Explicit Matrix Factorization: ALS, SGD, and All That Jazz

Insight Follow
Mar 16, 2016 · 14 min read

Ethan Rosenthal is a Data Scientist at Birchbox. He was an Insight Data Science Fellow in NYC in 2015 and holds a Ph.D. in Physics from Columbia University. He blogs here about all things data science, including recommender systems. In the post below he digs into the details of how to inject a bit more machine learning into these models.

Working at an e-commerce company, I spend a lot of time thinking about recommender systems. The goal of a recommendation model is to present a ranked list of objects given an input object. Typically, this ranking is based on the similarity between the input object and the listed objects. To be less vague, one often wants to either present similar products to a given product or present products that are personally recommended for a given user.

The astounding thing is that if one has enough user-to-product "interaction" data (ratings, purchases, clicks, etc…), then no other information is necessary to make decent recommendations. This is quite different than regression and classification problems where one must explore various features in order to boost a model's predictive powers.

Often, one's first introduction to recommender systems is collaborative filtering; specifically, one learns user- and item-based collaborative filtering. These are relatively old methods, and, through the lens of modern machine learning, these methods might feel a bit off. To me, machine learning almost always deals with some function which we are trying to maximize or minimize. In simple linear regression, we minimize the mean squared distance between our predictions and the true values. Logistic regression involves maximizing a likelihood function. However, in user- and item-based collaborative filtering, one randomly tries a bunch of different parameters and watches what happens to the error metric. This sure doesn't feel like machine learning. To see this in action, check out my blog post on these methods.

> *To bring some technical rigor to recommender systems, I would like to talk about matrix factorization where we do, in fact, learn parameters to a model in order to directly minimize a function.*

This blog post will be organized as follows: We'll start by grabbing a well known dataset of movie ratings on which to train our models. Then, I'll introduce and derive two different learning algorithms for matrix factorization. Next, we'll spend some time optimizing model parameters.

Lastly, we'll use a free API to visualize our recommendations for easy evaluation of our results.

- GRABBING AND PARSING THE MOVIELENS DATASET
- INTRODUCING MATRIX FACTORIZATION FOR RECOMMENDER SYSTEMS
- ALTERNATING LEAST SQUARES FOR TRAINING THE MODEL
- ALS DERIVATION
- ALS COMPUTATION
- OPTIMIZING ALS MODEL PARAMETERS
- STOCHASTIC GRADIENT DESCENT AS AN ALTERNATIVE TRAINING ALGORITHM
- SGD DERIVATION
- SGD COMPUTATION
- OPTIMIZING SGD MODEL PARAMETERS
- USING THEMOVIEDB.ORG'S API TO EYE-TEST RECOMMENDATIONS

Grabbing and parsing the MovieLens dataset

For this introduction, I'll use the MovieLens dataset — a classic dataset for training recommendation models. It can be obtained from the GroupLens website. There are various datasets, but the one that I will use below consists of 100,000 movie ratings by users (on a 1–5 scale). Note that this is considered an explicit feedback dataset. The users are explicitly telling us how they would rate a movie. This is different than in implicit feedback dataset like clicks or purchases where we hope that the data is a proxy for the user's preference. Different algorithms must be used for such datasets.

The main data file consists of a tab-separated list with user-id (starting at 1), item-id (starting at 1), rating, and timestamp as the four fields. We can use bash commands in the Jupyter notebook to download the file and then read it into with pandas. We'll then build a ratings matrix with users as rows, items as columns, and ratings as the elements of the matrix.

In this dataset, every user has rated at least 20 movies which results in a reasonable sparsity of 6.3%. This means that 6.3% of the user-item ratings have a value. Note that, although we filled in missing ratings as 0, we should not assume these values to truly be zero. More appropriately, they are just empty entries. We will split our data into training and test sets by removing 10 ratings per user from the training set and placing them in the test set.

```
In [1]: import numpy as np
        import pandas as pd
        np.random.seed(0)

In [2]: !curl -O http://files.grouplens.org/datasets/movielens/ml-10
        0k.zip
        !unzip ml-100k.zip
        !cd ml-100k/

In [3]: names = ['user_id', 'item_id', 'rating', 'timestamp']
        df = pd.read_csv('u.data', sep='\t', names=names)
        df.head()

Out[3]:
```

	user_id	item_id	rating	timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596

```
In [4]: n_users = df.user_id.unique().shape[0]
```

insight-nb1.ipynb hosted with ❤ by GitHub view raw

Introducing matrix factorization for recommender systems

With our training and test ratings matrices in hand, we can now move towards training a recommendation system. Explanations of matrix

factorization often start with talks of "low-rank matrices" and "singular value decomposition". While these are important for a fundamental understanding of this topic, I don't find math-speak to be too helpful in understanding the basic concepts of various algorithms. Let me simply state the assumptions that basic matrix factorization makes.

Matrix factorization *assumes* that:

- Each user can be described by *k attributes* or *features*. For example, feature 1 might be a number that says how much each user likes sci-fi movies.
- Each item (movie) can be described by an analagous set of *k* attributes or features. To correspond to the above example, feature 1 for the movie might be a number that says how close the movie is to pure sci-fi.
- If we multiply each feature of the user by the corresponding feature of the movie and add everything together, this will be a good approximation for the rating the user would give that movie.

That's it. The beauty is that we do not know what these features are. Nor do we know how many *(k)* features are relevant. We simply pick a number for

k and learn the relevant values for all the features for all the users and items. How do we learn? By minimizing a loss function, of course!

