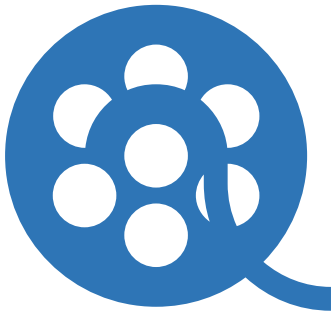
Movie Recommendation

2019 January



https://github.com/louisyang2015/movie_recommender

Requirements

Software Requirements

Python 3, with NumPy, SciPy, boto3, requests

C++ compiler

Memory Requirements

For the September 2018 dataset, **16 GB memory** is sufficient for a **quad core** machine running Windows 10.

Using more cores will use more memory. The Python script spawns as many processes as there are cores. This behavior can be overridden with the "cpu_count=x" command line parameter.

The task with the highest memory usage is "build_similar_movies_db.py", which uses 3 GB per process.

As time goes on, the dataset gets larger, requiring more memory.

Basics

Directory: basics

Files here demonstrate basic algorithms.

LSMR

file: lsmr.py

This file demonstrates how to use Scipy's LSMR algorithm to find the least square solution for

$$A\vec{x} = \vec{b}$$

when "A" is a very sparse matrix.

ALS

file: als.py

Summary

This file demonstrates how to solve the alternating least square problem.

The code locks the number of factors to 3, for simplicity.

Suppose "user id 0" has rated "movie id 0" as "3 stars". The model is:

$$u_{0,0}m_{0,0} + u_{0,1}m_{0,1} + u_{0,2}m_{0,2} + u_{0,bias} = 3 - m_{0,median}$$

The code in this file works on the slightly simplified version:

$$u_{0,0}m_{0,0} + u_{0,1}m_{0,1} + u_{0,2}m_{0,2} + u_{0,bias} = 3$$

The "$m_{0,median}$" offset, which represent the median score of movie #0, will be applied before the data is fed to the ALS algorithm.

A very large, very sparse system

Each movie rating produces one equation. For user "i" and movie "j", the equation is:

$$u_{i,0}m_{j,0} + u_{i,1}m_{j,1} + u_{i,2}m_{j,2} + u_{i,bias} = score$$

All other coefficients are zero.

Alternating Least Squares

To find a solution, a randomly generated set of factors is used as a starting point.

Since the users have a "bias" factor, the movie factors are smaller - each movie has 3 factors, and each user effectively has 4 factors.

Start by generating random movie factors - because it's the smaller problem of the two. The dot product of the factors are meant to adjust a user's standard "bias" score, as well as a movie's "median bias" score. So the factors are overall likely to be symmetric, between -1 and +1. Therefore, the initial vector is set to be random values between -1 and +1.

With the movie factors held constant, do a least square fit with the user factors. Then while holding the user factors constant, do a least square fit for the movie factors.

The two functions are:

```
def solve_for_users(movies, user_ids, movie_ids, ratings, num_users, factors):
def solve_for_movies(users, user_ids, movie_ids, ratings, num_movies, factors):
```

In given "movies", solve for "users". The movie factors are known constants. Each movie rating looks like:

0 + ... + (1 * user_factor0) + (2 * user_factor1) + ... + user_bias + 0 ... + 0 = movie_rating

The {1, 2, ...} are loaded from the "movies" vector.

The column numbers are determined by user_id.

In given "users", solve for "movies". The user factors are known constants. The "user bias" needs to be subtracted from the "movie rating" value.

Each movie rating looks like:

0 + ... + (1 * movie_factor0) + (2 * movie_factor1) + 0 + ... + 0 = movie_rating - user_bias

An ill proposed problem

Alternating least squares is an ill proposed problem that does not have a unique solution.

For example:

$$[3 \quad 4] \cdot [5 \quad 6] = [4 \quad 3] \cdot [6 \quad 5]$$

So, one can swap columns of the user and movie factors, while retaining the same dot product.

Similarly,

$$[3 \quad 4] \cdot [5 \quad 6] = [-4 \quad 3] \cdot [-6 \quad 5]$$

So, one can invert signs and still maintain the same dot product.

Therefore, there are many numerically different solutions that will produce the exact same output score.

Furthermore, this demonstration demonstration code shows that totally different solutions can produce almost identical scores.

The code generates a data set via:

```
(users_real, movies_real, user_ids, movie_ids, ratings) = generate_data(...)
```

The "users_real" and "movies_real" are the real factors used to generate the "ratings" score.

The alternating square procedure produces "users" and "movies" vectors, that are totally different from "users_real" and "movies_real", yet produces almost the same score.

The code splits the generated data into a training set and a test set. The vectors "users" and "movies" produces very little loss, while the vectors "users_real" and "movies_real" produces zero loss. So, it seems that there are infinitely many satisfactory solutions.

Determining the number of movies

The

```
generate_data(num_users, k)
```

function takes the number of users, and internally determines the number of movies to use to generate the dataset.

Let "u" be the number of users and "m" be the number of movies. The "u" is set by the user, while the "m" needs to be determined.

The number of variables in the system is = "4u + 3m". There are three factors, but each user has a "bias" variable, making it four variables per user.

In this fake generated dataset, every user has reviewed every movie. So, the number of equations in the system is "u * m"

For least square to be meaningful, the number of equations must greatly exceed the number of variables. To have four times as many equations as there are variables:

$$(4u + 3m) * 4 = u * m$$

To be able to tweak the number of equations:

$$(4u + 3m) * k = u * m$$

Solving for "m" then gives:

$$m = \frac{4ku}{u - 3k}$$

which is the expression implemented inside "generate_data(...)".

Multiprocessing

Location: python\basics\multiprocessing

Multi-threaded problem-solving using Python's "multiprocessing" module

add.py, add_proc.py

Adds two lists in parallel

Call tree:

```
main()
  |→ add_proc.start_processes()
  |→ add_proc.split_list_and_send(...)
  |→ add_proc.run_function(...)
  |→ add_proc.concat_var_into_list(...)
  |→ add_proc.clear_all_data()
  |→ add_proc.end_processes()
```

lsmr.py, lsmr_proc.py

Generates a $A\vec{x} = \vec{b}$ least square problem and uses Scipy's "lsmr" and "lsqr" to solve it. The A matrix is 27e6 by 280e3.

This programmed showed that "lsmr" and "lsqr" are single threaded. On my computer the LSMR run time is 165 seconds, and LSQR will take 202 seconds.

This project led me to the "C++ LS" project - writing a multi-threaded conjugate gradient solver.

Call tree:

```
main()
  |→ run_test(...)
       |→ _proc.start_processes()
       |→ generate_data(...)
       |     |→ _proc.define_list(...)
       |     |→ _proc.send_same_data(...)
       |     |→ _proc.run_function(...)
       |     |→ _proc.concat_var_into_numpy_array(...)
       |
       |→ _proc.end_processes()
```

Distributed Work

Directory: python\basics\distributed_work

This example shows how to add two arrays on multiple computers and on multiple processes.

To run the example (on Windows):

Open four command prompt windows and move to the "distributed_work" directory.

python cluster_server.py	starts cluster server at port 10000
python worker_server.py	starts worker node at default port 10001
python worker_server.py port_number=10002	starts worker node at port 10002
python add.py	The add example program to generate two arrays of numbers, and then add them up in a distributed manner

Start a new command prompt window:

python worker_server.py port_number=10003	starts worker node at port 10003

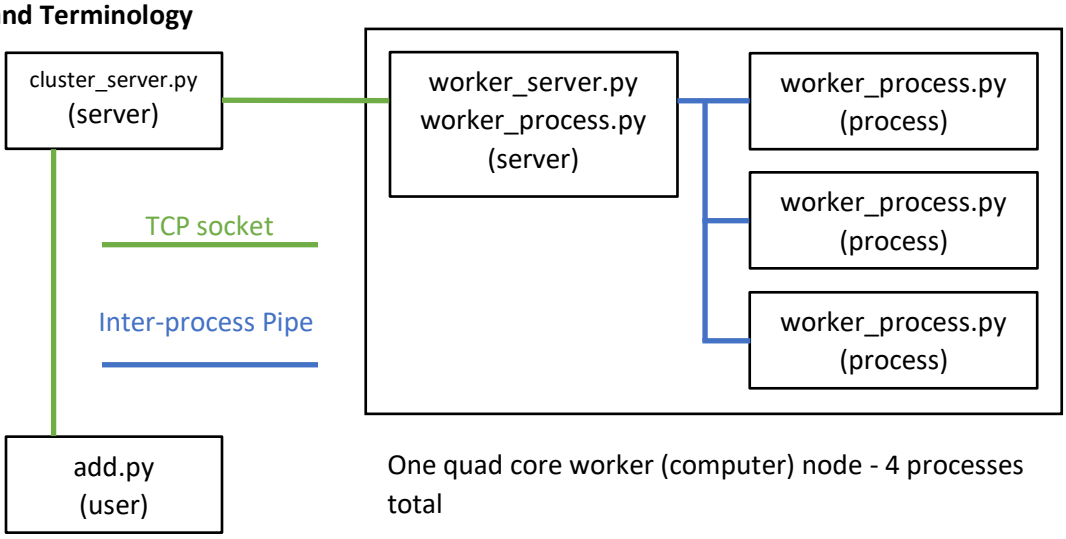
which would increase the adding speed.

Sequence of events in the example

The print outs from the servers, as well as the changes in the "distributed_work\bin" directory, gives an indication of what is happening.

1. The cluster server creates and manages a "cluster_info.json". This contains addresses of the cluster nodes.
2. Starting "worker_server" registers new worker nodes with the cluster server.
3. "add.py" creates "input.bin", which contains two Numpy arrays.
4. Distributed add:
    a. The cluster server assigns each node to add a subset of the array
    b. The worker server gets the following messages: a single "add_setup", followed by numerous "add" and "echo", followed by a "add_cleanup".
    c. The worker server all run multiple processes. Each process adds a subset of the "add" assignment.
    d. The worker server writes output files to "distributed_work\bin".
5. "add.py" merges the output files in "distributed_work\bin" into a single "output.bin".

Setup and Terminology



One quad core worker (computer) node - 4 processes total

cluster_server.py - distributes the work to multiple computers (worker servers)

- cluster.py - functions for talking to cluster_server.py

worker_server.py - distributes the work to multiple processes

worker_process.py - multiprocessing support for worker_server.py

adder.py - generate random data, add them using the cluster, check the result

Only add.py is a transient / interactive process. The other scripts (cluster server, worker server, and worker process) are all long running servers or processes.

Software requirements

- completion time estimation
- redistribute processing to new worker nodes as they join
- health check so to use AWS spot instances

cluster_info.json

It looks like:

```
[
    {
        "port_number": 10000,
        "address": "192.168.1.66"
    },
    {
        "port_number": 10001,
        "cpu_count": 4,
        "address": "192.168.1.66"
    },
    {
        "port_number": 10003,
        "cpu_count": 4,
        "address": "192.168.1.66"
    }
]
```

The very first entry in the list always refers to the cluster server. The other entries are worker nodes.

In actual use, the worker nodes should all have different IP addresses. During testing, the worker nodes can have the same IP address, but different port numbers.

The cluster server location is always first, but the workers are extracted from a dictionary indexed by "address:port". So the workers can show up in any order.

Messages

Accepted by cluster_server.py

Distribute	Request	{
		"op": "distribute",
		"worker_op": op to distribute to worker servers,
		"length": op argument range from 0 to length-1
		}

Done	Request	{
		"op": "done",
		"address": ip address
		"port_number": port number
		}

Echo	Request	{
		"op": "echo",
		"value": a random number
		}

	Response	{
		"op": "reply",
		"value": the number received
		}

Register	Request	{
		"op": "register",
		"address": ip address,
		"port_number": port number,
		"cpu_count": cpu count
		}

	Response	{
		"op": "reply"
		}

Progress	Request	{
		"op": "progress"
		}

	Response	{
		"length": total number of items,
		"completed": number of items completed,
		"seconds_left": estimated number of seconds left
		}

Shutdown	Request	{
		"op": "shutdown"
		}

Status	Request	{
		"op": "status"
		}

	Response	{
		"current_op": current operation name,
		"workers":
		{
		"worker node address:port":
		{
		"state": worker node state,
		"total_time": total work time
		"total_units": total work completed
		}
		}
		}

Accepted by worker_server.py

Echo	Request	{
		"op": "echo",
		"value": a random number
		}

	Response	{
		"op": "reply",
		"value": the number received
		}

Shutdown	Request	{
		"op": "shutdown"
		}

Add messages

add.py → cluster server:

```
{
    "op": "distribute",
    "worker_op": "add",
    "length": len(x1)
}
```

cluster server → all workers:

{"op": "run_op", "op_name": "add_setup"}

cluster server → each worker:

{"op": "run_op", "op_name": "add", "start": 0, "length": 100}
{"op": "run_op", "op_name": "add", "start": 100, "length": 100}
{"op": "run_op", "op_name": "add", "start": 200, "length": 100}
...
The exact indices need to be based on timing of prior operation. So the very first message might be indices 0 through 3. Then based on how long that took, the next set of indices can be decided.

Each message needs to be processed within some time limit, say 20 seconds. This limit needs to be a fraction of the AWS spot instance time limit, which is 2 minutes.

each worker → processes interaction is similar to the "multiprocessing" basic example:

```
_proc.split_list_and_send(_data["x1"], ...)
_proc.split_list_and_send(_data["x2"], ...)
_proc.run_function("add", None)

# merge result
sum = _proc.concat_var_into_numpy_array("sum")
```

cluster server → all workers:

{"op": "run_op", "op_name": "add_cleanup"}

Call Trees

Three key application functions - the adder.py issues a "worker_op" of "add", and the framework calls the following three functions:

- **worker_server.py :: add_setup(...)** - reads input from disk
- **worker_server.py :: add(...)** - break request down into arrays, send arrays to processes for adding, merge result together, save to disk
    - **worker_process.py :: add(...)** - add two arrays

cluster_server.py :: main()

```
ClusterServer()
  |→ __init__()
  |     |→ register_with_disk()
  |
  |→ message_loop()
        |→ handle_xxx(...)
```

worker_server.py :: main()

```
_proc.start_processes()

WorkerServer()
  |→ __init__()
  |→ register_with_cluster()
  |→ message_loop()
        |→ handle_xxx(...)
        |→ add_setup(...)
        |→ add(...)
        |     |→ worker_process.split_list_and_send(_data["x1"], ...)
        |     |→ worker_process.split_list_and_send(_data["x2"], ...)
        |     |→ worker_process.run_function("add", None)
        |     |→ sum = worker_process.concat_var_into_numpy_array("sum")
        |
        |→ _send_done_reply()

_proc.end_processes()
```

Adding numbers

```
add.py :: main()

      cluster.send_command({
          "op": "distribute",
          "worker_op": "add",
          "length": len(x1)
      })
```

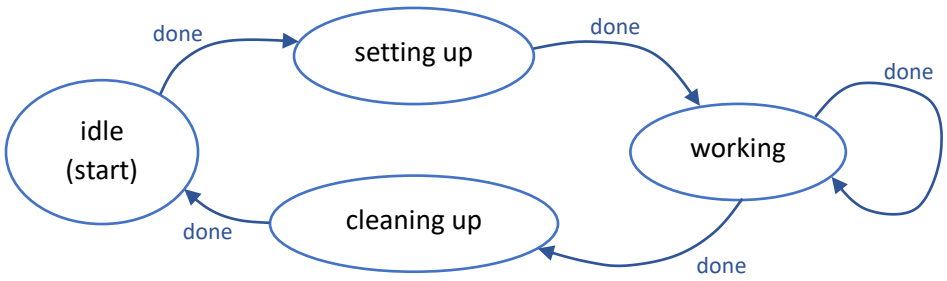
```
ClusterServer :: handle_distribute(...)

worker_server.py :: add_setup(...)
WorkerServer :: _send_done_reply()

ClusterServer :: handle_done(...)
  |→ send_work(...)
       |→ worker_health_check(...)

worker_server.py :: add(...)

    worker_process.py :: add(...)
WorkerServer :: send_done_reply()

ClusterServer :: handle_done(...)
```

handle_done(...) will send more work, repeating the above cycle.
Eventually handle_done(...) will send "_cleanup" message.

