Lab Course Machine Learning Kalaiselvan Panneerselvam

Exercise 8 278920

Exercise 1: Analysis of Datasets

1 Wine Dataset

```
winered_data = pd.read_csv('winequality-red.csv', delimiter = ';')
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	0.307692	0.186275	0.216867	0.308282	0.106825	0.149826	0.373550	0.267785	0.254545	0.267442	0.129032	6
1	0.240385	0.215686	0.204819	0.015337	0.118694	0.041812	0.285383	0.132832	0.527273	0.313953	0.241935	6
2	0.413462	0.196078	0.240964	0.096626	0.121662	0.097561	0.204176	0.154039	0.490909	0.255814	0.338710	6
3	0.326923	0.147059	0.192771	0.121166	0.145401	0.156794	0.410673	0.163678	0.427273	0.209302	0.306452	6
4	0.326923	0.147059	0.192771	0.121166	0.145401	0.156794	0.410673	0.163678	0.427273	0.209302	0.306452	6

Size and columns

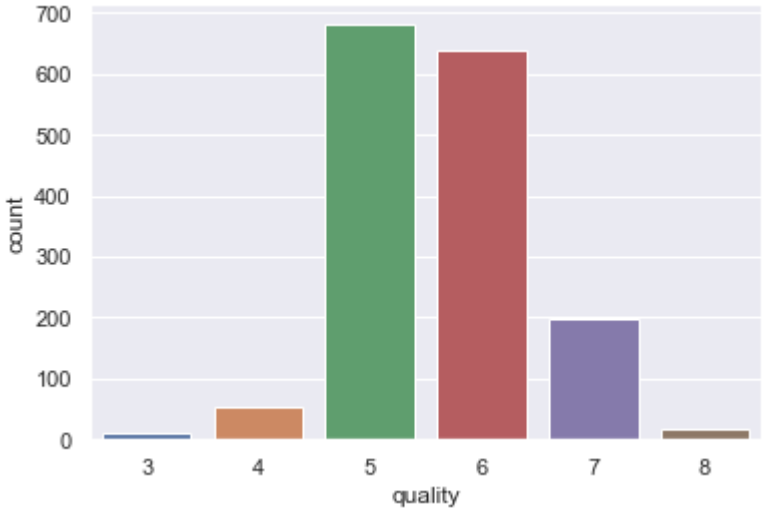
```
print("Shape of Red Wine dataset: {s}".format(s=wineData.shape))
print("Column headers/names: {s}".format(s = list(wineData)))
```

```
Shape of Red Wine dataset: (1599, 12)
Column headers/names: ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides', 'free sulfur dioxid
e', 'total sulfur dioxide', 'density', 'pH', 'sulphates', 'alcohol', 'quality']
```

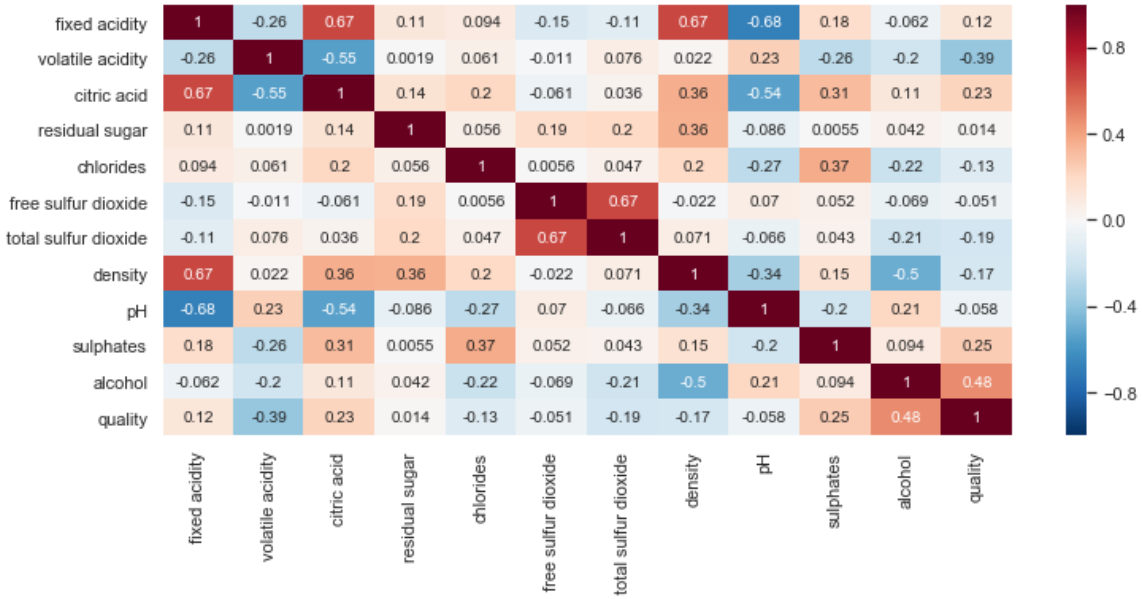
```
wineData.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
fixed acidity           1599 non-null float64
volatile acidity        1599 non-null float64
citric acid             1599 non-null float64
residual sugar          1599 non-null float64
chlorides               1599 non-null float64
free sulfur dioxide     1599 non-null float64
total sulfur dioxide    1599 non-null float64
density                 1599 non-null float64
pH                      1599 non-null float64
sulphates               1599 non-null float64
alcohol                 1599 non-null float64
quality                 1599 non-null int64
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

