
PMLS Documentation

Release

PMLS, Inc.

Jul 07, 2017

Contents

1	PMLS Bösen and Strads Installation	3
1.1	Foreword and Supported Operating Systems	3
1.2	Obtaining PMLS	3
1.3	Compiling PMLS	4
1.4	Compiling PMLS Bösen with cmake	4
1.5	Very important: Setting up password-less SSH authentication	5
1.6	Shared directories	5
1.7	Network ports to open	5
1.8	Cloud compute support	6
1.9	Getting started with applications	6
2	Configuration	7
2.1	Make sure password-less SSH is set up correctly	7
2.2	Bösen and Strads	7
2.3	Bösen configuration files	7
2.4	I want to run on my local machine	8
2.5	My cluster does not have shared directories	8
2.6	Caution - Please Read!	8
2.7	Strads configuration files	9
2.8	I want to run on my local machine	9
2.9	My cluster does not have shared directories	10
3	PMLS YARN+HDFS support	11
3.1	Preliminaries	11
3.2	Recompiling Bösen	12
3.3	Which applications are supported?	12
3.4	Troubleshooting	12
4	Frequently asked questions	15
5	Latent Dirichlet Allocation (LDA)	17
5.1	Introduction to LDA	17
5.2	Performance	17
5.3	Quick start	17
5.4	Input data format	18
5.5	Output format	18
5.6	Program options	18

6	MedLDA	21
6.1	Performance	21
6.2	Installation	21
6.3	Data Preparation	22
6.4	Running MedLDA	22
6.5	Configuration and using multiple machines	25
6.6	Command line flags	26
7	Deep Neural Network	27
7.1	Performance	27
7.2	Quick start	27
7.3	Making predictions	28
7.4	Input data format	28
7.5	Creating synthetic data	29
7.6	Running the Deep Neural Network Application	29
7.7	Format of DNN Configuration File	30
7.8	Terminating the DNN app	31
7.9	File IO from HDFS	31
7.10	Use Yarn to launch DNN app	31
8	Deep Neural Network for Speech Recognition	33
8.1	Installation	33
8.2	The Whole Pipeline	34
8.3	Running the Deep Neural Network application	35
8.4	Input data format	36
8.5	Format of DNN Configuration File	37
8.6	Output format	37
8.7	Terminating the DNN app	38
9	Matrix Factorization	39
9.1	Performance	39
9.2	Quick start	39
9.3	Input data format	40
9.4	Output format	40
9.5	Program options	40
10	Non-negative Matrix Factorization (NMF)	43
10.1	Introduction to NMF	43
10.2	Quick Start	43
10.3	Data format	44
10.4	Creating synthetic data	45
10.5	Running the NMF application	45
10.6	Terminating the NMF app	47
10.7	Data partitioning	47
10.8	File IO from HDFS	47
10.9	Use Yarn to launch NMF app	48
11	Sparse Coding	49
11.1	Performance	49
11.2	Quick Start	49
11.3	Data format	51
11.4	Creating synthetic data	51
11.5	Running the Sparse Coding application	51
11.6	Terminating the Sparse Coding app	53
11.7	Data partitioning	53

11.8	File IO from HDFS	54
11.9	Use Yarn to launch Sparse Coding app	54
12	Lasso and Logistic Regression	55
12.1	Performance	55
12.2	Input data format	56
12.3	Output format	56
12.4	Machine configuration	56
12.5	Program Options	56
13	Distance Metric Learning	59
13.1	Performance	59
13.2	Quick start	59
13.3	Running the Distance Metric Learning Application	60
13.4	Format of DML Configuration File	60
13.5	Input data format	61
13.6	Output model format	61
13.7	Terminate DML app	62
13.8	File IO from HDFS	62
13.9	Use Yarn to launch DML app	62
14	K-Means Clustering	63
14.1	Quick Start	63
14.2	Use HDFS	63
14.3	Use Yarn	64
14.4	Input Format	65
14.5	Setting up machines	65
14.6	Common Parameters	65
14.7	Center Initialization	66
14.8	Output	66
14.9	References	66
15	Random Forest	67
15.1	Quick Start	67
15.2	Use HDFS	68
15.3	Use Yarn	69
15.4	Input format	69
15.5	Setting up machines	69
15.6	Common parameters	70
15.7	Save prediction to file	70
15.8	Save trained trees to file	70
15.9	Load trained trees from file	71
15.10	Finish up	71
16	Support Vector Machine	73
16.1	Performance	73
16.2	Input data format	73
16.3	Output format	74
16.4	Machine configuration	74
16.5	Program Options	74
17	Multi-class Logistic Regression	75
17.1	Performance	75
17.2	Preliminaries	75
17.3	Quick Start	75

17.4	Use HDFS	76
17.5	Use Yarn	77
17.6	Data Format	77
17.7	Synthetic Data	78
17.8	MLR Details	78
17.9	Terminating the MLR app	80
18	Home	81
18.1	Foreword - please read	81
18.2	PMLS Bösen/Strads v1.1 manual	81
18.3	Introduction to PMLS	82
18.4	Key PMLS features	83
18.5	Support and Bug reports	83

The fastest way to get started with PMLS is to use this [script](#), which will setup Bösen and Strads systems on a single machine with just 1 command. After setting it up, you can run two demo applications to verify that they are working. To get the script:

```
wget https://gist.githubusercontent.com/holyglenn/dc3a2b8a5d496735a0a297b0d5ec3479/
↳raw/2b21c2cf23d0360d2b4760e92fdb308ab263dd49/petuum.py
```

Before start, run the following command to prepare compilation environment. This is the only setup where sudo is required. If you do not have sudo privilege, please contact your administrator for help.

```
sudo apt-get -y update && sudo apt-get -y install g++ make autoconf git libtool uuid-
↳dev openssh-server cmake libopenmpi-dev openmpi-bin libssl-dev libnuma-dev python-
↳dev python-numpy python-scipy python-yaml protobuf-compiler subversion libxml2-dev
↳libxslt-dev zlibc zlib1g zlib1g-dev libbz2-1.0 libbz2-dev
```

After getting the compilation environment ready, you are good to run PMLS with or without sudo.

If you have sudo privilege, you can install part of PMLS's dependencies to save compilation time.

```
sudo apt-get -y install libgoogle-glog-dev libzmq3-dev libyaml-cpp-dev \
libgoogle-perftools-dev libsnappy-dev libsparsehash-dev libgflags-dev libeigen3-dev
```

Then run the following command to setup PMLS, which takes approximately 10 minutes on a 2-core machine. The script will first enable passwordless ssh connection to localhost using default id_rsa.pub key or generate one if without. Then it will download and compile Bösen and Strads systems and their customized dependencies. By default PMLS is under ~/petuum_test.

```
python petuum.py setup
```

If you don't have sudo, run the setup command with --no-sudo argument. In addition to the setup process above, the script will compile all PMLS's dependencies and install them in its local folder. This process takes about 20 minutes.

```
python petuum.py setup --no-sudo
```

Now PMLS is ready to go. To run the Multi-class Logistic Regression demo (in Bösen system),

```
python petuum.py run_mlr
```

The app launches locally and trains multi-class logistic regression model using a subset of the [Covertypes dataset](#). You should see something like below. The numbers will be slightly different as it's executed indeterministically with multi-threads.

```
40 400 0.253846 0.61287 520 0.180000 50 7.43618
I0701 00:35:00.550900 9086 mlr_engine.cpp:298] Final eval: 40 400 train-0-1: 0.
↳253846 train-entropy: 0.61287 num-train-used: 520 test-0-1: 0.180000 num-test-used:
↳50 time: 7.43618
I0701 00:35:00.551867 9086 mlr_engine.cpp:425] Loss up to 40 (exclusive) is saved to
↳/home/ubuntu/petuum/app/mlr/out.loss in 0.000955387
I0701 00:35:00.552652 9086 mlr_sgd_solver.cpp:160] Saved weight to /home/ubuntu/
↳petuum/app/mlr/out.weight
I0701 00:35:00.553907 9031 mlr_main.cpp:150] MLR finished and shut down!
```

To run the MedLDA supervised topic model (in STRADS system), run

```
python petuum.py run_lda
```

The app launches 3 workers locally and trains with [20 newsgroup](#) dataset. You will see outputs like below. Once all workers have reported “Ready to exit program”, you may Ctrl-C to terminate the program.

```
.....
Rank (2) Ready to exit program from main function in ldall.cpp
I1222 20:38:31.271615 2687 trainer.cpp:464] (rank:0) Dict written into /tmp/dump_dict
I1222 20:38:31.271632 2687 trainer.cpp:465] (rank:0) Total num of words: 53485
I1222 20:38:46.930896 2687 trainer.cpp:487] (rank:0) Model written into /tmp/dump_
↪model
Rank (0) Ready to exit program from main function in ldall.cpp
```

Use the following command to display top 10 words in each of the topics that’s just generated.

```
python petuum.py display_topics
```

If you seek further deployment or prefer a more detailed hands-on experience, please refer to the [full installation guide](#) or the [manual](#). Also check out [PMLS-Caffe](#), the multi-GPU distributed deep learning framework of PMLS.

PMLS Bösen and Strads Installation

Foreword and Supported Operating Systems

PMLS Bösen is a communication-efficient distributed key-value store (parameter server) for data-parallel Machine Learning, and PMLS Strads is a dynamic scheduler for model-parallel Machine Learning. Both Bösen and Strads have been officially tested on **64-bit Ubuntu Desktop 14.04** (available at: <http://www.ubuntu.com/download/desktop>). The instructions in this tutorial are meant for Ubuntu 14.04.

We have also successfully tested PMLS on some versions of RedHat and CentOS. However, the commands for installing dependencies in this manual are specific to 64-bit Ubuntu Desktop 14.04. They do not apply to RedHat/CentOS; you will need to know the corresponding packages in `yum`.

Note: Server versions of Ubuntu may require additional packages above those listed here, depending on your configuration.

Obtaining PMLS

The best way to download PMLS is via the `git` command. Install `git` by running

```
sudo apt-get -y update
sudo apt-get -y install git
```

Then, run the following commands to download PMLS Bösen and Strads:

```
git clone -b stable https://github.com/sailing-pmls/bosen.git
git clone https://github.com/sailing-pmls/strads.git
cd bosen
git clone https://github.com/sailing-pmls/third_party.git third_party
cd ..
```

Next, for each machine that PMLS will be running on, execute the following commands to install dependencies:

```
sudo apt-get -y update
sudo apt-get -y install g++ make autoconf git libtool uuid-dev openssh-server cmake
↪libopenmpi-dev openmpi-bin libssl-dev libnuma-dev python-dev python-numpy python-
↪scipy python-yaml protobuf-compiler subversion libxml2-dev libxslt-dev zlibc zlib1g
↪zlib1g-dev libbz2-1.0 libbz2-dev
```

Warning: Some parts of PMLS require openmpi, but are incompatible with mpich2 (e.g. in the Anaconda scientific toolkit for Python). If you have both openmpi and mpich2 installed, make sure `mpirun` points to openmpi's executable.

Compiling PMLS

You're now ready to compile PMLS. From the directory in which you started, run

```
cd strads
make
cd ../bosen/third_party
make
cd ../../bosen
cp defns.mk.template defns.mk
make
cd ..
```

If you are installing PMLS to a shared filesystem, **the above steps only need to be done from one machine.**

The first make builds Strads, and the second and third makes build Bösen and its dependencies. All commands will take between 5-30 minutes each, depending on your machine. We'll explain how to compile and run PMLS's built-in apps later in this manual.

Compiling PMLS Bösen with cmake

Run the following commands to download PMLS Bösen.

```
git clone https://github.com/sailing-pmls/bosen.git
```

For each machine that PMLS will be running on, execute the following commands to install dependencies and libraries.

```
sudo apt-get -y install libgoogle-glog-dev libzmq3-dev libyaml-cpp-dev \
  libgoogle-perftools-dev libsnappy-dev libsparsehash-dev libgflags-dev \
  libboost-thread1.55-dev libboost-system1.55-dev libleveldb-dev \
  libconfig++-dev libeigen3-dev libevent-pthreads-2.0-5
```

You're now ready to compile PMLS. Run

```
cd bosen
mkdir build
cd build && cmake .. && make -j
```

If you are installing PMLS to a shared filesystem, **the above steps only need to be done from one machine.** The process takes about 5 minutes.

Very important: Setting up password-less SSH authentication

PMLS uses `ssh` (and `mpirun`, which invokes `ssh`) to coordinate tasks on different machines, **even if you are only using a single machine**. This requires password-less key-based authentication on all machines you are going to use (PMLS will fail if a password prompt appears).

If you don't already have an SSH key, generate one via

```
ssh-keygen
```

You'll then need to add your public key to each machine, by appending your public key file `~/.ssh/id_rsa.pub` to `~/.ssh/authorized_keys` on each machine. If your home directory is on a shared filesystem visible to all machines, then simply run

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

If the machines do not have a shared filesystem, you need to upload your public key to each machine, and then append it as described above.

Note: Password-less authentication can fail if `~/.ssh/authorized_keys` does not have the correct permissions. To fix this, run `chmod 600 ~/.ssh/authorized_keys`.

Shared directories

We **highly recommend using PMLS in an cluster environment with a shared filesystem, such as NFS**. To set up NFS on Ubuntu machines, you may refer to [here](#).

Provided all machines are identically configured and have the necessary packages/libraries from `apt-get`, **you only need to compile PMLS and its applications once, from one machine**. The PMLS ML applications are all designed to work in this environment, as long as the input data and configuration files are also available through the shared filesystem.

If your cluster doesn't have shared directories, some PMLS applications can still work, but we do not officially support this. You'll need to take the following extra steps:

1. **Ensure all machines are identically configured:** they must be running the same Linux distro (with the same machine architecture, e.g. `x86_64`), and the packages described earlier must be installed on every machine. The machines need not have exactly identical hardware, but the Linux software environment must be the same. **PMLS will fail if you have different versions of `gcc` or the needed software packages across different machines.**
2. You need to copy or `git clone` PMLS onto every machine, **at exactly the same path** (e.g. `/home/username/pmls`). **You must compile PMLS and the PMLS apps separately on each machine.**
3. When running PMLS apps, the input data and configuration files must be present on every machine, **at exactly the same path.**

Network ports to open

If you have a firewall, you must open these ports on all machines:

- SSH port: 22
- Bösen apps: port range 9999-10998 (you can change these)
- Strads apps: port ranges 47000-47999 and 38000-38999

Cloud compute support

PMLS can run in any Linux-based cloud environment that supports SSH; we recommend using 64-bit Ubuntu 14.04. If you wish to run PMLS on Amazon EC2, we recommend using the official 64-bit Ubuntu 14.04 Amazon Machine Images provided by Canonical: <http://cloud-images.ubuntu.com/releases/14.04/release/>.

If you're using Red Hat Enterprise Linux or CentOS on Google Compute Engine, you need to turn off the `iptables` firewall (which is on by default), or configure it to allow traffic through ports 9999-10998 (or whatever ports you intend to use). See <https://developers.google.com/compute/docs/troubleshooting#knownissues> for more info.

Getting started with applications

Now that you have successfully set up PMLS on one or more machines, you can try out some applications. We recommend getting started with:

- *Bosen: Non-negative Matrix Factorization*
- *Strads: Latent Dirichlet Allocation*

Make sure password-less SSH is set up correctly

This is the most common issue people have when running PMLS. Please read!

You must be able to `ssh` into all machines without a password - even if you are only using your local machine. The PMLS apps will fail in unexpected ways if password-less `ssh` is not working. When this happens, you will not see any error output stating that this is the problem!

Hence, you will save yourself a lot of trouble by taking the time to ensure password-less `ssh` actually works, before you attempt to run the PMLS apps. For example, if you are going to run PMLS on your local machine, make sure that `ssh 127.0.0.1` logs you in without asking for a password. See [Installing PMLS](#) for instructions on how to do this.

Bösen and Strads

PMLS includes two platforms for writing and running ML applications: Bösen for data-parallel execution, and Strads for model-parallel execution. Each PMLS ready-to-run application is either a Bösen application, or a Strads application. The two systems use different machine configuration files; please see the following guides.

Note: This page explains machine configuration for non-YARN, stand-alone operation. If you are looking to run PMLS on YARN, please see [this page](#).

Bösen configuration files

Some PMLS ML applications require Bösen configuration files, in the following format:

```
0 ip_address_0 9999
1 ip_address_1 9999
2 ip_address_2 9999
```

```
3 ip_address_3 9999
...
```

IMPORTANT: There cannot be any trailing blank lines in the configuration file. If there are trailing blank lines, the applications will fail.

