Recommender System: Suggesting the Items Most Relevant to a Current Query

Alvaro Morales
School of Business and Economics
HEC Lausanne
Lausanne 1007, Switzerland
alvaro.morales@unil.ch

Pascal Schenk
School of Business and Economics
HEC Lausanne
Lausanne 1007, Switzerland
pascal.schenk@unil.ch

Andre Zommerfelds
School of Business and Economics
HEC Lausanne
Lausanne 1007, Switzerland
andre.zommerfelds@unil.ch

***Abstract*—This document deals with the question of how one can implement a simple recommender system for an e-commerce platform. We investigate this question by building a two-sided interface: from one side, the business owner can adjust the model to fit the specific business situation and from the other side, the customer receives product recommendations for a chosen product. To show the practicality of this interface we follow in the footsteps of a manager who wants to make a simple recommender system for the the sales of Amazon products. Thus, we use the official Amazon product data set.**

***Index Terms*—k-nearest neighbors, recommender system, web interface**

I. Introduction

Sophisticated product recommendations have become increasingly popular and have found application in a variety of areas like e-commerce, movies, music or news (Schafer et al., 2001 [24]). They enhance the user experience by making predictions of what the customer really wants. It can have a significant positive impact on revenues by effectively positioning a product line towards the target customer's needs. Amazon too uses a recommender system on their website, which automatically suggests similar items for a given query. Such a system tries to make correct predictions of what products the user would like to see next.

The aim of this document is to show how one can implement a simple recommender system for a hypothetical e-commerce business. We seek the answer to the question of how one can implement a simple recommender system for an e-commerce online shop. We investigate this question by building a two-sided interface as follows. From one side, the business owner can adjust the model specifications for the recommender such that it fits the specific business situation. From the other side, the interface takes the perspective of the customer, where one chooses a specific item and receives the respective product recommendations. To show the practicality of this interface we follow in the footsteps of a manager who wants to make a simple recommender system for the the sales of Amazon products. Therefore, we use the Amazon product data base from their official Simple Storage Service (Amazon S3) [2].

II. Research question and relevant literature

The focus of this document is to investigate the question of how one could implement a simple recommender system for a hypothetical web-shop. The implementation of such a system has to be broken down into smaller pieces because it touches subjects from different fields. We recognize three different parts, each with its unique questions. First, there is the development of the recommender algorithm itself. Then, after having built the core of the system, one can tweak the model such that it gives better results. This is a more theoretical question which we will discuss very briefly. Finally, such a system is only useful if it is interactive, that is, it changes the recommendations based on a given query of the customer. Thus, we have to implement a code which takes action whenever we call for it. In that matter we develop a simple web-interface.

A lot of research has been done on recommender systems and there are different algorithms such as k-nearest neighbors, collaborative filtering, clustering or any combination of these methods. We decided on using the method of k-nearest neighbors because it is known to be simple and efficient for not too large data sets (Sarwar et al., 2002 [23]). Large businesses have adapted and extended their recommender systems making them more sophisticated for their specific needs. Amazon for example is using a user-based collaborative filtering method which essentially also relies on a nearest neighbors method (Zhang and Pu, 2007 [33]). To conclude, we start answering the research question by building a simple recommender system based on k-nearest neighbor, which is in a sense the core of the algorithm used by large businesses. Having the perspective of an entrepreneur trying to build an e-commerce business from zero, we seek to find the minimum viable option so that we can quickly adapt to the customers' needs.

III. Methodology

As stated before we look at three problems when implementing the recommender: coding the algorithm, choosing the best model parameters and making the system interactive. While we discuss the development of the algorithm and the interface more thoroughly it is still important to understand the basic principles of the k-nearest neighbors algorithm behind the recommender system. The following paragraph is a brief introduction to the algorithm.

A. General approach

Imagine a data set of an e-commerce business consisting of three dimensions: customers, products and ratings for each product bought by the customer. When the user starts looking at a product, the recommender system takes this query and plugs this unique product identification to the k-nearest neighbors algorithm. This means that it will take all vectors of customers who bought this product, which includes all other products that these customers have bought and the respective ratings. Finally, the k-nearest neighbors algorithm looks at the distance of these vectors (each consisting of a customer, a product and a rating) and takes the k-nearest vectors using a specified distance metric (e.g. euclidean or cosine). As such, we extract the "nearest" products given by these vectors. For a more detailed explanation of k-nearest neighbors the reader can consult for example chapter 2 in Pattern Recognition and Machine Learning by Bishop (2006) [4].

