Learning to match: methods and challenges in the context of public policy applications

Teresa Morales, Juan Vila
University of Chicago graduate students
tmoralesgl@uchicago.edu and jiv@uchicago.edu

Abstract

This project analyzes the methodological challenges of learning to match models when applied to policy problems. First we discuss the difficulties of defining a matching score when there are different potential indicators of such result. Second, we show that matrix factorization can help address the fact that we do not observe most combinations of matches. Finally, we consider factorization machines as a way to incorporate the complexity of interactions between different features.

1 Introduction

Well known applications of machine learning, such as document retrieval, data integration or recommender systems can be categorized as "learning to match" models. A matching problem can be described as any situation where two objects from different spaces (query and target domains) have to be associated by a similarity or matching degree. Matching algorithms have recently been applied outside of the Computer Science and Engineering realm to different policy problems such as; hospital bed assignments, organ transplants, refugee allocation to different locations, active unemployment policies, among others.

This project will analyze the methodological challenges posed by this type of method and in particular, the following:

1. Definition of a "good match".
2. The fact that in most policy problems you do not observe diverse combinations of matches.
3. Complexity of feature interactions.

The structure of this project is as follows: first, we will summarize the characteristics of the data set that we used to explore different methods. Second, we will explain how we defined the matching score. Third, we will discuss the three models that we applied. Finally we will summarize our results and conclusions.

2 Data set

In order to explore different models for our matching problem we have used the "European Soccer Database". Manipulating the data set we obtain a structure where each observation is a contractual relation between a player and a team for one season (see data appendix). For our models we have:

1. A matrix concatenating both team's features (matrix Y)and player's features (matrix X).
2. A vector r including the ranking or matching score for each pair of team and player in the season.

3 Matching score

In recommender systems, the matching score is normally given by users through some kind of explicit or implicit rating (a score given to movies, for instance). When we want to apply these methods to public policy problems many times the specific score is not defined. For instance if you want to use these methods to try to help people find a good employment, you first need to define what a good match means. The quality of the relation between a company and a worker could be determined by many different factors, including both employer and employee's satisfaction, length of the contractual relationship, productivity of the worker, etc.

We think this poses an important methodological challenge since how you define the score will have an effect both on the ranking you assign to new observations and on the performance of your model. Additionally, choosing a definition only based on how well the model performs might not be the best option from a policy and/or ethical perspective.

We experiment using two different definitions of a "good match" between player and team:

1. Share of games won by the team when the player was playing over the total number of games played by the team.
2. Length of contract (number of years of each player with the same team). We normalize this value by subtracting the mean and dividing by the standard deviation.

4 Methods applied

Formally we want to minimize a loss function where the estimated score is a function of both team's and player's features: min $L(r, f(X, Y))$

A potential approach could be to apply a least squares type of loss pulling together both team's and player's features. However, this simple method doesn't address important challenges posed by this type of problem:

1. Sparse data:

   When trying to apply learning to match models to policy problems you rarely observe different combinations of matches. In this particular case, if you want to obtain a good predicting ranking score it would be desirable to observe how each player performs in different teams.

   The fact that you don't results in a matrix completion problem that a simple least squares regression does not address. We will show how to address it using matrix factorization.

2. Complex interactions:

   Similarly, the least squares method does not consider the complexity of interactions between different combinations of match.

   A potential way to address this could be to include different combinations of interaction terms in our simple linear regression problem. However, when dealing with huge data sets, this process might be challenging from a computational point of view. Additional it would not address sparsity of observations. This is why we explore Factorization Machines as a possible way to incorporate complex relations between features.

4.1 Standard linear regression

We perform the following problem using a regularization factor (ridge regression):

$$w = argmin \, \|r - \hat{r}\|_2^2 + \lambda \, \|w\|_2^2$$

$$\hat{r}_i = \sum_{i=1}^{m} w_i Z_i$$

where Z is the horizontal concatenation of the matrix of player's features (X) and the matrix of team's features (Y) and a bias (column of 1s) and m is the total number of features

4.2 Matrix Factorization

Matrix factorization allows to estimate what the score between different matches would be based on the score of observed matches. This means that instead of using the actual features to estimate the score we use the scores available to define a latent space of features that relates teams to players:

Let R be the matrix of scores where rows are p players and columns are t teams. This matrix, that has a lot of missing data can be represented as a product of two latent features matrices:

$$R \approx PT^T = \hat{R}$$

P is a p by k matrix where k are the "latent features" and each row represents the strength of the association of each player and those latent features.

