Lab 4 - Statistical Data Analytics

Table of Contents

I. Classification 3
II. Linear Regression 6
III. Recommendation Systems 11
IV. Exercises (OPTIONAL) 12
1. Classification 12
2. Recommendation Systems (Advanced) 12

I. Classification

In this exercise, you'll be working with the MNIST digits recognition dataset, which has 10 classes, the digits 0 through 9! A reduced version of the MNIST dataset is one of scikit-learn's included datasets, and that is the one we will use in this exercise.

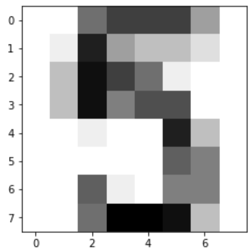
Each sample in this scikit-learn dataset is an 8x8 image representing a handwritten digit. Each pixel is represented by an integer in the range 0 to 16, indicating varying levels of black.

To load dataset, using the following code:

```
# Import necessary modules
from sklearn import datasets
import matplotlib.pyplot as plt

# Load the digits dataset: digits
digits = datasets.load_digits()
```

Display a random number to verify the dataset

```
# Display image 1010
plt.imshow(digits.images[1010], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```



Before applying the classifier, we need to split the dataset into training and testing parts.

https://developers.google.com/machine-learning/crash-course/training-and-test-sets/splitting-data

```
from sklearn.model_selection import train_test_split
X = digits.data
y  = digits.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42, stratify=y)
```

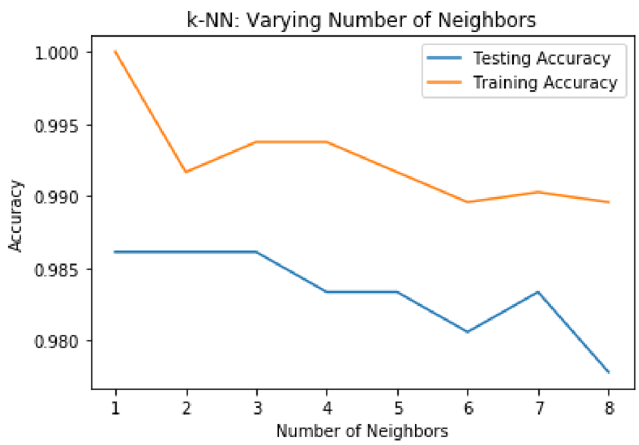
Build KNN classifier for the above dataset.

```
# Import necessary modules
from sklearn.neighbors import KNeighborsClassifier

import numpy as np

# Create a k-NN classifier with 3 neighbors: knn
knn = KNeighborsClassifier(n_neighbors=3)

# Fit the classifier to the training data
knn.fit(X_train, y_train)

# Print the accuracy
print("Accuracy: {0}".format(knn.score(X_test, y_test)))
```

✓ Varying Number of Neighbours

In this exercise, you need to compute and plot the training and testing accuracy scores with different values of k (e.g. 1 to 8).

```
# Setup arrays to store train and test accuracies
neighbors = np.arange(1, 9)
train_accuracy = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

# Loop over different values of k
for i, k in enumerate(neighbors):
    # Setup a k-NN Classifier with k neighbors: knn
    knn = KNeighborsClassifier(n_neighbors=k)

    # Fit the classifier to the training data
    knn.fit(X_train, y_train)

    #Compute accuracy on the training set
    train_accuracy[i] = knn.score(X_train, y_train)

    #Compute accuracy on the testing set
    test_accuracy[i] = knn.score(X_test, y_test)

# Generate plot
plt.title('k-NN: Varying Number of Neighbors')
plt.plot(neighbors, test_accuracy, label = 'Testing Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training Accuracy')
plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```



