# xlearn_doc Documentation

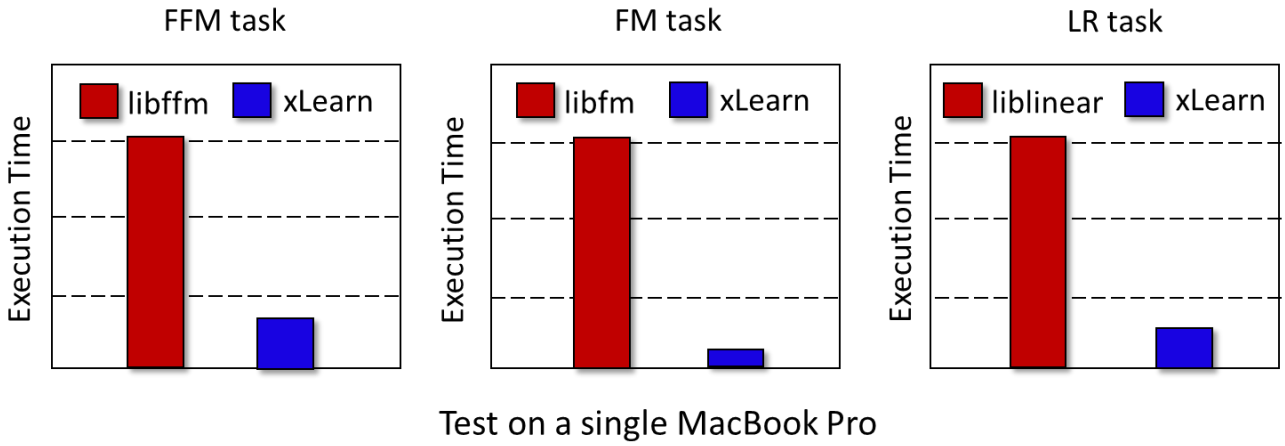
***Release 0.3.1***

**Chao Ma**

**Sep 23, 2018**

# Contents

xLearn is a high-performance, easy-to-use, and scalable machine learning package, which can be used to solve large-scale machine learning problems, especially for the problems on large-scale sparse data, which is very common in scenes like CTR prediction and recommender system. If you are the user of liblinear, libfm, or libffm, now xLearn is your another better choice. This is because xLearn handles all of models and features in these platforms using an uniform way, and it provides better performance, ease-of-use, and scalability.



# CHAPTER 1

# A Quick Example

This is a quick start tutorial showing snippets for you to quickly try out xLearn on a small demo dataset (Criteo CTR prediction) for a binary classification task. In this example, the machine learning algorithm will predict that if current user will click a specified online advertisment.

## 1.1 Installation

The easiest way to install xLearn Python package is to use `pip`. The following command will download the xLearn source code, build and install python package on your locally machine:

```
sudo pip install xlearn
```

The installation process will take a while to complete, please wait with patience. After the installation, users can use the following script in Python shell to check whether the xLearn has been installed successfully:

```
>>> import xlearn as xl
>>> xl.hello()
```

You will see the following message if you installed xLearn successfully:

```
--------------------------------------------------------------------------------
         _
        | |
   __  _| |     ___  __ _ _ __ _ __
   \ \/ / |    / _ \/ _` | '__| '_ \
    >  <| |___|  __/ (_| | |  | | | |
   /_/\_\_____/\___|\__,_|_|  |_| |_|

      xLearn   -- 0.32 Version --
--------------------------------------------------------------------------------
```

If you meet any installation problem, or you want to build the latest code from Github, or you want to use the xLearn command line API instead of the Python API, you can see how to build xLearn from source code in Installation Guide.

## 1.2 Python Demo

Here is a simple Python demo no how to use *FFM* algorithm of xLearn for solving a binary classification problem:

```
import xlearn as xl

# Training task
ffm_model = xl.create_ffm()                # Use field-aware factorization machine␣
↪(ffm)
ffm_model.setTrain("./small_train.txt")    # Set the path of training dataset
ffm_model.setValidate("./small_test.txt")  # Set the path of validation dataset

# Parameters:
#  0. task: binary classification
#  1. learning rate: 0.2
#  2. regular lambda: 0.002
#  3. evaluation metric: accuracy
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'metric':'acc'}

# Start to train
# The trained model will be stored in model.out
ffm_model.fit(param, './model.out')

# Prediction task
ffm_model.setTest("./small_test.txt")  # Set the path of test dataset
ffm_model.setSigmoid()                 # Convert output to 0-1

# Start to predict
# The output result will be stored in output.txt
ffm_model.predict("./model.out", "./output.txt")
```

This example shows how to use *field-aware factorization machines (FFM)* to solve a simple binary classification task. You can check out the demo data (`small_train.txt` and `small_test.txt`) from the path `demo/classification/criteo_ctr`.

CHAPTER 2

# Link to the Other Helpful Resources

## 2.1 Installation Guide

For now, xLearn can support both Linux and Mac OS X. We will support it on Windows platform in the near future. This page gives instructions on how to build and install the xLearn using `pip` and how to build it from source code. No matter what way you choose, make sure that your OS has already installed `GCC` or `Clang` (with the support of `C++ 11`) and `CMake`.

### 2.1.1 Install GCC or Clang

*If you have already installed your C++ compiler before, you can skip this step.*

- On Cygwin, run `setup.exe` and install `gcc` and `binutils`.
- On Debian/Ubuntu Linux, type the command:

```
sudo apt-get install gcc binutils
```

  to install GCC (or Clang) by using:

```
sudo apt-get install clang
```

- On FreeBSD, type the following command to install Clang:

```
sudo pkg_add -r clang
```

- On Mac OS X, install `XCode` gets you Clang.

### 2.1.2 Install CMake

*If you have already installed CMake before, you can skip this step.*

- On Cygwin, run `setup.exe` and install cmake.

- On Debian/Ubuntu Linux, type the command to install cmake:

```
sudo apt-get install cmake
```

- On FreeBSD, type the command:

```
sudo pkg_add -r cmake
```

On Mac OS X, if you have `homebrew`, you can use the command:

```
brew install cmake
```

or if you have `MacPorts`, run:

```
sudo port install cmake
```

You won't want to have both Homebrew and MacPorts installed.

### 2.1.3 Install xLearn from Source Code

Building xLearn from source code consists two steps:

First, you need to build the executable files (`xlearn_train` and `xlearn_predict`), as well as the shared library (`libxlearn_api.so` for Linux or `libxlearn_api.dylib` for Mac OSX) from the C++ code. After that, users need to install the xLearn Python Package.

#### Build from Source Code

Users need to clone the code from github:

```
git clone https://github.com/aksnzhy/xlearn.git

cd xlearn
mkdir build
cd build
cmake ../
make
```

If the building is successful, users can find two executable files (`xlearn_train` and `xlearn_predict`) in the `build` path. Users can test the installation by using the following command:

```
./run_example.sh
```

#### Install Python Package

Then, you can install the Python package through `install-python.sh`:

```
cd python-package
sudo ./install-python.sh
```

You can also test the Python package by using the following command:

```
cd ../
python test_python.py
```

#### One-Button Building

We have already write a script `build.sh` to do all the cumbersome work for users, and users can just use the folloing commands:

```
git clone https://github.com/aksnzhy/xlearn.git

cd xlearn
sudo ./build.sh
```

You may be asked to input your password during installation.

