# Page: Wiki Home

# Petuum v0.9 manual



# Introduction to Petuum

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

Petuum 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, Petuum provides the following tools:

* Distributed parameter server (i.e. key-value storage)
* Distributed model scheduler (STRADS)
* Out-of-core (disk) storage for limited-memory situations

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

## Ready-to-run ML applications

In addition to distributed ML programming tools, Petuum comes with distributed ML algorithms that not only serve to demo Petuum’s capabilities, but can also be used at massive scale for real Big Data analytics. All programs are implemented on top of the Petuum framework for speed and scalability. Currently, Petuum includes:

* Latent Dirichlet Allocation (a.k.a Topic Modeling)

* Least-squares Matrix Factorization (a.k.a Collaborative Filtering)
* Lasso regression and Logistic Regression
* Deep Neural Network

Future releases will include even more ML applications.

## What does "Petuum" mean?

Petuum comes from "perpetuum mobile," which is a musical style characterized by a continuous steady stream of notes. [Paganini's Moto Perpetuo](https://www.youtube.com/watch?feature=player_detailpage&v=dPRWshWq9E4#t=3) is an excellent example. It is our goal to build a system that runs efficiently and reliably -- in perpetual motion.

# Page: Installation

# Which Linux distro to use?

Petuum has been tested on **64-bit Ubuntu Desktop 14.04 and 12.04** (available at: http://www.ubuntu.com/download/desktop). The instructions on this guide assume a fresh installation of either Ubuntu 14.04 or 12.04.

We have also successfully tested Petuum on some versions of RedHat and CentOS. However, the commands provided in this manual for installing dependencies are specific to 64-bit Ubuntu Desktop 14.04 and 12.04. They do not apply to RedHat/CentOS; you will need to know how to install them yourself.

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

# Obtaining Petuum

You can download Petuum as a zip file from GitHub: navigate to http://github.com/sailinglab/petuum, and click on the “Download Zip” link on the right (or use `wget` to download it on Linux).

Alternatively, you can install `git` and clone the Petuum repository from GitHub. You will need to set up a GitHub account and upload a public SSH key. Once done, enter the following commands to install `git` and clone the Petuum repository into the current directory:

```
sudo apt-get update
sudo apt-get install git
git clone git@github.com:sailinglab/petuum.git
```

# Necessary packages
Petuum requires the following packages:
* `g++` version 4.8 (`clang` is known to work, but we do not officially support it)
* `autoconf` version 2.68
* `libtool`
* `uuid-dev` (libuuid)
* `openssh-server`
* `cmake`
* `libopenmpi-dev`
* `openmpi-bin`

* `libssl-dev`

We provide commands for installing these packages on 64-bit Ubuntu Desktop 14.04 and 12.04:

## Ubuntu 14.04

All packages can be easily installed through `apt-get`:

```
sudo apt-get update
sudo apt-get install g++ autoconf libtool uuid-dev openssh-server cmake libopenmpi-dev
openmpi-bin libssl-dev
```

## Ubuntu 12.04

Ubuntu 12.04's default package repository does not have `g++` 4.8, so you will need to add an external repository to install it:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-4.8 g++-4.8
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 50
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 50
```

Once you have installed `g++` 4.8, you can install the remaining packages the same way as Ubuntu 14.04:

```
sudo apt-get update
sudo apt-get install autoconf libtool uuid-dev openssh-server cmake libopenmpi-dev
openmpi-bin libssl-dev
```

# Compiling Petuum

You’re now ready to compile Petuum. This requires a working internet connection, as our make system will download additional dependencies. In the Petuum root directory, just type

```
```

make
```

This process will take between 5-30 minutes, depending on your machine and the speed of your internet connection. We'll explain how to compile and run Petuum's built-in apps later in this manual, as well as how to write your own applications.

**Note:** If the build process fails for any reason (e.g. your internet connection goes down while the make system is downloading dependencies), we recommend performing a full-clean of Petuum and all downloaded dependencies via `make distclean`, before you attempt to recompile with `make`. Note that the usual `make clean` only deletes the Petuum libraries, but not the dependencies (so if the dependencies are corrupt, `make` will still fail).

# Very important: Setting up password-less SSH authentication

Petuum 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 (Petuum 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 the 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 Petuum in an cluster environment with a shared filesystem** (e.g. shared home directories). Provided all machines are identically configured and have the necessary packages/libraries, you only need to compile Petuum (and any apps you want to use) once, from one machine. The Petuum 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, you can still use Petuum, but 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. **Petuum 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` Petuum onto every machine, **at exactly the same path** (e.g. /home/username/petuum). **You must compile Petuum and the Petuum apps separately on each machine.**
3. When running Petuum 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
* Petuum apps: 9999 and 10000 by default (you can change these)

# Cloud compute support
Petuum should run in any Linux-based cloud environment that supports SSH; we recommend using 64-bit Ubuntu 14.04. If you wish to run Petuum 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 and 10000 (or whatever ports you intend to use). See https://developers.google.com/compute/docs/troubleshooting#knownissues for more info.

# Page: ML App Preliminaries

# Make sure password-less SSH is set up correctly

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

**You must be able to `ssh` into all machines without a password - even if you are only using your local machine.** The Petuum 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 Petuum apps.** For example, if you are going to run Petuum on your local machine, make sure that `ssh 127.0.0.1` logs you in without asking for a password. See [[Installing Petuum|Installation]] for instructions on how to do this.

# Parameter Server configuration files

Many Petuum ML applications require Parameter Server configuration files, in the following format:

```
0 ip_address_0 10000
1 ip_address_0 9999
1000 ip_address_1 9999
2000 ip_address_2 9999
3000 ip_address_3 9999
...
```

You can give your configuration file whatever filename you want (you will pass the filename as an argument to the ML app). 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 port number (9999 or 10000). Every machine is assigned to one ID and one port, except for the first machine, which is assigned two IDs and two ports because it has a special role.

