wormhole Documentation

Release 0.1

wormhole developers

Mar 14, 2017

Contents

CHAPTER 1

Tutorial

Binary Classification on the Criteo CTR Dataset

This tutorial gives a step-by-step example for training a binary classifier on the Criteo Kaggle CTR competetion dataset. In this dataset, each example (text line) presents a displayed ad with the label clicked (+1) or not (-1). The goal is to predict the probability of being clicked for a new ad. This is a standard click-through rate (CTR) estimation problem.

In the following we assume a recent Ubuntu (>= 13.10) and bash is used, it should apply to other Linux distributions and Mac OS X too.

Preparation

We first build wormhole using 4 threads:

```
git clone https://github.com/dmlc/wormhole
cd wormhole && make deps -j4 && make -j4
```

Then download the dataset, which has two text files `train.txt` and `test.txt`. Even though wormhole can directly read these two files, we split `train.txt` to multiple files to easy training and validation. The following command divides `train.txt` into multiple 300MB size files, and store them in a compressed row block (crb) format:

```
wget https://s3-eu-west-1.amazonaws.com/criteo-labs/dac.tar.gz
tar -zxvf dac.tar.gz
mkdir data
wormhole/bin/convert.dmlc -data_in train.txt -format_in criteo -data_out data/train -
↪format_out libsvm -part_size 300
```

Linear Method

We first learn a linear logistic regression using `linear.dmlc`. We train on the first 20 parts and validate the model on the last 6 parts. A sparse regularizer $4|w|_1$ is used to control the model complexity. Furthermore, we solve the problem via asynchronous SGD with minibatch size 10000 and learning rate 0.1.

Now generate the configuration file (learn more):

```
cat >train.conf <<EOF
train_data = "data/train-part_[0-1].*"
val_data = "data/train-part_2.*"
data_format = "libsvm"
model_out = "model/criteo"
lambda_l1 = 4
lr_eta = .1
minibatch = 10000
max_data_pass = 1
EOF
```

We train the model using 10 workers and 10 servers:

```
mkdir model
wormhole/tracker/dmlc_local.py -n 10 -s 10 wormhole/bin/linear.dmlc train.conf
```

A possible training log is

```
2015-07-22 04:50:55,285 INFO start listen on 192.168.0.112:9091
connected 10 servers and 10 workers
training #iter = 1
sec #example delta #ex    |w|_0      logloss     AUC    accuracy
  1  1.8e+06  1.8e+06        30509  0.507269  0.758684  0.769462
  2  3.7e+06  1.9e+06        50692  0.469855  0.782046  0.780102
  3  5.5e+06  1.9e+06        70856  0.462922  0.785047  0.784311
  4  7.5e+06    2e+06        85960  0.462718  0.786288  0.783614
  ...
 18  3.4e+07    2e+06       231968  0.453590  0.793880  0.789032
 19  3.6e+07    2e+06       242017  0.454674  0.794033  0.788652
 20  3.7e+07  8.4e+05       248066  0.461133  0.791255  0.784265
validating #iter = 1
sec #example delta #ex    |w|_0      logloss     AUC    accuracy
 30  4.6e+07  9.3e+06       248066  0.459048  0.791334  0.785863
hit max number of data passes
saving final model to model/criteo
training is done!
```

Then we can perform prediction using the trained model. Generate the prediction config file

```
cat >pred.conf <<EOF
val_data = "test.txt"
data_format = "criteo_test"
model_in = "model/criteo"
predict_out = "output/criteo"
EOF
```

and predict:

```
mkdir output
wormhole/tracker/dmlc_local.py -n 10 -s 10 wormhole/bin/linear.dmlc pred.conf
cat output/criteo* >pred.txt
```

Then the i-th line of `pred.txt` will contains the prediction $p = \langle w, x \rangle$ for be i-th example (line) in `test.txt`. We can convert it into a probability by $1/(1 + \exp(-p))$.

Factorization Machine

Factorization machine learns an additional embedding comparing to the linear model, which catches the high-order interactions between features. The usage of `difacto.dmlc` is similar to `linear.dmlc`. First generate the configure file

```
cat >train.conf <<EOF
train_data = "data/train-part_[0-1].*"
val_data = "data/train-part_2.*"
data_format = "libsvm"
model_out = "model/criteo"
embedding {
  dim = 16
  threshold = 16
  lambda_l2 = 0.0001
}
lambda_l1 = 4
lr_eta = .01
max_data_pass = 1
minibatch = 1000
early_stop = 1
EOF
```

Then train the model:

```
wormhole/tracker/dmlc_local.py -n 10 -s 10 wormhole/bin/difacto.dmlc train.conf
```

We can reuse the previous `pred.conf` for prediction:: config file

```
cat >pred.conf <<EOF
val_data = "data/train-part_2.*"
data_format = "libsvm"
model_in = "model/criteo"
predict_out = "output/criteo"
embedding {
  dim = 16
  threshold = 16
  lambda_l2 = 0.0001
}
EOF
```

and predict:

```
wormhole/tracker/dmlc_local.py -n 10 -s 10 wormhole/bin/difacto.dmlc pred.conf
cat output/criteo* >pred.txt
```

What's Next?