The "_cleanup" message is handled within WorkerServer :: _message_loop()

Worker States

Worker states are tracked via WorkerNode.state



The worker nodes start in the "idle" state.

The ClusterServer :: handle_distribute(...) sends a "_setup" message to each worker node, and put that worker node into the "setting up" state. This kicks off the whole "add" process. Every time the worker node is finished with its task, it will reply with a "done" message. The ClusterServer :: handle_done(...) then advances the state machine.

The addition of numbers happens in the "working" state.

Completion Time Estimation

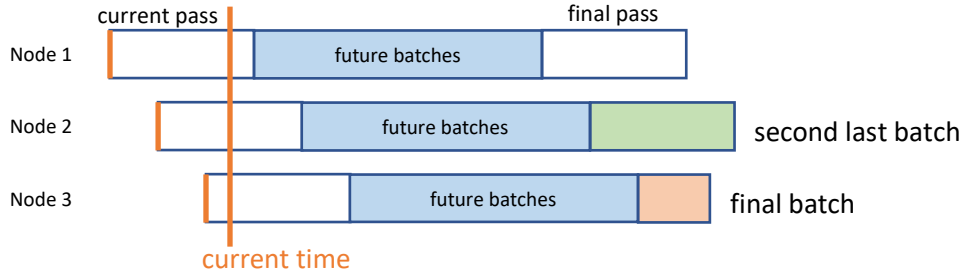
Location: cluster_server.py :: DistributedOp :: estimate_seconds_left(...)

Predicting node work

The WorkerNode class has "total_time" and "total_units". So all previous work done is used to predict how much work the node can do.

Completion time estimation general concept

Assume: batch time = 10 seconds



The completion time has two possibilities. It can be determined by either the second last batch, or the final batch of work.

- The second last batch starts before the final batch, but the second last batch is a full (10 second) batch.
- The final batch of work is the last batch to start, but it is shorter than the second last batch.

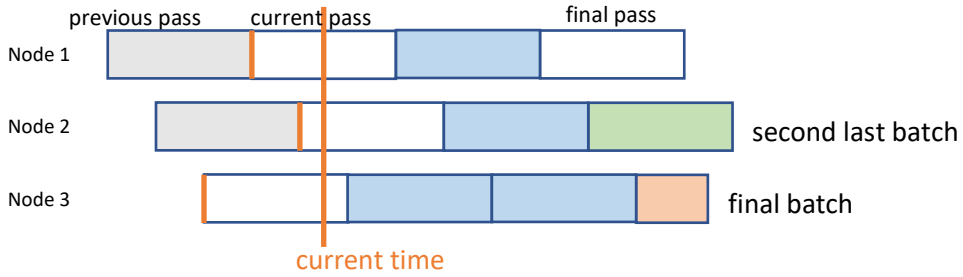
The code estimates the ending time by locating the nodes that will be working on the final batch and the second last batch.

The code sorts the nodes according to their "start time" - as shown in the picture. The node working on the second last batch is usually one index below the node working on the final batch.

Roll over case (final node index = 0)

If the node with the earliest start time is predicted to have the final batch (< 10 seconds) of work, then the second last node is on the opposite side of the sorted list (of worker nodes) - it's the node with the most recent start time.

Next page shows the exact same situation, but with the current time bar advanced, such that the node getting the final batch will have the earliest start time.



The completed batches have been marked with gray.

The node 2, the node doing the second last batch, has the latest batch "start time", while node 3, the node doing the final batch, has the earliest batch "start time".

Also node the number of batches is different from the standard case.

In the general case (first diagram), node 2 has the same number of "future batches" as node 3:

Node 2: current batch (white) + **2** future batches + 1 full (2nd last) batch

Node 3: current batch (white) + **2** future batches + 1 partial final batch

In this special case (second diagram), node 2 has 1 future batch while node 3 has 2 future batches:

Node 2: current batch (white) + **1** future batches + 1 full (2nd last) batch

Node 3: current batch (white) + **2** future batches + 1 partial final batch

AWS modification

File: worker_server.py

```
aws_instance = True
```

The AWS specific code is in WorkerServer :: handle_echo(...).

```
# for AWS (spot) instance, check the special url for interruption notice
if aws_instance:
    r = requests.get("http://169.254.169.254/latest/meta-data/spot/instance-action")
    # standard response is error code 404
    if r.status_code != 404:
        # if status_code is not 404, then there is a cancellation notice
        value += 1
```

This is to support AWS spot instances. This kind of instance is low cost, but Amazon can terminate it with a 2 minute warning. The warning shows up at the URL "169.254.169.254/latest/meta-data/spot/instance-action".

Before asking a worker node to add, the ClusterServer does a health check by sending an "echo" message. The worker node, in handling the "echo" message, check the Amazon URL for instance termination messages. If Amazon is terminating the instance, then a wrong value is replied to the sender, so to fail the health check. The worker node will then be told to shut down and will not be excluded from the cluster.

Windows Occasional Crash

This example occasionally hangs on Windows, due to network error.

To duplicate, see the function cluster.py :: run_echo_test(...). This function repeated sends "echo" messages to a server and checks the reply. Both ClusterServer and WorkerServer support "echo" messages, and can be stressed tested using run_echo_test(...).

Inside the run_echo_test(...), there's a block of comments:

```
# On Windows, a slight delay between connection is needed
# (tested with Python 3.5.4)
# time.sleep(0.01)
# without this slight delay, I get:
# [WinError 10048] Only one usage of each socket address is normally permitted
```

When tested on AWS Linux, no delay is necessary for run_echo_test(...). On Windows, when tested with Python 3.5.4, a delay is necessary.

Since the ultimately goal is to deploy clusters on AWS, no delay has been added to the main program.

100k Dataset

Summary

Directory: 100k_data

The scripts here are for the 100k dataset. They are independent of each other, so there's some copy and paste style duplication between the scripts.

Location of the data files

Since the scripts are independent, each script has a line at the top that points to the data files:

```
data_dir = "data" + os.sep
```

Scripts

median_predictor.py	predict ratings using median of the training set - this is the base line algorithm
count_tags.py	predict ratings using previous ratings, as well as tags. This approach is called "tag counting" here because the algorithm counts, for example, how many "science fiction" tags are in the user's positively rated movies. The relative tag counts is then used as the basis for prediction.
ls_tags.py	predict ratings using a linear combination of tags. In "count_tags.py", the tag coefficients are based on counting the presence of tags, so that all the tag information are always used. In "ls_tags.py", the tag coefficients are derived from least squares (LS).
ratings_als.py	predicts ratings using only past ratings - so this approach only uses the "ratings.csv" data
similar_movies.py	searches for similar movies, using genre tags and existing user ratings
title_search.py	Builds an index and searches the movie titles. For example, the user can type in "dragon ball" and this software lists movies with "dragon" and "ball" in the movie title. This file depends on "PorterStemmer.py", which is the Porter Stemmer from: https://tartarus.org/martin/PorterStemmer/python.txt

Agreement Percentage

Agreement percentage is a metric made up for this project.

In the "count_tags.py" and "ratings_als.py" scripts, the data is divided into a training set and a testing set. "Agreement percentage" is how success is measured using the test set.

Example

Suppose the test set is:

Movies	Rating
A, B, C	5 stars
D, E, F	4 stars

This is implying the following:

rating(A) > rating(D) rating(A) > rating(E) rating(A) > rating(F)	rating(B) > rating(D) rating(B) > rating(E) rating(B) > rating(F)	rating(C) > rating(D) rating(C) > rating(E) rating(C) > rating(F)

For a total of 9 requirements.

Suppose the model predicts the following:

Movies	Rating
A, B	5 stars
C, D, E, F	4 stars

which satisfies 6 out of the 9 requirements:

rating(A) > rating(D) rating(A) > rating(E) rating(A) > rating(F)	rating(B) > rating(D) rating(B) > rating(E) rating(B) > rating(F)	rating(C) > rating(D) rating(C) > rating(E) rating(C) > rating(F)

As a result, the agreement percentage is 6/9 = 66.7%

A Top-Heavy Metric

In making recommendations, the top recommendations matter more than the bottom ones - the users are far more likely to pay attention to the top part of the list.

The agreement percentage, defined above, is a top-heavy metric - because the items ranked at the top naturally produce more requirements.

For example, if the test data is:

Movies	Rating
A, B	5 stars
C, D	4 stars
E, F	3 stars

Then A has four requirements (A>C, A>D, A>E, A>F), while C has only two requirements (C>E, C>F). So having A positioned correctly is more important than positioning C correctly.

Rankings Comparison Implementation

The actual ratings are in a list of list format. Example:

Suppose user #0 rated the movies as:

Movie ID	Ratings
1000	4
1001	5
1002	5
1003	1
1004	4

The actual ratings are stored as:

```
actual_ratings[
  [1001, 1002],
  [1000, 1004],
  [1003]
]
```

This list of list format spells out the rank requirements:

rating(1001) > rating(1000)
rating(1001) > rating(1004)
rating(1001) > rating(1003)
...

The predicted ratings are in a dictionary that map movie_id to its rating value.

So to check:

rating(1001) > rating(1000)

we check:

predicted_ratings[1001] > predicted_ratings[1000]

Implementation Functions

Files: count_tags.py, median_predictor.py, and ratings_als.py. All three files use the same functions:

```
Tester :: convert_ratings_to_list_of_list(self, movie_ids, ratings)
Tester :: convert_ratings_to_list_of_list(self, movie_ids, ratings)
```

Results

Agreement percentages:

Approach	Agreement Percentage (typical)
Movie average only*	0.66
tag counting	0.64 ~ 0.65
LS tag	0.63 ~ 0.64
ALS rank 3	0.60 ~ 0.62
ALS rank 4	0.59 ~ 0.61
ALS rank 5	0.60
ALS rank 9	0.575, 0.60
Movie median only	0.48 ~ 0.49

* There is no separate script for average. The result comes from modifying the "median_predictor.py" script.

The "average" is not a valid recommender algorithm since everyone will get the same recommendations.

Tag counting has the best agreement percentage, and it's also much easier to train than ALS.

The ALS models are still useful even though they underperform the tag counting approach. The tag counting approach require the data to be tagged, which is true for this particular data set, but is not true in general.

Increasing ALS rank, meaning more factors variables, reduces the training loss, but the performance on test data set did not improve. This is because for a stable solution, the least squares part of ALS require at least as many equations as there are unknowns. For example, to satisfy a 3 factor model, each user must have rated 4 or more movies, and each movie must have been rated 3 or more times. Users and movies that do not satisfy this constraint are excluded from both the training and testing data sets.

Therefore, increasing ALS factors, while making the model more flexible, also reduces the data eligible for the model. That is why at some point, increasing ALS factors no longer help.

Median

File: median_predictor.py

Summary

The three ratings predictors are median_predictor.py, count_tags.py, ratings_als.py.

The median predictor uses the median movie score from the training set as a ratings predictor. This is the simplest predictor of the three predictors.

Classes

CsvData	Contains data read from the csv files.

ModelData	Additional data, derived from "csv_data", that is needed for modeling.
	
	Splits the data into a training set and a testing set:
	`user_ratings_train - {user_id: [movie_ids_list, ratings_list]}`
	`user_ratings_test - {user_id: [movie_ids_list, ratings_list]}`
	
	Note that this split is done on a per user basis.
	
	Computes the medians for all movies using just the training data:
	`movie_medians - {movie_id : movie_median_rating}`

Tester	`def test(self, user_id):` `  """go through the test set, returns the rank agreement."""`

Tag Counting

File: count_tags.py

Algorithm

Each user has a "profile" composed of tags and a value associated with that tag.

Example

A user rates a "science fiction" movie 4 stars.

This translates into "+1" for "science fiction".

The "+1" comes from "4 - 3", since 3 is the neutral rating.

So a "5 star" rating would contribute a "+2" to the associated tag. A "1 star" rating would contribute a "-2" to the associated tag. Ratings that deviate more from the neutral "3" contributes more points.

Scaling by document genre and tag count

Some genres, such as drama, are more popular than others. Likewise, some tags are more popular than others.

To prevent a highly popular genre or tag from accumulating a high score by accident, the genre and tag scores are divided by the document count.

For example, if a user has accumulated 10 points in the "science fiction" category, and the document has 100 "science fiction" movies, then the user's final score is:

{science_fiction: 0.1}

User profile summary

genres - this is counted and scaled - to prevent over counting more popular genres.

tags - this is counted and scaled.

Example:

```
genre_profile = {
   "Sci-Fi": +1.2
   "Thriller": -0.5
}

tag_profile = {
   "violent": -1
}
```

This user likes science fictions, and dislike thrillers and violent movies.

Movie profile

Each movie also gets a profile, based on the genres and tags.

genre - this is treated as an indicator, either 1 or 0. It is not scaled.

tags - this is scaled and counted. So the more rare tags are worth more. If a movie is labeled as "funny" two times, then it's more likely to be funny than a movie that has "funny" labeled once. So the "tag" attribute should be increasing with counts, but the increase should be a diminishing increase. Currently the function is "ln(count) + 1". Having a single "funny" vote count as a "1", while having 10 "funny" vote counts as "ln(10) + 1 = 3.3".

Score computation

prediction = median + $[\text{user tags}]^T$ [movie tags] * x_0 + $[\text{user genres}]^T$ [movie genres] * x_1 + x_2

prediction = predicted movie rating

median = median rating for the movie

user tags = the user profile, where each tag has a modifier

movie tags = the movie profile, where each tag has a modifier

x_0 = tag importance. Higher value means tags matter more for this particular user.

user genres = the user profile, where each genre has a modifier

movie genres = the movie profile, where each genre is either a 1 or a 0

x_1 = genre importance. Higher value means genres matter more for this particular user.

x_2 = user bias. Higher value indicates a higher bias from the median movie rating.

