Published in Towards Data Science

Mauro Di Pietro Follow

Apr 16 · 14 min read · · Listen

Image by author

Modern Recommendation Systems with Neural Networks

Build hybrid models with Python & TensorFlow

Summary

In this article, I will show how to build modern Recommendation Systems with Neural Networks, using Python and TensorFlow.

Photo by Alexander Shatov on Unsplash

Recommendation Systems are models that predict users' preferences over multiple products. They are used in a variety of areas, like video and music services, e-commerce, and social media platforms.

The most common methods leverage product features (Content-Based), user similarity (Collaborative Filtering), personal information (Knowledge-Based). However, with the increasing popularity of Neural Networks, companies have started experimenting with new hybrid Recommendation Systems that combine them all.

In this tutorial, I'm going to show how to use traditional models and how to build a modern Recommendation System from scratch. I will present some useful Python code that can be easily applied in other similar cases (just copy, paste, run) and walk through every line of code with comments so that you can replicate this example (link to the full code below).

DataScience_ArtificialIntelligence_Utils/example_recommendation.ipynb at master ·...

Examples of Data Science projects and Artificial Intelligence use cases ...

github.com

I will use the **MovieLens** dataset that contains thousands of movies rated by hundreds of users, created by GroupLens Research (link below).

MovieLens Latest Datasets

These datasets will change over time, and are not appropriate for reporting research results. We will keep the download...

grouplens.org

In particular, I will go through:

- Setup: import packages, read data, preprocessing
- Cold Start problem
- Content-Based methods with *tensorflow* and *numpy*
- Traditional Collaborative Filtering and Neural Collaborative Filtering with *tensorflow/keras*
- Hybrid (context-aware) model with *tensorflow/keras*

Setup

First of all, I shall import the following **packages**:

```
## for data
import pandas as pd
```

```
import numpy as np
import re
from datetime import datetime

## for plotting
import matplotlib.pyplot as plt
import seaborn as sns

## for machine learning
from sklearn import metrics, preprocessing

## for deep learning
from tensorflow.keras import models, layers, utils  #(2.6.0)
```

Then I'm gonna read the **data**, both product data (movies in this case) and user data.

```
dtf_products = pd.read_excel("data_movies.xlsx",
sheet_name="products")
```

Features

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy
...	...	...	...
9737	193581	Black Butler: Book of the Atlantic (2017)	Action|Animation|Comedy|Fantasy
9738	193583	No Game No Life: Zero (2017)	Animation|Comedy|Fantasy
9739	193585	Flint (2017)	Drama
9740	193587	Bungo Stray Dogs: Dead Apple (2018)	Action|Animation
9741	193609	Andrew Dice Clay: Dice Rules (1991)	Comedy

9742 rows × 3 columns

Image by author

In the product table, every row represents an item and the two columns on the right contain its features, which are static (you can see it as movie metadata). Let's read user data:

```
dtf_users = pd.read_excel("data_movies.xlsx",
sheet_name="users").head(10000)
```

Target variable Context

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931
...	...	...	...	...
9995	66	248	3.0	1113190892
9996	66	255	0.5	1113188840
9997	66	260	2.5	1093747550
9998	66	272	3.5	1113190319
9999	66	273	3.5	1113190315

10000 rows × 4 columns

Image by author

Every row of this other table is a pair of user-product and shows the rating that users have given to products, which is the **target variable**. Obviously, not every user has seen all the products. In fact, that is why we need Recommendation Systems. They have to predict what kind of rating a user would give to a new product, and if the predicted rating is high/positive then it is recommended. Moreover, here there are also pieces of information regarding the context of the target variable (when the user gave the rating).

Let's do some **data cleaning** and **feature engineering** to understand better what we have and how we can use it.

```
# Products
dtf_products = dtf_products[~dtf_products["genres"].isna()]

dtf_products["product"] = range(0,len(dtf_products))

dtf_products["name"] = dtf_products["title"].apply(lambda x: re.sub("
[\(\[].*?[\)\]]", "", x).strip())

dtf_products["date"] = dtf_products["title"].apply(lambda x:
int(x.split("(")[-1].replace(")","").strip())
if "(" in x else np.nan)

dtf_products["date"] = dtf_products["date"].fillna(9999)
dtf_products["old"] = dtf_products["date"].apply(lambda x: 1 if x <
2000 else 0)


# Users
dtf_users["user"] = dtf_users["userId"].apply(lambda x: x-1)

dtf_users["timestamp"] = dtf_users["timestamp"].apply(lambda x:
datetime.fromtimestamp(x))

dtf_users["daytime"] = dtf_users["timestamp"].apply(lambda x: 1 if
6<int(x.strftime("%H"))<20 else 0)

dtf_users["weekend"] = dtf_users["timestamp"].apply(lambda x: 1 if
x.weekday() in [5,6] else 0)

dtf_users = dtf_users.merge(dtf_products[["movieId","product"]],
how="left")

dtf_users = dtf_users.rename(columns={"rating":"y"})


# Clean
dtf_products =
dtf_products[["product","name","old","genres"]].set_index("product")

dtf_users =
dtf_users[["user","product","daytime","weekend","y"]]
```

product	name	old	genres
0	Toy Story	1	Adventure|Animation|Children|Comedy|Fantasy
1	Jumanji	1	Adventure|Children|Fantasy
2	Grumpier Old Men	1	Comedy|Romance
3	Waiting to Exhale	1	Comedy|Drama|Romance
4	Father of the Bride Part II	1	Comedy

	user	product	daytime	weekend	y
0	0	0	0	1	4.0
1	0	2	0	1	4.0
2	0	5	0	1	4.0
3	0	43	0	1	5.0
4	0	46	0	1	5.0

Image by author

Please note that I extracted 2 context variables from the *timestamp* column: *daytime* and *weekend*. I shall save them into a dataframe as we might need them later.

```
dtf_context = dtf_users[["user","product","daytime","weekend"]]
```

