LIBLINEAR is a simple package for solving large-scale regularized linear classification. It currently supports L2-regularized logistic regression/L2-loss support vector classification/L1-loss support vector
classification, and L1-regularized L2-loss support vector classification/
logistic regression. This document explains the usage of LIBLINEAR.

To get started, please read the ``Quick Start'' section first. For developers, please check the ``Library Usage'' section to learn how to integrate LIBLINEAR in your software.

Table of Contents
=================

When to use LIBLINEAR but not LIBSVM
====================================

There are some large data for which with/without nonlinear mappings gives similar performances. Without using kernels, one can efficiently train a much larger set via a linear classifier. These data usually have a large number of features. Document classification is an example.

Warning: While generally liblinear is very fast, its default solver may be slow under certain situations (e.g., data not scaled or C is large). See Appendix B of our SVM guide about how to handle such cases.
http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

Warning: If you are a beginner and your data sets are not large, you should consider LIBSVM first.

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

Quick Start
===========

See the section ``Installation'' for installing LIBLINEAR.

After installation, there are programs `train' and `predict' for training and testing, respectively.

About the data format, please check the README file of LIBSVM. Note that feature index must start from 1 (but not 0).

A sample classification data included in this package is `heart_scale'.

Type `train heart_scale', and the program will read the training data and output the model file `heart_scale.model'. If you have a test

set called heart_scale.t, then type `predict heart_scale.t
heart_scale.model output' to see the prediction accuracy. The
`output'
file contains the predicted class labels.

For more information about `train' and `predict', see the sections
`train' Usage and `predict' Usage.

To obtain good performances, sometimes one needs to scale the
data. Please check the program `svm-scale' of LIBSVM. For large and
sparse data, use `-l 0' to keep the sparsity.

Installation
============

On Unix systems, type `make' to build the `train' and `predict'
programs. Run them without arguments to show the usages.

On other systems, consult `Makefile' to build them (e.g., see
'Building Windows binaries' in this file) or use the pre-built
binaries (Windows binaries are in the directory `windows').

This software uses some level-1 BLAS subroutines. The needed
functions are
included in this package. If a BLAS library is available on your
machine, you may use it by modifying the Makefile: Unmark the
following line

        #LIBS ?= -lblas

and mark

        LIBS ?= blas/blas.a

`train' Usage
=============

Usage: train [options] training_set_file [model_file]
options:
-s type : set type of solver (default 1)
      0 -- L2-regularized logistic regression (primal)
      1 -- L2-regularized L2-loss support vector classification
(dual)
      2 -- L2-regularized L2-loss support vector classification
(primal)
      3 -- L2-regularized L1-loss support vector classification
(dual)
      4 -- multi-class support vector classification by Crammer and
Singer
      5 -- L1-regularized L2-loss support vector classification
      6 -- L1-regularized logistic regression
      7 -- L2-regularized logistic regression (dual)
-c cost : set the parameter C (default 1)
-e epsilon : set tolerance of termination criterion
      -s 0 and 2
            |f'(w)|_2 <= eps*min(pos,neg)/l*|f'(w0)|_2,
            where f is the primal function and pos/neg are # of
            positive/negative data (default 0.01)
      -s 1, 3, 4 and 7
            Dual maximal violation <= eps; similar to libsvm (default
0.1)
      -s 5 and 6
            |f'(w)|_inf <= eps*min(pos,neg)/l*|f'(w0)|_inf,
            where f is the primal function (default 0.01)
-B bias : if bias >= 0, instance x becomes [x; bias]; if < 0, no bias
term added (default -1)

-wi weight: weights adjust the parameter C of different classes (see README for details)
-v n: n-fold cross validation mode
-q : quiet mode (no outputs)

Option -v randomly splits the data into n parts and calculates cross validation accuracy on them.

Formulations:

For L2-regularized logistic regression (-s 0), we solve

min_w w^Tw/2 + C \sum log(1 + exp(-y_i w^Tx_i))

For L2-regularized L2-loss SVC dual (-s 1), we solve

min_alpha 0.5(alpha^T (Q + I/2/C) alpha) - e^T alpha
    s.t.   0 <= alpha_i,

For L2-regularized L2-loss SVC (-s 2), we solve

min_w w^Tw/2 + C \sum max(0, 1- y_i w^Tx_i)^2

For L2-regularized L1-loss SVC dual (-s 3), we solve

min_alpha 0.5(alpha^T Q alpha) - e^T alpha
    s.t.   0 <= alpha_i <= C,

For L1-regularized L2-loss SVC (-s 5), we solve

min_w \sum |w_j| + C \sum max(0, 1- y_i w^Tx_i)^2

For L1-regularized logistic regression (-s 6), we solve

min_w \sum |w_j| + C \sum log(1 + exp(-y_i w^Tx_i))

where

Q is a matrix with Q_ij = y_i y_j x_i^T x_j.

For L2-regularized logistic regression (-s 7), we solve

min_alpha 0.5(alpha^T Q alpha) + \sum alpha_i*log(alpha_i) + \sum (C-alpha_i)*log(C-alpha_i) - a constant
    s.t.   0 <= alpha_i <= C,

If bias >= 0, w becomes [w; w_{n+1}] and x becomes [x; bias].

