0, 2, 0, 2]))))
>>> param = svm_parameter('-c 4')
>>> m = libsvm.svm_train(prob, param) # m is a ctype pointer to an svm_model
# Convert a tuple of ndarray (index, data) to feature_nodearray, a ctypes structure
# Note that index starts from 0, though the following example will be changed to 1:1, 3:1 internally
```

```
>>> x0, max_idx = gen_svm_nodearray((scipy.asarray([0,2]), scipy.asarray([1,1])))
>>> label = libsvm.svm_predict(m, x0)
```

Design Description

======

There are two files svm.py and svmutil.py, which respectively correspond to
low-level and high-level use of the interface.

In svm.py, we adopt the Python built-in library "ctypes," so that
Python can directly access C structures and interface functions defined
in svm.h.

While advanced users can use structures/functions in svm.py, to
avoid handling ctypes structures, in svmutil.py we provide some easy-to-use
functions. The usage is similar to LIBSVM MATLAB interface.

Data Structures

===

Four data structures derived from svm.h are svm_node, svm_problem, svm_parameter,
and svm_model. They all contain fields with the same names in svm.h. Access
these fields carefully because you directly use a C structure instead of a
Python object. For svm_model, accessing the field directly is not recommanded.
Programmers should use the interface functions or methods of svm_model class
in Python to get the values. The following description introduces additional
fields and methods.

Before using the data structures, execute the following command to load the
LIBSVM shared library:

```
>>> from svm import *
```

- class svm_node:

Construct an svm_node.

>>> node = svm_node(idx, val)

idx: an integer indicates the feature index.

val: a float indicates the feature value.

Show the index and the value of a node.

>>> print(node)

- Function: gen_svm_nodearray(xi [,feature_max=None [,isKernel=False]])

Generate a feature vector from a Python list/tuple/dictionary, numpy ndarray or tuple of (index, data):

>>> xi_ctype, max_idx = gen_svm_nodearray({1:1, 3:1, 5:-2})

xi_ctype: the returned svm_nodearray (a ctypes structure)

max_idx: the maximal feature index of xi

feature_max: if feature_max is assigned, features with indices larger than
feature_max are removed.

isKernel: if isKernel == True, the list index starts from 0 for precomputed
kernel. Otherwise, the list index starts from 1. The default
value is False.

- class svm_problem:

Construct an svm_problem instance

>>> prob = svm_problem(y, x)

y: a Python list/tuple/ndarray of l labels (type must be int/double).

x: 1. a list/tuple of l training instances. Feature vector of
each training instance is a list/tuple or dictionary.
2. an l * n numpy ndarray or scipy spmatrix (n: number of features).
Note that if your x contains sparse data (i.e., dictionary), the internal
ctypes data format is still sparse.
For pre-computed kernel, the isKernel flag should be set to True:
>>> prob = svm_problem(y, x, isKernel=True)
Please read LIBSVM README for more details of pre-computed kernel.
- class svm_parameter:
Construct an svm_parameter instance
>>> param = svm_parameter('training_options')
If 'training_options' is empty, LIBSVM default values are applied.
Set param to LIBSVM default values.
>>> param.set_to_default_values()
Parse a string of options.
>>> param.parse_options('training_options')
Show values of parameters.
>>> print(param)
- class svm_model:
There are two ways to obtain an instance of svm_model:
>>> model = svm_train(y, x)

```
>>> model = svm_load_model('model_file_name')
```

Note that the returned structure of interface functions
libsvm.svm_train and libsvm.svm_load_model is a ctypes pointer of
svm_model, which is different from the svm_model object returned
by svm_train and svm_load_model in svmutil.py. We provide a
function toPyModel for the conversion:

```
>>> model_ptr = libsvm.svm_train(prob, param)
>>> model = toPyModel(model_ptr)
```

If you obtain a model in a way other than the above approaches,
handle it carefully to avoid memory leak or segmentation fault.

Some interface functions to access LIBSVM models are wrapped as
members of the class svm_model:

```
>>> svm_type = model.get_svm_type()
>>> nr_class = model.get_nr_class()
>>> svr_probability = model.get_svr_probability()
>>> class_labels = model.get_labels()
>>> sv_indices = model.get_sv_indices()
>>> nr_sv = model.get_nr_sv()
>>> is_prob_model = model.is_probability_model()
>>> support_vector_coefficients = model.get_sv_coef()
>>> support_vectors = model.get_SV()
```

Utility Functions

=====

To use utility functions, type

