Recommendation Systems via Matrix Factorization

Background

Dataset of ratings of 1682 movies by a set of 943 possible users.

Ratings were collected on a 5-star scale. Each rating is one of 5 possible integer values -- 1, 2, 3, 4, or 5 -- with 5 being 'best' and 1 being 'worst'.

Build a *recommendation system* to help guess which movies a user will like. The prediction system should be able to take as inputs specific user (denoted by index i one of 943 possibilities) and movie (denoted by index j one of 1682 possibilities). The output of our predictor (e.g. the quantity produced by calling model.predict(i,j)) will be a scalar rating $\hat{y}_{ij} \in \mathbb{R}$. For each possible user-movie pair i,j, we'd like $\hat{y}_{ij}$ to be as close as possible to the "real" rating y_{ij}.

Of course, not all users have seen and rated all movies, so some true ratings are simply unknown. Thus, some entries y_{ij} of the true rating matrix y are simply missing and it's impossible to know if your guesses for these entries are any good.

To sidestep the missing data problem, you can evaluate by simply *ignoring* missing entries and concentrating on the **observed** entries of y. You can create a list I of all distinct *observed* pairs of (user_id, item_id) in the complet dataset. You can then randomly divide these examples into training and test sets: I^{train} and I^{test}.

Two CSV files, ratings_train.csv and ratings_test.csv, to give you everything you need. Each row of these files specifies a (user_id, item_id) pair and its observed 5-star rating. The first few rows of ratings_train.csv are:

```
user_id, item_id, rating

772,36,3
471,228,5
641,401,4
312,98,4
...
```

Background: Evaluation

You'll use the training set of user-movie ratings to fit a model, and then evaluate on the test set. To measure prediction accuracy, use *mean absolute error*, which on the test set is evaluated as:

$$\text{MAE}(y, \hat{y}) = \frac{1}{|\mathcal{I}^{\text{test}}|} \sum_{i,j \in \mathcal{I}^{\text{test}}} |y_{ij} - \hat{y}_{ij}|$$

Background: SGD, Automatic Differentiation and autograd

Across # 1 and # 2, develop your own Python code and build a series of increasingly more powerful models to perform recommendation. For each model, view training as an optimization problem, and solve it with *stochastic gradient descent*. This will be somewhat similar to the gradient descent, but only visiting or using at each step "noisy" gradient estimates computed by visiting a small random subset of data (entries in the ratings matrix) at each update step.

How to compute gradients? Use the starter code to start with. To save lots of effort for many possible models, use autograd （https://github.com/HIPS/autograd） python package to perform *automatic differentiation*.

For each model of interest (defined mathematically below), create a separate class file for that model. For examples in starter code, see CollabFilterMeanOnly.py for # 1 and CollabFilterOneScalarPerItem.py for # 2. Define each model's methods for performing prediction and computing the loss used for gradient-based training.

All these models will be subclasses of a "base" class AbstractBaseCollabFilterSGD, which contains logic to construct and fit collaborative filtering models to data via stochastic gradient descent. Using this base class, you don't need to write SGD yourself, or even how to compute the gradient!

Below, review the methods/attributes you'll need to write, and the methods/attributes you only need to use that are provided in the base class.

Methods need to write

To complete the implementation of a given model, follow the following pattern:

- Use the instance attribute `param_dict` to store all model parameters

```
param_dict : dict
    Keys are string names of parameters
        Values are *numpy arrays* of parameter values
```

- Define method `predict` to make predictions:
    - Input: Specific user-movie example pairs, indicated by integer ids
    - Output: Predicted ratings for each example

- Define method `calc_loss_wrt_parameter_dict` to compute the loss to minimize with SGD:
    - Input: a minibatch of training data, a parameter dict
    - Output: scalar float, indicating the loss on the batch given the parameters

- Define method `init_parameter_dict` to initialize the param_dict attribute to random values:
    - Input: Number of possible users, Number of possible items
    - Output: None, internal attribute `param_dict` updated

The steps above are *all* need to do for each possible model. Each class *inherits* a complete `fit` method from the predefined `AbstractBaseCollabFilterSGD`, which knows how to perform SGD given the pieces above.