Classification with deep learning

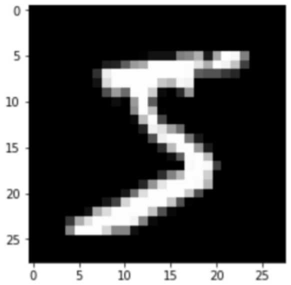
```
In [1]: from __future__ import print_function
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        from torch.autograd import Variable
```

```
In [2]: from torchvision import datasets, transforms
        mnist = datasets.MNIST(root='.', train=True, download=True)
```

```
In [3]: print("Number of training examples", mnist.train_data.shape)
        print("Image information", mnist[0])

        Number of training examples torch.Size([60000, 28, 28])
        Image information (<PIL.Image.Image image mode=L size=28x28 at 0x11B6B1B38>, tensor(5))
```

```
In [4]: import matplotlib.pyplot as plt
        %matplotlib inline
        plt.imshow(mnist[0][0])
```

Out[4]: <matplotlib.image.AxesImage at 0x122e437f0>



In [5]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.fully = nn.Sequential(
            nn.Linear(28*28, 10)
        )

    def forward(self, x):
        x = x.view([-1,28*28])
        x = self.fully(x)
        x = F.log_softmax(x, dim=1)
        return x
```

In [6]:
```
train_loader = torch.utils.data.DataLoader(datasets.MNIST(root='.',train=True,transform=transforms.Compose([transform
s.ToTensor()])), batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(datasets.MNIST(root='.',train=False,transform=transforms.Compose([transform
s.ToTensor()])), batch_size=1, shuffle=True)
```

In [7]:
```
def train():
    learning_rate = 1e-3
    num_epochs = 3

    net = Net()
    optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)

    for epoch in range(num_epochs):
        for batch_idx, (data, target) in enumerate(train_loader):
            output = net(data)

            loss = F.nll_loss(output, target)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if batch_idx % 100 == 0:
                print('Epoch = %f. Batch = %s. Loss = %s' % (epoch, batch_idx, loss.item()))

    return net
```

In [8]:
```
net = train()
```

```
Epoch = 0.000000. Batch = 0. Loss = 2.303990364074707
Epoch = 0.000000. Batch = 100. Loss = 0.944770097732544
Epoch = 0.000000. Batch = 200. Loss = 0.6024922132492065
Epoch = 0.000000. Batch = 300. Loss = 0.37279099225997925
Epoch = 0.000000. Batch = 400. Loss = 0.4637526273727417
Epoch = 0.000000. Batch = 500. Loss = 0.30020996928215027
Epoch = 0.000000. Batch = 600. Loss = 0.46635979413986206
Epoch = 0.000000. Batch = 700. Loss = 0.28683096170425415
Epoch = 0.000000. Batch = 800. Loss = 0.34193897247314453
Epoch = 0.000000. Batch = 900. Loss = 0.2998521625995636
Epoch = 1.000000. Batch = 0. Loss = 0.3675362169742584
Epoch = 1.000000. Batch = 100. Loss = 0.3356616795063019
Epoch = 1.000000. Batch = 200. Loss = 0.46262598037719727
Epoch = 1.000000. Batch = 300. Loss = 0.45991790294647217
Epoch = 1.000000. Batch = 400. Loss = 0.41947293281555176
Epoch = 1.000000. Batch = 500. Loss = 0.44147467613220215
Epoch = 1.000000. Batch = 600. Loss = 0.32572752237319946
Epoch = 1.000000. Batch = 700. Loss = 0.1993783563375473
Epoch = 1.000000. Batch = 800. Loss = 0.31467846035957336
Epoch = 1.000000. Batch = 900. Loss = 0.1647953987121582
Epoch = 2.000000. Batch = 0. Loss = 0.3867589831352234
Epoch = 2.000000. Batch = 100. Loss = 0.4215034246444702
Epoch = 2.000000. Batch = 200. Loss = 0.21791870892047882
Epoch = 2.000000. Batch = 300. Loss = 0.20025384426116943
Epoch = 2.000000. Batch = 400. Loss = 0.33563050627708435
Epoch = 2.000000. Batch = 500. Loss = 0.28268325328826904
Epoch = 2.000000. Batch = 600. Loss = 0.16347536444664001
Epoch = 2.000000. Batch = 700. Loss = 0.3204724192619324
Epoch = 2.000000. Batch = 800. Loss = 0.3459726572036743
Epoch = 2.000000. Batch = 900. Loss = 0.13796009123325348
```

In [9]:
```
net.eval()
test_loss = 0
correct = 0
total = 0