Regarding the products, the next step is to create the *Products-Features* matrix:

```
tags = [i.split("|") for i in dtf_products["genres"].unique()]
columns = list(set([i for lst in tags for i in lst]))
columns.remove('(no genres listed)')

for col in columns:
    dtf_products[col] = dtf_products["genres"].apply(lambda x: 1 if
col in x else 0)
```

product	name	old	genres	Thriller	Horror	Musical	Adventure	Action	Mystery	Romance	...	Drama	Comedy
0	Toy Story	1	Adventure|Animation|Children|Comedy|Fantasy	0	0	0	1	0	0	0	...	0	1
1	Jumanji	1	Adventure|Children|Fantasy	0	0	0	1	0	0	0	...	0	0
2	Grumpier Old Men	1	Comedy|Romance	0	0	0	0	0	0	1	...	0	1
3	Waiting to Exhale	1	Comedy|Drama|Romance	0	0	0	0	0	0	1	...	1	1
4	Father of the Bride Part II	1	Comedy	0	0	0	0	0	0	0	...	0	1

Image by author

The matrix is sparse as most of the products don't have all the features. Let's visualize it to understand better the situation.

```
fig, ax = plt.subplots(figsize=(20,5))
sns.heatmap(dtf_products==0, vmin=0, vmax=1, cbar=False,
ax=ax).set_title("Products x Features")
plt.show()
```

Image by author

The sparsity gets even worse with the *Users-Products* matrix:

```
tmp = dtf_users.copy()
dtf_users = tmp.pivot_table(index="user", columns="product",
values="y")
missing_cols = list(set(dtf_products.index) - set(dtf_users.columns))
for col in missing_cols:
    dtf_users[col] = np.nan
dtf_users = dtf_users[sorted(dtf_users.columns)]
```

product	0	1	2	3	4	5	6	7	8	9	...	9731	9732	9733	9734	9735	9736	9737	9738	9739	9740
user																					
0	4.0	NaN	4.0	NaN	NaN	4.0	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
61	NaN	4.0	NaN	NaN	NaN	4.5	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
62	5.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	3.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
63	4.0	NaN	3.5	NaN	NaN	4.5	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
64	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
65	4.0	NaN	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Image by author

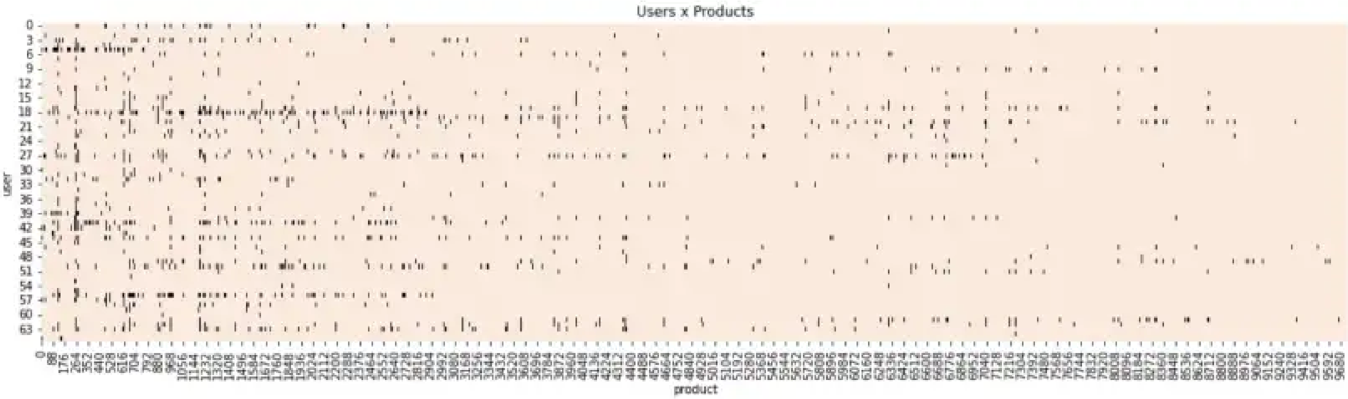


Image by author

The last step before digging into the models is **Preprocessing**. Since we will deal with Neural Networks, it's always good practice to scale the data.

```
dtf_users = pd.DataFrame(preprocessing.MinMaxScaler(feature_range=
(0.5,1)).fit_transform(dtf_users.values),
columns=dtf_users.columns, index=dtf_users.index)
```

product	0	1	2	3	4	5	6	7	8	9	...	9731	9732	9733	9734	9735	9736	9737	9738	9739	9740
user																					
0	0.8	NaN	0.750	NaN	NaN	0.750	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	0.8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
61	NaN	0.833333	NaN	NaN	NaN	0.875	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
62	1.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.666667	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
63	0.8	NaN	0.625	NaN	NaN	0.875	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
64	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
65	0.8	NaN	NaN	NaN	0.75	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

66 rows × 9741 columns

Image by author

Finally, we shall partition the data into *train* and *test* sets. I'm going to split the dataset vertically, such that all the users will be in both *train* and *test,* while 80% of the products are kept for training and 20% for testing. Like this:

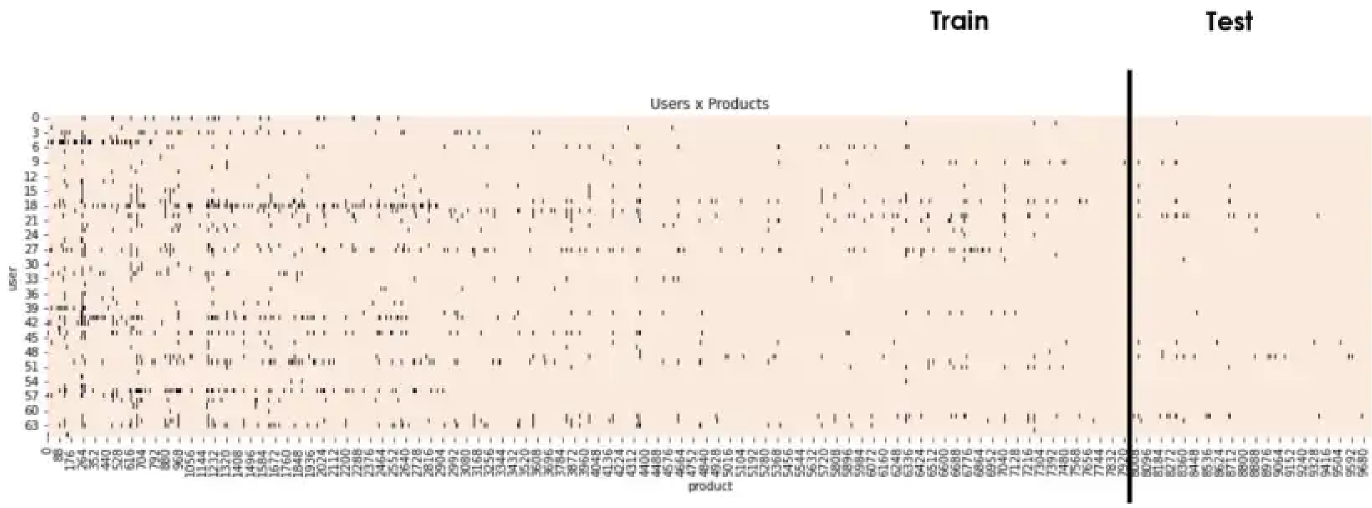


Image by author

```
split = int(0.8*dtf_users.shape[1])
dtf_train = dtf_users.loc[:, :split-1]
dtf_test = dtf_users.loc[:, split:]
```

Okay, now we can start… maybe.

Cold Start

Imagine owning a brand new app similar to Netflix and the first user subscribes. We need to be able to offer recommendations without depending on the user's previous interactions, as none have been recorded yet. When a user (or a product) is new, we have the **Cold Start problem**. The system is unable to form any relation between users and products because it doesn't have enough data.

In order to solve the problem, the primary technique is the **Knowledge-Based approach**: for example, asking for user's preferences in order to create an initial profile, or using demographic information (i.e. high school shows for teenagers and cartoons for kids).

If there are only a few users, one could work with Content-Based methods. Then, when we have enough ratings (i.e. at least 10 products per user and more than 100 total users), more complex models can be applied.

Content-Based

Content-Based methods are based on the product contents. For instance, if *User A* likes *Product 1,* and *Product 2* is similar to *Product 1,* then *User A* would probably like *Product 2* as well. Two products are similar if they have similar features.

In a nutshell, the idea is that users actually rate the features of the product and not the product itself. To put it in another way, if I like products related to music and art, it's because I like those features (music and art). Based on that, we can estimate how much I would like other products with the same features. This method is best suited for situations where there are known data on products but not on users.

Image by author

Let's pick one user from the data as an example of our first subscriber that has now used enough products, and let's create the *train* and *test* vectors.

```
# Select a user
i = 1
train = dtf_train.iloc[i].to_frame(name="y")
test = dtf_test.iloc[i].to_frame(name="y")

# add all the test products but hide the y
tmp = test.copy()
tmp["y"] = np.nan
train = train.append(tmp)
```

Now we need to estimate the weights that the user gives to each feature. We have the *User-Products* vector and the *Products-Features* matrix.

```
# shapes
usr = train[["y"]].fillna(0).values.T
prd = dtf_products.drop(["name","genres"],axis=1).values
print("Users", usr.shape, " x  Products", prd.shape)
```

```
Users (1, 9741)  x  Products (9741, 20)
```

By multiplying those 2 objects, we obtain a *User-Features* vector containing the estimated weights that this user gives to each feature. Those weights shall be re-applied to the *Products-Features* matrix in order to get the predicted ratings.

```
# usr_ft(users,fatures) = usr(users,products) x prd(products,features)
usr_ft = np.dot(usr, prd)

# normalize
weights = usr_ft / usr_ft.sum()

# predicted rating(users,products) = weights(users,fatures) x
prd.T(features,products)
pred = np.dot(weights, prd.T)

test = test.merge(pd.DataFrame(pred[0], columns=["yhat"]), how="left",
left_index=True, right_index=True).reset_index()
test = test[~test["y"].isna()]
test
```

	product	y	yhat
271	8063	0.812500	0.364068
513	8305	1.000000	0.441084
584	8376	0.777778	0.077697
674	8466	0.800000	0.215465
717	8509	0.500000	0.477922
758	8550	0.875000	0.356925
889	8681	1.000000	0.327055
1036	8828	0.500000	0.033233

Image by author

As you can see, I developed this easy approach using simply *numpy*. One can do the same by using just raw *tensorflow* as well:

```
import tensorflow as tf

# usr_ft(users,fatures) = usr(users,products) x prd(products,features)
usr_ft = tf.matmul(usr, prd)

# normalize
weights = usr_ft / tf.reduce_sum(usr_ft, axis=1, keepdims=True)

# rating(users,products) = weights(users,fatures) x
prd.T(features,products)
pred = tf.matmul(weights, prd.T)
```

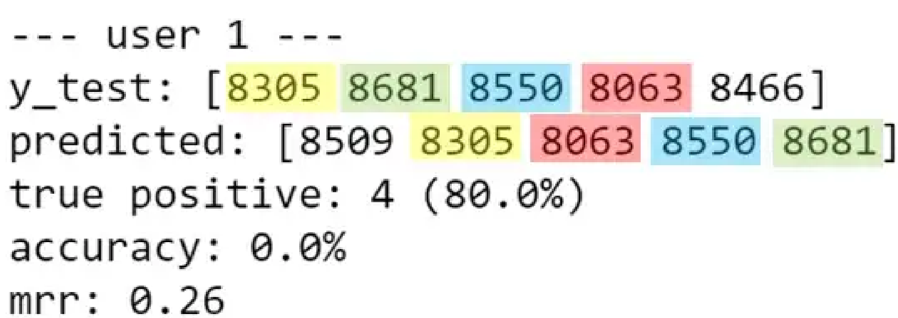
How to **evaluate** our predicted recommendations? I usually apply the Accuracy and the Mean Reciprocal Rank (MRR). The latter is a statistic measure for evaluating any list of possible responses ordered by the probability of correctness.

```
def mean_reciprocal_rank(y_test, predicted):
    score = []
    for product in y_test:
        mrr = 1 / (list(predicted).index(product) + 1) if product
        in predicted else 0
```

```
        score.append(mrr)
    return np.mean(score)
```

Please note that metrics change based on the number of products we are recommending. Since we are comparing our predicted *top k* items with the ones in the *test* set, also the order matters.

```
print("--- user", i, "---")

top = 5
y_test = test.sort_values("y", ascending=False)
["product"].values[:top]
print("y_test:", y_test)

predicted = test.sort_values("yhat", ascending=False)
["product"].values[:top]
print("predicted:", predicted)

true_positive = len(list(set(y_test) & set(predicted)))
print("true positive:", true_positive, "
("+str(round(true_positive/top*100,1))+"%)")
print("accuracy:",
str(round(metrics.accuracy_score(y_test,predicted)*100,1))+"%")
print("mrr:", mean_reciprocal_rank(y_test, predicted))
```



```
--- user 1 ---
y_test: [8305 8681 8550 8063 8466]
predicted: [8509 8305 8063 8550 8681]
true positive: 4 (80.0%)
accuracy: 0.0%
mrr: 0.26
```

Image by author

We got 4 products right, but the order doesn't match. That's why Accuracy and MRR are low.

```
# See predictions details
test.merge(
       dtf_products[["name","old","genres"]], left_on="product",
```

```
        right_index=True
    ).sort_values("yhat", ascending=False)
```

product	y	yhat	name	date	genres
8509	0.500000	0.535976	The Drop	2014	Crime|Drama|Thriller
8305	1.000000	0.494663	Wolf of Wall Street, The	2013	Comedy|Crime|Drama
8063	0.812500	0.408292	Django Unchained	2012	Action|Drama|Western
8550	0.875000	0.400281	Ex Machina	2015	Drama|Sci-Fi|Thriller
8681	1.000000	0.366783	Mad Max: Fury Road	2015	Action|Adventure|Sci-Fi|Thriller
8466	0.800000	0.241638	Whiplash	2014	Drama
8828	0.500000	0.037270	The Jinx: The Life and Deaths of Robert Durst	2015	Documentary
8376	0.777778	0.022086	Interstellar	2014	Sci-Fi|IMAX

Image by author

Collaborative Filtering

Collaborative Filtering is based on the assumption that similar users like similar products. For instance, if *User A* likes *Product 1*, and *User B* is similar to *User A*, then *User B* would probably like *Product 1* as well. Two users are similar if they like similar products.

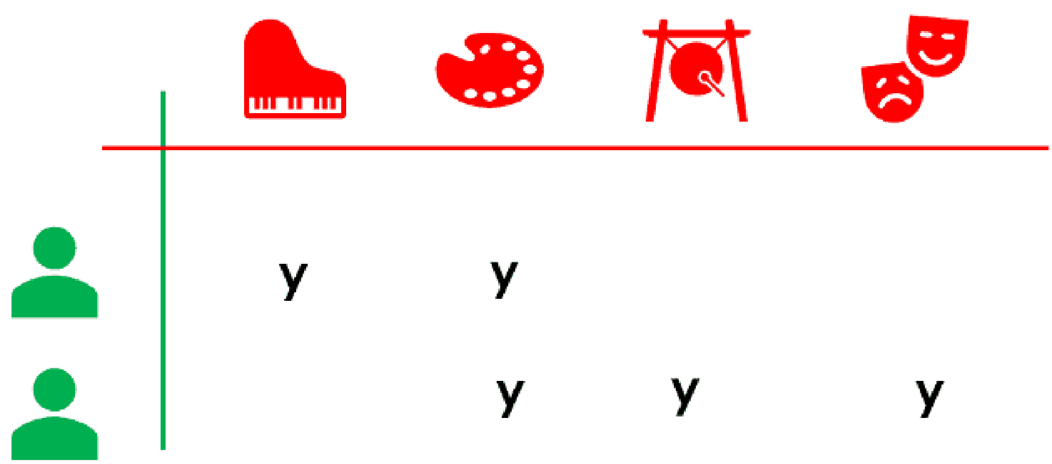


Image by author

This method doesn't need product features to work, it requires many ratings from many users instead. To continue the example of our platform, imagine that our first subscriber is not alone anymore and we have enough users to apply this model.

Collaborative Filtering gained its popularity when Netflix held an open competition (2009) for the best algorithm and people came up with several implementations. They can be grouped into 2 families:

- **Memory-based —** find similar users with correlation metrics, cosine similarity, and clustering.
- **Model-based —** predict how users would rate a certain product by applying supervised machine learning and matrix factorization, which splits the large *Users-Products* matrix into 2 smaller factors representing the *Users* matrix and the *Products* matrix.

In Python, the most user-friendly package is *surprise*, a simple library for building and analyzing recommender systems with explicit rating data (similar to *scikit-learn*). It can be used for both Memory-based approaches as well as Model-based. Alternatively, one can use *tensorflow/keras* to create embeddings for a more sophisticated Model-based approach, which is exactly what I'm going to do.

First of all, we need to have data in the following form:

```
train = dtf_train.stack(dropna=True).reset_index().rename(columns=
{0:"y"})
train.head()
```

	user	product	y
0	0	0	0.80
1	0	2	0.75
2	0	5	0.75
3	0	43	1.00
4	0	46	1.00

Image by author (do the same for the Test set)

The main idea is to leverage the Embedding layer of a Neural Network to create the *Users* and *Products* matrices. It's important to understand that the inputs are user-product pairs and the output is the rating. When predicting a new pair of user-product, the model is going to lookup the user in the *Users* embedding space and the product in the *Products* space. For that reason, you need to specify in advance the total number of users and products.

```
embeddings_size = 50
usr, prd = dtf_users.shape[0], dtf_users.shape[1]


# Users (1,embedding_size)
xusers_in = layers.Input(name="xusers_in", shape=(1,))

xusers_emb = layers.Embedding(name="xusers_emb", input_dim=usr,
output_dim=embeddings_size)(xusers_in)

xusers = layers.Reshape(name='xusers', target_shape=
(embeddings_size,))(xusers_emb)


# Products (1,embedding_size)
xproducts_in = layers.Input(name="xproducts_in", shape=(1,))

xproducts_emb = layers.Embedding(name="xproducts_emb", input_dim=prd,
output_dim=embeddings_size)(xproducts_in)

xproducts = layers.Reshape(name='xproducts', target_shape=
(embeddings_size,))(xproducts_emb)


# Product (1)
xx = layers.Dot(name='xx', normalize=True, axes=1)([xusers,
xproducts])


# Predict ratings (1)
y_out = layers.Dense(name="y_out", units=1, activation='linear')(xx)


# Compile
model = models.Model(inputs=[xusers_in,xproducts_in], outputs=y_out,
name="CollaborativeFiltering")
model.compile(optimizer='adam', loss='mean_absolute_error', metrics=
['mean_absolute_percentage_error'])
```