Not all movie has tags, so there is an alternative model, for situations where the movie has no tag.

prediction = median + $[\text{user genres}]^T$ [movie genres] * x_0 + x_1

Information leak

tags - these are leaked - the training is done on tags of the entire data set

median - this is not leaked. Training is done using only the median of the movies in the training set. When making predictions on the testing set, new movies, meaning movies without any known median, are skipped.

An alternative approach, to avoid the information leak, would be to sort the movies chronologically. The later data would be used for testing. Only tags made before a certain timestamp is used to train the model.

With the alternative approach, there is more coding, and only one set of training data is possible. It's not possible to shuffle data around randomly, as the training data must be the earlier part of the full dataset. Algorithm stability has to be tested by extracting more 100k rating subsets from the full dataset. In the end, the benefits don't seem to out weight the costs.

Classes

CsvData	Contains data read from the csv files. There are dictionaries that map: • genre string → genre id • movie id → genre id • user id → [movie ids list, ratings list] • tag string → tag id • movie id → List of tag ids
ModelData	Additional data, derived from "csv_data", that is needed for modeling. There are dictionaries that map: • movie id → movie median rating • genre id → genre count • movie id → {tag id → ln(count) + 1} The "ln(count) + 1" is to make more tags matter more, but also have it be a decaying increase. • tag id → tag count • user training set - {user id → [movie ids list, ratings list]} • user test set - {user id → [movie ids list, ratings list]}
UserProfile	Goes over a user's ratings and tally the genre and tag attributes. • tag_profile - { tag id → score } • genre_profile - { genre id → score } • x_factors0 - [x_0, x_1, x_2] constants for the prediction model • x_factors1 - [x_0, x_1] constants for the simplified (no tag score) model
Tester	`def test(self, user_id):` `  """Returns ranking agreement percentage for "user_id"."""`

Call Trees

```
UserProfile :: __init__(...)
  |→ compute_tag_profile(...)
  |→ compute_genre_profile(...)
  |→ compute_x_factors(...)
        |→ dot_product_with_movie_tag_scores(...)
        |→ dot_product_with_movie_genre_ids(...)

UserProfile :: predict(...)
  |→ dot_product_with_movie_tag_scores(...)
  |→ dot_product_with_movie_genre_ids(...)


Tester::test(user_id)
  |→ user_profile = UserProfile :: __init__(user_id, ...)
  |→ predict(self, user_profile, movie_id)
        |→ user_profile.predict(...)
```

LS Tag

File: ls_tag.py

Prediction Equation

If four factor variables are used:

$$rating = median + f_0 x_0 + f_1 x_1 + f_2 x_2 + 1(x_3)$$

rating = predicting movie rating

median = movie median score

$[f_0 \ldots f_2]$ = factor variables, they vary for each movie, and for each user

$[x_0 \ldots x_2]$ = user coefficients. They vary for each user, and are used with all movies reviewed by that user.

x_3 = user bias. The final factor variable is always the user bias

Each user rating produces one equation, and the $[x_0 \ldots x_2]$ are determined using least squares.

The number of factors to use is determined in "UserProfile :: decide_num_factors(...)". The number of factors to use grows with the number of ratings (and hence equations).

The factors to use are determined in "UserProfile :: decide_profile(...)". The most encountered factors are used - so this usually means genres, followed by the most common tags.

The "decide_profile(...)" decides WHICH factors to use. The VALUE of each factor is determined according to factor type:

For movie genres, the value of the factor is either 1 or 0.

For movie tags, a score is used, similar to the tag counting algorithm. The number of tags is summed, then damped by a natural log, then divided by total number of such tags in the document.

Example:

A movie is tagged as "funny" 2 times, and there are a total of 10 "funny" tags in the whole data set. The "score" for the "funny" factor is:

$$\frac{\ln(2) + 1}{10} = 0.169$$

The "divide by 10" isn't really necessary here, since the $[x_0 \ldots x_{n-2}]$ user coefficients are being determined via least squares. Normalizing by the number of factors make the "x tag coefficients" look more uniform. The tag coefficients can then be sorted, to reveal the most important tags for each user. The tag coefficients still cannot be compared with the genre coefficients, due to genres being used vastly more than tags.

UserProfile call tree

```
UserProfile :: __init__(self, user_id, csv_data, model_data)
  |→ count_ids(movie_ids, movie_genres, movie_tags)
  |→ decide_num_factors(len(movie_ids))
  |→ decide_profile(id_count, num_factors)
  |→ compute_x_factors(...)
```

UserProfile :: decide_num_factors(...)

This function decides on the number of factors to use.

The number of factors depend on the number of equations available - more equations means the least square procedure can handle more factors. At the same time, the number of factors should grow slower and slower. The reasoning is that the most common factors are always selected first, so they have a good chance of being present in the equations. Later on, the less common factors are added to the system of equations, but they do not appear often in the equations - so more equations are necessary to avoid overfitting.

The number of factors is expressed as a "factor profile":

```
# factor profile
# (number of factors, number of equations(ratings))
factor_profile = [
    [5, 10], # 5 factors for the first 10 ratings
    [5, 5 * 4],
    [5, 5 * 8],
    [5, 5 * 16],
    [5, 5 * 32],
]
```

Literal interpretation:

- Five factors will be allowed for the first 10 equations. During this stage, one factor will be allowed for every 2 equations.
- Five factors will be allowed for the next 5*4 = 20 equations. During this stage, one factor will be allowed for every 4 equations.
- Five factors will be allowed for the next 5*8=40 equations. During this stage, one factor will be allowed for every 8 equations.

Expressed as a table:

x = Number of Equations	user_profile.decide_num_factors(x)
2	1
10	5
14	6
30	10

38	11
70	15

UserProfile :: decide_profile(...)

Returns a list of "profile IDs" indicating what factors are being used.

A "profile ID" is either:

- genre ID
- tag ID + 1000

Since there are far fewer than a thousand genres, any "profile ID" $\geq$ 1000 is a tag ID.

The most common factors encountered in the user ratings are used. Since every movie has multiple genres, and movie tags are relatively uncommon, the list of profile IDs usually end up being a list of genre IDs, followed by tag IDs.

ALS

File: ratings_als.py

Classes Overview

CsvData	Contains data read from the "ratings.csv" file. Contains: • user_ratings - {user_id: [movie_ids_list, ratings_list]}
ModelData	Additional data, derived from "csv_data", that is needed for modeling. Contains: • movie_medians - {movie_id : movie_median_rating} • user_ratings_train - {user_id: [movie_ids_list, ratings_list]} • user_ratings_test - {user_id: [movie_ids_list, ratings_list]}

ALS_Model	Construction requirement: model_data, factors
	
	The ALS_Model requires a certain number of reviews to work:
	`def can_predict(self, user_id, movie_id=None):`
	`"""Returns True if the model can be used for "user_id" and "movie_id"."""`
	
	To make prediction:
	`def predict(self, user_id, movie_id):`
	`"""Predict rating for the given user_id and movie_id.`
	
	More details in separate section below.

Tester	`def test(self, user_id : int):` `"""Returns ranking agreement percentage for "user_id".`

The linear model

ALS uses a model similar to tag counting, except that the factors are not given in terms of genres and tags. The factors have to be derived via least squares.

For a model using three factors:

$$\text{prediction} = \text{median} + u_0m_0 + u_1m_1 + u_2m_2 + u_{bias}$$

prediction = predicted movie rating

median = median rating for the movie

$[u_0, u_1, u_2, u_{bias}]$ = user factors. Each user has a different vector of factors.

$[m_0, m_1, m_2]$ = movie factors. Each movie has a different vector of factors.

Shrinking the data set

The least square procedure is stable only if:

(number of linear equations) $\geq$ (number of variables)

For a model using 3 factors for movies and 4 factors for users, the movie can be included in the model only if it has been reviewed at least 3 times, and the user can be included if he has made 4 reviews.

The dataset is shrunk, as needed, to remove ineligible users and movies:

```
ALS_Model :: def shrink_dataset(self, user_ids : list, movie_ids : list,
                  ratings : list, factors : int):
    """Reduce the training set "user_ids, movie_ids, ratings"
    dataset so to satisfy the "factors" constraint. Only users
    with "factors + 1" or more appearances, and movies with
    "factors" or more appearances will remain in the
    return dataset.
```

Remapping IDs

The ALS combines all the user IDs into a single column vector. In $A\vec{x} = \vec{b}$

User ID	User Factors
0	x[0], x[1], x[2]
1	x[3], x[4], x[5]
2	x[6], x[7], x[8]
...	...

This scheme requires the user IDs to start at zero and be continuous. The same requirement exists for movie IDs.

The dataset's IDs start at one. Also, after shrinking the dataset, the IDs no longer form a continuous sequence.

Function for remapping IDs:

```
ALS_Model :: def remap_ids(self, id_list: list):
    """Remap ids in "id_list" to zero based values.
```

Member variables for remapping IDs:

```
ALS_Model :: movie_id_lookup - {data set movie id : ALS model movie id}
ALS_Model :: user_id_lookup - {data set user id : ALS model user id}
```

Fitting model to data call tree

```
ALS_Model :: def fit_to_data(self, user_ids, movie_ids, ratings):
  |→ solve_for_users(...)
  |→ solve_for_movies(...)
  |→ compute_training_error(...)
       |→ training_predict(...)
```

Prediction call tree

```
def predict(self, user_id, movie_id)
  |→ training_predict(...)
```

The prediction equation is:

prediction = median + $u_0m_0 + u_1m_1 + u_2m_2 + u_{bias}$

The "training_predict(...)" does not include the "median". During training, the median is removed from the $\vec{b}$ in $A\vec{x} = \vec{b}$.

Similar movies

File: similar_movies.py

SimilarMovieFinder call tree

```
SimilarMovieFinder :: find_similar_movie(self, movie_id, num_results = 10)
  |→ compare_two_movies(self, movie_id1, movie_id2)
  |→ genres_similar(self, movie_id1, movie_id2)
  |→ scaled_dot_product(self, movie_id1, movie_id2, verbose=False)
```

Similar genre filtering

Movies have multiple genres listed. In order for two movies to be considered similar, they must have a 50% over lap in the genres listed. This is implemented in:

```
SimilarMovieFinder :: genres_similar(self, movie_id1, movie_id2)
```

Dot product of movie reviews

For movies that pass the "genres_similar(...)" check, the dot product of user ratings is used to compute a cosine similarity.

Example:

	User #101 Rating	User #102 Rating	User #103 Rating	User #104 Rating	User #105 Rating
Movie #1000	4	5	2		2
Movie #1001	3	4		3	4

$$\frac{[4 \quad 5 \quad 2] \cdot [3 \quad 4 \quad 4]}{\|[4 \quad 5 \quad 2]\| * \|[3 \quad 4 \quad 4]\|} = \frac{40}{(\sqrt{45})(\sqrt{41})} = 0.931$$

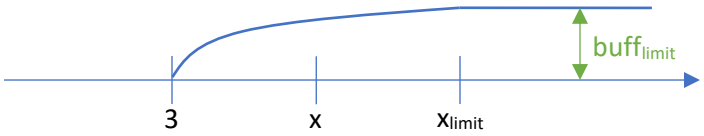
Note that only ratings from users who have reviewed both movies are used.

Dot product scaling

This cosine similarity metric, which is 0.931 in the example above, does not take into account the number of the users who have reviewed both movies. So if a user gives both movies "A" and "B" the same score, and no one else has watched both movies "A" and "B", that one review is sufficient to give a cosine similarity of one.

First, the function "scaled_dot_product(...)" will return a zero if two movies have too few reviewers in common.

Next, the cosine similarity of two movies with a large number of reviewers in common needs to be boosted. This scale factor should increase with the number of reviewers, but the increase should decay.



The natural log "ln(x)" is used as the increasing curve.

The code uses "3" as a base case - meaning a scale factor of "1" for two movies having three reviewers in common.

At a certain point, we have "buff_{point}" number of users, which corresponds to "x_{limit}" in the graph above. At this "x_{limit}", the scale factor reaches "1 + buff_{limit}", and there is no more increase for additional increase in users.

The "buff_{point}" and the "buff_{limit}" are variables in "scaled_dot_product(...)".

Derivation of equations used in the code:

$$\ln x_{limit} - \ln(3) = \text{buff}_{limit}$$

which simplifies to:

$$x_{limit} = 3e^{\text{buff}_{limit}}$$

```
x_limit = 3 * math.exp(buff_limit)
```

Let "n" be the number of users that have rated both movies.

$$\frac{n-3}{x-3} = \frac{\text{buff}_{point} - 3}{x_{limit} - 3}$$

which simplifies to:

$$x = 3 + \left(\frac{x_{limit} - 3}{\text{buff}_{point} - 3}\right)(n-3)$$

```
x = 3 + (x_limit - 3) * (n - 3) / (buff_point - 3)
```

The amount of buff is:

$$\text{buff} = \ln(x) - \ln(3)$$

```
buff = math.log(x) - math.log(3)
```

Dot product scale factor tuning

If the scale factor is set too low, the movies with just a few common reviewers are deemed highly similar.

If the scale factor is set too high, then popular movies in the same genre are boosted too much in terms of similarity.

In "main()", the scale factor has been tuned to make two "Star Wars" movies highly similar.

```
movie_id = 1196 # Star Wars: Episode V - The Empire Strikes Back (1980)

# id 1210 is Star Wars: Episode VI - Return of the Jedi (1983)
movie_finder.scaled_dot_product(movie_id, 1210, verbose=True)
```

This code prints out:

similarity = 0.9870188282262731 common reviewers = 162

The scale factor inside "scaled_dot_product(...)":

```
# Scale output due to number of common users.
# The settings below buff score by 21.6% for having 162 users in common
buff_limit = 0.216
buff_point = 162.0
```

These values are chosen to produce a similarity score of 1.20 for the two Star Wars movies: "Star Wars: Episode V - The Empire Strikes Back (1980)" and "Star Wars: Episode VI - Return of the Jedi (1983)".

Title Search

File: title_search.py

Class Summary

CsvData	Reads "movies.csv". Member variable: • movie_titles - {movie id: title string}

Index	Construction:
	`__init__(self, csv_data : CsvData, config : IndexerConfig)`
	
	Title search:
	`search(self, text)`

IndexConfig	Configuration options for how the index is built

Index build process call tree

```
IndexerConfig :: __init__()
  |→ define_ignored_words_set()
  |→ define_word_stems()


Index :: __init__(csv_data, config)
  |→ tokens, bigrams = tokenize_titles()
  |      |→ tokenize(text)
  |            |→ IndexerConfig :: text_splitter(word)
  |            |→ IndexerConfig :: word_filter(word)
  |                  | check for ignored words
  |                  |→ remove_punctuation(word)
  |                  |→ stem(word)
  |                  | check for ignored words
  |
  |→ tokens_index, bigrams_index = build_reverse_indices()
  |→ compute_word_counts()
```

Iteratively tuning the index

Run the program on the command line. This builds an "Index" object with existing settings, and calls various "Index::print_xxx(...)" functions to reveal the current status of the index. Analyze the output and modify "IndexerConfig" as necessary.