for data, target in test_loader:
    total += len(target)
    output = net(data)
    pred = output.max(1, keepdim=True)[1]
    correct += target.eq(pred.view_as(target)).sum()

print("Correct out of %s" % total, correct.item())
print("Percentage accuracy", correct.item()*100/10000.)
```

```
Correct out of 10000 9229
Percentage accuracy 92.29
```

II. Linear Regression

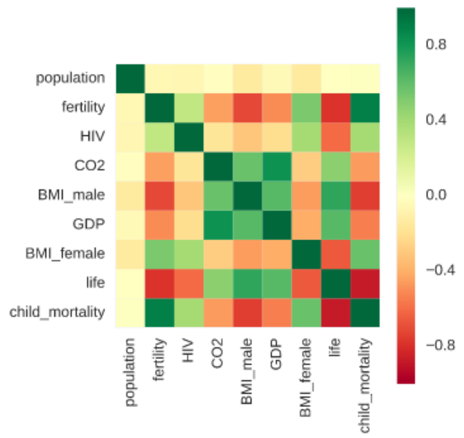
You will work with Gapminder data that in CSV file available in the workspace as 'gapminder.csv'. Specifically, your goal will be to use this data to predict the life expectancy in a given country based on features such as the country's GDP, fertility rate, and population.

❖ Load the dataset

In [1]:
```
# Import numpy and pandas
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Read the CSV file into a DataFrame: df
df = pd.read_csv('gapminder.csv')
```

❖ Use seaborn to visualize the data of Gapminder like following image:

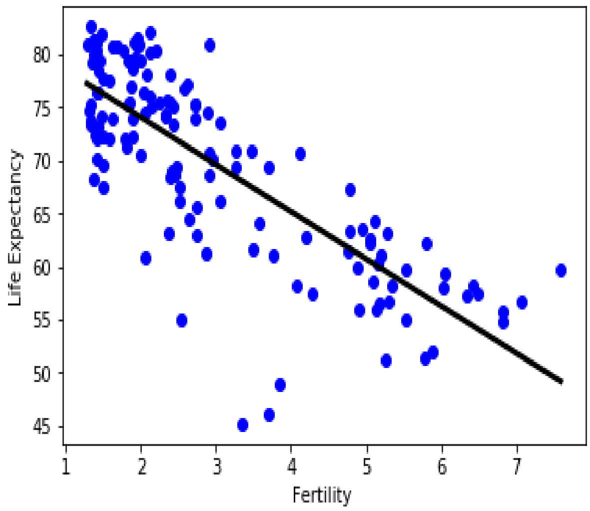
In [5]:
```
ax = sns.heatmap(df.corr(), square=True, cmap='RdYlGn')
plt.show()
```



❖ Apply linear regression with the 'fertility' feature to predict life expectancy.

In [6]:

```
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

x_fertility = df['fertility'].values.reshape(-1,1)
y_life = df['life'].values.reshape(-1,1)
prediction_space = np.linspace(min(x_fertility), max(x_fertility)).reshape(-1,1)
# Create training and test sets
x_train, x_test, y_train, y_test = train_test_split(x_fertility, y_life, test_size=0.3, random_state=42)