B. Data

To replicate the situation of a manager building or optimizing a recommender system for his customers, we take the Amazon product data set from their official Simple Storage Service (Amazon S3 - data set accessible here, [2]. This data set consists of almost 50 item categories, some examples being music, gardening or books - the latter being Amazon's most iconic product category. For each category you can find the different information where the only relevant ones for the purpose of this document are the following:

- customer identification
- product identification
- star rating

These data sets can be very large with more than 2 GB of *compressed* data. In some cases this makes it extremely computationally intensive to use k-nearest neighbors, indicating one of its drawbacks when using large data. For example the Video DVD data base table 1.4 GB large of data consists of more than 2 million unique customers and almost 300.000 unique products. The books categories are split up in three database tables, each of approximately 2.5 GB of data and 6 million unique customers who made a review to 1 million unique products. We discuss this large data problem in Section IV.

C. The source code

When it comes to building the recommender system, we can split up the program into three different parts. In the core of the back end lies the recommender algorithm, i.e. the k-nearest neighbors model. When it comes to the front end, we see an interface where from one side the manager can tweak the parameters and check any statistic as needed (e.g. number of unique customers, products, any errors or runtime problems, etc.) and from the other side, the customer searches for a product and receives suggestions of similar items. This situation clearly requires a solution which is accessible whenever the customer wants. Thus, we have to establish a web server which is reachable 24 hours a day. In other words, we have to return to the back end and build a server application which keeps the recommender system running. Summarized, we see three core parts in our program:

1) K-nearest neighbors algorithm
2) Web server
3) Web interface

D. Implementation of the program

The next paragraphs introduce the reader to the core parts of the program, which are explained in more detail in section IV.

1) The recommender algorithm: The program is coded in Python [18]. We use different packages for different purposes. First, for the k-nearest algorithm we use scikit-learn (see Jones et al., 2014 [11] and Pedregosa et al., 2011 [19]. We store and manipulate data using Numpy (see Oliphant, 2006 [17] and Van Der Walt et al., 2011 [27]) and Pandas (McKinney, 2010 [16]).

2) Web server and interface: For both the web server and the web interface we use the Flask toolbox (Lord et al., 2019 [14] and Grinberg, 2018 [8]). Flask is a web server gateway interface (WSGI) for Python and is one of the most popular web application frameworks with a big community and many extensions provided by their users. Since we are dealing with a web interface, we make use of the HTML to display the recommender system on the web (see W3C, 2019a [29] and WHATWG, 2019 [28]) and also very little of CSS (see W3C, 2019b [30]).

IV. Implementation of the recommender system algorithm

The goal of the program is to get a sense of how a basic recommender system can be implemented in a hypothetical e-commerce venture. We want an application where from one side the manager can check how the algorithm is working and more importantly, how tweaking different parameters can change the output of the suggestions. From the other side, we want the program to be able to accept a unique product

identification and return similar items. Figure 1 gives an overview of the workflow of the program.

As the reader can see, the application is an interaction between input from both the manager and customers, the data set and the recommender system embedded in a web server. To understand the program, we explain by following the work flow. Thus, we start from the beginning of the application, i.e. starting up the server, till the end, i.e. where we get the recommendations. We could also explain the application by distinguishing between back and front end, i.e. the recommender algorithm and then the interface. But since the processes in this application are so interlinked, we believe that going through each step of the work flow makes the understanding of the application much more straightforward to the reader.

A. Program structure

Figure 2 gives an overview of the program structure. We want to construct a program that can be run as a module in python. From the "Recommender System" directory we can open our application with the command "python -m program" which looks for a __main__.py file in "program" directory. The core of the program lies inside this folder. The mentioned "main" file triggers the application by starting the server (app_server.py). The server is the code that runs a Flask web server which takes some defined functions from the app_modules.py file used for various computations. Within these functions there is most importantly the code for the recommendendation using k-nearest neighbors. All the other elements in the app_modules.py file are data manipulators, i.e. downloading, saving, cleaning, etc. The data base is stored a separate "data" folder. From the application you can download a selected category, for example "Musical Instruments", which is then stored as a compressed tsv.gz file in this data directory. To keep the programm well documented, we have a documentation folder which includes the program structure, the work flow and all references that were used for building the program. Also, there is a README.md file on top of the directory hierarchy, which includes instructions and other information related to the program. Finally, there is a "static" and a "templates" folder which both make part of the Flask server structure. As the name of the folder suggests, we save any static file, e.g. images, used in the web interface inside the static folder. Since we are dealing with a web interface, everything will be displayed using HTML and very little CSS and we store the respective codes in the templates directory - which are basically the web pages the user accesses in the application.