We can turn our matrix factorization approximation of a k-attribute user into math by letting a user u take the form of a k-dimensional vector x_u. Similarly, an item i can be k-dimensional vector y_i. User u's predicted rating for item i is just the dot product of their two vectors.

$$\hat{r}_{ui} = \mathbf{x}_u^\top \cdot \mathbf{y}_i = \sum_k x_{uk} y_{ki}$$

where r_ui hat represents our prediction for the true rating r_ui, and y_i (x ⊺$_u$) is assumed to be a column (row) vector. These user and item vectors are often called latent vectors or low-dimensional embeddings in the literature. The k attributes are often called the latent factors. We will choose to minimize the square of the difference between all ratings in our dataset (S) and our predictions. This produces a loss function of the form

$$L = \sum_{u,i \in S} (r_{ui} - \mathbf{x}_u^\top \cdot \mathbf{y}_i)^2 + \lambda_x \sum_u \|\mathbf{x}_u\|^2 + \lambda_y \sum_u \|\mathbf{y}_i\|^2$$

Note that we've added on two *L2* regularization terms at the end to prevent overfitting of the user and item vectors. Our goal now is to minimize this loss function. Derivatives are an obvious tool for minimizing functions, so I'll cover the two most popular derivative-based methods. We'll start with Alternating Least Squares (ALS).

Alternating Least Squares for training the model

For ALS minimiztion, we hold one set of latent vectors constant. For this example, we'll pick the item vectors. We then take the derivative of the loss function with respect to the other set of vectors (the user vectors). We set the derivative equal to zero (we're searching for a minimum) and solve for the non-constant vectors (the user vectors). Now comes the alternating part: With these new, solved-for user vectors in hand, we hold *them* constant, instead, and take the derivative of the loss function with respect to the previously constant vectors (the item vectors). We alternate back and forth and carry out this two-step dance until convergence.

ALS Derivation

To explain things with math, let's hold the item vectors (y_i) constant and take the derivative of the loss function with respect to the user vectors (x_u)

$$\frac{\partial L}{\partial \mathbf{x}_u} = -2\sum_i (r_{ui} - \mathbf{x}_u^\intercal \cdot \mathbf{y}_i)\mathbf{y}_i^\intercal + 2\lambda_x \mathbf{x}_u^\intercal$$

$$0 = -(\mathbf{r}_u - \mathbf{x}_u^\intercal Y^\intercal)Y + \lambda_x \mathbf{x}_u^\intercal$$

$$\mathbf{x}_u^\intercal (Y^\intercal Y + \lambda_x I) = \mathbf{r}_u Y$$

$$\mathbf{x}_u^\intercal = \mathbf{r}_u Y (Y^\intercal Y + \lambda_x I)^{-1}$$

A couple things happen above: let us assume that we have n users and m items, so our ratings matrix is $n \times m$. We introduce the symbol Y (with dimensioins $m \times k$) to represent all item row vectors vertically stacked on each other. Also, the row vector r_u just represents users u's row from the ratings matrix with all the ratings for all the items (so it has dimension $1 \times m$). Lastly, I is just the identity matrix which has dimension $k \times k$ here.

Just to make sure that everything works, let's check our dimensions. I like doing this with Einstein notation. Although this seems like an esoteric physics thing, there was a reason it was invented — it makes life really simple! The basic tenant is that if one observes a variable for a matrix index more than once, then it is implicitly assumed that you should sum over that index. Including the indices in the last statement from the derivation above, this appears as the following for a single user's dimension k

$$x_{uk} = r_{ui}Y_{ik}(Y_{ki}Y_{ik} + \lambda_x I_{kk})^{-1}$$

When you carry out all the summations over all the indices on the right hand side of the above statement, all that's left are *u*'s as the rows and *k*'s as the columns. Good to go!

The derivation for the item vectors is quite similar

$$\frac{\partial L}{\partial \mathbf{y}_i} = -2\sum_i (r_{iu} - \mathbf{y}_i^\top \cdot \mathbf{x}_u)\mathbf{x}_u^\top + 2\lambda_y \mathbf{y}_i^\top$$
$$0 = -(\mathbf{r}_i - \mathbf{y}_i^\top X^\top)X + \lambda_y \mathbf{y}_i^\top$$
$$\mathbf{y}_i^\top (X^\top X + \lambda_y I) = \mathbf{r}_i X$$
$$\mathbf{y}_i^\top = \mathbf{r}_i X(X^\top X + \lambda_y I)^{-1}$$

Now that we have our equations, let's program this thing up!