You can give your configuration file any filename. The placeholders `ip_address_0`, `ip_address_1`, etc. are the IP addresses of each machine you want to use. If you only know the hostname of the machine, for example `work-machine-1.mycluster.com`, use `host` to get the actual IP (the hostname will not work):

```
host work-machine-1.mycluster.com
```

Each line in the server configuration file format specifies an ID (0, 1, 1000, 2000, etc.), the IP address of the machine assigned to that ID, and a starting port number (9999). Every machine is assigned to one ID and one starting port number. We say “starting port number” because some applications may use additional consecutive ports, up to a maximum of 100 ports (i.e. the port range 9999-10998).

I want to run on my local machine

Simply create this localhost configuration file:

```
0 127.0.0.1 9999
```

My cluster does not have shared directories

PMLS is meant to be used from a shared filesystem, such as NFS - you can find instructions for how to set up NFS on Ubuntu machines [here](#).

If you don’t have NFS, you can still run PMLS, but you need to pay attention to the following points:

Make sure that PMLS is compiled on every machine, and the machine configuration files are present on every machine. The paths to PMLS and machine configuration files must be identical across machines.

Different apps have different requirements for input data; if no requirements are explicitly stated, you should make sure the input data is present on every machine.

Some apps store their arguments and parameters in script files. If you modify these script files, you should copy them to every machine, to be safe.

Caution - Please Read!

You cannot `ctrl-c` to terminate a Bösen app, because they are invoked in the background via `ssh`. Each Bösen app comes with a kill script for this purpose; please see the individual app sections in this manual for instructions.

If you want to simultaneously run two PMLS apps on the same machines, make sure you give them separate Bösen configuration files with different starting ports, separated by at least 1000 (e.g. 9999 and 10999). **The apps cannot share the same port ranges!**

If you cannot run an app - especially if you see error messages with “Check failure stack trace” - the cause is probably another running (or hung) PMLS app using the same ports. In that case, you should use the offending app’s kill script to terminate it.

Strads configuration files

Some PMLS ML applications require Strads machine configuration files, which are simply a list of machine IP addresses (like an MPI hostfile). Unlike Bösen configuration files, Strads configuration files may repeat IP addresses.

Strads creates three types of processes: Workers, the Scheduler, and the Coordinator.

- The Coordinator is the master process that oversees the Scheduler and Workers, and is spawned on the last IP in the machine file.
- The Scheduler creates and dispatches model variables for the Workers to update. Unless otherwise stated, there is only one Scheduler, spawned on the 2nd last IP in the machine file.
- The Workers perform the actual variable updates to the ML program. They are spawned on the remaining IPs in the machine file. **At least 2 workers are required.**

You may repeat IP addresses to put multiple processes on the same machine. However, the repeat IPs must be contiguous.

Example 1: If you want 4 workers, then the machine file looks like this:

```
worker1-ip-address
worker2-ip-address
worker3-ip-address
worker4-ip-address
scheduler-ip-address
coordinator-ip-address
```

Example 2: Let's say you only have 2 machines, and you want to spawn 2 Workers on the 1st machine, and the Scheduler and the Coordinator on the 2nd machine. The machine file looks like this:

```
ip-address-1
ip-address-1
ip-address-2
ip-address-2
```

Because repeat IPs must be contiguous, the following configuration is invalid:

```
ip-address-1    (this configuration WILL NOT WORK)
ip-address-2
ip-address-1
ip-address-2
```

I want to run on my local machine

If you only have 1 machine, use this Strads machine file with 2 Workers (plus the Scheduler and Coordinator):

```
127.0.0.1
127.0.0.1
127.0.0.1
127.0.0.1
```

My cluster does not have shared directories

PMLS is meant to be used from a shared filesystem, such as NFS - you can find instructions for how to set up NFS on Ubuntu machines [here](#).

If you don't have NFS, you can still run PMLS, but you need to pay attention to the following points:

Make sure that PMLS is compiled on every machine, and the machine configuration files are present on every machine. The paths to PMLS and machine configuration files must be identical across machines.

Different apps have different requirements for input data; if no requirements are explicitly stated, you should make sure the input data is present on every machine.

Some apps store their arguments and parameters in script files. If you modify these script files, you should copy them to every machine, to be safe.

PMLS YARN+HDFS support

As of version 1.1, PMLS includes support for running apps on YARN and HDFS (Hadoop version 2.4 or later). To enable support for YARN and HDFS, please follow the instructions below.

Preliminaries

Before you begin, please ensure the following:

- **hadoop** command is working on every machine.
- **Java 7 or higher** is installed.

First, install PMLS on every machine in the YARN cluster, as described in [Installation](#). You may install PMLS to the local filesystem; NFS is not required for YARN operation.

Second, you will need to install Gradle to build the PMLS Java components required for YARN. Follow the instructions on the [Gradle webpage](#) to install Gradle, and ensure that the command `gradle` (from the `bin` subdirectory) is in your `PATH`.

We do not recommend installing Gradle via `apt-get` on Ubuntu 14.04, as the repository version may be out of date.

Finally, on each machine, open `bosen/defs.mk`, and note the following lines:

```
JAVA_HOME = /usr/lib/jvm/java-7-openjdk-amd64
HADOOP_HOME = /usr/local/hadoop/hadoop-2.6.0
HAS_HDFS = # Leave empty to build without hadoop.
#HAS_HDFS = -DHAS_HADOOP # Uncomment this line to enable hadoop
```

Change these lines to:

```
JAVA_HOME = <path to Java 7 or Java 8 on your machines>
HADOOP_HOME = <path to Hadoop on your machines>
HAS_HDFS = -DHAS_HADOOP # Uncomment this line to enable hadoop
```

Recompiling Bösen

You must now recompile Bösen and the supported Bösen apps on each machine. Recompile Bösen using

```
cd bosen
make clean
make
```

Now build the YARN support libraries:

```
cd bosen/src/yarn
gradle build
```

Finally, recompile the apps you want to use via

```
cd bosen/app/path_to_app
make clean
make
```

The list of supported apps, as well as instructions on how to run them, can be found below.

Which applications are supported?

The following Bösen apps have YARN+HDFS support:

- *General-purpose Deep Neural Network (DNN)*
- *Non-negative Matrix Factorization (NMF)*
- *Sparse Coding*
- *Distance Metric Learning*
- *K-means Clustering*
- *Random Forest*
- *Multi-class Logistic Regression*

Please refer to the respective wiki pages for running instructions. **Note that the YARN launch scripts are different from the regular SSH launch scripts.**

YARN/HDFS support for Strads will be coming in a future update.

Troubleshooting

Running out of virtual memory when using YARN

If you are running out of virtual memory when launching PMLS apps via YARN, you may need to edit `$HADOOP_CONF_DIR/yarn-site.xml` in order to increase the maximum amount of virtual memory that can be allocated to containers. Search for the lines

```
<property>
<name>yarn.nodemanager.vmem-pmem-ratio</name>
<value>100</value>
</property>
```

and increase `value` to a ratio higher than 100.

Frequently asked questions

Q: How does PMLS differ from Spark and Hadoop?

A: PMLS uses bounded-asynchronous communication and fine-grained update scheduling to speed up ML algorithms, over bulk-synchronous execution (used in Spark and Hadoop).

These techniques take advantage of ML-specific properties such as error tolerance and dependency structures, in order to speed up ML algorithms while still maintaining correct execution guarantees.

PMLS is specifically designed for algorithms used in ML, such as optimization algorithms and sampling algorithms. It is not intended as a general platform for all types of Big Data algorithms.

Q: How does PMLS differ from other Key-Value stores or Parameter Servers?

A: PMLS has two major components: a dynamic scheduler system (Strads) for model-parallel ML algorithms, and a bounded-asynchronous key-value store (Bösen) for data-parallel ML algorithms.

The Bösen key-value store supports tunable ML-specific consistency models that guarantee correct execution even under worst-case conditions. Eventually-consistent key-value stores may not have ML execution guarantees.

Q: What kind of clusters is PMLS targeted at?

A: PMLS is designed to improve the speed and maximum problem size of ML algorithms, on moderate cluster or cloud compute sizes (1-100 machines). PMLS can also be run on desktops or laptops.

Latent Dirichlet Allocation (LDA)

After compiling Strads, we can test that it is working correctly with Latent Dirichlet Allocation, a popular algorithm for topic modeling of documents.

Introduction to LDA

Topic modeling, a.k.a Latent Dirichlet Allocation (LDA), is an algorithm that discovers latent semantic structure from documents. LDA finds global topics, which are weighted vocabularies, and the topical composition of each document in the collection.

PMLS's Strads LDA app uses a new model-parallel Gibbs sampling scheme described in this 2014 [NIPS paper](#), and implemented on top of the Strads scheduler. The documents are partitioned onto different machines, which take turns to sample disjoint subsets of words. By keeping the word subsets disjoint, our model-parallel implementation exhibits improved convergence times and memory utilization over data-parallel strategies. We use the sparse Gibbs sampling procedure in Yao et al (2009).

Performance

The Strads LDA app can train an LDA model with 1K topics, from a corpus with 8M documents and vocabulary size 140K, in 1000 seconds (17 minutes) using 25 machines (16 cores each).

Quick start

PMLS LDA uses the **Strads** scheduler, and can be found in `strads/apps/lda_release/`. **From this point on, all instructions will assume you are in `strads/apps/lda_release/`.** After building Strads (as explained under Installation), you may build the LDA app from `strads/apps/lda_release/` by running

```
make
```

Test the app (on your local machine) by running

```
./run.py
```

This will learn 1000 topics from a small subset of the NYtimes dataset, and output the word-topic and doc-topic tables to `tmplog/wt-mach-*` and `tmplog/dt-mach-*` respectively.

Input data format

The LDA app takes a single file as input, with the following format:

```
0 dummyword word-id word-id ...
1 dummyword word-id word-id ...
2 dummyword word-id word-id ...
...
```

Caution: the input file must use UNIX line endings. Windows or Mac line endings will cause a crash.

Each line represents a single document: the first item is the document ID (0-indexed), followed by any character string (represented by `dummyword`), and finally a list of tokens in the document, each represented by its word ID.

Output format

The LDA app outputs two types of files: word-topic tables `tmplog/wt-mach-*` and doc-topic tables `tmplog/dt-mach-*`. The word-topic tables use this format:

```
word-id, topic-id count, topic-id count, topic-id count ...
word-id, topic-id count, topic-id count, topic-id count ...
...
```

Each line contains, for a particular word `word-id`, all the topics `topic-id` that word is seen in, and the number of times `count` that word is seen in each topic.

The doc-topic tables follow an identical format:

```
doc-id, topic-id count, topic-id count, topic-id count ...
doc-id, topic-id count, topic-id count, topic-id count ...
...
```

The number of files `wt-mach-*` and `dt-mach-*` depends on the number of worker processes used — see the next section for more information.

Program options

The LDA app is launched using a python script, e.g. `run.py` used earlier:

```
#!/usr/bin/python
import os
import sys

datafile = ['./sampledata/nytimes_subset.id']
topics = [' 1000 ']
```



```

iterations = [' 3 ']
threads = [' 16 ']

machfile = ['./singlemach.vm']

prog = ['./bin/ldall ']
os.system("mpirun -machinefile "+machfile[0]+" "+prog[0]+" --machfile "+machfile[0]+"
↪-threads "+threads[0]+" -num_topic "+topics[0]+" -num_iter "+iterations[0]+" -data_
↪file "+datafile[0]+" -logfile tmplog/1 -wtfile_pre tmplog/wt -dtfile_pre tmplog/dt
↪");

```

The basic options are:

- `datafile`: Path to the data file, which must be visible to all machines. If using multiple machines, provide the full path to the data file.
- `topics`: How many topics to find.
- `iterations`: How many iterations to run.
- `threads`: How many threads to use for each Worker process.
- `machfile`: Strads machine file; see below for details.

Strads requires a machine file - `singlemach.vm` in the above example. Strads machine files control which machines house Workers, the Scheduler, and the Coordinator (the 3 architectural elements of Strads). In `singlemach.vm`, we spawn all element processes on the local machine `127.0.0.1`, so the file simply looks like this:

```

127.0.0.1
127.0.0.1
127.0.0.1
127.0.0.1

```

To prepare a multi-machine file, please refer to the Strads section under *Configuration Files for PMLS Apps*.

Maximum entropy discrimination latent Dirichlet allocation (MedLDA, [JMLR'12 paper](#)) is a supervised topic model that jointly learns classifier and latent topic representation from the text, by integrating max-margin principle to hierarchical Bayesian topic model. Here we provide multi-class MedLDA using Gibbs sampling algorithm described in the [KDD'13 paper](#).

MedLDA is built on top of the **Strads** scheduler system, but uses a data-parallel style. A similar (though more complicated) implementation can be found [here](#).

Performance

Using 20 machines (12 cores each), the Strads MedLDA app can solve for 1.1 million documents, 20 labels, 1000 topics in 5000 seconds.

Installation

The application can be found at `strads/apps/medlda_release/`. All subsequent operations are done under this directory:

```
cd strads/apps/medlda_release/
```

Assuming you have completed the Strads installation instructions,

```
make
```

will do the right job. The generated binary executable is located under `bin/`.

Data Preparation

This application takes inputs in LIBSVM format:

```
<label>...<word>:<count>...
```

For instance,

```
21 5 cogito:1 sum:1 ergo:1
```

represents a short document “Cogito ergo sum” labeled as 21 and 5. Note that `<word>` is considered as a string while `<label>` should be an integer in `[0, num_label)`.

We included a toy data set `20newsgroups` under the `app` directory for demo purpose. If you type

```
wc -l 20news.{train,test}
```

you’ll see the training file contains 11269 documents and the test file contains 7505 documents.

The data needs to be partitioned according to the number of workers: e.g. if you use 3 workers, you will need 3 partitions. To partition the data, run

```
python split.py <filename> <num_worker>
```

For instance,

```
python split.py 20news.train 3
python split.py 20news.test 3
```

randomly partitions the training and testing corpus into three subsets: `20news.train.0`, `20news.train.1`, `20news.train.2`, and `20news.test.0`, `20news.test.1`, `20news.test.2`.

If your cluster doesn’t support Network File System (NFS), don’t forget to `scp` or `rsync` files to the right host. The workers will determine which file to read according to the file suffix, e.g., `worker_0` reads `<filename>.0`.

If your cluster supports NFS or you only intend to do a single machine experiment, you’re ready to go.

Running MedLDA

Quick example on a single machine

We will train MedLDA on `20newsgroups` dataset on a single machine. The dataset and machine configuration files are provided in the package. It will spawn 3 servers and 3 workers, each worker running 2 threads. You can then execute:

```
python single.py
```

It will run the `medlda` binary using default flag settings. You will see outputs like:

```
.....
I1222 20:36:42.225085 2688 trainer.cpp:78] (rank:1) num train doc: 3756
I1222 20:36:42.225152 2688 trainer.cpp:79] (rank:1) num train word: 37723
I1222 20:36:42.225159 2688 trainer.cpp:80] (rank:1) num train token: 446496
I1222 20:36:42.225164 2688 trainer.cpp:81] (rank:1) -----
↪-----
I1222 20:36:42.236769 2689 trainer.cpp:78] (rank:2) num train doc: 3756
I1222 20:36:42.236814 2689 trainer.cpp:79] (rank:2) num train word: 37111
```

```

I1222 20:36:42.236821 2689 trainer.cpp:80] (rank:2) num train token: 426452
I1222 20:36:42.236827 2689 trainer.cpp:81] (rank:2) -----
↪-----
I1222 20:36:42.238376 2687 trainer.cpp:78] (rank:0) num train doc: 3757
I1222 20:36:42.238426 2687 trainer.cpp:79] (rank:0) num train word: 37572
I1222 20:36:42.238435 2687 trainer.cpp:80] (rank:0) num train token: 445351
I1222 20:36:42.238440 2687 trainer.cpp:81] (rank:0) -----
↪-----
.....
I1222 20:38:18.517712 2689 trainer.cpp:139] (rank:2) Burn-in Iteration 39 1.99362_
↪sec
I1222 20:38:18.517719 2688 trainer.cpp:139] (rank:1) Burn-in Iteration 39 1.99363_
↪sec
I1222 20:38:18.517725 2687 trainer.cpp:139] (rank:0) Burn-in Iteration 39 1.99362_
↪sec
I1222 20:38:20.451874 2687 trainer.cpp:139] (rank:0) Burn-in Iteration 40 1.93407_
↪sec
I1222 20:38:20.451865 2689 trainer.cpp:139] (rank:2) Burn-in Iteration 40 1.93407_
↪sec
I1222 20:38:20.451872 2688 trainer.cpp:139] (rank:1) Burn-in Iteration 40 1.93408_
↪sec
I1222 20:38:20.456595 2689 trainer.cpp:374] (rank:2) -----
↪-----
I1222 20:38:20.456612 2689 trainer.cpp:375] (rank:2) Elapsed time: 98.1856 sec _
↪Train Accuracy: 0.999734 (3755/3756)
I1222 20:38:20.456607 2687 trainer.cpp:374] (rank:0) -----
↪-----
I1222 20:38:20.456634 2689 trainer.cpp:378] (rank:2) -----
↪-----
I1222 20:38:20.456622 2687 trainer.cpp:375] (rank:0) Elapsed time: 98.1783 sec _
↪Train Accuracy: 0.999468 (3755/3757)
I1222 20:38:20.456643 2687 trainer.cpp:378] (rank:0) -----
↪-----
I1222 20:38:20.457993 2688 trainer.cpp:374] (rank:1) -----
↪-----
I1222 20:38:20.458014 2688 trainer.cpp:375] (rank:1) Elapsed time: 98.1953 sec _
↪Train Accuracy: 0.999734 (3755/3756)
I1222 20:38:20.458036 2688 trainer.cpp:378] (rank:1) -----
↪-----
.....
I1222 20:38:30.521900 2688 trainer.cpp:398] (rank:1) Train prediction written into /
↪tmp/dump_train_pred.1
I1222 20:38:30.526638 2689 trainer.cpp:398] (rank:2) Train prediction written into /
↪tmp/dump_train_pred.2
I1222 20:38:30.592419 2687 trainer.cpp:398] (rank:0) Train prediction written into /
↪tmp/dump_train_pred.0
I1222 20:38:31.044430 2687 trainer.cpp:403] (rank:0) Train doc stats written into /
↪tmp/dump_train_doc.0
I1222 20:38:31.076773 2689 trainer.cpp:403] (rank:2) Train doc stats written into /
↪tmp/dump_train_doc.2
I1222 20:38:31.213727 2688 trainer.cpp:403] (rank:1) Train doc stats written into /
↪tmp/dump_train_doc.1
Rank (1) Ready to exit program from main function in ldall.cpp
I1222 20:38:31.256194 2687 trainer.cpp:449] (rank:0) Hyperparams written into /tmp/
↪dump_param
I1222 20:38:31.259068 2687 trainer.cpp:454] (rank:0) Classifier written into /tmp/
↪dump_classifier
Rank (2) Ready to exit program from main function in ldall.cpp