# Create the regression model: reg_all
reg = LinearRegression()

# Fit the regression to the training data
reg.fit(x_train, y_train)
y_predict = reg.predict(prediction_space)

# Print accuracy
print(reg.score(x_fertility, y_life))

# Plot regression line
plt.scatter(x_fertility, y_life, color='blue')
plt.plot(prediction_space, y_predict, color='black', linewidth=3)
plt.ylabel('Life Expectancy')
plt.xlabel('Fertility ')
plt.show()
```



❖ Apply linear regression with the **all** features to predict life expectancy. Compare the model score when using all features to one feature in previous step.

In [7]:

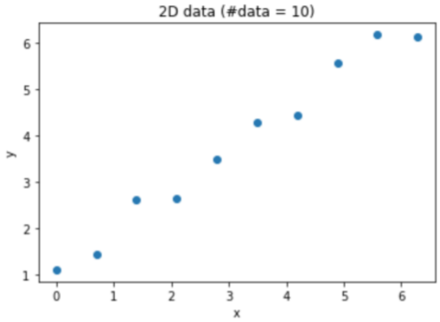
```
features = pd.read_csv('gapminder.csv')
df = pd.read_csv('gapminder.csv')
del features['life']
del features['Region']

y_life = df['life'].values.reshape(-1,1)

# Create training and test sets
x_train, x_test, y_train, y_test = train_test_split(features, y_life, test_size=0.3, random_state=42)

# Create the regression model: reg_all
reg_all = LinearRegression()
# Fit the regression to the training data
reg_all.fit(x_train, y_train)

# Print accuracy
print(reg_all.score(features, y_life))
```

```
0.8914651485793135
```

Linear Regression using PyTorch

In [16]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
```

In [17]:
```
N = 10 # number of data points
m = .9
c = 1
x = np.linspace(0,2*np.pi,N)
y = m*x + c + np.random.normal(0,.3,x.shape)
plt.figure()
plt.plot(x,y,'o')
plt.xlabel('x')
plt.ylabel('y')
plt.title('2D data (#data = %d)' % N)
plt.show()
```



In [18]:
```
import torch
```

Dataset

In [19]:
```
from torch.utils.data import Dataset
class MyDataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        sample = {
            'feature': torch.tensor([1,self.x[idx]]),
            'label': torch.tensor([self.y[idx]])}
        return sample
```

In [20]:
```
dataset = MyDataset(x, y)
for i in range(len(dataset)):
    sample = dataset[i]
    print(i, sample['feature'], sample['label'])
```

```
0 tensor([1., 0.]) tensor([1.0971])
1 tensor([1.0000, 0.6981]) tensor([1.4373])
2 tensor([1.0000, 1.3963]) tensor([2.6160])
3 tensor([1.0000, 2.0944]) tensor([2.6413])
4 tensor([1.0000, 2.7925]) tensor([3.4891])
5 tensor([1.0000, 3.4907]) tensor([4.2880])
6 tensor([1.0000, 4.1888]) tensor([4.4478])
7 tensor([1.0000, 4.8869]) tensor([5.5624])
8 tensor([1.0000, 5.5851]) tensor([6.1863])
```

```
9 tensor([1.0000, 6.2832]) tensor([6.1383])
```

Dataloader

In [21]:
```python
from torch.utils.data import DataLoader

dataset = MyDataset(x, y)
batch_size = 4
shuffle = True
num_workers = 4
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle, num_workers=num_workers)
```

In [22]:
```python
import pprint as pp
for i_batch, samples in enumerate(dataloader):
    print('\nbatch# = %s' % i_batch)
    print('samples: ')
    pp.pprint(samples)
```

```
batch# = 0
samples:
{'feature': tensor([[1.0000, 2.7925],
        [1.0000, 2.0944],
        [1.0000, 4.8869],
        [1.0000, 6.2832]]),
 'label': tensor([[3.4891],
        [2.6413],
        [5.5624],
        [6.1383]])}

batch# = 1
samples:
{'feature': tensor([[1.0000, 4.1888],
        [1.0000, 5.5851],
        [1.0000, 0.6981],
        [1.0000, 3.4907]]),

 'label': tensor([[4.4478],
        [6.1863],
        [1.4373],
        [4.2880]])}

batch# = 2
samples:
{'feature': tensor([[1.0000, 1.3963],
        [1.0000, 0.0000]]),
 'label': tensor([[2.6160],
        [1.0971]])}
```