## What if I want to run the ML apps on just my local machine?

Simply create this localhost configuration file:

```
0 127.0.0.1 9999
1 127.0.0.1 10000
```

## Caution - Please Read!

**You cannot `crtl-c` to terminate a Parameter Server app**, because they are invoked in the background via `ssh`. Each PS 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 Petuum apps on the same machines, make sure you give them separate Parameter Server configuration files with different ports. **The apps cannot share the same ports!**

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) Petuum app using the same ports. In that case, you should use the offending app's kill script to terminate it.

# Page: ML App: Least-Squares Matrix Factorization (Collaborative Filtering)

# Least-squares Matrix Factorization

Given an input matrix `X` (with some missing entries), the Matrix Factorization app (MF for short) learns two matrices `L` and `R` such that `L*R` is approximately equal to `X` (except where elements of `X` are missing).

If `X` is N-by-M, then `L` will be N-by-K and `R` 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 result in a more accurate factorization, but at the cost of additional computation.

MF is commonly used to perform Collaborative Filtering, where `X` represents the known relationships between two categories of things - for example, `X(i,j) = v` might mean that “person i gave product j rating v”. Because we might not know all possible relationships in `X`, we can use MF to predict the unknown relationships. Specifically, we can predict any missing entry `X(i,j)`, using the learnt matrices `L` and `R`:

`X(i,j) = L(i,1)*R(1,i) + L(i,2)*R(2,i) + ... + L(i,K)*R(K,i)`

Our MF app uses the Stochastic Gradient Descent (SGD) algorithm to learn the matrices `L`, `R`, with optional L2 regularization.

# Quick start

The MF app can be found in `apps/matrixfact/`. **From this point on, all instructions will assume you are in `apps/matrixfact/`.** After building the main Petuum libraries (as explained earlier in this manual), you can build the MF app from `apps/matrixfact` by running

```
make
```

This will put the MF binary in the subdirectory `bin/`. You can test that the app works on the local machine (with 4 worker threads) using:

```
scripts/run_matrixfact.sh sampledata/9x9_3blocks 3 100 mf_output scripts/localserver 4
```

You should get 2 output files:

```
mf_output.L
mf_output.R
```

In both files, you should see that lines 1-3 are similar, lines 4-6 are similar, and lines 7-9 are similar. For example, `mf_output.L` might look something like this:

```
0.308813 0.963073 -0.555122
0.308813 0.963073 -0.555122
0.308813 0.963073 -0.555122
1.29682 -0.362354 0.602023
1.29682 -0.362354 0.602023
1.29682 -0.362354 0.602023
```

You won't see the exact same numeric values in different runs --- that's expected behavior, since there are multiple ways to factorize `X = L*R`. All you need to confirm is that each group of 3 lines is similar.

# Input matrix format

The MF app takes a sparse matrix as input:

```
row_0 col_0 value_0
row_1 col_1 value_1
row_2 col_2 value_2
...
```

This says that the matrix has elements `(row_0,col_0) = value_0`, `(row_1,col_1) = value_1`, and so on. The row and column indices are 0-indexed (meaning that the upper left corner is 0,0). If a matrix element is not explicitly specified, it is treated as a missing value, which is **not** the same as a zero value. In the Collaborative Filtering task, missing values are those that you want to predict, e.g. "How much will user x like product y?"

## Creating synthetic data

We provide a synthetic data generator for testing purposes (requires python 2.7+). To see detailed instructions, run

```
python sampledata/make_synth_data.py
```

For example, you can create a 1000-by-1000 matrix `data` by running

```
python sampledata/make_synth_data.py 10 100 0.1 data
```

# Running the Matrix Factorization application

To see the instructions for the MF app, run

```
scripts/run_matrixfact.sh
```

The basic syntax is

```
scripts/run_matrixfact.sh <datafile> <K> <iters> <output_prefix> <petuum_ps_hostfile>
<client_worker_threads> "additional options"
```

* `<datafile>`: path to input data file
* `<K>`: factorization rank
* `<iters>`: number of iterations to run; one iteration = touch every matrix element once
* `<output_prefix>`: the factor matrices L and R will be written to `<output_prefix>.L`, `<output_prefix>.R`
* `<petuum_ps_hostfile>`: Petuum server configuration file, as described earlier in this manual
* `<client_worker_threads>`: how many worker threads to use on each machine

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:
* `--staleness x`: turn on Stale Synchronous Parallel (SSP) consistency at staleness level `x`; often improves performance when using many machines
* `--lambda x`: sets the L2 regularization strength to `x`; default is 0
* `--offsetsfile x`: used to provide an "offsets file" for limited-memory situations; this will be explained in a later section

* `--init_step_size x`, `--step_size_offset y`, `--step_size_pow z`: used to control the SGD step size. The step size at iteration `t` is `x * (y+t)^(-z)`. Default values are `x=0.5`, `y=100`, `z=0.5`.
* `--ps_row_cache_size x`: controls the cache size of each worker machine. By default, the MF app caches the whole `L`, `R` matrices for maximum performance, but this means every machine must have enough memory to hold a full copy of `L` and `R`. If you are short on memory, set `x` to the maximum number of `L` rows and `R` columns you wish to cache. For example, `--ps_row_cache_size 100` forces every client to only cache 100 rows of `L` and 100 columns of `R`.

For example, to use staleness 5 and lambda 0.1 in the Quick Start example, run

```
scripts/run_matrixfact.sh sampledata/9x9_3blocks 3 100 mf_output scripts/localserver 4
"--staleness 5 --lambda 0.1"
```

**Note:** If the objective function shows `nan` (not a number), it usually means the step size is too large. Divide `--init_step_size` by 10, and try again. You may need to do this several times.