```

```

I1222 20:38:31.271615 2687 trainer.cpp:464] (rank:0) Dict written into /tmp/dump_dict
I1222 20:38:31.271632 2687 trainer.cpp:465] (rank:0) Total num of words: 53485
I1222 20:38:46.930896 2687 trainer.cpp:487] (rank:0) Model written into /tmp/dump_
↪model
Rank (0) Ready to exit program from main function in ldall.cpp

```

Once all workers have reported Ready to exit program, you may Ctrl-c to terminate the program.

As the last few lines suggest, the training results will be stored at /tmp/dump_* by default. Specifically, (let D = num of docs in a partition, L = num of labels, and K = num of topics)

- `_train_pred.x` stores the predicted label of partition x . ($D \times 1$ integer vector)
- `_train_doc.x` stores the doc-topic distribution in log scale. ($D \times K$ matrix)
- `_param` stores the value of alpha, beta, num_topic, and num_label.
- `_classifier` stores the classifier weights. ($K \times L$ matrix, each column is a binary classifier)
- `_dict` stores the aggregated distinct words appeared in the train corpus.
- `_model` stores the topic-word distribution in log scale. ($K \times V$ matrix)

Now we're ready for test. You can run

```
python single_test.py
```

It will load the model files generated at the training phase and perform inference on test documents. You will see outputs like:

```

.....
I1222 20:39:30.173037 3258 tester.cpp:24] (rank:1) Hyperparams loaded from /tmp/dump_
↪param
I1222 20:39:30.173049 3259 tester.cpp:24] (rank:2) Hyperparams loaded from /tmp/dump_
↪param
I1222 20:39:30.173061 3258 tester.cpp:25] (rank:1) Alpha: 0.16 Beta: 0.01 Num Topic: 40
↪40 Num Label: 20
I1222 20:39:30.173069 3259 tester.cpp:25] (rank:2) Alpha: 0.16 Beta: 0.01 Num Topic: 40
↪40 Num Label: 20
I1222 20:39:30.173780 3257 tester.cpp:31] (rank:0) Classifier loaded from /tmp/dump_
↪classifier
I1222 20:39:30.176692 3258 tester.cpp:31] (rank:1) Classifier loaded from /tmp/dump_
↪classifier
I1222 20:39:30.176772 3259 tester.cpp:31] (rank:2) Classifier loaded from /tmp/dump_
↪classifier
I1222 20:39:30.213495 3257 tester.cpp:44] (rank:0) Dict loaded from /tmp/dump_dict
I1222 20:39:30.213523 3257 tester.cpp:45] (rank:0) Total num of words: 53485
I1222 20:39:30.228713 3259 tester.cpp:44] (rank:2) Dict loaded from /tmp/dump_dict
I1222 20:39:30.228745 3259 tester.cpp:45] (rank:2) Total num of words: 53485
I1222 20:39:30.235925 3258 tester.cpp:44] (rank:1) Dict loaded from /tmp/dump_dict
I1222 20:39:30.235959 3258 tester.cpp:45] (rank:1) Total num of words: 53485
I1222 20:39:31.196068 3258 tester.cpp:51] (rank:1) Model loaded into /tmp/dump_model
I1222 20:39:31.259271 3259 tester.cpp:51] (rank:2) Model loaded into /tmp/dump_model
I1222 20:39:31.306517 3258 tester.cpp:95] (rank:1) num test doc: 2502
I1222 20:39:31.306558 3258 tester.cpp:96] (rank:1) num test oov: 8212
I1222 20:39:31.306565 3258 tester.cpp:97] (rank:1) num test token: 300103
I1222 20:39:31.306571 3258 tester.cpp:98] (rank:1) -----
↪-----
I1222 20:39:31.348415 3259 tester.cpp:95] (rank:2) num test doc: 2501
I1222 20:39:31.348460 3259 tester.cpp:96] (rank:2) num test oov: 8227
I1222 20:39:31.348469 3259 tester.cpp:97] (rank:2) num test token: 292610

```

```

I1222 20:39:31.348476 3259 tester.cpp:98] (rank:2) -----
↪-----
I1222 20:39:31.350323 3257 tester.cpp:51] (rank:0) Model loaded into /tmp/dump_model
I1222 20:39:31.420966 3257 tester.cpp:95] (rank:0) num test doc: 2502
I1222 20:39:31.421005 3257 tester.cpp:96] (rank:0) num test oov: 7172
I1222 20:39:31.421013 3257 tester.cpp:97] (rank:0) num test token: 267530
I1222 20:39:31.421018 3257 tester.cpp:98] (rank:0) -----
↪-----
I1222 20:39:35.194511 3257 tester.cpp:118] (rank:0) Elapsed time: 3.77297 sec ↵
↪Test Accuracy: 0.797362 (1995/2502)
I1222 20:39:35.206197 3257 tester.cpp:212] (rank:0) Test prediction written into /
↪tmp/dump_test_pred.0
I1222 20:39:35.307680 3259 tester.cpp:118] (rank:2) Elapsed time: 3.95875 sec ↵
↪Test Accuracy: 0.822471 (2057/2501)
I1222 20:39:35.315475 3259 tester.cpp:212] (rank:2) Test prediction written into /
↪tmp/dump_test_pred.2
I1222 20:39:35.392273 3258 tester.cpp:118] (rank:1) Elapsed time: 4.08543 sec ↵
↪Test Accuracy: 0.804956 (2014/2502)
I1222 20:39:35.404650 3258 tester.cpp:212] (rank:1) Test prediction written into /
↪tmp/dump_test_pred.1
I1222 20:39:35.549335 3257 tester.cpp:217] (rank:0) Test doc stats written into /tmp/
↪dump_test_doc.0
Rank (0) Ready for exit program from main function in ldall.cpp
I1222 20:39:35.647891 3259 tester.cpp:217] (rank:2) Test doc stats written into /tmp/
↪dump_test_doc.2
Rank (2) Ready for exit program from main function in ldall.cpp
I1222 20:39:35.724900 3258 tester.cpp:217] (rank:1) Test doc stats written into /tmp/
↪dump_test_doc.1
Rank (1) Ready for exit program from main function in ldall.cpp

```

Once all workers have reported Ready to exit program, you may Ctrl-c to terminate the program.

We're done! Similar to the training phase, prediction results for test documents are stored at /tmp/dump_test_doc.x and /tmp/dump_test_pred.x.

Configuration and using multiple machines

Let us inspect the training script `single.py`:

```

#!/usr/bin/python
import os
import sys

machfile = ['./singlemach.vm']
traindata = ['./20news.train']
numservers = ['3']

prog=['./bin/medlda ']
os.system("mpirun -machinefile "+machfile[0]+" "+prog[0]+" -machfile "+machfile[0]+" -
↪schedulers "+numservers[0]+" -train_prefix "+traindata[0]);

```

Things to note:

- The last line `os.system` executes the MedLDA app; you may insert advanced command line flags here. See the end of this article for a full list of flags.
- `machfile` gives the machine configuration file.

- `traindata` gives the dataset prefix (`20news.train` in this case).
- `numservers` is the number of servers (key-value stores) to use.

MedLDA is built upon the Strads scheduler architecture, and uses a similar machine configuration file. This machine file is a simple list of IP addresses, corresponding to workers, followed by servers, and finally the Strads coordinator. For example, the `singlemach.vm` machine file used in `single.py` looks like this:

```
127.0.0.1      <--- this is worker 0
127.0.0.1      <--- this is worker 1
127.0.0.1      <--- this is worker 2
127.0.0.1      <--- this is server 0
127.0.0.1      <--- this is server 1
127.0.0.1      <--- this is server 2
127.0.0.1      <--- this is the coordinator
```

The last IP is always the coordinator, and the servers come immediately before it (`numservers` controls the number of servers). The workers make up the remaining IPs (you must use at least 2 workers).

Note: remember to partition your data for the correct number of workers.

To use multiple machines, simply change the machine file IPs to point to the desired machines. **You may repeat IP addresses to assign multiple processes to the same machine, but the repeat IPs must be contiguous - `ip1` followed by `ip2` followed by `ip1` is invalid.**

Important: do not forget to prepare the test script (e.g. `single_test.py`) in the same fashion. You must use the same machine configuration as the training script.

Command line flags

Flags for training:

- `train_prefix`: Prefix to the LIBSVM format training corpus.
- `dump_prefix`: Prefix to the dump results.
- `num_thread`: Number of worker threads.
- `alpha`: Parameter of Dirichlet prior on doc-topic distribution.
- `beta`: Parameter of Dirichlet prior on topic-word distribution.
- `cost`: Cost parameter on hinge loss, usually called “C”.
- `ell`: Margin parameter in SVM, usually set to 1. Hinge loss = $\max(0, \text{ell} - y < w, x >)$.
- `num_burnin`: Number of burn-in iterations.
- `num_topic`: Number of topics, usually called “K”.
- `num_label`: Total number of labels.
- `eval_interval`: Print out information every N iterations

Flags for testing:

- `test_prefix`: Prefix to LIBSVM format test corpus.
- `dump_prefix`: Prefix to the dump results.
- `num_thread`: Number of worker threads.

Deep Neural Network

This app implements a fully-connected Deep Neural Network (DNN) for multi-class classification, on **Bösen**. The DNN consists of an input layer, arbitrary number of hidden layers and an output layer. Each layer contains a certain amount of neuron units. Each unit in the input layer corresponds to an element in the feature vector. We represent the class label using 1-of-K coding and each unit in the output layer corresponds to a class label. The number of hidden layers and the number of units in each hidden layer are configured by the users. Units between adjacent layers are fully connected. In terms of DNN learning, we use the cross entropy loss and stochastic gradient descent where the gradient is computed using backpropagation method.

Performance

On a dataset with 1M data instances; 360 feature dimension; 24M parameters, the Bösen DNN implementation converges in approximately 45 minutes, using 6 machines (16 cores each).

Going from 1 to 6 machines results in a speedup of roughly 4.1x.

Quick start

The DNN app can be found in `bosen/app/dnn`. From this point on, all instructions will assume you are in `bosen/app/dnn`. After building PMLS (as explained earlier in this manual), you can build the DNN app from `bosen/app/dnn` by running

```
make
```

This will put the DNN binary in the subdirectory `bin/`.

Create a simulated dataset:

```
script/gen_data.sh 10000 360 2001 1 datasets
```

Change `app_dir` in `script/run_local.py`; see example below. You need to provide the full path to the DNN directory. This will allow for correct YARN operation (if required).

```
app_dir = "/home/user/bosen/app/dnn"
```

then you can test that the app works on the local machine (with 4 worker threads). From the `bosen/app/dnn` directory, run:

```
./script/launch.py
```

The DNN app runs in the background (progress is output to stdout). After the app terminates, you should get 2 output files:

```
datasets/weights.txt  
datasets/biases.txt
```

`weights.txt` saves the weight matrices. The order is: weight matrix between layer 1 (input layer) and layer 2 (the first hidden layer), weight matrix between layer 2 and layer 3, etc. All matrices are saved in row major order and each line corresponds to a row.

Making predictions

Change `app_dir` in `script/predict.py`; see example below. You need to provide the full path to the DNN directory.

```
app_dir = "/home/user/bosen/app/dnn"
```

You can now make predictions with the learned model using

```
./script/launch_pred.py
```

After the app terminates, you should get a prediction result file for each of the input data file. Please check the directory where you put the data files. Each line of the prediction result file contains the prediction for the corresponding data.

Input data format

We assume users have partitioned the data into M pieces, where M is the total number of clients (machines). Each client will be in charge of one piece. User needs to provide a file recording the data partition information. In this file, each line corresponds to one data partition. The format of each line is

```
<data_file> \t <num_data_in_partition>
```

`<num_data_in_partition>` is the number of data points in this partition. `<data_file>` stores the class label and feature vector of a data sample in each line. `<data_file>` must be an absolute path. The format of each line in `<data_file>` is:

```
<class_label> \t <feature vector>
```

Elements in the feature vector are separated with single blank. Note that class label starts from 0. If there are K classes, the range of class labels are $[0, 1, \dots, K-1]$.

Creating synthetic data

We provide a synthetic data generator for testing purposes. The generator generates random data and automatically partitions the data into different files. To see detailed instructions, run

```
scripts/gen_data.sh
```

The basic syntax is

```
scripts/gen_data.sh <num_train_data> <dim_feature> <num_classes> <num_partitions>
↪<save_dir>
```

- `<num_train_data>`: number of training data
- `<dim_feature>`: dimension of features
- `<num_classes>`: number of classes
- `<num_partitions>`: how many pieces the data is going to be partitioned into
- `<save_dir>`: the directory where the generated data will be saved

The generator will generate an amount of `<num_partitions>` files storing class labels and feature vectors. The data files are automatically named as `0_data.txt`, `1_data.txt`, etc. The generator will also generate a file `data_ptt_file.txt` where each line stores the data file of one data partition and the number of points in this partition.

For example, you can create a dataset by running the following command:

```
scripts/gen_data.sh 10000 360 2001 3 datasets
```

It will generate a dataset where the number of training examples is 10000, feature dimension is 360 and number of classes are 2001. Moreover, it will partition the dataset into 3 pieces and save the generated files to `datasets/`: `0_data.txt`, `1_data.txt`, `2_data.txt` and a `data_ptt_file.txt` file which will have three lines. Example:

```
/home/user/bosen/app/dnn/datasets/0_data.txt 3333
/home/user/bosen/app/dnn/datasets/1_data.txt 3333
/home/user/bosen/app/dnn/datasets/2_data.txt 3334
```

Each line corresponds to one data partition and contains the data file of this partition and number of data points in this partition. Note that the path of data file must be an absolute path.

Running the Deep Neural Network Application

Parameters of the DNN training are specified in `script/run_local.py`, including:

- `<num_worker_threads>`: how many worker threads to use in each machine
- `<staleness>`: staleness value
- `<parafire>`: configuration file on DNN parameters
- `<data_ptt_file>`: a file containing the data file path and the number of training points in each data partition
- `<model_weight_file>`: the path where the output weight matrices will be stored
- `<model_bias_file>`: the path where the output bias vectors will be stored

The machine file is specified in `script/launch.py`:

```
hostfile_name = "machinefiles/localserver"
```

After configuring these parameters, perform DNN training by:

```
./script/launch.py
```

Parameters of the DNN prediction are specified in `script/predict.py`, including:

- `<parafiler>`: configuration file on DNN parameters
- `<data_ptt_file>`: a file containing the data file path and the number of training points in each data partition
- `<model_weight_file>`: the file storing the weight matrices
- `<model_bias_file>`: the file storing the bias vectors

The machine file is specified in `script/launch_pred.py`:

```
hostfile_name = "machinefiles/localserver"
```

After configuring these parameters, perform prediction by:

```
./script/launch_pred.py
```

Format of DNN Configuration File

The DNN configurations are stored in `<parameter_file>`. Each line corresponds to a parameter and its format is

```
<parameter_name>: <parameter_value>
```

`<parameter_name>` is the name of the parameter. It is followed by a `:` (there is no blank between `<parameter_name>` and `:`). `<parameter_value>` is the value of this parameter. Note that `:` and `<parameter_value>` must be separated by a blank.