Plot for Quality Count



Correlation for HeatMap Plot

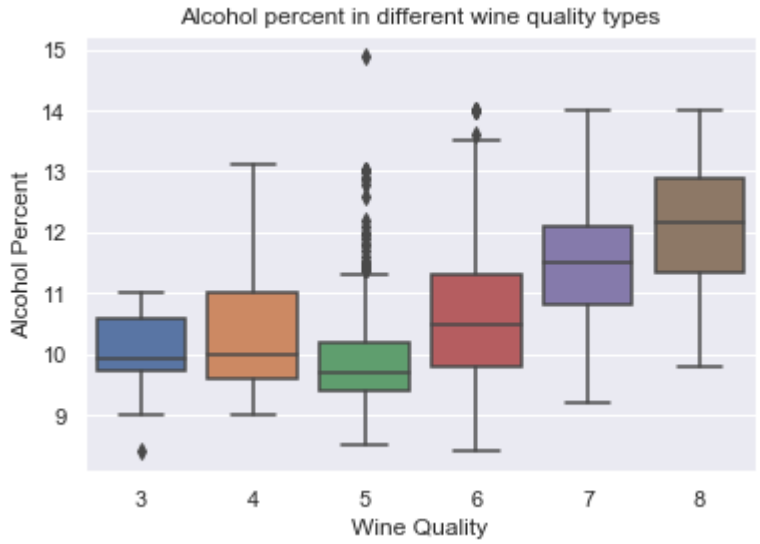


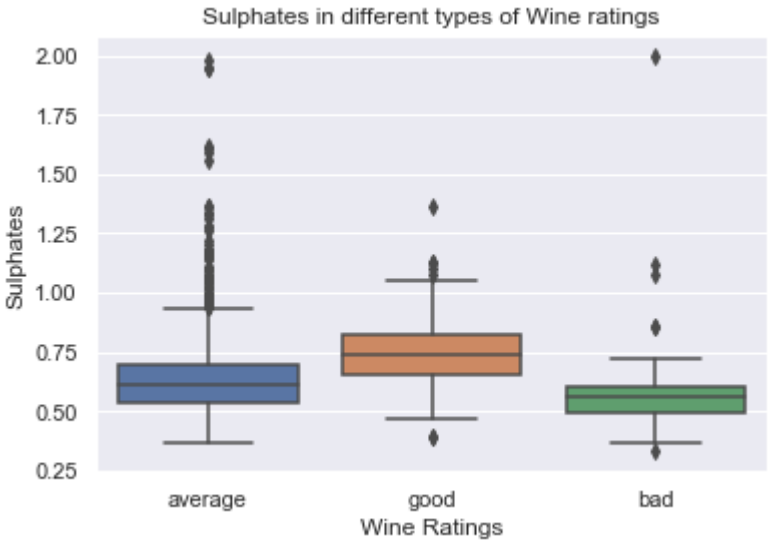

```
correlation['quality'].sort_values(ascending=False)
```