This may be helpful: Introduction_Autograd.ipynb

Methods you'll need to use (but not edit)

You should then skim through the provided **AbstractBaseCollabFilterSGD.py**. to be sure you understand what's going on.

- __init__ : Constructor

This is where you define the batch_size, the step_size (learning rate), and the number of epochs n_epochs to complete during training.

This constructor can also define the regularization strength alpha and the number of hidden factors n_factors for #3.

- fit : Method to fit model parameters to provided data

At the bottom of each starter code file, you can see example code for calling fit for that model.

Attributes available after calling fit

After fitting a model, you'll have these attributes available to you. They store useful diagnostic metrics computed at various checkpoints throughout the training procedure.

Epoch - A Definition: An epoch is a unit of training progress. One epoch is complete when our stochastic gradient descent has processed multiple minibatches whose total number of examples are "equivalent" to the size of the entire training dataset.

By default, compute "report" metrics on the initial parameters (before any updates), and then every 10% of an epoch for the first 5 or so epochs, then every 1 epoch or so after that. This helps you monitor progress, which is especially rapid in early iterations.

trace_epoch : 1D array-like

Contains the epochs (fractional) where model performance was assessed.

Value of 0.0 indicates the initial model parameters (before any gradient updates).

Value of 0.1 indicates that the total training examples seen represents 10% of the size of the training set.

trace_loss : 1D array-like

Contains training loss (at current batch only) whenever model was assessed.

This is reported as an average per example in the current batch.

```
trace_mae_train : 1D array-like
    MAE assessed on entire training set whenever model assessed.

trace_mae_valid : 1D array-like
    MAE assessed on entire validation set whenever model assessed.
```

So for example, to plot training MAE vs. epochs completed, you could do:

```
# After calling fit...
plt.plot( model.trace_epoch, model.trace_mae_train, 'b.-')
```

Datasets

use the MovieLens 100K dataset. This data set consists of:

- 100,000 ratings (1-5 stars) from 943 users on 1682 movies.
- Each user has rated at least 20 movies.
- Some movies have ratings from only a few users.
- Simple demographic info for the users (age, gender, etc.) are available

For more information about the original dataset,
see http://files.grouplens.org/datasets/movielens/ml-100k-README.txt.

Use the clean train/test split of this dataset: data_movie_lens_100k

Starter Code and Restrictions

Limited to the following Python packages for related functionality:

- All Problems: numpy, scipy, sklearn, etc. (not really fancy ones)
- All Problems: autograd package for automatic differentiation
- # 4: surprise package for recommendation
- # 5 (open-ended): any other packages you want

You can INSTALL the autograd and surprise packages as follows:

```
source activate your_env
pip install autograd
conda install -c conda-forge scikit-surprise
```

1: Simple Baseline Model with SGD and Autograd

Develop models using autograd framework, consider the simplest possible baseline model "M1": a model that makes the *same* scalar prediction for a movie's rating no matter what user or movie is considered. This model has one scalar parameter $\mu \in \mathbb{R}$, and the prediction for user i and movie j is simply:

$$\hat{y}_{ij} = \mu$$

Training model M1:

To train model M1 for N users and M movies, optimize the following objective:

$$\min_{\mu \in \mathbb{R}} \sum_{i,j \in \mathcal{I}^{\text{train}}} (y_{ij} - \mu)^2$$

Minimize the squared error on the training set, between the predicted rating (simply μ here) and the observed rating y_{ij}.

Code Implementation

Edit the starter code file: CollabFilterMeanOnly.py. Complete each required method (init_parameter_dict, predict, and calc_loss_wrt_parameter_dict), as described above in the Autograd background section.

.

Also write in details:

1a: Figure and caption: Trace plot showing *mean absolute error* vs. epoch completed, for both the training set and validation set.

1b: Write: Would adding regularization to training optimization problem (e.g. an L2 penalty on μ) noticeably improve performance with this model on this dataset? Why or why not? What's special about this task that you might want to consider?