Ignored Words

A set of ignored words is defined inside "IndexerConfig :: define_ignored_words_set()"

The "Index :: print_frequent_tokens(...)" can be used to print frequent words, which is a starting point to consider what words should be ignored.

Text splitting

The title is broken into words in "IndexerConfig :: text_splitter(text)".

Currently the splits use:

- The Python default string.split()
- Split on '-' and on '/'

Stemming

This is done at "IndexerConfig :: stem(word)".

Most of the stemming work is done using the PorterStemmer (PorterStemmer.py). Some code is written to cover additional cases:

- remove the ('s) at the end of words
- apply a stemming dictionary, created via "IndexerConfig :: define_word_stems(self)".

The various "Index :: print_xxx(...)" functions give an idea of what the troublesome terms are.

- "Index :: print_non_alpha_num_words(...)" - these words might be difficult to reach, so these are candidates for dictionary based stemming. For example, "u.f.o" is listed, but the user might type in "ufo" instead. Som dictionary stem "u.f.o" that into "ufo", meaning both "u.f.o" and "ufo" are equivalent.
- "Index :: print_words_with_ending(...)" - look for words ending in "s", "ing", and "ion" - to check the performance of the Porter stemmer, and to do additional stemming if needed.

C++

General Visual Studio C++ remarks

- disable pre-compiled headers (currently listed at project properties → C/C++ → Precompiled Headers)
- change CPU to x64

LS

Location: \cpp\ls

Summary - multi-threaded implementation of conjugate gradient least squares and ALS in C++.

Conjugate Algorithm source

> **Numerical Optimization, 2nd Edition, by Jorge Nocedal, Stephen Wright**; Chapter 5 - Conjugate Gradient Methods, Algorithm 5.2 (CG)

Content

> **matrix.cpp, matrix.h** - multi-threaded versions of matrix multiply by vector, conjugate gradient, and ALS
>
> **main.cpp** - testing and benchmark functions
>
> > main 8 --- run program with 8 threads

Linux build command

> g++ -O3 main.cpp matrix.cpp matrix.h -o main -pthread -std=c++11

Benchmarks

Benchmarks have been taken on:

- desktop - Core i5-3350, 3.10 GHz 4 cores, 16 GB memory
- AWS c5.18xlarge - 3.0 GHz Intel Xeon Platinum, 72 vCPU (it's 32 cores, with 2 threads per core), 144 GiB memory

All run times are in milliseconds.

Machine	Desktop	AWS c5.18xlarge				
Threads	4	4	9	18	36	72
Adding two vectors, 10 times	486	193	113	88	93	109
Matrix times vector, 10 times	5144	3005	1306	805	772	520
Matrix transpose times vector, 10 times	8413	4249	2397	1932	2371	2853
Matrix transpose	6135	8162	3967	2159	1540	1209
Least squares 1 (run time per iteration)	1422	763	422	275	360	367
Least squares 2 (run time per iteration)	1111	1042	481	261	278	208

All matrices, referred to as A, are 27e6 rows by 2.8e6 columns, with 10 values per row.

The point of diminishing return has been shaded.

Benchmark Descriptions

- Adding two vectors - randomly generate two vectors of length 27e6 and add them.
- Matrix times vector - randomly generate A, and a vector of length 2.8e6, and multiply them.
- Matrix transpose times vector - randomly generate A, and a vector $\vec{x}$ of length 27e6, and compute $A^T\vec{x}$. This computation is done without computing A^T.
- Matrix transpose - randomly generate A and compute its transpose.
- Least squares 1 - randomly generate A and vector $\vec{b}$ of length 27e6. Solve $A\vec{x} = \vec{b}$ in the least squares sense using conjugate gradient. This version of least squares does not compute the transpose *A*.
- Least squares 2 - same as "least squares 1", except that version 2 will compute the transpose of A. This version uses more memory than version 1. Also, the computation of the transpose will take time. In the benchmarks, this version is faster if all 72 threads are used. This version is also faster on my desktop, where memory access is limited. AWS Intel Xeons have much larger cache memory than my desktop.
  Although the benchmarks here seem promising, with the actual MovieLens dataset, the least squares process use so few iterations that computing the transpose (A^T) failed to result in a faster algorithm.

Classes and Functions

matrix.h, matrix.cpp

Array, ArrayPtr, ArrayArray - containers to ensure variable deletion

ColVector - column vector of "double" values

- rand(...), rand_norm(...) - randomize the vector
- add(...), times_constant(...), dot_product(...), add_merge(...) - math

SparseMatrix - sparse matrix of "double" values stored in row major format

- multiply(...), transpose_multiply(...) - math
- count_columns(...), copy_to_column_major_form(...) - matrix transpose

Thread Creators

The "ColVector" and "SparseMatrix" member functions mentioned above each works on a subset of the column vector or matrix. Multiple copies of these member functions are meant to be run in parallel on different threads. The global functions below create those threads.

vect_rand(...), vect_rand_norm(...)
vect_add(...), vect_times_constant(...), vect_dot_product(...)
sparse_matrix_multiply(...), sparse_matrix_transpose_multiply(...)
sparse_matrix_transpose(...)

wait_and_delete_threads(...) - wait for all current threads to complete

cg_least_squares(...), cg_least_squares2(...) - solves $A\vec{x} = \vec{b}$ in the least squares sense

Input: SparseMatrix* A, ColVector* b
Output: ColVector* x

als(...)

Input: int* user_ids, int* item_ids, ColVector* ratings
Output: ColVector* user_factors, ColVector* item_factors

main.cpp

Command line argument: the number of threads to use, stored as "**thread_count**" global

Test functions are for verifying correctness:

test_vect_add(), test_vect_dot_product()
test_sparse_matrix_multiply(), test_sparse_matrix_transpose_multiply()
test_sparse_matrix_transpose()

test_least_squares(int algorithm, int num_tests)

algorithm = 1 uses cg_least_squares(...)
algorithm = 2 uses cg_least_squares2(...)

The "num_tests" is to check that the consistency of the conjugate gradient algorithm by running a large number of tests.

test_als()

Benchmark functions creates large sparse matrices and then measure the run time:

benchmark_col_vector_add(...)
benchmark_sparse_matrix_multiply(...)
benchmark_sparse_matrix_transpose_multiply(...)
benchmark_sparse_matrix_transpose(...)
benchmark_least_squares(...)

Sparse Matrix Format

Values are stored in row major order.

From source code

```
class SparseMatrix
{
    int* row_indices = nullptr; // row_indices[5] = 100 means first value on
row #5 is at values[100]
    int* col_indices = nullptr; // col_indices[5] = 90 means values[5] is on
column #90
    double* values = nullptr;
```

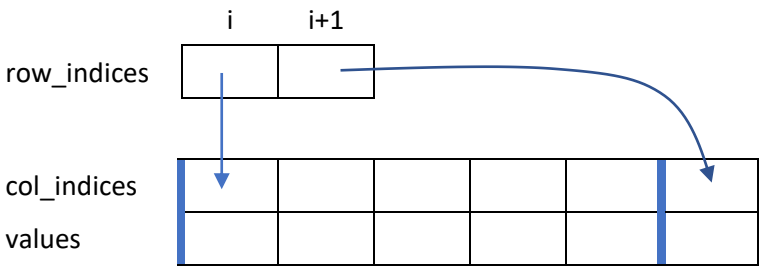
Example

$$\begin{bmatrix} & 1 & & 3 \\ 5 & & 7 & \\ & & & 9 \\ & 11 & & \end{bmatrix}$$

Index	Row	Column	Value
0	0	1	1
1		3	3
2	1	0	5
3		2	7
4	2	3	9
5	3	1	11
6			

row_indices = [0, 2, 4, 6]
col_indices = [1, 3, 0, 2, 3, 1]
values = [1, 3, 5, 7, 9, 11]

Locating a row within the sparse matrix



Transpose Multiply

This project solves least squares problems by solving the normal equation $A^T A\vec{x} = A^T\vec{b}$ using the conjugate gradient method.

The code approaches the problem $A^T * vector$ two different ways: direct multiplication and computation of the transpose.

Direct Multiplication

In this approach, $A^T\vec{v}$ is computed without computing A^T.

$$A = \begin{bmatrix}\vec{r}_0\\ \vec{r}_1\\ \vec{r}_2\end{bmatrix}, \qquad \vec{v} = \begin{bmatrix}c_0\\ c_1\\ c_2\end{bmatrix}$$

$$A^T\vec{v} = [\vec{r}_0 \quad \vec{r}_1 \quad \vec{r}_2]\begin{bmatrix}c_0\\ c_1\\ c_2\end{bmatrix} = c_0\vec{r}_0 + c_1\vec{r}_1 + c_2\vec{r}_2$$

The code implements a multi-thread version, where thread 0 would do $(c_0\vec{r}_0 + c_1\vec{r}_1 + c_2\vec{r}_2)$, and thread 1 would do $(c_3\vec{r}_3 + c_4\vec{r}_4 + c_5\vec{r}_5)$, and then the two column vectors would be added together.

As shown in the benchmark section, this approach does not scale well when more CPU cores are added. The cores can handle the compute, but the memory cannot keep up.

Transpose Computation

As the benchmark section shows, standard matrix multiplication $A\vec{v}$ is faster than transpose multiplication $A^T\vec{v}$.

The transpose A^T need to be computed only once per least square problem. So it might be worth it to compute A^T, and then just use the faster standard matrix multiplication.

Matrix transpose, for the sparse matrix situation, is re-ordering the terms and indices from a row major storage, to a column major storage.

For example:

$$A = \begin{bmatrix} & 1 & & 3\\ 5 & & 7 & \\ & & & 9\\ & 11 & & \end{bmatrix}$$

A.values = [1, 3, 5, 7, 9, 11]

A^T.values = [5, 1, 11, 7, 3, 9]

The question is where will a particular value, such as 3, go?

The answer is to count how many terms each column has. The 3 goes into index 4 because there are 4 terms in the columns to the left of 3.

A two threaded example

The following diagrams show computing the transpose using two threads.

Thread 0 (t0) is responsible for the upper half of the original matrix *A*, while thread 1 (t1) is responsible for the lower half.

Each thread counts the number of elements in each column:

t0	2	0	1	4	column_counts[0]
t1	0	1	2	1	column_counts[1]

Combine the count from two threads into:

0	2	1	3	5	column_count_total

Note "column_count_total" has one more element than column_counts.

Accumulate the counts in "column_count_total":

0	2	3	6	11	column_count_total

This will be the "row_indices" for the transposed matrix A^T.

Compute indices for each column. Thread 0's indices are copied from column_count_total:

0	2	3	6	col_indices[0]

Thread 1's indices are col_indices[0] + column_counts[0]

	0	2	3	6	col_indices[0]
+	2	0	1	4	column_counts[0]
	2	2	4	10	col_indices[1]

The "col_indices" can be computed from the "column_counts" in place. So in the code, the "col_indices" is actually the same as "column_counts".

So after computing "col_indices", we have:

0	2	3	6	col_indices[0] for thread 0
2	2	4	10	col_indices[1] for thread 1

Example interpretations:

Thread 0, column 2 item, goes into index 3, where there is 1 spot {3}.

Thread 1, column 2 item, goes into index 4, where there are two spots {4, 5}.

The "col_indices" arrays give the threads enough information to parallel copy from (A.col_indices and A.values) to (A^T.col_indices and A^T.values).

Call Tree

```
// m^T => mt
void sparse_matrix_transpose(SparseMatrix* m, SparseMatrix* mt)
  |→ void SparseMatrix::count_columns(...)
  |→ void add_arrays(...)
  |→ void compute_col_indices(...)
  |→ void SparseMatrix::copy_to_column_major_form(...)
```

Benchmarks remarks

The computation of matrix A will roughly double the memory usage of the conjugate gradient process.

It takes time to compute A, so there is only net savings in time if there is a large number of $A^T\vec{v}$ kind of calculations. The number of iterations in the conjugate gradient procedure is dependent on the problem. The benchmarks show that for AWS machines, which are Intel Xeons, computing matrix A is only worth it for the very largest instance. Yet, this situation is not economical, as the AWS price scales exactly with the number of cores. So going from 16 cores to 32 cores will double the price, but will not half the execution time.

However, benchmark on the home computer shows that even for 4 cores, computing the matrix transpose can lead to speed increase, in contrast with the AWS situation, where 72 threads (32 cores) is needed to justify the matrix transpose computation. The explanation could be that the home computer's memory interface is much less capable than the Xeon server platform. The CPU cache is also much smaller - the home computer's CPU has a 6 MB L3 cache while Intel Xeon Platinum typically has 15 ~ 40 MB L3 cache. The better memory capabilities of the Intel Xeon improves the matrix transpose multiply in comparison with the standard matrix multiply (see "row reordering" section), making the gain of using more standard multiply less effective.

ALS A Matrices

Assume there are three factors. It's one equation per rating.

Solving for user factors

The equation for (user x, item y, rating r):

$$\underbrace{\begin{bmatrix} \cdots & i_{y0} & i_{y1} & i_{y2} & 1 & \cdots \end{bmatrix}}_{\text{user\_A}} \underbrace{\begin{bmatrix} \vdots \\ u_{x0} \\ u_{x1} \\ u_{x2} \\ u_{x3} \\ \vdots \end{bmatrix}}_{\text{user\_factors}} = \underbrace{\begin{bmatrix} \vdots \\ r \\ \vdots \end{bmatrix}}_{\text{ratings}}$$

Solving for item factors

The equation for (user x, item y, rating r):

$$\underbrace{\begin{bmatrix} \cdots & u_{x0} & u_{x1} & u_{x2} & \cdots \end{bmatrix}}_{\text{item\_A}} \underbrace{\begin{bmatrix} \vdots \\ i_{y0} \\ i_{y1} \\ i_{y2} \\ \vdots \end{bmatrix}}_{\text{item\_factors}} = \underbrace{\begin{bmatrix} \vdots \\ r \\ \vdots \end{bmatrix}}_{\text{ratings}}$$

Conjugate Gradient Termination

The conjugate gradient termination has been checked by going through a large number of randomly generated problems. The check is at main.cpp, function "benchmark_least_squares".

The source of the algorithm is **Numerical Optimization, 2nd Edition, by Jorge Nocedal, Stephen Wright**; Chapter 5 - Conjugate Gradient Methods, Algorithm 5.2 (CG).

The termination method used is the percentage decrease in the residual term. In algorithm 5.2, this corresponds to the β term. The residual vectors are $\vec{r}_{k+1}$ and $\vec{r}_k$, and $\beta = \frac{(\vec{r}_{k+1}^T \vec{r}_{k+1})}{(\vec{r}_k^T \vec{r}_k)}$. Conjugate gradient does "get stuck" sometimes, making little progress in a single iteration. So the percentage decrease criteria has to be met on two consecutive iterations.

