DATA SCIENCE WITH PYTHON: EXPLORATORY ANALYSIS OF MOVIE RECOMMENDATION

BING MIU , RONG ZHANG , MEILAN CHEN , AND JINCHAO FENG

Abstract.

In this project, we explore the Netflix dataset with two linear models (with/without regularization) and two non-parametric models (Random Forest and Matrix Factorization) to predict the movie rating and build accurate movie recommendation system. Comparing the results based on Mean Square Error (MSE), we find that Matrix Factorization has the best performance.

1. Introduction.

Movie recommendation systems offer users with personalized suggestions for movie choice based on their preferences. It is extensively used by Netflix to suggest movies to the users and to provide users with information to help them decide which movie to watch [2]. In 2006, Netflix released over 100 million customer generated movie ratings on nearly 18,000 movies as part of the Netflix Prize competition [1]. Contestants were rated based on their algorithm's error in the predictions of users' ratings of movies [3]. We analyze the dataset in Section 2.1, and build the two linear models in Section 2.2; then we attempt Random Forest and Matrix Factorization methods on Section 2.3; finally, we include the compared results in Section 3 and summary on Section 4.

2. Methods.

To explore the best model for future movie rating prediction, we try not only linear regression, ridge and lasso regression but also non-parametric models including random forest and matrix factorization. We split the dataset into train set and test set with size 0.2. And we choose the best model based on the mean square error (MSE) of the validation set.

2.1. Data Analysis.

The Netflix dataset contains 31620 observations and 10 variables. The response variable (rating) is on an integral scale from 1 to 5. The level of 1, 2, 3, 4 and 5 each received a total of 1847, 3350, 8158, 11300 and 6965 ratings respectively. The other key variables include userID, movieID, age group, gender, year of the movie and genres of the movie. There are 1465 movieID for movies from the year 1926-2000 and 2353 userID for users across five age groups. Females contributed to 23.58% of the overall ratings. There are missing data in the genres, but after we transformed data into dummy variable, we solve the missing data issue, and as a result, we have 25 variables.

The correlation plot (FIG.1) shows a relatively week to moderate correlation between each of the variable. The highest we see is the positive correlation between musical and comedy.

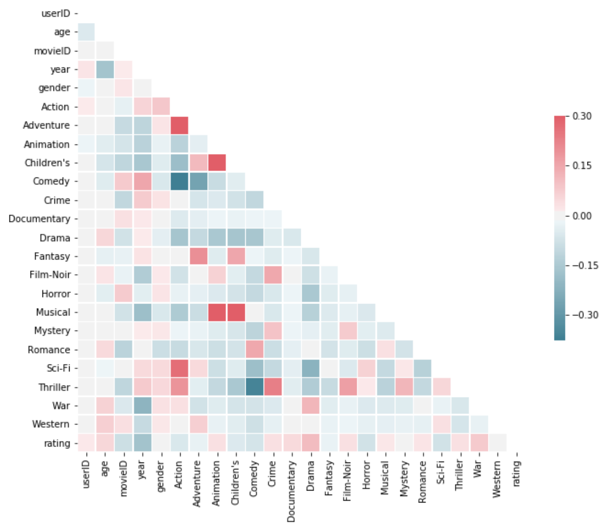

FIG. 1. *Correlation Map*

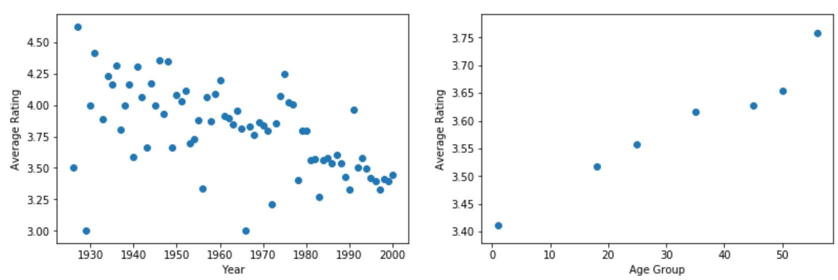

FIG. 2. *Left:Scatterplot Average Rating vs Year; Right: Scatterplot Average Rating vs Age Group;*

The average rating group by year plot (FIG.2) shows a negative linear relationship between rating and the year of the movie. The average rating by age group plot (FIG. 2) shows a slightly positive linear relationship between average rating and age group.

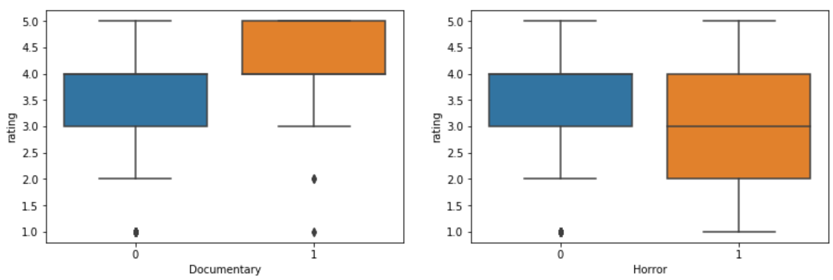

FIG. 3. *Left: Boxplot of Documentary Movie Rating; Right: Boxplot of Horror Movie Rating*

Movies in documentary genera have higher average rating than movies that are not in documentary genera. Movies with genera listed as horror have average ratings lower than movies not in the horror genera.