T is a t by k matrix and each row represents the strength of the association of each team and those latent features.

The problem with the features that we do observe is that they belong to different spaces so you cannot directly compare them. When doing matrix factorization we obtain features vectors that are in the same space and can therefore be directly comparable, using a cosine similarity index.

We therefore now have the following estimated score:

$$\hat{r}_i j = p_i^T t_j = \sum_{k=1}^{k} p_{ik} t_{kj}$$

It is possible to include biases, including separate biases for team, player and global score.

Our error to minimize is:

$$e_{ij} = (r_{ij} - \sum_{k=1}^{k} p_{ik} t_{kj})^2 + \frac{\beta}{2} \|P\|^2 + \|T\|^2$$

It is possible to solve for matrices P and T, using gradient descent since the derivative of the above expression with respect to elements p_{ik} and t_{kj} can be easily calculated. In particular, given a step size τ we have that:

$$\begin{aligned} p_{ik}^{(1)} &= p_{ik}^{(0)} + \tau(2e_{ij}t_{kj} - \beta p_{ik}^{(0)}) \\ t_{kj}^{(1)} &= t_{kj}^{(0)} + \tau(2e_{ij}p_{ik} - \beta t_{kj}^{(0)}) \end{aligned}$$

We have used code by Albert Au Yeung `http://www.albertauyeung.com/post/python-matrix-factorization/` to run this model in our data set using training and testing samples and experimenting with different ranking scores and different hyper-parameters (see code appendix).

4.3 Factorization Machines

The downside of matrix factorization for a case like the one we are studying is that it does not take into account all the information provided by the features matrices. We consider instead Factorization Machines. The model that we will use is very similar to a regular linear model and easily applicable. However it considers a much more complex set of interactions between features and addresses sparsity of data.

The estimated score is the following:

$$\hat{r} = w_0 + \sum_{i=1}^{m} w_i z_i + \sum_{i=1}^{m} \sum_{j=i+1}^{m} < v_i, v_j > z_i z_j$$

The last term of the above expression represents the interactions between all features. This model allows to estimate the resulting weights matrix W, where element w_{ij} is the weight of the interaction between features i and j by factorizing it, so that:

$$W = VV^T$$

V has dimensions m by k and can be easily estimated using gradient descent. In particular it can be shown that the interactions term can be expressed as:

$$\sum_{i=1}^{m}\sum_{j=i+1}^{m} < v_i, v_j > Z_i Z_j = \frac{1}{2}\sum_{k=1}^{k}((\sum_{i=1}^{m} v_{ik} z_i)^2 - \sum_{i=1}^{m} v_{ik}^2 z_i^2)$$

The derivative of this expression with respect to v_{if} allows to estimate V as follows:

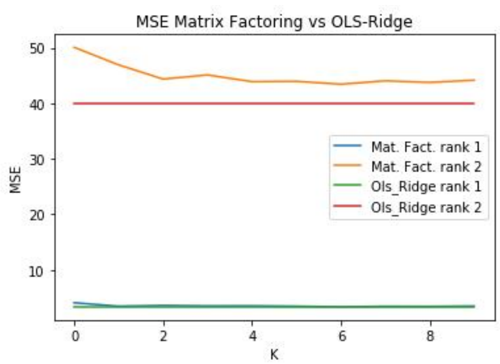
$$v_{ik}^{(1)} = v_{ik}^{(0)} + \tau(z_i \sum_{j=i}^{m} v_{jk} z_j - v_{ik}^{(0)} z_i^2)$$

According to Steffen Rendle(2010) this type of model is helpful when we are dealing with sparse settings since the data for one interaction helps to estimated the related interactions. He argues that k has to be large enough but that with sparse settings, such as the one we are dealing with, a smaller k is better.

In order to experiment the results of this type of model we have used fastFM and the code provided in its guide `https://ibayer.github.io/fastFM/guide.html` (see code appendix)

5 Results and conclusions

The first thing we observe is the fact that our two normalized ranking scores result in very different errors for our two first models. In particular the ranking score based on the length of the contractual relation (rank 2 in the graph below) seems to perform much worse than the one based on share of games won with the player (rank 1 in the graph below).