1c: Write: there is a closed-form operation you could apply to the training set to compute the optimal μ value. How would you compute this "exact" solution? Report the computed optimal μ value. Does this result agree with your SGD solution?

2: One-Scalar-Per-Item Baseline with SGD and Autograd

Consider a model "M2" with three parameters:

- μ : scalar mean rating
- b_i : scalar bias term for each user i
- c_j : scalar bias term for each movie j

Prediction under model "M2" becomes:

$$\hat{y}_{ij} = \mu + b_i + c_j$$

Training Model M2:

To train model M2 for N users and M movies, optimize the following objective:

$$\min_{\mu \in \mathbb{R}, b \in \mathbb{R}^N, c \in \mathbb{R}^M} \sum_{i,j \in \mathcal{I}^{\text{train}}} (y_{ij} - \mu - b_i - c_j)^2$$

This is to minimize the squared error on the training set, between the predicted rating and the observed rating y_{ij}.

 # 2: Code Implementation

Edit the starter code file: CollabFilterOneScalarPerItem.py. Complete each required method, as described above in the Autograd background section.

 # 2: Write in details:

2a: Figure and caption: Trace plot showing *mean absolute error* vs. epoch completed, for both the training set and validation set.

2b: Write: Taking the best result of model M2 above, how does model M2 compare to M1 in terms of predictive performance?

2c: Figure and caption: Inspect the learned per-movie rating adjustment parameters c_j for the short list of movies in select_movies.csv. Notice any interpretable trends? What does it mean for a movie to have a large positive c_j or large negative c_j value?

3: Matrix Factorization with Autograd

Consider a full matrix factorization model "M3" with five parameters:

- μ : scalar mean rating
- b_i : scalar bias term for user i
- c_j : scalar bias term for movie j
- u_i : K-dimensional vector for user i
- v_j : K-dimensional vector for movie j

Think about how to set K, the number of "factors" or "dimensions" to learn when representing each user/movie in a vector space. Within the starter code, you set this with the n_factors keyword argument to the constructor.

Prediction under model "M3" is:

$$\hat{y}_{ij} = \mu + b_i + c_j + u_i^T v_j$$

Training:

To train model M3 for N users and M movies, optimize the following objective:

$$\min_{\mu,b,c,\{u_i\}_{i=1}^N,\{v_j\}_{j=1}^M} \alpha\left(\sum_j \sum_k v_{jk}^2 + \sum_i \sum_k u_{ik}^2\right) + \sum_{i,j \in \mathcal{I}^{\text{train}}} (y_{ij} - \mu - b_i - c_j - u_i^T v_j)^2$$

Note that added L2 penalties on the uu and vv vectors. In the starter code, the penalty strength hyperparameter can be set via the keyword argument alpha=... **in the constructor, and accessed via the attribute** alpha**.**

 # 3: Code Implementation

Edit the starter code file: CollabFilterOneVectorPerItem.py. Complete each required method, as described above in the Autograd background section.

 # 3: Write in details:

3a: Figure and caption: Using NO regularization (alpha=0.0), make a trace plot showing *mean absolute error* vs. epoch completed, for both the training set and validation set. Repeat for several values of the number of factors K: 2, 10, 50.

3b: Figure and caption: Repeat 3a with a moderate regularization strength $\alpha>0$, such that any overfitting in 3a is no longer visible.

3c: Write: Other than L2/L1 regularization penalties on parameter values, what else could you do to avoid overfitting in this setting when using the same model M3 and still using SGD?

3d: Write: Taking the best result of model M3 so far, how does model M3 compare to M2 or M1 in terms of predictive performance? How many factors do you recommend? Should you try even more than 50 factors?

3e: Figure and caption: For the best M3 model with $K=2$ factors, consider the learned per-movie vectors v_j for the short list of movies listed in select_movies.csv. Can you make a scatter plot of the 2-dimensional "embedding" vector v_j of these movies (labeling each point with its movie title)? Do you notice any interpretable trends?