### 2.1.4 Install xLearn from pip

The easiest way to install xLearn Python package is to use `pip`. The following command will download the xLearn source code from pip and install Python package locally. You must make sure that you have already installed C++11 and CMake in your local machine:

```
sudo pip install xlearn
```

The installation process will take a while to complete. After that, you can type the following script in your python shell to check whether the xLearn has been installed successfully:

```
>>> import xlearn as xl
>>> xl.hello()
```

You will see the following message if the installation is successful:

```
----------------------------------------------------------------------------
          _
         | |
   __  _| |     ___  __ _ _ __ _ __
   \ \/ / |    / _ \/ _` | '__| '_ \
    >  <| |___|  __/ (_| | |  | | | |
   /_/\_\_____/\___|\__,_|_|  |_| |_|

      xLearn   -- 0.32 Version --
----------------------------------------------------------------------------
```

### 2.1.5 Install R Package

The R package installation guide is coming soon.

## 2.2 xLearn Command Line Guide

Once you built xLearn from source code successfully, you can get two executable files (`xlearn_train` and `xlearn_predict`) in your `build` directory. Now you can use these two executable files to perform training and prediction tasks.

### 2.2.1 Quick Start

Make sure that you are in the `build` directory of xLearn, and you can find the demo data `small_test.txt` and `small_train.txt` in this directory. Now we can type the following command to train a model:

```
./xlearn_train ./small_train.txt
```

Here, we show a portion of the output in this task. Note that the loss value shown in your local machine could be different with the following result:

```
[ ACTION     ] Start to train ...
[------------] Epoch      Train log_loss     Time cost (sec)
[   10%      ]     1             0.569292                0.00
[   20%      ]     2             0.517142                0.00
[   30%      ]     3             0.490124                0.00
[   40%      ]     4             0.470445                0.00
[   50%      ]     5             0.451919                0.00
[   60%      ]     6             0.437888                0.00
[   70%      ]     7             0.425603                0.00
[   80%      ]     8             0.415573                0.00
[   90%      ]     9             0.405933                0.00
[  100%      ]    10             0.396388                0.00
[ ACTION     ] Start to save model ...
[------------] Model file: ./small_train.txt.model
```

By default, xLearn uses the *logistic regression (LR)* to train the model within 10 epoch.

After that, we can see that a new file called `small_train.txt.model` has been generated in the current directory. This file stores the trained model checkpoint, and we can use this model file to make a prediction in the future:

```
./xlearn_predict ./small_test.txt ./small_train.txt.model
```

After that, we can get a new file called `small_test.txt.out` in the current directory. This is the output of xLearn's prediction. Here we show the first five lines of this output by using the following command:

```
head -n 5 ./small_test.txt.out

-1.9872
-0.0707959
-0.456214
-0.170811
-1.28986
```

These lines of data is the prediction score calculated for each example in the test set. The negative data represents the negative example and positive data represents the positive example. In xLearn, you can convert the score to (0-1) by using `--sigmoid` option, and also you can convert your result to binary result (0 and 1) by using `--sign` option:

```
./xlearn_predict ./small_test.txt ./small_train.txt.model --sigmoid
head -n 5 ./small_test.txt.out

0.120553
0.482308
0.387884
0.457401
0.215877

./xlearn_predict ./small_test.txt ./small_train.txt.model --sign
```

(continues on next page)

(continued from previous page)

```
head -n 5 ./small_test.txt.out

0
0
0
0
0
```

### 2.2.2 Model Output

Users may want to generate different model files (by using different hyper-parameters), and hence users can set the name and path of the model checkpoint file by using `-m` option. By default, the name of the model file is `training_data_name` + `.model`:

```
./xlearn_train ./small_train.txt -m new_model
```

Also, users can save the model in `TXT` format by using `-t` option. For example:

```
./xlearn_train ./small_train.txt -t model.txt
```

After that, we can get a new file called `model.txt`, which stores the trained model in `TXT` format:

```
head -n 5 ./model.txt

-0.688182
0.458082
0
0
0
```

For the linear and bias term, we store each parameter in each line. For FM and FFM, we store each vector of the latent factor in each line. For example:

Linear:

```
bias: 0
i_0: 0
i_1: 0
i_2: 0
i_3: 0
```

FM:

```
bias: 0
i_0: 0
i_1: 0
i_2: 0
i_3: 0
v_0: 5.61937e-06 0.0212581 0.150338 0.222903
v_1: 0.241989 0.0474224 0.128744 0.0995021
v_2: 0.0657265 0.185878 0.0223869 0.140097
v_3: 0.145557 0.202392 0.14798 0.127928
```

FFM:

```
bias: 0
i_0: 0
i_1: 0
i_2: 0
i_3: 0
v_0_0: 5.61937e-06 0.0212581 0.150338 0.222903
v_0_1: 0.241989 0.0474224 0.128744 0.0995021
v_0_2: 0.0657265 0.185878 0.0223869 0.140097
v_0_3: 0.145557 0.202392 0.14798 0.127928
v_1_0: 0.219158 0.248771 0.181553 0.241653
v_1_1: 0.0742756 0.106513 0.224874 0.16325
v_1_2: 0.225384 0.240383 0.0411782 0.214497
v_1_3: 0.226711 0.0735065 0.234061 0.103661
v_2_0: 0.0771142 0.128723 0.0988574 0.197446
v_2_1: 0.172285 0.136068 0.148102 0.0234075
v_2_2: 0.152371 0.108065 0.149887 0.211232
v_2_3: 0.123096 0.193212 0.0179155 0.0479647
v_3_0: 0.055902 0.195092 0.0209918 0.0453358
v_3_1: 0.154174 0.144785 0.184828 0.0785329
v_3_2: 0.109711 0.102996 0.227222 0.248076
v_3_3: 0.144264 0.0409806 0.17463 0.083712
```

Users can also set `-o` option to specify the output file. For example:

```
./xlearn_predict ./small_test.txt ./small_train.txt.model -o output.txt
head -n 5 ./output.txt

-2.01192
-0.0657416
-0.456185
-0.170979
-1.28849
```

By default, the name of the output file is `test_data_name + .out` .

### 2.2.3 Choose Machine Learning Algorithm

For now, xLearn can support three different machine learning algorithms, including linear model, factorization machine (FM), and field-aware factorization machine (FFM).

Users can choose different machine learning algorithms by using `-s` option:

```
-s <type> : Type of machine learning model (default 0)
   for classification task:
       0 -- linear model (GLM)
       1 -- factorization machines (FM)
       2 -- field-aware factorization machines (FFM)
   for regression task:
       3 -- linear model (GLM)
       4 -- factorization machines (FM)
       5 -- field-aware factorization machines (FFM)
```

For LR and FM, the input data format can be `CSV` or `libsvm`. For FFM, the input data should be the `libffm` format:

```
libsvm format:
```

(continues on next page)

(continued from previous page)

```
   label index_1:value_1 index_2:value_2 ... index_n:value_n

CSV format:

   label value_1 value_2 .. value_n

libffm format:

   label field_1:index_1:value_1 field_2:index_2:value_2 ...
```

Note that, if the csv file doesn't contain the label `y`, the user should add a `placeholder` to the dataset by themselves (Also in test data). Otherwise, xLearn will treat the first element as the label `y`.