The primal-dual relationship implies that -s 1 and -s 2 give the same model, and -s 0 and -s 7 give the same.

We implement 1-vs-the rest multi-class strategy. In training i vs. non_i, their C parameters are (weight from -wi)*C and C, respectively. If there are only two classes, we train only one model. Thus weight1*C vs. weight2*C is used. See examples below.

We also implement multi-class SVM by Crammer and Singer (-s 4):

min_{w_m, \xi_i} 0.5 \sum_m ||w_m||^2 + C \sum_i \xi_i
    s.t. w^T_{y_i} x_i - w^T_m x_i >= \e^m_i - \xi_i \forall m,i

where e^m_i = 0 if y_i = m,
      e^m_i = 1 if y_i != m,

Here we solve the dual problem:

```
min_{\alpha}  0.5 \sum_m ||w_m(\alpha)||^2 + \sum_i \sum_m e^m_i
alpha^m_i
    s.t.  \alpha^m_i <= C^m_i \forall m,i , \sum_m \alpha^m_i=0
\forall i

where w_m(\alpha) = \sum_i \alpha^m_i x_i,
and C^m_i = C if m  = y_i,
    C^m_i = 0 if m != y_i.

`predict' Usage
===============

Usage: predict [options] test_file model_file output_file
options:
-b probability_estimates: whether to predict probability estimates, 0
or 1 (default 0)

Examples
========

> train data_file

Train linear SVM with L2-loss function.

> train -s 0 data_file

Train a logistic regression model.

> train -v 5 -e 0.001 data_file

Do five-fold cross-validation using L2-loss svm.
Use a smaller stopping tolerance 0.001 than the default
0.1 if you want more accurate solutions.

> train -c 10 -w1 2 -w2 5 -w3 2 four_class_data_file

Train four classifiers:
positive        negative        Cp      Cn
class 1         class 2,3,4.    20      10
class 2         class 1,3,4.    50      10
class 3         class 1,2,4.    20      10
class 4         class 1,2,3.    10      10

> train -c 10 -w3 1 -w2 5 two_class_data_file

If there are only two classes, we train ONE model.
The C values for the two classes are 10 and 50.

> predict -b 1 test_file data_file.model output_file

Output probability estimates (for logistic regression only).

Library Usage
=============

- Function: model* train(const struct problem *prob,
                const struct parameter *param);

    This function constructs and returns a linear classification
model
    according to the given training data and parameters.

    struct problem describes the problem:

        struct problem
        {
```

```
            int l, n;
            int *y;
            struct feature_node **x;
            double bias;
        };
```

where `l' is the number of training data. If bias >= 0, we assume that one additional feature is added to the end of each data instance. `n' is the number of feature (including the bias feature if bias >= 0). `y' is an array containing the target values. And `x' is an array of pointers, each of which points to a sparse representation (array of feature_node) of one training vector.

For example, if we have the following training data:

LABEL	ATTR1	ATTR2	ATTR3	ATTR4	ATTR5
1	0	0.1	0.2	0	0
2	0	0.1	0.3	-1.2	0
1	0.4	0	0	0	0
2	0	0.1	0	1.4	0.5
3	-0.1	-0.2	0.1	1.1	0.1

and bias = 1, then the components of problem are:

```
l = 5
n = 6

y -> 1 2 1 2 3

x -> [ ] -> (2,0.1) (3,0.2) (6,1) (-1,?)
     [ ] -> (2,0.1) (3,0.3) (4,-1.2) (6,1) (-1,?)
     [ ] -> (1,0.4) (6,1) (-1,?)
     [ ] -> (2,0.1) (4,1.4) (5,0.5) (6,1) (-1,?)
     [ ] -> (1,-0.1) (2,-0.2) (3,0.1) (4,1.1) (5,0.1) (6,1) (-
1,?)
```

struct parameter describes the parameters of a linear classification model:

```
        struct parameter
        {
                int solver_type;

                /* these are for training only */
                double eps;             /* stopping criteria */
                double C;
                int nr_weight;
                int *weight_label;
                double* weight;
        };
```

