Project 4 Group 4

- Chongyu He (ch3379)
- Daniel Lee (dl3250)
- Yiwen Ma (ym2775)
- Runzi Qiang (rq2156)
- Yifan Yang (yy2955)

Introduction

As the data analysis became a key part of modern online services, number of recommendation methods have been developed. Content filtering is one recommender systems where we create a profile for each user to product to characterize its nature. For example, a movie profile could include attributes regarding its genre, participating actors, its box office popularity, etc. User profiles might include demographic information or answers provided on a suitable questionnaire. The profiles allow programs to associate users with matching products.

Our group's goal is to see the difference between the two models: SGD algorithm with temporal regularization and postprocessing SVD with KNN SGD algorithm with temporal regularization and postprocessing SVD with kernel ridge regression

Models:

1. Stochastic Gradient Descent + Temporal Dynamics + KNN Postprocessing
2. Stochastic Gradient Descent + Temporal Dynamics + Kernel Ridge Regression Postprocessing

Regularization: Temporal Dynamics

Without Temporal Dynamics

$$\hat{r} = q_i^T p_u + b_{ui}$$
$$b_{ui} = \mu + b_i + b_u$$

With Temporal Dynamics

1. Time-Changing Item Bias

$$bi(t) = b_i + b_{i,Bin(t)}$$

2. Time-Changing User Bias

$$dev_u(t) = sign(t - t_u) * |t - t_u|^{\beta}$$
$$b_u(t) = b_u + \alpha_u \times dev_u(t)$$

where

- t_u: mean date of rating by the user
- β: a hyperaparameter
- b_u: stationary portion of the user bias

1. Time-Changing User Preference

$$p_t(t) + p_u + \alpha_u \times dev_u(t)$$

where

- p_u: stationary portion
- $\alpha_u \times dev_u(t)$: the portion that changes linearly over time

Put Them All Together

$$\hat{r} = \mu + b_i(t) + b_u(t) + q_i^T p_u(t)$$
$$error = r - \hat{r}$$

Postprocessing

1. KNN

Item Similartiy:

$$s(q_i, q_j) = \frac{q_i^T q_j}{||q_i||||q_j||}$$

2. Kernel Ridge Regression

$$\hat{y}^i = K(x_i^T, X)(K(X, X) + \lambda I)^{-1} y$$

In [1]:

```
import os
import warnings
import time
import numpy as np
import numpy.linalg as npla
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime
from sklearn.preprocessing import normalize
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.kernel_ridge import KernelRidge
%matplotlib inline
```