We show the comparison between the two models in a plot as a function of k where the OLS error is constant since no k is applied to it. We can see that, for the ranking that performs best (share of games won), OLS with ridge regression and matrix factorization result in an almost exact error, irrespective of the k used. For the ranking based on years, matrix factorization seems to perform worse than using all features in a linear regression.

As for the method of factorization machines, we find that the k that minimizes the error is the smallest one. With k=1 and 15,000 iterations[1], we obtain an error in the testing sample of 1.9 for the ranking based on games and 2.2 for the ranking based on years in the team. Both are therefore smaller than OLS regression. The improvement is particularly significant for the second definition of matching score that had an error of 40 with OLS regresssion.

We also find that increasing k results in a much higher error. When we use k = 2, we obtain an error of 12 for the ranking based on games and 13 for the ranking based on years. However, we still see that both rankings result in similar errors with this model.

This analysis suggests that using a more sophisticated model that includes features interactions might allow to reduce the impact of having different potential definitions of a matching score, at least in terms of error rates. Further analysis could be made to try to understand what is the impact of changing the matching score definitions on the resulting ranking.

[1] We observe that we need a large number of iterations to converge to smaller errors

References

[1] Bansak et al (2018). “Improving refugee integration through data-driven algorithmic assignment”. *Science.* Vol 359, Issue 6373.

[2] Steffen Rendle (2010) Factorization Machines.*ICDM ’10 Proceedings of the 2010 IEEE International Conference on Data Mining.* Pages 995-1000

[3] Tianqi Chen, Zhao Zheng, Qiuxia Lu, Weinan Zhang, Yong Yu (2011). “Feature-Based Matrix Factorization”

[4] Zhengdong Lu, Hang Li and Wei Wu (2013). "Learning Bilinear Model for Matching Queries and Documents”. *Journal of Machine Learning Research (JMLR) | Vol 14: pp. 2519-2548*

Data appendix

The data set "European Soccer Database” can be found in the following link:

`https://www.kaggle.com/hugomathien/soccer`

This data set includes:

1. A players’ features matrix: 42 features for 11,000 players in 10 seasons . It includes details such as preferred foot, heading accuracy, ball control, acceleration, etc.
2. A Teams’ features matrix: 25 features of 299 teams in 10 seasons. It includes features such as build up speed, chance creation passing, etc.
3. A matrix of games played: 26,000 matches x 115 columns. It includes the id of the home team, the away team, the id of each of the players that played the match and variables that summarize the game results.

We select a single season to simplify our analysis.

Code Appendix

```python
##### OPEN DATA
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import sqlite3
import matplotlib.pyplot as plt
import datetime
database = 'database.sqlite'

conn = sqlite3.connect(database)

tables = pd.read_sql("""SELECT *
                        FROM sqlite_master
                        WHERE type='table';""", conn)

match = pd.read_sql("""SELECT *
                        FROM Match;""", conn)
player_attributes = pd.read_sql("""SELECT *
                        FROM Player_Attributes;""", conn)

team_attributes = pd.read_sql("""SELECT *
                        FROM Team_Attributes;""", conn)

player = pd.read_sql("""SELECT *
                        FROM Player;""", conn)


conn = sqlite3.connect(database)

tables = pd.read_sql("""SELECT *
                        FROM sqlite_master
                        WHERE type='table';""", conn)
tables
#### Functions
home_cols = match.columns[55:66].tolist()
id_team = match.columns[7]
home_cols.append(id_team)
away_cols = match.columns[66:72].tolist()
id_team = match.columns[8]
away_cols.append(id_team)

def transform_unique(data,cols,t_id):
    filtered = data[cols].set_index(t_id)
    filtered = filtered.stack().reset_index().rename(columns={0:'player_id',t_id: 'team_id'})
    filtered = filtered.drop(columns = ['level_1'])
    filtered = filtered.groupby(['team_id','player_id']).size().reset_index().rename(columns={0:'count'})
    return filtered
transform_unique(match, home_cols,'home_team_api_id' )


def tranform_whole_date(data, home_cols, away_cols, t_away_id, t_home_id ):
    rv = pd.DataFrame(columns=['league_id','season','team_id','player_id','loose+tie','wins'])
    for i in data['league_id'].unique():
        for j in data['season'].unique():

            data_filtered_sl = data[(data['league_id'] == i ) &( data['season'] == j)]