Users can also give a `libffm` file to LR and FM task. At that time, xLearn will treat this data as `libsvm` format. The following command shows how to use different machine learning algorithms to solve the binary classification problem:

```
./xlearn_train ./small_train.txt -s 0  # Linear model (GLM)
./xlearn_train ./small_train.txt -s 1  # Factorization machine (FM)
./xlearn_train ./small_train.txt -s 2  # Field-awre factorization machine (FFM)
```

### 2.2.4 Set Validation Dataset

A validation dataset is used to tune the hyper-parameters of a machine learning model. In xLearn, users can use `-v` option to set the validation dataset. For example:

```
./xlearn_train ./small_train.txt -v ./small_test.txt
```

A portion of xLearn's output:

```
Epoch      Train log_loss       Test log_loss     Time cost (sec)
    1            0.575049            0.530560             0.00
    2            0.517496            0.537741             0.00
    3            0.488428            0.527205             0.00
    4            0.469010            0.538175             0.00
    5            0.452817            0.537245             0.00
    6            0.438929            0.536588             0.00
    7            0.423491            0.532349             0.00
    8            0.416492            0.541107             0.00
    9            0.404554            0.546218             0.00
```

Here we can see that the training loss continuously goes down. But the validation loss (test loss) goes down first, and then goes up. This is because the model has already overfitted current training dataset. By default, xLearn will calculate the validation loss in each epoch, while users can also set different evaluation metrics by using `-x` option. For classification problems, the metric can be: `acc` (accuracy), `prec` (precision), `f1` (f1 score), `auc` (AUC score). For example:

```
./xlearn_train ./small_train.txt -v ./small_test.txt -x acc
./xlearn_train ./small_train.txt -v ./small_test.txt -x prec
./xlearn_train ./small_train.txt -v ./small_test.txt -x f1
./xlearn_train ./small_train.txt -v ./small_test.txt -x auc
```

For regression problems, the metric can be `mae`, `mape`, and `rmsd` (rmse). For example:

```
cd demo/house_price/
../../xlearn_train ./house_price_train.txt -s 3 -x rmse --cv
../../xlearn_train ./house_price_train.txt -s 3 -x rmsd --cv
```

Note that, in the above example we use cross-validation by using `--cv` option, which will be introduced in the next section.

### 2.2.5 Cross-Validation

Cross-validation, sometimes called rotation estimation, is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent dataset. In xLearn, users can use the `--cv` option to use this technique. For example:

```
./xlearn_train ./small_train.txt --cv
```

On default, xLearn uses 3-folds cross validation, and users can set the number of fold by using `-f` option:

```
./xlearn_train ./small_train.txt -f 5 --cv
```

Here we set the number of folds to `5`. The xLearn will calculate the average validation loss at the end of its output message:

```
 ...
[------------] Average log_loss: 0.549417
[ ACTION     ] Finish Cross-Validation
[ ACTION     ] Clear the xLearn environment ...
[------------] Total time cost: 0.03 (sec)
```

### 2.2.6 Choose Optimization Method

In xLearn, users can choose different optimization methods by using `-p` option. For now, xLearn can support `sgd`, `adagrad`, and `ftrl` method. By default, xLearn uses the `adagrad` method. For example:

```
./xlearn_train ./small_train.txt -p sgd
./xlearn_train ./small_train.txt -p adagrad
./xlearn_train ./small_train.txt -p ftrl
```

Compared to traditional `sgd` method, `adagrad` adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data. In addition, `sgd` is more sensitive to the learning rate compared with `adagrad`.

`FTRL` (Follow-the-Regularized-Leader) is also a famous method that has been widely used in the large-scale sparse problem. To use FTRL, users need to tune more hyper-parameters compared with `sgd` and `adagrad`.

### 2.2.7 Hyper-parameter Tuning

In machine learning, a *hyper-parameter* is a parameter whose value is set before the learning process begins. By contrast, the value of other parameters is derived via training. Hyper-parameter tuning is the problem of choosing a set of optimal hyper-parameters for a learning algorithm.

First, the `learning rate` is one of the most important hyper-parameters used in machine learning. By default, this value is set to `0.2` in xLearn, and we can tune this value by using `-r` option:

```
./xlearn_train ./small_train.txt -v ./small_test.txt -r 0.1
./xlearn_train ./small_train.txt -v ./small_test.txt -r 0.5
./xlearn_train ./small_train.txt -v ./small_test.txt -r 0.01
```

We can also use the `-b` option to perform regularization. By default, xLearn uses `L2` regularization, and the *regular_lambda* has been set to `0.00002`:

```
./xlearn_train ./small_train.txt -v ./small_test.txt -r 0.1 -b 0.001
./xlearn_train ./small_train.txt -v ./small_test.txt -r 0.1 -b 0.002
./xlearn_train ./small_train.txt -v ./small_test.txt -r 0.1 -b 0.01
```

For the `FTRL` method, we also need to tune another four hyper-parameters, including `-alpha`, `-beta`, `-lambda_1`, and `-lambda_2`. For example:

```
./xlearn_train ./small_train.txt -p ftrl -alpha 0.002 -beta 0.8 -lambda_1 0.001 -
↪lambda_2 1.0
```

For FM and FFM, users also need to set the size of *latent factor* by using `-k` option. By default, xLearn uses `4` for this value:

```
./xlearn_train ./small_train.txt -s 1 -v ./small_test.txt -k 2
./xlearn_train ./small_train.txt -s 1 -v ./small_test.txt -k 4
./xlearn_train ./small_train.txt -s 1 -v ./small_test.txt -k 5
./xlearn_train ./small_train.txt -s 1 -v ./small_test.txt -k 8
```

xLearn uses *SSE* instruction to accelerate vector operation, and hence the time cost for `k=2` and `k=4` are the same.

For FM and FFM, users can also set the hyper-parameter `-u` for scalling model initialization. By default, this value is `0.66`:

```
./xlearn_train ./small_train.txt -s 1 -v ./small_test.txt -u 0.80
./xlearn_train ./small_train.txt -s 1 -v ./small_test.txt -u 0.40
./xlearn_train ./small_train.txt -s 1 -v ./small_test.txt -u 0.10
```

### 2.2.8 Set Epoch Number and Early-Stopping

For machine learning tasks, one epoch consists of one full training cycle on the training set. In xLearn, users can set the number of epoch for training by using `-e` option:

```
./xlearn_train ./small_train.txt -e 3
./xlearn_train ./small_train.txt -e 5
./xlearn_train ./small_train.txt -e 10
```

If you set the validation data, xLearn will perform early-stopping by default. For example:

```
./xlearn_train ./small_train.txt -s 2 -v ./small_test.txt -e 10
```

Here, we set epoch number to `10`, but xLearn stopped at epoch `7` because we get the best model at that epoch (you may get different a stopping number on your local machine):

```
 ...
[ ACTION     ] Early-stopping at epoch 7
[ ACTION     ] Start to save model ...
```

Users can set the `window size` for early stopping by using `-sw` option:

```
./xlearn_train ./small_train.txt -e 10 -v ./small_test.txt -sw 3
```

Users can disable early-stopping by using `--dis-es` option:

```
./xlearn_train ./small_train.txt -s 2 -v ./small_test.txt -e 10 --dis-es
```

At this time, xLearn performed completed 10 epoch for training.

By default, xLearn will use the metric value to choose the best epoch if user has set the metric (`-x`). If not, xLearn uses the test_loss to choose the best epoch.

### 2.2.9 Lock-Free Learning

By default, xLearn performs *Hogwild! lock-free* learning, which takes advantages of multiple cores of modern CPU to accelerate training task. But lock-free training is *non-deterministic*. For example, if we run the following command multiple times, we may get different loss value at each epoch:

```
./xlearn_train ./small_train.txt

The 1st time: 0.396352
The 2nd time: 0.396119
The 3nd time: 0.396187
...
```

Users can set the number of thread for xLearn by using `-nthread` option:

```
./xlearn_train ./small_train.txt -nthread 2
```

If you don't set this option, xLearn uses all of the CPU cores by default.