Please note that I'm treating this use case as a regression problem by using the Mean Absolute Error as the loss, even if after all we won't need the score itself but the sorting of the predicted products.

```
utils.plot_model(model, to_file='model.png', show_shapes=True,
show_layer_names=True)
```

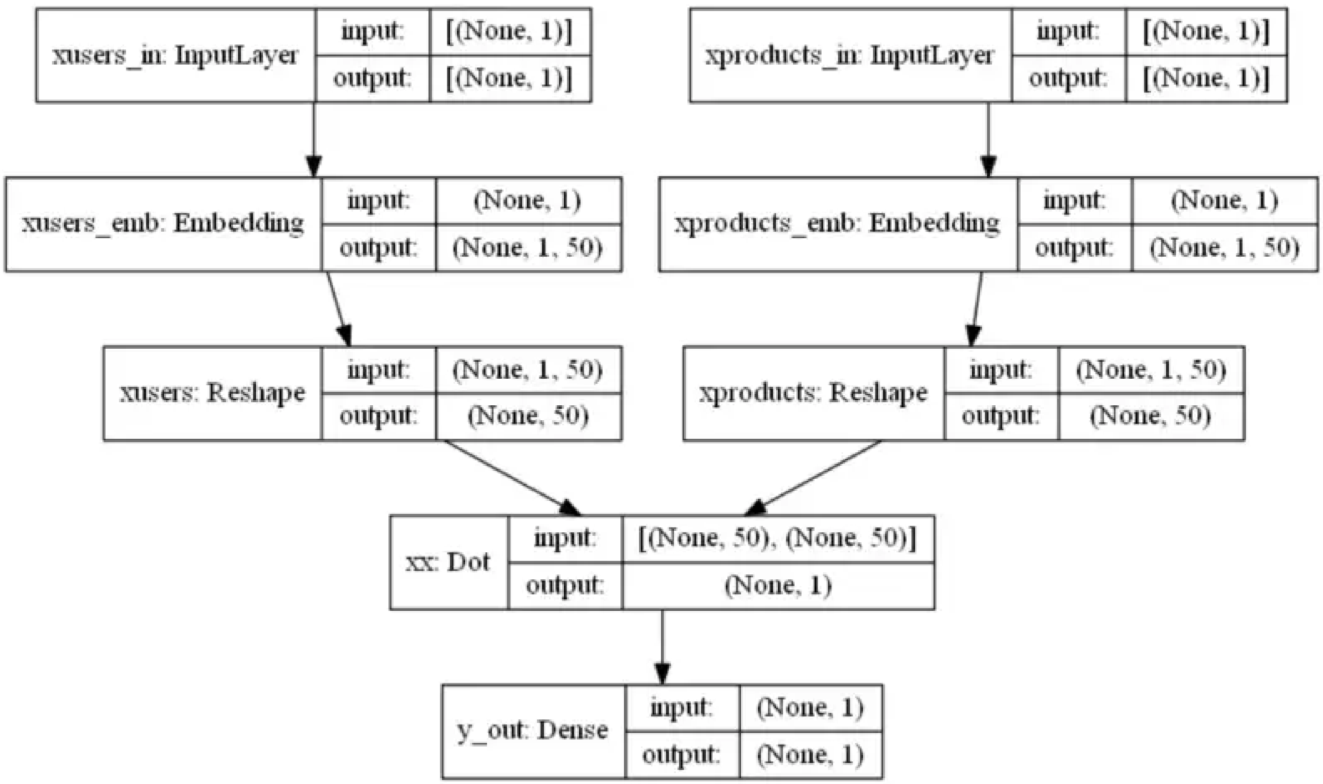


Image by author

Let's **train and test** the model.

```
# Train
training = model.fit(x=[train["user"], train["product"]],
y=train["y"], epochs=100, batch_size=128, shuffle=True, verbose=0,
validation_split=0.3)

model = training.model

# Test
test["yhat"] = model.predict([test["user"], test["product"]])
test
```

user	product	y	yhat
1	8063	0.812500	0.770686
1	8305	1.000000	0.654975
1	8376	0.777778	0.635748
1	8466	0.800000	0.745632
1	8509	0.500000	0.726420
...	...	...	...
64	8023	0.500000	0.749698
64	8376	0.944444	0.492666
64	8438	0.666667	0.592872
64	8569	0.900000	0.440547
64	8691	0.777778	0.587904

Image by author

We can evaluate the predictions by comparing the recommendations generated for our beloved first user (same code as before):

```
--- user 1 ---
y_test: [8305 8681 8550 8063 8466]
predicted: [8828 8063 8466 8509 8305]
true positive: 3 (60.0%)
accuracy: 0.0%
mrr: 0.21
```

Image by author

Currently, all the state-of-the-art Recommendation Systems leverage deep learning. In particular, **Neural Collaborative Filtering** (2017) combines non-linearity from Neural Networks and Matrix Factorization. The model is designed to make the most out of the Embedding space by using it not only for the traditional Collaborative Filtering, but also for a fully connected Deep Neural Network. The additional part should capture patterns and features that the Matrix Factorization might miss.

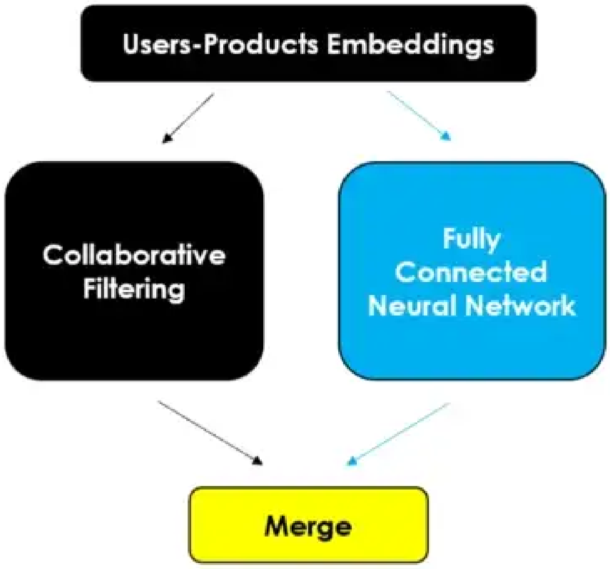
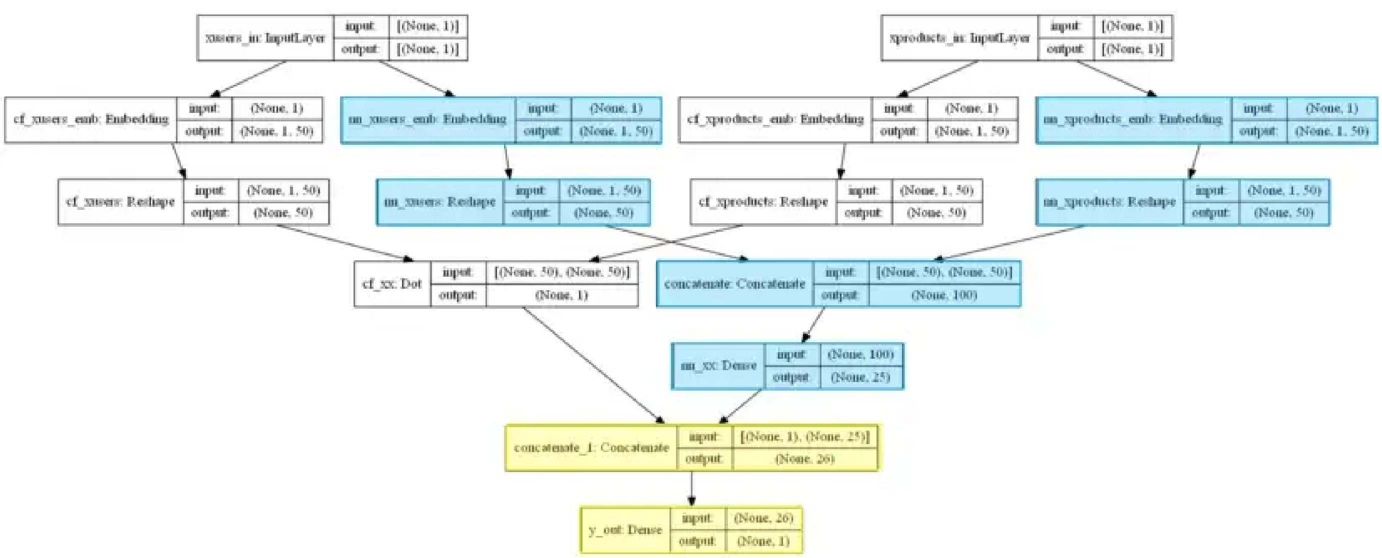


Image by author

In Python terms:

```
embeddings_size = 50
usr, prd = dtf_users.shape[0], dtf_users.shape[1]

# Input layer
xusers_in = layers.Input(name="xusers_in", shape=(1,))
xproducts_in = layers.Input(name="xproducts_in", shape=(1,))


# A) Matrix Factorization
## embeddings and reshape
cf_xusers_emb = layers.Embedding(name="cf_xusers_emb", input_dim=usr,
output_dim=embeddings_size)(xusers_in)
cf_xusers = layers.Reshape(name='cf_xusers', target_shape=
(embeddings_size,))(cf_xusers_emb)

## embeddings and reshape
cf_xproducts_emb = layers.Embedding(name="cf_xproducts_emb",
input_dim=prd, output_dim=embeddings_size)(xproducts_in)
cf_xproducts = layers.Reshape(name='cf_xproducts', target_shape=
(embeddings_size,))(cf_xproducts_emb)

## product
cf_xx = layers.Dot(name='cf_xx', normalize=True, axes=1)([cf_xusers,
cf_xproducts])


# B) Neural Network
## embeddings and reshape
nn_xusers_emb = layers.Embedding(name="nn_xusers_emb", input_dim=usr,
```

```
output_dim=embeddings_size)(xusers_in)
nn_xusers = layers.Reshape(name='nn_xusers', target_shape=
(embeddings_size,))(nn_xusers_emb)

## embeddings and reshape
nn_xproducts_emb = layers.Embedding(name="nn_xproducts_emb",
input_dim=prd, output_dim=embeddings_size)(xproducts_in)
nn_xproducts = layers.Reshape(name='nn_xproducts', target_shape=
(embeddings_size,))(nn_xproducts_emb)

## concat and dense
nn_xx = layers.Concatenate()([nn_xusers, nn_xproducts])
nn_xx = layers.Dense(name="nn_xx", units=int(embeddings_size/2),
activation='relu')(nn_xx)


# Merge A & B
y_out = layers.Concatenate()([cf_xx, nn_xx])
y_out = layers.Dense(name="y_out", units=1, activation='linear')
(y_out)

# Compile
model = models.Model(inputs=[xusers_in,xproducts_in], outputs=y_out,
name="Neural_CollaborativeFiltering")
model.compile(optimizer='adam', loss='mean_absolute_error', metrics=
['mean_absolute_percentage_error'])
```



utils.plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)

You can run it using the same code as before and check whether it performs better than the traditional Collaborative Filtering.

```
--- user 1 ---
y_test: [8305 8681 8550 8063 8466]
predicted: [8828 8681 8550 8466 8305]
true positive: 4 (80.0%)
accuracy: 40.0%
mrr: 0.26
```

Image by author

Hybrid Model

Let's start with a recap of what kind of data the real world offers:

- **Target variable** — ratings can be explicit (i.e. the user leaves feedback) or implicit (i.e. assuming positive feedback if the user watches the whole movie), anyway they are necessary.
- **Product features** — tags and descriptions of the items (i.e. movie genres), mostly used in the Content-Based methods.
- **User profile** — descriptive information about users can be demographics (i.e. gender and age) or behavioral (i.e. preferences, average time on screen, most frequent time of usage), mostly used for Knowledge-Based recommendations.
- **Context** — additional information regarding the situation around the rating (i.e. when, where, search history), often included in Knowledge-Based recommendations as well.