```

```
            filter_wins_away = data_filtered_sl[(data_filtered_sl.
    away_team_goal > data_filtered_sl.home_team_goal)]
            filter_wins_home = data_filtered_sl[(data_filtered_sl.
    away_team_goal < data_filtered_sl.home_team_goal)]
            filter_loose_away = data_filtered_sl[(data_filtered_sl.
    away_team_goal <= data_filtered_sl.home_team_goal)]
            filter_loose_home = data_filtered_sl[(data_filtered_sl.
    away_team_goal >= data_filtered_sl.home_team_goal)]

            filtered_home_w = transform_unique(filter_wins_home,
                                home_cols,t_home_id ).rename
                                (columns={'count':'wins'})
            filtered_away_w = transform_unique(filter_wins_away,
                                away_cols,t_away_id ).rename
                                (columns={'count':'wins'})
            filtered_home_l = transform_unique(filter_loose_home,
    home_cols,t_home_id ).rename(columns={'count':'loose+tie'})
            filtered_away_l = transform_unique(filter_loose_away,
    away_cols,t_away_id ).rename(columns={'count':'loose+tie'})

            unique_team_player = pd.concat([filtered_away_w,
    filtered_away_l,filtered_home_w,filtered_home_l])
            unique_team_player = unique_team_player.
                            groupby(['team_id','player_id'])
                                .sum().reset_index()

            temp  =  unique_team_player
            temp['league_id'] = i
            temp['season'] = j
            rv=rv.append(temp)
    return rv

def get_number_matches(data,t_away_id, t_home_id):
    number_matches_away = data.groupby(['league_id','season',
                            t_away_id]).size().reset_index()
    .rename(columns={0:'total_played_games',t_away_id:'team_id'})
    number_matches_home = data.groupby(['league_id','season',
                            t_home_id]).size().reset_index()
    .rename(columns={0:'total_played_games',t_home_id:'team_id'})
    number_maches = pd.concat([number_matches_away,
                                number_matches_home])
    return number_maches.groupby(['league_id','season','team_id'])
    .sum().reset_index()

def to_season(data, t_id,player=False):
    if player:
        label = 'player_id'
    else:
        label = 'team_id'
    wk_data = data.copy()
    wk_data['date']=pd.to_datetime(wk_data['date'])
    wk_data['year'] = wk_data.date.dt.year*100 +
    wk_data.date.dt.month
    wk_data['season']=np.where( wk_data.year>201506,'2015/2016',None)
    wk_data['season']=np.where( (wk_data.year<=201506) &
    (wk_data.year>201406) ,'2014/2015',wk_data['season'])
    wk_data['season']=np.where( (wk_data.year<=201406) &
    (wk_data.year>201306) ,'2013/2014',wk_data['season'])
    wk_data['season']=np.where( (wk_data.year<=201306) &
    (wk_data.year>201206) ,'2012/2013',wk_data['season'])
    wk_data['season']=np.where( (wk_data.year<=201206) &
    (wk_data.year>201106) ,'2011/2012',wk_data['season'])
    wk_data['season']=np.where( (wk_data.year<=201106) &
    (wk_data.year>201006) ,'2010/2011',wk_data['season'])
    wk_data['season']=np.where( (wk_data.year<=201006) &
```

```
    (wk_data.year>200906) ,'2009/2010',wk_data['season'])
    wk_data['season']=np.where( (wk_data.year<=200906) &
    (wk_data.year>200806) ,'2008/2009',wk_data['season'])
    wk_data['season']=np.where( (wk_data.year<=200806) &
    (wk_data.year>200706) ,'2007/2008',wk_data['season'])
    wk_data['season']=np.where( (wk_data.year<=200706) &
    (wk_data.year>200606) ,'2006/2007',wk_data['season'])
    wk_data_season = wk_data.groupby([t_id,'season']).
                    mean().reset_index()
    wk_data_season = wk_data_season.drop(columns =
    ['id','year']).rename(columns={t_id : label})
    return wk_data_season


    import numpy as np
### citation: http://www.albertauyeung.com/post/python-matrix-
    factorization/
class MF():

    def __init__(self, R, K, alpha, beta, iterations):
        """
        Perform matrix factorization to predict empty
        entries in a matrix.

        Arguments
        - R (ndarray)   : user-item rating matrix
        - K (int)       : number of latent dimensions
