In [ ]:

```
import os
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import LabelEncoder

import torch
import torch.utils.data as data
import torch.optim as optim
import torch.nn as nn
```

Select run configurations

In [ ]:

```
# 'MF' - baseline ... just standard Matrix Factorization
# 'NCF' - collaborative ... using NeuMF (GMF==MF + MLP) architecture according to the paper
# 'hybrid' - final model ... duplicated NCF with one branch using pretrained book embeddings
architecture = "hybrid"

# 'colab' vs 'paperspace'
environment = "paperspace"
```

Load data

In [ ]:

```
if environment == "colab":
    from google.colab import drive
    drive.mount("/content/drive", force_remount=True)
    data_path = "/content/drive/Shareddrives/KNN-Recommenders/data/"
else:
    data_path = "/notebooks/data/"

df = pd.read_csv(data_path + "book_interactions_kaggle.csv")
df.head()
```

Out[ ]:

	book_id	user_id	rating
0	2767052	314	5
1	2767052	439	3
2	2767052	588	5
3	2767052	1169	4
4	2767052	1185	4

It's necessary to tranform ids to labels as they can have higher values than their count (index errors)

In [ ]:

```
labelEncoder = LabelEncoder()
df["user_label"] = labelEncoder.fit_transform(df["user_id"])
df["book_label"] = labelEncoder.fit_transform(df["book_id"])

df.head()
```

Out[ ]:

	book_id	user_id	rating	user_label	book_label
0	2767052	314	5	313	6191
1	2767052	439	3	438	6191
2	2767052	588	5	587	6191
3	2767052	1169	4	1168	6191
4	2767052	1185	4	1184	6191

Basic statistics

In [ ]:

```
num_users = df["user_id"].unique().shape[0]
num_books = df["book_id"].unique().shape[0]
df_len = df.shape[0]
print("Users: {}".format(num_users))
print("Books: {}".format(num_books))
print("Interactions: {}".format(df_len))
print("Density: {}/{} ... {} %".format(df_len, num_users*num_books, round(100*df_len/(num_users*num_books), 4)))
```

```
Users: 53423
Books: 9996
Interactions: 979104
Density: 979104/534016308 ... 0.1833 %
```

In case of final hybrid model, load also the pretrained book embeddings

In [ ]:

```
def csv_embedding_to_dict(filename):
    df_embed = pd.read_csv(data_path + filename)
    df_embed.set_index("book_id", inplace=True)
    return {book_id_to_label[id]:row.values.tolist() for id, row in df_embed.iterrows()}

if architecture == "hybrid":
    book_id_to_label = {row["book_id"]:row["book_label"] for i, row in df.drop_duplicates("book_id").iterrows()}
    pretrained_book_embeddings = csv_embedding_to_dict("book_embedding_kaggle_64.csv")
```

Use GPU if possible

In [ ]:

```
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

Out[ ]:

```
'cuda'
```

Prepare torch dataloaders for the training

In [ ]:

```
class DataSet(data.Dataset):
    """ Base dataset for data loaders """
    def __init__(self, users, books, ratings):
        super(DataSet, self).__init__()
        self.users = torch.tensor(users, dtype=torch.long, device=device)
        self.items = torch.tensor(books, dtype=torch.long, device=device)
        self.ratings = torch.tensor(ratings, dtype=torch.float, device=device)

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

    def __getitem__(self, idx):
        return self.users[idx], self.items[idx], self.ratings[idx]
```

In [ ]:

```
batch_size = 256 # proposed in NCF paper + lower doesn't have better results, just trains longer (tried 32, 64, 1
28)

# split to train, validation, test datasets ... 70-20-10
if not os.path.exists(data_path + "train-df.csv"):
    train_df, valid_df, test_df = np.split(df.sample(frac=1), [int(.7 * df_len), int(.9 * df_len)])
    # session may be terminated, so to remember
    train_df.to_csv(data_path + "train-df.csv", index=False)
    valid_df.to_csv(data_path + "valid-df.csv", index=False)
    test_df.to_csv(data_path + "test-df.csv", index=False)
else:
    train_df = pd.read_csv(data_path + "train-df.csv")
    valid_df = pd.read_csv(data_path + "valid-df.csv")
    test_df = pd.read_csv(data_path + "test-df.csv")

# create datasets
train_dataset = DataSet(train_df["user_label"].values, train_df["book_label"].values, train_df["rating"].values)
valid_dataset = DataSet(valid_df["user_label"].values, valid_df["book_label"].values, valid_df["rating"].values)
test_dataset = DataSet(test_df["user_label"].values, test_df["book_label"].values, test_df["rating"].values)

# create dataloaders
train_dataloader = data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_dataloader = data.DataLoader(valid_dataset, batch_size=batch_size, shuffle=True)
```

Model - definition and training

In [ ]:

```
# model path for saving for given architecture
model_path = "{}{}-model".format(data_path, architecture)
```

In [ ]:

```
# inspired by https://github.com/guoyang9/NCF/blob/master/model.py
class Model(nn.Module):
    def __init__(self, user_num, book_num, embedding_dim=32, num_of_layers=3):
        super(Model, self).__init__()

        # baseline MF embeddings
        self.embed_user_GMF = nn.Embedding(user_num, embedding_dim)
        self.embed_book_GMF = nn.Embedding(book_num, embedding_dim)
        if architecture == "MF":
            self.create_predict_layer_and_init_weights(embedding_dim)
            return

        # added MLP branch of NCF architecture
        embed_mul_lambda = lambda x: 2 ** (num_of_layers - x)
        self.embed_user_MLP = nn.Embedding(user_num, embedding_dim * embed_mul_lambda(1))
        self.embed_book_MLP = nn.Embedding(book_num, embedding_dim * embed_mul_lambda(1))

        MLP_layers = []
        dropouts = [0.5] + [0.3] * (num_of_layers - 1)
        for i in range(num_of_layers):
            dim = embedding_dim * embed_mul_lambda(i)
            MLP_layers.append(nn.Dropout(p=dropouts[i]))
            MLP_layers.append(nn.Linear(dim, dim // 2))
            MLP_layers.append(nn.ReLU())

        self.MLP_layers = nn.Sequential(*MLP_layers)
        if architecture == "NCF":
            self.create_predict_layer_and_init_weights(2 * embedding_dim) # GMF + MLP concatenates -> twice the l
ength
            return

        # to NeuMF is concatenated pretrained book embedding of the same size
        self.create_predict_layer_and_init_weights(4 * embedding_dim)

    def create_predict_layer_and_init_weights(self, dimension):
        """ Helper method for creating last prediction layer and
            initializing weights as different architectures are supported.
        """

        self.predict_layer = nn.Linear(dimension, 1)
        self.init_weights()

    def init_weights(self):
        """ Initializes model according to original NCF paper. """

        # Xavier for prediction - in paper they used kaiming but we don't have sigmoid activation
        nn.init.xavier_uniform_(self.predict_layer.weight)

        # embeddings from normal distribution
        standard_deviation = 0.01
        nn.init.normal_(self.embed_user_GMF.weight, std=standard_deviation)
        nn.init.normal_(self.embed_book_GMF.weight, std=standard_deviation)
        if architecture == "MF":
            return

        nn.init.normal_(self.embed_user_MLP.weight, std=standard_deviation)
        nn.init.normal_(self.embed_book_MLP.weight, std=standard_deviation)

        # use Xavier for the MLP network
        for layer in [x for x in self.MLP_layers if isinstance(x, nn.Linear)]:
            nn.init.xavier_uniform_(layer.weight)

    def forward(self, users, books):
        """ Implementation of pytorch nn.Module forward method == computation. """

        output_GMF = self.embed_user_GMF(users) * self.embed_book_GMF(books)
        if architecture == "MF":
            return self.predict_layer(output_GMF).view(-1)

        output_MLP = self.MLP_layers(torch.cat([self.embed_user_MLP(users), self.embed_book_MLP(books)], -1))
        if architecture == "NCF":
            return self.predict_layer(torch.cat((output_GMF, output_MLP), -1)).view(-1)

        pretrained_embeddings = [pretrained_book_embeddings[label] for label in books.tolist()]
        pretrained_embeddings = torch.tensor(pretrained_embeddings, device=device)
        return self.predict_layer(torch.cat((pretrained_embeddings, output_GMF, output_MLP), -1)).view(-1)
```

In [ ]:

```
class ModelTrainer:
    """ Class responsible for training the model. """
```

```python
    def __init__(self, model, train_dataloader, valid_dataloader):
        self.model = model
        self.train_data = train_dataloader
        self.valid_data = valid_dataloader
        self.batch_iters = {"Train": len(train_dataloader), "Valid": len(valid_dataloader)}
        self.epochs = 1
        self.loss_values = {"Train": [], "Valid": []}
        self.best_loss = 1e6
        self.criterion = nn.MSELoss()
        self.optimizer = optim.Adam(model.parameters(), lr=1e-3)

    def train(self, epochs=6):
        """ Standard model training. In each batch are updated statistics.
            At the end of each epoch the current model is saved and validation run.
        """

        self.epochs = epochs
        for epoch in range(1, epochs + 1):
            self.model.train()