Computation: turning the math into code

With significant inspiration from Chris Johnson's implicit-mf repo, I've written a class that trains a matrix factorization model using ALS. In an attempt to limit this already long blog post, the code is relegated to this GitHub gist — feel free to check it out and play with it yourself. The class also has the functionality to calculate a learning curve where we plot our optimization metric against the number of iterations of our training algorithm.

```
from numpy.linalg import solve

class ExplicitMF():
    def __init__(self,
                 ratings,
                 n_factors=40,
                 item_reg=0.0,
                 user_reg=0.0,
                 verbose=False):
        """
        Train a matrix factorization model to predict empty
        entries in a matrix. The terminology assumes a
        ratings matrix which is ~ user x item

        Params
        ======
        ratings : (ndarray)
            User x Item matrix with corresponding ratings

        n_factors : (int)
            Number of latent factors to use in matrix
```

```
21                  Number of latent factors to use in matrix
22                  factorization model
23
24              item_reg : (float)
25                  Regularization term for item latent factors
26
27              user_reg : (float)
28                  Regularization term for user latent factors
29
30              verbose : (bool)
31                  Whether or not to printout training progress
32              """
33
34              self.ratings = ratings
35              self.n_users, self.n_items = ratings.shape
36              self.n_factors = n_factors
37              self.item_reg = item_reg
38              self.user_reg = user_reg
39              self._v = verbose
40
41          def als_step(self,
42                       latent_vectors,
43                       fixed_vecs,
44                       ratings,
45                       _lambda,
46                       type='user'):
47              """
48              One of the two ALS steps. Solve for the latent vectors
49              specified by type.
50              """
51              if type == 'user':
52                  # Precompute
53                  YTY = fixed_vecs.T.dot(fixed_vecs)
```

```
54             lambdaI = np.eye(YTY.shape[0]) * _lambda
55 
56             for u in xrange(latent_vectors.shape[0]):
57                 latent_vectors[u, :] = solve((YTY + lambdaI),
58                                              ratings[u, :].dot(fixed_vecs))
59         elif type == 'item':
60             # Precompute
61             XTX = fixed_vecs.T.dot(fixed_vecs)
62             lambdaI = np.eye(XTX.shape[0]) * _lambda
63 
64             for i in xrange(latent_vectors.shape[0]):
65                 latent_vectors[i, :] = solve((XTX + lambdaI),
66                                              ratings[:, i].T.dot(fixed_vecs))
67         return latent_vectors
68 
69     def train(self, n_iter=10):
70         """ Train model for n_iter iterations from scratch."""
71         # initialize latent vectors
72         self.user_vecs = np.random.random((self.n_users, self.n_factors))
73         self.item_vecs = np.random.random((self.n_items, self.n_factors))
74 
75         self.partial_train(n_iter)
76 
77     def partial_train(self, n_iter):
78         """
79         Train model for n_iter iterations. Can be
80         called multiple times for further training.
81         """
82         ctr = 1
83         while ctr <= n_iter:
84             if ctr % 10 == 0 and self._v:
85                 print '\tcurrent iteration: {}'.format(ctr)
86             self.user_vecs = self.als_step(self.user_vecs,
```

```
87                                             self.item_vecs,
88                                             self.ratings,
89                                             self.user_reg,
90                                             type='user')
91             self.item_vecs = self.als_step(self.item_vecs,
92                                             self.user_vecs,
93                                             self.ratings,
94                                             self.item_reg,
95                                             type='item')
96             ctr += 1
97 
98     def predict_all(self):
99         """ Predict ratings for every user and item. """
100         predictions = np.zeros((self.user_vecs.shape[0],
101                                 self.item_vecs.shape[0]))
102         for u in xrange(self.user_vecs.shape[0]):
103             for i in xrange(self.item_vecs.shape[0]):
104                 predictions[u, i] = self.predict(u, i)
105 
106         return predictions
107     def predict(self, u, i):
108         """ Single user and item prediction. """
109         return self.user_vecs[u, :].dot(self.item_vecs[i, :].T)
110 
111     def calculate_learning_curve(self, iter_array, test):
112         """
113         Keep track of MSE as a function of training iterations.
114 
115         Params
116         ======
117         iter_array : (list)
118             List of numbers of iterations to train for each step of
119             the learning curve. e.g. [1, 5, 10, 20]
```

```
            the learning curve. e.g. [1, 5, 10, 20]
        test : (2D ndarray)
            Testing dataset (assumed to be user x item).

        The function creates two new class attributes:

        train_mse : (list)
            Training data MSE values for each value of iter_array
        test_mse : (list)
            Test data MSE values for each value of iter_array
        """
        iter_array.sort()
        self.train_mse =[]
        self.test_mse = []
        iter_diff = 0
        for (i, n_iter) in enumerate(iter_array):
            if self._v:
                print 'Iteration: {}'.format(n_iter)
            if i == 0:
                self.train(n_iter - iter_diff)
            else:
                self.partial_train(n_iter - iter_diff)

            predictions = self.predict_all()

            self.train_mse += [get_mse(predictions, self.ratings)]
            self.test_mse += [get_mse(predictions, test)]
            if self._v:
                print 'Train mse: ' + str(self.train_mse[-1])
                print 'Test mse: ' + str(self.test_mse[-1])
            iter_diff = n_iter
```

ExplicitMF_ALS.py hosted with ❤ by GitHub view raw

I've included a helper function below to quickly calculate Mean Squared Error (MSE). Let's try an initial training with 40 latent factors and no regularization and see what the learning curve looks like.

```
In [8]: from sklearn.metrics import mean_squared_error

        def get_mse(pred, actual):
            # Ignore nonzero terms.
            pred = pred[actual.nonzero()].flatten()
            actual = actual[actual.nonzero()].flatten()
            return mean_squared_error(pred, actual)
```

```
In [9]: MF_ALS = ExplicitMF(train, n_factors=40, \
                            user_reg=0.0, item_reg=0.0)
        iter_array = [1, 2, 5, 10, 25, 50, 100]
        MF_ALS.calculate_learning_curve(iter_array, test)
```

```
In [10]: %matplotlib inline
         import matplotlib.pyplot as plt
         import seaborn as sns
         sns.set()

         def plot_learning_curve(iter_array, model):
             plt.plot(iter_array, model.train_mse, \
                      label='Training', linewidth=5)
             plt.plot(iter_array, model.test_mse, \
                      label='Test', linewidth=5)


             plt.xticks(fontsize=16);
```