B. Flask - starting the web-application

As the creators of the the toolbox explain, Flask is a lightweight microframework for web development [14]. The "micro" means that you can build the core of a web server and an interface using the Flask structure use other applications to manage everything else, e.g. the database structure using SQL and the ins and outs of the web interface using HTML,

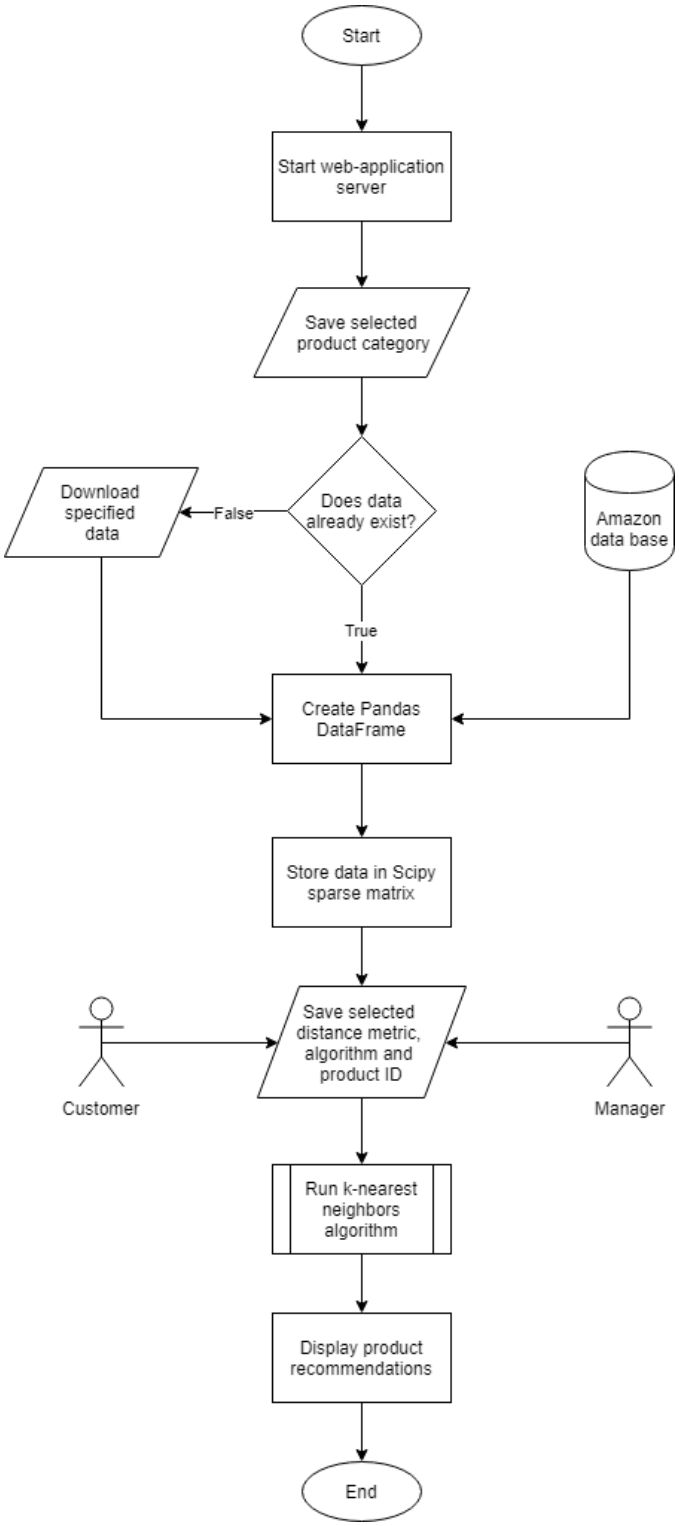

Fig. 1. Flowchart of code (own illustration)

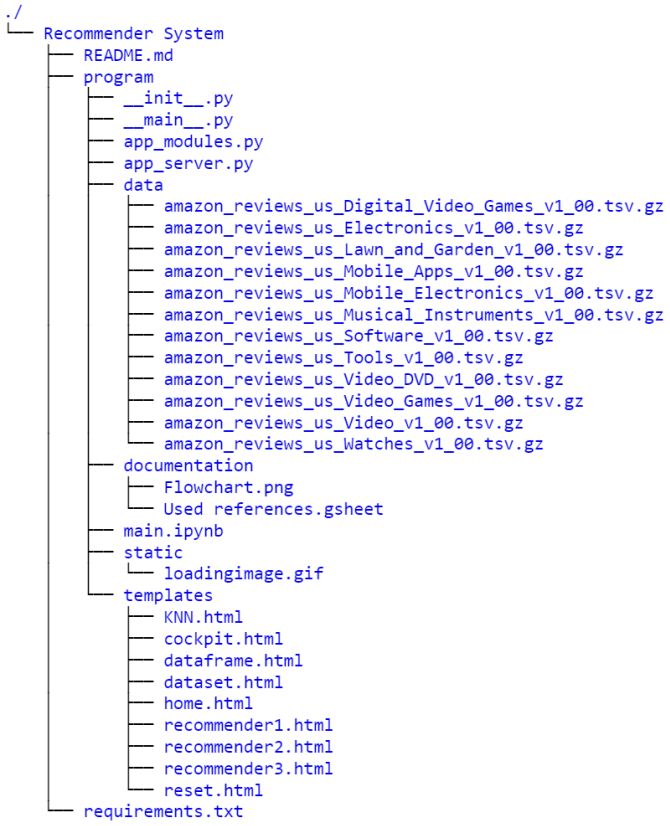

Fig. 2. Program structure (own illustration)