        - alpha (float) : learning rate
        - beta (float)  : regularization parameter
        """

        self.R = R
        self.num_users, self.num_items = R.shape
        self.K = K
        self.alpha = alpha
        self.beta = beta
        self.iterations = iterations

    def train(self):
        # Initialize user and item latent feature matrice
        self.P = np.random.normal(scale=1./self.K, size=(self.
    num_users, self.K))
        self.Q = np.random.normal(scale=1./self.K, size=(self.
    num_items, self.K))

        # Initialize the biases
        self.b_u = np.zeros(self.num_users)
        self.b_i = np.zeros(self.num_items)
        self.b = np.mean(self.R[np.where(self.R != 0)])

        # Create a list of training samples
        self.samples = [
            (i, j, self.R[i, j])
            for i in range(self.num_users)
            for j in range(self.num_items)
            if self.R[i, j] > 0
        ]

        # Perform stochastic gradient descent for number of iterations
        training_process = []
        for i in range(self.iterations):
            np.random.shuffle(self.samples)
            self.sgd()
            mse = self.mse()
            training_process.append((i, mse))
```

```
            if (i+1) % 10 == 0:
                print("Iteration: %d ; error = %.4f" % (i+1, mse))

        return training_process

    def mse(self):
        """
        A function to compute the total mean square error
        """
        xs, ys = self.R.nonzero()
        predicted = self.full_matrix()
        error = 0
        for x, y in zip(xs, ys):
            error += pow(self.R[x, y] - predicted[x, y], 2)
        return np.sqrt(error)

    def sgd(self):
        """
        Perform stochastic graident descent
        """
        for i, j, r in self.samples:
            # Computer prediction and error
            prediction = self.get_rating(i, j)
            e = (r - prediction)

            # Update biases
            self.b_u[i] += self.alpha * (e - self.beta * self.b_u[i])
            self.b_i[j] += self.alpha * (e - self.beta * self.b_i[j])

            # Update user and item latent feature matrices
            self.P[i, :] += self.alpha * (e * self.Q[j, :] -
                                self.beta * self.P[i,:])
            self.Q[j, :] += self.alpha * (e * self.P[i, :] -
                                self.beta * self.Q[j,:])

    def get_rating(self, i, j):
        """
        Get the predicted rating of user i and item j
        """
        prediction = self.b + self.b_u[i] + self.b_i[j] +
        self.P[i, :].dot(self.Q[j, :].T)
        return prediction

    def full_matrix(self):
        """
        Computer the full matrix using the resultant biases, P and Q
        """
        return self.b + self.b_u[:,np.newaxis] + self.b_i[np.newaxis
    :,]
                + self.P.dot(self.Q.T)

###### Create Data set
transform_data = tranform_whole_date(match, home_cols,
                away_cols, 'away_team_api_id',
                'home_team_api_id' )
transform_data = transform_data.fillna(0)
number_matches = get_number_matches(match,'away_team_api_id',
                                    'home_team_api_id')
transform_data = transform_data.merge(number_matches)
transform_data['rank'] = transform_data.wins/(transform_data.
    total_played_games)

player_attributes_season = to_season(player_attributes,'player_api_id'
    ,True)
team_attributes_season = to_season(team_attributes,'team_api_id')
```

```

final_data = transform_data.merge(player_attributes_season)
final_data = final_data.merge(team_attributes_season)
#player.rename(columns = {'player_api_id':'player_id'})
final_data = final_data.merge(player)
##### Second Ranking

final_data2 = final_data[['player_id','team_id','rank']]
            .groupby(['player_id','team_id']).count().reset_index()
mean_team_att = team_attributes_season.groupby('team_id')
                .mean().reset_index()
mean_player_att = player_attributes_season.groupby('player_id').
                mean().reset_index()
final_data2  = final_data2.merge(mean_player_att, on='player_id')
final_data2  = final_data2.merge(mean_team_att, on='team_id')