The list of parameters and their meanings are:

- `num_layers`: number of layers, including input layer, hidden layers, and output layer
- `num_units_in_each_layer`: number of units in each layer
- `num_epochs`: number of epochs in stochastic gradient descent training
- `stepsize`: learn rate of stochastic gradient descent
- `mini_batch_size`: mini batch size in each iteration
- `num_smp_evaluate`: when evaluating the objective function, we randomly sample `<num_smp_evaluate>` points to compute the objective -`num_iters_evaluate`: every `<num_iters_evaluate>` iterations, we do an objective function evaluation Note that, the order of the parameters cannot be switched.

Here is an example:

```
num_layers: 6
num_units_in_each_layer: 360 512 512 512 512 2001
num_epochs: 2
stepsize: 0.05
mini_batch_size: 10
num_smp_evaluate: 2000
num_iters_evaluate: 100
```

Terminating the DNN app

The DNN app runs in the background, and outputs its progress to stdout. If you need to terminate the app before it finishes, run

```
./script/kill.py <petuum_ps_hostfile>
```

File IO from HDFS

Put datasets to HDFS

```
hadoop fs -mkdir -p /user/bosen/dataset/dnn
hadoop fs -put datasets /user/bosen/dataset/dnn
```

Change the corresponding file paths in `script/run_local.py` to the right HDFS path. Comment out the local path.

```
# , "parafile": join(app_dir, "datasets/para_imnet.txt")
, "parafile": "hdfs://hdfs-domain/user/bosen/dataset/dnn/datasets/para_imnet.txt"
# , "data_ptt_file": join(app_dir, "datasets/data_ptt_file.txt")
, "data_ptt_file": "hdfs://hdfs-domain/user/bosen/dataset/dnn/datasets/data_ptt_file.
↳txt"
# , "model_weight_file": join(app_dir, "datasets/weights.txt")
, "model_weight_file": "hdfs://hdfs-domain/user/bosen/dataset/dnn/datasets/weights.txt
↳"
# , "model_bias_file": join(app_dir, "datasets/biases.txt")
, "model_bias_file": "hdfs://hdfs-domain/user/bosen/dataset/dnn/datasets/biases.txt"
```

Launch it over ssh

```
./script/launch.py
```

Check the output

```
hadoop fs -cat /user/bosen/dataset/dnn/datasets/biases.txt
```

Similar configurations apply to DNN prediction.

Use Yarn to launch DNN app

```
./script/launch_on_yarn.py
```

Deep Neural Network for Speech Recognition

This tutorial shows how the Deep Neural Network (DNN) application (implemented on **Bösen**) can be applied to speech recognition, using Kaldi (<http://kaldi.sourceforge.net/about.html>) as our tool for feature extraction and decoding. Kaldi is a toolkit for speech recognition written in C++ and licensed under the Apache License v2.0. It also provides two DNN applications (<http://kaldi.sourceforge.net/dnn.html>), and we follow Dan's setting in feature extraction, preprocessing and decoding.

Our DNN consists of an input layer, arbitrary number of hidden layers and an output layer. Each layer contains a certain amount of neuron units. Each unit in the input layer corresponds to an element in the feature vector. We represent the class label using 1-of-K coding and each unit in the output layer corresponds to a class label. The number of hidden layers and the number of units in each hidden layer are configured by the users. Units between adjacent layers are fully connected. In terms of DNN learning, we use the cross entropy loss and stochastic gradient descent where the gradient is computed using backpropagation method.

Installation

PMLS Deep Neural Network Application

The DNN for Speech Recognition app can be found in `bosen/app/dnn_speech/`. From this point on, all instructions will assume you are in `bosen/app/dnn_speech/`. After building PMLS (as explained earlier in this manual), you can build the DNN from `bosen/app/dnn_speech/` by running

```
make
```

This will put the DNN binary in the subdirectory `bin/`.

Kaldi

From `bosen/app/dnn_speech/`, extract Kaldi by running:

```
tar -xvf kaldi-trunk.tar.gz
cd kaldi-trunk/tools
```

Next, we must build the ATLAS libraries in a local directory. From `kaldi-trunk/tools`,

```
sudo apt-get install gfortran
./install_atlas.sh
```

This process will take some time, and will report some messages of the form `Error 1 (ignored)` - this is normal. More details can be found in `kaldi-trunk/INSTALL`, `kaldi-trunk/tools/INSTALL` and `kaldi-trunk/src/INSTALL`.

Once ATLAS has been set up,

```
make
cd ../src/
./configure
make depend
make
```

The first `make` may produce some “Error 1 (ignored)” messages, this is normal and can be ignored. The `./configure` may produce a warning about GCC 4.8.2, which can also be ignored for our purposes. Be advised that these steps will take a while (up to 1-2 hours).

The Whole Pipeline

Currently, we only support TIMIT dataset (<https://catalog.ldc.upenn.edu/LDC93S1>), a well-known benchmark dataset for speech recognition. You can process this dataset through following steps.

1. Feature extraction

WARNING: this stage will take several hours, and requires at least 16GB free RAM.

Run

```
sh scripts/PrepDNNFeature.sh <TIMIT_path>
```

where `<TIMIT_path>` is the **absolute** path to the TIMIT directory. This will extract features and do some pre-processing work to generate `Train.fea`, `Train.label`, `Train.para`, `head.txt` and `tail.txt` in `app/dnn_speech` directory and `exp/petuum_dnn` in `kaldi-trunk/egs/timit/s5` directory. The script will take 1-2 hours to complete.

- `<TIMIT_path>` is the absolute path of TIMIT dataset. You can get it through <https://catalog.ldc.upenn.edu/LDC93S1>.
- `Train.fea` and `Train.label` save the feature and label of training examples, one example per line.
- `Train.para` saves the information of training examples, including the feature dimension, the number of classes of labels and the number of examples.
- `head.txt` contains the transition model and the preprocessing information of training features, including splicing and linear discriminant analysis (LDA), saved in the format of Dan’s setup in Kaldi.
- `tail.txt` contains the empirical distribution of the classes of labels, saved in the format of Dan’s setup in Kaldi.

- `kaldi-trunk/egs/timit/s5/exp/petuum_dnn` contains all the log file and intermediate results.

2. DNN Training

According to the information in `Train para`, you can set the configuration file for DNN (more details can be found in `Input data format and Format of DNN Configuration file` section). For example, if `Train para` is

```
360 2001 1031950
```

You can set `datasets/data_partition.txt` by

```
/home/user/bosen/app/dnn_speech/Train 1031950
```

and `datasets/para_imnet.txt` by

```
num_layers: 4
num_units_in_each_layer: 360 512 512 2001
num_epochs: 2
stepsize: 0.1
mini_batch_size: 256
num_smp_evaluate: 2000
num_iters_evaluate: 100
```

And run

```
scripts/run_dnn.sh 4 5 machinefiles/localserver datasets/para_imnet.txt datasets/data_
↪partition.txt DNN_para.txt
```

The DNN app runs in the background (progress is output to stdout). After the app terminates, you should get 1 output files:

```
DNN_para.txt
```

which stores weight matrices and bias vectors as Dan's setup in Kaldi. More details can be found in next section.

3. Decoding

Run

```
scripts/NetworkDecode.sh DNN_para.txt datasets/para_imnet.txt
```

Wait for several minutes and you will see the decode result over the core test dataset of TIMIT.

Running the Deep Neural Network application

Notice that the interface of DNN for speech recognition is slightly different from the general purpose DNN in `app/dnn`. To see the instructions for the DNN for speech app, run

```
scripts/run_dnn.sh
```

The basic syntax is

```
scripts/run_dnn.sh <num_worker_threads> <staleness> <hostfile> <parameter_file>
↪<data_partition_file> <model_para_file> "additional options"
```

- `<num_worker_threads>`: how many worker threads to use in each machine
- `<staleness>`: staleness value
- `<hostfile>`: machine configuration file
- `<parameter_file>`: configuration file on DNN parameters
- `<data_partition_file>`: a file containing the data file path and the number of training points in each data partition
- `<model_para_file>`: the path where the output weight matrices and bias vector will be stored

The final argument, “additional options”, is an optional quote-enclosed string of the form “`--opt1 x --opt2 y ...`” (you may omit this if you wish). This is used to pass in the following optional arguments:

- `ps_snapshot_clock x`: take snapshots every `x` iterations
- `ps_snapshot_dir x`: save snapshots to directory `x` (please make sure `x` already exists!)
- `ps_resume_clock x`: if specified, resume from iteration `x` (note: if `-staleness s` is specified, then we resume from iteration `x-s` instead)
- `ps_resume_dir x`: resume from snapshots in directory `x`. You can continue to take snapshots by specifying
- `ps_snapshot_dir y`, but do make sure directory `y` is not the same as `x`!

For example, to run the DNN app on local machine (one client) where the number of worker thread is 4, staleness is 5, machine file is `machinefiles/localserver`, DNN configuration file is `datasets/para_imnet.txt`, data partition file is `datasets/data_partition.txt`, model parameter file is `DNN_para.txt`, use the following command:

```
scripts/run_dnn.sh 4 5 machinefiles/localserver datasets/para_imnet.txt datasets/data_
↪partition.txt DNN_para.txt
```

Input data format

We assume users have partitioned the data into `M` pieces, where `M` is the total number of clients (machines). Each client will be in charge of one piece. User needs to provide a file recording the data partition information. In this file, each line corresponds to one data partition. The format of each line is

```
<data_file> \t <num_data_in_partition>
```

`<num_data_in_partition>` is the number of data points in this partition. `<data_file>` is the prefix of the class label file (`<data_file>.label`) and the feature file (`<data_file>.fea`). And `<data_file>` must be an absolute path.

For example,

```
/home/user/bosen/app/dnn_speech/Train 1031950
```

means there are 1031950 training examples, the class label file is `/home/user/bosen/app/dnn_speech/Train.label` and the feature file is `/home/user/bosen/app/dnn_speech/Train.fea`.

The format of `<data_file>.fea` is:

```
<feature vector 1>
<feature vector 2>
...
```

Elements in the feature vector are separated with single blank.

The format of `<data_file>.label` is

```
<label 1>
<label 2>
...
```

Note that class label starts from 0. If there are K classes, the range of class labels are [0,1,...,K-1].

Format of DNN Configuration File

The DNN configurations are stored in `<parameter_file>`. Each line corresponds to a parameter and its format is

```
<parameter_name>: <parameter_value>
```

`<parameter_name>` is the name of the parameter. It is followed by a `:` (there is no blank between `<parameter_name>` and `:`). `<parameter_value>` is the value of this parameter. Note that `:` and `<parameter_value>` must be separated by a blank.

The list of parameters and their meanings are:

- `num_layers`: number of layers, including input layer, hidden layers, and output layer
- `num_units_in_each_layer`: number of units in each layer
- `num_epochs`: number of epochs in stochastic gradient descent training
- `stepsize`: learn rate of stochastic gradient descent
- `mini_batch_size`: mini batch size in each iteration
- `num_smp_evaluate`: when evaluating the objective function, we randomly sample `<num_smp_evaluate>` points to compute the objective
- `num_iters_evaluate`: every `<num_iters_evaluate>` iterations, we do an objective function evaluation. Note that, the order of the parameters cannot be switched.

Here is an example:

```
num_layers: 4
num_units_in_each_layer: 360 512 512 2001
num_epochs: 2
stepsize: 0.1
mini_batch_size: 256
num_smp_evaluate: 2000
num_iters_evaluate: 100
```

Output format

The DNN app outputs just one file:

```
<model_para_file>
```

<model_para_file> saves the weight matrices and bias vectors. The order is: weight matrix between layer 1 (input layer) and layer 2 (the first hidden layer), bias vector for layer 2, weight matrix between layer 2 and layer 3, bias vector for layer 3, etc. All matrices are saved in row major order and each line corresponds to a row. Elements in each row are separated with blank.

Terminating the DNN app

The DNN app runs in the background, and outputs its progress to stdout. If you need to terminate the app before it finishes, for distributed version, run

```
scripts/kill_dnn.sh <hostfile>
```

Matrix Factorization

Given an input matrix A (with some missing entries), MF learns two matrices W and H such that $W \cdot H$ approximately equals A (except where elements of A are missing). If A is N -by- M , then W will be N -by- K and H will be K -by- M . Here, K is a user-supplied parameter (the “rank”) that controls the accuracy of the factorization. Higher values of K usually yield a more accurate factorization, but require more computation.

MF is commonly used to perform Collaborative Filtering, where A represents the known relationships between two categories of things - for example, $A(i, j) = v$ might mean that “person i gave product j rating v ”. If some relationships $A(i, j)$ are missing, we can use the learnt matrices W and H to predict them:

$$A(i, j) = W(i, 1) \cdot H(1, j) + W(i, 2) \cdot H(2, j) + \dots + W(i, K) \cdot H(K, j)$$

The PMLS MF app uses a model-parallel coordinate descent scheme, implemented on the Strads scheduler. If you would like to use the older Bösen-based PMLS MF app, you may obtain it from the [PMLS v0.93 release](#).

Performance

The Strads MF app finishes training on the Netflix dataset (480k by 20k matrix) with rank=40 in 2 minutes, using 25 machines (16 cores each).

Quick start

PMLS MF uses the Strads scheduler, and can be found in `src/strads/apps/cdmf_release/`. **From this point on, all instructions will assume you are in `src/strads/apps/cdmf_release/`.** After building the main PMLS libraries (as explained under Installation), you may build the MF app from `src/strads/apps/cdmf_release/` by running

```
make
```

Test the app (on your local machine) by running

```
./run.py
```

This will perform a rank $K=40$ decomposition on a synthetic 10k-by-10k matrix, and output the factors W and H to `tmplog/wfile-mach-*` and `tmplog/hfile-mach-*` respectively.

Input data format

The MF app uses the [MatrixMarket](#) format:

```
%%MatrixMarket matrix coordinate real general
num_rows num_cols num_nonzeros
row col value
row col value
row col value
```

The first line is the MatrixMarket header, and should be copied as-is. The second line gives the number of rows N , columns M , and non-zero entries in the matrix. This is followed by `num_nonzeros` lines, each representing a single matrix entry $A(\text{row}, \text{col}) = \text{value}$ (where `row` and `col` are 0-indexed).

Output format

The MF app outputs W and H to `tmplog/wfile-mach-*` and `tmplog/hfile-mach-*` respectively. The W files have the following format:

```
row-id: value-0 value-1 ... value-(K-1)
row-id: value-0 value-1 ... value-(K-1)
...
```

Each line represents one row in W , beginning with the row index `row-id`, and followed by all K values that make up the row.

The H files follow a similar format:

```
col-id: value-0 value-1 ... value-(K-1)
col-id: value-0 value-1 ... value-(K-1)
...
```

The number of files `wfile-mach-*` and `hfile-mach-*` depends on the number of worker processes used — see the next section for more information.

Program options

The MF app is launched using a python script, e.g. `run.py` used earlier:

```
#!/usr/bin/python
import os
import sys

datafile = ['./sampledata/mftest.mmt ']
threads = [' 16 ']
rank = [' 40 ']
```

```

iterations = [' 10 ']
lambda_param = [' 0.05 ']

machfile = ['./singlemach.vm']

prog = ['./bin/lccdmf ']
os.system("mpirun -machinefile "+machfile[0]+" "+prog[0]+" --machfile "+machfile[0]+"
↪-threads "+threads[0]+" -num_rank "+rank[0]+" -num_iter "+iterations[0]+" -lambda
↪"+lambda_param[0]+" -data_file "+datafile[0]+" -wfile_pre tmplog/wfile -hfile_pre_
↪tmplog/hfile");

```

The basic options are:

- `datafile`: Path to the data file, which must be present/visible to all machines. We strongly recommend providing the full path name to the data file.
- `threads`: How many threads to use for each worker.
- `rank`: The desired decomposition rank K .
- `iterations`: How many iterations to run.
- `lambda_param`: Regularization parameter (PMLS MF uses an L2 regularizer)
- `machfile`: Strads machine file; see below for details.

Strads requires a machine file - `singlemach.vm` in the above example. Strads machine files control which machines house Workers, the Scheduler, and the Coordinator (the 3 architectural elements of Strads). In `singlemach.vm`, we spawn all element processes on the local machine `127.0.0.1`, so the file simply looks like this:

```

127.0.0.1
127.0.0.1
127.0.0.1
127.0.0.1

```

To prepare a multi-machine file, please refer to the Strads section under *Configuration Files for PMLS Apps*.