Model

In [23]:
```python
import torch.nn as nn
import torch.nn.functional as F
class MyModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(MyModel, self).__init__()
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        out = self.linear(x)
        return out
```

Setting a model for our problem

In [24]:
```
input_dim = 2
output_dim = 1

model = MyModel(input_dim, output_dim)
```

Cost function

Often called loss or error

In [25]:
```
cost = nn.MSELoss()
```

Minimizing the cost function

In other words training (or learning from data)

In [26]:
```
num_epochs = 10  # How many times the entire training data is seen?
l_rate = 0.01
optimiser = torch.optim.SGD(model.parameters(), lr = l_rate)

dataset = MyDataset(x, y)
batch_size = 4
shuffle = True
num_workers = 4
training_sample_generator = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle, num_workers=num_workers)

for epoch in range(num_epochs):
    print('Epoch = %s' % epoch)
    for batch_i, samples in enumerate(training_sample_generator):
        predictions = model(samples['feature'])
        error = cost(predictions, samples['label'])
        print('\tBatch = %s, Error = %s' % (batch_i, error.item()))
```

```
        # Before the backward pass, use the optimizer object to zero all of the
        # gradients for the variables it will update (which are the learnable
        # weights of the model). This is because by default, gradients are
        # accumulated in buffers( i.e, not overwritten) whenever .backward()
        # is called. Checkout docs of torch.autograd.backward for more details.
        optimiser.zero_grad()

        # Backward pass: compute gradient of the loss with respect to model
        # parameters
        error.backward()

        # Calling the step function on an Optimizer makes an update to its
        # parameters
        optimiser.step()
```

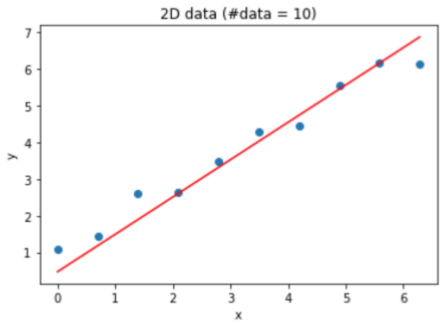
```
Epoch = 0
        Batch = 0, Error = 3.3245582580566406
        Batch = 1, Error = 1.638617753982544
        Batch = 2, Error = 0.3832966983318329
Epoch = 1
        Batch = 0, Error = 0.7121678590774536
        Batch = 1, Error = 0.21079406142234802
        Batch = 2, Error = 0.5607050061225891
Epoch = 2
        Batch = 0, Error = 0.5053210258483887
        Batch = 1, Error = 0.03993864730000496
        Batch = 2, Error = 0.30065447092056274
Epoch = 3
        Batch = 0, Error = 0.2350146770477295
        Batch = 1, Error = 0.30180448293685913
        Batch = 2, Error = 0.16363243758678436
Epoch = 4
        Batch = 0, Error = 0.07132617384195328
        Batch = 1, Error = 0.3243466913700104
        Batch = 2, Error = 0.3382103145122528

Epoch = 5
        Batch = 0, Error = 0.39112815260887146
        Batch = 1, Error = 0.1373337060213089
        Batch = 2, Error = 0.046036202460527420
Epoch = 6
        Batch = 0, Error = 0.2974059581756592
        Batch = 1, Error = 0.19180424511432648
        Batch = 2, Error = 0.04105750471353531
Epoch = 7
        Batch = 0, Error = 0.24729019403457642
        Batch = 1, Error = 0.06893004477024078
        Batch = 2, Error = 0.35701268911361694
Epoch = 8
        Batch = 0, Error = 0.16357268393039703
        Batch = 1, Error = 0.30955955386161804
        Batch = 2, Error = 0.04322194680571556
Epoch = 9
        Batch = 0, Error = 0.1650623083114624
        Batch = 1, Error = 0.12940038740634918
        Batch = 2, Error = 0.32625168561935425
```