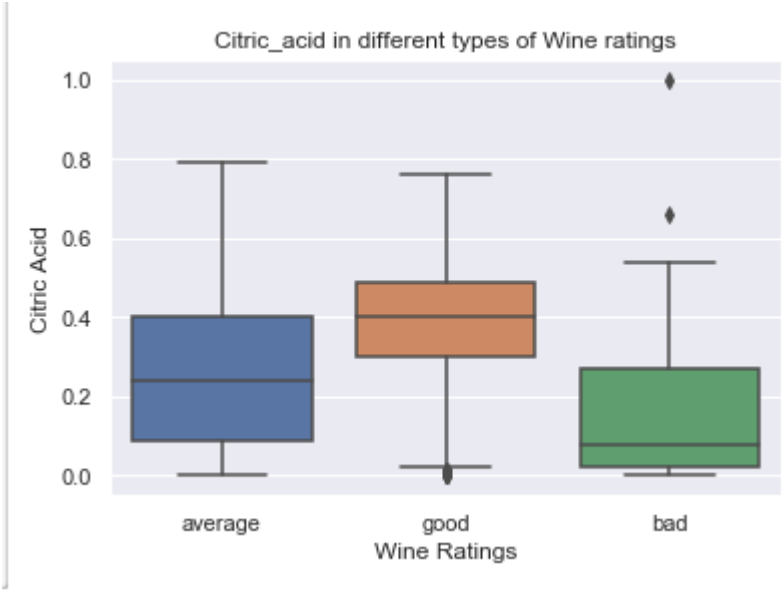
```
quality                1.000000
alcohol                0.476166
sulphates              0.251397
citric acid            0.226373
fixed acidity          0.124052
residual sugar         0.013732
free sulfur dioxide   -0.050656
pH                    -0.057731
chlorides             -0.128907
density               -0.174919
total sulfur dioxide  -0.185100
volatile acidity      -0.390558
Name: quality, dtype: float64
```

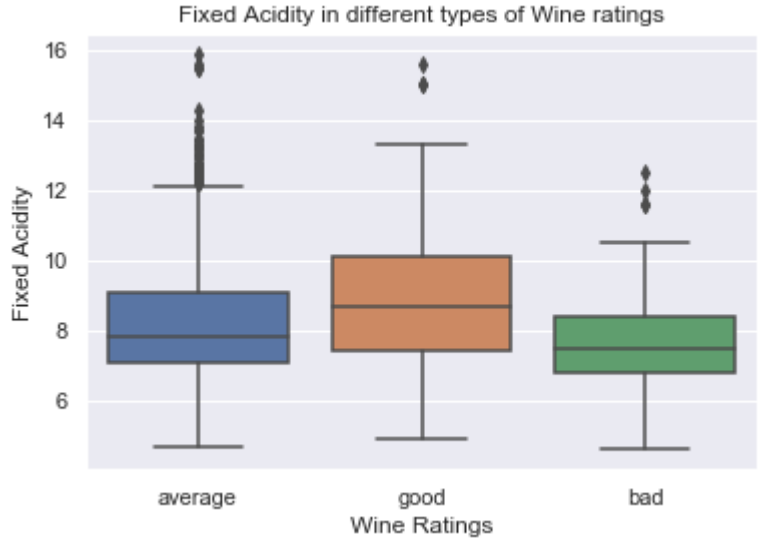
It can be observed that, the 'alcohol, sulphates, citric_acid & fixed_acidity' have maximum corelation with response variable 'quality'.