Users can disable lock-free training by using `--dis-lock-free`:

```
./xlearn_train ./small_train.txt --dis-lock-free
```

In thie time, our result are *determinnistic*:

```
The 1st time: 0.396372
The 2nd time: 0.396372
The 3nd time: 0.396372
```

The disadvantage of `--dis-lock-free` is that it is *much slower* than lock-free training.

### 2.2.10 Instance-wise Normalization

For FM and FFM, xLearn uses *instance-wise normalizarion* by default. In some scenes like CTR prediction, this technique is very useful. But sometimes it hurts model performance. Users can disable instance-wise normalization by using `--no-norm` option:

```
./xlearn_train ./small_train.txt -s 1 -v ./small_test.txt --no-norm
```

Note that we usually use `--no-norm` in regression tasks.

### 2.2.11 Quiet Training

When using `--quiet` option, xLearn will not calculate any evaluation information during the training, and it will just train the model quietly:

```
./xlearn_train ./small_train.txt --quiet
```

In this way, xLearn can accelerate its training speed significantly.

xLearn can also support Python API, and we will introduce it in the next section.

## 2.3 xLearn Python Package Guide

xLearn supports easy-to-use Python API for users. Once you install the xLearn Python package successfully, you can try it. Type `python` in your shell and use the following Python code to check your installation:

```
import xlearn as xl
xl.hello()
```

If you install xLearn Python package successfully, you will see:

```
--------------------------------------------------------------------------------
          _
         | |
   __  _| |     ___  __ _ _ __ _ __
   \ \/ / |    / _ \/ _` | '__| '_ \
    >  <| |___|  __/ (_| | |  | | | |
   /_/\_\_____/\___|\__,_|_|  |_| |_|

      xLearn   -- 0.32 Version --
--------------------------------------------------------------------------------
```

### 2.3.1 Quick Start

Here is a simple Python demo to show that how to use xLearn python API. You can checkout the demo data (`small_train.txt` and `small_test.txt`) from the path `demo/classification/criteo_ctr`.

```
import xlearn as xl

# Training task
ffm_model = xl.create_ffm()                # Use field-aware factorization machine
↪(ffm)
ffm_model.setTrain("./small_train.txt")    # Path of training data

# param:
#  0. task: binary classification
#  1. learning rate : 0.2
#  2. regular lambda : 0.002
param = {'task':'binary', 'lr':0.2, 'lambda':0.002}

# Train model
ffm_model.fit(param, "./model.out")
```

A portion of the xLearn's output:

```
  ...
[ ACTION     ] Start to train ...
[------------] Epoch      Train log_loss     Time cost (sec)
[   10%      ]     1            0.595881               0.00
[   20%      ]     2            0.538845               0.00
[   30%      ]     3            0.520051               0.00
[   40%      ]     4            0.504366               0.00
[   50%      ]     5            0.492811               0.00
[   60%      ]     6            0.483286               0.00
[   70%      ]     7            0.472567               0.00
[   80%      ]     8            0.465035               0.00
[   90%      ]     9            0.457047               0.00
[  100%      ]    10            0.448725               0.00
[ ACTION     ] Start to save model ...
```

In this example, xLearn uses *feild-ware factorization machines* (ffm) for solving a binary classification task. If you want train a model for regression task, you can reset the `task` parameter to `reg`:

```
param = {'task':'reg', 'lr':0.2, 'lambda':0.002}
```

We can see that a new file called `model.out` has been generated in the current directory. This file stores the trained model checkpoint, and we can use this model file to make a prediction in the future:

```
ffm_model.setTest("./small_test.txt")
ffm_model.predict("./model.out", "./output.txt")
```

After we run this Python code, we can get a new file called `output.txt` in current directory. This is output prediction. Here we show the first five lines of this output by using the following command:

```
head -n 5 ./output.txt

-1.58631
-0.393496
-0.638334
-0.38465
-1.15343
```

These lines of data are the prediction score calculated for each example in the test set. The negative data represents the negative example and positive data represents the positive example. In xLearn, you can convert the score to (0-1) by using `setSigmoid()` method:

```
ffm_model.setSigmoid()
ffm_model.setTest("./small_test.txt")
ffm_model.predict("./model.out", "./output.txt")
```

and then we can get the result

```
 head -n 5 ./output.txt

0.174698
0.413642
0.353551
0.414588
0.250373
```

We can also convert the score to binary result `(0 and 1)` by using `setSign()` method:

```
ffm_model.setSign()
ffm_model.setTest("./small_test.txt")
ffm_model.predict("./model.out", "./output.txt")
```

and then we can get the result

```
head -n 5 ./output.txt

0
0
0
0
0
```

### 2.3.2 Model Output

Also, users can save the model in `TXT` format by using `setTXTModel()` method. For example:

```
ffm_model.setSign()
ffm_model.setTXTModel("./model.txt")
ffm_model.setTest("./small_test.txt")
ffm_model.predict("./model.out", "./output.txt")
```

After that, we get a new file called `model.txt`, which stores the trained model in `TXT` format:

```
head -n 5 ./model.txt

-1.041
0.31609
0
0
0
```

For the linear and bias term, we store each parameter in each line. For FM and FFM, we store each vector of the latent factor in each line. For example:

Linear:

```
bias: 0
i_0: 0
i_1: 0
i_2: 0
i_3: 0
```

FM:

```
bias: 0
i_0: 0
i_1: 0
i_2: 0
i_3: 0
v_0: 5.61937e-06 0.0212581 0.150338 0.222903
v_1: 0.241989 0.0474224 0.128744 0.0995021
v_2: 0.0657265 0.185878 0.0223869 0.140097
v_3: 0.145557 0.202392 0.14798 0.127928
```

FFM:

```
bias: 0
i_0: 0
i_1: 0
i_2: 0
i_3: 0
v_0_0: 5.61937e-06 0.0212581 0.150338 0.222903
v_0_1: 0.241989 0.0474224 0.128744 0.0995021
v_0_2: 0.0657265 0.185878 0.0223869 0.140097
v_0_3: 0.145557 0.202392 0.14798 0.127928
v_1_0: 0.219158 0.248771 0.181553 0.241653
v_1_1: 0.0742756 0.106513 0.224874 0.16325
v_1_2: 0.225384 0.240383 0.0411782 0.214497
v_1_3: 0.226711 0.0735065 0.234061 0.103661
v_2_0: 0.0771142 0.128723 0.0988574 0.197446
v_2_1: 0.172285 0.136068 0.148102 0.0234075
v_2_2: 0.152371 0.108065 0.149887 0.211232
v_2_3: 0.123096 0.193212 0.0179155 0.0479647
v_3_0: 0.055902 0.195092 0.0209918 0.0453358
v_3_1: 0.154174 0.144785 0.184828 0.0785329
v_3_2: 0.109711 0.102996 0.227222 0.248076
v_3_3: 0.144264 0.0409806 0.17463 0.083712
```

### 2.3.3 Choose Machine Learning Algorithm

For now, xLearn can support three different machine learning algorithms, including linear model, factorization machine (FM), and field-aware factorization machine (FFM):

```
import xlearn as xl

ffm_model = xl.create_ffm()
fm_model = xl.create_fm()
lr_model = xl.create_linear()
```

For LR and FM, the input data format can be `CSV` or `libsvm`. For FFM, the input data should be the `libffm` format:

```
libsvm format:

  label index_1:value_1 index_2:value_2 ... index_n:value_n

CSV format:

  value_1 value_2 .. value_n label

libffm format:

  label field_1:index_1:value_1 field_2:index_2:value_2 ...
```

Note that, if the csv file doesn't contain the label `y`, user should add a `placeholder` to the dataset by themselves (Also in test data). Otherwise, xLearn will treat the first element as the label `y`.