Other Ideas

Row Reordering (canceled)

This is another attempt to improve matrix transpose multiply - but the benchmark results were poor, so those benchmarks were commented out in the code:

Inside main.cpp, under main():

```
// benchmark_sparse_matrix_transpose_multiply(.., true);
```

The "true" flag turns on the "row reordering" option. The code remains in the "C++ LS" project, but this option is no longer used.

The idea is to re-order the rows to improve the matrix transpose multiplication. As a recap:

$$A^T\vec{v} = [\vec{r}_0 \quad \vec{r}_1 \quad \vec{r}_2]\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = c_0\vec{r}_0 + c_1\vec{r}_1 + c_2\vec{r}_2$$

In terms of memory access, both the $\vec{r}$ and $\vec{v}$ is being accessed sequentially. Also, they are accessed exactly once - so this part of the memory access is already optimal. Yet the benchmarks show that this operation does not scale well beyond 9 thread. The problem is presumed to be writing the result to the memory.

From the code at matrix.cpp, SparseMatrix::transpose_multiply(...):

```
for (int j = row_start; j <= row_end; j++)
{
    // values[j] is on row 'i', at location col_indices[j]
    int col_number = col_indices[j];
    result_vec[col_number] += c_i * values[j];
}
```

The term "result_vec[col_number]" is fetched to the CPU, accumulated, then written back to memory. The problem is that since the matrix is so large, and so sparse, the "col_number" is jumping around a lot, so cache locality is poor.

The idea is to sort the rows according to their columns, so that the "col_number" will not jump around so much, and therefore "result_vec[col_number]" will stay in the CPU cache longer. This sorting does not have to be perfect - to improve CPU cache, it's only sufficient to have consecutive rows have roughly the same column number.

This idea re-orders the rows, and in the context of the $(c_0\vec{r}_0 + c_1\vec{r}_1 + c_2\vec{r}_2)$ expression, this idea will deteriorate the cache locality for the $\vec{c}$ and $\vec{r}$ terms. So it's not certain if there is any net improvement to be had. The idea has been implemented in the code, and can be activated by using the "use_sorted_rows = true" option. However, benchmarks were poor, and the idea was abandoned.

Implementation:

Each row is associated with a column value - based on the row's first value's column number. Only the first "n" bits of the column number is used, since an exact sort is not

needed. The rows are grouped into "bins", each bin number being the first "n" bits of column numbers.

Count the number of rows per bin:

bin_counts	3	2	0	2	1
bin	0	1	2	3	4

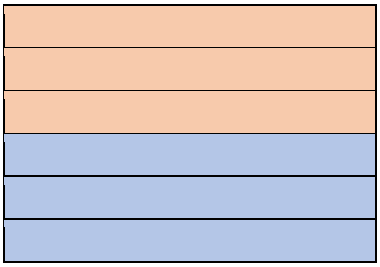
then accumulate the bin numbers:

bin_indices	0	3	5	5	7
bin	0	1	2	3	4

The "bin_indices" tells where each row goes. This produces a partially sorted list of rows.

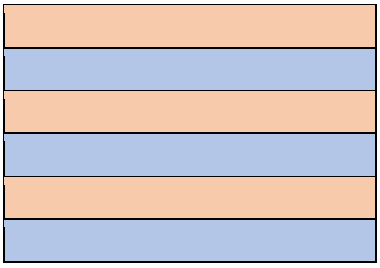
Interleaved Multi-Threading (idea not tested)

The multi-threading in the code splits the matrix into continuous blocks. As an example, if the matrix has 6 rows and there are 2 threads, then the matrix is split as:



The two threads read and write memory that are far away from each other.

An alternative is to interleave the two threads:



Interleaved threads might scale better. Thread scaling is a matter of optimizing memory access, and interleaved threads might have better memory access locality.

Python Wrapper

This project is about accessing the C++ least squares and ALS code in Python.

Files:

- **C++ library: cpp\ls_lib** - builds library file for use with Python
  - Windows output file name: cpp_ls.dll
  - Linux output file name: cpp_ls_lib.so
- **Python wrapper: cpp\python**
  - cpp_ls.py - Python wrapper for C++ library
  - cpp_ls_test.py - Python tester for C++ library

C++ Wrapper Files

	Windows	Linux
matrix.h	x	x
matrix.cpp	x	x
ls_windows_dll.cpp	x	
ls_linux_dll.cpp		x

Linux C++ Differences

Windows C++ DLL entry points look like:

```
extern "C" __declspec(dllexport) void set_thread_count(int thread_count)
```

Linux C++ library entry points do not have the "__declspec(dllexport)" phrase

Linux Build

g++ -O3 -fPIC -shared matrix.cpp matrix.h ls_linux_dll.cpp -o cpp_ls_lib.so -pthread -std=c++11
chmod 644 cpp_ls_lib.so

C++ Changes from the LS project

SparseMatrix, ColVector allows external construction, with a "delete_values" flag, - so their destructors do not try to free the arrays.

"use_sorted_rows" flag removed; related code removed

Running the Python Test Code

Build the C++ code to produce a library file. On Windows, this is a "dll", and on Linux this is an "so".

Copy the library file to the directory containing the Python code.

Run cpp_ls_test.py

Python Test Code Call Tree

File: cpp_ls_test.py

```
main()
  |→ test_cg_least_squares()
  |      |→ convert_dense_matrix_to_sparse_format(A)
  |
  |→ test_als()
         |→ als_predict(...)
```

Python Wrapper Interface

`has_dll_loaded()`	This is a health check function. It sets the thread count to a random number, and then read the thread count back.
`set_thread_count(thread_count)`	When the C++ library is loaded, the thread count is automatically set to the CPU count.
`cg_least_squares(...)`	Solves $A\vec{x} = \vec{b}$ in the least squares sense using conjugate gradient. This function requires matrix *A* to be expressed as a sparse matrix format.
`als(...)`	Input is a set of (user id, item id, rating), expressed as three parallel Numpy arrays. Output is an ALS model (user_factors, item_factors), expressed as two Numpy arrays.

`test_als()` - determination of number of users and items

The test data is generated such that every user reviews every item.

The least squares method is stable only if there are sufficient number of equations in the training set.

For example, if 80% of the data is set aside for training, then

(number of items) * 80% > (number of user factors)

In programming terms,

$$\text{num\_items} = \text{math.ceil}\left(\frac{\text{num\_user\_factors} * k}{\text{training\_set\_ratio}}\right), \qquad k > 1$$

The "k = 1" gives the minimum number of items, so to have enough equations. However, this is not enough in practice because there is a parallel set of requirement for "num_users". The training sets are generated after a random shuffle, and any imperfect split will result in insufficient items or users.

Even if the data is grouped according to users and the training set is generated by splitting each individual sets of user data, this only guarantees that there are sufficient ratings per user in the training set. This does not guarantee sufficient ratings per item in the training set.

Therefore, the "k" value is set to be much higher than 1.

Linux Python Differences

Numpy "int" must be explicitly specified as "int32" for the code to work on both Linux and Windows.

Example:

```
user_ids = numpy.zeros(num_users * num_items, dtype=numpy.int32)
```

Simply "numpy.int" worked on Windows but did not work on Linux. It seems that Linux numpy.int does not default to 32 bit.

Full Dataset

Project location - python\full_data

config.py - various directories, such as: shared_directory, txt_dir, in_dir, als_dir, ...

movie_lens_data, movie_lens_data_proc - reads in the raw ".csv" data and writes out binary (pickle) data needed by various models

Modifications from "100k dataset"

More list based data structures

The "100k dataset" uses largely dictionary based data structures. The full data set uses more list based data structures. For example, user_ratings_train and user_ratings_test use to be a dictionary leading to two lists:

{user_id: [movie_ids_list, ratings_list]}

now it's a list of tuples

[(user_id, [(movie_id, rating)])]

Test sets are checked to have at least two different rating levels

This prevents the model from automatically getting a 100% match agreement.

Pickle Binary files

The "100k dataset" project reads in everything from the original csv. With the full dataset, intermediate results are saved as binary files using pickle.

Multiprocessing for repeated data generation tasks

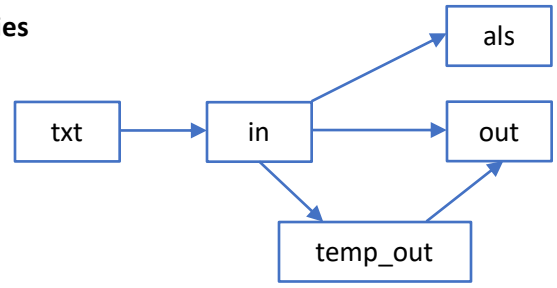
There is no multiprocessing for some one-time tasks. For example, reading in the "ratings.csv" and saving it as "user_ratings.bin" is time consuming, but "ratings.csv" currently changes only once per year. So, there's no attempt to speed up the processing of "ratings.csv".

Data Processing

Linux Commands for AWS

wget http://files.grouplens.org/datasets/movielens/ml-latest.zip	Download
unzip ml-latest.zip	Unzip

Directories



txt - the original data files, which are ".csv" text files

in - input files. They have mostly the same information as the "txt" files, but in binary form. The files are usually Python lists or dictionaries, so they are ready to be used by software.

out - output files

als - the ALS procedure has so many files that they are grouped into its own directory

temp_out - cluster processing temporary output files. These files will be merged into a single file, which then goes to the "out" directory.

Software Layers, Memory Use, and Run Time

		cluster	memory use, run time (cpu_count=18)
Application Tasks	randomize_training_set.py		14.9 GB, 1:52
	train_als_models.py		19 GB, 4:20*
	build_similar_movies_db.py	x	50.6 GB, 22:36
	build_title_search_index.py		
	download_tmdb_data.py		
Common Support Routines	movie_lens_data.py		
	movie_lens_data_proc.py		

Memory benchmark done using the "peak_mem.py" script.

Tests are done on AWS 72 core machine, with "cpu_count=18 overwrite" flags.

The "train_als_models.py" is tested using: "sudo python3 train_als_models.py overwrite training_set_ratio=1 cpu_count=18 als_thread_count=18"

*ALS train time can vary due to the use of a random starting vector. In five separate tests, total ALS training time varied between 0:54 to 1:50, for a total run time of 3:49 ~ 4:45.

Input and Output Files

data/txt

- links.csv (movie id, IMDB id, TMDB id)
- movies.csv (movie id, title, genres)
- ratings.csv (user id, movie id, rating)
- tags.csv (movie id, tag)

data/in

- genre_counts.bin - {genre id : genre count}
- genre_ids.bin - {genre string : genre id}
- links.bin - {movie_id: [imdb_str, tmdb_str] }
- movie_genres.bin - {movie id : set of genre ids}
- movie_medians_full.bin - {movie id : movie median rating}
- movie_medians_train.bin - {movie id : movie median rating}
- movie_ratings.bin - [(movie_id, {user_id: rating})]
- movie_tags.bin - {movie_id: {tag id: ln(count) + 1} }
- movie_titles.bin - {movie_id: title}
- tag_counts.bin - {tag id: count}
- tag_ids.bin - {tag string : tag id}
- tag_names.bin - {tag id: tag string}
- user_ratings.bin - [(user id, [(movie id, rating)])]
- user_ratings_test.bin - [(user id, [(movie id, rating)])]
- user_ratings_test_length.bin - length of "user_ratings_test"
- user_ratings_train.bin - [(user id, [(movie id, rating)])]

Running "download_tmdb_data.py" requires an API key from "www.themoviedb.org"

data/als - there's a set of files for each ALS model. For factor 3:

Input to the ALS model:

- als3_movie_ids.bin - {standard movie id : zero based movie id}
- als3_user_ids.bin - {standard user id : zero based user id}
- als3_user_ratings_test.bin - [(user id, [(movie id, rating)])]
- als3_user_ratings_test_length.bin - length of "als3_user_ratings_test" list
- als3_user_ratings_train.bin - [user_ids_train_numpy_array, movie_ids_train_numpy_array, ratings_train_numpy_array]

ALS model:

- als3_item_factors.bin - numpy array of item factors
- als3_user_factors.bin - numpy array of user factors

data/out

- similar_movies.bin - {movie_id: [similar_movie_ids]}
- tmdb_data.bin - {movie_id: [tmdb_id_str, poster_file_name]}
- tmdb_bad_id.bin - {movie_id: tmdb_id_str}
- title_search_index.bin - a title_search.py :: Index object

Evaluation results - all have the same format

- als3_eval_results.bin - [(user id, rank agreement percentage)]
- als5_eval_results.bin, als7_eval_results.bin, als9_eval_results.bin, als11_eval_results.bin
- median_eval_results.bin, tag_count_eval_results.bin, tag_ls_eval_results.bin

movie_lens_data.py

Data loading functions:

get_input_obj(object_name) - Use pickle to load "object_name" from "in_dir".
get_als_obj(object_name) - Use pickle to load "object_name" from "als_dir".
get_output_obj(object_name) - Use pickle to load "object_name" from "out_dir".

One-time binary files

	Requires	Produces
read_links_csv(...)	links.csv	links.bin
read_movies_csv(...)	movies.csv	genre_counts.bin genre_ids.bin movie_genres.bin movie_titles.bin
create_movie_ratings(...)	ratings.csv	movie_ratings.bin
create_user_ratings(...)	ratings.csv	user_ratings.bin
read_tags_csv(...)	tags.csv	movie_tags.bin tag_counts.bin tag_ids.bin tag_names.bin

Repeatedly generated files

The "_mp" suffix means the function uses multiprocessing.

	Requires	Produces
refresh_training_sets_mp(...)	user_ratings	movie_medians_train.bin user_ratings_test.bin user_ratings_test_length.bin user_ratings_train.bin leaves "user_ratings_train" and "user_ratings_test" in process memory
als_data_set_shrink_mp(...)	"user_ratings_train" and "user_ratings_test" are still in process memory	One set of files per ALS factor: als3_movie_ids.bin als3_user_ids.bin als3_user_ratings_test.bin als3_user_ratings_test_length.bin als3_user_ratings_train.bin

als_train(...)	One set of files per ALS factor: als3_movie_ids.bin als3_user_ids.bin als3_user_ratings_train.bin	One set of files per ALS factor: als3_item_factors.bin als3_user_factors.bin

movie_lens_data_proc

Multiprocessing support for "movie_lens_data"

start_processes() - Spawns processes

Sending data to processes

send_same_data(data_dict) - Send the same data to all processes.

split_list_and_send(data_list, var_name) - Split a list and send the pieces to each process.

split_range_and_send(start_index, length, range_name) - example: start_index = 0, length = 10, cpu_count = 4

Process #	range_name_start	range_name_length
1	0	2
2	2	3
3	5	2
0	7	3

The current process gets the final split.

Running function on data in parallel

run_function(function_name : str, arg_dict)

Merging data from processes

concat_var_into_list(var_name) - Retrieve "var_name" from all processes and use list concatenate to combine them into a single list.