Lets see how well the model has learnt the data

In [27]:
```
x_for_plotting = np.linspace(0, 2*np.pi, 1000)
design_matrix = torch.tensor(np.vstack([np.ones(x_for_plotting.shape), x_for_plotting]).T, dtype=torch.float32)
print('Design matrix shape:', design_matrix.shape)

y_for_plotting = model.forward(design_matrix)
print('y_for_plotting shape:', y_for_plotting.shape)
```

```
Design matrix shape: torch.Size([1000, 2])
y_for_plotting shape: torch.Size([1000, 1])
```

In [28]:
```
plt.figure()
plt.plot(x,y,'o')
plt.plot(x_for_plotting, y_for_plotting.data.numpy(), 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.title('2D data (#data = %d)' % N)
plt.show()
```



III. Recommendation Systems

You are one of the organizers a festival on a university campus with plenty of food and drinks. You are put in charge of ordering beers for the event, and you want to use a recommender system to make sure that you can better model the preferences of the students in different sections. For such reason, you meet a few students in different sections and ask them to rate the 4 beers for which you gathered information (in a scale from 1 to 5). Unfortunately, not all of them know the beers in question, therefore the rating table is incomplete. (Complete the TODOs in recommendation.ipynb)

Student from:	Desperados	Guinness	chimay triple	Leffe
ICT	4	3	2	3
Medicine	1	2	3	1
Business	?	2	1	?
Environment	4	3	?	?

- ❖ Use cosine similarity to compute the missing rating in this table using user-based collaborative filtering (CF).

```
def user_cf(M, metric='cosine'):
    pred = np.copy(M)
    n_users, n_items = M.shape
    avg_ratings = np.nanmean(M, axis=1)
    sim_users = sim_matrix(M, 'user', metric)
    for i in range(n_users):
        for j in range(n_items):
            if np.isnan(M[i,j]):
                pred[i,j] = avg_ratings[i] + np.nansum(sim_users[i] * (M[:,j] - avg_ratings)) / sum(sim_users[i])
    return pred
```

❖ Similarly, computing the missing rating using item-based CF.

```
def item_cf(M, metric='cosine'):
    pred = np.copy(M)
    n_users, n_items = M.shape
    avg_ratings = np.nanmean(M, axis=0)
    sim_items = sim_matrix(M, 'item', metric)
    for i in range(n_users):
        for j in range(n_items):
            if np.isnan(M[i,j]):
                pred[i,j] = avg_ratings[j] + np.nansum(sim_items[j] * (M[i,:] - avg_ratings)) / sum(sim_items[j])
    return pred
```

This is the rating ground truth for the above data:

Student from:	Desperados	Guinness	Chimay triple	Leffe
ICT	4	3	2	3
Medicine	1	2	3	1
Business	1	2	1	2
Environment	4	3	2	4

❖ Compute the predictive accuracy of the above recommendations

```
evaluateRS(M, M_result, 'user_cf', 'cosine')
evaluateRS(M, M_result, 'user_cf', 'correlation')
evaluateRS(M, M_result, 'item_cf', 'cosine')
evaluateRS(M, M_result, 'item_cf', 'correlation')
```

❖ Compute the ranking quality of the above recommendations

```
results = []
for method in ['user_cf', 'item_cf']:
    for metric in ['cosine', 'correlation']:
        rank_acc = evaluate_rank(M, M_result, method, metric)
        results += ["Rank accuracy of {0} with {1} metric: {2}".format(method[1], metric, rank_acc)]
print("\n".join(results))
```

IV. Exercises

1. Classification

In this part, you will be working with the Iris dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set).