final_data2 = final_data2.merge(player)

####### OLS RIDGE IMPLEMENTATION ##########
ef svd_ridge(y,x,lambda_r,cons=True):
    if cons:
        X = np.column_stack((x,np.ones([len(x),1])))
    else:
        X=x.copy()
    U,Sig,VT = np.linalg.svd(X)
    n,p = X.shape
    S_inv = np.zeros((p,n))
    S = np.zeros((n,p))
    for i in range(p):
        S_inv[i][i]= Sig[i]/(Sig[i]**2 + lambda_r)
    w = VT.T.dot(S_inv).dot(U.T).dot(y)
    return w


def cross_val(y,X,n):
    '''
    Description: This function create a cross validation for
    y set and X set, require that the number of
    cross validation divide in int numbers the
    amount of rows of the original
    data set. The cross validation is implemented
    thought a ridge regression with lambda = .5.
    This implementation assume that y
    is a continuous one.
    input:
    y: y variable to predict
    x: features
    n: number of cross val sets
    output:
    average error rate
    '''
    data = np.column_stack((y,X))
    np.random.shuffle(data)
    data_cv = np.split(data,n)
    rv =[]
    for i in range(n):
        temp=data_cv.copy()
        test_set = temp.pop(i)
        y_test, x_test =  test_set[:,[0]], test_set[:,1:]
        train_set = np.concatenate(temp,axis=0)
        y_train, x_train = train_set[:,[0]], train_set[:,1:]
        w = svd_ridge(y_train,x_train,.5)
        x_test = np.column_stack((x_test,np.ones([len(x_test),1])))
        y_hat = np.dot(x_test,w)
        error_sq =  (y_hat - y_test)**2
```

```
        rv.append(np.sqrt(np.sum(error_sq)))
    rv=np.array(rv)

    return rv.mean()
##### DATA PREP

x_cols= ['crossing', 'finishing', 'heading_accuracy',
       'short_passing', 'volleys', 'dribbling', 'curve', '
    free_kick_accuracy',
       'long_passing', 'ball_control', 'acceleration',
       'sprint_speed', 'agility', 'reactions',
       'balance', 'shot_power','jumping', 'stamina',
       'strength', 'long_shots', 'aggression',
       'interceptions', 'positioning', 'vision',
       'penalties', 'marking',
       'standing_tackle', 'sliding_tackle',
       'gk_diving', 'gk_handling', 'gk_kicking',
       'gk_positioning','gk_reflexes', 'buildUpPlaySpeed',
       'buildUpPlayDribbling', 'buildUpPlayPassing', '
    chanceCreationPassing',
       'chanceCreationCrossing',
       'chanceCreationShooting', 'defencePressure',
       'defenceAggression', 'defenceTeamWidth','height', 'weight','
    year_b']

data_r1 = final_data.copy()
data_r2 = final_data2.copy()
data_r1.dropna(inplace=True)
data_r2.dropna(inplace=True)
drop_indices = np.random.choice(data_r1.index, 2, replace=False)
data_r1 = data_r1.drop(drop_indices)
y_rank1=data_r1['rank'].values
y_rank2=((data_r2['rank']-data_r2['rank'].
        mean())/data_r2['rank'].std()).values
data_r1['birthday']=pd.to_datetime(data_r1['birthday'])
data_r2['birthday']=pd.to_datetime(data_r2['birthday'])
data_r1['year_b'] = data_r1.birthday.dt.year
data_r2['year_b'] = data_r2.birthday.dt.year
x_final_1 = data_r1[x_cols].values
x_final_2 = data_r2[x_cols].values

#### run models

mse_r1_ols_Ridge = cross_val(y_rank1,x_final_1,10)
mse_r2_ols_Ridge = cross_val(y_rank2,x_final_2,8)

####### MATRIX FACTORIZATION########
k=[1,2,3,4,5,6,7,8,9,10]

def matrix_fact_k(data,k_list,id_cols, season,n):
    rv_k = []
    ids = data_r1[['rank','player_id', 'team_id']]
    for k in k_list:
        randomize_indices = np.random.choice(data.index,
                            int(.1*len(data)),
                            replace=False)
        random_ids = ids.loc[randomize_indices]
        data_changed = data.copy()
        data_changed.at[randomize_indices, 'rank']=0
        data_trans = data_changed[(data_changed['season']==season)].
                    pivot(index='player_id',
                    columns='team_id',
                    values='rank')
        data_trans = data_trans.fillna(0)
        data_trans_array = np.array(data_trans)
```

```
        mf = MF(data_trans_array, K=k,
                alpha=0.1, beta=0.01, iterations=100)
        mf.train()
        result = mf.full_matrix()
        result = pd.DataFrame(result)
        result.columns = data_trans.columns