	Combination Method
concat_var_into_list(...)	list concatenate (+)
append_var_into_list(...)	list.append(...)
concat_var_into_numpy_array(...)	numpy.concatenate(...)
update_var_into_set(...)	set.update(...)
update_var_into_dict(...)	dictionary.update(...)
add_merge_var_into_dict(...)	dictionary summation of common keys

or_var_into_boolean(...)	"or" boolean variables into a single boolean

Process memory clean up

delete_variable(var_name)
clear_all_data()

end_processes() - Shutdown processes.

ALS Data Preparation

Location: movie_lens_data.py :: als_data_set_shrink_mp(...)

Overview

The ALS can only be applied to users and movies with sufficient number of ratings.

For an ALS model of size "factor", the movies in the data has to have at least "factor" number of ratings, and the users has to rate "factor+1" movies.

To support distributed processing, the rating data is distributed across different processors. There are two options:

- group data by user id, as in [(user id, [(movie id, rating)])], then spread the list around different processors
- group data by movie id, as in [(movie id, [(user id, rating)])]

The algorithm in "movie_lens_data::_als_data_set_shrink(...)" group the data by user id.

Grouping by user_id should have faster processing speed

The full data set has 280k users but only 58k movies. So on average, there are far more ratings associated with each movie, compare to ratings for each user.

The requirement for users are also higher to begin with - each user needs "factor +1" movie ratings, while each movie only require "factor" movie ratings.

Therefore, more users will be dropped from the data set compared with movies.

Grouping the data by user id makes it easier to drop data according to users.

Movies that fail to show up "factor" times within each processor will have to be counted by merging.

Grouping by movie_id should use less memory

Grouping by a field is like compressing the data using that field. The more repetitive the field, the higher the compression. Movie id is far more common than user id. Grouping by movie id results in fewer lists, and so there might be less memory over allocation as well.

Call Trees

variables eventually pickled to disk

process memory initialization

```
refresh_training_sets_mp(user_ratings, training_set_ratio = 0.80)
  |→ _proc.split_list_and_send(user_ratings, "user_ratings")
  |
  |→ _proc.run_function("_compute_training_set", ...)
  |     |→ _compute_training_set(...)
  |           |→ _shuffle(...)
  |           |→ my_util.has_different_ratings(...)
  |           _process_data["user_ratings_train"] = ...
  |           _process_data["user_ratings_test"] = ...
  |
  |→ user_ratings_train = _proc.concat_var_into_list(...)
  |→ user_ratings_test = _proc.concat_var_into_list(...)
  |
  |→ _proc.run_function("_extract_movie_ratings", ...)
  |      _process_data["movie_ratings"] = ...
  |
  |→ movie_ratings_lists = _proc.append_var_into_list(...)
  |→ movie_ratings = _merge_movie_ratings_lists(movie_ratings_lists)
  |→ _proc.split_list_and_send(movie_ratings ,...)
  |
  |→ _proc.run_function("_compute_medians", ...)
  |      _process_data["movie_medians"] = ...
  |
  |→ movie_medians = _proc.update_var_into_dict(...)
  |
  |→ _proc.delete_variable("user_ratings") # probably not needed in future
  |→ _proc.delete_variable("movie_ratings") # intermediate result only
  |→ _proc.delete_variable("movie_medians") # intermediate result only
     # user_ratings_train, user_ratings_test remain in memory
```

```
als_data_set_shrink_mp(movie_medians_train, factors_list, ...)
   |   present in process memory from refresh_training_sets_mp(...):
   |   _process_data["user_ratings_train"], _process_data["user_ratings_test"]
   |
   |-> _proc.send_same_data({"movie_medians": ...})
   |
   |-> _proc.run_function("_drop_users", ...)
   |   has_changed = _proc.or_var_into_boolean(...)
   |
   |-> _proc.run_function("_count_movies", ...)
   |       _process_data["movie_counts"] = ...
   |
   |   movie_counts = _proc.add_merge_var_into_dict(...)
   |
   |-> _proc.run_function("_drop_movies", ...)
   |
   |-> _proc.run_function("_collect_ids", ...)
   |       _process_data["movie_ids"] = ...
   |       _process_data["user_ids"] = ...
   |
   |-> movie_ids = _proc.update_var_into_set(...)
   |-> user_ids = _proc.update_var_into_set(...)
   |
   |-> _proc.send_same_data({"als_movie_ids": ..., "als_user_ids": ...})
   |       _process_data["als_movie_ids"] = ...
   |       _process_data["als_user_ids"] = ...
   |
   |-> _proc.run_function("_convert_training_data_to_numpy", ...)
   |       _process_data["user_ids_train_numpy"] = ...
   |       _process_data["movie_ids_train_numpy"] = ...
   |       _process_data["ratings_train_numpy"] = ...
   |
   |-> user_ids_train_numpy = _proc.concat_var_into_numpy_array(...)
   |-> movie_ids_train_numpy = _proc.concat_var_into_numpy_array(...)
   |-> ratings_train_numpy = _proc.concat_var_into_numpy_array(...)
   |
   |-> user_ratings_test = _proc.concat_var_into_list(...)
```

randomize_training_set.py

Overview

Input requirement: original ".csv" files in "txt" directory

Outputs: user_ratings_train, user_ratings_test, movie_medians_train - different each time

Command line options: overwrite, training_set_ratio, cpu_count

See top of the source file for details

Call Tree

```
main()
  |→ _proc.start_processes(cpu_count)
  |
  |→ movie_lens_data.read_movies_csv(...)
  |→ movie_lens_data.read_tags_csv(...)
  |→ movie_lens_data.create_user_ratings(...)
  |
  |→ movie_lens_data.refresh_training_sets_mp(...)
  |     (see movie_lens_data's call trees section)
  |
  |→ _proc.end_processes()
```

train_als_models.py

Overview

Input requirement: original ".csv" files in "txt" directory

Outputs: one set of ALS files for each ALS factor. For factor 3: als3_movie_ids, als3_user_ids, als3_user_ratings_test, als3_user_ratings_train, als3_item_factors, als3_user_factors

Command line options: overwrite, training_set_ratio, cpu_count, als_thread_count, algorithm

See top of the source file for details

Call Tree

```
main()
  |→ _proc.start_processes(cpu_count)
  |
  |→ movie_lens_data.refresh_training_sets_mp(...)
  |
  |→ movie_lens_data.als_data_set_shrink_mp(...)
  |     (see movie_lens_data's call trees section)
  |
  |→ _proc.end_processes()
  |
  |→ movie_lens_data.als_train(...)
        |→ user_factors, item_factors, iterations = cpp_ls.als(...)
```

build_similar_movies_db.py

See "100k dataset → Similar movies" for basics

Overview

Input requirement: original ".csv" files in "txt" directory

Output: similar_movies

Command line options: overwrite, **cpu_count** (vital for AWS)

See top of the source file for details

Resource usage: 3 GB per thread

Building a table of similar movies is the most resource consuming task in this project.

With a 16 GB, quad core computer, exiting everything should leave sufficient memory. However, the process is very slow. The script will print an estimate after each core has processed 200 movies.

On AWS "compute optimized" instances, the machines have only 2 GB per hyper thread. The solution is to use the "**cpu_count**" parameter to limit the processes spawned by "worker_server".

Modifications from "100k dataset"

- **Automatic tuning** - the "buff_limit" and "buff_point" parameters were manually tuned back in "100k dataset". Here these are automatically tuned such that when searching for movies similar to "Star Wars, The Empire Strikes Back", the movie "Star Wars: Return of the Jedi" needs to be in the top two.
- **Reliable data is prioritized and added as a third filter** - In the context of cosine similarity, "reliable" means two movies having a large number of reviewers in common.

  For example, 1000 movies pass through the first two filters of "genre check" and "minimum number of reviewers in common". The search only wants the 20 most similar movies, and it's not necessary to consider all 1000 movies. Sort the 1000 movies according to reviewers in common - so that the movies with the most reviewers in common are deemed most reliable for the cosine similarity algorithm. Take the top 400 (20 * 20) movies that have the most reviewers in common, and compute similarity scores only for those 400. The other 600 movies are ignored.
- **multi-processor or cluster build** - both are supported. As of late 2018, the full Movie Lens data set requires 3 GB per process.
- **randomization of movies** - when "movie_ratings.bin" is build from the ".csv" file, the movies are randomized. By default, older movies have more reviews, and so without randomization the

process responsible for the earlier movies will take up much more time. Randomizing the movie listing allows the "movie_ratings" list to be divided up without workflow management.

Automatic Tuning

This is done by "SimilarMovieFinder :: tune(...)". Two highly similar movies' ids have to be supplied. The number of common reviewers between the two movies is the "buff_point" parameter. The "buff_limit" is gradually increased until the two movies rank very high in the similarity ranking.

During tuning, the "buff_limit" value is referred to as "buff", and is estimated by:

$$\frac{buff\_new}{buff\_old} = \frac{top\_score}{score}$$

buff_old - the "buff_limit" value in the current iteration

score - the current score between the two movie ids (of the two similar movies)

top_score - the highest score in the similarity search for movie_id1

The "buff_new" is only sufficient to boost the similarity score to match "top_score". Any increase in "buff_limit" will increase all similarity scores. So the "buff_new" for the new iteration is:

$$buff\_new = (buff\_old) * \frac{top\_score}{score} + 0.01$$

Call Tree

```
main()
  |→ movie_lens_data.read_movies_csv()
  |→ movie_lens_data.create_movie_ratings(...)
  |→ movie_finder = SimilarMovieFinder(...)
  |→ movie_finder.tune(movie_id1, movie_id2, ...)
  |     |→ find_movie_index(...)
  |     |→ _scaled_dot_product(...)
  |     |→ find_similar_movie(...)
  |
  |→ if cluster.cluster_info is None:
        build_locally(...)
    else:
        build_with_cluster(...)
```

variables eventually pickled to disk

process memory initialization

```
build_locally(movie_genres, movie_ratings, buff_point, buff_limit, ...)
  |→ _proc.start_processes(cpu_count)
  |→ _proc.send_same_data({
  |       "movie_genres": ..., "movie_ratings": ..., "buff_point": ...,
  |       "buff_limit": ...
  |   })
  |
  |→ _proc.split_range_and_send(0, len(movie_ratings), "movie_ratings")
  |      _process_data["movie_ratings_start"] = ...
  |      _process_data["movie_ratings_length"] = ...
  |
  |→ _proc.run_function("_find_similar_movies", {})
  |      _process_data["similar_movies"] = ...
  |
  |→ similar_movies = _proc.update_var_into_dict(...)
  |
  |→ _proc.end_processes()
```

```
build_with_cluster(buff_point, buff_limit, length, output_file_name)
  |→ cluster.send_command({
  |       "op": "distribute",
  |       "worker_op": "build_similar_movies_db", ...})
  |
  |   worker_server.py :: build_similar_movies_db_setup(...)
  |
  |   worker_server.py :: build_similar_movies_db(...)
  |     |→ worker_process.run_function("_build_similar_movies_db")
  |     |
  |     |      _build_similar_movies_db(...)
  |     |         |→ movie_finder = build_similar_movies_db.SimilarMovieFinder(...)
  |     |         |→ movie_finder.find_similar_movie(...)
  |     |
  |     |→ write_output_to_disk(...)
  |
  |→ cluster.wait_for_completion()
```

Cost Benchmark

	total_units / total_time	spot $ / hour	cost
Intel (m5.4xlarge, 16 CPU, 64GB, 2.5 GHz Intel Xeon® Platinum 8175)	16005 / 602 = 26.59 5969 / 244 = 24.46 20637 / 786 = 26.26 5292 / 232 = 22.81 5986 / 217 = 27.59 --- 53889 / 2081 = 25.90	$0.2704	$0.1563

build_title_search_index.py

This part is mostly the same as the "100k dataset title search" - see that documentation for the basics.

Fewer member variables

Pickle is used to save "Index" to disk and restore it back. Since pickle has namespace related quirks, the pickle package is used on the actual member variables of the "Index" class, rather than the whole class. Fewer member variables mean less to pickle to disk.

Construction routine changed

The constructor now accepts a file name:

```
__init__(self, config : IndexerConfig, file_name)
```

To build an index for the first time, use a file name of "None". Then call

```
build(self, movie_titles : dict, file_name = None)
```

Output is redirected to disk

This is due to the Python "print" function throwing "UnicodeEncodeError" exceptions for certain characters.

For example, the following throws an exception:

```
print("中文") # Fails on Python 3.5 and Windows 10
```

Changes from "100k dataset"

The index file is now saved to disk as "title_search_index.bin".

download_tmdb_data.py

Purpose

In the "links.csv" file, for each movie_id, there is a "tmdb_id". Query www.themoviedb.org to confirm this ID actually exists, and retrieve a "poster_file_name" for the movie.

API key required

The variable "**api_key**" must be correctly set for this script to work. The free version has a rate limit attached to it.

The "pause_time" should be set so the API rate limit is not exceeded.

This script save results periodically

This script is designed to work over a long period of time, over multiple invocations. Results are saved to "tmdb_data.bin" periodically. The TMDB IDs that are bad are saved to "tmdb_bad_id.bin", which prevents redownloading data from bad IDs.

No all bad IDs are in "tmdb_bad_id.bin"

Only TMDB IDs that are not recognized by "www.themoviedb.org" are saved as "tmdb_bad_id.bin".

Not having a poster does not mean the ID is bad.

A better way to clean the data is to match the title from "movies.csv" against the title downloaded from www.themoviedb.org. Often, these titles do not match exactly. Some movies are known by multiple titles. Even when a partial matching algorithm is used, there are many mismatches that require human intervention to resolve.

Model Evaluation

Overview

Model evaluation uses the cluster setup documented in the section "Basics → distributed work".

From an application perspective, a cluster operation is a trigger followed by three handler functions. For example, in the case of evaluating the median model:

median_predictor.py :: main()

```
cluster.send_command({
    "op": "distribute",
    "worker_op": "median_eval",
    "length": length
})
```

The "**median_eval**" message is then handled by:

```
worker_server.py :: median_eval_setup(...)
worker_server.py :: median_eval(...)
  |→ worker_process.run_function("_median_eval")
     worker_process.py :: _median_eval(...)
```

See the ""Basics → distributed work"" documentation for information on how the system works.

Memory Usage and Run Time

	memory use, run time (cpu_count=18)
als_predictor.py	7.8 GB, 25 seconds
median_predictor.py	4.2 GB, 4 seconds
tag_count_predictor.py	8.8 GB, 62 seconds
tag_ls_predictor.py	8.6 GB, 91 seconds

All model evaluations use the cluster setup. There is a single worker node, an AWS 72 core machine: "sudo python3 worker_server.py cpu_count=18".

Memory benchmark done using the "peak_mem.py" script.

Triggers and Handlers

Trigger File "worker_op"	worker_server.py	worker_process.py
als_predictor.py "als_eval"	als_eval_setup(...) als_eval(...)	_als_eval(...)
median_predictor.py "median_eval"	median_eval_setup(...) median_eval(...)	_median_eval(...)
tag_count_predictor.py "tag_count_eval"	tag_count_eval_setup(...) tag_count_eval(...)	_tag_count_eval(...)
tag_ls_predictor.py "tag_ls_eval"	tag_ls_eval_setup(...) tag_ls_eval(...)	_tag_ls_eval(...)