- Use another dataset with different formats or storing on HDFS, Amazon S3
- Train the model over multiple machines on Apache Yarn, Amazon EC2

CHAPTER 2

User Guide

Build & Run

Prerequisites

Wormhole can be built on both Linux and Mac OS X. Some apps are also tested on Windows. To build wormhole, both `git` and a recent C++ compiler supporting `C++11`, such as `g++ >= 4.8` and `clang >= 3.5`, are required. Install them on

1. Ubuntu >= 13.10:

```
$ sudo apt-get update && sudo apt-get install -y build-essential git
```

2. Older version Ubuntu via ppa:ubuntu-toolchain-r/test:
3. Centos via devtoolset
4. Mac OS X: can either use the `clang` provided by command line tools or download a compiled gcc from hpc.sourceforge.net

Build

Type `make` to build all apps. It may take several minutes for the first time due to building all dependencies such as `gflags`. There are several options for advanced usages.

make xgboost selectively builds xgboost. Similarly for linear, difactor, ...

make -j4 uses 4 threads for parallel building. For the first building, we suggest to build deps and apps separately: `make deps -j4 && make -j4`

make CXX=g++-4.9 uses a different compiler

make DEPS_PATH=your_path changes the path of the deps. In default all deps will be installed on `wormhole/deps`. We can change the path if them are installed on another place.

make USE_HDFS=1 supports read/write HDFS. It requires `libhdfs`, which is often installed with Hadoop. Apparently Cloudera only ships static version of libhdfs. Hortonworks includes the shared version but not in the lib/native folder. Used ldconfig etc to point compiler, linker and runtime to correct location.

make USE_S3=1 supports read/write AWS S3. `libcurl4-openssl-dev` is required, it can be installed via `sudo apt-get install libcurl4-openssl-dev` on Ubuntu

make dmlc=<dmlc_core_path> in order to run XGBOOST in distributed mode on YARN. Combine with USE_HDFS=1.

Run

Wormhole runs both in a laptop and in a cluster. A typical command to run a application:

```
$ tracker/dmlc_xxx.py -n num_workers [-s num_servers] app_bin app_conf
```

tracker/dmlc_xxx.py the tracker provided by dmlc-core to launch jobs on various platforms

-n number of workers

-s number of servers. Only required for parameter server applications

app_bin the binary of the application, which is available under `bin/`

app_conf the text configuration file specifying dataset and learning method, see each app's documents for details

Local machine

The following command runs linear logistic regression using two workers and a single server on a small dataset:

```
$ tracker/dmlc_local.py -n 2 -s 1 bin/linear.dmlc learn/linear/guide/demo.conf
```

Apache Yarn

First make sure the environments `HADOOP_HOME` and `JAVA_HOME` are set properly. Next compile the Yarn tracker:

```
$ cd repo/dmlc-core/yarn && ./build.sh
```

Then a Yarn job can be submitted via `tracker/dmcl_yarn.py`. For example, the following codes run xgboost on Yarn

```
hdfs_path=/your/path

hadoop fs -mkdir ${hdfs_path}/data
hadoop fs -put learn/data/agaricus.txt.train ${hdfs_path}/data
hadoop fs -put learn/data/agaricus.txt.test ${hdfs_path}/data

tracker/dmlc_yarn.py  -n 4 --vcores 2 bin/xgboost.dmlc \
  learn/xgboost/mushroom.hadoop.conf nthread=2 \
  data=hdfs://${hdfs_path}/data/agaricus.txt.train \
  eval[test]=hdfs://${hdfs_path}/data/agaricus.txt.test \
  model_out=hdfs://${hdfs_path}/mushroom.final.model
```

Run `tracker/dmlc_yarn.py -h` for more details.

Sun Grid Engine

Use `tracker/dmlc_sge.py`

MPI

Wormhole can be run over multiple machines via `mpirun`, which is often convenient for a small cluster. Assume file `hosts` stores the hostnames of all machines, then use:

```
$ tracker/dmlc_mpi.py -n num_workers -s num_servers -H hosts bin conf
```

to launch wormhole on these machines. See next section for an example to setup a cluster with mpirun.