insight-nb3.ipynb hosted with ❤ by GitHub view raw

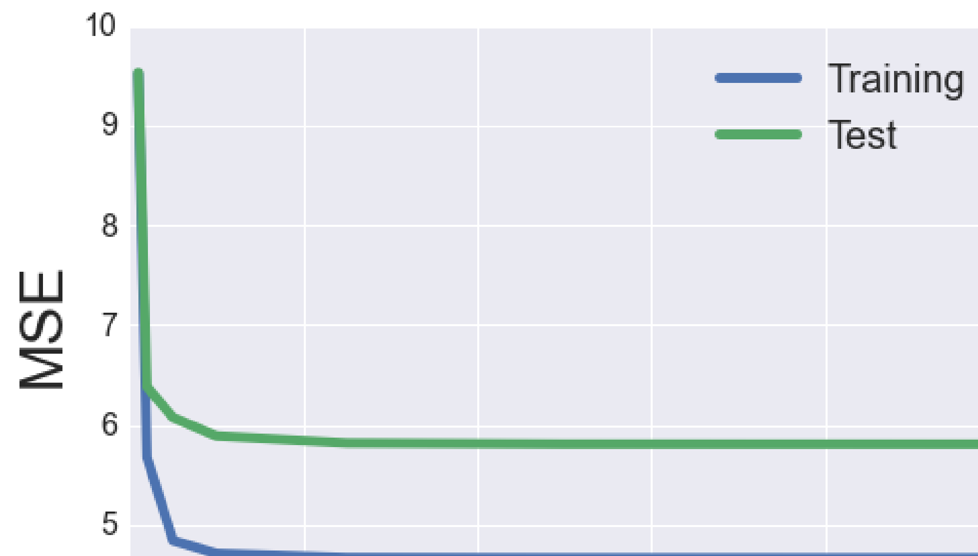
Optimizing ALS model parameters

Looks like we have a reasonable amount of overfitting (our test MSE is ~50% greater than our training MSE). Also, the test MSE bottoms out around 5 iterations then actually increases after that (even more overfitting). We can try adding some regularization to see if this helps to alleviate some of the overfitting.

```
In [12]: MF_ALS = ExplicitMF(train, n_factors=40, \
                             user_reg=30., item_reg=30.)

         iter_array = [1, 2, 5, 10, 25, 50, 100]
         MF_ALS.calculate_learning_curve(iter_array, test)

In [13]: plot_learning_curve(iter_array, MF_ALS)
```



insight-nb4.ipynb hosted with ❤ by GitHub view raw

Hmmm, the regularization narrowed the gap between our training and test MSE, but it did not decrease the test MSE too much. We could spend all day searching for optimal hyperparameters. We'll just setup a small grid search and tune both the regularization terms and number of latent factors. The item and user regularization terms will be restricted to be equal to each other.

```
In [14]: latent_factors = [5, 10, 20, 40, 80]
         regularizations = [0.1, 1., 10., 100.]
         regularizations.sort()
         iter_array = [1, 2, 5, 10, 25, 50, 100]

         best_params = {}
         best_params['n_factors'] = latent_factors[0]
         best_params['reg'] = regularizations[0]
         best_params['n_iter'] = 0
         best_params['train_mse'] = np.inf
         best_params['test_mse'] = np.inf
         best_params['model'] = None

         for fact in latent_factors:
             print 'Factors: {}'.format(fact)
             for reg in regularizations:
                 print 'Regularization: {}'.format(reg)
                 MF_ALS = ExplicitMF(train, n_factors=fact, \
                                     user_reg=reg, item_reg=reg)
                 MF_ALS.calculate_learning_curve(iter_array, test)
```

```
            MF_ALS.calculate_learning_curve(iter_array, test)
            min_idx = np.argmin(MF_ALS.test_mse)
            if MF_ALS.test_mse[min_idx] < best_params['test_mse'
]:
                best_params['n_factors'] = fact
                best_params['reg'] = reg
                best_params['n_iter'] = iter_array[min_idx]
                best_params['train_mse'] = MF_ALS.train_mse[min
```

insight-nb5.ipynb hosted with ❤ by GitHub view raw

So it looks like the best performing parameters were 10 factors and a regularization value of 0.1. It creeps me out a bit that the test set error is actually *lower* than the training error, but so be it. Before visualizing the recommendaton results of our ALS algorithm, let's first explore the other minimization algorithm: stochastic gradient descent (SGD).

Stochastic Gradient Descent as an alternative training algorithm

With SGD, we again take derivatives of the loss function, but we take the derivative with respect to each variable in the model. The "stochastic" aspect of the algorithm involves taking the derivative and updating feature weights one individual sample at a time. So, for each sample, we take the derivative of each variable, set them all equal to zero, solve for the feature weights, and update each feature. Somehow this method actually converges.