In [2]:

```
class mfsgd(object):
    def __init__(self, filename, test_size=0.1, random_state=0, n=10, penalty=0.
5, learning_rate=0.01):
        """
        param learning_rate: minimum 1e-6
        """
        self.data = mfsgd.preprocess(filename)
        self.lr = max(learning_rate, 1e-6)
        self.origlr = max(learning_rate, 1e-6)
        self.decrement = 1
        self.nepoch = 1e6
        self.n = n
        self.penalty = penalty
        self.train_size = None
        self.validation_size = 0
        if test_size >= 1:
            raise Exception('test_size must be < 1')
        self.test_size = test_size
        self.test = self.data.groupby('userId').apply(lambda x: x.sample(frac=te
st_size, random_state=random_state)).reset_index(level=0, drop=True)
        self.n_users = len(self.data.loc[:, 'userId'].unique())
        unique_items = self.data.loc[:, 'movieId'].unique()
        self.n_items = len(unique_items)
        self.item_mapping = dict(zip(unique_items, list(range(len(unique_items
)))))
        self.time_window = (self.data.loc[:, 'timestamp'].min(), self.data.loc
[:, 'timestamp'].max()+1)

    def setLearningRateSchedule(self, start=0.01, decrement=0.1, nepoch=100):
        """
        param start: starting learning rate
        param decrement: multiplier to the learning rate per nepoch epochs
        param nepoch: number of epochs between two decrements
        """
        self.lr = start
        self.origlr = start
        self.decrement = decrement
        self.nepoch = nepoch
        return self

    def fit(self, train_size=0.7, user_nbins=10, item_nbins=3, beta=0.4, n_init=
1, n_iter=50):
        if train_size + self.test_size > 1:
            train_size = 1 - self.test_size
            warnings.warn('train size truncated to',train_size)
        pct = train_size / (1 - self.test_size)
        self.r = self.data.drop(self.test.index).groupby('userId').apply(lambda
x: x.sample(frac=pct)).reset_index(level=0, drop=True)
        self.train_size = train_size
        self.beta = beta
        self.user_nbins = user_nbins
        self.user_binsize = self.__binify(self.time_window, self.user_nbins)
        self.avg_user_bin = {k: self.__timestampToBin(v, self.user_binsize) for
k, v in self.r.groupby('userId')['timestamp'].mean().items()}
        self.item_nbins = item_nbins
        self.item_binsize = self.__binify(self.time_window, self.item_nbins)
        self.user_dict = self.r.groupby('userId')['movieId']
        self.ru = self.user_dict.count().apply(lambda x:x**(-0.5))
        self.train_loss = np.nan
```

```
        for i in range(n_init):
            result = self.__trainEach(n_iter)
            if np.isnan(self.train_loss) or result['loss'] < self.train_loss:
                self.mu = result['mu']
                self.q = result['q']
                self.p_user = result['p_user']
                self.pa_user = result['pa_user']
                self.b_user = result['b_user']
                self.a_user = result['a_user']
                self.b_item = result['b_item']
                self.b_item_bin = result['b_item_bin']
                self.y = result['y']
                self.train_loss = result['loss']
        self.__resetLR()
        return self

    def validate(self):
        if self.train_size + self.test_size == 1:
            warnings.warn('no data can be used to validate')
            return
        self.validation_size = 1 - self.train_size - self.train_size
        self.validation = self.data.drop(self.test.index.union(self.r.index)).gr
oupby('userId').reset_index(level=0, drop=True)
        rmse, r_pred = self.__computeLoss(dataset='validation')
        print('validation rmse:', rmse)
        return r_pred

    def predict(self, method='RSVD', **kwargs):
        if self.train_size is None:
            raise Exception('model is not trained')
        if method == 'RSVD':
            rmse, r_pred = self.__computeLoss(dataset='test')
        else:
            rmse, r_pred = self.__computeDefLoss(method, **kwargs)
        print(method, 'test rmse:', rmse)
        return r_pred

    def __trainEach(self, n_iter):
        mu = np.random.uniform(-0.01, 0.01, 1)
        q = np.random.uniform(-0.01, 0.01, (self.n, self.n_items))
        p_user = np.random.uniform(-0.01, 0.01, (self.n, self.n_users))
        pa_user = np.random.uniform(-0.01, 0.01, (self.n, self.n_users))
        b_user = np.random.uniform(-0.01, 0.01, self.n_users)
        a_user = np.random.uniform(-0.01, 0.01, self.n_users)
        b_item = np.random.uniform(-0.01, 0.01, self.n_items)
        b_item_bin = np.random.uniform(-0.01, 0.01, (self.item_nbins, self.n_ite
ms))
        y = np.random.uniform(-0.01, 0.01, (self.n, self.n_items))

        c = 0
        for it in range(n_iter):
            loss = 0
            sTime = time.time()
            for ind, s in self.r.iterrows():
                u, i, r, t = int(s['userId'])-1, self.item_mapping[int(s['movieI
d'])], s['rating'], s['timestamp']
                pu, pua, qi = p_user[:, u], pa_user[:, u], q[:, i]
                i_bin = self.__timestampToBin(t, self.item_binsize)
                bi, bibin = b_item[i], b_item_bin[i_bin, i]
                bu, au = b_user[u], a_user[u]
                dev = self.__dev(self.__timestampToBin(t, self.user_binsize), se
```

```
lf.avg_user_bin[u+1], self.beta)
                ru = self.ru[u+1]
                user_items = [self.item_mapping[x] for x in self.user_dict.get_g
roup(u+1)]
                yu = np.sum(y[:, user_items], axis=1)
                r_hat = mu+bi+bibin+bu+au*dev+qi@(pu+pua*dev+ru*yu)
                res = r - r_hat
                # update based on gradient
                mu -= self.lr * self.__muDeriv(res)
                q[:,i] -= self.lr * self.__qDeriv(res, pu, pua, qi, ru, yu, dev)
                p_user[:,u] -= self.lr * self.__puDeriv(res, pu, qi)
                pa_user[:, u] -= self.lr * self.__puaDeriv(res, pua, qi, dev)
                b_user[u] -= self.lr * self.__buDeriv(res, bu)
                a_user[u] -= self.lr *self.__auDeriv(res, au, dev)
                b_item[i] -= self.lr * self.__biDeriv(res, bi)
                b_item_bin[i_bin, i] -= self.lr * self.__bibinDeriv(res, bibin)
                y[:, user_items] -= self.lr * self.__yuDeriv(res, qi, ru, y[:, u
ser_items])

                loss += res**2
            # update learning rate
            c += 1
            if not c%self.nepoch:
                self.lr = max(self.lr * self.decrement, 1e-6)

            # use avg residual as loss
            loss = np.sqrt(loss / len(self.r))
            execTime = time.time() - sTime

            print('epoch', it+1, '----learning rate: {:.6f}'.format(self.lr), '-
---unpenalized training loss:', loss,
                 '----execution time: %s'%execTime)

        return {'loss':loss,
                'mu':mu,
                'q':q,
                'p_user':p_user,
                'pa_user':pa_user,
                'b_user':b_user,
                'a_user':a_user,
                'b_item':b_item,
                'b_item_bin':b_item_bin,
                'y':y}

    def __computeLoss(self, dataset='train', **kwargs):
        loss = 0
        r_pred = None
        if dataset == 'train':
            data = self.r
            mu, q, p_user, pa_user, b_user, a_user, b_item, b_item_bin, y = kwar
gs['mu'], kwargs['q'], kwargs['p_user'], kwargs['pa_user'], kwargs['b_user'], kw
args['a_user'], kwargs['b_item'], kwargs['b_item_bin'], kwargs['y']
        elif dataset in ['test', 'validation']:
            data = self.test if dataset == 'test' else self.validation
            r_pred = np.zeros(len(data))
            mu, q, p_user, pa_user, b_user, a_user, b_item, b_item_bin, y = self
.mu, self.q, self.p_user, self.pa_user, self.b_user, self.a_user, self.b_item, s
elf.b_item_bin, self.y
        else:
            raise Exception('ambiguous compute loss inputs')
```

```
        for ind, s in data.reset_index().iterrows():
            u, i, r, t = int(s['userId'])-1, self.item_mapping[int(s['movieId'
])], s['rating'], s['timestamp']
            pu, pua, qi = p_user[:, u], pa_user[:, u], q[:, i]
            bi, bibin = b_item[i], b_item_bin[self.__timestampToBin(t, self.item
_binsize), i]
            bu, au = b_user[u], a_user[u]
            dev = self.__dev(self.__timestampToBin(t, self.user_binsize), self.a
vg_user_bin[u+1], self.beta)
            ru = self.ru[u+1]
            user_items = [self.item_mapping[x] for x in self.user_dict.get_group
(u+1)]
            yu = np.sum(y[:, user_items], axis=1)
            r_hat = mu+bi+bibin+bu+au*dev+qi@(pu+pua*dev+ru*yu)
            res = (r-r_hat)**2
            if dataset == 'train':
                loss += res + self.penalty*(bi**2+bibin**2+bu**2+au**2+npla.norm
(pu)**2+npla.norm(pua)**2+npla.norm(qi)**2)
            else:
                loss += res
                r_pred[ind] = r_hat

        return np.sqrt(loss / len(data)), r_pred

    def __computeDefLoss(self, method='KNN', **kwargs):
        gb = self.test.groupby('userId')
        sum_res = 0
        all_r = []
        all_r_pred = []
        for user in gb.groups.keys():
            item_ind = sorted([self.item_mapping[x] for x in self.user_dict.get_
group(user)])
            X = normalize(np.transpose(self.q[:, item_ind]))
            y = np.array(self.r.groupby('userId').get_group(user).sort_values(by
='movieId')['rating'])
            test_item_ind = [self.item_mapping[x] for x in gb.get_group(user)['m
ovieId']]
            test_X = normalize(np.transpose(self.q[:, test_item_ind]))
            r = gb.get_group(user)['rating']
            all_r.append(r)
            if method == 'KNN':
                y = [str(x) for x in y]
                r_pred = KNeighborsClassifier(**kwargs).fit(X, y).predict(test_X
)
                r_pred = np.array([float(x) for x in r_pred])
            elif method == 'KernelRidge':
                r_pred = KernelRidge(**kwargs).fit(X, y).predict(test_X)
            else:
                raise Exception('NYI')
            all_r_pred.append(r_pred)
        all_r = np.concatenate(all_r)
        all_r_pred = np.concatenate(all_r_pred)
        rmse = np.sqrt(np.sum((all_r - all_r_pred)**2) / len(self.test))

        return rmse, all_r_pred

    # FIXME, update qDeriv
    def __muDeriv(self, res):
        return -res

    def __qDeriv(self, res, pu, pua, qi, ru, yu, dev):
```

```
        return -res * (pu+pua*dev+ru*yu) + self.penalty * qi

    def __puDeriv(self, res, pu, qi):
        return -res * qi + self.penalty * pu

    def __puaDeriv(self, res, pua, qi, dev):
        return -res * qi * dev + self.penalty * pua

    def __buDeriv(self, res, bu):
        return -res + self.penalty * bu

    def __auDeriv(self, res, au, dev):
        return -res * dev + self.penalty * au

    def __biDeriv(self, res, bi):
        return -res + self.penalty * bi

    def __bibinDeriv(self, res, bibin):
        return -res + self.penalty * bibin

    def __yuDeriv(self, res, qi, ru, yu):
        return -res * qi[:, np.newaxis] * ru + self.penalty * yu

    # FIXME, add y and R(u)^(-1/2)
    def __dev(self, t, avg, b):
        return np.sign(t-avg) * np.abs(t-avg)**b

    def __binify(self, window, nbins):
        return (window[1] - window[0]) / nbins

    def __timestampToBin(self, t, binsize):
        if t < self.time_window[0] or t > self.time_window[1]:
            raise Exception('t outside of time window')
        return int((t - self.time_window[0]) // binsize)

    def __resetLR(self):
        self.lr = self.origlr
        return

    @staticmethod
    def preprocess(filename):
        data = pd.read_csv(filename)
        return data
```

In [3]:

```
f = os.path.join('G:\mawenwen\Columbia\Fall 2019\Applied Data Science\proj4','fa
ll2019-project4-sec1-grp4-master\data\ml-latest-small','ratings.csv')
```

In [4]:

```
s = mfsgd(filename=f, test_size=0.1, n=30, penalty=0.1) # learning rate should n
ot be > 0.1 as it results in overflow in loss calculation
s.setLearningRateSchedule(start=0.05, decrement=0.2, nepoch=5)
```

Out[4]:

```
<__main__.mfsgd at 0x202f4a3c710>
```

In [5]:

```
s.fit(train_size=0.9, user_nbins=10, item_nbins=3, beta=0.6, n_iter=30)
```

```
epoch 1 ----learning rate: 0.050000 ----unpenalized training loss:
[0.91545374] ----execution time: 74.87599110603333
epoch 2 ----learning rate: 0.050000 ----unpenalized training loss:
[0.86125939] ----execution time: 75.81782841682434
epoch 3 ----learning rate: 0.050000 ----unpenalized training loss:
[0.8280277] ----execution time: 82.8188533782959
epoch 4 ----learning rate: 0.050000 ----unpenalized training loss:
[0.79321738] ----execution time: 99.7977340221405
epoch 5 ----learning rate: 0.010000 ----unpenalized training loss:
[0.75668924] ----execution time: 90.78104066848755
epoch 6 ----learning rate: 0.010000 ----unpenalized training loss:
[0.70522007] ----execution time: 75.35979270935059
epoch 7 ----learning rate: 0.010000 ----unpenalized training loss:
[0.69489607] ----execution time: 76.86985754966736
epoch 8 ----learning rate: 0.010000 ----unpenalized training loss:
[0.68831764] ----execution time: 74.90982174873352
epoch 9 ----learning rate: 0.010000 ----unpenalized training loss:
[0.68274608] ----execution time: 74.87777900695801
epoch 10 ----learning rate: 0.002000 ----unpenalized training loss:
[0.67762191] ----execution time: 74.65812110900879
epoch 11 ----learning rate: 0.002000 ----unpenalized training loss:
[0.67556884] ----execution time: 75.55739617347717
epoch 12 ----learning rate: 0.002000 ----unpenalized training loss:
[0.67228036] ----execution time: 74.9938645362854
epoch 13 ----learning rate: 0.002000 ----unpenalized training loss:
[0.67037078] ----execution time: 74.44936466217041
epoch 14 ----learning rate: 0.002000 ----unpenalized training loss:
[0.668883] ----execution time: 74.88188767433167
epoch 15 ----learning rate: 0.000400 ----unpenalized training loss:
[0.66759262] ----execution time: 75.72918128967285
epoch 16 ----learning rate: 0.000400 ----unpenalized training loss:
[0.67035985] ----execution time: 76.40582227706909
epoch 17 ----learning rate: 0.000400 ----unpenalized training loss:
[0.66927267] ----execution time: 76.19786334037781
epoch 18 ----learning rate: 0.000400 ----unpenalized training loss:
[0.66841737] ----execution time: 76.11367201805115
epoch 19 ----learning rate: 0.000400 ----unpenalized training loss:
[0.66772831] ----execution time: 77.4092960357666
epoch 20 ----learning rate: 0.000080 ----unpenalized training loss:
[0.66714392] ----execution time: 77.21005630493164
epoch 21 ----learning rate: 0.000080 ----unpenalized training loss:
[0.66920818] ----execution time: 76.80108308792114
epoch 22 ----learning rate: 0.000080 ----unpenalized training loss:
[0.66851072] ----execution time: 76.78957319259644
epoch 23 ----learning rate: 0.000080 ----unpenalized training loss:
[0.66825981] ----execution time: 76.83793902397156
epoch 24 ----learning rate: 0.000080 ----unpenalized training loss:
[0.66802671] ----execution time: 77.15731358528137
epoch 25 ----learning rate: 0.000016 ----unpenalized training loss:
[0.66780908] ----execution time: 76.28913354873657
epoch 26 ----learning rate: 0.000016 ----unpenalized training loss:
[0.66821328] ----execution time: 76.4177017211914
epoch 27 ----learning rate: 0.000016 ----unpenalized training loss:
[0.66798625] ----execution time: 76.32525038719177
epoch 28 ----learning rate: 0.000016 ----unpenalized training loss:
[0.66791368] ----execution time: 76.12155747413635
epoch 29 ----learning rate: 0.000016 ----unpenalized training loss:
[0.66786152] ----execution time: 80.2863883972168
epoch 30 ----learning rate: 0.000003 ----unpenalized training loss:
[0.66781372] ----execution time: 80.72739577293396
```

Out[5]:

```
<__main__.mfsgd at 0x202f4a3c710>
```

In [ ]:

```
r_validate = s.validate() # return predicted ratings
```

In [6]:

```
r_test_RSVD = s.predict(method='RSVD')
```

```
RSVD test rmse: [0.85535767]
```

In [37]:

```
r_test_KNN = s.predict(method='KNN', n_neighbors=1) # return predicted ratings
```

```
KNN test rmse: 1.321723622393826
```

In [55]:

```
r_test_ridge = s.predict(method='KernelRidge', alpha=0.5, kernel='rbf', gamma=0.
5)
```

```
KernelRidge test rmse: 0.9883413377798275
```

Conclusion

RSVD gives the best out-of-sample RSVD of 0.855, while kernel ridge and KNN deliver 0.988 and 1.322 RMSE respectively. The outperformance Kernel ridge over KNN is expected as KNN is a classification method. Classification disallows the existence of ambiguous ratings (such as 3.25), so the expense of misclassification is expected to be bigger than kernel ridge regression.

The outperformance of RSVD could be explained by two things:

1. When performing matrix factorization, it is the loss function of RSVD that gets optimized, rather than that of KNN or kernel ridge;
2. Temporal dynamics are incorporated in the loss function. The binified time components are able to explain some variation in the sample dataset, so it is possible that sometimes a rating is dominated by the bias in time dimension.