The Model Interface in "my_util.py"

```
class Model:
    """Interface for movie prediction models."""
    def predict(self, movie_id : int):
        """Predicts a rating for the given movie_id. Returns None if
        no prediction can be made."""
        return None
```

The different models all follow this interface and implement a "predict" function.

```
als_predictor.py :: class ALS_Model(my_util.Model)
tag_count_predictor.py :: class UserProfile(my_util.Model)
tag_ls_predictor.py :: class UserProfile(my_util.Model)
```

These models are all evaluated by "worker_process.py :: _test_model(...)"

```
def _test_model(model : my_util.Model, movie_ratings):
    """Test "model" using "movie_ratings".

    :param model: a derivative of my_util.Model
    :param movie_ratings: [(movie_id, rating)]
    :return: prediction rank agreement
    """
```

The exception to this pattern is the "median" predictor. This "model" is just looking up the movie median in a dictionary, so it's deemed too simple and this pattern was not applied.

Tag Least Squares Limitation

The "tag least squares" model does not use all tags and genres. It uses only the most common tags and genres for that user. The number of tags and genres used is set via "tag_ls_predictor.py :: UserProfile._decide_num_factors(...)". For example, if a user reviewed a lot of science fiction movies, then the science fiction genre will be chosen as one of the factors.

When this model encounters genres and tags not in use, then this model will reduce to the median model and perform poorly.

Tag Least Squares failure to converge

The model's coefficients are decided by least squares, and this procedure can at times fail to converge. The least squares routine being used is SciPy's LSQR, in the routine "tag_ls_predictor.py :: UserProfile :: _compute_x_factors(...)". When LSQR fail to converge, it will print:

> The exact solution is x = 0

This happened only twice out of over 200,000 users, so this issued was not investigated further. To improve, try different least square algorithms.

An alternative approach is to use fewer factors. For a dense matrix, "n" equations can support "n" unknowns. However, the situation here is a sparse matrix, so more than "n" equations are needed for stability. The "tag_ls_predictor.py :: UserProfile._decide_num_factors(...)" routine is already conservative when it comes to allowing new factors. Limiting the number of factors is good for the least squares stability, but bad for model performance.

Model Evaluation Results

Agreement Percentages

Approach	Agreement Percentage (range)
tag counting	0.65
ALS rank 3	0.63~0.66
ALS rank 5	0.63~0.65
ALS rank 7	0.63~0.645
ALS rank 9	0.63~0.65
ALS rank 11	0.63~0.64
tag least squares	0.625
Movie median only	0.445

Data

Using movie median only

Evaluation run time (compute time): 21 seconds on quad core computer

Rank agreement average values: 0.4455, 0.4451, 0.4454, 0.4463, 0.4456
Number of data points: around 231,276

Rank agreement average with < 0.05 and > 0.95 values removed: 0.4696
Number of data points: 198,030

Using the tag counting model

Evaluation run time (compute time): 7 minutes 20 seconds on quad core computer

Rank agreement average values: 0.6501, 0.6505, 0.6500, 0.6502, 0.6503
Number of data points: 231,279; 231,287; 231,267; 231,253; 231,275

Rank agreement average with < 0.05 and > 0.95 values removed: 0.6360, 0.6366, 0.6364, 0.6363, 0.6367
Number of data points: 200,621; 200,445; 200,340; 200,662; 200,637

Using the tag least square model

Evaluation run time (compute time): 12 minutes 26 seconds on quad core computer

Rank agreement average values: 0.62608, 0.62590, 0.62565, 0.62612, 0.62630
Number of data points: 231,278; 231,280; 231,281; 231,286; 231,274

Rank agreement average with < 0.05 and > 0.95 values removed: 0.62080, 0.62023, 0.62047, 0.62091, 0.62079
Number of data points: 202,737; 202,556; 202,803; 202,763; 202,360

ALS

Evaluation run time (compute time): 158 seconds on quad core computer

ALS model coverage

	Users	Movies
ALS 3	231274	33598
ALS 5	231271	27690
ALS 7	231264	24353
ALS 9	227440	22142
ALS 11	220995	20576

ALS training speed

AWS 36 vCPU instance (18 core with hyper-threading)
Command: sudo python3 train_als_models.py cpu_count=18 als_thread_count=18

	(typical) seconds
ALS 3	7
ALS 5	7
ALS 7	7
ALS 9	22
ALS 11	26

The number of seconds do fluctuate more and more as the number of factors increase. It's all about how "lucky" the starting condition of the conjugate gradient is.

As the number of factors increase, the number of qualified ratings decrease, leading to fewer equations to solve. However, the number of variables increased, leading to more work per equation.

Rank Agreement

Model	Test #1	Test #2	Test #3	Test #4	Test #5
ALS 3	0.63647	0.64050	0.63493	0.63595	0.64255
ALS 5	0.63732	0.64673	0.64468	0.63366	0.63323
ALS 7	0.63327	0.63419	0.63530	0.63428	0.64147
ALS 9	0.64105	0.63855	0.64492	0.63311	0.65857
ALS 11	0.63359	0.63385	0.64927	0.63936	0.63605

Number of values (number of users) in the test set

Model	Test #1	Test #2	Test #3	Test #4	Test #5
ALS 3	231,249	231,249	231,239	231,266	231,249
ALS 5	231,236	231,233	231,222	231,252	231,227
ALS 7	231,219	231,215	231,199	231,231	231,204
ALS 9	227,397	227,380	227,387	227,403	227,375
ALS 11	220,958	220,943	220,926	220,967	220,951

Similar movies database compilation compute time

AWS 36 vCPU instance (18 cores with hyper-threading), 72 GB memory

sudo python3 worker_server.py cpu_count=18 aws_instance

22 minutes, 18 seconds when cluster is using one worker node

App

Recommendation

Building ALS models using the full data set

```
E:\proj2018\movies_recommend\python\full_data>python train_als_models.py overwrite training_set_ratio=1
0:00:00 Reading ratings.csv
0:01:20 Saving .\data\in\user_ratings.bin
0:01:46 Reading ratings.csv
0:03:14 Saving .\data\in\movie_ratings.bin
    Saving .\data\in\\movie_medians_full.bin
0:03:34 Shrinking training data to satisfy ALS requirements.
0:03:48 Saving "als_user_ids" and "als_movie_ids" for ALS factor 3
0:03:57 Saving "user_ratings_train" and "user_ratings_test" for ALS factor 3
0:04:13 Saving "als_user_ids" and "als_movie_ids" for ALS factor 5
0:04:22 Saving "user_ratings_train" and "user_ratings_test" for ALS factor 5
0:04:37 Saving "als_user_ids" and "als_movie_ids" for ALS factor 7
0:04:46 Saving "user_ratings_train" and "user_ratings_test" for ALS factor 7
0:05:01 Saving "als_user_ids" and "als_movie_ids" for ALS factor 9
0:05:11 Saving "user_ratings_train" and "user_ratings_test" for ALS factor 9
0:05:25 Saving "als_user_ids" and "als_movie_ids" for ALS factor 11
0:05:34 Saving "user_ratings_train" and "user_ratings_test" for ALS factor 11
          Users      Movies
ALS 3     270115     37395
ALS 5     259346     30806
ALS 7     249496     27094
ALS 9     243579     24659
ALS 11    231995     22809
0:05:43 Building ALS factor 3 model
0:06:13 Saving "user_factors" and "item_factors" to disk
0:06:16 Building ALS factor 5 model
0:06:51 Saving "user_factors" and "item_factors" to disk
0:06:53 Building ALS factor 7 model
0:08:33 Saving "user_factors" and "item_factors" to disk
0:08:34 Building ALS factor 9 model
0:10:10 Saving "user_factors" and "item_factors" to disk
0:10:14 Building ALS factor 11 model
0:12:10 Saving "user_factors" and "item_factors" to disk
Total run time 0:12:11
```

Manually copy the following files to "app\recommend"

- in/movie_medians_full.bin
- ALS files, two per ALS model:
    - als/als3_item_factors.bin
    - als/als3_movie_ids.bin

App

Front End Files: python\app\web_page

index.html	Movies title search
recommendations.html	Movie recommendations
user_ratings.html	Displays user ratings
js/global.js	lambda_apis - location of API backends user_id fixed at -1
js/ModelParams.js	Renders model parameters using <table>
js/MovieGrid.js	Renders a movie poster and lets user rate it

Backend Files: python\app

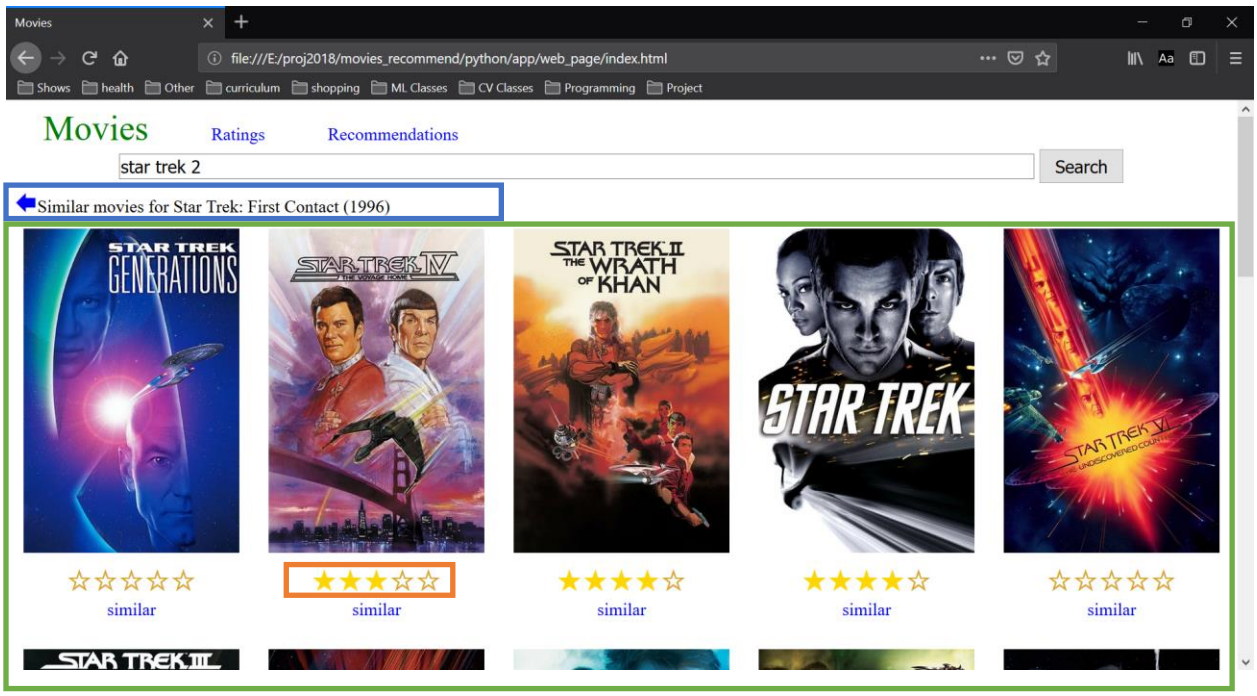
get_rated_movies/	Retrieves user's rated movies, sorted first by rating, then by title.
rate/	Rate a movie or delete an existing rating.
recommend/	Retrieve recommendations from database, generating them if they do not exist.
search/	Title search.
similar/	Retrieve similar movies.

Front End

MovieGrid.js

This file contains four GUI objects: MovieGrid, Stars, HistoryBar, and MovieTextGrid

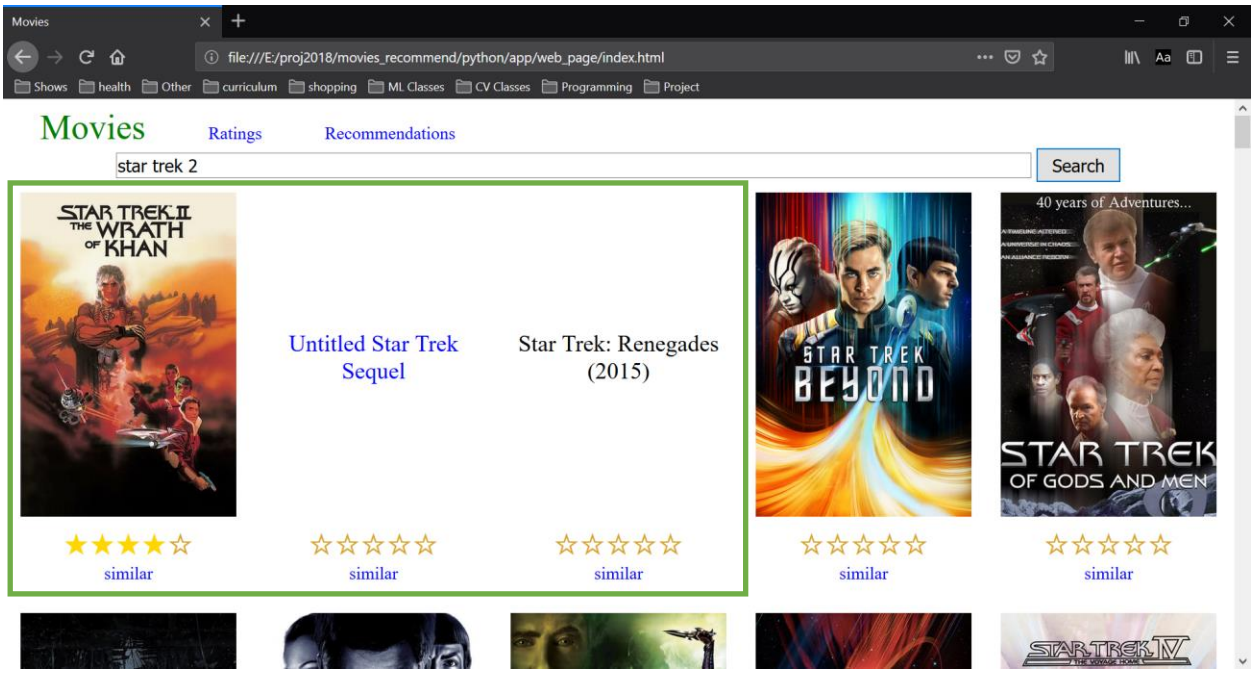
Screen shots of: **MovieGrid**, **Stars**, and **HistoryBar**