## Terminating the MF app

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

```
scripts/kill_matrixfact.sh <petuum_ps_hostfile>
```

# Output format

The MF app outputs the two factor matrices `L`, `R` to `<output_prefix>.L`, `<output_prefix>.R`. These are whitespace-separated arrays of floating-point values:
* `<output_path>.L` will contain N lines, with K numbers on each line
* `<output_path>.R` will contain M lines, with K numbers on each line (it is the transpose of R)

# Handling large input data

By default, the MF app loads the whole input matrix into memory for maximum speed. If memory is limited, you can make the MF app read the matrix off disk. This requires some preprocessing; run the following command for instructions:

```
python scripts/matrixfact_data_partitioner.py
```

The basic syntax is

```
python scripts/matrixfact_data_partitioner.py <entries-per-block> <input-matrix-file>
<output-offsets-file>
```
* `<entries-per-block>`: number of matrix entries to assign to each block; we suggest 1000000 (1 million)
* `<input-matrix-file>`: the input matrix file to be processed
* `<output-offsets-file>`: the output offsets file

**Note:** the total number of blocks (number of lines in `<input-matrix-file>` divided by `<entries-per-block>`) must be larger than `<client_worker_threads>` multiplied by the total number of machines (i.e. the total number of worker threads across the cluster), or the MF app will fail.

Once you have created `<output-offsets-file>`, you just need to provide it to the MF app using `"--offsetsfile <output-offsets-file>"`. As an example, please try the following command:

```
scripts/run_matrixfact.sh sampledata/9x9_3blocks 3 100 mf_output scripts/localserver 4
"--offsetsfile sampledata/9x9_3blocks.offsets"
```

We provide the file `sampledata/9x9_3blocks.offsets`, but you can also generate it yourself by running

```
python scripts/matrixfact_data_partitioner.py 9 sampledata/9x9_3blocks
sampledata/9x9_3blocks.offsets
```

where we have used `<entries-per-block> = 9`.

# Special features

The following features provide additional functionality, but may slow down execution.

## Snapshots and failure recovery

The MF app supports snapshots and failure recovery via the following `"additional options"`:
* `--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`!

You can try this with the Quick Start example:

```
mkdir snap
scripts/run_matrixfact.sh sampledata/9x9_3blocks 3 100 mf_output scripts/localserver 4
"--ps_snapshot_clock 10 --ps_snapshot_dir snap"
```

This takes snapshots every 10th iteration, and stores them in `snap/`. To resume from iteration 50 in `snap/`, run

```
mkdir snap2
scripts/run_matrixfact.sh sampledata/9x9_3blocks 3 100 mf_output scripts/localserver 4
"--ps_snapshot_clock 10 --ps_snapshot_dir snap2 --ps_resume_clock 50 --ps_resume_dir
snap"
```

Note that this will produce a new series of snapshots in `snap2/`.

**Note:** Do not reuse `--ps_snapshot_dir` between app executions. Either delete everything in the directory, or specify a new one. The current snapshot implementation is not guaranteed to work if existing snapshots are present in `--ps_snapshot_dir`.

## Limited-memory single-machine MF (out-of-core Parameter Server)

If you only have one machine with limited memory, you may wish to try the single-machine PS version of the MF app. This version has out-of-core support for the factor matrices `L`, `R`, meaning that it can store them on disk rather than in memory (but at the cost of performance). To see the instructions, run

```
scripts/run_matrixfact_sn.sh
```

```

The syntax is almost identical to the regular MF app:

```
scripts/run_matrixfact_sn.sh <datafile> <K> <iters> <output_prefix>
<ps_row_in_memory_limit> <client_worker_threads> "additional options"
```

In place of `<petuum_ps_hostfile>`, you instead provide an integer `<ps_row_in_memory_limit>` that controls how many rows of `L`, `R` can be cached in memory. For example, if `<ps_row_in_memory_limit> = 5`, then at most 5 rows of `L` and 5 rows of `R` can be cached in memory, while the full matrices are stored on disk. The more rows you can afford to cache in memory, the faster the app will run.

Here is an example based off the Quick Start:

```
scripts/run_matrixfact_sn.sh sampledata/9x9_3blocks 3 100 mf_output 6 4
```

In this example, we have only allowed 6 (out of 9) rows of `L`, `R` to be held in memory.

**Note 1:** The single machine MF is optional. You can always use the normal MF app with just one machine.

**Note 2:** The single machine MF creates a temporary directory `<output_prefix>.matrixfact_sn.localooc` in order to store `L`, `R` on disk. Do not delete this directory while the program is running. You may delete it afterwards.

**Note 3:** The single machine MF also supports `--offsetsfile`. This way, you can keep both the input matrix and the factors `L`, `R` on disk, without worrying about memory limits.

**Note 4:** The single machine MF does not support snapshots and recovery. `--ps_snapshot_clock` and `--ps_resume_clock` have absolutely no effect.

**Note 5:** The `--ps_row_cache_size` option has no effect on the single machine MF. Instead, `<ps_row_in_memory_limit>` controls the memory usage.
```

# Page: ML App: Latent Dirichlet Allocation (topic modeling)

# Latent Dirichlet Allocation (LDA)

Topic model, a.k.a Latent Dirichlet Allocation (LDA), is an algorithm that discovers latent semantic structure in a set of documents. Specifically, LDA finds topics, defined as weighted set of vocabularies, and the topics of each document from a document collection. Here we are solving LDA using Gibbs sampling. We partition the documents onto different machines, and make word-topic counts globally accessible to all workers by storing it as a PS table (where the row id is the vocab ID). Note that in this scheme the document-topic counts are stored locally instead of in PS. Our implementation uses the sparse sampling procedure in Yao et al (2009).

# Quick start

The LDA app can be found in `apps/lda/`. **From this point on, all instructions will assume you are in `apps/lda/`.** After building the main Petuum libraries (as explained earlier in this manual), you can build the LDA app from `apps/lda` by running

```
# Assuming you have 2 cores, this parallelizes the build
make -j2
```

This will put the LDA binary in the subdirectory `bin/`. You can test that the app works on the local machine using:

```
sh scripts/run_lda.sh
```

This runs LDA on [20 Newsgroups dataset](http://qwone.com/~jason/20Newsgroups/) with 100 topics for 5 iterations using 1 (worker) thread. The printout should end with something like

```
1 -1.79782e+07 7.06098
2 -1.70722e+07 12.2925
3 -1.6426e+07 17.138
4 -1.58477e+07 21.7576
5 -1.5303e+07 26.0027
```

```
I0629 18:34:52.677211 17623 lda_main.cpp:189] LDA finished!
I0629 18:34:52.735905 17623 lda_main.cpp:191] LDA shut down!
```

The three columns are: `iteration-number` `log-likelihood` `time so far in seconds`. You should get similar log-likelihood, but the time will depends on your machine spec. The log likelihood outputs is also stored in `output/lda.S0.M1.T1/lda_out.llh`. Note that the log-likelihood is still increasing and hasn't converged in 5 iterations. It should converge to around `-1.24e+07` when running 100 iterations (with 100 topics, and both hyperparameters of LDA set to 0.1).

