Movie_Similarity_Model

December 13, 2016

In [6]:
```
import heapq
import math
import time
from collections import defaultdict
from collections import namedtuple
from contextlib import contextmanager
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split

%matplotlib inline
```

In [2]:
```
@contextmanager
def elapsed_time(title):
    start = time.time()
    yield
    elapsed = time.time() - start
    print '%s: %.2f secs' % (title, elapsed)


def get_xy(ratings_df):
    y = ratings_df['rating']
    x = ratings_df.drop('rating', axis=1)
    return x, y


def date_parse(time_in_secs):
    return datetime.utcfromtimestamp(float(time_in_secs))


def read_ratings_df_with_timestamp(file_name):
    with elapsed_time('loaded csv'):
```

```
            ratings_df = pd.read_csv(file_name, parse_dates=['timestamp'], date
        return ratings_df


    def root_mean_squared_error(y, y_pred):
        return math.sqrt(mean_squared_error(y, y_pred))
```

In [3]:
```
class BaselineModel(object):
    def predict_rating(self, user_id, movie_id):
        pass

    def predict(self, x):
        return [self.predict_rating(row['userId'], row['movieId']) for _, r

    def score(self, x, y):
        return r2_score(y, self.predict(x))


class BaselineEffectsModel(BaselineModel):
    def __init__(self, movie_lambda=5.0, user_lambda=20.0):
        self.movie_lambda = movie_lambda
        self.user_lambda = user_lambda

        self.y_mean = None
        self.movie_effects = None
        self.user_effects = None
        self.user_groups = None

    def calculate_movie_effect(self, ratings):
        return (ratings - self.y_mean).sum() / (self.movie_lambda + len(rat

    def calculate_movie_effects(self, movie_ratings):
        return movie_ratings.agg(lambda ratings: self.calculate_movie_effec

    def calculate_user_effect(self, ratings_df):
        s = 0.0
        for _, row in ratings_df.iterrows():
            s += row['rating'] - self.y_mean - self.movie_effects[row['movi

        return s / (self.user_lambda + len(ratings_df))

    def calculate_user_effects(self, user_groups):
        user_ids = []
        user_effects = []

        for user_id, group in user_groups:
            user_effect = self.calculate_user_effect(group)
```

```
            user_ids.append(user_id)
            user_effects.append(user_effect)

        return pd.Series(user_effects, index=user_ids)

    def fit(self, ratings_df):
        with elapsed_time('effects init'):
            _, y_train = get_xy(ratings_df)
            self.y_mean = y_train.mean()

            movie_ratings = ratings_df.groupby('movieId')['rating']
            self.user_groups = ratings_df.groupby('userId')

            self.movie_effects = self.calculate_movie_effects(movie_ratings
            self.user_effects = self.calculate_user_effects(self.user_grou

        return self

    def create_modified_ratings(self, ratings_df):
        ratings_df = ratings_df.copy()

        for index, row in ratings_df.iterrows():
            user_id = row['userId']
            movie_id = row['movieId']
            rating = row['rating']
            pred_rating = self.predict_baseline_rating(user_id, movie_id)

            residual = rating - pred_rating

            ratings_df.loc[index, 'rating'] = residual

        return ratings_df

    def predict_baseline_rating(self, user_id, movie_id):
        return self.y_mean + self.movie_effects.get(movie_id, 0.0) + self.u

    def predict_rating(self, user_id, movie_id):
        return self.predict_baseline_rating(user_id, movie_id)
```

Movie similarity model.

First we removed all main global effects the same way as we did for our baseline model. We subtracted the total rating mean, then we removed the movie effects and then user effects.

As a result our utility matrix was the residuals after applying our baseline models.

It allowed us to remove some scale differences in the way different users rate movies.

In our movie similarity model we predict the rating for a movie (movie_id) and a user (user_id) by

1) finding k closest neighbors (k=40)

For assessing item-item similarity we used a distance based on the mean squared error between items [1]:

$s_{ij} = \frac{|U(i,j)|}{\sum_{u \in U(i,j)} (r_{ui} - r_{uj})^2 + \alpha}$, where U(i,j) is the set of users who rated both items j and k.