In addtion, users can also give a `libffm` file to LR and FM task. At that time, xLearn will treat this data as `libsvm` format.

### 2.3.4 Set Validation Dataset

A validation dataset is used to tune the hyper-parameters of a machine learning model. In xLearn, users can use `setValdiate()` API to set the validation dataset. For example:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
ffm_model.setValidate("./small_test.txt")
param = {'task':'binary', 'lr':0.2, 'lambda':0.002}

ffm_model.fit(param, "./model.out")
```

A portion of xLearn's output:

```
[ ACTION     ] Start to train ...
[------------] Epoch      Train log_loss       Test log_loss     Time cost (sec)
[   10%      ]     1            0.589475            0.535867                0.00
[   20%      ]     2            0.540977            0.546504                0.00
[   30%      ]     3            0.521881            0.531474                0.00
[   40%      ]     4            0.507194            0.530958                0.00
[   50%      ]     5            0.495460            0.530627                0.00
[   60%      ]     6            0.483910            0.533307                0.00
[   70%      ]     7            0.470661            0.527650                0.00
[   80%      ]     8            0.465455            0.532556                0.00
[   90%      ]     9            0.455787            0.538841                0.00
[ ACTION     ] Early-stopping at epoch 7
```

goes down first, and then goes up. This is because the model has already overfitted current training dataset. By default, xLearn will calculate the validation loss in each epoch, while users can also set different evaluation metrics by using `-x` option. For classification problems, the metric can be : `acc` (accuracy), `prec` (precision), `f1` (f1 score), and `auc` (AUC score). For example:

```
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'metric': 'acc'}
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'metric': 'prec'}
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'metric': 'f1'}
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'metric': 'auc'}
```

For regression problems, the metric can be `mae`, `mape`, and `rmsd` (rmse). For example:

```
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'metric': 'rmse'}
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'metric': 'mae'}
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'metric': 'mape'}
```

### 2.3.5 Cross-Validation

Cross-validation, sometimes called rotation estimation, is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent dataset. In xLearn, users can use the `cv()` API to use this technique. For example:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
```

(continues on next page)

(continued from previous page)

```
param = {'task':'binary', 'lr':0.2, 'lambda':0.002}

ffm_model.cv(param)
```

On default, xLearn uses 3-folds cross validation, and users can set the number of fold by using the `fold` parameter:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'fold':5}

ffm_model.cv(param)
```

Here we set the number of folds to `5`. The xLearn will calculate the average validation loss at the end of its output message:

```
[------------] Average log_loss: 0.549758
[ ACTION     ] Finish Cross-Validation
[ ACTION     ] Clear the xLearn environment ...
[------------] Total time cost: 0.05 (sec)
```

### 2.3.6 Choose Optimization Method

In xLearn, users can choose different optimization methods by using `opt` parameter. For now, xLearn can support `sgd`, `adagrad`, and `ftrl` method. By default, xLearn uses the `adagrad` method. For example:

```
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'opt':'sgd'}
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'opt':'adagrad'}
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'opt':'ftrl'}
```

Compared to traditional `sgd` method, `adagrad` adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data. In addition, `sgd` is more sensitive to the learning rate compared with `adagrad`.

`FTRL` (Follow-the-Regularized-Leader) is also a famous method that has been widely used in the large-scale sparse problem. To use FTRL, users need to tune more hyperparameters compared with `sgd` and `adagrad`.

### 2.3.7 Hyper-parameter Tuning

In machine learning, a hyper-parameter is a parameter whose value is set before the learning process begins. By contrast, the value of other parameters is derived via training. Hyper-parameter tuning is the problem of choosing a set of optimal hyper-parameters for a learning algorithm.

First, the `learning rate` is one of the most important hyperparameters used in machine learning. By default, this value is set to `0.2` in xLearn, and we can tune this value by using `lr` parameter:

```
param = {'task':'binary', 'lr':0.2}
param = {'task':'binary', 'lr':0.5}
param = {'task':'binary', 'lr':0.01}
```

We can also use the `lambda` parameter to perform regularization. By default, xLearn uses `L2` regularization, and the *regular_lambda* has been set to `0.00002`:

```
param = {'task':'binary', 'lr':0.2, 'lambda':0.01}
param = {'task':'binary', 'lr':0.2, 'lambda':0.02}
param = {'task':'binary', 'lr':0.2, 'lambda':0.002}
```

For the `FTRL` method, we also need to tune another four hyper-parameters, including `alpha`, `beta`, `lambda_1`, and `lambda_2`. For example:

```
param = {'alpha':0.002, 'beta':0.8, 'lambda_1':0.001, 'lambda_2': 1.0}
```

For FM and FFM, users also need to set the size of latent factor by using `k` parameter. By default, xLearn uses `4` for this value:

```
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'k':2}
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'k':4}
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'k':5}
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'k':8}
```

xLearn uses *SSE* instruction to accelerate vector operation, and hence the time cost for `k=2` and `k=4` are the same.

For FM and FFM, users can also set the parameter `init` for model initialization. By default, this value is set to `0.66`:

```
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'init':0.80}
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'init':0.40}
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'init':0.10}
```

### 2.3.8 Set Epoch Number and Early-Stopping

For machine learning tasks, one epoch consists of one full training cycle on the training set. In xLearn, users can set the number of epoch for training by using `epoch` parameter:

```
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'epoch':3}
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'epoch':5}
param = {'task':'binary', 'lr':0.2, 'lambda':0.01, 'epoch':10}
```

If you set the validation data, xLearn will perform early-stopping by default. For example:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
ffm_model.setValidate("./small_test.txt")
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'epoch':10}

ffm_model.fit(param, "./model.out")
```

Here, we set epoch number to `10`, but xLearn stopped at epoch `7` because we get the best model at that epoch (you may get different a stopping number on your local machine):

```
Early-stopping at epoch 7
Start to save model ...
```

Users can set `window size` for early-stopping by using `stop_window` parameter:

```
param = {'task':'binary',  'lr':0.2,
         'lambda':0.002, 'epoch':10,
```

(continues on next page)

(continued from previous page)

```
          'stop_window':3}

ffm_model.fit(param, "./model.out")
```

Users can also disable early-stopping by using `disableEarlyStop()` API:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
ffm_model.setValidate("./small_test.txt")
ffm_model.disableEarlyStop();
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'epoch':10}

ffm_model.fit(param, "./model.out")
```

At this time, xLearn performed completed 10 epoch for training.

By default, xLearn will use the metric value to choose the best epoch if user has set the metric (`-x`). If not, xLearn uses the test_loss to choose the best epoch.

### 2.3.9 Lock-Free Learning

By default, xLearn performs Hogwild! lock-free learning, which takes advantages of multiple cores of modern CPU to accelerate training task. But lock-free training is non-deterministic. For example, if we run the following command multiple times, we may get different loss value at each epoch:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
param = {'task':'binary', 'lr':0.2, 'lambda':0.002}

ffm_model.fit(param, "./model.out")

The 1st time: 0.449056
The 2nd time: 0.449302
The 3nd time: 0.449185
```

Users can set the number of thread for xLearn by using `nthread` parameter:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
param = {'task':'binary', 'lr':0.2, 'lambda':0.002, 'nthread':4}

ffm_model.fit(param, "./model.out")
```

Users can also disable lock-free training by using `disableLockFree()` API:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
```

(continues on next page)

(continued from previous page)

```
ffm_model.disableLockFree()
param = {'task':'binary', 'lr':0.2, 'lambda':0.002}

ffm_model.fit(param, "./model.out")
```

In this time, our result are *deterministic*:

```
The 1st time: 0.449172
The 2nd time: 0.449172
The 3nd time: 0.449172
```

The disadvantage of `disableLockFree()` is that it is much slower than lock-free training.