CSS and JavaScript. Thus, it is highly adaptable to the needs of the software developer. The goal of this paper is not to explain Flask and neither HTML in detail. Still we introduce the reader to the very basics to make the application more understandable. For further readings we recommend the online documentations (for Flask, see (Lord et al., 2019 [14] and Grinberg, 2018 [8]) and for HTML, see W3C, 2019a [29] and WHATWG, 2019 [28]). In our program we start the Flask server as can be seen in Figure 3. Basically, we are starting a web server with a random port which automatically opens up the home page on your standard web browser. This start page is coded in the "home.html" file saved in the templates directory explained before. Generally, in Flask you define "routes" to trigger a defined function in the code. The last two lines in the same Figure 3 are an example of this basic structure of a Flask application. There we define a function called "home( )" which simply opens up the mentioned "home.html" page. As the reader can see, when we start the route with the line *@app.route("/")*, the Flask app assigns the function "home( )" to the URL followed with a slash "/". Summarized, you have the following work flow:

1) by accessing the URL "http://127.0.0.1:PORT/" (notice the slash in the end of the URL) ...
2) you trigger the function "home( )" ...
3) which displays the "home.html" on your browser.

```
app = Flask(__name__)

# Automatically open in web browser
port = 5000 + random.randint(0, 999)
home_url = "http://127.0.0.1:{0}".format(port)
threading.Timer(1.25, lambda: webbrowser.open(home_url) ).start()

@app.route("/")
def home():
    return render_template('home.html')


app.run(port=port, debug=False)
return app
```

Fig. 3. Start the Flask application (own illustration)

One last note to the Flask toolbox is that we often create routes with an HTTP method called "POST", e.g. @app.route("/recommender/1", methods=['POST']). HTTP methods simply define how to communicate between a server and a client. In this regard, the "POST" method is used to send any data from the client to the server. This data can come from different sources, for example triggering a button which sends a message to the server, or submitting a string value to the server (for more on HTTP methods, see W3C, 2019c [31]). In the end, we want the server to do computations with this kind of input. An example using our program is the following. The customer searches for a product, this sends the item identification to the server, which is then used as an input to the recommender algorithm. After computations, the function same function triggers an HTML page where the item suggestions are displayed.

C. Dealing with data

Continuing the work flow, we now have to deal with the data. We want a program which downloads the data from the Amazon Simple Storage Service (Amazon S3) as needed. Therefore we create a function inside the app_modules.py file called data_download( ), which has the following properties. As soon as the necessary database is selected and submitted, the program checks if the data is already downloaded and if not, it requests the file online and saves it into the data directory. The process continues with the creation of a Pandas DataFrame with the data_frame( ) function. This allows us to create a usable table which can be manipulated later on. Included in this step is also the important process of automatically cleaning the downloaded database by removing unnecessary rows or columns and converting. Going further down the flowchart, we now have to change the way to handle the data so that it becomes more manageable for the recommender algorithm. As already mentioned, the databases can include more than a million unique customers and often more than 200 thousand unique products which makes it computationally intensive to work with. The way the recommender works with the data is that it creates a pivot table with unique customers in the rows, unique products in the

columns and the star rating as a value to each customer-product pair. As the reader can imagine, this pivot table would have an extraordinarily amount of zeros because we are dealing with dimensions of 2 million customers times 200 thousand products and obviously for each customer there are only a few products that have been bought and rated. A way to deal with such sparse matrices is to create a compressed sparse row (CSR) matrix with the purpose of getting rid of all non-zero values for much more efficient matrix operations. Scipy offers a function called scipy.sparse.csr_matrix( ) which is used in our recommender algorithm (see Scipy, 2019 [26]). In essence, this takes the mentioned pivot table and creates a matrix with three row-vectors: one containing all non-zero values, another with column indices for each of these values and a third with row-pointers (column indices) for the first non-zero value in each row (for a more complete understanding we refer to the mentioned Scipy documentation). This CSR matrix is created within the function data_KNN( ) defined in the app_modules.py file. Note that this step necessarily comes with the process of assigning a numeric index for each unique customer and product and creating the mentioned pivot table by removing duplicates, i.e. only taking unique customers in the rows and unique products in the columns. This indexing process is needed for the computations of the k-nearest neighbors algorithm.