            # Adam overfits extremely quickly here (almost done after first epoch) -> SGD to slow down
            if epoch == 2:
                self.optimizer = optim.SGD(model.parameters(), lr=5e-4)

            loss_sum = 0
            for users, books, ratings in self.train_data:
                self.optimizer.zero_grad()

                predictions = self.model(users, books)
                loss = self.criterion(predictions, ratings)
                loss_sum += loss.item()

                loss.backward()
                self.optimizer.step()

            self.eval_epoch(loss_sum, "Train", epoch)
            self.validate(epoch)

    def validate(self, epoch):
        """ Validates the model after each epoch on validation dataset. """

        self.model.eval()
        loss_sum = 0
        for users, books, ratings in self.valid_data:
            predictions = self.model(users, books)
            loss = self.criterion(predictions, ratings)
            loss_sum += loss.item()

        self.eval_epoch(loss_sum, "Valid", epoch)

    def eval_epoch(self, loss_sum, phase, epoch):
        """ Helper method for finalizing and printing epoch statistics. """

        data = self.train_data if phase == "Train" else self.valid_data
        count = len(data.dataset.items)
        loss = loss_sum / self.batch_iters[phase]
        self.loss_values[phase].append(loss) # update for plot

        # save the best in case we overtrain - quite fast in collaborative filtering
        if phase == "Valid" and loss < self.best_loss:
            self.best_loss = loss
            self.save_model(model_path)

        print_stats = [phase, epoch, self.epochs, loss]
        print("{}: Epoch: [{}/{}]  Loss: {:.6f} ".format(*print_stats))

    def plot_loss(self):
        """ Plots loss during training and validation. """

        plt.plot(self.loss_values["Train"], label = "Train")
        plt.plot(self.loss_values["Valid"], label = "Val")
        plt.legend()
        plt.show()

    def save_model(self, location="model"):
        """ Saves model to specified location. """

        torch.save(self.model.state_dict(), location)
```

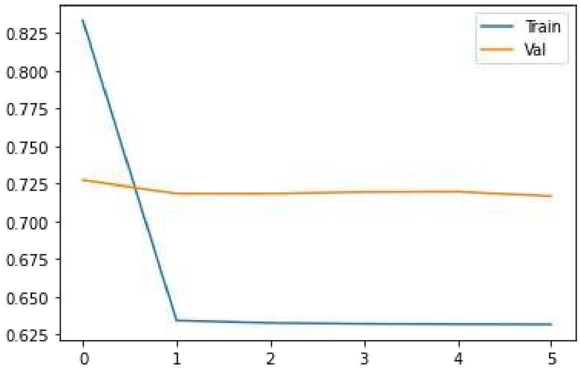
In [ ]:

```
model = Model(num_users, num_books).to(device)
trainer = ModelTrainer(model, train_dataloader, valid_dataloader)
trainer.train()
```

```
Train: Epoch: [1/6]  Loss: 0.833265
Valid: Epoch: [1/6]  Loss: 0.727301
Train: Epoch: [2/6]  Loss: 0.634055
Valid: Epoch: [2/6]  Loss: 0.718427
Train: Epoch: [3/6]  Loss: 0.632417
Valid: Epoch: [3/6]  Loss: 0.718393
Train: Epoch: [4/6]  Loss: 0.631808
Valid: Epoch: [4/6]  Loss: 0.719452
Train: Epoch: [5/6]  Loss: 0.631551
Valid: Epoch: [5/6]  Loss: 0.719691
Train: Epoch: [6/6]  Loss: 0.631422
Valid: Epoch: [6/6]  Loss: 0.716689
```

In [ ]:

```
trainer.plot_loss()
```



Evaluate the model on the test dataset

In [ ]:

```
class ModelTester:
    """ Loads a trained model and runs it against test dataset. """

    def __init__(self, model, location):
        self.model = model.to(device)
        self.model.load_state_dict(torch.load(location))
        self.model.eval()
        self.criterion = nn.MSELoss()
        self.stats = {}
        self.predictions = []

    def test(self, test_dataset, test_df):
        """ Tests the model against given data """

        self.stats = {"loss": 0, "hits": 0}
        self.predictions = []

        for user, book, rating in test_dataset:
            # make prediction
            prediction = self.model(torch.reshape(user, (-1,)), torch.reshape(book, (-1,)))
            self.predictions.append(prediction.item())

            # compute statistics
            self.stats["loss"] += self.criterion(prediction.squeeze(), rating).item()
            if rating == round(prediction.item()):
                self.stats["hits"] += 1

        # print results
        count = len(test_dataset.items)
        print("Test dataset metrics: ")
        print("Loss: {:.6f}".format(self.stats["loss"] / count  ))
        print("Hit acc: {:.3f} %".format(100 * self.stats["hits"] / count))
```

In [ ]:

```
tester = ModelTester(Model(num_users, num_books), model_path)
tester.test(test_dataset, test_df)
```

```
Test dataset metrics:
Loss: 0.713658
Hit acc: 46.030 %
```