To get the topic vocabularies and topic vectors, please see "Execution Parameters" under "Running the LDA Application" below.

# Data Format and Processing

`datasets/20news.dat` is an example of raw data. It looks like this:

```
0 0:4 2:10 3:4 4:2 6:1 7:1 8:3 9:9 10:2 12:1 13:4 14:2 16:6 17:2 ...
0 38:2 233:1 579:1 611:1 769:1 770:1 771:2 772:2 773:2 775:1 776:1 777:1 ...
...
```

Each line is a document. The first column is the document label, which is ignored as LDA is an unsupervised algorithm. Second column onward are `word:count` pairs. For example `0:4` means vocabulary 0 occurs 4 times in that document.

We then convert this data to store in [LevelDB](http://leveldb.googlecode.com/svn/trunk/doc/index.html). LevelDB is a disk-based key-value store that's pretty easy to use. The processing code is in `src/data_preprocessor.cpp`. You can use the following script:

```
# This prints the usage message
sh scripts/run_lda_processor.sh

# Creating two partitions from 20 newsgroup dataset.
# Naming convention: 20news2 means 20news in 2 partitions.
sh scripts/run_lda_processor.sh datasets/20news.dat datasets/20news2 2
```

This partitions 20 newsgroup dataset into two partitions: `datasets/20news2.0` `datasets/20news2.1`. They are backed up at `datasets/20news2.0.bak`,

`datasets/20news2.1.bak` which are useful in the disk-streaming use case (see below). There's also associated meta-data: `datasets/20news2.0.meta`, `datasets/20news2.1.meta`. Each process reads from one partition, so create as many partitions as the number of machines you have.

20 newsgroup dataset has ~11k documents, ~53k vocabularies, and 1.3M tokens. We are also providing two bigger datasets:

*
[NYTimes](https://docs.google.com/uc?id=0BxRiAB3QAruLZmtQSVctamZkNVE&export=download): Originally from [UCI's repository](https://archive.ics.uci.edu/ml/datasets/Bag+of+Words). The NYT corpus has about 300k documents, 100k vocabularies, and 100M tokens.

*
[Pubmed](https://drive.google.com/uc?id=0BxRiAB3QAruLVGM2RE9idU5vVnM&export=download): Also from UCI's repository. The Pubmed corpus has about 8.2M documents, 141k vocabularies, and 740M tokens.

Decompress the dataset with `tar jxf nytimes.tar.bz2` (note that the size would increase by ~5x). The downloaded dataset is in the same format as 20news.dat, so same processing procedure applies. Since they both very sizable (especially Pubmed), you will need hundreds of cores to run these datasets to finish reasonable amount of time.

# Running the LDA Application

There are many parameters involved for running LDA. We provide default values for all of them in `scripts/run_lda.sh`. Some important ones are:

- Input files
  * `doc_filename="datasets/20news1"`: This tells the client machine 0 to read `datasets/20news1.0` (and client machine 1 to read `datasets/20news1.1` etc, if applicable.).
  * `host_filename="scripts/localserver"`: The server file. See [ML App Preliminaries](ML-App-Preliminaries#parameter-server-configuration-files).

- LDA parameters
  * `num_topics=100`: Number of topics in LDA.
  * `alpha=0.1`: The Dirichlet prior for document-topic vectors.
  * `beta=0.1`: The Dirichlet prior for word-topic vectors.

- Execution parameters
  * `num_work_units=5`: Number of work units. Each unit consists of one or more iterations, as defined in `num_iters_per_work_unit`.

    * `num_iters_per_work_unit=1`: Number of iterations per work unit. Has to be integer. Default to 1 iteration per work unit.
    * `num_clocks_per_work_unit=1`: Number of clocks per work unit. Higher values generally uses more bandwidth. Tune this in conjunction with `num_iters_per_work_unit`
    * `compute_ll_interval=1`: Compute log-likelihood every `compute_ll_interval` work units. `-1` turns off log-likelihood computation. The log-likelihood computation is usually pretty fast compared with the sampling time.
    * `output_topic_word=false`: Set to true to output the topic word estimates ("phi") to output_file_prefix.phi. Every line represents one topic.
    * `output_doc_topic=false`: Set to true to output the doc topic estimates ("theta") to output_file_prefix.theta.X where X is the client ID. Every line represents one document.

- System parameters
    * `client_worker_threads=1`: Number of threads running LDA.
    * `staleness=0`: The staleness for all tables.
    * `word_topic_table_process_cache_capacity=-1`: Process cache size. -1 caches all vocabs in word_topic_table. Don't change it unless you don't have enough memory.
    * `disk_stream=false`: If your dataset cannot fit in memory, set it to true to stream from disk. Currently there's about 2x speed penalty.

#### Terminating the LDA app

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

```
sh scripts/kill_lda.sh <petuum_ps_hostfile>
```

#### Handling Large Data

If you run out of memory, you can have some combination of the following two options:

1. Adjust `word_topic_table_process_cache_capacity` to less than `num_vocabs` (which are available in the meta data files generated from the processing step). This limit the cache for word-topic table but the system would fetch more parameters over the network.
2. Use disk streaming by setting `disk_stream=true`.  There's currently about 2x runtime penalty for using disk stream. If you are using disk streaming you can tune `disk_stream_batch_size` in `src/lda_main.cpp`. The default batch size of 1000 docs usually works well enough.