D. The k-nearest neighbors algorithm

For our recommender system we take Scipy's nearest neighbors algorithm and integrate it in our data_recommender( ) function. Scipy's nearest neighbors algorithm can be adjusted by choosing different algorithms for the regression and different distance metrics as mentioned in Section III. For more information on Scipy's nearest neighbor function and the possible algorithms and distances, please consult their documentation [25]. Since we want to build an interface which allows the manager to tweak and optimize the model, we integrate a selection of algorithms and distance metrics to be explicitly chosen by the user when steering the recommender application. To conclude, the input needed for making the suggestions is the prepared CSR matrix of customer-product pairs, the chosen algorithm, the distance metric and last but not least, the product chosen by the customer. The reader can jump to Section VI to see how these steps are actually implemented in the web interface.

E. Displaying the product recommendations

After having prepared the data and run the recommender algorithm, the program now displays the item suggestions. At the time of writing, our early-stage program requires the user to actually go to the Amazon website and search for an item within the chosen category. To make this process simpler, the interface can redirect the user with a selection of links to the selected product category. There, one still has to extract the product identification from the URL as shown with the ID B003VWJ2K8 in Figure 4. Unfortunately, the data set offered by the Amazon Simple Storage System is not completely synchronized with their current offered products. This means that often when the user submits a product product ID into the recommender, the program returns an error message that it did not find the chosen item in the database. This problematic is left for future experimentation with the recommender system.

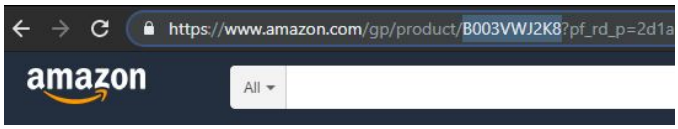

Fig. 4. Amazon product identification

F. Additional features

For each step we implement important features which make the interface more user-friendly. The application is often returning messages (including information on possible errors) to better communicate with the users and inform on the status of a given query. An example is a download bar on the terminal for the back end user which is very useful when handling databases with more than 1 GB in size. Furthermore, the interface includes explanations for each step, there are buttons which steer the user to the right direction and whenever needed, one can switch back and forth between the steps. To summarize, the user should be aware that the program is constantly giving information on its status, both on the interface (front end) as well as on the terminal (back end processes).

V. Program Maintenance

We recognize different focus points for maintaining our program:

- fixing errors
- extending the program with more features
- optimizing the performance

Our approach to this problem is first and foremost to keep the program well documented, such that whenever one comes back to the source code, the intentions will be understandable. That is why we include many comments in the code. Furthermore, we write files which help the user understand the program, such as the requirements.txt, the README.md and all files included in the documentation folder (including this report). More importantly we want a solution where the program is stored publicly, such that other people can share and contribute with new ideas and suggestions. That is why we create a GitHub repository called git-RecommenderSystem (see GitHub, 2019 [6] and git-RecommenderSystem, 2019 [7], directly accessible here). This way we can not only keep track of all changes committed to the program but also share new ideas which can then be merged together.

VI. Results

Having explained the core of the recommender system we now turn to the presentation of the main results. Again, we take the approach of working through the flowchart in Figure 1 for better comprehension.

A. Flask - starting the web-application

When the application starts, the standard web browser opens a new tab with the home page of the recommender system shown in Figure 5. This is the basic HTML layout chosen for the program. The user continues by assessing the cockpit of the recommender system. Remember that we are mainly taking the position of a software developer or a manger who is in charge of the model and wants to see how it is implemented for his e-commerce venture.

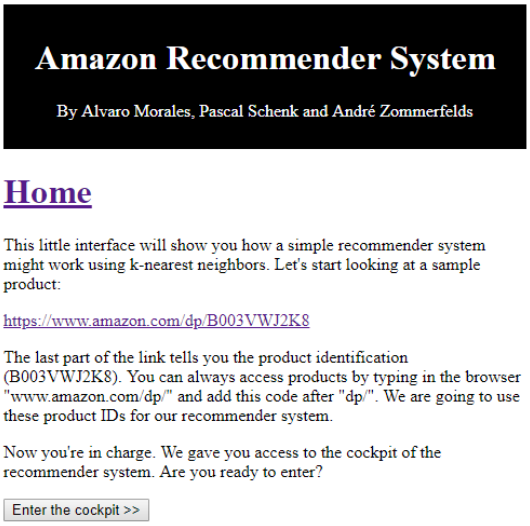

Fig. 5. Home page of the recommender system

B. Dealing with data

Figure 6 shows the cockpit of the application. Here the user has to go through a 3-step process: first download the selected database, then create a Pandas DataFrame to manipulate the data and finally convert the data into a CSR matrix as discussed in Section IV-C. We build the program this way so that the user gets an intuition for what is needed for the recommender system. We even argue that it is necessary to actually keep track of what database you have to store in your server and how some data manipulations can be very computationally intensive.

