Data Science Lab 2018: A shopping assistant for Digitec Galaxus

Mélanie Bernhardt
ETH Zürich
melanibe@ethz.ch

Mélanie Gaillochet
ETH Zürich
gamelani@ethz.ch

Laura Manduchi
ETH Zürich
lauraman@ethz.ch

Abstract

The goal of this project was to provide the most comfortable and efficient way for customers to find the product they need. More precisely, our aim was to find the optimal sequence of questions such that the customers were offered a reduced subset of products after answering the least number of questions possible. To achieve this goal, we implemented a greedy algorithm that selects the best question to ask next by maximizing the mutual information between the product and the question at each step. Since this algorithm was computationally very expensive, we increased the computational speed of our system by using the Imitation Learning procedure. Both algorithms (Greedy and Imitation Learning) were then evaluated in terms of number of questions asked to the user, under different noise settings, to output a subset of less than 50 products. Finally, we created a user interface to show how our model could be used in practice.

1 Introduction

Currently, customers browsing on Digitec's website can filter out products using up to 27 different filters. This is not very convenient for the user. Hence, our goal was to develop a new method that interactively asks the users the minimal number of questions, one by one, in order to progressively restrict the set of products according to their needs. At each time-step, a question is selected according to the history of questions-answers. The customer is then asked to provide an answer to this question (including 'I don't know'). The procedure is then repeated until the set of remaining products is smaller than a certain threshold (here 50 products). Our prototype was implemented for the 'notebook' product category of Digitec's product offer. Nevertheless, it can be used for any product category.

We first implemented a greedy algorithm that generates the next question to ask, based on maximizing the mutual information between a possible question and the products. To increase the responsiveness of our prototype, we then used Imitation Learning to train a deep neural network to mimic our algorithm. Additionally, we developed a user interface that demonstrates how our algorithm can be used in practice.

2 Data preprocessing

Digitec's database gave us access to the following tables, for products in the notebooks category:

- **Product catalog** containing information on the products such as filters (i.e. questions) and properties (i.e. answers)
- **Purchase table** containing history of past purchased products.
- **Traffic table** containing the list of filters used by a user for a given purchase session.

Besides from extracting the relevant tables, our preprocessing involved redefining some answers to the questions (i.e. filters). If the answer was an 'option' (i.e. discrete set of answers such as 'yellow', 'Apple' or 'English'), we did not change it. However, for the continuous case, we created bins of values such that the user had no more than 10 options to choose from. For example, this avoided to have answers such as '460g', '461g' or '521g' when the user was asked about the computer's weight.

In the next sections, we refers to 'questions' as the filters provided in the product catalog, and 'answers' as the processed properties associated to the filters.

3 Greedy algorithm

Given a threshold T, the general purpose of our algorithm is to ask the minimal amount of questions to the user to output a subset of at most T matched products. A product matches with the history of answers if the answers corresponds to its properties. For example, if the first question is 'color' and the answers given by the user are 'blue' and 'yellow', then all blue and yellow products are potential matches.

This problem can be translated as sequential selection in the optimal information gathering framework. In this setting, the goal is to obtain the most informative data in the minimum amount of actions, and to find the best policy that selects the next action to carry out, depending on previously made observations.
We define as 'state' the history of answers given by the user and the 'action' is the next question to ask. The action will then produce a response (user's answer) and the state will be updated accordingly.

One of the most promising methods to face this problem, is to greedily reduce the uncertainty of the potential subset of products to return, using the Shannon mutual information (see Chen et al. (2015) and Chen et al. (2017)). Interestingly this method is shown to give optimal results in different noise setting (see Section 4), a property that is particularly appealing for a real world application such as this one.

To introduce the algorithm formally, let's first define two random variables: $\boldsymbol{Y}$ represents the target product, and $\boldsymbol{Q_i}$ represents question i where $i = 0, ..., 135$ (in our data 136 questions are present). The realizations of $\boldsymbol{Q_i}$ are the possible answers $a_1, ..., a_{m(i)}$ where $m(i)$ that depends on the question Q_i. At each time step, the set of possible products is reduced according to the answers given by the user. Similarly the set of possible questions is also updated, eliminating questions that have already been asked previously. For this reason, we will define as $setQ(t)$ the set of possible questions we can ask at time step t. The history of questions and answers asked before time step t is defined as $\boldsymbol{H_t}$ and h_t its realization (set to an empty list for the first time step).