# Limited-memory single-machine LDA (out-of-core Parameter Server)

If you only have one machine with limited memory, you may wish to try the single-machine PS version of the LDA app. This version has out-of-core support for word-topic table, meaning that it can store them partially on disk rather than all in memory (but at the cost of performance). After `make`, just run

```
sh scripts/run_lda_sn.sh
```

The single-machine LDA is faster than the distributed LDA running on one machine (due to some system optimizations in the parameter). For example a run on 20 newsgroup, comparable to the one in Quick Start above takes ~14 seconds for 5 iterations, compared with 26 seconds for distributed LDA. Note that the code for LDA is the same. The only difference is in the PS.

#### Important Parameters

The run script for single-machine LDA `scripts/run_lda_sn.sh` is very similar to `scripts/run_lda.sh`. All of them are covered in the parameter descriptions above except:

* `ps_row_in_memory_limit=100000`: Number of rows in word-topic table to store in memory. The rest will go to disk. Default value stores all of the word-topic table in memory for 20 newsgroup.

**Note**: A new leveldb database is created (e.g., `output/lda_sn.S0.T1/lda_sn_out.lda_sn.localooc`) as single-machine PS uses leveldb to store table out-of-core. You can delete it after the program finishes.

# Page: ML App: Deep Neural Network

# Deep Neural Network

In this app, we implemented Deep Neural Network (DNN) for multi-class classification. 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.

# Quick start

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

```
make
```

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

Create a simulated dataset:
```
scripts/gen_data.sh 10000 440 1993 3 datasets
```
then you can test that the app works on the local machine (with 4 worker threads) using:

```
scripts/run_dnn.sh 4 5 machinefiles/localserver datasets/para_imnet.txt
datasets/data_ptt_file.txt weights.txt biases.txt
```

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

```
weights.txt
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.

`biases.txt` saves the bias vectors. The order is: bias vector on layer 2 (the first hidden layer), bias vector on layer 3, etc. Each line is a bias vector.

# 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,...,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. It will generate a dataset where the number of training examples is 10000, feature dimension is 440 and number of classes are 1993. Moreover, it will partition the dataset into 3 pieces and save the generated files to `datasets`.
```
scripts/gen_data.sh 10000 440 1993 3 datasets
```
In the directory `datasets`, you can find three data files 0_data.txt, 1_data.txt, 2_data.txt and a `data_ptt_file.txt` file in which you can find three lines:
```
/home/pengtaox/petuum_junrel/petuum/apps/dnn/datasets/0_data.txt    3333
/home/pengtaox/petuum_junrel/petuum/apps/dnn/datasets/1_data.txt    3333
/home/pengtaox/petuum_junrel/petuum/apps/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

To see the instructions for the DNN 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_weight_file> <model_bias_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_weight_file>`: the path where the output weight matrices will be stored
* `<model_bias_file>`: the path there the output bias vectors 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_ptt_file.txt`, weight matrices path is `weights.txt`, bias vectors path is `biases.txt`, use the following command:
```
scripts/run_dnn.sh 4 5 machinefiles/localserver datasets/para_imnet.txt
datasets/data_ptt_file.txt weights.txt biases.txt
```

## 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: 440 512 512 512 512 1993
num_epochs: 2
stepsize: 0.05
mini_batch_size: 10
num_smp_evaluate: 2000
num_iters_evaluate: 100
```
## Single Machine Version DNN
We also provide a single machine DNN, whose usage is almost the same as the distributed DNN.
To see the instructions for the single machine DNN app, run

```
scripts/run_dnn_sn.sh
```

The basic syntax is

```
scripts/run_dnn_sn.sh   <num_worker_threads> <staleness> <parameter_file> <data_partition_file> <model_weight_file> <model_bias_file> "additional options"
```
The parameters are nearly the same as those in distributed DNN, except in "additional options", you can add
```
--ps_row_in_memory_limit <ps_row_in_memory_limit_value>
```
which is the number of rows of weight matrices to store in memory. The rest will go to disk.
Here is an example based off the Quick Start:
```
```

scripts/run_dnn_sn.sh 4 5 datasets/para_imnet.txt datasets/data_ptt_file.txt weights.txt biases.txt
```
# Output format
The app outputs two files:
```
<model_weight_file>
<model_bias_file>
```

`<model_weight_file>` 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. Elements in each row are separated with blank.

`<model_bias_file>` saves the bias vectors. The order is: bias vector on layer 2 (the first hidden layer), bias vector on layer 3, etc. Each line is a bias vector. Elements in each vector 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 <petuum_ps_hostfile>
```
for single machine version, run
```
scripts/kill_dnn_sn.sh
```
```

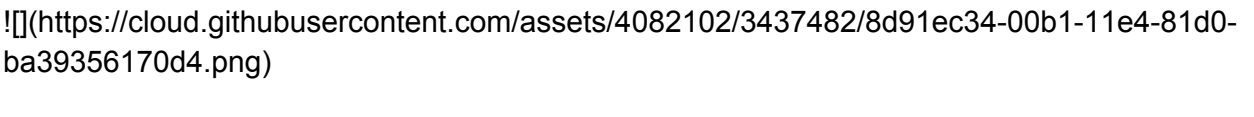
# Page: ML App: Lasso LR

Lasso/Logistic Regression INSTALL
==========
Build Requirements
---
- OS: Ubuntu 12.04LTS / Ubuntu 14.04LTS
- cmake 2.8.7

Build instructions
---
Assume that your current working directory is STRADS root directory `src/strads`
- Building Lasso
```` sh
  mkdir build
  mkdir build/lasso
  cd build/lasso
  cmake -D APP_COMPILE:string=LASSO -D CMAKE_CXX_COMPILER=g++ ../../
  make
```
- Building Logistic Regression
```` sh
  mkdir build
  mkdir build/logistic
  cd build/logistic
  cmake -D APP_COMPILE:string=LOGI -D CMAKE_CXX_COMPILER=g++ ../../
  make
```

Lasso
==========

Given a design matrix X (possibly with some missing entries) and a response vector Y (no missing entries in Y), the Lasso app estimates one vector (regression coefficient vector) BETA, where coefficients corresponding to the features relevant to Y become nonzero. If X is N-by-M and Y is N-by-1, then BETA will be M-by-1.  Lasso is commonly used to perform feature selection, where X represents the M features of N samples, and Y represents an output of interest for the same N samples. The Lasso algorithm optimizes the following objective function:



where lambda parameter determines the level of sparsity of BETA vector. The larger the lambda, the larger number of zero coefficients in BETA. With Petuum STRADS, we implement parallel distributed coordinate descent based lasso. STRADS schedules model parameters to update in parallel while control degree of dependency among model parameters. In each dispatch, STRADS do weight-based sampling so that the selected model parameters are to make more contribution on the convergence speed than randomly selected model parameters.