SGD Derivation

We will use a similar loss function to before, but I am going to add some more details to the model. Instead of assuming that a user *u*'s rating for item *i* can be described simply by the dot product of the user and item latent vectors, we will consider that each user and item can have a bias term associated with them. The rational is that certan users might tend to rate all movies highly, or certain movies may tend to always have low ratings. The way that I think about it is that the bias term takes care of the "DC" part of the signal which allows the latent factors to account for the more detailed variance in signal (kind of like the AC part). We will also include a global bias term as well. With all things combined, our predicted rating becomes

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{x}_u^\intercal \cdot \mathbf{y}_i$$

$$L = \sum_{u,i} (r_{ui} - (\mu + b_u + b_i + \mathbf{x}_u^\intercal \cdot \mathbf{y}_i))^2 + \lambda_{xb} \sum_u \|b_u\|^2 + \lambda_{yb} \sum_i \|b_i\|^2 + \lambda_{xf} \sum_u \|\mathbf{x}_u\|^2 + \lambda_{yf} \sum_u \|\mathbf{y}_i\|^2$$

where we have added on extra bias regularization terms. We want to update each feature (user and item latent factors and bias terms) with each

sample. The update for the user bias is given by

$$b_u \leftarrow b_u - \eta \frac{\partial L}{\partial b_u}$$

where η is the *learning rate* which weights how much our update modifies the feature weights. The derivative term is given by

$$\frac{\partial L}{\partial b_u} = 2(r_{ui} - (\mu + b_u + b_i + \mathbf{x}_u^\intercal \cdot \mathbf{y}_i))(-1) + 2\lambda_{xb} b_u$$
$$\frac{\partial L}{\partial b_u} = 2(e_{ui})(-1) + 2\lambda_{xb} b_u$$
$$\frac{\partial L}{\partial b_u} = -e_{ui} + \lambda_{xb} b_u$$

where *e_ui* represents the error in our prediction, and we have dropped the factor of 2 (we can assume it gets rolled up in the learning rate). For all of our features, the updates end up being

$$b_u \leftarrow b_u + \eta\left(e_{ui} - \lambda_{xb} b_u\right)$$

$$b_i \leftarrow b_i + \eta \left(e_{ui} - \lambda_{yb} b_i \right)$$

$$\mathbf{x}_u \leftarrow \mathbf{x}_u + \eta \left(e_{ui} \mathbf{y}_i - \lambda_{xf} \mathbf{x}_u \right)$$

$$\mathbf{y}_i \leftarrow \mathbf{y}_i + \eta \left(e_{ui} \mathbf{x}_u - \lambda_{yf} \mathbf{y}_i \right)$$

Computation

I've modified the original ExplicitMF class to allow for either sgd or als learning. The modified class is located at this gist. Similar to the ALS section above, let's try looking at the learning curve for 40 latent factors, no regularizaton, and a learning rate of 0.001.

```
1   class ExplicitMF():
2       def __init__(self,
3                    ratings,
4                    n_factors=40,
5                    learning='sgd',
6                    item_fact_reg=0.0,
7                    user_fact_reg=0.0,
8                    item_bias_reg=0.0,
9                    user_bias_reg=0.0,
10                   verbose=False):
11          """
12          Train a matrix factorization model to predict empty
13          entries in a matrix. The terminology assumes a
14          ratings matrix which is ~ user x item
```

```
15
16          Params
17          ======
18          ratings : (ndarray)
19              User x Item matrix with corresponding ratings
20
21          n_factors : (int)
22              Number of latent factors to use in matrix
23              factorization model
24          learning : (str)
25              Method of optimization. Options include
26              'sgd' or 'als'.
27
28          item_fact_reg : (float)
29              Regularization term for item latent factors
30
31          user_fact_reg : (float)
32              Regularization term for user latent factors
33
34          item_bias_reg : (float)
35              Regularization term for item biases
36
37          user_bias_reg : (float)
38              Regularization term for user biases
39
40          verbose : (bool)
41              Whether or not to printout training progress
42          """
43
44          self.ratings = ratings
45          self.n_users, self.n_items = ratings.shape
46          self.n_factors = n_factors
47          self.item_fact_reg = item_fact_reg
```

```
        self.user_fact_reg = user_fact_reg
        self.item_bias_reg = item_bias_reg
        self.user_bias_reg = user_bias_reg
        self.learning = learning
        if self.learning == 'sgd':
            self.sample_row, self.sample_col = self.ratings.nonzero()
            self.n_samples = len(self.sample_row)
        self._v = verbose

    def als_step(self,
                 latent_vectors,
                 fixed_vecs,
                 ratings,
                 _lambda,
                 type='user'):
        """
        One of the two ALS steps. Solve for the latent vectors
        specified by type.
        """
        if type == 'user':
            # Precompute
            YTY = fixed_vecs.T.dot(fixed_vecs)
            lambdaI = np.eye(YTY.shape[0]) * _lambda

            for u in xrange(latent_vectors.shape[0]):
                latent_vectors[u, :] = solve((YTY + lambdaI),
                                             ratings[u, :].dot(fixed_vecs))
        elif type == 'item':
            # Precompute
            XTX = fixed_vecs.T.dot(fixed_vecs)
            lambdaI = np.eye(XTX.shape[0]) * _lambda