### 2.3.10 Instance-wise Normalization

For FM and FFM, xLearn uses *instance-wise normalizarion* by default. In some scenes like CTR prediction, this technique is very useful. But sometimes it hurts model performance. Users can disable instance-wise normalization by using `disableNorm()` API:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
ffm_model.disableNorm()
param = {'task':'binary', 'lr':0.2, 'lambda':0.002}

ffm_model.fit(param, "./model.out")
```

Note that we usually use `disableNorm()` in regression tasks.

### 2.3.11 Quiet Training

When using `setQuiet()` API, xLearn will not calculate any evaluation information during the training, and it just train the model quietly:

```
import xlearn as xl

ffm_model = xl.create_ffm()
ffm_model.setTrain("./small_train.txt")
ffm_model.setQuiet()
param = {'task':'binary', 'lr':0.2, 'lambda':0.002}

ffm_model.fit(param, "./model.out")
```

In this way, xLearn can accelerate its training speed significantly.

### 2.3.12 Scikit-learn API for xLearn

xLearn can support scikit-learn-like api for users. Here is an example:

```
import numpy as np
import xlearn as xl
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load dataset
iris_data = load_iris()
X = iris_data['data']
y = (iris_data['target'] == 2)

X_train,  \
X_val,    \
y_train,  \
y_val = train_test_split(X, y, test_size=0.3, random_state=0)

# param:
#  0. binary classification
#  1. model scale: 0.1
#  2. epoch number: 10 (auto early-stop)
#  3. learning rate: 0.1
#  4. regular lambda: 1.0
#  5. use sgd optimization method
linear_model = xl.LRModel(task='binary', init=0.1,
                          epoch=10, lr=0.1,
                          reg_lambda=1.0, opt='sgd')

# Start to train
linear_model.fit(X_train, y_train,
                 eval_set=[X_val, y_val],
                 is_lock_free=False)

# Generate predictions
y_pred = linear_model.predict(X_val)
```

In this example, we use linear model to train a binary classifier. We can also create FM and FFM by using `xl.FMModel()` and `xl.FMModel()` . Please see the details of these examples in (Link)

## 2.4 xLearn R Package Guide

xLearn R package guide is coming soon.

## 2.5 xLearn API List

This page gives the xLearn API List for the command line, Python package, and R package.

### 2.5.1 xLearn Command Line Usage

For Training:

```
xlearn_train <train_file_path> [OPTIONS]
```

Options:

```
-s <type> : Type of machine learning model (default 0)
   for classification task:
       0 -- linear model (GLM)
       1 -- factorization machines (FM)
       2 -- field-aware factorization machines (FFM)
   for regression task:
       3 -- linear model (GLM)
       4 -- factorization machines (FM)
       5 -- field-aware factorization machines (FFM)

-x <metric>          :  The metric can be 'acc', 'prec', 'recall', 'f1', 'auc' for␣
↪classification, and
                        'mae', 'mape', 'rmsd (rmse)' for regression. On defaurt,␣
↪xLearn will not print
                        any evaluation metric information (only print loss value).

-p <opt_method>      :  Choose the optimization method, including 'sgd', adagrad',␣
↪and 'ftrl'. On default,
                        xLearn uses the 'adagrad' optimization method.

-v <validate_file>   :  Path of the validation data. This option will be empty by␣
↪default. In this way,
                        xLearn will not perform validation process.

-m <model_file>      :  Path of the model dump file. On default, the model file name␣
↪is 'train_file' + '.model'.
                        If we set this value to 'none', the xLearn will not dump the␣
↪model checkpoint.

-t <txt_model_file>  :  Path of the TEXT model checkpoint file. On default, we do not␣
↪set this option
                        and xLearn will not dump the TEXT model.

-l <log_file>        :  Path of the log file. xLearn uses '/tmp/xlearn_log.*' by␣
↪default.

-k <number_of_K>     :  Number of the latent factor used by FM and FFM tasks. Using 4␣
↪by default.
                        Note that, we will get the same model size when setting k to␣
↪1 and 4.
                        This is because we use SSE instruction and the memory need to␣
↪be aligned.
                        So even you assign k = 1, we still fill some dummy zeros from␣
↪k = 2 to 4.

-r <learning_rate>   :  Learning rate for optimization method. Using 0.2 by default.
                        xLearn can use adaptive gradient descent (AdaGrad) for␣
↪optimization problem,
                        if you choose AdaGrad method, the learning rate will be␣
↪changed adaptively.

-b <lambda_for_regu> :  Lambda for L2 regular. Using 0.00002 by default. We can␣
↪disable the
                        regular term by setting this value to zero.

-alpha               :  Hyper parameters used by ftrl.
```

(continues on next page)

(continued from previous page)

```
-beta                 :  Hyper parameters used by ftrl.

-lambda_1             :  Hyper parameters used by ftrl.

-lambda_2             :  Hyper parameters used by ftrl.

-u <model_scale>      :  Hyper parameter used for initialize model parameters. Using 0.
↪66 by default.

-e <epoch_number>     :  Number of epoch for training process. Using 10 by default.␣
↪Note that xLearn will perform
                         early-stopping by default, so this value is just a upper␣
↪bound.

-f <fold_number>      :  Number of folds for cross-validation (If we set --cv option).␣
↪Using 5 by default.

-nthread <thread_number> :  Number of thread for multiple thread lock-free learning␣
↪(Hogwild!).

-block <block_size>   :  Block size for on-disk training.

-sw <stop_window>     :  Size of stop window for early-stopping. Using 2 by default.

--disk                :  Open on-disk training for large-scale machine learning␣
↪problems.

--cv                  :  Open cross-validation in training tasks. If we use this␣
↪option, xLearn will ignore
                         the validation file (set by -t option).

--dis-lock-free       :  Disable lock-free training. Lock-free training can accelerate␣
↪training but the result
                         is non-deterministic. Our suggestion is that you can open␣
↪this flag if the training data
                         is big and sparse.

--dis-es              :  Disable early-stopping in training. By default, xLearn will␣
↪use early-stopping
                         in training process, except for training in cross-validation.

--no-norm             :  Disable instance-wise normalization. By default, xLearn will␣
↪use instance-wise
                         normalization in both training and prediction processes.

--quiet               :  Don't print any evaluation information during the training␣
↪and just train the
                         model quietly. It can accelerate the training process.
```

For Prediction:

```
xlearn_predict <test_file_path> <model_file_path> [OPTIONS]
```

Options:

```
-o <output_file>      :  Path of the output file. On default, this value will be set␣
↪to 'test_file' + '.out'
```

(continues on next page)

(continued from previous page)

```
-l <log_file_path>   :  Path of the log file. xLearn uses '/tmp/xlearn_log' by␣
↪default.