Setup an EC2 Cluster from Scratch

In this section we give a tutorial to setup a small cluster and launch wormhole jobs on Amazon EC2.

1. Assume all data are stored Amazon S3.
2. Use a middle range instance as the master node to build wormhole and submit jobs, and several high end instances to do the computations.
3. Use `NFS` to dispatch binaries and configurations and `mpirun` to launch jobs.

Setup the master node

First launch an Ubuntu 14.04 instance as the master node. It is mainly used for compiling codes, a middle end instance such as *c4.xlarge* is often good enough. Install required libraries via:

```
$ sudo apt-get update && sudo apt-get install -y build-essential git libcurl4-openssl-
↪dev
```

Then build wormhole with S3 support:

```
$ git clone https://github.com/dmlc/wormhole.git
$ cd wormhole && make deps -j4 && make -j4 USE_S3=1
```

Next setup NFS:

```
$ sudo apt-get install nfs-kernel-server mpich2
$ echo "/home/ubuntu/  *(rw,sync,no_subtree_check)" | sudo tee /etc/exports
$ sudo service nfs-kernel-server start
```

Finally copy the *pem* file used to access the master node to master node's *~/.ssh/id_rsa* so that this node can access to all other machines.

Setup the slave nodes

First launch several Ubuntu 12.04 instances with the same pem file as the slaves nodes. High-end instances such as c4.4xlarge and c4.8xlarge are recommended. Save their private IPs in file *hosts*:

```
$ cat hosts
172.30.0.172
172.30.0.171
172.30.0.170
```

Then install both NFS and mpirun on these slave nodes. Assume the master node has private IP `172.30.0.160`:

```
while read h; do
  echo $h
  ssh -o StrictHostKeyChecking=no $h <<'ENDSSH'
sudo apt-get update
sudo apt-get install -y nfs-common mpich2
sudo mount 172.30.0.160:/home/ubuntu /home/ubuntu
ENDSSH
done <hosts
```

Next install depended libraries on all slave nodes:

```
$ mpirun -hostfile hosts sudo apt-get install -y build-essential libcurl4-openssl-dev
```

Put all things together

Test if everything is OK:

```
$ mpirun -hostfile hosts uname -a
$ mpirun -hostfile hosts ldd wormhole/bin/linear.dmlc
```

Now we can submit jobs from the master node via:

```
$ wormhole/tracker/dmlc_mpi.py -n ? -s ? -H hosts wormhole/bin/? ?.conf
```

Input Data

Wormhole supports various input data sources and formats.

Data Formats

Both text and binary formats are supported.

LIBSVM

Wormhole supports a more general version of the LIBSVM format. Each example is presented as a text line:

```
label feature_id[:weight] feature_id[:weight] ... feature_id[:weight]
```

label a `float` label

feature_id a `unsigned 64-bit integer` feature index. It is not required to be continuous.

weight: the according `float` weight, which is optional

Compressed Row Block (CRB)

This is a compressed binary data format. One can use `bin/text2crb` to convert any supported data format into it.

Customized Format

Adding a customized format requires only two steps.

1. Define a subclass to implement the function `ParseNext` of ParserImpl. Examples:
   - Parse the text Criteo CTR dataset criteo_parser
   - Parse the binary `crb` format: crb_parser
2. Then add the this new parser to a reader. For example, adding them in the minibatch reader

Data Sources

Besides standard filesystems, wormhole supports the following distributed filesystems.

HDFS

To support HDFS, compile with the flag `USE_HDFS=1` such as `make USE_HDFS=1` or set the flag in `config.mk`. An example filename of a HDFS file

```
hdfs:///user/you/ctr_data/day_0
```

Amazon S3

To supports Amazon S3, compile with the flag `USE_S3=1`. Besides, one needs to set the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` properly. For example, add the following two lines in `~/.bashrc` (replace the strings with your AWS credentials):

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

An example filename of a S3 file

```
s3://ctr-data/day_0
```

Microsoft Azure Blob Storage (Alpha support)

To support Azure blob storage, compile with the flag `USE_AZURE=1` and `DEPS_PATH=deps`, which needs the Azure C++ Storage SDK (https://github.com/Azure/azure-storage-cpp)