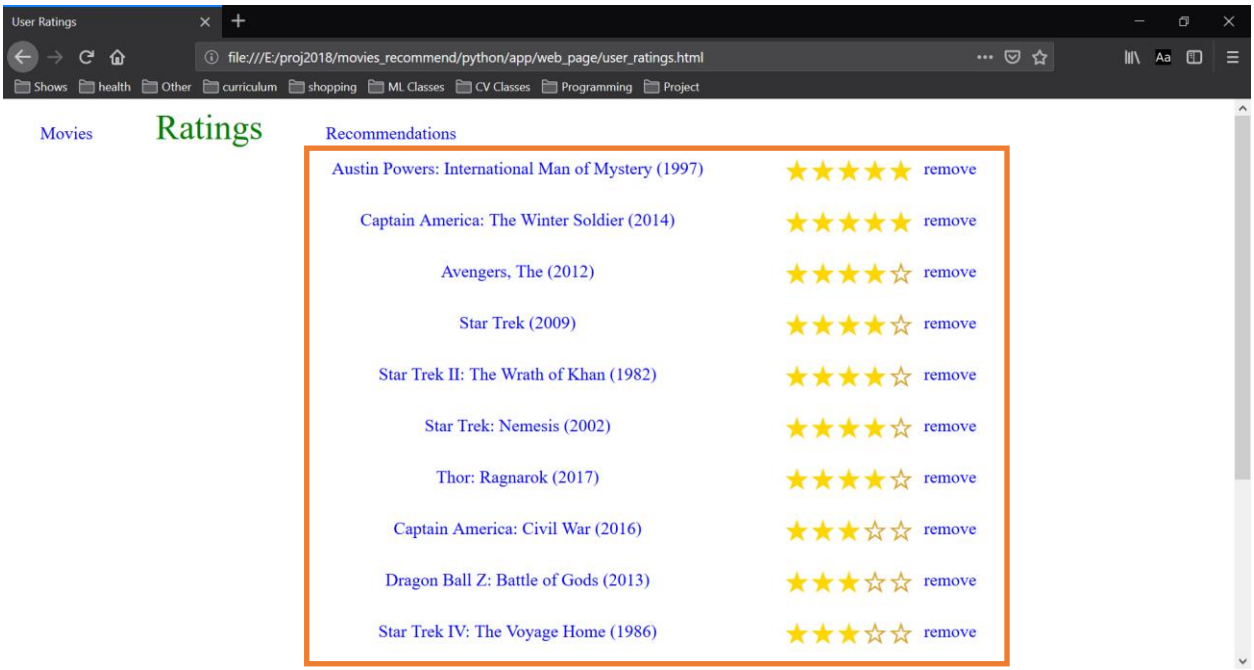
MovieGrid - renders the list of movies returned by the backend API. Each movie can have three different renderings:

- poster and link
- link only - because the TMDB API did not return a poster link
- title only, no link - because the TMDB API did not return a valid result for a movie's TMDB ID

Example of the three types of rendering:



Screenshot of **MovieTextGrid**:



MovieGrid vs MovieTextGrid

Both render the list of movie results returned by the backend APIs.

MovieGrid is used to render results from title search and movie recommendation, where the number of results is limited.

MovieTextGrid is used to render the result from the "get_rated_movies" API. Some reviewers have thousands of movies rated. There is currently no pagination support. If the rendering is limited to text, without the posters, the browser should be able to display it.

How star rating works

Each star is rendered as ★.

Each has its own event handler, created at at Stars :: add_star(i)

```
star_tag.addEventListener("mouseover", function () {
  this_obj.render_rating(i + 1);
});

star_tag.addEventListener("mouseleave", function () {
  this_obj.render_rating(this_obj.rating);
});

star_tag.addEventListener("click", function () {
  this_obj.rate(i + 1);
});
```

Hovering over a star can affect the appearance of other stars.

Clicking on a star triggers an API call to rate the movie.

This system can only be used for full stars, because it relies on the ★ character. Currently there is no half-star character.

HistoryBar's linked to MovieGrid

The history bar shows the current situation in the "MovieGrid" and can move back to a prior view.

The history bar is linked to "MovieGrid" during construction. Each object has a reference to the other.

"MovieGrid" constructor has "option", and one option is a reference to a "history_bar".

When there is a "history_bar" object, that "history_bar" object also gets a reference to the "MovieGrid" object.

```
class MovieGrid{

  /**
   * @param {Element} parent_tag - html <div> tag
   * @param {Object} options - keys: "similar_movies", "history_bar"
```

```
 */
constructor(parent_tag, options) {
  ...

  if (options.history_bar != null)
    options.history_bar.set_movie_grid(this);
}
```

Adding data to the history bar

This is done via: HistoryBar::add(...).

When the user search for a movie:

index.html → <script> → search() → call_api(...) → function(response) {
history_bar.add(...)
}

When the user clicks on a "similar" link:

MovieGrid → render(movies) → similar_movies_tag.addEventListener("click", ...) → function() → call_api(...) → function(response) {
history_bar.add(...)
}

Other Notes

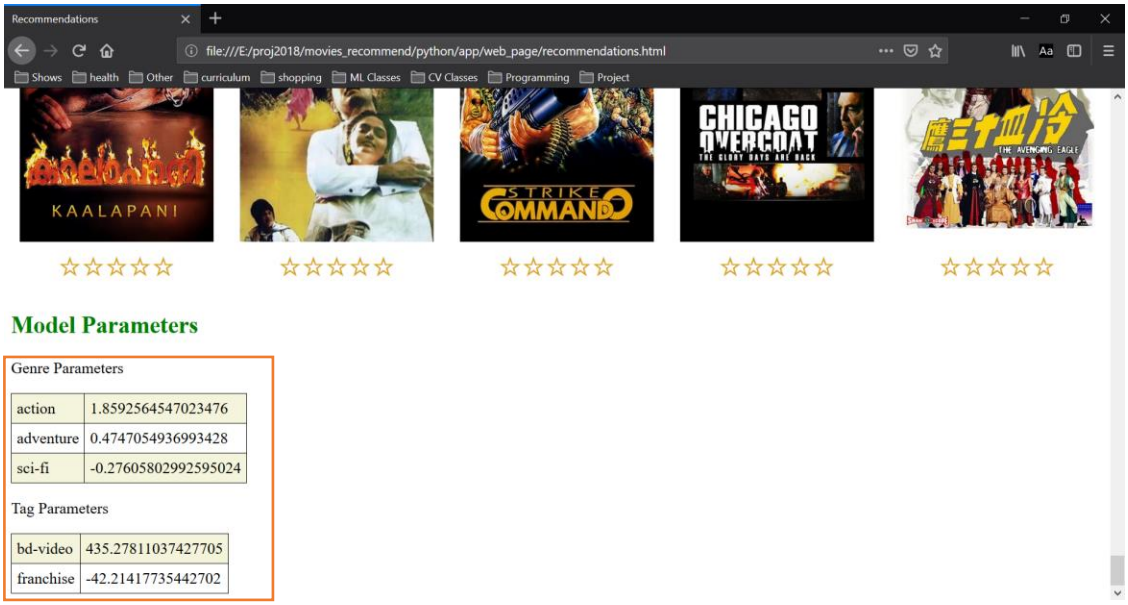
recommendations.html - stores user selection in session storage

```
sessionStorage.setItem("algorithm_choice", algorithm);
```

so returning to this page restores the previous algorithm selection

ModelParam.js - **ModelParams** screenshot

Scroll to the bottom of the recommendation page to see recommendation model's parameters.



Back End

API Usage

	index.html	recommendations.html	user_ratings.html
get_rated_movies			x
rate, "op" = "add_rating"	x	x	x
rate, "op" = "remove_rating"			x
recommend		x	
search	x		
similar	x		

API summary

All APIs use POST

get_rated_movies	Retrieves user's rated movies, sorted first by rating, then by title. **Request** object key: ["user_id"] **Response** is a list of movies, and each movie is an object with keys: ["title", "movie_id", "tmdb_id", "poster_url", "rating"].
rate	Rate a movie or delete an existing rating. If rating a movie, the **request** object keys are: ["user_id", "op" = "add_rating", "movie_id", "rating"] If deleting an existing rating, the **request** object keys are: ["user_id", "op" = "remove_rating", "movie_id"] **Response** is {"error": "None"}
recommend	Retrieve recommendations, generating them if they do not exist. **Request** object keys: ["algorithm", "user_id"] **Response** is { "error": "None", "movie_data": a list of movies as mentioned previously, "model_params": a list where each item is [parameter_name, parameter_value]}

search	Title search. **Request** object keys: ["title", "user_id"] **Response** is a list of movies as mentioned previously.
similar	Retrieve similar movies. **Request** object keys: ["movie_id", "user_id"] **Response** is a list of movies as mentioned previously.

Lambda memory usage (MB)

get_rated_movies	70
rate	32
recommend	234*
search	182
similar	104

*There are many global variables, which are loaded only as needed.

Database

DynamoDB table "users"

partition key user_id (Number)	sort key attrib (String)	obj
-1	"ratings"	{movie_id: rating} stored as binary
	"status"	{"ratings_mod_time": timestamp, "tag_count_mod_time": timestamp, "tag_count_rotation": integer} stored as binary
	"tag_count_rec"	[movie_ids] stored as binary
	"tag_count_params"	[[param0_name, param0_value], [param1_name, param1_value], ...]

"_rec" means "recommendations"

"_rotation" refers to which subset of the recommendations to serve. For example, the recommendation engine might generate 400 recommendations, served in four sets of 100 each. In this case, the "_rotation" value would vary from 0 to 3.

Each recommendation algorithm gets its own "mod_time", "rec", "rotation", and "params". So there's "als3_mod_time", "als3_rec", "al3_rotation", and "als3_params".

"mod_time" Timestamp

When movies are rated by the user, a timestamp is written to:

user_id.status.ratings_mod_time

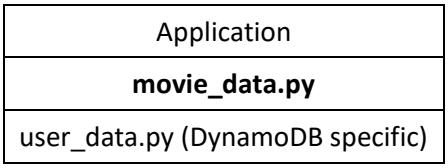
When the "tag_count" algorithm is used to generate recommendations, a timestamp is written to:

user_id.status.tag_count_mod_time

If the "tag_count_mod_time" is older than "ratings_mod_time", then a new set of recommendations need to be generated.

movie_data.py

This module provides routines that extract movie data.



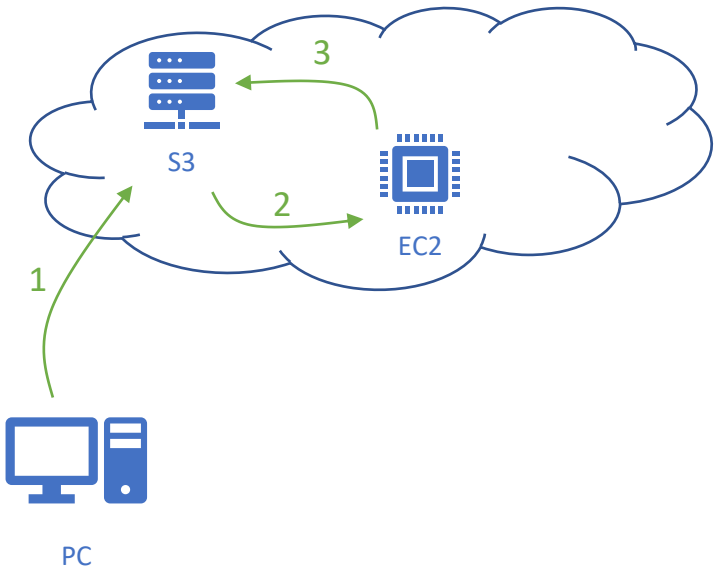
Call Tree

```
get_movie_data(movie_ids, user_id = None)
get_user_rated_movie_data(user_id)
   |
   |→ user_data.get_ratings(user_id)
   |     |→ _db_get(user_id, attrib)
   |
   |→ _get_movie_data(movie_ids : list, user_ratings : dict)
       loads data from "movie_titles.bin", "tmdb_data.bin"
```

Build Process

Files

- project_files.py - describes the contents of the Lambda APIs
- build.py - PC side build script
- ec2_build.py - AWS EC2 side build script



project_files.py

```
upstream_files = {
    "genre_counts.bin": "../full_data/data/in/genre_counts.bin",
```

The "../full_data/data/in/genre_counts.bin" is the upstream source of "genre_counts.bin". When the upstream version is newer, copy that to the Lambda project. However, if the Lambda project version is newer, then nothing is done.

```
shared_files = {"movie_data.py",
```

The "movie_data.py" is a shared file. All Lambda projects will get the latest copy of the "movie_data.py".

build.py

For the upstream files, make sure the Lambda projects have a version that is at least as new as the upstream version.

For the shared files, find the latest version, and make sure each Lambda project has the latest version.

Upload files to S3. The file modification time is included in the S3 meta data, so only recently modified files are uploaded.

ec2_build.py

Download project files from S3. Check the timestamp from the S3 meta data and only download files that are new.

Zip all files into a single ".zip" file.

Upload to S3.

Manual Steps

Not everything has been automated.

1. **Copy the ALS model files from full_data to "app/recommend/"**

   The "full_data" project normally train using 80% of the data and use the remaining 20% for testing. For application though, train using 100% of the data. Then manually copy the ALS model files to "app/recommend/".

   For each ALS algorithm, the "item factor" and "movie ids" information is needed. For example, the "ALS3" would need "als3_item_factors.bin" and "als3_movie_ids.bin".

   To automate this step, the "full_data" project needs to be coded such that training using 100% of the data automatically produces files with special names, such as "als3_item_factors_full.bin". Then have "project_files.py" request the "_full" version.

   This is the approach taken for "movie_medians_full.bin" - because it's vital that the median used for model evaluation comes from only the training data, not the full data.

2. **Build "numpy" and "scipy" on AWS EC2 for the "recommend" API**

   The "recommend" API uses "numpy" and "scipy" modules. These modules use native code and so must be built on Linux, ideally the same Linux that AWS Lambda uses.

   The Linux commands are:

   ```
   sudo python3 -m pip install numpy --target .
   sudo python3 -m pip install scipy --target .
   ```

App Local

Port number hard coded at "server.py" to 8000

API URL Changes

web_page/js/global.js - the API URLs now look like: "http://**localhost:8000**/get-rated-movies",

Back End Changes

- Everything in one directory, no more repetition of source files.
- server.py - redirects POST request to Lambda
- user_data.bin - instead of DynamoDB
- user_data.py - rewrite "db_write" and "db_get" functions to pickle "user_data.bin"

Known Issues

Python timestamp inconsistence across platforms

To see the issue, run the same "utcnow().timestamp()" comand on both AWS EC2 instance and on local desktop

On AWS EC2 instance (Linux):

```
>>> print(datetime.datetime.utcnow())
2019-01-07 04:10:13.741735
>>> print(datetime.datetime.utcnow().timestamp())
1546834216.749268
```

On my desktop (Windows, Python 3.5):

```
print(datetime.datetime.utcnow())
2019-01-07 04:10:23.900812
print(datetime.datetime.utcnow().timestamp())
1546863028.639443
```

The "utcnow()" is the same on both systems, but not the ".timestamp()". The Windows time is massively later.

This is an issue if testing the site by running some functions on Windows, and others on Linux. Keep in mind that when running the lambda functions on Windows, the timestamps that get written to the database will be Windows timestamps.

To remove existing time stamps, delete the "status" attribute of the "user" table.