        result.index = data_trans.index
        trained_data = result.stack().reset_index().
                        rename(columns={0:'rank_hat'})
        trained_data = trained_data.merge(random_ids)
        trained_data['error_sq']=(trained_data['rank']-
                                trained_data['rank_hat'])**2
        rv_k.append(np.sqrt(trained_data['error_sq'].sum()))
    return rv_k

matrix_factfor_10 = matrix_fact_k(data_r1,k,
                    ['rank','player_id', 'team_id'],
                    '2015/2016',10)


def matrix_fact_k2(data,k_list,id_cols,n):
    '''
    Description: This function implement the
    matrix factoring for the second ranking.
    For a list of parameters K
    '''
    rv_k = []
    ids = data[['rank_rc','player_id', 'team_id']]
    for k in k_list:
        randomize_indices = np.random.choice
                        (data.index, int(.1*len(data)),
                        replace=False)
        random_ids = ids.loc[randomize_indices]
        data_changed = data.copy()
        data_changed.at[randomize_indices, 'rank_rc']=0
        data_trans = data_changed.pivot(
                        index='player_id',
                        columns='team_id',
                        values='rank_rc')
        data_trans = data_trans.fillna(0)
        data_trans_array = np.array(data_trans)
        mf = MF(data_trans_array, K=k, alpha=0.1,
        beta=0.01, iterations=100)
        mf.train()
        result = mf.full_matrix()
        result = pd.DataFrame(result)
        result.columns = data_trans.columns
        result.index = data_trans.index
        trained_data = result.stack().reset_index().
        rename(columns={0:'rank_hat'})
        trained_data = trained_data.merge(random_ids)
        trained_data['error_sq']=(trained_data['rank_rc']
        -trained_data['rank_hat'])**2
        rv_k.append(np.sqrt(trained_data['error_sq'].sum()))
    return rv_k
data_r2['rank_rc']=(data_r2['rank']-data_r2['rank'].mean())/
                    data_r2['rank'].std()

matrix_factfor_10_2 = matrix_fact_k2(data_r2,k,
                    ['rank_rc','player_id', 'team_id'],10)

##### GRAPH

mse_rank1=[mse_r1_ols_Ridge]*10
mse_rank2=[mse_r2_ols_Ridge]*10
```

```
plt.plot(matrix_factfor_10,label='Mat. Fact. rank 1')
plt.plot(matrix_factfor_10_2,label='Mat. Fact. rank 2')
plt.plot(mse_rank1,label='Ols_Ridge rank 1')
plt.plot(mse_rank2,label='Ols_Ridge rank 2')
plt.xlabel('K')
plt.ylabel('MSE')
plt.title('MSE Matrix Factoring vs OLS-Ridge')
plt.legend()

### Matrix factorization

from fastFM.datasets import make_user_item_regression
from fastFM import mcmc
from sklearn.metrics import mean_squared_error
import scipy.sparse as sp
from fastFM import als
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

X_train = sp.csc_matrix(X_train)
X_test = sp.csc_matrix(X_test)
    # add padding for features not in test
X_test = sp.hstack([X_test, sp.csc_matrix((X_test.shape[0], X_train.shape[1] -
X_test.shape[1]))])

n_iter = 15000
step_size = 1
l2_reg_w = 0
l2_reg_V = 0

fm = als.FMRegression(n_iter=0, l2_reg_w=0.1, l2_reg_V=0.1, rank=2)
# Allocates and initalizes the model parameter.
fm.fit(X_train, y_train)

rmse_train = []
rmse_test = []
r2_score_train = []
r2_score_test = []

for i in range(1, n_iter):
    fm.fit(X_train, y_train, n_more_iter=step_size)
    y_pred = fm.predict(X_test)

    rmse_train.append(np.sqrt(mean_squared_error(fm.predict(X_train), y_train)))
    rmse_test.append(np.sqrt(mean_squared_error(fm.predict(X_test), y_test)))

    r2_score_train.append(r2_score(fm.predict(X_train), y_train))
    r2_score_test.append(r2_score(fm.predict(X_test), y_test))

from matplotlib import pyplot as plt
fig, axes = plt.subplots(ncols=2, figsize=(15, 4))

x = np.arange(1, n_iter) * step_size
with plt.style.context('fivethirtyeight'):
    axes[0].plot(x, rmse_train, label='RMSE-train', color='r', ls="--")
    axes[0].plot(x, rmse_test, label='RMSE-test', color='r')
    axes[1].plot(x, r2_score_train, label='R^2-train', color='b', ls="--")
    axes[1].plot(x, r2_score_test, label='R^2-test', color='b')
axes[0].set_ylabel('RMSE', color='r')
axes[1].set_ylabel('R^2', color='b')
```

```
axes[0].legend()
axes[1].legend()
```