Non-negative Matrix Factorization (NMF)

After compiling Bösen, we can test that it is working correctly with Non-negative Matrix Factorization, a popular algorithm for recommender systems.

Introduction to NMF

Given an input matrix X , the NMF app on **Bösen** learns two non-negative matrices L and R such that $L \cdot R$ is approximately equal to X .

If X is N -by- M , then L will be N -by- K and R will be K -by- M where N is the number of data points, M is the dimension of the data, K is a user-supplied parameter that controls the rank of the factorization. The objective of NMF app is to minimize the function $\|X - L \cdot R\|^2$ subject to the constraint that the elements in matrices are non-negative.

Our NMF app uses the projected Stochastic Gradient Descent (SGD) algorithm to learn the two matrices L and R .

Quick Start

The NMF app can be found in `bosen/app/NMF/`. From this point on, all instructions will assume you are in `bosen/app/NMF/`. After building PMLS (as explained earlier in this manual), you can build the NMF app from `bosen/app/NMF` by running

```
make
```

This will put the NMF binary in the subdirectory `bin/`.

Create directory

```
mkdir sample
mkdir sample/data
mkdir sample/output
```

Create a simulated dataset:

```
./script/make_synth_data.py 3 3 sample/data/sample.txt
```

Change `app_dir` in `script/run_local.py`; see example below. You need to provide the full path to the NMF directory. This will allow for correct YARN operation (if required).

```
app_dir = "/home/user/bosen/app/NMF"
```

then you can test that the app works on the local machine (with 4 worker threads). From the `bosen/app/NMF/` directory, run:

```
./script/launch.py
```

The NMF app runs in the background (progress is output to stdout). After a few minutes, you should get the following output files in the subdirectory `sample/output/`:

```
L.txt.0
R.txt
loss.txt
time.txt
```

The file `L.txt.0` and the file `R.txt` are the optimization results of two matrices `L` and `R`, which looks something like this (note that there exist many valid solutions; the NMF app produces one at random):

```
0.80641 0 0
0.80641 0 0
0.80641 0 0
0 0 1.47865
0 0 1.47865
0 0 1.47865
0 2.2296 0
0 2.2296 0
0 2.2296 0
```

```
1.24006 1.24006 1.24006 0 0 0 0 0 0
0 0 0 0 0 0 1.34553 1.34553 1.34553
0 0 0 1.35258 1.35258 1.35258 0 0 0
```

The file `loss.txt` contains the statistics of loss function evaluated at iterations specified by user. If there are `num_client` machines in the experiment, it will have `num_client` columns, and the *i*-th column is the loss function evaluated by the *i*-th machine. Note that loss function is evaluated by averaging loss function at randomly sampled data points. The file `time.txt` also has `num_client` columns, and contains the time (seconds) taken between evaluations on each client (Evaluating time is excluded). Don't be surprised if the first row of `time.txt` is nearly 0, that's because the loss function is evaluated once before any optimization is conducted. Also note that at the end of `time.txt` and `loss.txt` there might be some N/As, that's invalid and is due to the fact that different clients may evaluate different times.

You won't see the exact same numeric values in different runs as there are multiple optimums of the optimization problem. All you need to confirm is that in optimized `L`, lines 1-3, lines 4-6 and line 7-9 shall be similar. Similarly, in optimized `R`, columns 1-3, 4-6, 7-9 shall be similar.

Data format

The NMF app takes a dense matrix `X` as input, where each row corresponds to a data sample and each column corresponds to a feature. The file containing `X` should be a binary or text file, and elements are sorted by rows then columns:

```

ele_0_0      ele_0_1      ... ele_0_n-1
ele_1_0      ele_1_1      ... ele_1_n-1
...
ele_n-1_0    ele_n-1_1    ... ele_n-1_n-1

```

You must specify the format of file (binary or text) by using the parameter `file_format`, which will be explained in *Running the NMF application* section.

Creating synthetic data

We provide a synthetic data generator for testing purposes (we have tested it on python 2.7). To see detailed instructions, run

```
python script/make_synth_data.py
```

Running the NMF application

There are many parameters involved for running NMF. We provide default values for all of them in `script/run_local.py`. Some important ones are:

- Input files
 - `data_filename="sample/data/sample.txt"`: The data file name. The data file, which represents input matrix X , can be text file or binary file. The format of file needs to be specified in `data_format`. Matrix elements shall be sorted by rows then columns. Matrix elements in text file needs to be separated by blank characters. Matrix elements in binary file needs to be single-precision floating-point format which occupies 4 bytes each.
 - `is_partitioned=0`: Indicates whether or not the input file has been partitioned. For distributed setting, each machine needs to access their part of data. If `is_partitioned` is set to 0, then each machine needs to get access to the whole data file and NMF app will do the partitioning automatically. Otherwise, the whole data file needs to be partitioned by column id (See *Data partitioning* for details), and each machine will read file named `data_filename.client_id`, e.g. if `data_filename` is “sample.txt”, then machine 0 will read file “sample.txt.0”.
 - `data_format="text"`: Specify the format of input and output data. Can be “text” or “binary”. Note that the format of `time.txt` and `loss.txt` does not depend on this parameter and is always text.
 - `load_cache=0`: Specify if whether or not load previous saved results. If `load_cache` is set to 1, the app will load the results in directory `cache_dirname`, which has to contain the matrix R and the partial matrix L_i . The data format of files in `cache_dirname` need to be the same as specified in `data_format`.
 - `cache_dirname="N/A"`: Required if `load_cache` is set to 1. See `load_cache`.
- Output files
 - `output_dirname="sample/output"`: Directory where the results will be put into. The `output_dirname` is path relative to `bosen/app/NMF`. Set `output_path` directly if you want to use absolute path.
- Objective function parameters
 - `m`: Dimension of input data. It is also the number of columns (for text format input file) in input data.
 - `n`: Size of input data. It is also the number of rows (for text file) in input data.

- rank: Rank of matrix factorization.
- Optimization parameters
 - num_epochs=500: Perform how many epochs during optimization. Each epoch approximately visit all data points once (not exactly because the data points are visited stochastically).
 - minibatch_size=100: Size of minibatch.
 - init_step_size=0.01: Base init step size. The step size at the t-th iter is in the form of $\text{init_step_size} * (t + \text{step_size_offset})^{(-\text{step_size_pow})}$. As we are alternatively optimizing R and L, and there might be multiple threads or multiple clients running, the actual step size for R and L is rescaled by a factor related to number of threads and dimension of data. For most applications, it is enough to only tune this parameter, keeping `step_size_offset` and `step_size_pow` to 0. If the output contains nan during optimization, then the `init_step_size` shall be decreased. But smaller step size may result in lower convergence speed.
 - step_size_offset=0.0: See `init_step_size`.
 - step_size_pow=0.0: See `init_step_size`.
 - init_L_low=0.0: Elements in matrix L are initialized from uniform distribution with lower bound `init_L_low` and upper bound `init_L_high`.
 - init_L_high=0.0: Elements in matrix L are initialized from uniform distribution with lower bound `init_L_low` and upper bound `init_L_high`.
 - init_R_low=0.0: Elements in matrix R are initialized from uniform distribution with lower bound `init_R_low` and upper bound `init_R_high`.
 - init_R_high=0.0: Elements in matrix R are initialized from uniform distribution with lower bound `init_R_low` and upper bound `init_R_high`.
 - num_iter_L_per_minibatch=10: How many iterations to perform gradient on a randomly picked data point to update the corresponding row in L matrix. The default value is enough for most applications. A bigger value will result in better optimization results at a given iteration, but at the cost of more time.
 - init_step_size_R=0.0: Optional. Valid when it is set to nonzero values. For advanced users, the step size for L and R can be set directly by setting `init_step_size_R`, `step_size_offset_R`, `step_size_pow_R`, `init_step_size_L`, `step_size_offset_L`, `step_size_pow_L` directly. Note that `init_step_size_R` or `init_step_size_L` must be set to nonzero values if you want to set them directly instead of using step size determined by base step size. For example, if `init_step_size_R` is set to a nonzero value, the step size at the t-th iter will be $\text{init_step_size_R} * (t + \text{step_size_offset_R})^{(-\text{step_size_pow_R})}$. The step size formula for L is analogous.
 - step_size_offset_R=0.0: Optional. See `init_step_size_R`.
 - step_size_pow_R=0.0: Optional. See `init_step_size_R`.
 - init_step_size_L=0.0: Optional. See `init_step_size_R`.
 - step_size_offset_L=0.0: Optional. See `init_step_size_R`.
 - step_size_pow_L=0.0: Optional. See `init_step_size_R`.
- Evaluation parameters
 - num_eval_minibatch=100: Evaluate objective function per how many minibatches.
 - num_eval_samples=\$n: How many samples to pick for each worker thread to evaluate objective function. If the size of data is so large that evaluating objective function takes too much time, you can use either a smaller `num_eval_samples` or a larger `num_eval_minibatch` value.

- System parameters
 - `num_worker_threads=4`: Number of threads running NMF on each machine.
 - `table_staleness=0`: The staleness for tables.
 - `maximum_running_time=0.0`: The app will try to terminate after running `maximum_running_time` hours. Valid if the value is greater than 0.0.

After all parameters have been chosen appropriately, use the command `./script/run_local.py`, then NMF application will run in background. The results will be put as specified in `output_dirname`. Matrix `R` will be stored in `B.txt` in client 0 (whose ip appears in the first line in `hostfile`). Matrix `L` will be stored in a row-partitioned manner, i.e., client `i` will have a `L.txt.i` in `output_dirname`, and the whole matrix `L` can be obtained by putting all `L.txt.i` together, which will be explained in [Data partitioning](#).

Terminating the NMF app

The NMF app runs in the background, and outputs its progress to log files in user-specified directory. If you need to terminate the app before it finishes, just run

```
./script/kill.py <petuum_ps_hostfile>
```

Data partitioning

If there are multiple machines in host file, each machine will only take a part of input matrix. Concretely, if there are `num_client` clients, client `i` will read the `j`-th row of `X` if `j mod num_client=i`. For example, if the data matrix `X` is:

```
1.0 1.0 1.0 1.0
2.0 2.0 2.0 2.0
3.0 3.0 3.0 3.0
4.0 4.0 4.0 4.0
5.0 5.0 5.0 5.0
```

In the above `X`, the feature dimension is 4, and the size of data is 5. Suppose there are 3 machines in host file, then machine 0 will read the 1-st, 4-th row of `X`, machine 2 will read the 2-nd, 5-th row of `X` and machine 3 will read the 3-rd row of `X`.

Each machine only needs to read part of data, so we provide a parameter `is_partitioned`. In order to use partitioned data (for example, when each client's disk is not enough to hold all data), `is_partitioned` shall be set to 1, and data needs to be partitioned according to the above partitioning strategy. Note that the name of partitioned data on each client needs to be `data_filename.client_id`. We have provided a tool for partitioning data in `script/partition_data.py`. Run `python script/partition_data.py` for its usage.

When using multiple machines, the result matrix `L` will be stored distributedly corresponding to the part of input data that client reads. For example, In the above example, machine 0 will store `L.txt.0`, which is the 1-st, 4-th row of `L`. We have provided a tool for merging partitioned `L.txt.i` together in `script/merge_data.py`. Run `python script/merge_data.py` for its usage.

File IO from HDFS

Put datasets to HDFS

```
hadoop fs -mkdir -p /user/bosen/dataset/nmf
hadoop fs -put sample /user/bosen/dataset/nmf
```

Change the corresponding file paths in `script/run_local.py` to the right HDFS path. Comment out the local path.

```
# , "output_path": join(app_dir, "sample/output")
, "output_path": "hdfs://hdfs-domain/user/bosen/dataset/nmf/sample/output"
# , "data_file": join(app_dir, "sample/data/sample.txt")
, "data_file": "hdfs://hdfs-domain/user/bosen/dataset/nmf/sample/data/sample.txt"
```

Launch it over ssh

```
./script/launch.py
```

Check the output

```
hadoop fs -cat /user/bosen/dataset/nmf/sample/output/time.txt
```

Use Yarn to launch NMF app

```
./script/launch_on_yarn.py
```

CHAPTER 11

Sparse Coding

Given an input matrix X , the Sparse Coding app (implemented on **Bösen**) learns the dictionary B and the coefficients matrix S such that $S*B$ is approximately equal to X and the coefficients matrix S is sparse.

If X is N -by- M , then B will be K -by- M and S will be N -by- K , where N is the number of data points, M is the dimension of the data, K is a user-supplied parameter that controls the size of the dictionary. The objective of sparse coding app is to minimize the function $\|X - S*B\|_2^2 + \lambda \|S\|_1$ subject to the constraint that $\|B(i, :)\|_2 \leq c$, where λ and c are hyperparameters and λ/c controls the regularization strength.

Our Sparse Coding app uses the projected Stochastic Gradient Descent (SGD) algorithm to learn the dictionary B and the coefficients S .

Performance

On a dataset with 1M data instances; 22K feature dimension; 22M parameters, the Bösen Sparse Coding implementation converges in approximately 12.5 hours, using 8 machines (16 cores each).

Quick Start

The Sparse Coding app can be found in `bosen/app/sparscoding/`. From this point on, all instructions will assume you are in `bosen/app/sparscoding/`. After building PMLS (as explained earlier in this manual), you can build the Sparse Coding app from `bosen/app/sparscoding` by running

```
make
```

This will put the Sparse Coding binary in the subdirectory `bin/`.

Create directory

```
mkdir sample
mkdir sample/data
mkdir sample/output
```

Create a simulated dataset:

```
./script/make_synth_data.py 5 100 6 sample/data/sample.txt 1
```

Change `app_dir` in `script/run_local.py`; see example below. You need to provide the full path to the Sparse Coding directory. This will allow for correct YARN operation (if required).

```
app_dir = "/home/user/bosen/app/sparsecoding"
```

then you can test that the app works on the local machine (with 4 worker threads). Now, run:

```
./script/launch.py
```

The Sparse Coding app runs in the background (progress is output to stdout). After a few minutes, you should get the following output files in the subdirectory `sample/output/`:

```
B.txt
S.txt.0
loss.txt
time.txt
```

The file `B.txt` and the file `S.txt.0` are the optimization results of dictionary `B` and coefficients `S`, which looks like this:

```
0.187123    0.584781    0.716686    0.259561    -0.20495
0.378984    0.0456092   -0.667287   -0.218985   -0.600886
-0.863578   -0.162164   -0.244522   -0.171278   -0.372572
0.188104    0.558853    0.744098    0.252362    0.186899
-0.71181    -0.165808   -0.562194    0.00220546  -0.386997
0.075108    0.496782    0.0201851   0.692971    0.516672
```

```
0   -0.232664   0   0.273726   -0.123055   0.216856
0.342858   0   0   0.249487   0   0.121574
0.392845   0   -0.214571   0.486131   -0.486699   0
0.346318   0   0   0.258979   0   0.118132
0   -0.221088   0   0.297128   -0.113161   0.209509
0   0   -0.058158   0.0697259   -0.00593826  0.378869
0   -0.252994   0   0.286514   -0.102685   0.203422
0.156121   0   -0.0727321  0.183034   -0.0765836   0.0734904
0.359529   0   0   0.249562   0   0.120825
...
```

The file `loss.txt` contains the statistics of loss function evaluated at iterations specified by user. If there are `num_client` machines in the experiment, it will have `num_client` columns, and the i -th column is the loss function evaluated by the i -th machine. Note that loss function is evaluated by averaging loss function at randomly sampled data points. The file `time.txt` also has `num_client` columns, and contains the time (seconds) taken between evaluations on each client (Evaluating time is excluded). Don't be surprised if the first row of `time.txt` is nearly 0, that's because the loss function is evaluated once before any optimization is conducted. Also note that at the end of `time.txt` and `loss.txt` there might be some `N/As`, that's invalid and is due to the fact that different clients may evaluate different times.

You won't see the exact same numeric values in different runs as there are multiple optimums of the optimization problem. All you need to confirm is that the optimized `S` shall be sparse and average objective function value of the final result should be lower than or close to 0.1.

Data format

The Sparse Coding app takes a dense matrix X as input, where each row corresponds to a data sample and each column corresponds to a feature. The file containing X should be a binary or text file, and elements are sorted by rows then columns:

```
ele_0_0    ele_0_1    ... ele_0_n-1
ele_1_0    ele_1_1    ... ele_1_n-1
...
ele_n-1_0  ele_n-1_1  ... ele_n-1_n-1
```

You must specify the format of file (binary or text) by using the parameter `file_format`, which will be explained in *Running the Sparse Coding application* section.