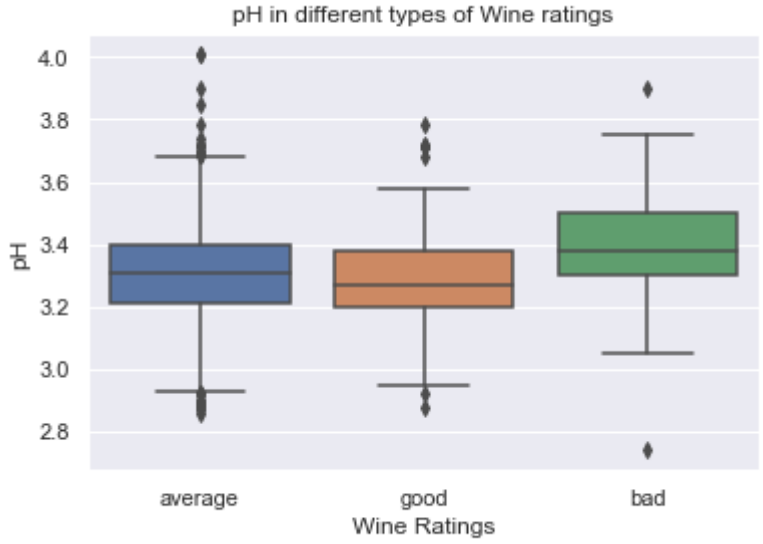
This means that, they need to be further analysed for detailed pattern and corelation exploration. Hence, only these 4 variables are analysed in detail.






Citric_acid in different types of Wine ratings
Citric Acid
1.0
0.8
0.6
0.4
0.2
0.0
average
good
bad
Wine Ratings


Fixed Acidity in different types of Wine ratings
Fixed Acidity
16
14
12
10
8
6
average
good
bad
Wine Ratings


pH in different types of Wine ratings
pH
4.0
3.8
3.6
3.4
3.2
3.0
2.8
average
good
bad
Wine Ratings

2 MovieLens 100k dataset

```
df.head()
```

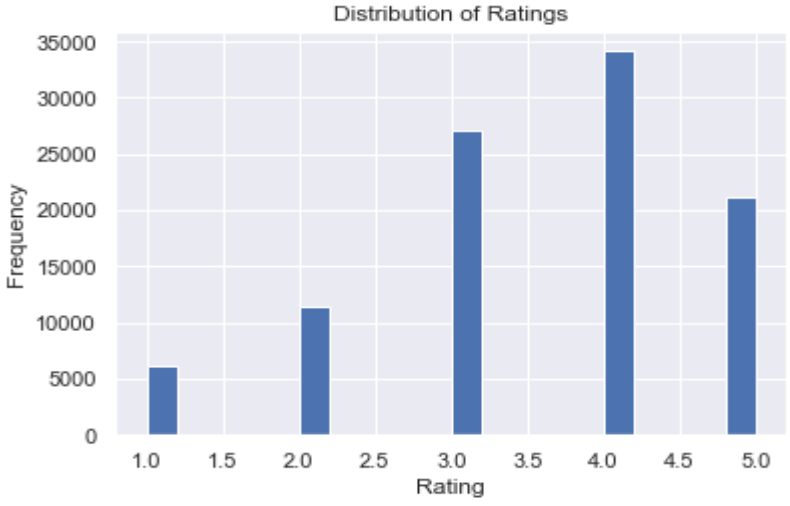
	user_id	item_id	rating	timestamp	year	month
214	259	255	4	1997-09-20 05:05:10	1997	9
83965	259	286	4	1997-09-20 05:05:27	1997	9
43027	259	298	4	1997-09-20 05:05:54	1997	9
21396	259	185	4	1997-09-20 05:06:21	1997	9
82655	259	173	4	1997-09-20 05:07:23	1997	9

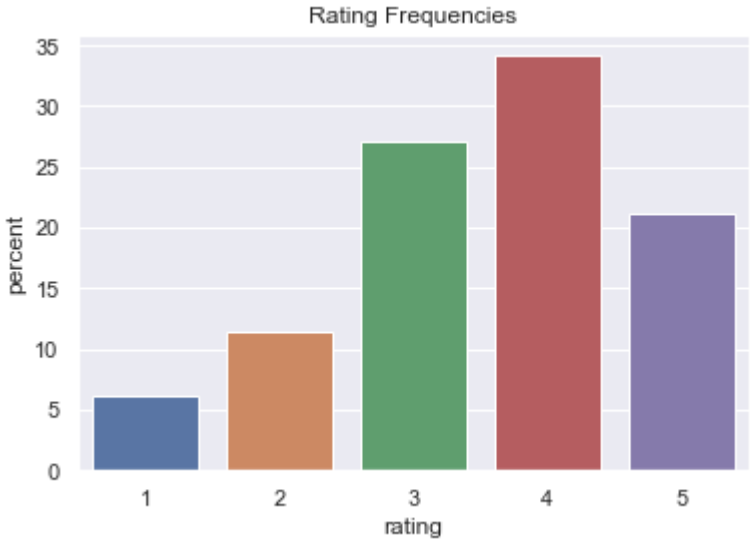
```
df.describe()
```

	user_id	item_id	rating	year	month
count	100000.00000	100000.000000	100000.000000	100000.00000	100000.000000
mean	462.48475	425.530130	3.529860	1997.47116	6.823180
std	266.61442	330.798356	1.125674	0.49917	4.316246
min	1.00000	1.000000	1.000000	1997.00000	1.000000
25%	254.00000	175.000000	3.000000	1997.00000	2.000000
50%	447.00000	322.000000	4.000000	1997.00000	9.000000
75%	682.00000	631.000000	4.000000	1998.00000	11.000000
max	943.00000	1682.000000	5.000000	1998.00000	12.000000

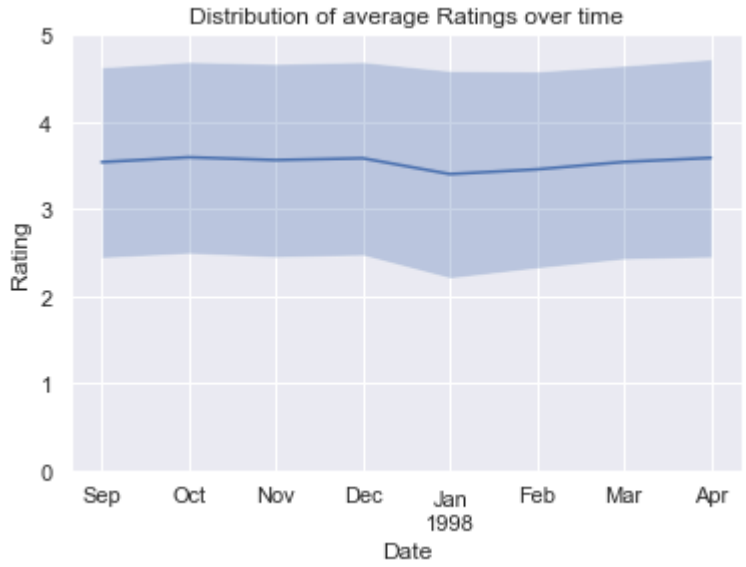
Distribution of Ratings:

Plot counts of each rating with respect to frequency and percentage.



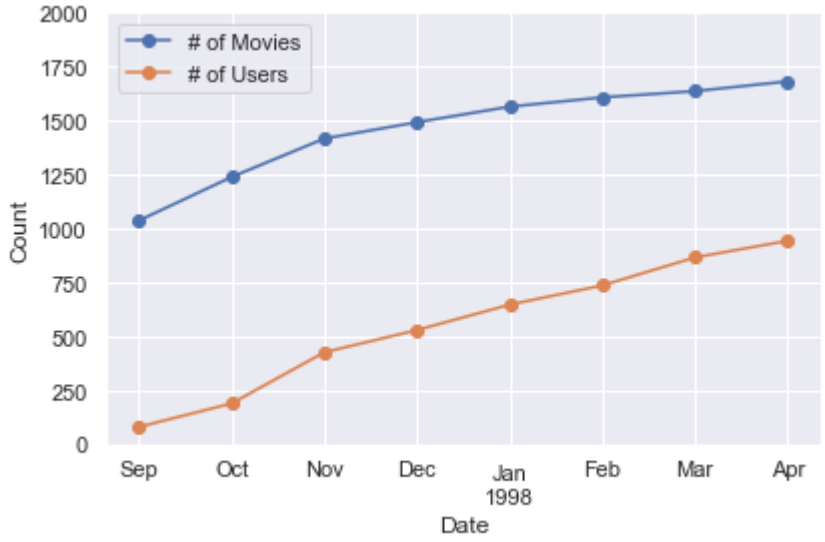


How consistent are the average ratings over time?



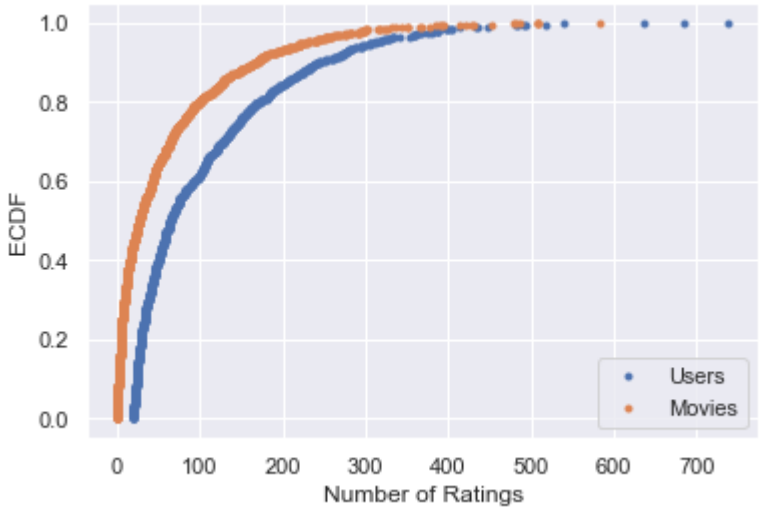
This average ratings were fairly consistent around 3.5. The lack of large changes over time will simplify modeling a little bit.

No_of_Movies and No_of_Users over time



New users seen in the dataset look fairly linear over time, although the number of movies start out withover 1000 already in the system.

Distribution of ratings per User



In the plot above, we can learn, for example, that 40% of all users rated 50 or less movies, and 90% of movies have 169 or less ratings. In general, we seen that a large fraction of movies and users have few ratings associated with them, but a few movies and users have many more ratings.

The main thing to take from this though is that the matrix of possible ratings is quite sparse, and that we need to use models that deal with this lack of data.

Exercise 2: Implement basic matrix factorization (MF) technique for recommender systems

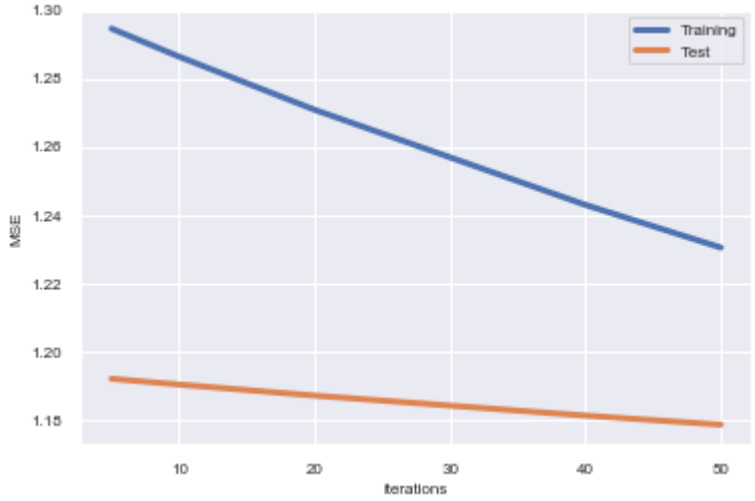
```
class mf():
    def __init__(self,R,K=40,item_fact_reg=0.0,user_fact_reg=0.0,item_bias_reg=0.0,user_bias_reg=0.0,verbose=False):
        self.R = R
        self.n_users, self.n_items = R.shape
        self.K = K
        self.item_fact_reg = item_fact_reg
        self.user_fact_reg = user_fact_reg
        self.item_bias_reg = item_bias_reg
        self.user_bias_reg = user_bias_reg
        self.sample_row, self.sample_col = self.R.nonzero()
        self.n_samples = len(self.sample_row)
        self._v = verbose
    def sgd(self):
        for idx in self.training_indices:
            u = self.sample_row[idx]
            i = self.sample_col[idx]
            prediction = self.predict(u, i)
            e = (self.R[u,i] - prediction)
            self.user_bias[u] += self.learning_rate * (e - self.user_bias_reg * self.user_bias[u])
            self.item_bias[i] += self.learning_rate * (e - self.item_bias_reg * self.item_bias[i])
            self.user_vecs[u, :] += self.learning_rate * (e * self.item_vecs[i, :] - self.user_fact_reg * self.user_vecs[u,:])
            self.item_vecs[i, :] += self.learning_rate * (e * self.user_vecs[u, :] - self.item_fact_reg * self.item_vecs[i,:])
    def train(self, n_iter=10, learning_rate=0.1):
        self.user_vecs = np.random.normal(scale=1./self.K,size=(self.n_users, self.K))
        self.item_vecs = np.random.normal(scale=1./self.K,size=(self.n_items, self.K))
        self.learning_rate = learning_rate
        self.user_bias = np.zeros(self.n_users)
        self.item_bias = np.zeros(self.n_items)
        self.global_bias = np.mean(self.R[np.where(self.R != 0)])
        self.partial_train(n_iter)
    def predict(self, u, i):
        prediction = self.global_bias + self.user_bias[u] + self.item_bias[i]
        prediction += self.user_vecs[u, :].dot(self.item_vecs[i, :].T)
        return prediction
    def partial_train(self, n_iter):
        ctr = 1
        while ctr <= n_iter:
            if ctr % 10 == 0 and self._v:
                print('\tcurrent iteration: {}'.format(ctr))
            self.training_indices = np.arange(self.n_samples)
            np.random.shuffle(self.training_indices)
            self.sgd()
            ctr += 1
    def model_fit(self, iter_array, test, learning_rate=0.1):
        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.R)]
            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
    def predict_all(self):
        predictions = np.zeros((self.user_vecs.shape[0],self.item_vecs.shape[0]))
        for u in range(self.user_vecs.shape[0]):
            for i in range(self.item_vecs.shape[0]):
                predictions[u, i] = self.predict(u, i)
```

optimize the hyper-parameters i.e. λ regularization constant, α learning rate, k latent dimensions.

```
k = [5, 20, 40]
lamda = [0.001, 0.000001, 0.01]
alpha = [0.00001, 0.0001, 0.001]
lamda.sort()
alpha.sort()
best_params = {}
best_params['k'] = k[0]
best_params['reg'] = lamda[0]
best_params['rate'] = alpha[0]
best_params['train_mse'] = np.inf
best_params['test_mse'] = np.inf
best_params['model'] = None
for fact in k:
    for reg in lamda:
        for rate in alpha:
            print('\nK: {}'.format(fact))
            print('Lamda: {}'.format(reg))
            print('Alpha: {}'.format(rate))
            ##3-Fold Cross Validation
            for i in range(0, len(folds)):
```

Compute the test RMSE (averaged across the 3-folds).