By individually going through each step, the user actually experiences how long it can take to actually prepare the data. In order to give the user more information on each step, the program actively communicates its status. Figure 7 and 8 show this is implemted in the front as well as in the back end of the application. Along with its status, the program gives some additional information on the current data instance being prepared. For example it prints out the number of unique

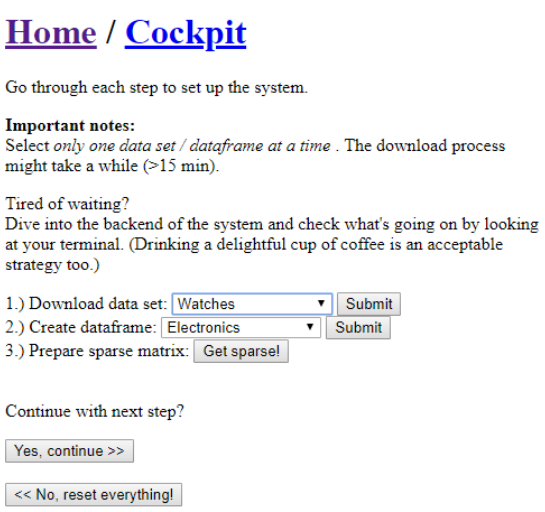

Fig. 6. The cockpit

customers and products for the chosen database as can be seen in Figure 9.

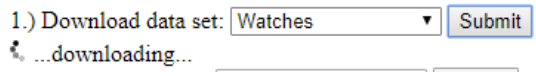

Fig. 7. Downloading the database

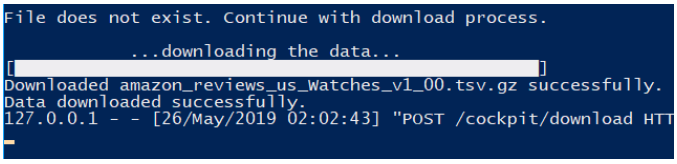

Fig. 8. Back end download status

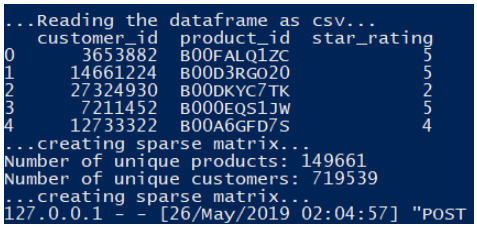

Fig. 9. Additional back end information

C. The k-nearest neighbors algorithm

Now, the user gets to choose a product from the chosen category. To make the process of finding an item easier we provide a drop down list with the categories as shown in Figure 10. The user can then extract the product identification and plugging it in the recommender system. Having found a product and copied the product identification, the user can

paste this and submit it along with the chosen distance metric and the algorithm for the k-nearest neighbor function. In the current program version at the moment of writing, there is no clear distinction between what we call the customer interface and the administrator interface. As illustrated in Figure 11, the item query and the choice of the algorithm along with the distance metric happen simultaneously. Obviously, this program can and should be extended by separating the interface in two: an interface for the administrator and one for the customer (the latter being hopefully much more well-designed) - we leave this for a future undertaking.

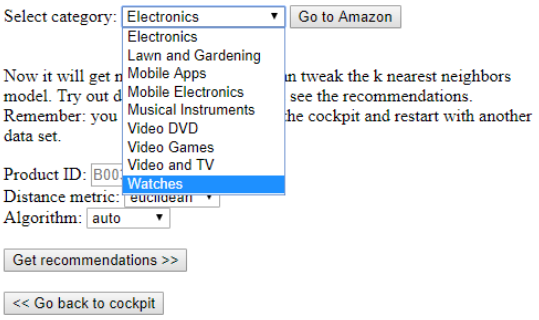

Fig. 10. Link to Amazon

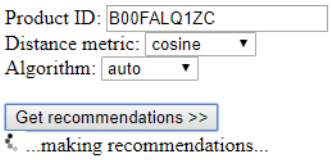

Fig. 11. Making recommendations

D. Displaying the product recommendations

Finally, we arrive at the main purpose of this application: showing the recommendations for the given item query. The program is built in such a way that it takes you to a next page as shown in Figure 12 with 15 item suggestions for the given product. In this example we have chosen a watch and we can now look at the recommended products by clicking on their link. This leads us to the official Amazon store as is visible in Figure 13.

Fig. 12. Item suggestions

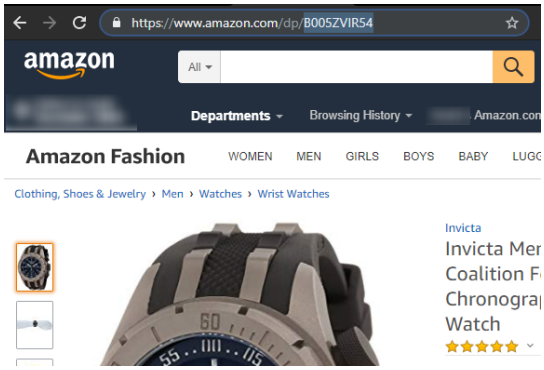

Fig. 13. Suggested Amazon product

VII. Conclusion

The purpose of this document is to show how an entrepreneur willing to open his own e-commerce project can build a web interface with a built-in recommender system from scratch. We code the application with Python along with some toolboxes and us the product review data set from Amazon's Simple Storage Service (S3). The result is an interface where the administrator can prepare the necessary data and select different parameters for the k-nearest neighbors model (the algorithm and the distance metric). It takes the user to an easy to understand step-by-step process, so that the concept of the recommender gets very clear to the user. This finally leads to the page where, for given parameters in the algorithm, one can submit a product ID and get recommendations. The program opens up a new page with 15 suggested items, each with a link to the official Amazon store.

This application should be seen as a minimum viable product (MVP), which essentially is the product with the minimum amount of features, but with enough benefits to satisfy early-stage customers (see Robinson, 2001 [21] and Ries, 2011 [20]). As such, we show how possible it is for an entrepreneur with some knowledge in coding to actually implement a simple interface with a recommender system within a few days.

Of course, the program comes with many drawbacks. This starts by the process of loading the data every time we start the server. Imagining that the serve can be kept running permanently, this is not really a big problem because the application is built in such a way that after it has loaded and prepared the data, the user can get as many recommendations as needed. Thus, the two processes are decoupled from each other. Still, the user has to keep in mind that for every change in the application, one has to restart the server. The recommender algorithm is as simple as it can be. By looking at different products we find that it gives decent results using cosine as a distance metric and setting the algorithm to automatic. Unfortunately, we still find odd results from time to time - a good example being the suggestion shown in Figure 13. The reader might notice that the original item in the query is categorized as a watch for women, while the first suggested product is in Amazon's category for men [1]. These situations reminds the user of how a recommender system based solely on k-nearest neighbors can be be very easy to implement, but at the same time, it lacks a lot of necessary complexity to understand the customers' needs and behaviours. Luckily, there is extended research made in this area and it is just a matter of extending the existing algorithm with more fancier models such as extended user-based collaborative filtering methods (which is actually used by Amazon and other famous websites like Youtube or Netflix - see for example Linden et al., 2003 [13]). We also find that oftentimes items are not found in our data set and we conclude that this is a simple matter of having the database actually synchronized with the offered products - which can be attributed to the problem of taking the current official product offerings from Amazon and working with a separate historic data set of reviews. Last but not least, the performance can be massively improved from one side, by handling with the data and from the other, by making the suggestions. At the moment of writing, the program cannot handle bigger data sets like Amazon's books category. In this situation it simply returns a memory error even before creating the CSR matrix. Also, the server cannot handle more than one database at a time but the application allows the user to open parallel servers for each database and generate independent item suggestions for each category. This solution is far from optimal, since it requires an URL and maintenance for each server. The product recommendation takes around 3 to 10 seconds on average, which is very slow compared to an actual item query on Amazon. We believe this can be heavily optimized not only by using better hardware but arguably even more importantly, optimizing the code. We strongly believe that using parallel programming along with matrix factorization techniques could make our program significantly faster (see Koren et al., 2009 [12] and Yu et al., 2014 [32]). In particular, this approach might be achieved either by extending the program with Cython, by programming the back end in C++ and the interface in Python or yet another option is to implement the program in a cluster environment for example with Apache Spark (see Cython, 2019 [5], Apache Spark, 2019 [3] and A Medium Corporation, 2019 [1] for an exemplary project that implements a recommender sytem with collaborative filtering using Python, Cython and Apache Spark).

[1]We do not support the belief that a watch is to be sold *per se* to one gender only - here we simply refer to how Amazon categorizes it products and how this might affect the customers' demand and behaviour.

We conclude that it is possible and straightforward to implement a simple recommender system for an e-commerce platform with basic programming skills, although there are a few important problems that should be taken care of. We recognize three problematic areas in particular when undertaking such a project. First, one should make use of fancier recommender algorithms to better meet the customers' needs and behaviour. Second, the program has to be coded in an efficient way, such that it can handle large data manipulations (even if the data is larger than a few GB) and make computations quicker to make the user experience smoother. Third, the customer interface has to be elaborated on by making it well-designed and very interactive since it can massively increase the user experience. In the end, the purpose of a good recommender system is to make the customer happy with our products and services, which should be the first priority of such a venture. In that sense, we believe that our application combined with a build-feedback-learn loop, is a valuable example and a strong case for a minimum viable product which most importantly sets the focus on the customers.

References

[1] A Medium Corporation (2019). How We Implemented a Fully Serverless Recommender System Using GCP, https://medium.com/google-cloud/how-we-implemented-a-fully-serverless-recommender-system-using-gcp-9c9fbbdc46cc [Online; accessed 2019-05-25].