4: Matrix Factorization using Surprise

Using surprise to tune collaborative filtering models

The surprise package can fit a collaborative filtering model with K hidden factors to ratings data using surprise's SVD.
(https://surprise.readthedocs.io/en/stable/matrix_factorization.html
)

4a: Figure and caption: Show the results of 5-fold cross-validation on the full-training set (no need for a separate validation set here), across a range of possible K and α values to get the best possible *mean absolute error* performance.

What is your recommendation for the choice of the hyperparameters K and α? How did you set the learning rate?
4b: Write: How does this model's performance compare with that from #3, which you trained using your own implementation? If big differences exist, why do you think they occurred?

Inspecting Learned Vectors for Gender Information

Let U be the matrix of user vectors, with its i-th row as the vector u_i for user i. Each column of U can be seen as a "learned" feature vector for a particular user. The question here is, do your learned features that are optimized for predicting movie ratings contain anything to do with gender?

The provided data file user_info.csv contains an is_male column indicating which users in the dataset are male. Can you predict this signal given the features U?

Feel free to use any sklearn classifier.

4c: Figure and caption: Show the results of your gender-from-user-features classifier.

#5 Open-Ended Recommendation Challenge

The starter code includes an additional test dataset -- ratings_test_masked.csv -- which you haven't used yet. This contains an additional 10,000 entries of user-movie pairs for the same set of users and movies as above. It omitted the ratings here, so you won't be able to access them. In this problem, your goal is to obtain the best possible prediction results on this heldout test data, in terms of mean absolute error.

You can try *any model* you want. You can make use of the other ratings you've already observed in the training set, as well as the user-specific info found in user_info.csv and the movie-specific attributes found in item_info.csv. You can use autograd, sklearn, surprise, or any other package. Your goal is to get the best score.

Example ideas:

- Throughout this project, you have used mean-squared error in the calc_loss... method, but then evaluated with mean-absolute-error. What if you just used mean absolute error in the loss? This should be easy with autograd.
- Can you somehow use user-specific features (like gender and age) as well as the learned features to improve accuracy?

- Throughout this project, you used mean-squared error in the calc_loss... method for training the models, but then evaluated with mean-absolute-error. What if you just used mean absolute error in the loss? This should be easy with autograd.
- Can you somehow use user-specific features (like gender and age) as well as the learned features to improve accuracy?

Try one of the k-nearest neighbor approaches to recommendation (https://surprise.readthedocs.io/en/stable/knn_inspired.html) in surprise

- Try dropout as an alternative to L2 regularization

 # 5: as before

As before the predictions as the text file named predicted_ratings_test.txt

This file should have one line for each non-header line of ratings_test_masked.csv.

Problem 5: Write in detail:

Include writing parts describing method, and 1 or 2 tables/figures relevant to reporting how you trained the model and how it performed on the test set. Compare performance of the txt file you send to me(in terms of mean absolute error) and discuss how this number compared to your internal validation efforts.

#6

Predicting Gender from Learned Per-User Embedding Vectors

Consider the best M3 model you've trained (either with surprise or with your own implementation in Problem 3). Let U be the learned matrix of user vectors, with its i-th row as the vector u_i for user i. Each row of U can be seen as a "feature vector" for a particular user.

Do your learned per-user features that are optimized for predicting movie ratings contain anything to do with gender?

The provided data file user_info.csv contains an is_male column indicating which users in the dataset are male. Can you predict this signal given the features U?

Implementation Tasks

Train a binary classifier to predict the is_male target variable given the fixed per-user embedding features U. Feel free to use any sklearn binary classifier (logistic regression, SVM, random forest, etc.). You should use best practices for model selection (cross validation, hyperparameter search, etc).

You'll need to set up this classification task from scratch. You have info for all 943 users. Should you include them all in the training process? How will you fairly report the accuracy on heldout data?

WRITE:

a: Describe your analysis. How did you split the data (train/test)? What classifier did you choose and why? How did you tune hyperparameters?

b: Figure and caption: Show a confusion matrix for your gender-from-user-features classifier. What error rate do you get? Is it significantly better than chance for this dataset?