The algorithm selects the next question that maximizes the mutual information between $\boldsymbol{Q_i}$ and $\boldsymbol{Y}$ given the history.

$$Q_{opt[t]} = \arg \max_{\boldsymbol{Q_i} \in setQ(t)} \mathrm{I}(\boldsymbol{Q_i}; \boldsymbol{Y} \mid \boldsymbol{H_t} = h_t) \tag{1}$$

The mutual information is defined as :

$$\mathrm{I}(\boldsymbol{Q_i}; \boldsymbol{Y} \mid \boldsymbol{H_t} = h_t) = H(\boldsymbol{Y} \mid h_t) - H(\boldsymbol{Y} | \boldsymbol{Q_i}, h_t) \tag{2}$$

where $H(\boldsymbol{Y}|h_t)$ refers to the entropy of $\boldsymbol{Y}$ given the history and $H(\boldsymbol{Y}|\boldsymbol{Q_i}, h_t)$, the conditional entropy between the two random variables $\boldsymbol{Q_i}$ and $\boldsymbol{Y}$ given the history.
The first term of eq. (2) can be discarded since it does not depend on $\boldsymbol{Q_i}$. Regarding the second term, it can be rewritten as:

$$-H(\boldsymbol{Y}|\boldsymbol{Q_i}, \boldsymbol{H_t} = h_t) = \sum_{a \in [a_0 \dots a_{m(i)}]} p(a \mid h_t) \sum_{y \in Y} p(y \mid a, h_t) \log(p(y \mid a, h_t))$$

In this formula, $p(y \mid a, h_t)$ is a linear combination between two different distributions: the uniform distribution and the one derived by the history of products bought in the past.

More precisely, let's define as $N_{a,h}$ the number of products that match h and a, N_b the total number of products bought in the past and $N_{a,h,b}$ the number of products bought in the past that match h and a, then:

$$p(y \mid a, h_t) = \frac{1}{Z}\left(\frac{1}{N_{a,h}} + \alpha\frac{N_{a,h,b}}{N_b}\right) \tag{3}$$

where Z is the normalization constant to get a probability distribution. We tuned the α hyper parameter to 1, such that the uniform distribution and the one derived by the history of products bought in the past had the same weight.

Similarly $p(a \mid h_t)$ is the estimated probability of answering a to the question given the history of answers. We estimated this probability also as a linear combination of the uniform distribution and the one derived by the history of answers given to question q in the past. We will denote the hyperparameter controlling the importance given to the history of answers by β in the rest of this report. We tuned this β hyperparameter to 1.

Algorithm 1 gives the steps of our greedy algorithm, given a randomly sampled target product.

Algorithm 1 Max MI Algorithm

```
 1: Get Product = ProductSet, Bought = PurchasedTable and Traffic = TrafficTable
 2: Set Threshold = 50
 3: Set $\alpha = 1, \beta = 1$                ▷ Assign weights to be given to history for sampling Y
 4: Y ← sample(distributionY)
 5: q_list ← []
 6: while |Product| > Threshold do
 7:     q ← $\arg\max_q$(I(q, Y | Product))
 8:     q_list.append(q)
 9:     a_y ← get_answer(q, Y)                  ▷ Gets property of product Y for given filter
10:     Product, Bought, Traffic ← subset(q, a_y, Product, Bought, Traffic)      ▷ Reduce tables
    return q_list, Product, Y
```

As some questions available in the database are very specific, our algorithm tends to ask question that a real user would not be able to answer. For example, one such question would be "specify the model of battery desired for your computer". Hence, in order to prevent our algorithm from asking too specific questions and thus to make it more user-friendly, we introduced a prior on $\boldsymbol{Q_i}$. This allows us to give more weight to questions that had previously been used by users (from the real historical data). The prior is computed as a linear combination between a non-informative uniform probability (to prevent having 0 probabilities for questions that have never been used in the past) and a probability distribution that depends on the history of filters used in the past. This is done in a very similar way to what is described in eq. (3). In the rest of this report we will denote by a the hyperparameter that controls the importance given to the history of used filters.

With this improvement, equation (1) then becomes:

$$Q_{opt[t]} = \arg\max_{\boldsymbol{Q_i} \in setQ(t)} p(Q_i)\mathrm{I}(\boldsymbol{Q_i}; \boldsymbol{Y} \mid \boldsymbol{H_t} = h_t)$$

4 Noisy setting and user simulation

Given that our algorithm was intended to be used interactively, we needed to simulate a user that could provided answers to the given questions in order to evaluate its performance. Hence, we built a user simulator, ran our algorithm based on answers provided by the simulated user, and recorded the number of questions asked until the set of products left was smaller than our threshold T. The procedure is described in more details in the Section 6.