2) since relatively large number of movies have low number of ratings (1-3), some of the movies have zero neighbors, in this case we use a baseline prediction (in 6.9-7.2% of the cases for the test set).

[1] R.Bell, Y.Koren, "Improved Neighborhood-based Collaborative Filtering", *KDD-Cup and Workshop*, ACM press, 2007

```
In [7]: MovieSimilarity = namedtuple('MovieSimilarity', ['movie_id', 'similarity'])


        class MovieSimilarityModel(BaselineModel):
            def __init__(self, k_neighbors=40):
                self.k_neighbors = k_neighbors

                self.baseline_model = BaselineEffectsModel()
                self.ratings_by_movie = defaultdict(dict)
                self.ratings_by_user = defaultdict(dict)
                self.raters_by_movie = {}
                self.movie_similarity = {}
                # self.movie_aij = {}

            def set_k_neighbors(self, k_neighbors):
                self.k_neighbors = k_neighbors

            def calculate_common_raters(self, movie_id_1, movie_id_2):
                raters1 = self.raters_by_movie[movie_id_1]
                raters2 = self.raters_by_movie[movie_id_2]
                return raters1 & raters2

            def get_common_ratings(self, movie_id, raters):
                all_ratings = self.ratings_by_movie[movie_id]
                ratings = []
                for rater_id in raters:
                    ratings.append(all_ratings[rater_id])

                return np.array(ratings)

            def calculate_similarity(self, movie_id_1, movie_id_2):
                common_raters = self.calculate_common_raters(movie_id_1, movie_id_2
                support = len(common_raters)
                if support <= 1:
                    similarity = 0.0
                    # aij = 0.0
                else:
```

```
            ratings1 = self.get_common_ratings(movie_id_1, common_raters)
            ratings2 = self.get_common_ratings(movie_id_2, common_raters)

            alpha = 4.0

            similarity = support / (np.power(ratings1 - ratings2, 2).sum()

            # aij = np.multiply(ratings1, ratings2).sum() / support

        return similarity

    def fit(self, ratings_df):
        with elapsed_time('fit'):
            self.baseline_model.fit(ratings_df)

            ratings_df = self.baseline_model.create_modified_ratings(rating

            unique_movie_ids = np.array(sorted(ratings_df['movieId'].unique

            for _, row in ratings_df.iterrows():
                movie_id = row['movieId']
                user_id = row['userId']
                rating = row['rating']
                self.ratings_by_movie[movie_id][user_id] = rating
                self.ratings_by_user[user_id][movie_id] = rating

            for movie_id in unique_movie_ids:
                self.raters_by_movie[movie_id] = set(self.ratings_by_movie[

            for movie_index_1, movie_id_1 in enumerate(unique_movie_ids):
                for movie_index_2 in xrange(movie_index_1 + 1, len(unique_m
                    movie_id_2 = unique_movie_ids[movie_index_2]

                    similarity = self.calculate_similarity(movie_id_1, movi
                    movie_pair = (movie_id_1, movie_id_2)
                    self.movie_similarity[movie_pair] = similarity
                    # self.movie_aij[movie_pair] = aij

        return self

    def get_similarity(self, movie_id_1, movie_id_2):
        if movie_id_1 < movie_id_2:
            id_1 = movie_id_1
            id_2 = movie_id_2
        else:
            id_1 = movie_id_2
            id_2 = movie_id_1
```