Creating synthetic data

We provide a synthetic data generator for testing the app (requires Python 2.7). To see detailed instructions, run

```
python script/make_synth_data.py
```

Running the Sparse Coding application

There are many parameters involved for running Sparse Coding. We provide default values for all of them in `scripts/run_local.py`. Some important ones are:

- Input files
 - `data_filename="sample/data/sample.txt"`: The data file name. The data file, which represents input matrix X , can be text file or binary file. The format of file needs to be specified in `data_format`. Matrix elements shall be sorted by rows then columns. Matrix elements in text file needs to be separated by blank characters. Matrix elements in binary file needs to be single-precision floating-point format which occupies 4 bytes each.
 - `is_partitioned=0`: Indicates whether or not the input file has been partitioned. For distributed setting, each machine needs to access their part of data. If `is_partitioned` is set to 0, then each machine needs to get access to the whole data file and Sparse Coding app will do the partitioning automatically. Otherwise, the whole data file needs to be partitioned by column id (See *Data partitioning* for details), and each machine will read file named `data_filename.client_id`, e.g. if `data_filename` is “sample.txt”, then machine 0 will read file “sample.txt.0”.
 - `data_format="text"`: Specify the format of input and output data. Can be “text” or “binary”. Note that the format of `time.txt` and `loss.txt` does not depend on this parameter and is always text.
 - `load_cache=0`: Specify if whether or not load previous saved results. If `load_cache` is set to 1, the app will load the results in directory `cache_dirname`, which has to contain the matrix B and the partial matrix S_i . The data format of files in `cache_dirname` need to be the same as specified in `data_format`.
 - `cache_dirname="N/A"`: Required if `load_cache` is set to 1. See `load_cache`.
- Output files

- `output_dirname`: Directory where the results will be put into. The `output_dirname` is path relative to `bosen/app/sparsecoding`. Change `output_path` directly if you want to use absolute path.
- Objective function parameters
 - `m`: Dimension of input data. It is also the number of columns (for text format input file) in input data.
 - `n`: Size of input data. It is also the number of rows (for text file) in input data.
 - `dictionary_size`: Size of dictionary.
 - `c`: The l2-norm constraint on dictionary elements.
 - `lambda`: The regularization strength in objective function.
- Optimization parameters
 - `num_epochs=500`: Perform how many epochs during optimization. Each epoch approximately visit all data points once (not exactly because the data points are visited stochastically).
 - `minibatch_size=100`: Size of minibatch.
 - `init_step_size=0.01`: Base init step size. The step size at the t -th iter is in the form of $\text{init_step_size} * (t + \text{step_size_offset})^{-(\text{step_size_pow})}$. As we are alternatively optimizing B and S , and there might be multiple threads or multiple clients running, the actual step size for B and S is rescaled by a factor related to number of threads and dimension of data. For most applications, it is enough to only tune this parameter, keeping `step_size_offset` and `step_size_pow` to 0. If you get nan or inf during optimization, then you should decrease `init_step_size`. However, too small a step size will cause slow convergence speed.
 - `step_size_offset=0.0`: See `init_step_size`.
 - `step_size_pow=0.0`: See `init_step_size`.
 - `init_S_low=0.0`: Elements in matrix S are initialized from uniform distribution with lower bound `init_S_low` and upper bound `init_S_high`.
 - `init_S_high=0.0`: Elements in matrix S are initialized from uniform distribution with lower bound `init_S_low` and upper bound `init_S_high`.
 - `init_B_low=0.0`: Elements in matrix B are initialized from uniform distribution with lower bound `init_B_low` and upper bound `init_B_high`.
 - `init_B_high=0.0`: Elements in matrix B are initialized from uniform distribution with lower bound `init_B_low` and upper bound `init_B_high`.
 - `num_iter_S_per_minibatch=10`: How many iterations to perform gradient on a randomly picked data point to update the corresponding coefficient. The default value is enough for most applications. A bigger value will result in better optimization results at a given iteration, but at the cost of more time.
 - `init_step_size_B=0.0`: Optional. Valid when it is set to nonzero values. For advanced users, the step size for B and S can be set directly by setting `init_step_size_B`, `step_size_offset_B`, `step_size_pow_B`, `init_step_size_S`, `step_size_offset_S`, `step_size_pow_S` directly. Note that `init_step_size_B` or `init_step_size_S` must be set to nonzero values if you want to set them directly instead of using step size determined by base step size. For example, if `init_step_size_B` is set to a nonzero value, the step size at the t -th iter will be $\text{init_step_size_B} * (t + \text{step_size_offset_B})^{-(\text{step_size_pow_B})}$. The step size formula for S is analogous.
 - `step_size_offset_B=0.0`: Optional. See `init_step_size_B`.
 - `step_size_pow_B=0.0`: Optional. See `init_step_size_B`.

- `init_step_size_S=0.0`: Optional. See `init_step_size_B`.
- `step_size_offset_S=0.0`: Optional. See `init_step_size_B`.
- `step_size_pow_S=0.0`: Optional. See `init_step_size_B`.

- Evaluation parameters

- `num_eval_minibatch=100`: Evaluate objective function per how many minibatches.
- `num_eval_samples=$n`: How many samples to pick for each worker thread to evaluate objective function. If the size of data is so large that evaluating objective function takes too much time, you can use either a smaller `num_eval_samples` or a larger `num_eval_minibatch` value.

- System parameters

- `num_worker_threads=4`: Number of threads running Sparse Coding on each machine.
- `table_staleness=0`: The staleness for tables.
- `maximum_running_time=0.0`: The app will try to terminate after running `maximum_running_time` hours. Valid if the value is greater than 0.0.

After all parameters have been chosen appropriately, use the command `./script/run_local.py`, then Sparse Coding application will run in background. The results will be put as specified in `output_dirname`. Matrix `B` will be stored in `B.txt` in client 0 (whose ip appears in the first line in `hostfile`). Matrix `S` will be stored in a row-partitioned manner, i.e., client `i` will have a `S.txt.i` in `output_dirname`, and the whole matrix `S` can be obtained by putting all `S.txt.i` together, which will be explained in [Data partitioning](#).

Terminating the Sparse Coding app

The Sparse Coding app runs in the background, and outputs its progress to log files in user-specified directory. If you need to terminate the app before it finishes, just run

```
./script/kill.py <petuum_ps_hostfile>
```

Data partitioning

If there are multiple machines in host file, each machine will only take a part of input matrix. Concretely, if there are `num_client` clients, client `i` will read the `j`-th row of `X` if `j mod num_client=i`. For example, if the data matrix `X` is:

```
1.0 1.0 1.0 1.0
2.0 2.0 2.0 2.0
3.0 3.0 3.0 3.0
4.0 4.0 4.0 4.0
5.0 5.0 5.0 5.0
```

In the above `X`, the feature dimension is 4, and the size of data is 5. Suppose there are 3 machines in host file, then machine 0 will read the 1-st, 4-th row of `X`, machine 2 will read the 2-nd, 5-th row of `X` and machine 3 will read the 3-rd row of `X`.

Each machine only needs to read part of data, so we provide a parameter `is_partitioned`. In order to use partitioned data (for example, when each client's disk is not enough to hold all data), `is_partitioned` shall be set to 1, and data needs to be partitioned according to the above partitioning strategy. Note that the name of partitioned data on each client needs to be `data_filename.client_id`. We have provided a tool for partitioning data in `scripts/partition_data.py`. Run `python scripts/partition_data.py` for its usage.

When using multiple machines, the result coefficients S will be stored in a distributed fashion, corresponding to the part of input data that client reads. For example, In the above example, machine 0 will store `S.txt.0`, which is the 1-st, 4-th row of X . We have provided a tool for merging partitioned `S.txt.i` together in `scripts/merge_data.py`. Run `python scripts/merge_data.py` for its usage.

File IO from HDFS

Put datasets to HDFS

```
hadoop fs -mkdir -p /user/bosen/dataset/sc
hadoop fs -put sample /user/bosen/dataset/sc
```

Change the corresponding file paths in `script/run_local.py` to the right HDFS path. Comment out the local path.

```
# , "output_path": join(app_dir, "sample/output")
, "output_path": "hdfs://hdfs-domain/user/bosen/dataset/sc/sample/output"
# , "data_file": join(app_dir, "sample/data/sample.txt")
, "data_file": "hdfs://hdfs-domain/user/bosen/dataset/sc/sample/data/sample.txt"
```

Launch it over ssh

```
./script/launch.py
```

Check the output

```
hadoop fs -cat /user/bosen/dataset/sc/sample/output/time.txt
```

Use Yarn to launch Sparse Coding app

```
./script/launch_on_yarn.py
```

CHAPTER 12

Lasso and Logistic Regression

PMLS provides a linear solver for Lasso and Logistic Regression, using the **Strads** scheduler system. These apps can be found in `strads/apps/linear-solver_release/`. From this point on, all instructions will assume you are in `strads/apps/linear-solver_release/`.

After building the Strads system (as explained in the installation page), you may build the the linear solver from `strads/apps/linear-solver_release/` by running

```
make
```

Test the app (on your local machine) by running

```
python lasso.py for lasso
```

```
python logistic.py for LR
```

This will perform Lasso/LR on two separate synthetic data sets in `./input`. The estimated model weights can be found in `./output`.

Note: on some configurations, MPI may report that the program “exited improperly”. This is not an issue as long as it occurs after this line:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ Congratulation ! Finishing task. output file  ./output/  
↪coeff.out
```

If you see this line, the Lasso/LR program has finished successfully.

Performance

The logistic regression app on Strads can solve a 10M-dimensional sparse problem (30GB) in 20 minutes, using 8 machines (16 cores each). The Lasso app can solve a 100M-dimensional sparse problem (60GB) in 30 minutes, using 8 machines (16 cores each).

Input data format

The Lasso/LR apps use the [MatrixMarket](#) format:

```
%%MatrixMarket matrix coordinate real general
num_rows num_cols num_nonzeros
row col value
row col value
row col value
```

The first line is the MatrixMarket header, and should be copied as-is. The second line gives the number of rows N , columns M , and non-zero entries in the matrix. This is followed by `num_nonzeros` lines, each representing a single matrix entry $A(\text{row}, \text{col}) = \text{value}$ (where `row` and `col` are 1-indexed as like Matlab).

Output format

The output file of Lasso/LR also follows the MatrixMarket format, and looks something like this:

```
%%MatrixMarket matrix coordinate real general
1 1000 108
1 6 -0.02388
1 9 -0.08180
1 40 -0.13945
1 63 -0.07788
...
```

This represents the model weights as a single row vector.

Machine configuration

See *Strads configuration files*

Program Options

The Lasso/LR is launched using a python script, e.g. `lasso.py/logistic.py`.

```
machfile = ['./singlemach.vm']

# data setting
↳
datafilex = ['./input/lasso500by1K.X.mmt']
datafiley = ['./input/lasso500by1K.Y.mmt']
csample = [' 500 ']
column = [' 1000 ']

# scheduler setting
↳
cscheduler = [' 1 ']
scheduler_threads = [' 1 ']

# worker thread per machine
↳
```

```

worker_threads = [' 1 ']

# degree of parallelism
↪
set_size = [' 1 ']

prog = ['./bin/cdsolver ']

os.system(" mpirun -machinefile "+machfile[0]+" "+prog[0]+" --machfile "+machfile[0]+
↪ " -threads "+worker_threads[0]+" \
-samples "+csample[0]+" -columns "+column[0]+" -data_xfile "+datafilex[0]+" -data_
↪ yfile "+datafiley[0]+" -schedule_si\
ze "+set_size[0]+" -max_iter 30000 -lambda 0.001 -scheduler "+cscheduler[0]+" -
↪ threads_per_scheduler "+scheduler_threa\
ds[0]+" -weight_sampling=false -check_interference=false -algorithm lasso");

```

The basic options are:

- `datafilex`: Path to the design matrix file, which must be present/visible to all machines. We strongly recommend providing the full path name to the data file.
- `datafiley`: Path to the observation file, which must be present/visible to all machines. We strongly recommend providing the full path name to the data file.
- `samples, columns`: number of rows and columns of the design matrix
- `threads`: number of threads to create per worker machine
- `scheduler`: number of scheduler machines to use
- `threads_per_scheduler`: number of threads to create per scheduler machine
- `max_iter`: maximum number of iterations
- `algorithm`: flag to specify algorithm to run (lasso/logistic)

The following options are available for advanced users, who wish to control the dynamic scheduling algorithm used in the linear solver:

- `weight_sampling, check_interference`: flags to disable/enable parameter-weight-based dynamic scheduling and parameter dependency checking. We strongly recommend to set both of them to the same value (either false or true).
- `schedule_size`: the number of parameters to schedule per iteration. Increasing this can improve performance, but only up to a point.

Distance Metric Learning

This app implements Distance Metric Learning (DML) as proposed in [1], on **Bösen**. DML takes data pairs labeled either as similar or dissimilar to learn a Mahalanobis distance matrix such that similar data pairs will have small distances while dissimilar pairs are separated apart.

[1] Eric P. Xing, Michael I. Jordan, Stuart Russell, and Andrew Y. Ng. “Distance metric learning with application to clustering with side-information.” In Advances in neural information processing systems, pp. 505-512. 2002.

Performance

On a dataset with 1M data instances; 22K feature dimension; 22M parameters, the Bösen DML implementation converges in approximately 15 minutes, using 4 machines (64 cores each).

Going from 1 to 4 machines results in a speedup of roughly 3.75x.

Quick start

The DML app can be found in `bosen/app/dml/`. **From this point on, all instructions will assume you are in `bosen/app/dml/`.** After building PMLS (as explained earlier in this manual), you can build the DML app from `bosen/app/dml` by running

```
make
```

This will put the DML binary in the subdirectory `bin/`.

Next, download the preprocessed MNIST dataset from http://www.cs.cmu.edu/~pengtaox/dataset/mnist_petuum.zip. Unzip the files and copy them to the `datasets` folder. You can use the following commands:

```
cd datasets
wget http://www.cs.cmu.edu/%7Epengtaox/dataset/mnist_petuum.zip
unzip mnist_petuum.zip
cd ..
```

Change `app_dir` in `script/run_local.py`; see example below. You need to provide the full path to the DML directory. This will allow for correct YARN operation (if required).

```
app_dir = "/home/user/bosen/app/dml"
```

Using the MNIST dataset, you can test that the app works on the local machine (with 4 worker threads). From the `app/dml/` directory, run:

```
./script/launch.py
```

The DML app runs in the background (progress is output to stdout). After the app terminates, you should get one output file:

```
datasets/dismat.txt
```

`dismat.txt` saves the distance matrix in row major order and each line corresponds to a row.

Running the Distance Metric Learning Application

Parameters of the DML app are specified in `script/run_local.py`, including:

- `<num_worker_threads>`: how many worker threads to use in each machine
- `<staleness>`: staleness value
- `<parafire>`: configuration file of DML parameters
- `<feature_file>`: file storing features of data samples
- `<simi_pairs_file>`: file storing similar pairs
- `<diff_pairs_file>`: file storing dissimilar pairs
- `<model_weight_file>`: the path where the learned distance matrix will be stored

The machine file is specified in `script/launch.py`:

```
hostfile_name = "machinefiles/localserver"
```

After configuring these parameters, launch the app by:

```
./script/launch.py
```

Format of DML Configuration File

The DML configurations are stored in `<parameter_file>`. Each line corresponds to a parameter and its format is

```
<parameter_name>: <parameter_value>
```

`<parameter_name>` is the name of the parameter. It is followed by a `:` (there is no blank between `<parameter_name>` and `:`). `<parameter_value>` is the value of this parameter. Note that `:` and `<parameter_value>` must be separated by a blank.

The list of parameters and their meanings are:

- `src_feat_dim`: dimension of the observed feature space

- `dst_feat_dim`: dimension of the latent space
- `learn_rate`: learning rate of Stochastic Gradient Descent (SGD)
- `epoch`: number of epochs in SGD
- `num_total_pts`: number of data samples
- `num_simi_pairs`: number of similar pairs
- `num_diff_pairs`: number of dissimilar pairs
- `mini_batch_size`: size of mini-batch
- `num_iters_evaluate`: every `<num_iters_evaluate>` iterations, we do an objective function evaluation
- `num_smp_evaluate`: when evaluating the objective function, we randomly sample `<num_smp_evaluate>` points to compute the objective

Note that, the order of the parameters cannot be switched. Here is an example:

```
src_feat_dim: 780
dst_feat_dim: 600
lambda: 1
thre: 1
learn_rate: 0.001
epoch: 50
num_total_pts: 60000
num_simi_pairs: 1000
num_diff_pairs: 1000
mini_batch_size: 100
num_iters_evaluate: 1000
num_smps_evaluate: 1000
```

Input data format

The input data is stored in three files

```
<feat file>
<simi pairs file>
<diff pairs file>
```

`<feat file>` stores the features of data samples in sparse format. Each line corresponds to one data sample. The format is

```
<class label> <num of nonzero features> <feature id>:<feature value> <feature
id>:<feature value> ...
```

`<simi pairs file>` stores the data pairs labeled as similar. Each line contains a similar data pair separated with tab. `<diff pairs file>` stores the data pairs labeled as dissimilar. Each line contains a dissimilar data pair separated with tab.

Output model format

The app outputs the learned model to one file:

```
<dismat_file>
```

<dismat_file> saves the distance matrix in row major order and each line corresponds to a row. Elements in each row are separated with blank.

Terminate DML app

The DML app runs in the background, and outputs its progress to stdout. If you need to terminate the app before it finishes, for distributed version, run

```
./script/kill.py <hostfile>
```

File IO from HDFS

Put datasets to HDFS

```
hadoop fs -mkdir -p /user/bosen/dataset/dml
hadoop fs -put datasets /user/bosen/dataset/dml
```

Change the corresponding file paths in `script/run_local.py` to the right HDFS path. Comment out the local path.

```
# , "parafire": join(app_dir, "datasets/dml_para.txt")
, "parafire": "hdfs://hdfs-domain/user/bosen/dataset/dml/datasets/dml_para.txt"
# , "feature_file": join(app_dir, "datasets/mnist_petuum/mnist_reformatted.txt")
, "feature_file": "hdfs://hdfs-domain/user/bosen/dataset/dml/datasets/mnist_petuum/
↳mnist_reformatted.txt"
# , "simi_pairs_file": join(app_dir, "datasets/mnist_petuum/mnist_simi_pairs.txt")
, "simi_pairs_file": "hdfs://hdfs-domain/user/bosen/dataset/dml/datasets/mnist_petuum/
↳mnist_simi_pairs.txt"
# , "diff_pairs_file": join(app_dir, "datasets/mnist_petuum/mnist_diff_pairs.txt")
, "diff_pairs_file": "hdfs://hdfs-domain/user/bosen/dataset/dml/datasets/mnist_petuum/
↳mnist_diff_pairs.txt"
# , "model_weight_file": join(app_dir, "datasets/dismat.txt")
, "model_weight_file": "hdfs://hdfs-domain/user/bosen/dataset/dml/datasets/dismat.txt"
```

Launch it over ssh

```
./script/launch.py
```

Check the output

```
hadoop fs -cat /user/bosen/dataset/dml/datasets/dismat.txt
```

Use Yarn to launch DML app

```
./script/launch_on_yarn.py
```

K-Means Clustering

K-means is a clustering algorithm, which identifies cluster centers based on Euclidean distances. Our K-Means app on **Bösen** uses the Mini-Batch K-means algorithm [1].

Quick Start

The app can be found at `bosen/app/kmeans`. From this point on, all instructions will assume you are at `bosen/app/kmeans`. After building the main PMLS libraries, you can build `kmeans`:

```
make -j2
```

Then

```
# Create run script from template
cp script/launch.py.template script/launch.py

chmod +x script/launch.py
./script/launch.py
```

The last command runs `kmeans` using the provided sample dataset `dataset/sample.txt` and output the found centers in `output/out.centers` and the cluster assignments in `output/out.assignmentX.txt` where `X` is the worker ID (each worker outputs cluster assignments in its partition).

Use HDFS

Kmeans supports HDFS read and output. You need to build Bösen with `HAS_HDFS = -DHAS_HADOOP` in `bosen/defs.mk`. See the [YARN/HDFS page](#) for detailed instructions.

Rebuild the binary if you rebuilt the library with Hadoop enabled (under `bosen/app/kmeans`):

```
make clean all
```

Let's copy the demo data to HDFS, where `/path/to/data/` should be replaced:

```
hadoop fs -mkdir -p /path/to/data/  
hadoop fs -put dataset/sample.txt /path/to/data/  
  
# quickly verify  
hadoop fs -ls /path/to/data/
```

Change a few paths in `script/launch.py`, where `<ip>:<port>` points to HDFS (You can find them on the HDFS web UI):

```
"train_file": "hdfs://<ip>:<port>/path/to/data/sample.txt"  
"output_file_prefix": "hdfs://<ip>:<port>/path/to/data/out"
```

Also uncomment this line in `script/launch.py`:

```
cmd += "export CLASSPATH=`hadoop classpath --glob`:$CLASSPATH; "
```

to add hadoop path. Older hadoop might not have `hadoop classpath --glob`. You need to make sure the class path is set appropriately.

Then launch it as before:

```
./script/launch.py  
  
# Check the result  
hadoop fs -ls /path/to/data/out*
```

Use Yarn

We will launch job through Yarn and read/output to HDFS. Make sure you've built Yarn by running `gradle build` under `bosen/src/yarn` and have HDFS enabled in `bosen/defs.mk` like before.

Remove the outputs from previous runs:

```
hadoop fs -rm /path/to/data/out.*
```

Create run script from template

```
cp script/run_local.py.template script/run_local.py
```

In `scripts/run_local.py`, set `train_file`, `output_file_prefix` as previously described in the Use HDFS section. Also set the `app_dir` to the absolute path, e.g., `/path/to/bosen/app/kmeans`. Then launch it:

```
chmod +x script/launch_on_yarn.py  
  
# script/launch_on_yarn.py will call script/run_local.py  
./script/launch_on_yarn.py
```

You can monitor the job progress in Yarn's WebUI. There you can also find the application ID (e.g., `application_1431548686685_0240`). You can then get the `stderr/stdout` outputs:

```
yarn logs -applicationId application_1431548686685_0240
```

There you should see similar output as before. As before, you can check the results by `hadoop fs -cat /path/to/data/out.centers`.

Input Format

The input data needs to be in the libsvm format. A data point would look like:

```
+1 1:0.504134425797 2:-0.259344641268 3:1.74953783689 4:2.62475223767 5:-3.
↪12607240823 6:8.9244514355 7:5.69376634865 8:8.41921260536 9:0.0805904064359 10:4.
↪36430063362
```

where +1 is ignored and feature index starts at 1. We provide a script to generate synthetic dataset:

```
cd dataset
# Generates 100 10-dim points drawn from 10 centers
python data_generate.py 10 10 100 libsvm
```

2 files are generated. `dataset/synthetic.txt` contains the datapoints and `dataset/centers.txt` contains the centers for these data points.

Setting up machines

Put the desired machine IP addresses in the Parameter Server machine file. See this page for more information: [Configuration Files for PMLS apps](#).

Common Parameters

In `script/launch.py.template` and `script/run_local.py.template`:

- `host_filename` = **Parameter Server machine file**. Contains a list of the ip addresses and port numbers for machines in the cluster.
- `train_file` = Name of the training file present under the dataset folder. It is assumed the entire file is present on a shared file system.
- `total_num_of_training_samples` = Number of data points that need to be considered.
- `num_epochs` = Number of mini batch Iterations.
- `mini_batch_size` = Size of the mini batch.
- `num_centers` = The number of cluster centers.
- `dimensionality` = The number of dimensions in each vector.
- `num_app_threads` = The number of application threads to run minibatch iterations in parallel.
- `load_clusters_from_disk` = true/false indicating whether to do initialization of centers from external source. If set False, a random initialization is performed.
- `cluster_centers_input_location` = Location to the file containing the cluster centers. This file is read if the above argument is set to true.

- `output_dir` = The directory location where the cluster centers information is written after optimization. The assignments for the data points is written in the assignments sub-directory of the same folder.

Center Initialization

As of now, we are only providing support for a random initialization for centers. But you could also provide any choice of clusters centers as input to the algorithm. This option can be enabled in the script provided for running the application. The data format for the input centers is same as the one specified for the training dataset.

Output

The centers obtained by running the application can be found under the `output/` subdirectory. The indices of the centers run from 0 to $k-1$, where k is the number of cluster centers. The assignment of the nearest cluster center to the training data points can be found in the assignments subdirectory. The files in this subdirectory contain the line number of the training data point in the training file and its corresponding cluster center index.

References

[1]: Sculley, D. “Web-scale k-means clustering.” Proceedings of the 19th international conference on World wide web. ACM, 2010.

CHAPTER 15

Random Forest

A **Random Forest** is a classification algorithm that uses a large number of decision trees. Our Random Forest app is implemented on **Bösen**, using **C4.5** to learn the trees.

Quick Start

Random Forest can be found in `bosen/app/rand_forest`. From this point on, all instructions will assume you are in `bosen/app/rand_forest`. After building the main PMLS libraries (as explained earlier in this manual), you can build the Rand Forest app from `bosen/app/rand_forest` by running

```
make -j2
```

This will put the `rand_forest` binaries in the subdirectory `bin`.

We now turn to the run script. A template is provided for you:

```
cp script/launch.py.template script/launch.py

# Make it executable
chmod +x script/launch.py

# Launch it through ssh.
./script/launch.py
```

The last command launches 4 threads on local node (single node) using **Iris dataset**. You should see something like

```
I0702 03:30:21.282187 18509 rand_forest.cpp:31] Each thread trained 122/125 trees.
I0702 03:30:21.290820 18509 rand_forest.cpp:31] Each thread trained 123/125 trees.
I0702 03:30:21.308017 18509 rand_forest.cpp:31] Each thread trained 124/125 trees.
SaveTrees to hdfs://cogito.local:8020/user/wdai/dataset/rand_forest/out.part0
I0702 03:30:21.315143 18509 rand_forest.cpp:31] Each thread trained 125/125 trees.
SaveTrees to hdfs://cogito.local:8020/user/wdai/dataset/rand_forest/out.part0
SaveTrees to hdfs://cogito.local:8020/user/wdai/dataset/rand_forest/out.part0
SaveTrees to hdfs://cogito.local:8020/user/wdai/dataset/rand_forest/out.part0
```

```

I0702 03:30:22.283545 18509 rand_forest_engine.cpp:210] client 0 train error: 0.025_
↳(evaluated on 120 training data)
I0702 03:30:22.283751 18509 rand_forest_engine.cpp:221] client 0 test error: 0.1_
↳(evaluated on 30 test data)
I0702 03:30:22.310386 18509 rand_forest_engine.cpp:316] Test using 500 trees.
I0702 03:30:22.321123 18509 rand_forest_engine.cpp:228] Test error: 0.1 computed on_
↳30 test instances.
I0702 03:30:22.321923 18392 rand_forest_main.cpp:158] Rand Forest finished and shut_
↳down!

```

The results are saved to output/.

Use HDFS

Random Forest supports HDFS read and output. You need to build Bösen with `HAS_HDFS = -DHAS_HADOOP` in `bosen/defs.mk`. See the [YARN/HDFS page](#) for detailed instructions.

Rebuild the binary if you rebuilt the library with Hadoop enabled (under `bosen/app/rand_forest`):

```
make clean all
```

Let's copy the demo data to HDFS, where `/path/to/data/` should be replaced:

```

hadoop fs -mkdir -p /path/to/data/
hadoop fs -put dataset/iris.* /path/to/data/

# quickly verify
hadoop fs -ls /path/to/data/

```

Change a few paths in `script/launch.py`, where `<ip>:<port>` points to HDFS (You can find them on the HDFS web UI):

```

"train_file": "hdfs://<ip>:<port>/path/to/data/iris.train"
"test_file":  "hdfs://<ip>:<port>/path/to/data/iris.test"
"pred_file":  "hdfs://<ip>:<port>/path/to/data/pred"
"output_file": "hdfs://<ip>:<port>/path/to/data/output"

```

Also uncomment this line in `script/launch.py`:

```
cmd += "export CLASSPATH=`hadoop classpath --glob`:$CLASSPATH; "
```

to add hadoop path. Older hadoop might not have `hadoop classpath --glob`. You need to make sure the class path is set appropriately.

Then launch it as before:

```

./script/launch.py

# Check the result
hadoop fs -ls /path/to/data/

```

Use Yarn

We will launch job through Yarn and read/output to HDFS. Make sure you've built Yarn by running `gradle build` under `bosen/src/yarn` and have HDFS enabled in `bosen/defs.mk` like before.

Remove the outputs from previous runs:

```
hadoop fs -rm /path/to/data/out*
hadoop fs -rm /path/to/data/pred
```

Create run script from template

```
cp script/run_local.py.template script/run_local.py
```

In `script/run_local.py`, set `train_file`, `test_file`, `output_file`, `pred_file` as previously described in the Use HDFS section. Also set the `app_dir` to the absolute path, e.g., `/path/to/bosen/app/rand_forest`. Then launch it:

```
chmod +x script/launch_on_yarn.py

# script/launch_on_yarn.py will call script/run_local.py
./script/launch_on_yarn.py
```

You can monitor the job progress in Yarn's WebUI. There you can also find the application ID (e.g., `application_1431548686685_0240`). You can then get the `stderr/stdout` outputs:

```
yarn logs -applicationId application_1431548686685_0240
```

There you should see similar output as before. As before, you can check the results by `hadoop fs -ls /path/to/data/`.

Input format

The input data needs to be in the `libsvm` format or `binary` format.

Setting up machines

The ip addresses and ports of all the machines used should be included in the `hostfile`. You also need to assign client id to each machine in the following format.

```
# Client id, IP, Port (do not include this line)
0 192.168.1.1 10000
1 192.168.1.2 10000
2 192.168.1.3 10000
...
```

See this [page](#) for more details.

Common parameters

All of the parameters required for running the application can be seen and changed in the sample script `script/launch.py.template`.

Here are several important parameters that should be paid attention to.

- **train_file**: Path of training set file.
- **test_file**: Path of test set file.
- **num_trees**: Overall number of trees in the forest,
- **max_depth**: Maximum depth of each tree (ignore it or set to 0 if you want it to grow freely).
- **num_data_subsample**: Number of samples used to train each tree (ignore it or set to 0 if you want it to use all the data).
- **num_feature_subsample**: Number of features used to find best split of each node (ignore it or set to 0 if you want it to use all the feature).
- **host_filename**: Path of hostfile.
- **num_app_threads**: Number of threads used by each client (machine).
- **perform_test**: Do the test if true.
- **save_pred** && **pred_file**: If `save_pred` is true, prediction on test set will be saved in `pred_file`.
- **save_trees** && **output_file**: If `save_trees` is true, trained trees will be saved in `output_file`.
- **load_trees** && **input_file**: If `load_trees` is set as true, training part will be ignored and the app only performs test on trees loaded from `input_file`.

Save prediction to file

The app allows users to save prediction on test set into file for future use.

To save prediction, set `save_pred` as true and set `pred_file`. If using provided script `run_rf.sh` or `run_rf_load.sh`, setting `pred_filename` will be enough.

You can find the output file of prediction results under `output/rf.dataset.Sa.Tb.Mx.Ty/` (*dataset* for the name of dataset, *a* for staleness, *b* for number of epoch, *x* for number of client used and *y* for number of thread in each client). Note that if your machines do not have a shared file system, you can find the file only on client 0 as only thread 0 on client 0 will do the test and save results.

Remember to move important output to somewhere safe since the output directory will be rewrite if rerun the app with parameters above remaining the same.

Save trained trees to file

The app provides the user with methods to do the training and testing separately.

To save trained model, set `save_trees` as true and set `output_file`.

Note that each client will generate one output file with postfix `.partn` where *n* is the client id. Each file will only contain trees trained by that client. So if the machines have a shared file system, you can find all output files in the output path. But if your machines do not have a shared file system, you have to collect them manually from all your clients and combine them to generate a final output file.

The format of output file is shown as below. Each line is the pre-order deserialization of a trained tree. Every `a:b` token indicates a node in the tree. For non-leaf node, `a` is the feature used to split this node and `b` is the split threshold. For leaf node, `a` is -1 and `b` is the label.

```
# Output file format
3:0.934434 -1:0 2:4.703931 -1:1 -1:2
3:1.609452 2:3.019952 -1:0 -1:1 -1:2
2:2.264446 -1:0 3:1.774834 -1:1 -1:2
...
```

Besides, by setting `perform_test` to false at the same time, the app will do the training only and save the trained trees for further test. Otherwise, the app will still do the test and the trained model is also saved to be reused later.

Load trained trees from file

To load trained model from file and do test with it, set `load_trees` as true and set `input_file`. If you want to load your trees after saving them with `save_trees` parameter, remember to merge all the output files generated by all the clients into one and let `input_file` point to that file.

Once `load_trees` is set to true, the app will then skip the training part. One thread will perform the test, ignoring parameters such as `max_depth`, `num_data_subsample`, etc. Set `save_pred=true` to store the test predictions.

Note that if your machines do not have a shared file system, make sure that input file at least exists on Client 0.

Finish up

To kill unfinished process, use

```
# Kill the app
python script/kill.py <hostfile>
```


CHAPTER 16

Support Vector Machine

PMLS provides a SVM solver on distributed system. SVM application can be found in `strads/apps/svm_release/`. From this point on, all instructions will assume you are in `strads/apps/svm_release/`.

After building the Strads system (as explained in the installation page), you may build the the SVM solver from `strads/apps/svm_release/` by running

```
make
```

Test the app (on your local machine) by running

```
python svm.py
```

This will perform SVM on `rcv1.binary sample data` in `./input`. The estimated model weights can be found in `./output`.

Performance

Coming soon

Input data format

The SVM use the [LIBSVM format](#):

```
y col:value col:value col:value col:value col:value
y col:value col:value col:value col:value
y col:value col:value col:value
```

A single line represents a sample that consists of `y` response values and non-zero entries with column indexes. `col` is 1-indexed as like Matlab.

Output format

The output file of SVM looks something like this:

```
col value
col value
col value
col value
col value
...
```

Each row with column id and value represents a non-zero model-parameter.

Machine configuration

See *Strads configuration files*

Program Options

The SVM is launched using a python script, e.g. svm.py.