The simulation of a user goes through the following steps:

- Sample a product from the product catalog, representing the simulated user's ideal product. Hence, the goal is to ask the minimum number of questions to return a small subset of products with this ideal product and 48 similar ones (given our threshold of strictly inferior to50)

- Simulate the user's answers to all questions available. The answers represent our user's prior knowledge about his target product, or in other words, his preferences regarding some properties of the product. To account for the fact that the user might not know exactly which properties he would like his product to have, or properties he does not care about, we imagined several noise settings. These noise settings allow us to sample different answers given a certain product and are described below.

Different noise settings:

- Non-noisy setting: given the sampled product, the users knows all the 'true' answers to the questions (i.e. properties of the product as saved in the product database). This perfect user always provide a unique 'true' answer to the questions asked by our algorithm. It is important to notice that if an answer is not provided in the database (i.e. missing property value in the product set), then the answer is considered as 'I don't know'. In this case, the reduced set of products is not updated and the algorithm goes to the next question.
- Probability of answering more than one answer: in real life, the user may choose several answers to a question because he or she might not be sure about the type of product he or she would like to buy precisely. We considered the case in which at least one answer matches with the final product's properties but not necessary all of them, or in other words the 'true' answer was always contained in the set of given answers. We did so by adding probability of giving 2 and 3 answers to the question.
- Additional probability of answering "I don't know": we also wanted to investigate the impact of increasing the probability to answer "I don't know" compared to the perfect user.

5 Imitation learning

Although our greedy algorithm was optimal, running it was too computationally expensive and was not a viable solution to directly answer customers' requests in real-time. For this reason, we turned to Imitation Learning (Attia et al. (2018)).

In the Imitation Learning setting, we assume that we have access to a teacher (here our greedy algorithm) that already knows the best policy to follow. A student then observes the teacher's actions and learns to imitate its behavior. In our specific setting, we considered Imitation Learning by classification, where the model attempts to mimic the teacher's action by choosing this action among a discrete set of possible actions (i.e. possible next questions, each question representing a class).

In our case, a deep neural network of 4 fully connected layers (details in Section 5.3) was trained to imitate questions produced by our greedy algorithm.

We first implemented the Dataset Aggregation (DAgger) algorithm as pipeline for the learning process. However, since we were able to generate a huge initial training dataset (> 70,000 data points), it turned out that the use of DAgger could not significantly improve classical imitation learning (i.e. where there is no exploration of new states after initial training). In the next section we will first detail how DAgger works, before explaining its behavior on our dataset. Finally we will present our model and the final training procedure.

5.1 DAgger

Supervised imitation learning may fail when the data given by the teacher is not enough, this is because as soon as the trained policy makes a mistake it can go 'off the expert path' and encounter states that it has never seen before. Ideally, we would like to train our policy so that it is able to deal with any possible situation it could encounter. DAgger (Ross et al. (2011)) is an iterative algorithm that attempt to solve this problem.

First, it collects N different trajectories $\tau_n = s_0, q_1, a_1, s_1, q_2, a_2, ...$ from the teacher. Each trajectory is composed of states (s_i), actions (q_i) and responses (a_i). In our framework, state s_i refer

to the sequence of question-answers obtained up to step i, the action q_i is the next question asked, and responses a_i are the answers given by a user. The state is updated after every question and associated answers are provided. Each trajectory is then divided to form a dataset of the form: {state $\rightarrow$ next question}. This dataset is used to pre-train a neural network, where the inputs are the states and the labels are next questions asked, and to obtain the first trained policy.
In a second part, DAgger deviates from the classical Imitation Learning by classification and begins to explore new states by iteratively updating the training dataset. At each times step, the user answers the question outputed by the trained policy. The teacher is called to predict the next question, and the new pair (state, next question) is then added to the training set. In the classification task, the next question represents the label assigned to the state. The trained policy then predicts the next question, that could be different from the one predicted by the teacher, based on the current state and it updates the state.
We call as episode one complete run of the student algorithm (until the threshold is reached). For each episode we get several (state, next question) pairs to add to our dataset. Every 200 episodes, the student network is retrained on the aggregated dataset, and the procedure is repeated. The pseudocode is shown in Algorithm 2.

Algorithm 2 Training DAgger

```
1: $\langle \tau_n^{(0)} \rangle_{n=1}^N \leftarrow$ run Max MI algorithm N times
2: $D_0 \leftarrow \langle (s, next\_q) : \forall n, \forall (s, next\_q) \in \tau_n^{(0)} \rangle$        $\triangleright$ $s = \{q_1 : [a_{11}, a_{12}], q_2 : [a_{21}], \dots\}$
3: $f_0 \leftarrow classifier(D_0)$
4: q_list $\leftarrow$ []
5: **for** $i = 1 \dots MaxIter$ **do**
6:     $\langle \tau_n^{(1)} \rangle_{n=1}^M \leftarrow$ run policy $f_{i-1}$ M times
7:     $D_i \leftarrow \langle (s, MaxMI(s)) : \forall n, \forall (s, next\_q) \in \tau_n^{(i)} \rangle$  $\triangleright$ new training data is pairs (state, next
   question predicted by Max MI)
8:     $f_i \leftarrow classifier(\bigcup_{j=0}^{i} D_j)$
   **return** $f_0, f_1, \dots, f_{MaxIter}$
```

5.2 Getting teacher data

We first generated the initial training data from our greedy algorithm (the 'teacher') by running it on products chosen at random from the product set. Recall that sampling one product and the corresponding answers is equivalent to simulating a user (see Section 4). To account for the popularity of certain products, the probability of a product being selected was given by the product distribution (estimated as described in Section 3). Running the greedy algorithm on each sampled product yielded a list of explored states (list of questions asked by the algorithm up to time step t and corresponding sampled answers) and associated labels (i.e. next question to ask at time step t). In order to make our network more robust, we generated states with our teacher under different noise settings and aggregated all the teacher runs to construct our initial training set. By running the algorithm on 7000 sampled products (with replacement), we were able to generate 70,436 training states.

5.3 Training and tuning the model

We defined the state as a dictionary where the keys were the questions already asked and the values were the corresponding answers given by our sampled user. In order to use these states as input to our neural network, we needed to transform this dictionary into a vector input. Therefore, we converted each input of the dictionary to its one-hot representation.
We also defined a secondary input to our model: the 'mask'. The 'mask' was used to prevent our network from predicting a question that had already been asked. It was a simple vector such mask[i] $= 0$ if question i had already been asked and mask[i] $= 1$ otherwise. Hence, by multiplying the original predicted class probabilities by this mask vector we obtained our final output out$_i$ such

that if i had already been asked then $\text{out}_i = 0$, else $\text{out}_i = \text{probas}_i$, where probas_i was the original output probability layer of the network for question i.

In order to tune the hyperparameters and the architecture of our model, we first focused on the initial training (i.e. we only trained on the initial dataset without further steps generation), using a validation split of 20%. We tried several architecture (varying the number of hidden layers and the number of hidden units). Our final architecture is described in Table 1. With this network, we obtained a validation accuracy of 60% on 136 possible classes, where the validation set was a random split of 20% of the initial training set after the initial training phase (before exploration of new states).

Table 1: Architecture of our neural classifier

Layer	Type of layer	Layer parameters	Activation function
h1	Fully Connected	2048 output units	ReLu
dropout1	Dropout	$p = 0.5$	-
h2	Fully Connected	1024 output units	ReLu
dropout2	Dropout	$p = 0.3$	-
h3	Fully Connected	512 output units	ReLu
dropout3	Dropout	$p = 0.3$	-
h4	Fully Connected	1024 output units	ReLu
dropout4	Dropout	$p = 0.3$	-
probas	Fully Connected	<number_filters> output units (for us, <number_filters> = 135)	SoftMax
out	Custom	$\text{out}_i = \text{probas}_i * \text{mask}_i$	-

Our model was trained using the Adam Optimizer with a learning rate of 0.001. In order to prevent our model from overfitting the training data we used early stopping on the validation accuracy and a patience parameter set to 4.

5.4 Going back to classical supervised imitation learning

Figure 1 shows the training and validation loss and accuracy obtained during the training of DAgger. The vertical dashed lines delimit each ‘training round’. The initial training round corresponds to the training epochs on the initial training set. Subsequent training rounds each correspond to the training of the aggregated dataset after 200 episodes (as explained in Section 5.1).
However, in practice, iteratively retraining the model as described in the DAgger algorithm did not improve the accuracy obtained by our network. On the contrary, it slowly led to an overfit of the training data. This is can be explained by the fact that the ratio of newly added data points over the size of the initial training set (roughly 1000 points added after 200 episodes versus 54,000 initial training points) was extremely small, meaning that the model was unable to learn effectively during retraining.