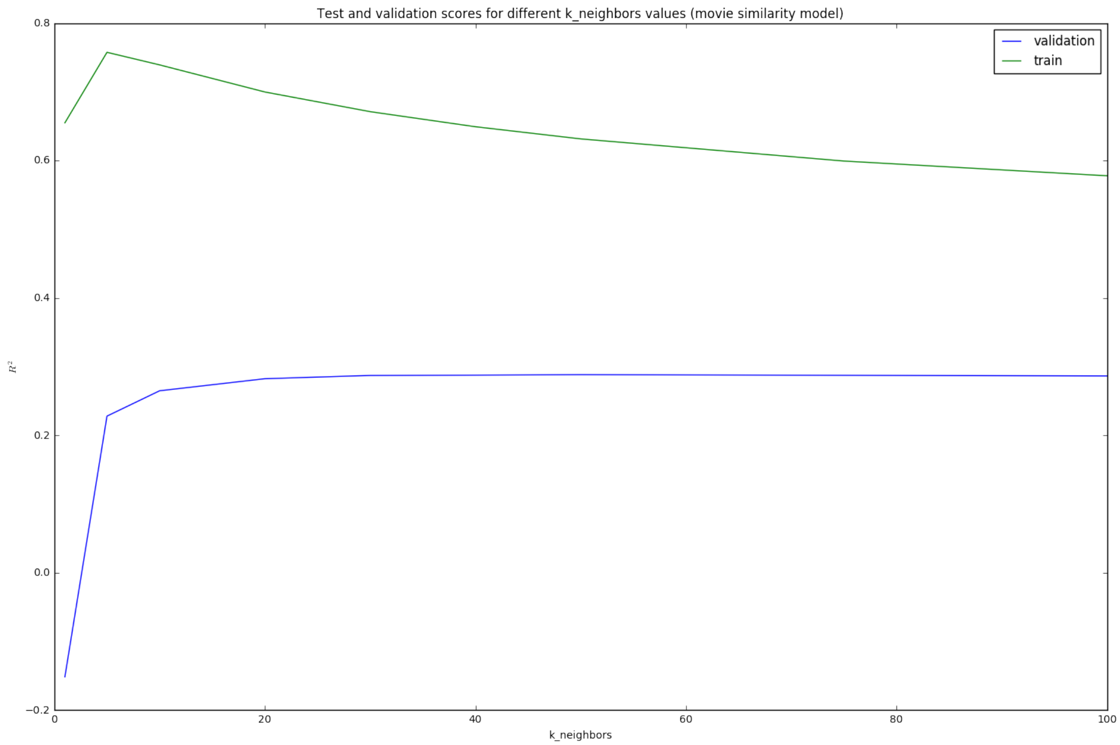
```
        return self.movie_similarity.get((id_1, id_2), -1.0)

    def clear_predict_caches(self):
        self.zero_prediction_count = 0

    def predict_rating(self, user_id, movie_id):
        ratings = self.ratings_by_user[user_id]

        elements = []

        for movie_id_2 in ratings:
            if movie_id != movie_id_2:
                similarity = self.get_similarity(movie_id, movie_id_2)
                if similarity > 0.0:
                    elements.append(MovieSimilarity(movie_id_2, similarity)

        movie_similarities = heapq.nlargest(self.k_neighbors, elements, key

        if len(movie_similarities) > 0:
            similarity_sum = 0.0
            product_sum = 0.0
            for movie_similarity in movie_similarities:
                movie_id_2 = movie_similarity.movie_id
                rating = ratings[movie_id_2]
                similarity = movie_similarity.similarity

                product_sum += similarity * rating
                similarity_sum += similarity

            rating = product_sum / similarity_sum
        else:
            rating = 0.0
            self.zero_prediction_count += 1

        result = self.baseline_model.predict_baseline_rating(user_id, movie

        return result

    def predict(self, x):
        self.clear_predict_caches()
        predictions = [self.predict_rating(row['userId'], row['movieId']) f
        print 'used baseline predictions: %.1f%%' % (100.0 * self.zero_pred
        return predictions


def show_scores_plot(k_neighbors_values, val_scores, train_scores):
    _, ax = plt.subplots(1, 1, figsize=(15, 10))
```

```python
    ax.plot(k_neighbors_values, val_scores, label='validation')
    ax.plot(k_neighbors_values, train_scores, label='train')

    ax.set_xlabel('k_neighbors')
    ax.set_ylabel('$R^2$')
    ax.set_title('Test and validation scores for different k_neighbors valu

    ax.legend(loc='best')

    plt.tight_layout()
    plt.show()


def build_model(ratings_df):
    train_val_ratings_df, test_ratings_df = train_test_split(ratings_df)

    train_ratings_df, validation_ratings_df = train_test_split(train_val_ra

    best_score = -float('inf')
    best_k_neighbors = None

    model = MovieSimilarityModel()

    model = model.fit(train_ratings_df)

    k_neighbors_values = [1, 5, 10, 20, 30, 40, 50, 75, 100]

    val_scores = []
    train_scores = []

    for k_neighbors in k_neighbors_values:
        model.set_k_neighbors(k_neighbors=k_neighbors)

        x_train, y_train = get_xy(train_ratings_df)
        x_val, y_val = get_xy(validation_ratings_df)

        y_train_pred = model.predict(x_train)
        y_val_pred = model.predict(x_val)

        train_score = r2_score(y_train, y_train_pred)