Data Format
---
STRADS assume that Input Data is stored in the form of Matrix Market Exchange Format called MMT format (For details, refer to http://math.nist.gov/MatrixMarket/formats.html). For fast data uploading, it is required to convert MMT format into STRADS binary format. For the convenience of conversion, file converter from MMT format to STRADS binary format is provided.

- Building of Data Converter

  Assume that your current working directory is STRADS root directory

```sh
   cd tool/mmt2bin
   make
   mmt2bin <input_mmt_file> <output_stradsfile_name>
```

Here is a simple example of an MMT-formatted file, representing a 3-by-3 matrix with 5 nonzero entries:

```
%%MatrixMarket matrix coordinate real general
% First line: rows cols entries
% Subsequent lines = entries: i j val (1-indexed)
3 3 5
1 1 1.0
2 2 3.0
3 3 5.0
1 3 2.0
2 3 3.0
```

Run Lasso Sample on single machine

---
For demonstration purpose, we put Lasso script file together with small sized input files. With assumption that Lasso is built according to the building instruction and your current working directory is the STRADS root directory `src/strads`. Run the following commands.

```sh
   cd pyscript
   python singlelasso.py
```

It will run STRADS Lasso application with one machine creating three processes. Results will be similar the following. Due to the stochastic property of scheduling, your results might be slightly different.

```sh
1000 iter  Elaptime(0.845536) sec	Object 0.001631 nz(4930)
2000 iter  Elaptime(1.700898) sec	Object 0.001600 nz(4664)
3000 iter  Elaptime(2.513912) sec	Object 0.001575 nz(4520)
4000 iter  Elaptime(3.328590) sec	Object 0.001554 nz(4421)
5000 iter  Elaptime(4.160760) sec	Object 0.001534 nz(4404)
6000 iter  Elaptime(4.990915) sec	Object 0.001517 nz(4381)
7000 iter  Elaptime(5.814701) sec	Object 0.001501 nz(4379)
8000 iter  Elaptime(6.637410) sec	Object 0.001485 nz(4388)
9000 iter  Elaptime(7.462466) sec	Object 0.001471 nz(4387)
10000 iter  Elaptime(8.292242) sec	Object 0.001457 nz(4358)
11000 iter  Elaptime(9.113457) sec	Object 0.001445 nz(4328)
12000 iter  Elaptime(9.910599) sec	Object 0.001433 nz(4294)
13000 iter  Elaptime(10.709856) sec	Object 0.001421 nz(4275)
14000 iter  Elaptime(11.459291) sec	Object 0.001410 nz(4323)
15000 iter  Elaptime(12.187501) sec	Object 0.001399 nz(4309)
16000 iter  Elaptime(12.917369) sec	Object 0.001389 nz(4277)
17000 iter  Elaptime(13.640178) sec	Object 0.001378 nz(4268)
18000 iter  Elaptime(14.364920) sec	Object 0.001368 nz(4228)
19000 iter  Elaptime(15.087067) sec	Object 0.001359 nz(4218)
20000 iter  Elaptime(15.836900) sec	Object 0.001350 nz(4175)
21000 iter  Elaptime(16.627429) sec	Object 0.001340 nz(4177)
22000 iter  Elaptime(17.377234) sec	Object 0.001331 nz(4185)
23000 iter  Elaptime(18.126490) sec	Object 0.001322 nz(4152)
24000 iter  Elaptime(18.869651) sec	Object 0.001314 nz(4149)
25000 iter  Elaptime(19.608713) sec	Object 0.001305 nz(4153)
26000 iter  Elaptime(20.355096) sec	Object 0.001298 nz(4166)
27000 iter  Elaptime(21.109876) sec	Object 0.001290 nz(4127)
28000 iter  Elaptime(21.862709) sec	Object 0.001283 nz(4138)
29000 iter  Elaptime(22.603951) sec	Object 0.001276 nz(4106)
```

30000 iter  Elaptime(23.345057) sec  Object 0.001269 nz(4123)
Congratulation ! Finishing task.
````

Run Lasso Sample on multiple Machines
---
To run STRADS LS in a cluster, you need to create configuration files and modify lasso.py to point to the location of configuration files.

**- MPI machine file**
Each line contains a IP address of a machine you are going to use to run a MPI program. One example is following with the assumption that your cluster has five machines with IP address 10.55.1.1 ~ 5.
```sh
  10.55.1.1
  10.55.1.2
  10.55.1.3
  10.55.1.4
  10.55.1.5
```
Let's save this file as./pyscript/mach5.vm file
**- Node file **

Each entry represents an vertex with IP address, id, default handler. One example is following.
```sh
  10.55.1.1  0  default_handler
  10.55.1.2  1  default_handler
  10.55.1.3  2  default_handler
  10.55.1.4  3  default_handler
  10.55.1.5  4  default_handler
```
**- Link file**

Each line represent a link with source vertex id, port number, destination vertex id, port number. Current version of STRADS support start topology. Center vertex is a coordinator and neighbors of the center vertex are divided into scheduler and workers. Examples are following. Port numbers (46000 ~ 46003, 36000 ~ 36004) are randomly chosen. You should make sure that no other process in the system is using your chosen port numbers.
```sh
   4 46000 0 36000
   4 46001 1 36000
   4 46002 2 36000
```
```

```
    4 46003 3 36000
    0 46000 4 36001
    1 46000 4 36002
    2 46000 4 36003
    3 46000 4 36004
```
Let's save this file as./pyscript/node5.conf

