# Recommendation system with python and ML

To build a recommender system using machine learning and Python, you can follow these steps:

1. Collect historical user interaction data with items. This data often includes information such as user identifiers, item identifiers, and user preference ratings for each item.
2. Pre-process the data to ensure it is clean and organized. For example, you might need to remove missing values, convert categorical data to numerical data, and normalize the data.
3. Split the data into a training set and a test set. The training set will be used to train the recommendation model, while the test set will be used to evaluate the performance of the model.
4. Train the recommendation model using a collaborative filtering technique such as k-Nearest Neighbors (k-NN), Singular Value Decomposition (SVD) or Factorization Machines (FM). scikit-learn is a popular library for modeling machine learning in Python.
5. Evaluate model performance using metrics such as root mean square error (RMSE) or top-K accuracy. This will allow you to adjust model parameters and improve its performance.
6. Use the trained model to make recommendation predictions for new users or items. For example, you can recommend the top rated items by the template to a new user.
7. Continue to update and improve the recommendation model as more data becomes available.

There are many Python machine learning libraries that can help you build a recommendation system, including scikit-learn, TensorFlow, PyTorch, and Apache Spark. To get started, it's a good idea to choose a library you're comfortable with and explore its functionality. In addition, you can find many tutorials and examples online to walk you through the process of building a recommender system using machine learning and Python.

## Data collection

Flow in the GeoFree:

User info (looker or giver):

* Email
* Password
* Localization (via ID)

User - looker:

* What are you looking for? Is it limited to categories?
* Where? District? Coordinates?

User – post = Item:

* Item Categories: All – Furniture – Clothes – Plants - Kids
* Condition: Poor – Good – Like new
* Localization (address) or mark it in the map (coordinates)

My concerning:

* The app has not enough info to make a recommendation system. I suggest include features
* Delay in make design decisions is causing inconsistences (DS/WD)
* If the feature were not added in the app because of bugs or miscommunication?

|  |  |
| --- | --- |
| item | user |
| item\_category  item\_postcode  item\_address  item\_lat  item\_lng  item\_id  item\_condition (1,2,3)  n\_stars  n\_reviews  date\_time\_posted  distance\_item\_user | user\_id  user\_lat  user\_lng  views  clicks  likes  preference about item category (asking or history?)  preference about item condition (asking or history?)  preference postcode (asking or history?)  preference district (asking or history?) |

**Legend:**

mandatory

optional

needs to include

My questions to DS mentor:

- How many records in the mock dataset?  
330 with complete address

1500 only coordinates

- Packaging python: is it needed? When?

Feature engineering  
- Remove nulls

- Convert categorical data in numerical data

import pandas as pd

# Create a pandas dataframe with categorical data

data = {'color': ['red', 'green', 'blue', 'red', 'green']}

df = pd.DataFrame(data)

# Convert categorical data into numerical data using get\_dummies() function

df = pd.get\_dummies(df, columns=['color'])

print(df)

color\_blue color\_green color\_red

0 0 0 1

1 0 1 0

2 1 0 0

3 0 0 1

4 0 1 0

**Using `factorize()` function**

import pandas as pd

# Create a pandas dataframe with categorical data

data = {'color': ['red', 'green', 'blue', 'red', 'green']}

df = pd.DataFrame(data)

# Convert categorical data into numerical data using factorize() function

df['color'] = pd.factorize(df['color'])[0]

print(df)

color

0 0

1 1

2 2

3 0

4 1

- Normalize data

import pandas as pd

ratings\_data = pd.read\_csv('ratings.csv')

user\_rating\_means = ratings\_data.groupby('userId')['rating'].mean()

user\_rating\_std = ratings\_data.groupby('userId')['rating'].std()

ratings\_data['rating\_normalized'] = ratings\_data.apply(lambda row: (row['rating'] - user\_rating\_means[row['userId']])/user\_rating\_std[row['userId']], axis=1)

Examples

* <https://www.datacamp.com/tutorial/recommender-systems-python>
* <https://www.analyticsvidhya.com/blog/2022/02/introduction-to-collaborative-filtering/>
* <https://machinelearningmastery.com/recommender-systems-resources/>

## Split dataset (train/test)

from sklearn.model\_selection import train\_test\_split

# Load your dataset into a pandas dataframe

dataset = ...

# Split your dataset into training and testing sets

train\_set, test\_set = train\_test\_split(dataset, test\_size=0.2, random\_state=42)

# Print the size of your training and testing sets

print(f'Training set size: {len(train\_set)}')

print(f'Testing set size: {len(test\_set)}')

1. Train the model k-Nearest Neighbors (k-NN), Singular Value Decomposition (SVD) or Factorization Machines (FM).
2. Evaluate model performance using metrics such as root mean square error (RMSE) or top-K accuracy.