        val_score = r2_score(y_val, y_val_pred)

        if val_score > best_score:
            best_score = val_score
            best_k_neighbors = k_neighbors

        val_scores.append(val_score)
        train_scores.append(train_score)
```

```
            print 'k: %d, validation score: %.5f, train score: %.5f\n' % (k_nei
        print 'best k: %d, best score: %.5f' % (best_k_neighbors, best_score)

        model = MovieSimilarityModel(k_neighbors=best_k_neighbors)

        model = model.fit(train_val_ratings_df)

        x_train_val, y_train_val = get_xy(train_val_ratings_df)
        x_test, y_test = get_xy(test_ratings_df)

        y_train_val_pred = model.predict(x_train_val)
        y_test_pred = model.predict(x_test)

        train_val_score = r2_score(y_train_val, y_train_val_pred)
        test_score = r2_score(y_test, y_test_pred)

        train_val_rmse = root_mean_squared_error(y_train_val, y_train_val_pred)
        test_rmse = root_mean_squared_error(y_test, y_test_pred)

        print 'train score: %.4f, test score: %.4f' % (train_val_score, test_sc
        print 'train rmse: %.4f, test rmse: %.4f' % (train_val_rmse, test_rmse)

        show_scores_plot(k_neighbors_values, val_scores, train_scores)


    ratings_df = read_ratings_df_with_timestamp('ml-latest-small/ratings.csv')

    with elapsed_time('build model'):
        build_model(ratings_df)
```

```
loaded csv: 0.47 secs
effects init: 5.23 secs
fit: 82.74 secs
used baseline predictions: 4.9%
used baseline predictions: 9.0%
k: 1, validation score: -0.15159, train score: 0.65502

used baseline predictions: 4.9%
used baseline predictions: 9.0%
k: 5, validation score: 0.22808, train score: 0.75770

used baseline predictions: 4.9%
used baseline predictions: 9.0%
k: 10, validation score: 0.26498, train score: 0.73939

used baseline predictions: 4.9%
```

```
used baseline predictions: 9.0%
k: 20, validation score: 0.28253, train score: 0.69995

used baseline predictions: 4.9%
used baseline predictions: 9.0%
k: 30, validation score: 0.28729, train score: 0.67131

used baseline predictions: 4.9%
used baseline predictions: 9.0%
k: 40, validation score: 0.28775, train score: 0.64936

used baseline predictions: 4.9%
used baseline predictions: 9.0%
k: 50, validation score: 0.28838, train score: 0.63166

used baseline predictions: 4.9%
used baseline predictions: 9.0%
k: 75, validation score: 0.28754, train score: 0.59941

used baseline predictions: 4.9%
used baseline predictions: 9.0%
k: 100, validation score: 0.28654, train score: 0.57795

best k: 50, best score: 0.28838
effects init: 6.53 secs
fit: 138.46 secs
used baseline predictions: 3.9%
used baseline predictions: 7.3%
train score: 0.6495, test score: 0.3083
train rmse: 0.6264, test rmse: 0.8799
```




```
build model: 653.40 secs
```

In [ ]: