Amazon_Food_Reviews

February 19, 2018

1 RECOMMENDATION / MATRIX COMPLETION ALGORITHM BASED ON AMAZON FOOD REVIEWS

The problem tackled here is how to help users select products which they may like and to make recommendation to stimulate sales and increase profits. The Amazon Fine Food Reviews dataset which consists of 568,454 food reviews is used for the building the recommendations. Amazon users left up to October 2012 are part of the dataset. Recommendation system is based on users rating prediction. The rating varies between 1 and 5 with 1 being the worst rating and 5 being the best. It is assumed that users tend to like the products that have a score of greater than 4 and the highest 5 scores product are considered as recommendation candidates. Collaborative filtering algorithm is implemented to predict the scores of each product for each user.

1.1 Importing required libraries

```
In [1]: import numpy as np # linear algebra
        import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
        import csv
        from sklearn.preprocessing import MinMaxScaler
        from sklearn.model_selection import train_test_split
        from scipy import optimize
        # from main import method0
```

1.2 Data Wrangling and cleaning

Here, Take the data in which the user and item appear more than 10 times in order to reduce the data size.

The data() function returns the total number of users and products, the user-item table and also the train & test data set.

```
In [2]: def data_clean(df, feature, m):
            count = df[feature].value_counts()
            df = df[df[feature].isin(count[count > m].index)]
            return df

        def data_clean_sum(df,features,m):
            fil = df.ProductId.value_counts()
            fil2 = df.UserId.value_counts()
```

```
    df['#Products'] = df.ProductId.apply(lambda x: fil[x])
    df['#Users'] = df.UserId.apply(lambda x: fil2[x])
    while ((df.ProductId.value_counts(ascending=True)[0]) < m
           or  (df.UserId.value_counts(ascending=True)[0] < m)):
        df = data_clean(df,features[0],m)
        df = data_clean(df,features[1],m)
        print(df.shape)
    return df
```

1.3 Importing data and formatting it

```
In [3]: def data():
            print('loading data...')
            df = pd.read_csv('./input/Reviews.csv')

             # Shape of data frame after loading
            print(df.shape)

            df['datetime'] = pd.to_datetime(df.Time, unit='s')
            raw_data = data_clean_sum(df, ['ProductId', 'UserId'], 10)

            # Shape of data frame after loading
            print(raw_data.shape)

            # find X,and y
            # It is like indexing
            raw_data['uid'] = pd.factorize(raw_data['UserId'])[0]
            raw_data['pid'] = pd.factorize(raw_data['ProductId'])[0]
            sc = MinMaxScaler()

            #reshape returns a numpy array with 1 column which is transformed to values b/w [0
            raw_data['time']=sc.fit_transform(raw_data['Time'].values.reshape(-1,1))
            raw_data['nuser']=sc.fit_transform(raw_data['#Users'].values.reshape(-1,1))
            raw_data['nproduct']=sc.fit_transform(raw_data['#Products'].values.reshape(-1,1))
            # Seperate the features into three groups
            X1 = raw_data.loc[:,['uid','pid']]
            X2 = raw_data.loc[:,['uid','pid','time']]
            X3 = raw_data.loc[:,['uid','pid','time','nuser','nproduct']]
            y = raw_data.Score

            # train_test split
            X1_train,X1_test,y_train,y_test = \
             train_test_split(X1,y,test_size=0.3,random_state=2017)
            X2_train,X2_test,y_train,y_test = \
            train_test_split(X2,y,test_size=0.3,random_state=2017)
            X3_train,X3_test,y_train,y_test = \
             train_test_split(X3,y,test_size=0.3,random_state=2017)
```

```
    train = np.array(X1_train.join(y_train))
    test = np.array(X1_test.join(y_test))
    # got the productId to pid index
    pid2PID = raw_data.ProductId.unique()

    data_mixed = X1.join(y)
    data_mixed['uid'] = data_mixed['uid'].astype(int)
    data_mixed['pid'] = data_mixed['pid'].astype(int)
    data_mixed['Score'] = data_mixed['Score'].astype(int)
    total_p = data_mixed['pid'].unique().shape[0]
    total_u = data_mixed['uid'].unique().shape[0]

    # make the user-item table
    table = np.zeros([total_u,total_p])
    z = np.array(data_mixed)
    for line in z:
        u,p,s = line
        if table[u][p] < s:
            table[u][p] = s #if some one score a single thing several times
    print('the table\'s shape is:' )
    print(table.shape)
    return z, total_u,total_p,pid2PID,train,test,table,raw_data, data_mixed
```

1.4 Calculating the Mean Squared Error

```
In [4]: from sklearn.metrics import mean_squared_error
        import matplotlib.pyplot as plt
        def calculate_mse(x):
            MSE1=[]
            MSE2=[]
            for line in train:
                u,p,s = line
                MSE1.append(s)
                MSE2.append(x[u,p])
            MSE_in_sample = mean_squared_error(MSE1,MSE2)
            MSE3=[]
            MSE4 = []
            for line in test:
                u,p,s = line
                MSE3.append(s)
                MSE4.append(x[u,p])
            MSE_out_sample = mean_squared_error(MSE3,MSE4)
            print('the in sample MSE = {} \nthe out sample MSE = {}'
                  .format(MSE_in_sample,MSE_out_sample))
            return MSE_in_sample,MSE_out_sample

        def draw_mse(method,maxIter):
            import time
```

```
    c = []
    d = []
    timetime = []
    for i in [1,2,5,7,10,20,50,70,100]:
        tic = time.time()
        data = method(factors=i,maxIter=maxIter)
        a,b = calculate_mse(data)
        c.append(a)
        d.append(b)
        toc = time.time()
        timetime.append(toc-tic)
    aa = [1, 2, 5, 7, 10, 20, 50, 70, 100]
    for i in range(len(timetime)):

        print('latent factors = {}, time = {}'.format(aa[i],timetime[i]))
    plt.figure()
    plt.plot(aa,c,label = 'in_sample_MSE')
    plt.plot(aa,d,label = 'out_sample_MSE')
    plt.xticks([1,2,5,7,10,20,50,70,100])
    plt.legend()
    plt.show()
    return 0
```

1.5 Plotting the confusion matrix

```
In [7]: import itertools
        import matplotlib.pyplot as plt
        from sklearn.metrics import confusion_matrix
        def plot_confusion_matrix(cm, classes,
                                  normalize=False,
                                  title='Confusion matrix',
                                  cmap=plt.cm.Blues):
            """
            This function prints and plots the confusion matrix.
            Normalization can be applied by setting `normalize=True`.
            """
            if normalize:
                cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
                print("Normalized confusion matrix")
            else:
                print('Confusion matrix, without normalization')

            print(cm)
            plt.imshow(cm, interpolation='nearest', cmap=cmap)
            plt.title(title)
            plt.colorbar()
            tick_marks = np.arange(len(classes))
            plt.xticks(tick_marks, classes, rotation=45)
```

```
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')



def drawcm(y_pred,y_test ,title=''):
    print('caculating cm..')
    y1=[]
    y2=[]
    for line in y_test:
        u,p,s = line
        y1.append(s)
        y2.append(y_pred[u,p])
    temp1 = []
    temp2 = []
    for i in range(len(y1)):
        if np.array(y1)[i] >= 4:
            temp1.append(1)
        elif np.array(y1)[i] <= 2:
            temp1.append(0)
        else:
            temp1.append(0)
        if y2[i] >= 4:
            temp2.append(1)
        elif y2[i] <= 2:
            temp2.append(0)
        else:
            temp2.append(0)
    cm = confusion_matrix(temp1,temp2)
    plt.figure()
    plot_confusion_matrix(cm, classes=['not','recommand'], normalize=True,
                          title=title)
    plt.show()
```

1.6 Calculating the cost and gradient functions for Collaborative Filtering Algorithm

If we're given the the product features we can use that to find out the users' preference parameters.

Given $x^{(1)}, ..., x^{(n_m)}$***, estimate*** $\theta^{(1)}, ..., \theta^{(n_u)}$***:***

$$\min_{\theta^{(1)},....,\theta^{(n_u)}} 1/2\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \lambda/2\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

If we're given the users' preferences parameters we can use them to work out the product features.

Given $\theta^{(1)}, ..., \theta^{(n_u)}$***, estimate*** $x^{(1)}, ..., x^{(n_m)}$***:***

$$\min_{x^{(1)},....,x^{(n_m)}} 1/2\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \lambda/2\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2$$

The loss function for the Collaborative Filtering can be defined as:

Minimizing $x^{(1)}, ..., x^{(n_m)}$ ***and*** $\theta^{(1)}, ..., \theta^{(n_u)}$ ***simultaneously:***

$$J(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}) = 1/2\sum_{(i,j):r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \lambda/2\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2 + \lambda/2\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

where we want to estimate the users' preferences parameters and product features such that the loss function is minimized.

$$\min_{x^{(1)},....,x^{(n_m)}\theta^{(1)},....,\theta^{(n_u)}} J(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)})$$

```
In [8]: def normalize (Y, A):
            m, n = Y.shape
            Y_mean = np.zeros((m, 1))
            Y_norm = np.zeros(Y.shape)
            for i in range(0,Y.shape[0]):
                idx = np.nonzero(A[i])
                Y_mean[i] = np.mean(Y[i, idx], axis = 1)
                Y_norm[i,idx] = Y[i,idx] - Y_mean[i]

            Ymean = np.nan_to_num(Y_mean)
            return Y_norm, Ymean

        def CostFunc(params, Y, A, num_users, num_prod, num_features, lamda):
            # Unfold the X and Theta matrices from params
            X = np.reshape(params[0:num_prod*num_features],(num_prod,num_features))
            Theta = np.reshape(params[num_prod*num_features:],(num_users,num_features))

            J = (1/2)*sum(sum(((X.dot(Theta.T) - Y) * A)**2)) + (lamda/2) * sum(sum(Theta**2))

            return J

        def GradFunc(params, Y, A, num_users, num_prod, num_features, lamda):

            # Unfold the X and Theta matrices from params
            X = np.reshape(params[0:num_prod*num_features],(num_prod,num_features))
            Theta = np.reshape(params[num_prod*num_features:],(num_users,num_features))


            # You need to return the following values correctly
            X_grad = np.zeros(X.shape)
            Theta_grad = np.zeros(Theta.shape)

            X_grad = ((X.dot(Theta.T) - Y) * A).dot(Theta) + lamda * X

            Theta_grad = ((X.dot(Theta.T) - Y) * A).T.dot(X) + lamda * Theta

            grad = np.concatenate((X_grad.flatten(),Theta_grad.flatten()), axis=0)
            return grad
```

1.7 Collaborative Filtering algorithm can be summarized as:

1) Initialize $x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ...\theta^{(n_u)}$ to small random values.
2) Minimize $J(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ...\theta^{(n_u)})$ using gradient descent (or an advanced optimization algorithm). E.g. for every: $j = 1, ..., n_u, i = 1, ..., n_m$:

$$x_k{}^{(i)} := x_k{}^{(i)} - \alpha\left(\sum_{j:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})\theta_k{}^{(j)} + \lambda x_k{}^{(i)}\right)$$
$$\theta_k{}^{(j)} := \theta_k{}^{(j)} - \alpha\left(\sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k{}^{(i)} + \lambda \theta_k{}^{(j)}\right)$$

3) For a user with parameters θ and a product with (learned) features x, predict a star rating of $\theta^T x$.

1.8 Defining system parameters and defining initial feature and parameter values for the Collaborative Filtering algorithm

```
In [9]: z, total_u,total_p,pid2PID,train,test,table,raw_data, data_mixed = data()
        A = ((table!=0) * 1).T
        Y = table.T
        num_features = 10; num_prod = A.shape[0]; num_users = A.shape[1];
        X = np.random.random((num_prod,num_features))
        Theta = np.random.random((num_users,num_features))
```

```
loading data...
(568454, 7)
(95552, 10)
(67756, 10)
(64771, 10)
(64340, 10)
(64340, 10)


C:\Users\saura\Anaconda3\lib\site-packages\sklearn\utils\validation.py:475: DataConversionWarn
  warnings.warn(msg, DataConversionWarning)
```

```
the table's shape is:
(3666, 1102)
```

1.9 Running the optimization algorithm and finding the optimal feature and parameter values for the model

```
In [10]: # Mean normalization
         Y_norm, Y_mean = normalize(Y, A)

         # Merging the X and Theta values into a 1-D array for input to optimization algo
         Inp = np.concatenate((X.flatten(),Theta.flatten()), axis=0)

         # Defining system parameter values
         lamda = 10
         args = (Y_norm, A, num_users, num_prod, num_features, lamda)  # arguments values

         # Conjugate gradient optimization algorithm
         res = optimize.fmin_cg(CostFunc, Inp, fprime=GradFunc, args=args, maxiter = 200)

         # Optimal Cost and Gradient values after optimization
         J = CostFunc(Inp, *args)
         grad = GradFunc(Inp, *args)

         # Unfold the returned theta back into P and U
         X = np.reshape(res[0:num_prod*num_features],(num_prod,num_features))
         Theta = np.reshape(res[num_prod*num_features:],(num_users,num_features))

         print('Recommender system learning completed.\n')
```

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 21026.954254
         Iterations: 200
         Function evaluations: 295
         Gradient evaluations: 295
Recommender system learning completed.
```

1.10 Generating recommendations

```
In [11]: ## ================== Generating recommendation ================
         #  After training the model, we can now make recommendations
         # by computing the prediction matrix.

         p = X.dot(Theta.T)
         prediction = np.around(p + Y_mean.reshape(-1,1),2)
```

```
k = 1

# adding an index column to ratings
t1 = np.vstack((np.arange(0,Y.shape[0]),prediction[:,k])).T
# Sorting in descending order by ratings
Rating_userK = t1[t1[:,1].argsort()[::-1]]

print('\nTop 10 recommended products (pid) for userid %d are:\n' % k)
pd.DataFrame(Rating_userK[0:10].astype(int), columns = ['pid', 'ratings'])
```

```
Top 10 recommended products (pid) for userid 1 are:
```

```
Out[11]:    pid  ratings
         0  768        5
         1  468        5
         2  154        5
         3  374        4
         4  905        4
         5  858        4
         6  837        4
         7  781        4
         8  323        4
         9  607        4
```

```
In [12]: Y[0:10]
```

```
Out[12]: array([[5., 5., 5., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.],
                ...,
                [0., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.]])
```

```
In [13]: prediction[0:10]
```

```
Out[13]: array([[4.89, 4.87, 4.86, ..., 3.95, 3.98, 3.95],
                [4.22, 4.27, 4.27, ..., 4.15, 4.1 , 4.15],
                [4.49, 4.49, 4.49, ..., 4.51, 4.5 , 4.51],
                ...,
                [4.19, 4.22, 4.22, ..., 4.2 , 4.15, 4.2 ],
                [4.46, 4.47, 4.46, ..., 4.48, 4.47, 4.48],
                [2.94, 2.97, 2.98, ..., 3.16, 3.13, 3.16]])
```