2.2. Original Linear Regression.

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i \tag{2.1}$$

Linear regression models the relationship between outcomes and predictors by fitting a linear equation to the observed data. The Linear regression formula is:

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \cdots + \beta_{p-1} X_{i,p-1} + \epsilon_i \tag{2.2}$$

The intercept β_0 is the constant term. The model parameters β_1 to β_{p-1} are unknown constants relating the p-1 explanatory variables to the variables of interest. The ϵ_i are normally distributed, independent random error components [6, 10].

The advantages of linear regression models are easy to implement and straightforward interpretations. However, multiple linear regression analysis makes several fundamental assumptions: linear relationship between the outcome and independent variables, uncorrelated errors normally distributed with mean zero and variance σ^2, constant variance, and no or little multicollinearity [10].

Stepwise selection methods are widely utilized to identify predictors for inclusion in regression models [8]. We attempt the backward stepwise selection, which involves setting the significance level at 0.05, starting with a model with all predictors, and remove the variable with the highest non-significant p-value for the coefficient. We repeat this process until no further variables can be deleted.

After five rounds of selection, predictors for musical (p-value 0.950), mystery(p-value 0.864), gender(p-value 0.437), adventure(p-value 0.418) and age(p-value 0.166) are removed from the model sequentially. The final model with MSE of 1.1502 is as follow:

$$\hat{Y}_{rating} = 29.815 - 0.013x_{year} - 0.143x_{Act} + 0.467x_{Ani} - 0.467x_{Child} - 0.087x_{Comedy}$$
$$+0.175x_{Crime} + 0.687x_{Doc} + 0.202x_{Dra} + 0.090x_{Fan} - 0.138x_{Film} - 0.377x_{Hor}$$
$$+0.088x_{Rom} - 0.080x_{Sci} + 0.167x_{Thr} + 0.145x_{War} - 0.108x_{West}$$

However, variable screening based on statistical significance and stepwise variable selection may lead to unreliable models with biased regression coefficients that need shrinkage[5, 7]. In the next section, we apply Ridge and Lasso regression to address this issue.

2.3. Ridge and Lasso Regression.

We apply Ridge and Lasso regularization techniques to the Ordinary Linear Squares(OLS), which shrink the coefficients of the linear model and prevent overfitting of the model[10]. Ridge regression adds the squared magnitude of coefficients as penalty terms to the function and minimizes the function to find the parameters:

$$min(\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 + \lambda \sum_{j=1}^{p} \beta^2) \tag{2.3}$$

The parameter λ controls how much we want to shrink the coefficients. When $\lambda = 0$, we restore the equation to OLS. When λ increases, we add more weight to the penalty term and shrink the $\beta's$ to a more considerable extent. Lasso regression adds absolute value of magnitude of coefficients as penalty terms and shrinks the less essential features' coefficients to zero and thus useful in removing some features when we have a large number of features [10].

$$min(\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 + \alpha \sum_{j=1}^{p} \mid \beta \mid) \tag{2.4}$$

The following graphs show how the coefficients of the predictors change with λ in Ridge and α in Lasso regression for our data, with Ridge on the left and Lasso on the right:

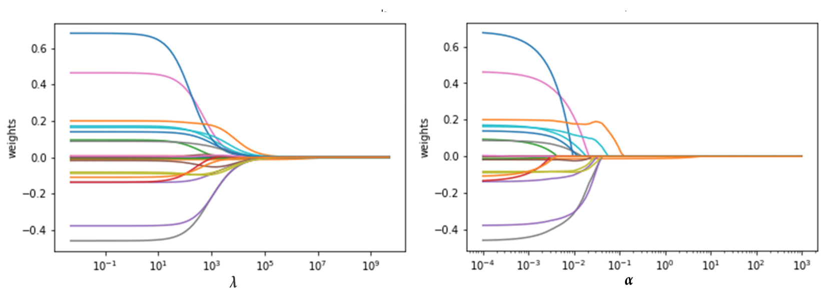

Fig. 4. *Left:Regularization of Ridge Regression;Right:Regularization of Lasso Regression*

Using *RidgeCV* and *LassoCV* in scikit-learn in python, we find the optimal parameters to be $\lambda = 9.7$ and $\alpha = 0.0003$, which give the smallest MSE of 1.1500 and 1.1499 respectively.

As $\lambda = 9.7$, the Ridge regression reduces the MSE in OLS by a number smaller than 0.0001 and shrinks the coefficients of the variables on a small scale as shown in *Figure 4*. As $\alpha = 0.0003$, Lasso regression shrinks the coefficients of the variables of movie genres of Musical and Mystery to zero and reduces the MSE in OLS by 0.001. Since Lasso regression performs better than Ridge regression, we display the final result of Lasso regression:

$$\begin{aligned}\hat{Y}_{rating} = {} & 29.560 + 0.0008x_{age} - 0.013x_{year} - 0.011x_{gender} - 0.135x_{Act} - 0.018x_{adv} \\ & +0.446x_{Ani} - 0.449x_{Child} - 0.088x_{Comedy} + 0.165x_{Crime} + 0.639x_{Doc} \\ & +0.199x_{Dra} + 0.082x_{Fan} - 0.110x_{Film} - 0.372x_{Hor} + 0.084x_{Rom} \\ & -0.082x_{Sci} + 0.160x_{Thr} + 0.133x_{War} - 0.093x_{West}\end{aligned}$$

2.4. Random Forest.

A random forest is an ensemble of trees. Each tree consists of split nodes and leaf nodes. The basic idea is that we bootstrap the data for each iteration first, then build a tree on those bootstrap samples. For each split in the tree, we randomly select a subset of features and use the best predictors to split. In other words, we only consider a subset of the variables for each potential split, and this makes it possible to build a diverse set of possible trees. After growing a large number of trees, we average those trees to get the prediction for a new outcome.

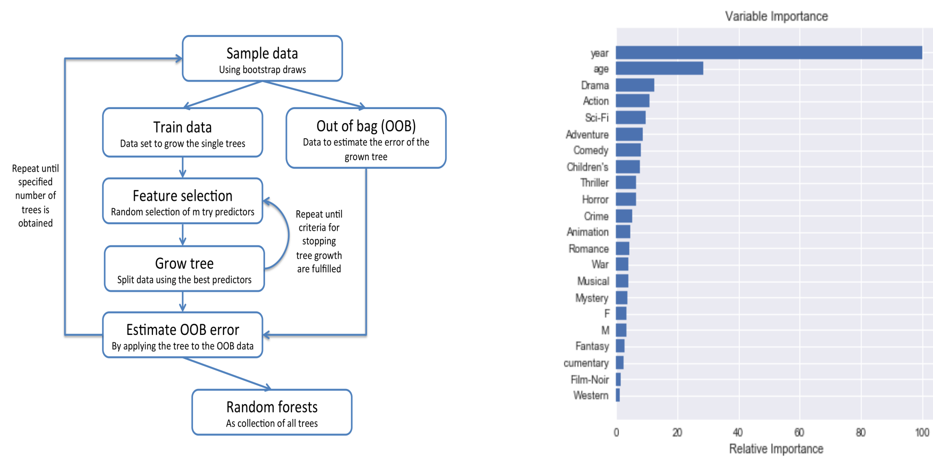

FIG. 5. *Left: The process of building random forest; Right: Diagram of variable importance*

This approach is one of the most widely used and highly accurate methods. However, this approach can be hard to interpret since we build a large number of trees that are averaged together to represent bootstrap samples with bootstrap nodes. It is also computational expansive and can lead to other issues such as overfitting.

To optimize the model performance in the random forest, we apply hyperparameter tunning. By using Randomized Search Cross Validation and Grid Search with Cross Validation, we finalize the optimization hyperparameter and reduce the

MSE from 1.08 to 0.98.

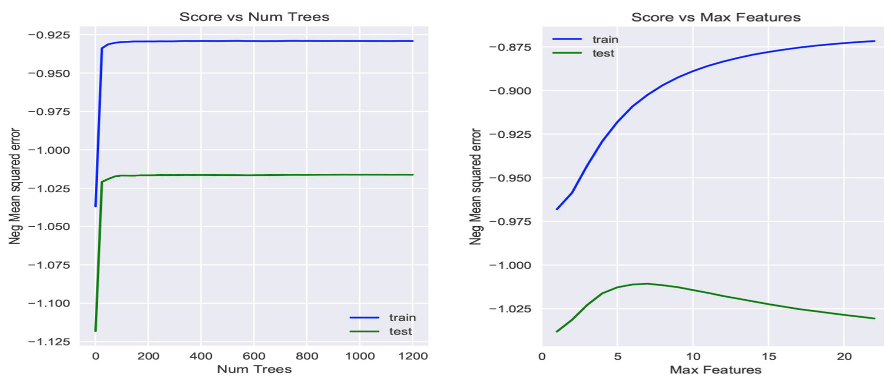

FIG. 6. *Left: relationship of number of trees in the forest with negative mean squared error; Right: relationship of maximum features in the forest with negative mean squared error;*

2.5. Matrix Factorization.

The idea of Matrix Factorization is approximating the rating matrix R as the product of two matrices: $R \approx PQ$, where P is an $N \times K$ and Q is a $K \times M$ matrix. This factorization gives a low dimensional numerical representation of both users and movies.

In our case, we have lots of unknown ratings of R ($\sim 99\%$) which cannot be represented by zero. Thus, we formulate our problem by minimizing

$$MSE = \frac{1}{|\mathcal{T}|} \sum_{(u,i)\in\mathcal{T}} e_{ui}^2 \tag{2.5}$$

where $e_{ui} = r_{ui} - \hat{r}_{ui}$ for $(u,i) \in \mathcal{T}$ and $\hat{r} = \sum_{k=1}^{K} p_{uk} q_{ki}$. And we apply non-negative matrix factorization method, which restricts elements of P, Q to be non-negative since each matrix element $r \in [1, 5]$, and solving the optimization problem

$$(P^*, Q^*) = \underset{(P \geq 0, Q \geq 0)}{argmin} \ MSE. \tag{2.6}$$

Furthermore, we want to introduce regularization to avoid overfitting and this is done by adding a parameter β and modify the squared error as following:

$$e_{ui}^2 = (r_{ui} - \hat{r}_{ui})^2 + \frac{\beta}{2} \sum_{k=1}^{K} (p_{uk}^2 + q_{ki}^2). \tag{2.7}$$

Then we apply the gradient decent method to find a local minimum of MSE, i.e. update P, Q with

$$p'_{uk} = p_{uk} + \alpha \frac{\partial}{\partial p_{uk}} e_{ui}^2 = p_{uk} + \alpha(2 e_{ui} q_{ki} - \beta p_{uk}) \tag{2.8}$$

$$q'_{ki} = q_{ki} + \alpha \frac{\partial}{\partial q_{ki}} e_{ui}^2 = q_{ki} + \alpha(2 e_{ui} p_{uk} - \beta q_{ki}) \tag{2.9}$$

Since we want to optimize the MSE for the validation set, not for the training set, we separate the training set $\mathcal{T}$ into two part: $\mathcal{T}_1$, $\mathcal{T}_2$, then train with $\mathcal{T}_1$ and terminate when the MSE for $\mathcal{T}_2$ does not decrease during two epochs. And we also select the dimensional parameter $K \in \{2, 5, 10, 15, 20\}$, and regularization parameter $\beta \in (0, 0.3)$ base on the MSE of $\mathcal{T}_2$.

With this method, we can achieve more precise prediction without detailed information from users and movies. But the results are not interpretable and need higher computational cost than the previous models. Moreover, there are some upgraded methods by tuning the regularization parameter and learning rate, or pre-process data [9].

3. Main Results.

The following table shows the MSE of the validation set by the four different models. We can see that with regularization, the result display only marginal improvement to the linear regression. The two non-parametric models have a more extensive reduction of MSE compare to the linear models. Matrix factorization is the most accurate model with the lowest MSE of 0.9334. However, all models are inadequate with high MSE, and we think it is due to the sparsity of the data and the data with high variance. In addition, all models have more accurate predictions for higher ratings than lower ratings. One explanation is that we have less data for lower ratings, e.g. about 5% of the total ratings is for rating of 1.

Model	OLR	Ridge/Lasso	RF	MF
MSE	1.1502	1.1499	0.9819	0.9334

TABLE 1
Result of MSE of the four models

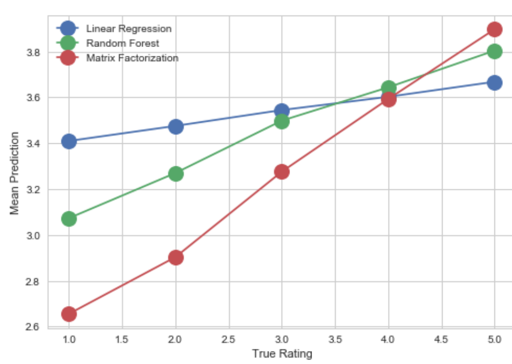

FIG. 7. *prediction vs true value*

4. Summary.

In this project, we design and implement parametric and non-parametric statistical models on a real-world data set and compare their performance. Upon analysis of the movie recommendation data using Python, we explore Python packages such as pandas to deal with data, seaborn and matplotlib to perform data visualization, statsmodels and scikit-learn to build the model, etc. After exploration, we find matrix factorization method, with the largest parameter size, is superior to linear regression and random forest techniques for producing movie recommendation. In future, we will continue to explore other models that may fit the data better to improve the accuracy of the recommendation system further, and practice better data processing strategies such as missing data imputation[4].

Appendix A. Data Preprocessing.

```
# Movie Recommendation Lab
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import seaborn as sns
from sklearn import datasets, linear_model
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold # import KFold
from sklearn import feature_selection
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import Ridge, Lasso
from sklearn.linear_model import LassoCV
import statsmodels.formula.api as smf
import statsmodels.api as sm

#upload data
data_users = pd.read_csv("users.csv")
data_movies = pd.read_csv("movies.tsv", sep='\t')
data_ratings = pd.read_csv("ratings.csv")
data_all = pd.read_csv("allData.tsv", sep='\t')

#view data
data_all.head()

#view data
data_all.head()
data_all.columns
data_all.info()

#analyze data
```

```
data_all['age'].describe()
data_all['gender'].describe()

#missing data
data_all.count()
data_all.isnull().sum()

# Create a set of dummy variables from the gender and genre variable
df_gender = pd.get_dummies(data_all['gender'])
df_genre1 = pd.get_dummies(data_all['genre1'])
df_genre2 = pd.get_dummies(data_all['genre2'])
df_genre3 = pd.get_dummies(data_all['genre3'])

df_genre = pd.concat([df_genre1,df_genre2,df_genre3]).groupby(level=0).any().astype(int)

data_new = pd.concat([data_all[['userID', 'age', 'movieID', 'name', 'year']],
           df_gender, df_genre,data_all['rating']], axis=1)

data_X = pd.concat([data_all[['age', 'year']], df_gender, df_genre], axis=1)
data_y = data_all['rating']

# create training and testing vars
X_train, X_test, y_train, y_test = train_test_split(data_X, data_y, test_size=0.2, random_state=1234)
print (X_train.shape, y_train.shape)
print (X_test.shape, y_test.shape)

# create Cross-Validation K-Fold
kf = KFold(n_splits=10, random_state=1234, shuffle=True)
for train_index, test_index in kf.split(X_train):
    X_CV_train, X_CV_test = X_train.iloc[train_index], X_train.iloc[test_index]
    y_CV_train, y_CV_test = y_train.iloc[train_index], y_train.iloc[test_index]

#Drop M (only include F in the model)
X_train = X_train.drop ('M', 1)
X_test = X_test.drop('M', 1)
```

Appendix B. Data visualization.

```
# Correlation matrix
d = data_new
# Compute the correlation matrix
corr = d.corr()
# Generate a mask for the upper triangle
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(22, 19))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
```

```
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,annot=True,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})

plt.savefig('fig/correlation_v4.png')



#Plot year against average rating by year
rating_year=data_all.groupby('year')
means=rating_year.mean()
plt.scatter(means.index, means['rating'])
plt.ylabel('Average Rating')
plt.xlabel('Year')
plt.show()

#Plot age against average rating by age
rating_year=data_all.groupby('age')
means=rating_year.mean()
plt.scatter(means.index, means['rating'])
plt.ylabel('Average Rating')
plt.xlabel('Age Group')
plt.show()
```

Appendix C. Linear Regression.

```
#Linear regression
# Train the model using the training sets
X = sm.add_constant(X_train)
olsmod = sm.OLS(y_train, X).fit()
print(olsmod.summary())

# Test the model using the test sets
X2 = sm.add_constant(X_test)
ypred = olsmod.predict(X2)

# Test linear regression model using MSE
np.mean((ypred - y_test)**2)

#Stepwise Selection:

#Drop Musical
Xa = X.drop('Musical', 1)
X2a= X2.drop('Musical', 1)
olsmod = sm.OLS(y_train, Xa).fit()
print(olsmod.summary())
ypred = olsmod.predict(X2a)
MSE = np.mean((ypred - y_test)**2)
print(MSE)

#Drop Musical+ Mystery
Xb = Xa.drop('Mystery', 1)
X2b= X2a.drop('Mystery', 1)
olsmod = sm.OLS(y_train, Xb).fit()
```

```
print(olsmod.summary())
ypred = olsmod.predict(X2b)
MSE = np.mean((ypred - y_test)**2)
print(MSE)

#Drop Musical+ Mystery+Adventure
Xc = Xb.drop('Adventure', 1)
X2c= X2b.drop('Adventure', 1)
olsmod = sm.OLS(y_train, Xc).fit()
print(olsmod.summary())
ypred = olsmod.predict(X2c)
MSE = np.mean((ypred - y_test)**2)
print(MSE)

#Drop Musical+ Mystery+Adventure+F
Xd = Xc.drop('F', 1)
X2d= X2c.drop('F', 1)
olsmod = sm.OLS(y_train, Xd).fit()
print(olsmod.summary())
ypred = olsmod.predict(X2d)
MSE = np.mean((ypred - y_test)**2)
print(MSE)

#Drop Musical+ Mystery+Adventure+F+age
Xe = Xd.drop('age', 1)
X2e= X2d.drop('age', 1)
olsmod = sm.OLS(y_train, Xe).fit()
print(olsmod.summary())
ypred = olsmod.predict(X2e)
MSE = np.mean((ypred - y_test)**2)
print(MSE)


#Linear model using forward_selection using adj R^2 as selection criterion
def forward_selected(data):
    """
    model: an "optimal" fitted statsmodels linear model
           with an intercept
           selected by forward selection
           evaluated by adjusted R-squared
    """
    remaining = set(data.columns)
    remaining.remove('rating')
    selected = []
    current_score, best_new_score = 0.0, 0.0
    while remaining and current_score == best_new_score:
        scores_with_candidates = []
        for candidate in remaining:
            formula = "{} ~ {} + 1".format('rating',
                                           ' + '.join(selected + [candidate]))
            score = smf.ols(formula, data).fit().rsquared_adj
            scores_with_candidates.append((score, candidate))
        scores_with_candidates.sort()
        best_new_score, best_candidate = scores_with_candidates.pop()
        if current_score < best_new_score:
            remaining.remove(best_candidate)
            selected.append(best_candidate)
```

```
            print(selected)
            current_score = best_new_score
    formula = "{} ~ {} + 1".format('rating',
                                   ' + '.join(selected))
    model = smf.ols(formula, data).fit()
    return model

model = forward_selected(data)
print (model.model.formula)
print (model.rsquared_adj)
```

Appendix D. Ridge and Lasso.

```
#Ridge and Lasso Regression
from sklearn.linear_model import Ridge, Lasso
from sklearn.linear_model import RidgeCV, LassoCV

#Find the best alpha in Lasso
alphas = np.logspace(-5, -2, 1000)
lassocv = linear_model.LassoCV(alphas=alphas, cv=10, random_state=1234)
lassocv.fit(data_X, data_y)
lassocv_score = lassocv.score(data_X, data_y)
lassocv_alpha = lassocv.alpha_
lassocv_alpha


#Find the best alpha in Ridge
alphas = np.linspace(0,20,200)
ridgecv = linear_model.RidgeCV(alphas=alphas, scoring=None, cv=10)
ridgecv.fit(data_X, data_y)
ridgecv_score = ridgecv.score(data_X, data_y)
ridgecv_alpha = ridgecv.alpha_
ridgecv_alpha

#Fit Lasso Regression
lasso=Lasso(alpha=0.0003)
lasso.fit(X_train, y_train)
y_est=lasso.predict(X_test)
mse=np.mean(np.square(y_test-y_est))
print(mse)
print(lasso.coef_)
lassodf=pd.DataFrame(lasso.coef_, index=data_X.columns)
lasso.intercept_

#Fit Ridge Regression
ridge=Ridge(alpha=9.7)
ridge.fit(X_train, y_train)
y_est=ridge.predict(X_test)
mse=np.mean(np.square(y_test-y_est))
print(mse)
print(ridge.coef_)
ridgedf=pd.DataFrame(ridge.coef_, index=data_X.columns)

#Plot alpha against coefficients in Ridge
```

```
alphas = 10**np.linspace(10,-2,100)*0.5
coefs = []
for a in alphas:
    ridge.set_params(alpha=a)
    ridge.fit(X_train, y_train)
    coefs.append(ridge.coef_)
ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale("log")
plt.axis("tight")
plt.xlabel("alpha")
plt.ylabel("weights")

#Plot alpha against coefficients in Lasso
alphas = 10**np.linspace(3,-4,100)*0.5
coefs = []
for a in alphas:
    lasso.set_params(alpha=a)
    lasso.fit(X_train, y_train)
    coefs.append(lasso.coef_)
ax = plt.gca()
ax.plot(alphas*2, coefs)
ax.set_xscale("log")
plt.axis("tight")
plt.xlabel("alpha")
plt.ylabel("weights")
```

Appendix E. Random Forest.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error as MSE
import pydot
from sklearn.tree import export_graphviz# default parameter
rf = RandomForestRegressor(random_state = 1234)
rf.fit(X_train, y_train)

MSE(y_test, rf.predict(X_test))
#1.0776258432691597

# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
```

```
bootstrap = [True, False]

# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}


# Use the random grid to search for best hyperparameters
# First create the base model to tune
rf = RandomForestRegressor(random_state = 1234)
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
rf_random = RandomizedSearchCV(estimator=rf, param_distributions=random_grid,
                               n_iter = 100, scoring='neg_mean_squared_error',
                               cv = 10, verbose=2, random_state=1234, n_jobs=-1,
                               return_train_score=True)

# Fit the random search model
rf_random.fit(X_train, y_train)
rf_random.best_params_
#{'bootstrap': True,
# 'max_depth': 90,
# 'max_features': 'sqrt',
# 'min_samples_leaf': 4,
# 'min_samples_split': 2,
# 'n_estimators': 1000}
rf_random.cv_results_

best_random = rf_random.best_estimator_
random_mse = MSE(best_random, X_test, y_test)
print(random_mse)
#0.98936

# Fit the grid search model
base_model = RandomForestRegressor(n_estimators = 10, random_state = 1234)
base_model.fit(X_train, y_train)
MSE(y_test, base_model.predict(X_test))
#1.0776


# Create the parameter grid based on the results of random search
param_grid = {
    'bootstrap': [True],
    'max_depth': [50, 60, 70, 80, 90],
    'max_features': ['sqrt', 'auto'],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [2, 3, 4],
    'n_estimators': [500, 800, 1000, 1200]
}

# Create a base model
rf = RandomForestRegressor(random_state = 1234)
```

```
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                          cv = 10, n_jobs = -1, verbose = 2, return_train_score=True)


# Fit the grid search to the data
grid_search.fit(X_train, y_train)


grid_search.best_params_
#{'bootstrap': True,
# 'max_depth': 50,
# 'max_features': 'sqrt',
# 'min_samples_leaf': 4,
# 'min_samples_split': 2,
# 'n_estimators': 1000}
best_grid = grid_search.best_estimator_
MSE(y_test, best_grid.predict(X_test))
#MSE: 0.9894.


#Training Curves
#Number of Trees

# Grid with only the number of trees changed
tree_grid = {'n_estimators': [int(x) for x in np.linspace(1, 1200, 50)]}

# Create the grid search model and fit to the training data
tree_grid_search = GridSearchCV(best_grid, param_grid=tree_grid, verbose = 2,
                                n_jobs=-1, cv = 10, scoring = 'neg_mean_squared_error')
tree_grid_search.fit(X_train, y_train)

tree_grid_search.cv_results_

def plot_results(model, param = 'n_estimators', name = 'Num Trees'):
    param_name = 'param_%s' % param

    # Extract information from the cross validation model
    train_scores = model.cv_results_['mean_train_score']
    test_scores = model.cv_results_['mean_test_score']
    train_time = model.cv_results_['mean_fit_time']
    param_values = list(model.cv_results_[param_name])

    # Plot the scores over the parameter
    plt.subplots(1, 2, figsize=(10, 6))
    plt.subplot(121)
    plt.plot(param_values, train_scores, 'b', label = 'train')
    plt.plot(param_values, test_scores, 'g', label = 'test')
    plt.legend()
    plt.xlabel(name)
    plt.ylabel('Neg Mean squared error')
    plt.title('Score vs %s' % name)

    plt.subplot(122)
    plt.plot(param_values, train_time, 'r')
    plt.xlabel(name)
    plt.ylabel('Train Time (sec)')
```

```
    plt.title('Training Time vs %s' % name)

plot_results(tree_grid_search)
plt.savefig("fig/Number_of_Trees.pdf")


#Number of Features at Each Split
# Define a grid over only the maximum number of features
feature_grid = {'max_features': list(range(1, X_train.shape[1] + 1))}
feature_grid_search = GridSearchCV(best_grid, param_grid=feature_grid, cv = 10,
                                   n_jobs=-1, verbose= 2, scoring = 'neg_mean_squared_error')
feature_grid_search.fit(X_train, y_train)
plot_results(feature_grid_search, param='max_features', name = 'Max Features')
plt.savefig("fig/max_features.pdf")




#Another Round of Grid Search
# Create a base model
rf = RandomForestRegressor(random_state = 1234)

param_grid_2 = {
    'bootstrap': [True],
    'max_depth': [50,55, 60],
    'max_features': [5, 7, 9, 11],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [2, 3],
    'n_estimators': [200,400, 800, 1000, 1200]
}


# Instantiate the grid search model
grid_search_final = GridSearchCV(estimator = rf, param_grid = param_grid_2,
                                 cv = 10, n_jobs = -1, verbose = 2, return_train_score=True)

grid_search_final.fit(X_train, y_train)
grid_search_final.best_params_
best_grid_final = grid_search_final.best_estimator_
MSE(y_test, grid_search_final.predict(X_test))
#0.9819552658313127
final_model = grid_search.best_estimator_

#using re_substitution best_estimator
rf = RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=50,
           max_features='sqrt', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=4, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=1000, n_jobs=1,
           oob_score=False, random_state=1234, verbose=0, warm_start=False)

rf.fit(X_train, y_train)
pred = []
pred = rf.predict(X_test)

print(pred)
len(pred)
```

```
d = {'test': y_test, 'predict': pred}
df_res = pd.DataFrame(data=d)
df_res.groupby('test').mean()

residual=np.subtract(y_test,pred)
print(residual)
sns.stripplot(y_test, residual, jitter=True, alpha=.85)
sns.despine()

visual_tree = final_model.estimators_[12]
feature_names = list(X_train)
export_graphviz(visual_tree, out_file = 'fig/best_tree.dot', feature_names = feature_names,
                precision = 2, filled = True, rounded = True, max_depth = None)

# Use pydot for converting to an image file
# Import the dot file to a graph and then convert to a png
(graph, ) = pydot.graph_from_dot_file('fig/best_tree.dot')
graph.write_png('fig/best_tree.png')

# Plot feature importance
feature_importance = final_model.feature_importances_
# make importances relative to max importance
feature_importance = 100.0 * (feature_importance / feature_importance.max())
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5
plt.figure(figsize=(6, 6))
plt.barh(pos, feature_importance[sorted_idx], align='center')
feature_names= np.array(feature_names)
plt.yticks(pos, feature_names[sorted_idx])
plt.xlabel('Relative Importance')
plt.title('Variable Importance')
#plt.show()
plt.savefig('fig/Variable_Importance_v2.png')
```

Appendix F. Matrix Factorization.

```
def matrix_factorization(data, df, R, P, Q, K, alpha, beta, steps=100, ):
    indices = range(data.shape[0])
    indices_train, indices_test = train_test_split(indices, test_size=0.1, random_state=123)
    Q = Q.T
    e = 100*len(indices_test)
    P_temp = P.copy()
    Q_temp = Q.copy()
    for step in range(steps):
        for k in indices_train:
            uid = data.iloc[k,0]
            mid = data.iloc[k,3]
            i = df.index.get_loc(uid)
            j = df.columns.get_loc(mid)
            eij = R[i][j] - np.dot(P_temp[i,:],Q_temp[:,j])
            for k in range(K):
                P_temp[i][k] = max(P_temp[i][k] + alpha * (2 * eij * Q_temp[k][j] - beta * P_temp[i][k]),0)
```

```
                Q_temp[k][j] = max(Q_temp[k][j] + alpha * (2 * eij * P_temp[i][k] - beta * Q_temp[k][j]),0)


        #eR = np.dot(P,Q)
        print(step)

        e_temp = 0
        for k in indices_test:
            uid = data.iloc[k,0]
            mid = data.iloc[k,3]
            i = df.index.get_loc(uid)
            j = df.columns.get_loc(mid)
            e_temp = e_temp + pow(R[i][j] - np.dot(P_temp[i,:],Q_temp[:,j]), 2)
#                    for k in range(K):
#                        e = e + (beta/2) * ( pow(P[i][k],2) + pow(Q[k][j],2) )
        print(e_temp/len(indices_test))
#        if (e_temp/31620) < 0.1:
#            break
        if e_temp < e:
            if e-e_temp < 0.00001:
                break
            alpha = alpha*1.05
            P = P_temp.copy()
            Q = Q_temp.copy()
            e = e_temp
            temp = 0
        else:
            alpha = alpha*0.5
            P_temp = P.copy()
            Q_temp = Q.copy()
            temp += 1
        if temp == 3:
            break
    return P, Q.T

R = df.values

N = len(R)
M = len(R[0])
K = 15 #{2, 5, 10, 15, 20}
alpha = 0.01
beta = 0.25 #[0,0.3]

np.random.seed(123)
P = np.random.rand(N,K)
Q = np.random.rand(M,K)

nP, nQ = matrix_factorization(data_all.iloc[indices_train], df, R, P, Q, K, alpha, beta)
nR = np.dot(nP, nQ.T)

# compute MSE on validation set with given P,Q
e = 0
r_test = []
r_pred = []
for k in indices_test:
    uid = data_all.iloc[k,0]
    mid = data_all.iloc[k,3]
```

```
    i = df.index.get_loc(uid)
    j = df.columns.get_loc(mid)
    e += (R[i,j]-nR[i,j])**2
    r_test.append(R[i,j])
    r_pred.append(nR[i,j])


mse = e/len(indices_test)
```

REFERENCES

[1] R. M. Bell and Y. Koren, *Lessons from the netflix prize challenge*, Acm Sigkdd Explorations Newsletter, 9 (2007), pp. 75–79.

[2] J. Bennett, S. Lanning, et al., *The netflix prize*, in Proceedings of KDD cup and workshop, vol. 2007, New York, NY, USA, 2007, p. 35.

[3] H. Byström, *Movie recommendations from user ratings*, 2013.

[4] C. Christakou, S. Vrettos, and A. Stafylopatis, *A hybrid movie recommender system based on neural networks*, International Journal on Artificial Intelligence Tools, 16 (2007), pp. 771–792.

[5] S. Derksen and H. J. Keselman, *Backward, forward and stepwise automated subset selection algorithms: Frequency of obtaining authentic and noise variables*, British Journal of Mathematical and Statistical Psychology, 45 (1992), pp. 265–282.

[6] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1, Springer series in statistics New York, 2001.

[7] M. S. Lewis-Beck, *Stepwise regression: A caution*, Political Methodology, (1978), pp. 213–240.

[8] E. W. Steyerberg, M. J. Eijkemans, and J. D. F. Habbema, *Stepwise selection in small data sets: a simulation study of bias in logistic regression analysis*, Journal of clinical epidemiology, 52 (1999), pp. 935–942.

[9] G. Takács, I. Pilászy, B. Németh, and D. Tikk, *Matrix factorization and neighbor based algorithms for the netflix prize problem*, in Proceedings of the 2008 ACM conference on Recommender systems, ACM, 2008, pp. 267–274.

[10] S. Weisberg, *Applied linear regression*, vol. 528, John Wiley & Sons, 2005.