```
>>> from svmutil import *
```

The above command loads

svm_train() : train an SVM model

svm_predict() : predict testing data

svm_read_problem() : read the data from a LIBSVM-format file.

svm_load_model() : load a LIBSVM model.

svm_save_model() : save model to a file.

evaluations() : evaluate prediction results.

csr_find_scale_param() : find scaling parameter for data in csr format.

csr_scale() : apply data scaling to data in csr format.

- Function: svm_train

There are three ways to call svm_train()

>>> model = svm_train(y, x [, 'training_options'])

>>> model = svm_train(prob [, 'training_options'])

>>> model = svm_train(prob, param)

y: a list/tuple/ndarray of l training labels (type must be int/double).

x: 1. a list/tuple of l training instances. Feature vector of

each training instance is a list/tuple or dictionary.

2. an l * n numpy ndarray or scipy spmatrix (n: number of features).

training_options: a string in the same form as that for LIBSVM command

mode.

prob: an svm_problem instance generated by calling

svm_problem(y, x).

For pre-computed kernel, you should use

svm_problem(y, x, isKernel=True)

param: an svm_parameter instance generated by calling
svm_parameter('training_options')

model: the returned svm_model instance. See svm.h for details of this
structure. If '-v' is specified, cross validation is
conducted and the returned model is just a scalar: cross-validation
accuracy for classification and mean-squared error for regression.

To train the same data many times with different
parameters, the second and the third ways should be faster..

Examples:

```
>>> y, x = svm_read_problem('../heart_scale')
>>> prob = svm_problem(y, x)
>>> param = svm_parameter('-s 3 -c 5 -h 0')
>>> m = svm_train(y, x, '-c 5')
>>> m = svm_train(prob, '-t 2 -c 5')
>>> m = svm_train(prob, param)
>>> CV_ACC = svm_train(y, x, '-v 3')
```

- Function: svm_predict

To predict testing data with a model, use

```
>>> p_labs, p_acc, p_vals = svm_predict(y, x, model [,'predicting_options'])
```

y: a list/tuple/ndarray of l true labels (type must be int/double).
It is used for calculating the accuracy. Use [] if true labels are
unavailable.

x: 1. a list/tuple of l training instances. Feature vector of
each training instance is a list/tuple or dictionary.

```
2. an l * n numpy ndarray or scipy spmatrix (n: number of features).

predicting_options: a string of predicting options in the same format as
that of LIBSVM.

model: an svm_model instance.

p_labels: a list of predicted labels

p_acc: a tuple including accuracy (for classification), mean
squared error, and squared correlation coefficient (for
regression).

p_vals: a list of decision values or probability estimates (if '-b 1'
is specified). If k is the number of classes in training data,
for decision values, each element includes results of predicting
k(k-1)/2 binary-class SVMs. For classification, k = 1 is a
special case. Decision value [+1] is returned for each testing
instance, instead of an empty list.
For probabilities, each element contains k values indicating
the probability that the testing instance is in each class.
Note that the order of classes is the same as the 'model.label'
field in the model structure.

Example:

>>> m = svm_train(y, x, '-c 5')
>>> p_labels, p_acc, p_vals = svm_predict(y, x, m)

- Functions: svm_read_problem/svm_load_model/svm_save_model

See the usage by examples:

>>> y, x = svm_read_problem('data.txt')
```

```
>>> m = svm_load_model('model_file')
>>> svm_save_model('model_file', m)

- Function: evaluations

Calculate some evaluations using the true values (ty) and the predicted
values (pv):

>>> (ACC, MSE, SCC) = evaluations(ty, pv, useScipy)

ty: a list/tuple/ndarray of true values.

pv: a list/tuple/ndarray of predicted values.

useScipy: convert ty, pv to ndarray, and use scipy functions to do the evaluation

ACC: accuracy.

MSE: mean squared error.

SCC: squared correlation coefficient.

- Function: csr_find_scale_parameter/csr_scale

Scale data in csr format.

>>> param = csr_find_scale_param(x [, lower=l, upper=u])
>>> x = csr_scale(x, param)

x: a csr_matrix of data.

l: x scaling lower limit; default -1.

u: x scaling upper limit; default 1.

The scaling process is: x * diag(coef) + ones(l, 1) * offset'

param: a dictionary of scaling parameters, where param['coef'] = coef and param['offset'] = offset.
```

coef: a scipy array of scaling coefficients.

offset: a scipy array of scaling offsets.

Additional Information

======

This interface was written by Hsiang-Fu Yu from Department of Computer

Science, National Taiwan University. If you find this tool useful, please

cite LIBSVM as follows

Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support

vector machines. ACM Transactions on Intelligent Systems and

Technology, 2:27:1--27:27, 2011. Software available at

http://www.csie.ntu.edu.tw/~cjlin/libsvm

For any question, please contact Chih-Jen Lin cjlin@csie.ntu.edu.tw,

or check the FAQ page:

http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html