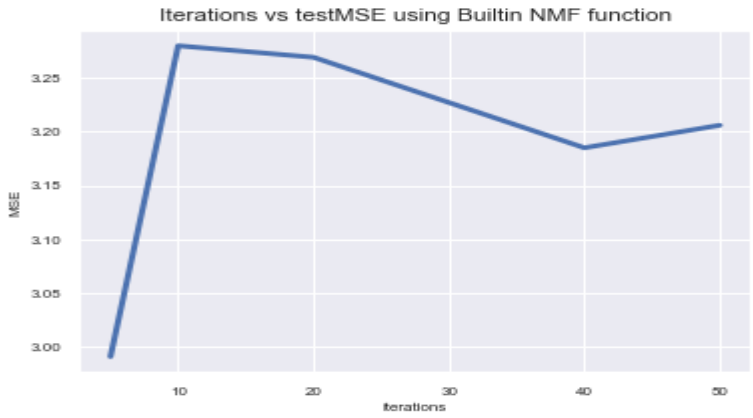
Plot of Iterations vs Train and Test MSE for the Best model using Matrix Factorisation



Exercise 3: Recommender Systems using matrix factorization libmf /sckit-learn

```
model_NMF = NMF(n_components=fact,alpha=reg,l1_ratio=0.01,verbose=True,max_iter =10, solver = 'cd')
w=model_NMF.fit_transform(train_data)
h=model_NMF.components_
pred=np.dot(w,h)
mselist.append(get_rmse(pred,test_data))
```

Plot of Iterations vs Test MSE for the Best model using built in NMF Function

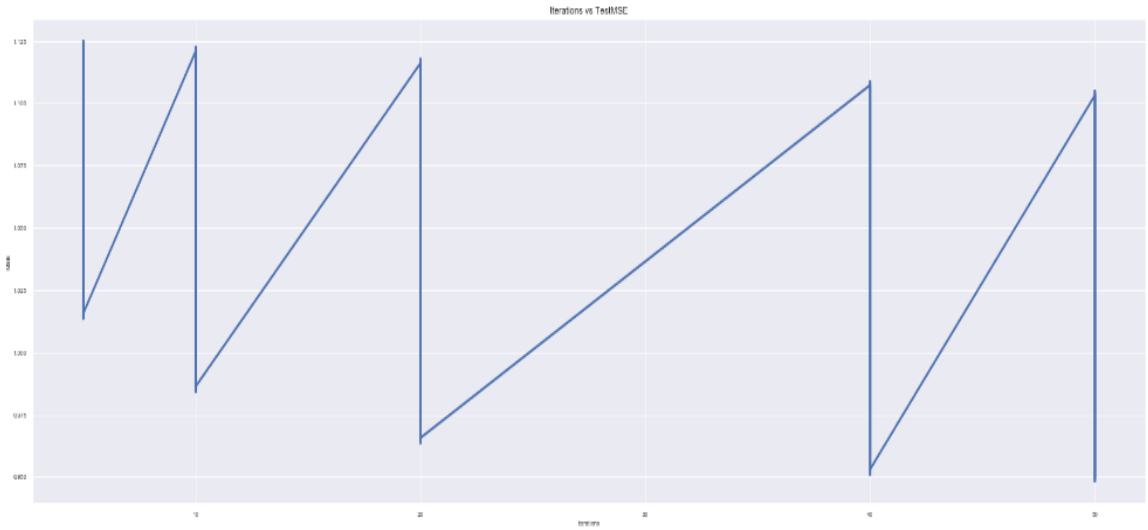


Surprise package using SVD

```
param_grid1 = {'n_factors': [5, 20, 40], 'n_epochs': [5,10,20,40,50], 'lr_all': [0.00001, 0.0001, 0.001],
               'reg_all': [0.001, 0.000001, 0.01]}
```

```
gs2 = GridSearchCV(SVD, param_grid1, measures=['rmse'], cv=3)
gs2.fit(data)
algo = gs2.best_estimator['rmse']
print(gs2.best_score['rmse'])
print(gs2.best_params['rmse'])
cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=3, verbose=True)
```

Plot of Iterations vs Test MSE for the Best model using built in SVD Function



The main difference that I noticed between recommender systems using Matrix factorisation class and built in functions is that the builtin functions like NMF, SVD by surprise are faster in computations than the matrix factorisation class.