Modern Recommendation Systems combine them all when making a prediction about our taste. For instance, YouTube recommends the next video using everything Google knows about you, and they know a lot.

In this example, I have product features and data about when the user gave the rating, which I'm going to use as the context (alternatively it could be used to build a user profile).

```
features = dtf_products.drop(["genres","name"], axis=1).columns
print(features)
```

```
context = dtf_context.drop(["user","product"], axis=1).columns
print(context)
```

```
Index(['old', 'Mystery', 'Children', 'Comedy', 'Adventure', 'Thriller',
       'Drama', 'Horror', 'Action', 'IMAX', 'Fantasy', 'Western', 'War',
       'Sci-Fi', 'Film-Noir', 'Romance', 'Animation', 'Crime', 'Musical',
       'Documentary'],
      dtype='object')
Index(['daytime', 'weekend'], dtype='object')
```

Image by author

Let's add that extra information to the *train* set:

```
train = dtf_train.stack(dropna=True).reset_index().rename(columns=
{0:"y"})

## add features
train = train.merge(dtf_products[features], how="left",
left_on="product", right_index=True)

## add context
train = train.merge(dtf_context, how="left")
```

	user	product	y	old	Mystery	Children	Comedy	Adventure	Thriller	Drama	...	War	Sci-Fi	Film-Noir	Romance	Animation	Crime	Musical
0	0	0	0.80	1	0	1	1	1	0	0	...	0	0	0	0	1	0	0
1	0	2	0.75	1	0	0	1	0	0	0	...	0	0	0	1	0	0	0
2	0	5	0.75	1	0	0	0	0	1	0	...	0	0	0	0	0	1	0
3	0	43	1.00	1	1	0	0	0	1	0	...	0	0	0	0	0	0	0
4	0	46	1.00	1	1	0	0	0	1	0	...	0	0	0	0	0	1	0

5 rows × 25 columns

Image by author

Please note that you could do the same for the *test* set, but if you want to simulate real production you should insert a static value for the context. To put it in simple terms, if we are making predictions for a user of our platform on a Monday evening, the context variable shall be *daytime=0* and *weekend=0*.

Now we have all the ingredients to build a **context-aware hybrid model**. The flexibility of Neural Networks allows us to add anything we want, so I'm going to take the Neural Collaborative Filtering network structure and include as many modules as possible.

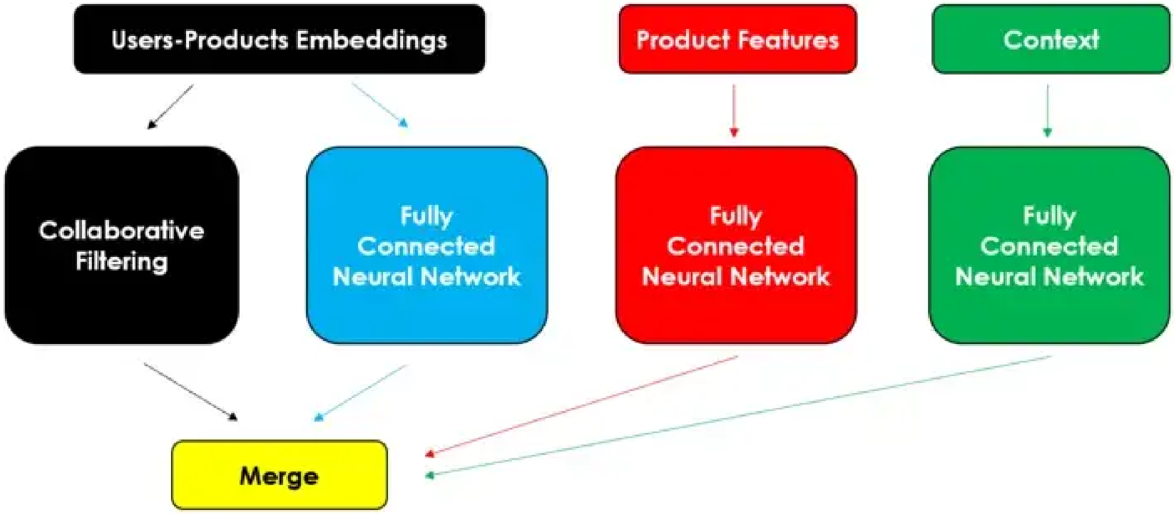
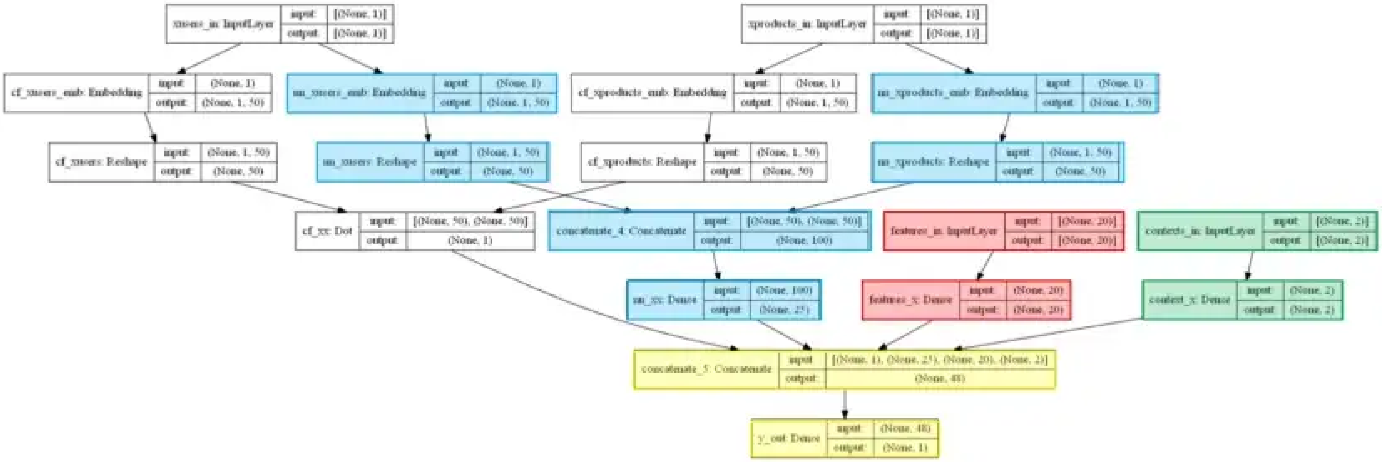