```

```
80              for i in xrange(latent_vectors.shape[0]):
81                  latent_vectors[i, :] = solve((XTX + lambdaI),
82                                               ratings[:, i].T.dot(fixed_vecs))
83          return latent_vectors
84
85      def train(self, n_iter=10, learning_rate=0.1):
86          """ Train model for n_iter iterations from scratch."""
87          # initialize latent vectors
88          self.user_vecs = np.random.normal(scale=1./self.n_factors,\
89                                            size=(self.n_users, self.n_factors))
90          self.item_vecs = np.random.normal(scale=1./self.n_factors,
91                                            size=(self.n_items, self.n_factors))
92
93          if self.learning == 'als':
94              self.partial_train(n_iter)
95          elif self.learning == 'sgd':
96              self.learning_rate = learning_rate
97              self.user_bias = np.zeros(self.n_users)
98              self.item_bias = np.zeros(self.n_items)
99              self.global_bias = np.mean(self.ratings[np.where(self.ratings != 0)])
100             self.partial_train(n_iter)
101
102
103     def partial_train(self, n_iter):
104         """
105         Train model for n_iter iterations. Can be
106         called multiple times for further training.
107         """
108         ctr = 1
109         while ctr <= n_iter:
110             if ctr % 10 == 0 and self._v:
111                 print '\tcurrent iteration: {}'.format(ctr)
112             if self.learning == 'als':
```

```
113                 self.user_vecs = self.als_step(self.user_vecs,
114                                                self.item_vecs,
115                                                self.ratings,
116                                                self.user_fact_reg,
117                                                type='user')
118                 self.item_vecs = self.als_step(self.item_vecs,
119                                                self.user_vecs,
120                                                self.ratings,
121                                                self.item_fact_reg,
122                                                type='item')
123             elif self.learning == 'sgd':
124                 self.training_indices = np.arange(self.n_samples)
125                 np.random.shuffle(self.training_indices)
126                 self.sgd()
127             ctr += 1
128 
129     def sgd(self):
130         for idx in self.training_indices:
131             u = self.sample_row[idx]
132             i = self.sample_col[idx]
133             prediction = self.predict(u, i)
134             e = (self.ratings[u,i] - prediction) # error
135 
136             # Update biases
137             self.user_bias[u] += self.learning_rate * \
138                                 (e - self.user_bias_reg * self.user_bias[u])
139             self.item_bias[i] += self.learning_rate * \
140                                 (e - self.item_bias_reg * self.item_bias[i])
141 
142             #Update latent factors
143             self.user_vecs[u, :] += self.learning_rate * \
144                                     (e * self.item_vecs[i, :] - \
145                                      self.user_fact_reg * self.user_vecs[u,:])
```

```
146                 self.item_vecs[i, :] += self.learning_rate * \
147                                         (e * self.user_vecs[u, :] - \
148                                          self.item_fact_reg * self.item_vecs[i,:])
149         def predict(self, u, i):
150             """ Single user and item prediction."""
151             if self.learning == 'als':
152                 return self.user_vecs[u, :].dot(self.item_vecs[i, :].T)
153             elif self.learning == 'sgd':
154                 prediction = self.global_bias + self.user_bias[u] + self.item_bias[i]
155                 prediction += self.user_vecs[u, :].dot(self.item_vecs[i, :].T)
156                 return prediction
157 
158         def predict_all(self):
159             """ Predict ratings for every user and item."""
160             predictions = np.zeros((self.user_vecs.shape[0],
161                                     self.item_vecs.shape[0]))
162             for u in xrange(self.user_vecs.shape[0]):
163                 for i in xrange(self.item_vecs.shape[0]):
164                     predictions[u, i] = self.predict(u, i)
165 
166             return predictions
167 
168         def calculate_learning_curve(self, iter_array, test, learning_rate=0.1):
169             """
170             Keep track of MSE as a function of training iterations.
171 
172             Params
173             ======
174             iter_array : (list)
175                 List of numbers of iterations to train for each step of
176                 the learning curve. e.g. [1, 5, 10, 20]
177             test : (2D ndarray)
178                 Testing dataset (assumed to be user x item).
```

```
            testing dataset (assumed to be user x item).

        The function creates two new class attributes:

        train_mse : (list)
            Training data MSE values for each value of iter_array
        test_mse : (list)
            Test data MSE values for each value of iter_array
        """
        iter_array.sort()
        self.train_mse =[]
        self.test_mse = []
        iter_diff = 0
        for (i, n_iter) in enumerate(iter_array):
            if self._v:
                print 'Iteration: {}'.format(n_iter)
            if i == 0:
                self.train(n_iter - iter_diff, learning_rate)
            else:
                self.partial_train(n_iter - iter_diff)

            predictions = self.predict_all()

            self.train_mse += [get_mse(predictions, self.ratings)]
            self.test_mse += [get_mse(predictions, test)]
            if self._v:
                print 'Train mse: ' + str(self.train_mse[-1])
                print 'Test mse: ' + str(self.test_mse[-1])
            iter_diff = n_iter
```

ExplicitMF.py hosted with ❤ by GitHub view raw

```
In [14]: MF_SGD = ExplicitMF(train, 40, learning='sgd', verbose=True)
         iter_array = [1, 2, 5, 10, 25, 50, 100, 200]
         MF_SGD.calculate_learning_curve(iter_array, test, learning_r
         ate=0.001)
```

```
Iteration: 1
Train mse: 1.1419376708
Test mse: 1.07081066329
Iteration: 2
Train mse: 1.07186223696
Test mse: 1.00654383987
Iteration: 5
Train mse: 0.975972057215
Test mse: 0.926091276051
Iteration: 10
Train mse: 0.919170129465
Test mse: 0.88774317347
Iteration: 25
        current iteration: 10
Train mse: 0.868550680386
Test mse: 0.861884799308
Iteration: 50
        current iteration: 10
        current iteration: 20
Train mse: 0.842385086053
Test mse: 0.850655185536
Iteration: 100
```

insight-nb6.ipynb hosted with ❤ by GitHub view raw

Wow, quite a bit better than before! I assume that this is likely due to the inclusion of bias terms (especially because the ratings are not normalized).

Optimizing SGD model parameters

Let's try to optimize some hyperparameters. We'll start with a grid search of the learning rate.