Install Azure Storage SDK (TODO: move to make/deps.mk) :: sudo apt-get -y install libboost1.54-all-dev libssl-dev cmake libxml++2.6-dev libxml++2.6-doc uuid-dev

cd deps && mkdir -p lib include

git clone https://git.codeplex.com/casablanca cd casablanca/Release mkdir build.release cd build.release CXX=g++ cmake .. -DCMAKE_BUILD_TYPE=Release make -j4 cp Binaries/libcpprest* ../../../lib cp -r ../include/* ../../../include/ cd ../../..

git clone https://github.com/Azure/azure-storage-cpp cd azure-storage-cpp/Microsoft.WindowsAzure.Storage mkdir build.release cd build.release CASABLANCA_DIR=../../../../casablanca/ CXX=g++ cmake .. -DCMAKE_BUILD_TYPE=Release make -j4 cp Binaries/libazurestorage* ../../../lib cp -r ../includes/* ../../../include/ cd ../../../..

One also needs to set the environment variables properly (About Azure storage account):

```
export AZURE_STORAGE_ACCOUNT=mystorageaccount
export AZURE_STORAGE_ACCESS_KEY=EXAMPLEKEY
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/dmlc-core/deps/lib
```

An example filename of an Azure file :: azure://container/agaricus.txt.test

Linear Method

Given data pairs (x, y), the linear method learns the model vector w by minizing the following objective function:

$$\sum_{(x,y)} \ell(y, \langle x, w \rangle) + \lambda_1 |w|_1 + \lambda_2 \|w\|_2^2$$

where $\ell(y, p)$ is the loss function, see *Config.Loss*.

Configuration

The configuration is defined in the protobuf file config.proto

Input & Output

Type	Field	Description
string	train_data	The training data, can be either a directory or a wildcard filename
string	val_data	The validation or test data, can be either a directory or a wildcard filename
string	data_format	data format. supports libsvm, crb, criteo, adfea, ...
string	model_out	model output filename
string	model_in	model input filename
string	predict_out	the filename for prediction output. if specified, then run/ prediction. otherwise run training

Model and Optimization

Type	Field	Description
Config.Loss	loss	the loss function. default is LOGIT
float	lambda_l1	l1 regularizer: $\lambda_1 \|w\|_1$
float	lambda_l2	l2 regularizer: $\lambda_2 \|w\|_2^2$
Config.Algo	algo	the learning method, default is FTRL
int32	minibatch	the size of minibatch. the smaller, the faster the convergence, but the/ slower the system performance
int32	max_data_pass	the maximal number of data passes
float	lr_eta	the learning rate η (or α). often uses the largest/ value when not diverged

Config.Loss

Name	Description
SQUARE	square loss: $\frac{1}{2}(p-y)^2$
LOGIT	logistic loss: $\log(1+\exp(-yp))$
SQUARE_HINGE	squared hinge loss: $\max\left(0,(1-yp)^2\right)$

Config.Algo

Name	Description
SGD	asynchronous minibatch SGD
ADAGRAD	similar to SGD, but use adagrad
FTRL	similar to ADAGRAD, but use FTRL for better sparsity

Adavanced Configurations

Type	Field	Description
int32	save_iter	save model for every k data pass. default is -1, which only saves for the/ last iteration
int32	load_iter	load model from the k-th iteration. default is -1, which loads the last/ iteration model
bool	local_data	give a worker the data only if it can access. often used when the data has/ been dispatched to workers' local filesystem
int32	num_parts_per_file	virtually partition a file into n parts for better loadbalance. default is 10
int32	rand_shuffle	randomly shuffle data for minibatch SGD. a minibatch is randomly picked from/ rand_shuffle * minibatch examples. default is 10.
float	neg_sampling	down sampling negative examples in the training data. no in default
bool	prob_predict	if true, then outputs a probability prediction. otherwise $\langle x, y\rangle$
float	dropout	the probably to set a gradient to 0. no in default
float	print_sec	print the progress every n sec during training. 1 sec in default
float	lr_beta	learning rate β, 1 in default
int32	num_threads	number of threads used by a worker / a server. 2 in default
int32	max_concurrency	the maximal concurrent minibatches being processing at the same time for/ sgd, and the maximal concurrent blocks for block CD. 2 in default.
bool	key_cache	cache the key list on both sender and receiver to reduce communication/ cost. it may increase the memory usage
bool	msg_compression	compression the message to reduce communication cost. it may increase the/ computation cost.
int32	fixed_bytes	convert floating-points into fixed-point integers with n bytes. n can be 1,/ 2 and 3. 0 means no compression.

Performance

Factorization Machine

Difacto is refined factorization machine (FM) with sparse memory adaptive constraints.