[2] Amazon data set. Official data set consisting of product reviews, ratings and more from their offered products, downloaded from https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt, May 12, 2019

[3] Apache Spark (2019). Apache Spark: a unified analytics engine for large-scale data processing, https://spark.apache.org/ [Online; accessed 2019-05-25].

[4] Bishop, C. M. (2006). Pattern recognition and machine learning. springer.

[5] Cython (2019). Cython: C-Extensions for Python, https://cython.org/ [Online; accessed 2019-05-25].

[6] GitHub, (2019). https://github.com/ [Online; accessed 2019-05-25].

[7] Git-RecommenderSystem Github Repository, (2019). https://github.com/azommerf/git-RecommenderSystem [Online; accessed 2019-05-25].

[8] Grinberg, M. (2018). Flask web development: developing web applications with python. " O'Reilly Media, Inc.".

[9] He, R., McAuley, J. (2016, April). Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In proceedings of the 25th international conference on world wide

web (pp. 507-517). International World Wide Web Conferences Steering Committee.
[10] Herlocker, J. L., Konstan, J. A., Terveen, L. G., Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. ACM Transactions on Information Systems (TOIS), 22(1), 5-53.
[11] Jones, E., Oliphant, T., Peterson, P. (2014). SciPy: Open source scientific tools for Python, http://www.scipy.org/ [Online; accessed 2019-05-25].
[12] Koren, Y., Bell, R., Volinsky, C. (2009). Matrix factorization techniques for recommender systems. Computer, (8), 30-37.
[13] Linden, G., Smith, B., York, J. (2003). Amazon. com recommendations: Item-to-item collaborative filtering. IEEE Internet computing, (1), 76-80.
[14] Lord, D., Mnnich, A., Ronacher, A., Unterwaditzer, M. (2019). Flask: web development toolbox for Python, http://flask.pocoo.org/ [Online; accessed 2019-05-25].
[15] McAuley, J., Targett, C., Shi, Q., Van Den Hengel, A. (2015, August). Image-based recommendations on styles and substitutes. In Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (pp. 43-52). ACM.
[16] McKinney, W. (2010, June). Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference (Vol. 445, pp. 51-56).
[17] Oliphant, T. E. (2006). A guide to NumPy (Vol. 1, p. 85). USA: Trelgol Publishing.
[18] Python Software Foundation. Python Language Reference, version 3.7.3. Available at http://www.python.org
[19] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), 2825-2830.
[20] Ries, E. (2011). The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses. Crown Books.
[21] Robinson, F. (2001). Minimum Viable Product, http://www.syncdev.com/minimum-viable-product/ [Online; accessed 2019-05-25].
[22] Sarwar, B., Karypis, G., Konstan, J., Riedl, J. (2000). Application of dimensionality reduction in recommender system-a case study (No. TR-00-043). Minnesota Univ Minneapolis Dept of Computer Science.
[23] Sarwar, B. M., Karypis, G., Konstan, J., Riedl, J. (2002, December). Recommender systems for large-scale e-commerce: Scalable neighborhood formation using clustering. In Proceedings of the fifth international conference on computer and information technology (Vol. 1, pp. 291-324).
[24] Schafer, J. B., Konstan, J. A., Riedl, J. (2001). E-commerce recommendation applications. Data mining and knowledge discovery, 5(1-2), 115-153.
[25] Scikit-learn, 2019. Nearest Neighbors, https://scikit-learn.org/stable/modules/neighbors.html [Online; accessed 2019-05-25].
[26] SciPy, 2019. The scipy.sparse.csr_matrix( ) function, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html [Online; accessed 2019-05-25].
[27] Van Der Walt, S., Colbert, S. C., Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. Computing in Science Engineering, 13(2), 22.
[28] Web Hypertext Application Technology Working Group (WHATWG) (2019). HTML: Hypertext Markup Language, https://whatwg.org/ [Online; accessed 2019-05-25].
[29] World Wide Web Consortium (W3C) (2019a). HTML: Hypertext Markup Language, https://www.w3.org/html/ [Online; accessed 2019-05-25].
[30] World Wide Web Consortium (W3C) (2019b). CSS: Cascading Style Sheets, https://www.w3.org/Style/CSS/ [Online; accessed 2019-05-25].
[31] World Wide Web Consortium (W3C) (2019c). HTTP Method Definition, https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html [Online; accessed 2019-05-25].
[32] Yu, H. F., Hsieh, C. J., Si, S., Dhillon, I. S. (2014). Parallel matrix factorization for recommender systems. Knowledge and Information Systems, 41(3), 793-819.
[33] Zhang, J., Pu, P. (2007, October). A recursive prediction algorithm for collaborative filtering recommender systems. In Proceedings of the 2007 ACM conference on Recommender systems (pp. 57-64). ACM.