In [16]:
```
iter_array = [1, 2, 5, 10, 25, 50, 100, 200]
learning_rates = [1e-5, 1e-4, 1e-3, 1e-2]

best_params = {}
best_params['learning_rate'] = None
best_params['n_iter'] = 0
best_params['train_mse'] = np.inf
best_params['test_mse'] = np.inf
best_params['model'] = None


for rate in learning_rates:
    print 'Rate: {}'.format(rate)
    MF_SGD = ExplicitMF(train, n_factors=40, learning='sgd')
    MF_SGD.calculate_learning_curve(iter_array, test, learni
ng_rate=rate)
    min_idx = np.argmin(MF_SGD.test_mse)
    if MF_SGD.test_mse[min_idx] < best_params['test_mse']:
        best_params['n_iter'] = iter_array[min_idx]
        best_params['learning_rate'] = rate
        best_params['train_mse'] = MF_SGD.train_mse[min_idx]
        best_params['test_mse'] = MF_SGD.test_mse[min_idx]
        best_params['model'] = MF_SGD
        print 'New optimal hyperparameters'
        print pd.Series(best_params)
```

```
Rate: 1e-05
```

insight-nb7.ipynb hosted with ❤ by GitHub view raw

Looks like a learning rate of 0.001 was the best value. Note that the best test error was for only 100 iterations, not 200 — it's likely that the model started to overfit after this point. On that note, we'll now complete the hyperparameter optimization with a grid search through regularization terms and latent factors. This takes a while and could easily be parallelized, but that's beyond the scope of this post.

In [17]:
```
iter_array = [1, 2, 5, 10, 25, 50, 100, 200]
latent_factors = [5, 10, 20, 40, 80]
regularizations = [0.001, 0.01, 0.1, 1.]
regularizations.sort()

best_params = {}
best_params['n_factors'] = latent_factors[0]
best_params['reg'] = regularizations[0]
best_params['n_iter'] = 0
best_params['train_mse'] = np.inf
best_params['test_mse'] = np.inf
best_params['model'] = None

for fact in latent_factors:
    print 'Factors: {}'.format(fact)
    for reg in regularizations:
        print 'Regularization: {}'.format(reg)
        MF_SGD = ExplicitMF(train, n_factors=fact, learning=
'sgd',\
                            user_fact_reg=reg, item_fact_reg
=reg, \
                            user_bias_reg=reg, item_bias_reg
=reg)
        MF_SGD.calculate_learning_curve(iter_array, test, le
arning_rate=0.001)
        min_idx = np.argmin(MF_SGD.test_mse)
        if MF_SGD.test_mse[min_idx] < best_params['test_mse'
```

insight-nb8.ipynb hosted with ❤ by GitHub view raw

It should be noted that both our best latent factors and best iteration count were at the maximums of their respective grid searches. In hindsight, we should have set the grid search to a wider range. In practice, I am going to just go with these parameters. We could spend all day optimizing, but this is just a blog post on extensively studied data.

Using themoviedb.org's API to eye-test recommendations

We spent a fair bit of time optimizing the MSE of our models, and we are now ready to actually make some recommendations. However, how will we really know if we are making good recommendations? Because we are dealing with a domain where many of us have intuition (movies), we can generate item-to-item recommendations and see if similar items "make sense".

And just for fun, let us really *look* at the items. The MovieLens dataset contains a file with information about each movie. It turns out that there is a website calledthemoviedb.org which has a free API. If we have the IMDB "movie id" for a movie, then we can use this API to return the posters of

movies. Looking at the movie data file below, it seems that we at least have the IMDB url for each movie.

If you follow one of the links in this dataset, then your url will get redirected. The resulting url contains the IMDB movie ID as the last information in the url starting with "tt". For example, the redirected url for ***Toy Story*** ishttp://www.imdb.com/title/tt0114709/, and the IMDB movie ID is tt0114709.

Using the Python requests library, we can automatically extract this movie ID. The***Toy Story*** example is shown in the next code snippet.

I requested a free API key from themoviedb.org. The key is necessary for querying the API. I've omitted it below, so be aware that if you will need your own key if you want to reproduce this. We can search for movie posters by movie id and then grab links to the image files. The links are relative paths, so we need the base_url query at the top of the next cell to get the full path. Also, some of the links don't work, so we can instead search for the movie by title and grab the first result.

```
In [17]: !head -5 u.item
```

```
1|Toy Story (1995)|01-Jan-1995||http://us.imdb.com/M/title-ex
act?Toy%20Story%20(1995)|0|0|0|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|
0
2|GoldenEye (1995)|01-Jan-1995||http://us.imdb.com/M/title-ex
act?GoldenEye%20(1995)|0|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0
3|Four Rooms (1995)|01-Jan-1995||http://us.imdb.com/M/title-e
xact?Four%20Rooms%20(1995)|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|
0|0
4|Get Shorty (1995)|01-Jan-1995||http://us.imdb.com/M/title-e
xact?Get%20Shorty%20(1995)|0|1|0|0|0|1|0|0|1|0|0|0|0|0|0|0|0|
0|0
5|Copycat (1995)|01-Jan-1995||http://us.imdb.com/M/title-exac
t?Copycat%20(1995)|0|0|0|0|0|0|1|0|1|0|0|0|0|0|0|0|1|0|0
```

In [18]:

```
import requests
import json

response = requests.get('http://us.imdb.com/M/title-exact?To
y%20Story%20(1995)')
print response.url.split('/')[-2]
```

```
tt0114709
```

In [19]:

```
# Build function to query themoviedb.org's API
```

insight-nb9.ipynb hosted with ❤ by GitHub view raw

Ta-da! Now we have a pipeline to go directly from the IMDB url in the data file to displaying the movie poster. With this machinery in hand, let us investigate our movie-to-movie similarity by visualizing the top-5 most similar movie posters for an input movie. We'll use the cosine similarity of the item latent vectors to calculate the similarity. Let's go for gold and use

the entire dataset to train the latent vectors and calculate similarity. We'll do this for both ALS and SGD models and compare the results.