```
machfile = ['./singlemach.vm']

# data setting
↪ input = ['./input/rcv']

# degree of parallelism
↪ set_size = [' 1 ']

prog = ['./bin/svm-dual ']

os.system(" mpirun -machinefile "+machfile[0]+" "+prog[0]+" --machfile "+machfile[0]+
↪ "-input "+inputfile[0]+" -max_iter 200 -C 1.0 "+" -parallels "+dparallel[0]+" ");
```

The basic options are:

- `inputfile`: Path to the design matrix file, which must be present/visible to all machines. We strongly recommend providing the full path name to the data file.
- `max_iter`: maximum number of iterationsThe following options are available for advanced users, who wish to control the dynamic scheduling algorithm used in the linear solver:
- `dparallel`: the number of parameters to schedule per iteration. Increasing this can improve performance, but only up to a point.
- `C`: is a SVM penalty parameter, which should be larger than 0.

Multi-class Logistic Regression

Multi-class logistic regression, or Multinomial Logistic Regression (MLR) generalizes binary logistic regression to handle settings with multiple classes. It can be applied directly to multiclass classification problem, or used within other models (e.g. the last layer of a deep neural network). Our MLR app is implemented on the Bösen system.

Performance

Using 8 machines (16 cores each), the MLR application converges in approximately 20 iterations or 6 minutes, on the [MNIST dataset](#) (8M samples, 784 feature dimensions, taking 19GB of hard disk space as libsvm format).

Preliminaries

MLR uses the `ml` module in Bösen. If you followed the installation instructions, this should already be built, but if not, go to `bosen/` and run

```
make ml_lib
```

Quick Start

PMLS MLR can be found in `bosen/app/mlr`. From this point on, all instructions will assume you are in `bosen/app/mlr`. After building the main PMLS libraries (as explained earlier in this manual), you can build the MLR app from `bosen/app/mlr` by running

```
make -j2
```

This will put the MLR binaries in the subdirectory `bin`.

We now turn to the run scripts. A template is provided for you:

```
cp script/launch.py.template script/launch.py

# Make it executable
chmod +x script/launch.py

# Launch it through ssh.
./script/launch.py
```

The last command launches 4 threads on local node (single node) using a subset of the [Covertype dataset](#). You should see something like

```
37 370 0.257692 0.618548 520 0.180000 50 6.85976
38 380 0.261538 0.61826 520 0.180000 50 7.03964
39 390 0.257692 0.616067 520 0.180000 50 7.22944
40 400 0.253846 0.61287 520 0.180000 50 7.41488
40 400 0.253846 0.61287 520 0.180000 50 7.43618
I0701 00:35:00.550900 9086 mlr_engine.cpp:298] Final eval: 40 400 train-0-1: 0.
↪253846 train-entropy: 0.61287 num-train-used: 520 test-0-1: 0.180000 num-test-used:
↪50 time: 7.43618
I0701 00:35:00.551867 9086 mlr_engine.cpp:425] Loss up to 40 (exclusive) is saved to
↪/home/wdai/petuum-yarn/app/mlr/out.loss in 0.000955387
I0701 00:35:00.552652 9086 mlr_sgd_solver.cpp:160] Saved weight to /home/wdai/petuum-
↪yarn/app/mlr/out.weight
I0701 00:35:00.553907 9031 mlr_main.cpp:150] MLR finished and shut down!
```

The numbers will be slightly different as it's executed indeterministically with multi-threads. The evaluations are saved to output/out.loss and model weights saved to output/out.weight.

Use HDFS

MLR supports HDFS read and output. You need to build Bösen with `HAS_HDFS = -DHAS_HADOOP` in `bosen/defs.mk`. See the [YARN/HDFS page](#) for detailed instructions.

Rebuild the binary if you rebuilt the library with Hadoop enabled (under `bosen/app/mlr`):

```
make clean all
```

Let's copy the demo data to HDFS, where `/path/to/mlr_data/` should be replaced:

```
hadoop fs -mkdir -p /path/to/mlr_data/
hadoop fs -put datasets/covtype.scale.t* /path/to/mlr_data/

# quickly verify
hadoop fs -ls /path/to/mlr_data/
```

Change a few paths in `script/launch.py`, where `<ip>:<port>` points to HDFS (You can find them on the HDFS web UI):

```
"train_file": "hdfs://<ip>:<port>/path/to/mlr_data/covtype.scale.train.small"
"trest_file": "hdfs://<ip>:<port>/path/to/mlr_data/covtype.scale.test.small"
"output_file_prefix": "hdfs://<ip>:<port>/path/to/mlr_data/out"
```

Also uncomment this line in `script/launch.py`:

```
cmd += "export CLASSPATH=`hadoop classpath --glob`:$CLASSPATH; "
```

to add hadoop path. Older hadoop might not have `hadoop classpath --glob`. You need to make sure the class path is set appropriately.

Then launch it as before:

```
./script/launch.py
# Check the result
hadoop fs -cat /path/to/mlr_data/out.loss
```

Use Yarn

We will launch job through Yarn and read/output to HDFS. Make sure you've built Yarn by running `gradle build` under `bosen/src/yarn` and have HDFS enabled in `bosen/defs.mk` like before.

Remove the outputs from previous runs:

```
hadoop fs -rm /path/to/mlr_data/out.*
```

Create run script from template

```
cp script/run_local.py.template script/run_local.py
```

In `scripts/run_local.py`, set `train_file`, `test_file`, `output_file_prefix` as previously described in the Use HDFS section. Also set the `app_dir` to the absolute path, e.g., `/path/to/bosen/app/mlr`. Then launch it:

```
chmod +x script/launch_on_yarn.py
# script/launch_on_yarn.py will call script/run_local.py
./script/launch_on_yarn.py
```

You can monitor the job progress in Yarn's WebUI. There you can also find the application ID (e.g., `application_1431548686685_0240`). You can then get the `stderr/stdout` outputs:

```
yarn logs -applicationId application_1431548686685_0240
```

There you should see similar output as before. As before, you can check the results by `hadoop fs -cat /path/to/mlr_data/out.loss`.

Data Format

MLR accepts both `libsvm` format (good for data with sparse features) and `dense binary` format (for dense features). Here we focus on `libsvm`. The `covtype` data set (`datasets/covtype.scale.train.small`) uses `libsvm` and looks like:

```
2 1:0.341596 2:0.637566 3:3.05741 4:1.32943 5:1.22779 6:0.909315 7:-2.65783 8:0.
→995448 9:1.89365 10:-0.596055 13:1 47:1
1 1:0.195157 2:0.771602 3:-0.680908 4:2.20452 5:1.22779 6:2.26125 7:-0.379156 8:1.
→24836 9:1.05749 10:2.58174 11:1 43:1
```

where the first column is the class label and the rest are `feature_id:feature_value` pairs. Each data file is associated with a meta data. For example, the `covtype` training data has meta file `datasets/covtype.scale.train.small.meta` with the following fields:

- `num_train_total`: Number of training data.
- `num_train_this_partition`: Number of training data in this partition (different from `num_train_total` if partitioned)
- `feature_dim`: Number of features.
- `num_labels`: Number of classes.
- `format`: Data format: `libsvm` or `bin`.
- `feature_one_based`: 1 if feature id starts at 1 instead of 0.
- `label_one_based`: 1 if class label starts at 1 instead of 0.
- `snappy_compressed`: 1 if the file is compressed by [Snappy](#) which often leads to 2~4x reduction in size.

Similarly, the covtype test data has meta file `datasets/covtype.scale.test.small.meta` with the following fields:

- `num_data`: Number of test data
- `num_test`: Number of data to use in test
- `feature_dim`: Number of features
- `num_labels`: Number of classes.
- `format`: Data format: `libsvm` or `bin`.
- `feature_one_based`: 1 if feature id starts at 1 instead of 0.
- `label_one_based`: 1 if class label starts at 1 instead of 0.
- `snappy_compressed`: 1 if the file is compressed by [Snappy](#) which often leads to 2~4x reduction in size.

Synthetic Data

While we are on the topic of data, let's look at the synthetic data generator which was built in previous `make`. An example script is provided for generating sparse synthetic data in `scripts/run_gen_data.py`. The parameters in the scripts are explained in the source code's flag definitions `bosen/app/mlr/src/tools/gen_data_sparse.cpp`. In particular, `num_train`, `feature_dim`, `nnz_per_col` will primarily determine the size of your data set. The generation mechanism is well documented in the header comment in the source code if you are interested in how multi-class sparse data is generated. The generator will automatically output both the data file and the associated meta data required by MLR to make it easy for MLR to consume.

You can run it with default parameters:

```
python script/run_gen_data.py
```

and find the data at `datasets/lr2_dim10_s100_nnz10.x1.libsvm.X.0`.

MLR Details

With the data in place, let's look at the input parameters for MLR in `script/launch.py` (similar set of parameters are also in `script/run_local.py` which is for Yarn.):

- Input Files:

- `train_file="covtype.scale.train.small"`: Training file. This assumes the existence of two files: `datasets/covtype.scale.train.small` and the meta file `datasets/covtype.scale.train.small.meta`.
- `test_file="covtype.scale.test.small"`: Test file, analogous to `train_file`. Optional if `perform_test=false`.
- `global_data=true`: true to let all client machines read the same file. false to use partitioned data set where each machine reads from a partition named `train_file.X`, X being the machine ID (0 to `num_machines - 1`).
- `perform_test=true`: true to perform test. `test_file` is ignored / not required if `perform_test=false`.

- Initialization:

- `use_weight_file=false`: true to continue from a previous run.
- `weight_file=`: When `use_weight_file=true`, `weight_file` is the file output by previous run, e.g. `output/mlr.covtype.scale.train.small.S0.E40.M1.T4/mlr_out.weight` generated from Quick Start above. `weight_file` is ignored if `use_weight_file=false`.

- Execution Parameters:

- `num_epochs=40`: Number of passes over the entire data set.
- `num_batches_per_epoch=300`: Number of mini-batches in each epoch. We clock the parameter server at the end of each mini-batch.
- `learning_rate=0.01`: Learning rate for gradient in stochastic gradient descent.
- `decay_rate=0.95`: We use multiplicative decay, i.e., learning rate at epoch `t` is `learning_rate*decay_rate^t`.
- `num_batches_per_eval=300`: Evaluate (approximately) the training (and test error if `perform_test=true`) every `num_batches_per_eval`. Usually set to `num_batches_per_epoch` to evaluate after each epoch.
- `num_train_eval=10000`: Number of training examples used to evaluate the intermediate training error. The final evaluation will use all training example.
- `num_test_eval=20`: Number of test examples used to evaluate the approximate training error.
- `lambda=0`: L2 regularization parameter
- `init_lr` and `lr_decay_rate`: Learning rate is `init_lr*lr_decay_rate^T` where `T` is the epoch number.

- System Parameters: <<<<<< HEAD

- `hostfile="scripts/localserver"`: Machine file. See Configuration page for more details.
=====
- `hostfile="scripts/localserver"`: Machine file. See *Configuration Files for PMLS Apps*

```
45f3aef19b7152701f51e72148f0f2008fe1e7c2      *
num_app_threads=4:      Number of applica-
tion worker threads.      * staleness=0:  Stal-
eness for the weight table (the main table).      *
num_comm_channels_per_client=1: The number
of threads running server and back ground communication.
Usually 1~2 is good enough.
```

Terminating the MLR app

The MLR app runs in the background, and outputs its progress to standard error. If you need to terminate the app before it finishes, just run

```
python scripts/kill_mlr.py <petuum_ps_hostfile>
```

Foreword - please read

Parallel ML System (PMLS) is a distributed machine learning framework. It takes care of the difficult system “plumbing work”, allowing you to focus on the ML. PMLS runs efficiently at scale on research clusters and cloud compute like Amazon EC2 and Google GCE.

The PMLS project is organized into 4 open-source (BSD 3-clause license) Github repositories:

- Bösen (C++ bounded-async key-value store)
- Strads (C++ model-parallel scheduler)
- JBösen (Java bounded-async key-value store)
- PMLS-Caffe (Deep Learning framework)

To install Bösen and Strads, please continue reading this manual. If you have a Java environment and want to use JBösen, please start [here](#). If you wish to use PMLS-Caffe for Deep Learning, please go [here](#).

PMLS Bösen/Strads v1.1 manual

1. *Quickstart*
 - (a) *Detailed Installation Instructions*
 - (b) *Configuration and Machine Files for PMLS Apps*
 - (c) *Running on Hadoop clusters with YARN/HDFS*
 - (d) *Frequently Asked Questions*
2. ML Applications
 - (a) Topic Models
 - i. *Latent Dirichlet Allocation (topic modeling)*

- ii. *MedLDA (supervised topic modeling)*
- (b) Deep Learning
 - i. PMLS-Caffe: Distributed Deep Learning Framework on PMLS
 - ii. *General-purpose Deep Neural Network (DNN)*
 - A. *DNN for Speech Recognition*
- (c) Matrix Factorization and Sparse Coding
 - i. *Matrix Factorization (collaborative filtering)*
 - ii. *Non-negative Matrix Factorization (NMF)*
 - iii. *Sparse Coding*
- (d) Regression
 - i. *Lasso Regression*
- (e) Metric Learning
 - i. *Distance Metric Learning*
- (f) Clustering
 - i. *K-means Clustering*
- (g) Classification
 - i. *Random Forest*
 - ii. *Logistic Regression*
 - iii. *SVM* (Newly added in v1.1)
 - iv. *Multi-class Logistic Regression*
- 3. Programming API
 - (a) Bosen Bounded-Async Key-Value Store
 - (b) *Strads Model-Parallel Scheduler* (Coming soon)

Introduction to PMLS

PMLS is a distributed machine learning framework. It takes care of the difficult system “plumbing work”, allowing you to focus on the ML. PMLS runs efficiently at scale on research clusters and cloud compute like Amazon EC2 and Google GCE.

PMLS provides essential distributed programming tools to tackle the challenges of ML at scale: **Big Data** (many data samples), and **Big Models** (very large parameter and intermediate variable spaces). To address these challenges, PMLS provides two key platforms:

- Bösen, a bounded-asynchronous key-value store for Data-Parallel ML algorithms
- Strads, a scheduler for Model-Parallel ML algorithms

Unlike general-purpose distributed programming platforms, PMLS is designed specifically for ML algorithms. This means that PMLS takes advantage of data correlation, staleness, and other statistical properties to maximize the performance for ML algorithms.

ML programs are built around update functions that are iterated repeatedly until convergence, as the following diagram illustrates:

The update function takes the data and model parameters as input, and outputs a change to the model parameters. **Data parallelism** divides the data among different workers, whereas **model parallelism** divides the parameters among different workers. Both styles of parallelism can be found in modern ML algorithms: for example, Sparse Coding via Stochastic Gradient Descent is a data-parallel algorithm, while Lasso regression via Coordinate Descent is a model-parallel algorithm. The PMLS Bösen and Strads systems are built to enable data-parallel and model-parallel styles, respectively.

Key PMLS features

- Runs on compute clusters and cloud compute, supporting up to 100s of machines
- **Bösen**, a bounded-asynchronous distributed key-value store for data-parallel ML programming
 - Bösen uses the Stale Synchronous Parallel consistency model, which allows asynchronous-like performance that outperforms MapReduce and bulk synchronous execution, yet does not sacrifice ML algorithm correctness
- **Strads**, a dynamic scheduler for model-parallel ML programming
 - Strads performs fine-grained scheduling of ML update operations, prioritizing computation on the parts of the ML program that need it most, while avoiding unsafe parallel operations that could hurt performance
- Programming interfaces for C++ and Java
- YARN and HDFS support, allowing execution on Hadoop clusters
- **ML library with 10+ ready-to-run algorithms**
 - Newer algorithms such as discriminative topic models, deep learning, distance metric learning and sparse coding
 - Classic algorithms such as logistic regression, k-means, and random forest

Support and Bug reports

For support, or to report a bug, please send email to pmls-support@googlegroups.com. Please provide your name and affiliation; we do not support anonymous inquiries.