-nthread <thread number> :  Number of thread for multiple thread lock-free learning␣
↪(Hogwild!).

--sign                   :  Converting output result to 0 and 1.

--sigmoid                :  Converting output result to 0 ~ 1 (problebility).
```

### 2.5.2 xLearn Python API

API List:

```
import xlearn as xl       # Import xlearn package

xl.hello()                # Say hello to user

model = create_linear()   #  Create linear model.

model = create_fm()       #  Create factorization machines.

model = create_ffm()      #  Create field-aware factorizarion machines.

model.show()              #  Show model information.

model.fit(param, "model_path")    #  Train model.

model.cv(param)     # Perform cross-validation.

model.predict("model_path", "output_path")  # Perform prediction.

model.setTrain("data_path")       #  Set training data for xLearn.

model.setValidate("data_path")    #  Set validation data for xLearn.

model.setTest("data_path")        #  Set test data for xLearn.

model.setQuiet()     #  Set xlearn to train model quietly.

model.setOnDisk()    #  Set xlearn to use on-disk training.

model.setSign()      # Convert prediction to 0 and 1.

model.setSigmoid()   # Convert prediction to (0, 1).

model.disableNorm() # Disable instance-wise normalization.

model.disableLockFree()    # Disable lock-free training.

model.disableEarlyStop()   # Disable early-stopping.
```

Parameter List:

```
task     : {'binary',  # Binary classification
            'reg'}     # Regression

metric   : {'acc', 'prec', 'recall', 'f1', 'auc',   # for classification
            'mae', 'mape', 'rmse', 'rmsd'}  # for regression

lr       : float value  # learning rate

lambda   : float value  # regular lambda

k        : int value    # latent factor for fm and ffm

init     : float value  # model initialize

alpha    : float value  # hyper parameter for ftrl

beta     : float value  # hyper parameter for ftrl

lambda_1 : float value  # hyper parameter for ftrl

lambda_2 : float value  # hyper parameter for ftrl

nthread  : int value    # the number of CPU cores

epoch    : int vlaue    # number of epoch

fold     : int value    # number of fold for cross-validation

opt      : {'sgd', 'agagrad', 'ftrl'}  # optimization method

stop_window : Size of stop window for early-stopping.

block_size : int value  # block size for on-disk training
```

### 2.5.3 xLearn R API

xLearn R API page is coming soon.

## 2.6 Large-Scale Machine Learning

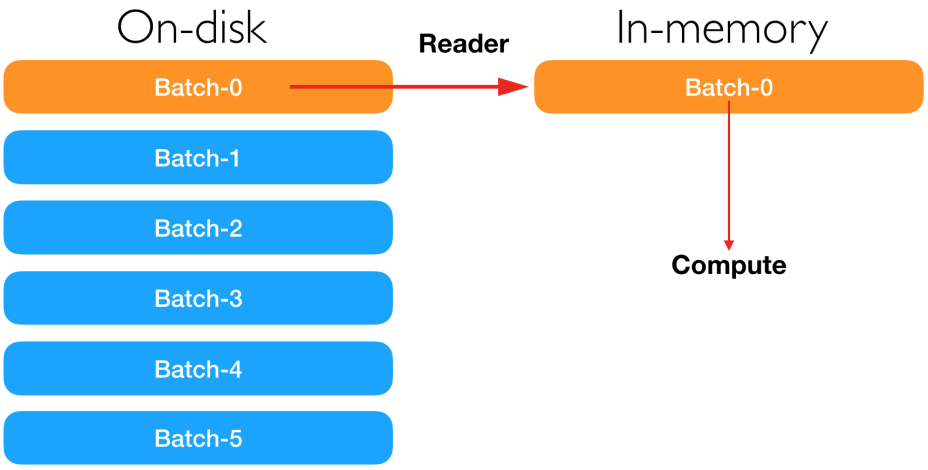
This page shows how to use xLearn to solve large-scale machine learning problems. In recent years, challenges arise with the fast-growing data. For "big-data", we focus on datasets with potentially trillions of training examples, which cannot fit into the memory of a single machine. Motivated by this, we design xLearn to solve large-scale machine learning problems. First, xLearn can handle very large data (TB) on a single PC by using *out-of-core* learning. In addition, xLearn can scale beyond billions of example across many machines to support distributed training by using the *parameter server* framework.

### 2.6.1 Out-of-Core Learning

*Out-of-core* leanring refers to the machine learning algorithms working with data cannot fit into the memory of a single machine, but that can easily fit into some data storage such as local hard disk or web repository. Your available RAM, the core memory on your single machine, may indeed range from a few gigabytes (sometimes 2 GB, more commonly

4 GB, but we assume that you have 2 GB at maximum) up to 256 GB on large server machines. Large servers are like the ones you can get on cloud computing services such as Amazon Elastic Compute Cloud (EC2), whereas your storage capabilities can easily exceed terabytes of capacity using just an external drive (most likely about 1 TB but it can reach up to 4 TB).

Actually, the ability to learn incrementally from a mini-batch of instances is key to *out-of-core* learning as it gurantees that at any given time there will be only a small amount of data in the main memory. Choose a good size for the mini-batch that balances relevancy and memory footprint could involve some tuning.




### Out-of-Core Learning Using xLearn Command Line

Is's very easy to perform *out-of-core* learning in xLearn command line, where users can just use the `--disk` option, and xLearn will help you do all the other things. For example:

```
 ./xlearn_train ./big_data.txt -s 2 --disk

Epoch       Train log_loss      Time cost (sec)
    1             0.483997                4.41
    2             0.466553                4.56
    3             0.458234                4.88
    4             0.451463                4.77
    5             0.445169                4.79
    6             0.438834                4.71
    7             0.432173                4.84
    8             0.424904                4.91
    9             0.416855                5.03
   10             0.407846                4.53
```

In this example, xLearn can finish the training of each epoch in nearly `4.5` second. If you delete the `--disk` option, xLearn can train faster.

```
./xlearn_train ./big_data.txt -s 2

Epoch       Train log_loss      Time cost (sec)
    1             0.484022                1.65
    2             0.466452                1.64
    3             0.458112                1.64
```

(continues on next page)

(continued from previous page)

```
    4             0.451371                1.76
    5             0.445040                1.83
    6             0.438680                1.92
    7             0.432007                1.99
    8             0.424695                1.95
    9             0.416579                1.96
   10             0.407518                2.11
```

In this time, the training of each epoch will only spend nearly `1.8` seconds.

We can set the block size for on-disk training by using `-block` options.

### Out-of-Core Learning Using xLearn Python API

In Python, users can use `setOnDisk` API to perform *out-of-core* learning. For example:

```
import xlearn as xl

# Training task
ffm_model = xl.create_ffm() # Use field-aware factorization machine

# On-disk training
ffm_model.setOnDisk()

ffm_model.setTrain("./small_train.txt")  # Training data
ffm_model.setValidate("./small_test.txt")  # Validation data

# param:
#  0. binary classification
#  1. learning rate: 0.2
#  2. regular lambda: 0.002
#  3. evaluation metric: accuracy
param = {'task':'binary', 'lr':0.2,
         'lambda':0.002, 'metric':'acc'}