We start by training both models with the best parameters we found. I'll also use a small function to calculate both the ALS and the SGD movie-to-movie similarities. Lastly, let's read in the movie's IMDB urls and use those to uery themoviedb.org API.

In [21]:
```
best_als_model = ExplicitMF(ratings, n_factors=10, learning=
'als', \
                            item_fact_reg=0.1, user_fact_reg
=0.1)
best_als_model.train(100)
```

In [22]:
```
best_sgd_model = ExplicitMF(ratings, n_factors=80, learning=
'sgd', \
                            item_fact_reg=0.01, user_fact_re
g=0.01, \
                            user_bias_reg=0.01, item_bias_re
g=0.01)
best_sgd_model.train(200, learning_rate=0.001)
```

In [23]:
```
def cosine_similarity(model):
    sim = model.item_vecs.dot(model.item_vecs.T)
    norms = np.array([np.sqrt(np.diagonal(sim))])
    return sim / norms / norms.T

als_sim = cosine_similarity(best_als_model)
sgd_sim = cosine_similarity(best_sgd_model)
```

In [24]:
```
# Load in movie data
```

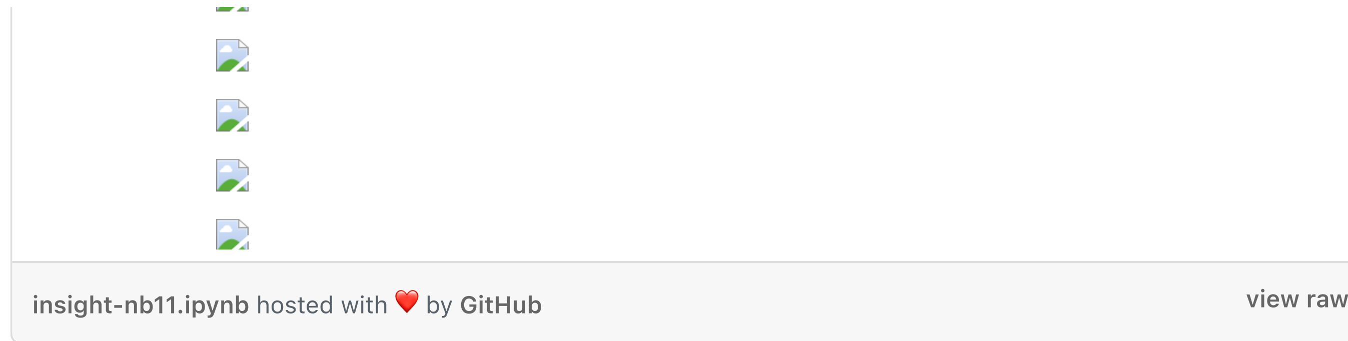
```
idx_to_movie = {}
with open('u.item', 'r') as f:
```

insight-nb10.ipynb hosted with ❤ by GitHub view raw

To visualize the posters in the Jupyter notebook's cells, we can use the IPython.display method. Special thanks to this Stack Overflow answer for the idea to use straight HTML.

I'll let you look through 5 different movie-to-movie recommendations below.

```
In [25]: idx = 0 # Toy Story
         compare_recs(als_sim, sgd_sim, idx_to_movie, idx, base_url,
         api_key)
```



So how do we think we did? I find it very interesting that the best test MSE for our ALS model was 5.04 compared to 0.76 for SGD. That's a giant difference, and yet I think the ALS recommendations might actually beat out the SGD ones; particularly, the GoldenEye and Dumbo recommendations.

I have found similar behavior in some of my own work, as well. I have a vague hunch that SGD tends to overfit more than ALS and is more susceptible to popularity bias. Unfortunately, I have zero math to back this up, so it'll remain purely anecdotal for now. An alternative explanation could be that the SGD movie-to-movie recommendations are actually better than the ALS ones even if they seem like less similar movies. In a recent Netflix paper, they show a similar comparison of two different models' movie-to-movie recommendations. It turns out that the model with movies that look less similar by eye (but are generally more popular movies)

performs better on A/B tests. And really, A/B tests are a much better way of truly benchmarking recommendation systems compared to this offline data modeling.

And on a final note, maybe we would want to just combine both models into an ensemble which seems to be what everbody does nowadays.

. . .

Interested in transitioning to career in data science?
Find out more about the Insight Data Science Fellows Program in New York and Silicon Valley, apply today, or sign up for program updates.

Already a data scientist or engineer?
Find out more about our Advanced Workshops for Data Professionals. Register for two-day workshops in Apache Spark and Data Visualization, or sign up for workshop updates.

Insight Data Science | Data Science

Learn more.

Medium is an open platform where 170 million readers come to find insightful and dynamic thinking. Here, expert and undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. Learn more

Make Medium yours.

Follow the writers, publications, and topics that matter to you, and you'll see them on your homepage and in your inbox. Explore

Write a story on Medium.

If you have a story to tell, knowledge to share, or a perspective to offer — welcome home. It's easy and free to post your thinking on any topic. Start a blog

About Write Help Legal