Image by author

Despite the code that might look difficult, we are just adding a few layers to what we've already used.

```
embeddings_size = 50
usr, prd = dtf_users.shape[0], dtf_users.shape[1]
feat = len(features)
ctx = len(context)

################### COLLABORATIVE FILTERING ########################
# Input layer
xusers_in = layers.Input(name="xusers_in", shape=(1,))
xproducts_in = layers.Input(name="xproducts_in", shape=(1,))

# A) Matrix Factorization
## embeddings and reshape
cf_xusers_emb = layers.Embedding(name="cf_xusers_emb", input_dim=usr,
output_dim=embeddings_size)(xusers_in)
cf_xusers = layers.Reshape(name='cf_xusers', target_shape=
(embeddings_size,))(cf_xusers_emb)

## embeddings and reshape
cf_xproducts_emb = layers.Embedding(name="cf_xproducts_emb",
input_dim=prd, output_dim=embeddings_size)(xproducts_in)
cf_xproducts = layers.Reshape(name='cf_xproducts', target_shape=
(embeddings_size,))(cf_xproducts_emb)

## product
cf_xx = layers.Dot(name='cf_xx', normalize=True, axes=1)([cf_xusers,
cf_xproducts])

# B) Neural Network
## embeddings and reshape
```

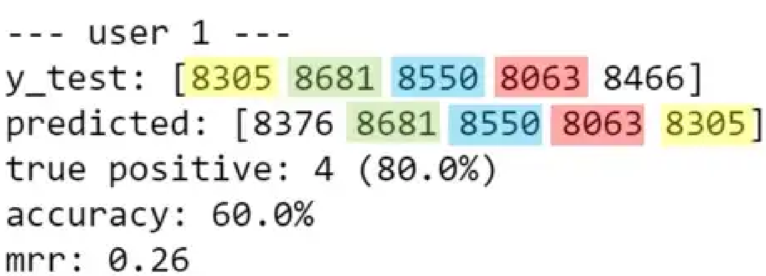
```
nn_xusers_emb = layers.Embedding(name="nn_xusers_emb", input_dim=usr,
output_dim=embeddings_size)(xusers_in)
nn_xusers = layers.Reshape(name='nn_xusers', target_shape=
(embeddings_size,))(nn_xusers_emb)

## embeddings and reshape
nn_xproducts_emb = layers.Embedding(name="nn_xproducts_emb",
input_dim=prd, output_dim=embeddings_size)(xproducts_in)
nn_xproducts = layers.Reshape(name='nn_xproducts', target_shape=
(embeddings_size,))(nn_xproducts_emb)

## concat and dense
nn_xx = layers.Concatenate()([nn_xusers, nn_xproducts])
nn_xx = layers.Dense(name="nn_xx", units=int(embeddings_size/2),
activation='relu')(nn_xx)


######################### CONTENT BASED ############################
# Product Features
features_in = layers.Input(name="features_in", shape=(feat,))
features_x = layers.Dense(name="features_x", units=feat,
activation='relu')(features_in)


######################## KNOWLEDGE BASED ###########################
# Context
contexts_in = layers.Input(name="contexts_in", shape=(ctx,))
context_x = layers.Dense(name="context_x", units=ctx,
activation='relu')(contexts_in)


########################## OUTPUT ##################################
# Merge all
y_out = layers.Concatenate()([cf_xx, nn_xx, features_x, context_x])
y_out = layers.Dense(name="y_out", units=1, activation='linear')
(y_out)

# Compile
model = models.Model(inputs=[xusers_in,xproducts_in, features_in,
contexts_in], outputs=y_out, name="Hybrid_Model")
model.compile(optimizer='adam', loss='mean_absolute_error', metrics=
['mean_absolute_percentage_error'])
```



utils.plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)

This hybrid model expects more inputs, so don't forget to feed in the new data as well:

```
# Train
training = model.fit(x=[train["user"], train["product"], 
train[features], train[context]], y=train["y"], 
                     epochs=100, batch_size=128, shuffle=True, 
verbose=0, validation_split=0.3)

model = training.model

# Test
test["yhat"] = model.predict([test["user"], test["product"], 
test[features], test[context]])
```



```
--- user 1 ---
y_test: [8305 8681 8550 8063 8466]
predicted: [8376 8681 8550 8063 8305]
true positive: 4 (80.0%)
accuracy: 60.0%
mrr: 0.26
```

Image by author

Compared to the other methods, for this specific user, the hybrid model got the highest Accuracy as three predicted products have matching orders.

Conclusion

This article has been a tutorial to demonstrate **how to design and build Recommendation Systems with Neural Networks.** We saw different use cases based on the data availability: applied a Content-based approach for a single-user scenario, and dived into Collaborative Filtering applications for multiple users-products. More importantly, we understood how to use Neural Networks to improve traditional techniques and build modern hybrid Recommendation Systems that can include context and any other additional information.

I hope you enjoyed it! Feel free to contact me for questions and feedback or just to share your interesting projects.

👉 Let's Connect 👈

> *This article is part of the series* ***Machine Learning with Python****, see also:*

Deep Learning with Python: Neural Networks (complete tutorial)

Build, Plot & Explain Artificial Neural Networks with TensorFlow

towardsdatascience.com

Machine Learning with Python: Classification (complete tutorial)

Data Analysis & Visualization, Feature Engineering & Selection, Model Design & Testing, Evaluation & Explainability

towardsdatascience.com

Machine Learning with Python: Regression (complete tutorial)

Data Analysis & Visualization, Feature Engineering & Selection, Model Design & Testing, Evaluation & Explainability towardsdatascience.com

Clustering Geospatial Data

Plot Machine Learning & Deep Learning Clustering with interactive Maps

towardsdatascience.com

Python Machine Learning Data Science Deep Learning Deep Dives

Thanks to Ludovic Benistant