**Accuracy: This is the proportion of correctly predicted instances over the total number of instances in the test set. It is commonly used for classification problems when the classes are balanced.**

**Precision:** This is the proportion of correctly predicted positive instances over the total number of instances predicted as positive. It is used to measure the model's ability to correctly identify positive instances.

**Recall (Sensitivity):** This is the proportion of correctly predicted positive instances over the total number of positive instances in the test set. It is used to measure the model's ability to correctly detect positive instances.

**F1 Score:** This is the harmonic mean of precision and recall. It provides a balanced measure of both precision and recall, making it useful when the classes are imbalanced.

**Area Under the Curve (AUC):** This is the area under the Receiver Operating Characteristic (ROC) curve. It provides a measure of the model's ability to distinguish between positive and negative instances across different probability thresholds.

**Mean Squared Error (MSE):** This is the average squared difference between the predicted and actual values in a regression problem. It is commonly used to measure the quality of continuous predictions.

**Root Mean Squared Error (RMSE):** This is the square root of the average squared difference between the predicted and actual values in a regression problem. It is commonly used to measure the quality of continuous predictions, and is preferred over MSE when you want the error metric to be in the same units as the target variable.

There are many other evaluation metrics depending on the specific problem and algorithm you are using. It is important to choose an appropriate evaluation metric that aligns with the problem you are trying to solve and the specific requirements of your application.

## Apply the model with new users

## Continue and improve model

## Packaging python

To package a Python module or application, you can use tools such as setuptools or distutils. Here's a high-level overview of the steps involved:

1. **Create a project directory:** Create a new directory to hold your Python code, along with any necessary files such as READMEs, license files, or configuration files.
2. **Write your code:** Write your Python code and make sure it is organized in a modular and reusable way.
3. **Create a setup.py file:** Create a **setup.py** file in the root of your project directory. This file should define the metadata for your project (e.g. name, version, author, dependencies, etc.) and specify how to install your package.
4. **Create a package directory:** Create a directory to hold the Python modules that make up your package. This directory should have the same name as your package.
5. **Add an init.py file:** Inside your package directory, create an **\_\_init\_\_.py** file. This file tells Python that the directory is a package and can be imported.
6. **Organize your modules:** Put your Python modules inside your package directory. Make sure they are organized in a way that makes sense and is easy to use.
7. **Add package data (optional):** If your package includes data files (e.g. images, configuration files, etc.), you can include them in your package by adding them to the **MANIFEST.in** file.
8. **Build your package:** Use the **setup.py** file to build your package using the command **python setup.py sdist**.
9. **Upload to a package repository (optional):** You can upload your package to a package repository such as PyPI so that others can easily install and use your package.

Once you have created and packaged your Python module or application, you can distribute it to others who can then easily install and use it by running the appropriate installation command.

Top of Form

https://www.youtube.com/watch?v=GIF3LaRqgXoBottom of Form

## How to use a GoogleMaps API to have postcodes, addresses and coordinates in Berlin?

To use the Google Maps API to get addresses, postcodes, and coordinates in Berlin, you can follow these steps:

Get an API key: If you haven't already, you'll need to get an API key fro the Google Cloud Console. Follow the instructions in the Google Maps API documentation to create a new project, enable the Maps JavaScript API, and generate an API key.

Set up the API client: Install the googlemaps Python client library using pip: pip install googlemaps. Then, import the library and create a client object with your API key:

python

Copy code

import googlemaps

# Replace YOUR\_API\_KEY with your actual API key

gmaps = googlemaps.Client(key='YOUR\_API\_KEY')

Geocode addresses: Use the geocode function to convert an address or place name into a latitude and longitude coordinate pair:

python

Copy code

# Replace ADDRESS\_OR\_PLACE with the address or place you want to geocode

geocode\_result = gmaps.geocode('ADDRESS\_OR\_PLACE, Berlin, Germany')

# Extract the latitude and longitude from the geocode result

lat = geocode\_result[0]['geometry']['location']['lat']

lng = geocode\_result[0]['geometry']['location']['lng']

Reverse geocode coordinates: Use the reverse\_geocode function to convert a latitude and longitude coordinate pair into an address:

python

Copy code

# Replace LATITUDE and LONGITUDE with the actual coordinates you want to reverse geocode

reverse\_geocode\_result = gmaps.reverse\_geocode((LATITUDE, LONGITUDE))

# Extract the address components from the reverse geocode result

address\_components = reverse\_geocode\_result[0]['address\_components']

# Extract the postcode from the address components

for component in address\_components:

if 'postal\_code' in component['types']:

postcode = component['short\_name']

break

# Extract the formatted address from the reverse geocode result

formatted\_address = reverse\_geocode\_result[0]['formatted\_address']

Note that the Google Maps API has usage limits and may require payment for high volume usage. Be sure to check the documentation and pricing information before using the API in a production environment.