**- Lasso script file (lasso.py)**

You need to modify some entries to specify the location of configuration files and input file.

``` sh
Regarding configuration files,
``machfile=['./mach5.vm']``
``node = [' --nodefile ./node5.conf ']``
``link = [' --linkfile ./star5.conf ']``

Regarding input files
``udfile = ['--filename "./sparse.X20000.txt.unsorted.bin"'] \\ your design matrix A``
``udfile3 = ['--filename3 "./sparse_Y20000.txt.bin"'] \\ your response vector Y``
``modelparams = ['--modelsize 20000'] \\ the number of coefficients``
``samples=['--samples 500'] \\ the number of samples``
```

Logistic Regression
==========

Given a design matrix X (possibly with some missing entries) and a response vector Y (binary value), the Logistic Regression app estimates one vector (regression coefficient vector) BETA, where coefficients are used to predict response value for a given sample. If X is N-by-M and Y is N-by-1, then BETA will be M-by-1.  Logistic Regression is commonly used to perform binary classification. We focus on L1-regularized Logistic Regression to avoid overfitting. With Petuum STRADS, we implement parallel distributed coordinate descent based LR. STRADS schedules model parameters to update in parallel while control degree of dependency among model parameters. In each dispatch, STRADS do weight-based sampling so that the selected model parameters are to make more contribution on the convergence speed than randomly selected model parameters.

Data Format
---
See data format in Lasso. Logistic Regression use the same data forms as Lasso.

Run LR Sample on Single Machine
---

For demonstration purpose, we put Lasso script file together with small sized input files. With assumption that Lasso is built according to the building instruction and your current working directory is STRADS root directory. Run the following commands.

```sh
cd pyscript
python singlelogi.py
```

Singlelogi.py run STRADS LR application with one machine creating three processes. Results will be similar the following. Due to the stochastic property of scheduling, your results might be slightly different.

```sh
1000 iter  Elaptime(0.686693) sec    Object 0.687371 nz(281)
2000 iter  Elaptime(1.619452) sec    Object 0.675143 nz(287)
3000 iter  Elaptime(2.310968) sec    Object 0.665100 nz(291)
4000 iter  Elaptime(3.020370) sec    Object 0.655755 nz(295)
5000 iter  Elaptime(3.732172) sec    Object 0.647713 nz(298)
6000 iter  Elaptime(4.440351) sec    Object 0.640571 nz(300)
7000 iter  Elaptime(5.146547) sec    Object 0.634278 nz(300)
8000 iter  Elaptime(5.851053) sec    Object 0.628781 nz(296)
9000 iter  Elaptime(6.555436) sec    Object 0.623623 nz(296)
10000 iter  Elaptime(7.261213) sec   Object 0.618909 nz(300)
11000 iter  Elaptime(7.965513) sec   Object 0.614345 nz(297)
12000 iter  Elaptime(8.668896) sec   Object 0.610357 nz(293)
13000 iter  Elaptime(9.372893) sec   Object 0.606761 nz(292)
14000 iter  Elaptime(10.076099) sec  Object 0.603155 nz(293)
15000 iter  Elaptime(10.779550) sec  Object 0.600088 nz(293)
16000 iter  Elaptime(11.484313) sec  Object 0.597178 nz(291)
17000 iter  Elaptime(12.197356) sec  Object 0.594638 nz(288)
18000 iter  Elaptime(12.911297) sec  Object 0.592231 nz(286)
19000 iter  Elaptime(13.625945) sec  Object 0.590099 nz(281)
20000 iter  Elaptime(14.342025) sec  Object 0.588047 nz(276)
21000 iter  Elaptime(15.055050) sec  Object 0.586147 nz(273)
22000 iter  Elaptime(15.768304) sec  Object 0.584381 nz(271)
23000 iter  Elaptime(16.480155) sec  Object 0.582622 nz(273)
24000 iter  Elaptime(17.194103) sec  Object 0.580994 nz(271)
25000 iter  Elaptime(17.903679) sec  Object 0.579480 nz(266)
26000 iter  Elaptime(18.725017) sec  Object 0.578009 nz(262)
27000 iter  Elaptime(19.644875) sec  Object 0.576603 nz(261)
28000 iter  Elaptime(20.574573) sec  Object 0.575277 nz(259)
29000 iter  Elaptime(21.513274) sec  Object 0.574094 nz(260)
```

30000 iter  Elaptime(22.439621) sec  Object 0.572985 nz(257)
Congratulation ! Finishing task.
````

Run LR Sample on multiple Machines
---
See the section **Run Lasso Sample on multiple machines**
````

# Page: Petuum Parameter Server

# New - Parameter Server v0.93 Reference Manual
Please find the parameter server reference manual here for more details:[ps_refman.pdf](https://github.com/petuum/public/blob/release_0.93/ps_refman.pdf?raw=true).

# General Description
The Petuum Parameter Server (PS) is a distributed key-value store. It allows multiple processes, typically one per physical machine, to share a large number of parameters. The interface of accessing the globally shared parameters closely mimics that of a in-memory hash table. The PS supports the classic BSP consistency model and the novel SSP consistency model. One may also choose to use the SSPPush model which communicates writes more aggressively with SSP as the worst case guarantee. This document describes how to create a distributed machine learning application using the parameter server step by step. In this document, we will use the provided Matrix Factorization (MF) application as an example.

The following operations should be done by all processes.

### Register Row Types

The parameter server organizes parameters as tables. Each tables consists of a set of rows and the table is simply a mapping from row id (32-bit integer) to row. A row is a mapping from column id (32-bit integer) to parameter.

In order to efficiently support a wide range of ML applications, each table is allowed to have a unique data structure for row. The system provides 3 row types:

1. `petuum::DenseRow` implements a fixed-sized vector;

2. `petuum::SparseRow` implements a hash table;

3. `petuum::SortedVectorMapRow` is a special row type used by the LDA application.

In order to use a row type for a table, it must be registered first, as
> petuum::PSTableGroup::RegisterRow\<petuum::DenseRow\<float> >(0);

The registration creates a mapping from row type ID (0 in this case) to a row type, which will be supplied to the PS when creating a table.