- Load this dataset from scikit-learn
- Classify using kNN with different k and simple neural network as described in Classification section.
- Compare the accuracy of the classifier in the plot.
- Classify using deep learning with CNN (1 plus)

2. Recommendation Systems

You are provided 3 csv files: movies.csv, users.csv and ratings.csv. Please use those datasets and complete the following challenges.

a. Content-Based Recommendation Model

- ❖ Find list of used genres which is used to category the movies.

```
print(listGen)
```

```
['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy', 'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
'Sci-Fi', 'Documentary', 'War', 'Musical']
```

- ❖ Vectorize the relationship between movies and genres and put them into Ij.

```
print(Ij[:4])
```

```
[[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]]
```

- ❖ Vectorize the relationship between users and genres and put them into Uj (if user rate for a movie, he/she has the related history with the movies'genres).

```
print(Uj[:4])
```

```
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1]]
```

- ❖ Compute the cosine_similarity between movies and users. Hint: you can use sklearn.metrics.pairwise and cosine_similarity for quick calculation.

```
[[0.46291005 0.46291005 0.37796447 ... 0.37796447 0.26726124 0.37796447]
 [0.46291005 0.46291005 0.37796447 ... 0.37796447 0.26726124 0.37796447]
 [0.4472136  0.4472136  0.36514837 ... 0.36514837 0.25819889 0.36514837]
 ...
 [0.46291005 0.46291005 0.37796447 ... 0.37796447 0.26726124 0.37796447]
 [0.4472136  0.4472136  0.36514837 ... 0.36514837 0.25819889 0.36514837]
 [0.4472136  0.4472136  0.36514837 ... 0.36514837 0.25819889 0.36514837]]
```

b. Collaborative Filtering Recommendation Model by Users

- ❖ Use train_test_split to split above dataset with the ratio 50/50. The test dataset will be used as groundtruth to evaluate the rating calculated by using the train dataset
- ❖ Create matrix for users, movies and ratings in both training and testing datasets. Hint:

  train_data_matrix = train_data.pivot_table(index='user_id', columns='movie_id', values='rating').astype('float64')

  test_data_matrix = test_data.pivot_table(index='user_id', columns='movie_id', values='rating').astype('float64')

- ❖ Calculate the user correlation. Hint: you can reference help_function.txt for some necessary functions, but you can write the function by yourself. The similarity between item and itself should be 0 to not affect the result.

```
[[ 0.         -0.01578146 -0.20121784 ...  0.08171063 -0.29064092
   0.05356102]
 [-0.01578146  0.          0.0073552  ... -0.04626997  0.09664223
  -0.07852209]
 [-0.20121784  0.0073552   0.         ... -0.01127893  0.00718984
   0.2729944 ]
 ...
 [ 0.08171063 -0.04626997 -0.01127893 ...  0.         -0.26604897
   0.05947466]
 [-0.29064092  0.09664223  0.00718984 ... -0.26604897  0.
  -0.08159598]
 [ 0.05356102 -0.07852209  0.2729944  ...  0.05947466 -0.08159598
   0.        ]]
```

- ❖ Implement a predict based on user correlation coefficient.
- ❖ Predict on train dataset and compare the RMSE with the test dataset.

c. Collaborative Filtering Recommendation Model by Items.

- ❖ Calculate the item correlation

- Implement function to predict ratings based on Item Similarity.

```
[[ 0.         -0.17105086  0.04233412 ...  0.36847422  0.08410575
   0.00899673]
 [-0.17105086  0.         -0.31577814 ... -0.06670856 -0.45442053
   0.34242022]
 [ 0.04233412 -0.31577814  0.         ...  0.04466245 -0.07067555
  -0.57321736]
 ...
 [ 0.36847422 -0.06670856  0.04466245 ...  0.         -0.1191302
   0.34675131]
 [ 0.08410575 -0.45442053 -0.07067555 ... -0.1191302   0.
  -0.4095297 ]
 [ 0.00899673  0.34242022 -0.57321736 ...  0.34675131 -0.4095297
   0.        ]]
```

- Predict on train dataset and compare the RMSE with the test dataset.
- Compare the results between User-based and Item-based. Make conclusion.