Introduction

The program is supposed to resolve the task as follows. There is a waste/recycled commodity marketplace with product offers and registered customers. The customers, besides posting offers, view products offered, view contacts of the seller, and message sellers. The activity of customers, that is *viewing page*, *viewing contacts* and *sending messages* is logged. Based on the record of customer behaviour and the product newly offered over a specific period of time, suggest to customers those of the new products, that match best their history of interest.

Method

We will shortly introduce the available data the method is based on, and then the method itself.

Data

The key data used for **customers** is *customer id* (int), *country*, preferred *amount* of waste, whether their accounts have been *deleted* or *blocked*, and a triplet of basic interests of whether the customer wants to *(buy, sell, other)*. The customer activity is tracked in terms of *product id* they interacted with, and whether the interaction was *page view*, *contact view* or *message* to the seller. The historical activity data and interactions with a particular type of product were used to establish customer preferences.

The key data used for **products**, particularly to evaluate product viability for the customer, were *unit price*, *location*, and *availability of shipping.* To evaluate customer interests based on interactions with a particular product id, the important fields were *category* (e.g. plastic, metal), *subcategory* (e.g. pp, aluminium), and *product type*, which is one or more of *waste*, *recycled*, *byproduct* for each item.

Note that the prices listed in the product data seem highly incorrect for many products, which is probably due to price input for an incorrect unit. Besides, a large proportion of products does not have price attached.

Processing

For further use, we converted all prices to euro, and all weight-based units to metric tons. We converted string based data (amount range, interest, product type) into tuple-like structures. We computed mean price for each unique *category-sub category-product type* combination in the data, but the averages are obviously incorrect (as stated above) and we didn't use the price-based logic.

For each customer, we collected three lists of *product id*s, a list of ids for which they *viewed page*, *viewed contacts*, and *sent message.* We (optionally) filtered out customers, whose accounts are blocked or deleted, and who set their preference to *sell* only.

Geolocation To be able to estimate the delivery distance, we used OSM api to compute representative distance (in thousands of kilometres) between each pair of countries appearing in the data. The matrix is cached, as its calculation takes some time because of the api calls.

Preference Scoring

To select the best recommendations for each customer, we computed their scored preferences based on the previous activity.

1. We assigned weight to each activity (configurable, see implementation) based on its type.
2. For each customer, we combined the activities relating to each particular *product id*, summing the weights of each aggregated activity type (e.g. one page view for with weight 1 and a message sent with weight 5 for a single *product id* gives weight 6 for the product).
3. For each customer, we sub-selected the *product ids* they interacted with, including the information about *category*, *subcategory*, *product type* of that product, including the aggregate *weight* of the interaction they had with the particular *product id.*
4. For each customer, we grouped this table of *product ids* and related information over the unique triplets of *category-subcategory-product type*, and for each such product categorization (triplet categorization), we summed the weights.

This method allows us to get scored (weighted) preference of each customer for category of products defined by such triplets (that is *category-subcategory-product type*). This scored preference is based on all unique triplets they interacted with by *viewing page* or *viewing contact* or *messaging.* Note that such preferences are defined for each user, but are aggregated over their *product id* interactions.

Affordability Scoring

Having computed weighted user preferences, we matched each users' (triple categorized) preferences to the list of products added within a time window of interest. For each matching triplet, we obtained personalized preference score of the given customer for the given product.

We filtered out such preference matched *product ids*, which the customer has already messaged, or which belong to them.

To further evaluate a particular new product choice, we assigned a series of real-world penalties, depending on the absence of shipping offer, distance to the product, offer being lower than customer's preferred range, and the offer being too expensive as compared to the average unit price.

Of such combined preference and affordability score, we select a given number (see implementation) of best scored options.

In case the number of found matches does not yield the required number of recommendations, we repeat the matching phase, but matching only on *category-product type* couplet. In case there are fewer matches after such relaxation, lower count is accepted. If nothing is found, the absence of match is reported for the given time window.

The final results are stored in form of an Excel file, with the *customer id*, a list of their 3 (optionally) top triplet preferences, and a list of top suggested products. Note that if a given recommended *product id* refers to more than one *product type*, all available types are listed on separate lines.

Implementation

File Structure

The program is implemented in Python, relying on Pandas library. The structure is as follows

```
productmatching
├── config
│   └── config.yaml
├── data
│   ├── distance.json
│   └── source.xlsx
├── doc
│   ├── documentation.tex
│   ├── documentation.pdf
│   └── requirements.txt
├── log
│   └── matchmaker.log
├── out
│   └── results.xlsx
└── src
    ├── definitions.py
    ├── data_reader.py
    └── matchmaker.py
```

The file *config.yaml* contains the configuration for the program run. Particularly the data source, the number of expected suggestions to show to a customer, the weights for customer activities (e.g. *messages*), the penalties for product affordability (e.g. *distance*), filter rules for customers (e.g. *blocked*), and the required time window definition. All options are commented.

distance.json contains distance matrix between relevant countries, *source.xlsx* is the file with the source data (**not gitted**).

The *documentation.pdf* (and its tex source) is this file, while *requirements.txt* contain the list of python libraries. The *matchmaker.log* contains reports of basic steps of the program run.

results.xlsx is an Excel file with the final suggestions.

definitions.py defines program paths, *data_reader.py* deals with basic reading and data processing, *matchmaker.py* implements the scoring logic.

Main Classes

The processing steps are mainly implemented in the *date_reader.py* in the class *RawData*. It adapts the input data into required form, changes the strings into tuple-like structures. The *Distance* class maintains the distance matrix (the one cached in *distance.json*).

The scoring logic is implemented in *matchmaker.py*. Upon object creation, the initialization method first reads the configuration from the *config.yaml* file through an object of *Config* class. Then, it invokes *date_reader.py*'s *RawData* class to read and parse the data. Then, it computes the customer triplet preferences as described in the section Preference Scoring. Then it filters the new products added within the required window, and matches those products against individual users preferences and affordability criteria.

Running the Program

In the default settings, one can simply run the program by running *matchmaker.py* as a script. As default, it reads configuration from *config.yaml* and writes results to *results.xlsx*. The run takes a couple seconds. This supposes one has the necessary packages installed within the used Python environment.

The analysing method, a *MatchMaker* class constructor, has two optional parameters, *MatchMaker(config_path_, write)*. The first argument is path to configuration file, the second argument is a switch to write results or not.

To change configuration, either edit *config.yaml* directly, or create a new yaml file, and pass the path as the first argument to *MatchMaker*. The computation is performed when object *MatchMaker* is created.

To run in a Python console, add the source path to the environment, import the class and create the object.

```
sys.path.extend(['.../productmatching/src'])
import␣matchmaker
matchmaker.MatchMaker()
```

Note To run the program, you need the source data under *data/source.xlsx*. This data is not gitted, so to test, you need to add such file manually.

Validation

The following methods could be used to validate the approach:

Comparison to random selection One would simply generate the suggestions by randomly choosing from the new products, and then compare, whether this choice corresponds to customer preferences (as outlined above), and how it differs from the current implementation.

Sampling and noise One should be able to sample only a part of the history data, and still obtain similar recommendation, using the implemented method. Equally, one could generate random extra noise with statistical properties (in terms of product category and type) of the historical data, and assign such noise activity randomly to the customers, and still obtain similar recommendations.

Splitting One could split the historical data into two random samplings, and performing the analysis of each, should yield similar results between one and the other, and the current implementation.

Historical comparison One could perform the same analysis for a time window in the past, and then verify, whether the customer activity corresponded with the method prediction.