To be noted:

1. The application may register as many row types as needed (we even allow user-defined row types);

2. Calls to `RegisterRow` must happen before any other call to the PS;

3. All calls to `RegisterRow` must happen in the same thread (usually the main thread of the process) and all processes must create the same mapping from row type ID to row type.

### Initialize PSTableGroup

`petuum::PSTableGroup` is the application's handler to access the parameter server. After registering the row types, each process must properly initialize `petuum::PSTableGroup` via `static int Init(const TableGroupConfig &table_group_config, bool table_access);`

In MF, `petuum::PSTableGroup` is initialized as

> petuum::TableGroupConfig table_group_config;

> // Initialize table_group_config

> petuum::PSTableGroup::Init(table_group_config, false);

PS configuration parameter is supplied to the PS via `petuum::TableGroupConfig`. Please refer to `src/petuum_ps_common/config.hpp` for detailed explanation of the configuration parameters.

`petuum::PSTableGroup::Init()` must be invoked by the same thread that invokes `RegisterRow`.

The thread that invokes `Init()` is considered the Init thread for the PS, which may or may not access the parameters in the tables. Set `table_access` to `true` if the Init thread accesses the table parameters, otherwise `false`.

### Create Tables

After `Init()`, the Init thread may create a set of tables via `petuum::PSTableGroup::CreateTable(0,table_config)`.

Each table is configured with a `ClientTableConfig` object (`table_config` in this case). The first parameter is table ID (32-bit integer) which will be used to retrieve the table later.

The MF application creates 3 tables.

After creating all tables, the Init thread calls `petuum::PSTableGroup::CreateTableDone()` to signal the PS that all tables have been created.

### Create Worker Threads

The Init thread (main thread) may create a bunch of worker threads after creating all the tables. If the Init thread is intended to access the parameters, it has to synchronize with all other threads via
`petuum::PSTableGroup::WaitThreadRegister()`. Note this function is not used in MF because the Init thread does not need to access any parameters.

### Register Worker Threads with the PSTableGroup

Call `petuum::PSTableGroup::RegisterThread()` from the worker thread to register it with the PSTableGroup.
The worker thread may not access any PS API before it registers itself.

After that, all worker threads may access the parameter server and do its computation.

### APIs for accessing parameters

Parameters can be accessed via Table APIs. The application may retrieve a table via "petuum::PSTableGroup::GetTable(table_id)".

Each Table provides the following APIs for access its parameters:

> void Get(int32_t row_id, RowAccessor* row_accessor);

> void Inc(int32_t row_id, int32_t column_id, UPDATE update);

> void BatchInc(int32_t row_id, const UpdateBatch<UPDATE>& update_batch);

# Page: Petuum STRADS Scheduler

# STRucture-Aware Dynamic Scheduler (STRADS)

STRADS
=====
STRADS is a distributed scheduler framework for solving iterative Machine Learning problems with big model. It schedules model parameters to update for improving convergence speed while avoiding inter dependency among model parameters. This release contains two applications, Lasso and Logistic Regression as applications.

Four Components of STRADS
---
STRADS consists of four different components including scheduler, coordinator, workers and aggregator. A Coordinator, multiple workers and an aggregator make a scatter/gather style topology.

- Scheduler

Scheduler maintain weight information of model parameter. In each iteration, scheduler selects a set of promising model parameters to dispatch based on the weight information so that updating the scheduled parameters is likely to increase convergence speed than updating randomly selected parameter as common in stochastic method. Scheduler update weight information on receiving weight change from the coordinator when the dispatched is completed in the coordinator side. In addition to weight based sampling, STRADS scheduler runs user defined model dependency checking routine for a give set of model parameters. If any pair of parameters has too strong interference, one of them will be removed from the set.

- Coordinator

Coordinator is in charge of keeping model parameters, scattering a dispatch of parameter over the worker machines, sending back weight change information to the scheduler. In i-th iteration, the coordinator receive a dispatch set from the scheduler and scatter the dispatch together to all over the worker machines. On receiving updated model parameter values from the aggregator, it will udpate model parameters and send weight change information to the scheduler.

- Worker

On receiving a dispatch, worker executes user function to make a partial result with a partition of input data that is assigned to the worker machine. Each worker sends back its partial results to the aggregator.

- Aggregator

On collecting all partial results of one dispatch, aggregator runs user defined aggregation function to get new value of model parameters. New model parameter values are sent to the coordinator to be kept.

## Other Primitives of STRADS

STRADS provide the following low level primitives to facilitate distributed programming.

- Scheduling
- Global Barrier
- Data Partitioning
- Message abstraction

## STRADS Programming Interface

STRADS provides programming interface that advanced ML developers can develop their own distributed ML algorithms as far as the algorithms fit into STRADS scatter/gather programming model. Programming with STRADS requires c++ programming skills since STRADS APIs are designed with C++ template primitives.

Basically, STRADS allows users to define functions to run on scheduler, coordinator, workers and aggregator vertexes. In addition to the functions, use can define message types as C++ template for communicating across different vertexes. STRADS programming interfaces are implemented in the form of two classes.

### Handler Class

Handler class is a template class<T1, T2, T3, T4, SYSMSG> where user can define user functions as class method here. T1 ~ T4 are template of user defined messages and used as parameter and return type of class methods(user functions). STRADS requires 4 major user functions for scheduling/updating parameters and three minor function for checking progress such as calculating objective function value.

**Major User Functions**
- T1 &dispatch_scheduling(SYSMSG, T3)
- void do_work(T1, T2)
- void do_msgcombiner(T2, stradsctx)
- void do_aggregator(T3, stradsctx)

- int check_dependency(list parameters)
- set_initi_priority(list weight, model_cnt)

**Minor User Functions for progress checking**
- void do_obj_calc(T4, stradsctx)
- void do_msgcombiner_obj(T4, stradsctx)
- void do_object_aggregation(T4, stradsctx)

Message Class
---
Message class is a template class that allows user to define a type of messages that contains arbitrary number of elements. Logically, user can define any type for the element. STRADS provides several template classes that can make a message with one, two or three different kinds of element types. Again, you can put arbitrary number of elements with different types on a message. If you define your message type only with POD type, you can simply finish message class definition with defining elements type.

- element class
- message class