Given an example $x \in \mathbb{R}^d$ and an embedding dimension k, FM models the example by

$$f(x) = \langle w, x\rangle + \frac{1}{2}\|Vx\|_2^2 - \sum_{i=1}^{d} x_i^2 \|V_i\|_2^2$$

where $w \in \mathbb{R}^d$ and $V \in \mathbb{R}^{d \times k}$ are the models we need to learn. The learning objective function is

$$\frac{1}{|X|} \sum_{(x,y)} \ell(f(x), y) + \lambda_1 |w|_1 + \frac{1}{2} \sum_{i=1}^{d} \left[\lambda_i w_i^2 + \mu_i \|V_i\|^2 \right]$$

where the first sparse regularizer $\lambda_1 |w|_1$ induces a sparse w, while the second term is a frequency adaptive regularization, which places large penalties for more frequently features.

Furthermore, Difacto adds two heuristics constraints

- $V_i = 0$ if $w_i = 0$, namely we mark the embedding for feature *i* is inactive if the according linear term is filtered out by the sparse regularizer. (You can disable it by `l1_shrk = false`)
- $V_i = 0$ if the occur of feature i is less the a threshold. In other words, Difacto does not learn an embedding for tail features. (You can specify the threshold via `threshold = 10`)

Train by Asynchronous SGD. *w* is updated via FTRL while *V* via adagrad.

Configuration

The configure is defined in the protobuf file config.proto

Input & Output

Type	Field	Description
string	train_data	The training data, can be either a directory or a wildcard filename
string	val_data	The validation or test data, can be either a directory or a wildcard filename
string	data_format	data format. supports libsvm, crb, criteo, adfea, ...
string	model_out	model output filename
string	model_in	model input filename
string	predict_out	the filename for prediction output. if specified, then run/ prediction. otherwise run training

Model and Optimization

Type	Field	Description
float	lambda_l1	l1 regularizer for w: $\lambda_1 \|w\|_1$
float	lambda_l2	l2 regularizer for w: $\lambda_2 \|w\|_2^2$
float	lr_eta	learning rate η (or α) for w
Config.Embedding	embedding	the embedding V
int32	minibatch	the size of minibatch. the smaller, the faster the convergence, but the/ slower the system performance
int32	max_data_pass	the maximal number of data passes
bool	early_stop	stop earilier if the validation objective is less than prev_obj - min_objv_decr

Config.Embedding

embedding V. basic:

Type	Field	Description
int32	dim	the embedding dimension k
int32	threshold	features with occurence < threshold have no embedding ($k = 0$)
float	lambda_l2	l2 regularizer for V: $\lambda_2 \|V_i\|_2^2$

advanced:

Type	Field	Description
float	init_scale	V is initialized by uniformly random weight in/ [-init_scale, +init_scale]
float	dropout	apply dropout on the gradient of V. no in default
float	grad_clipping	project the gradient of V into $[-cc]$. no in default
float	grad_normalization	normalized the l2-norm of gradient of V. no in default
float	lr_eta	learning rate η for V. if not specified, then share the same with w
float	lr_beta	leanring rate β for V.

Adavanced Configurations

Type	Field	Description
int32	save_iter	save model for every k data pass. default is -1, which only saves for the/ last iteration
int32	load_iter	load model from the k-th iteration. default is -1, which loads the last/ iteration model
bool	local_data	give a worker the data only if it can access. often used when the data has/ been dispatched to workers' local filesystem
int32	num_parts_per_file	virtually partition a file into n parts for better loadbalance. default is 10
int32	rand_shuffle	randomly shuffle data for minibatch SGD. a minibatch is randomly picked from/ rand_shuffle * minibatch examples. default is 10.
float	neg_sampling	down sampling negative examples in the training data. no in default
bool	prob_predict	if true, then outputs a probability prediction. otherwise $\langle x, y \rangle$
float	print_sec	print the progress every n sec during training. 1 sec in default
float	lr_beta	learning rate β, 1 in default
float	min_objv_decr	the minimal objective decrease in early stop
bool	l1_shrk	use or not use the contraint $V_i = 0$ if $w_i = 0$. yes in default
int32	num_threads	number of threads used within a worker and a server
int32	max_concurrency	the maximal concurrent minibatches being processing at the same time for/ sgd, and the maximal concurrent blocks for block CD. 2 in default.
bool	key_cache	cache the key list on both sender and receiver to reduce communication/ cost. it may increase the memory usage
bool	msg_compression	compression the message to reduce communication cost. it may increase the/ computation cost.
int32	fixed_bytes	convert floating-points into fixed-point integers with n bytes. n can be 1,/ 2 and 3. 0 means no compression.

Performance

CHAPTER 3

Developer Guide