solver_type can be one of L2R_LR, L2R_L2LOSS_SVC_DUAL, L2R_L2LOSS_SVC, L2R_L1LOSS_SVC_DUAL, MCSVM_CS, L1R_L2LOSS_SVC, L1R_LR, L2R_LR_DUAL.

```
    L2R_LR                L2-regularized logistic regression (primal)
    L2R_L2LOSS_SVC_DUAL   L2-regularized L2-loss support vector
classification (dual)
    L2R_L2LOSS_SVC        L2-regularized L2-loss support vector
classification (primal)
    L2R_L1LOSS_SVC_DUAL   L2-regularized L1-loss support vector
classification (dual)
```

```
    MCSVM_CS              multi-class support vector classification
by Crammer and Singer
    L1R_L2LOSS_SVC        L1-regularized L2-loss support vector
classification
    L1R_LR                L1-regularized logistic regression
    L2R_LR_DUAL           L2-regularized logistic regression (dual)

    C is the cost of constraints violation.
    eps is the stopping criterion.

    nr_weight, weight_label, and weight are used to change the
penalty
    for some classes (If the weight for a class is not changed, it is
    set to 1). This is useful for training classifier using
unbalanced
    input data or with asymmetric misclassification cost.

    nr_weight is the number of elements in the array weight_label and
    weight. Each weight[i] corresponds to weight_label[i], meaning
that
    the penalty of class weight_label[i] is scaled by a factor of
weight[i].

    If you do not want to change penalty for any of the classes,
    just set nr_weight to 0.

    *NOTE* To avoid wrong parameters, check_parameter() should be
    called before train().

    struct model stores the model obtained from the training
procedure:

        struct model
        {
                struct parameter param;
                int nr_class;           /* number of classes */
                int nr_feature;
                double *w;
                int *label;             /* label of each class */
                double bias;
        };

     param describes the parameters used to obtain the model.

     nr_class and nr_feature are the number of classes and features,
respectively.

     The nr_feature*nr_class array w gives feature weights. We use
one
     against the rest for multi-class classification, so each feature
     index corresponds to nr_class weight values. Weights are
     organized in the following way

     +------------------+------------------+------------+
     | nr_class weights | nr_class weights |  ...
     | for 1st feature  | for 2nd feature  |
     +------------------+------------------+------------+

     If bias >= 0, x becomes [x; bias]. The number of features is
     increased by one, so w is a (nr_feature+1)*nr_class array. The
     value of bias is stored in the variable bias.

     The array label stores class labels.

- Function: void cross_validation(const problem *prob, const
parameter *param, int nr_fold, int *target);
```

This function conducts cross validation. Data are separated to
nr_fold folds. Under given parameters, sequentially each fold is
validated using the model from training the remaining. Predicted
labels in the validation process are stored in the array called
target.

The format of prob is same as that for train().

- Function: int predict(const model *model_, const feature_node *x);

This functions classifies a test vector using the given
model. The predicted label is returned.

- Function: int predict_values(const struct model *model_,
const struct feature_node *x, double* dec_values);

This function gives nr_w decision values in the array
dec_values. nr_w is 1 if there are two classes except multi-class
svm by Crammer and Singer (-s 4), and is the number of classes
otherwise.

We implement one-vs-the rest multi-class strategy (-s 0,1,2,3)
and
multi-class svm by Crammer and Singer (-s 4) for multi-class SVM.
The class with the highest decision value is returned.

- Function: int predict_probability(const struct model *model_,
const struct feature_node *x, double* prob_estimates);

This function gives nr_class probability estimates in the array
prob_estimates. nr_class can be obtained from the function
get_nr_class. The class with the highest probability is
returned. Currently, we support only the probability outputs of
logistic regression.

- Function: int get_nr_feature(const model *model_);

The function gives the number of attributes of the model.

- Function: int get_nr_class(const model *model_);

The function gives the number of classes of the model.

- Function: void get_labels(const model *model_, int* label);

This function outputs the name of labels into an array called
label.

- Function: const char *check_parameter(const struct problem *prob,
const struct parameter *param);

This function checks whether the parameters are within the
feasible
range of the problem. This function should be called before
calling
train() and cross_validation(). It returns NULL if the
parameters are feasible, otherwise an error message is returned.

- Function: int save_model(const char *model_file_name,
const struct model *model_);

This function saves a model to a file; returns 0 on success, or -
1
if an error occurs.

- Function: struct model *load_model(const char *model_file_name);

    This function returns a pointer to the model read from the file,
    or a null pointer if the model could not be loaded.

- Function: void free_model_content(struct model *model_ptr);

    This function frees the memory used by the entries in a model
structure.

- Function: void free_and_destroy_model(struct model
**model_ptr_ptr);

    This function frees the memory used by a model and destroys the
model
    structure.

- Function: void destroy_param(struct parameter *param);

    This function frees the memory used by a parameter set.

- Function: void set_print_string_function(void (*print_func)(const
char *));

    Users can specify their output format by a function. Use
        set_print_string_function(NULL);
    for default printing to stdout.

Building Windows Binaries
==========================

Windows binaries are in the directory `windows'. To build them via
Visual C++, use the following steps:

1. Open a dos command box and change to liblinear directory. If
environment variables of VC++ have not been set, type

"C:\Program Files\Microsoft Visual Studio 10.0\VC\bin\vcvars32.bat"

You may have to modify the above command according which version of
VC++ or where it is installed.

2. Type

nmake -f Makefile.win clean all

MATLAB/OCTAVE Interface
=======================

Please check the file README in the directory `matlab'.

PYTHON Interface
================

Please check the file README in the directory `python'.

Additional Information
======================

If you find LIBLINEAR helpful, please cite it as

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin.
LIBLINEAR: A Library for Large Linear Classification, Journal of
Machine Learning Research 9(2008), 1871-1874. Software available at
http://www.csie.ntu.edu.tw/~cjlin/liblinear

For any questions and comments, please send your email to
cjlin@csie.ntu.edu.tw