# Start to train
# The trained model will be stored in model.out
ffm_model.fit(param, './model.out')

# Prediction task
ffm_model.setTest("./small_test.txt")  # Test data
ffm_model.setSigmoid()  # Convert output to 0-1

# Start to predict
# The output result will be stored in output.txt
ffm_model.predict("./model.out", "./output.txt")
```

We can set the block size for on-disk training by using `block_size` parameter.

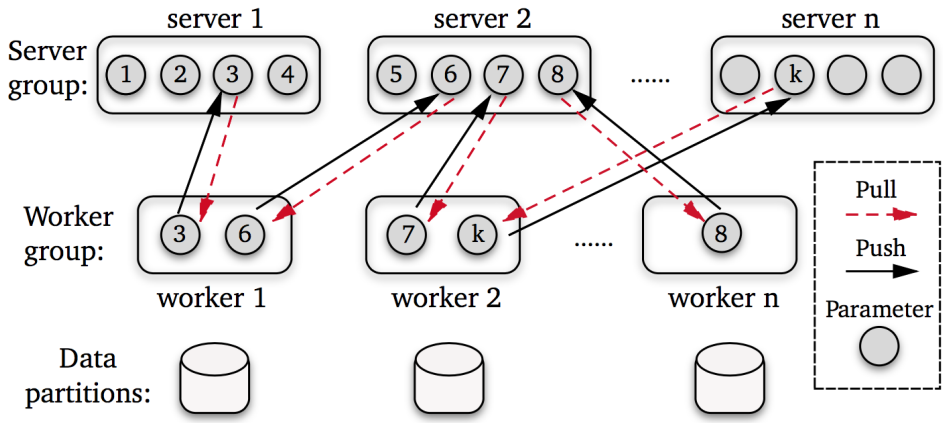
### Out-of-Core Learning Using xLearn R API

The R guide is coming soon.

### 2.6.2 Distributed Learning

As we mentioned before, for some large-scale machine challenges like computational advertising, we focus on the problem with potentially trillions of training examples and billions of model parameters, both of which cannot fit into the memory of a single machine, which brings the *scalability challenge* for users and system designer. For this challenge, parallelizing the training process across machines has become a prerequisite.

The *Parameter Server* (PS) framework has emerged as an efficient approach to solve the "big model" machine learning challenge recently. Under this framework, both the training data and workloads are spread across worker nodes, while the server nodes maintain the globally shared model pa- rameters. The following figure demonstrates the architecture of the PS framework.




As we can see, the *Parameter Server* provides two concise APIs for users.

*Push* sends a vector of (key, value) paris to the server nodes. To be more specific – in the distributed gradient descent, the worker nodes might send the locally computed gradients to servers. Due to the data sparsity, only a part the gradients is non-zero. Often it is desirable to present the gradient as a list of (key, value) pairs, where the feature index is the key and the according gradient item is value.

*Pull* requests the values associated with a list of keys, which will get the newest parameters from the server nodes. This is particularly useful whenever the main memory of a single worker cannot hold a full model. Instead, workers prefetch the model entries relevant for solving the model only when needed.

The distributed training guide for xLearn is coming soon.

## 2.7 xLearn Demo

Copyright of the dataset belongs to the original copyright holder.

### 2.7.1 Criteo CTR Prediction

Predict click-through rates on display ads (Link)

Display advertising is a billion dollar effort and one of the central uses of machine learning on the Internet. However, its data and methods are usually kept under lock and key. In this research competition, CriteoLabs is sharing a week's worth of data for you to develop models predicting ad click-through rate (CTR). Given a user and the page he is visiting, what is the probability that he will click on a given ad?

You can find the data used in this demo in the path `/demo/classification/criteo_ctr/`.

The follow code is the Python demo:

```
import xlearn as xl

# Training task
ffm_model = xl.create_ffm() # Use field-aware factorization machine
ffm_model.setTrain("./small_train.txt")  # Training data
ffm_model.setValidate("./small_test.txt")  # Validation data

# param:
#  0. binary classification
#  1. learning rate: 0.2
#  2. regular lambda: 0.002
#  3. evaluation metric: accuracy
param = {'task':'binary', 'lr':0.2,
         'lambda':0.002, 'metric':'acc'}

# Start to train
# The trained model will be stored in model.out
ffm_model.fit(param, './model.out')

# Prediction task
ffm_model.setTest("./small_test.txt")  # Test data
ffm_model.setSigmoid()  # Convert output to 0-1

# Start to predict
# The output result will be stored in output.txt
ffm_model.predict("./model.out", "./output.txt")
```

### 2.7.2 Mushroom Classification

This dataset comes from UCI Machine Learning Repositpry (Link)

This data set includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family (pp. 500-525). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like *leaflets three, let it be* for Poisonous Oak and Ivy.

You can find a small portion of data used in this demo in the path `/demo/classification/mushroom/`.

The follow code is the Python demo:

```
import xlearn as xl

# Training task
linear_model = xl.create_linear()  # Use linear model
linear_model.setTrain("./agaricus_train.txt")  # Training data
linear_model.setValidate("./agaricus_test.txt")  # Validation data

# param:
#  0. binary classification
#  1. learning rate: 0.2
#  2. lambda: 0.002
#  3. evaluation metric: accuarcy
#  4. use sgd optimization method
param = {'task':'binary', 'lr':0.2,
         'lambda':0.002, 'metric':'acc',
```

(continues on next page)

(continued from previous page)

```
         'opt':'sgd'}

# Start to train
# The trained model will be stored in model.out
linear_model.fit(param, './model.out')

# Prediction task
linear_model.setTest("./agaricus_test.txt")  # Test data
linear_model.setSigmoid()  # Convert output to 0-1

# Start to predict
# The output result will be stored in output.txt
linear_model.predict("./model.out", "./output.txt")
```

### 2.7.3 Predict Survival in Titanic

This challenge comes from the Kaggle. In this challenge, we ask you to complete the analysis of what sorts of people were likely to survive. In particular, we ask you to apply the tools of machine learning to predict which passengers survived the tragedy. (Link)

You can find the data used in this demo in the path `/demo/classification/titanic/`.

The follow code is the Python demo:

```
import xlearn as xl

# Training task
fm_model = xl.create_fm()  # Use factorization machine
fm_model.setTrain("./titanic_train.txt")  # Training data

# param:
#  0. Binary classification task
#  1. learning rate: 0.2
#  2. lambda: 0.002
#  3. metric: accuracy
param = {'task':'binary', 'lr':0.2,
         'lambda':0.002, 'metric':'acc'}

# Use cross-validation
fm_model.cv(param)
```

### 2.7.4 House Price Prediction

This demo shows how to use xLearn to solve the regression problem, and it comes from the Kaggle. The Ames Housing dataset was compiled by Dean De Cock for use in data science education. It's an incredible alternative for data scientists looking for a modernized and expanded version of the often cited Boston Housing dataset. (Link)

You can find the data used in this demo in the path `/demo/regression/house_price/`.

The follow code is the Python demo:

```
import xlearn as xl

# Training task
```

(continues on next page)

(continued from previous page)

```
ffm_model = xl.create_fm()  # Use factorization machine
ffm_model.setTrain("./house_price_train.txt")  # Training data

# param:
#  0. Binary task
#  1. learning rate: 0.2
#  2. regular lambda: 0.002
#  4. evaluation metric: rmse
param = {'task':'reg', 'lr':0.2,
         'lambda':0.002, 'metric':'rmse'}

# Use cross-validation
ffm_model.cv(param)
```

More Demo in xLearn is coming soon.

## 2.8 xLearn Tutorials

1. FFM()
2. FMpython
3. Introductory Guide – Factorization Machines & their application on huge datasets (with codes in Python)
4.