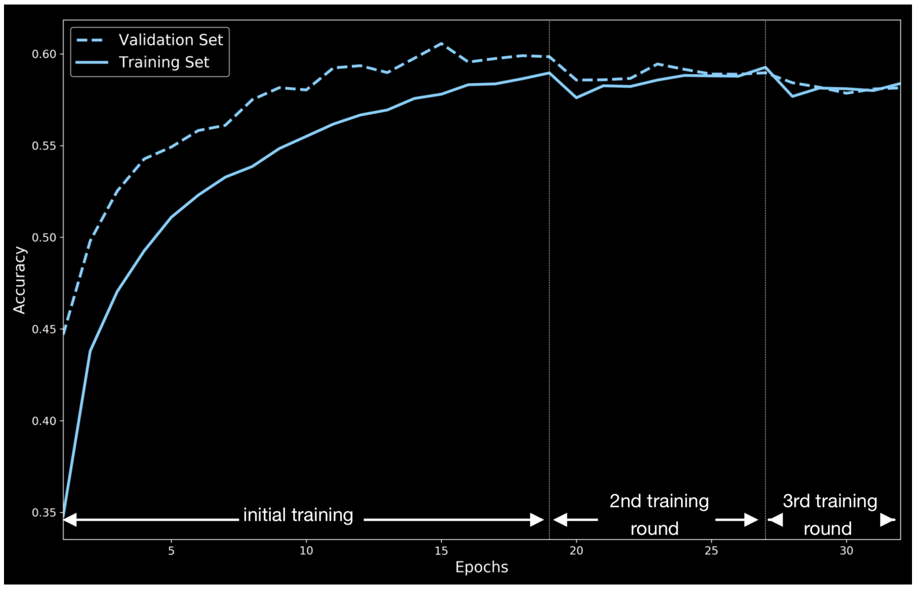
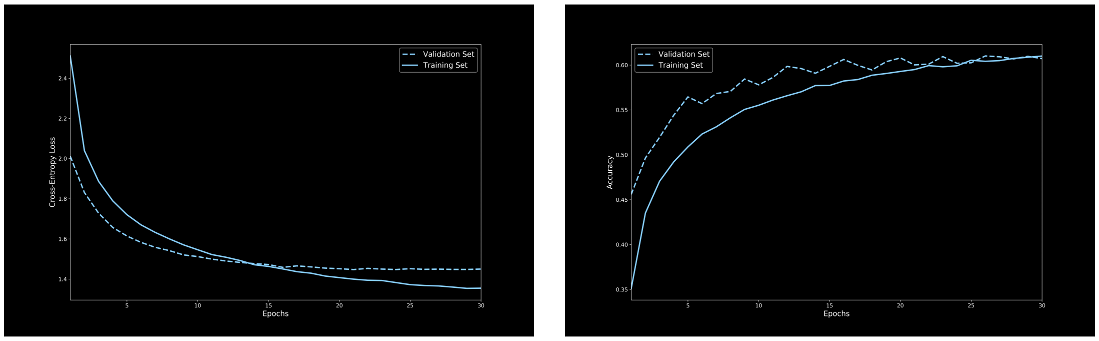


Figure 1: Training and validation accuracy for DAgger algorithm.

Our model was already satisfying after the initial training round, hence we chose not to follow DAgger's iterative training and stopped after the initial training phase. This corresponds to the traditional classical supervised learning setting where the data from the greedy algorithm is used altogether during training. Figure 2 shows the training and validation accuracy and loss for our final model (trained solely on the teacher dataset). The final validation accuracy was of 60%.



Training and validation loss

Training and validation accuracy

Figure 2: Training and validation accuracy and loss for our final model (trained solely on the teacher dataset)

6 Evaluation and results

The results of our experiment are shown in Table 6

The aim of our algorithm was to ask the minimal number of questions until the product set size was reduced to a predefined threshold. As explained in Section 4 we used several noise settings to simulate different kind of users. This allowed us to evaluate the performance and robustness of our algorithms. We compare the performance of our algorithm to the random baseline (i.e. where we simply select one question uniformly at random as the next question among the set of remaining questions).

Our final evaluation procedure can be described as follows:

- Choose one noise setting and a test set size

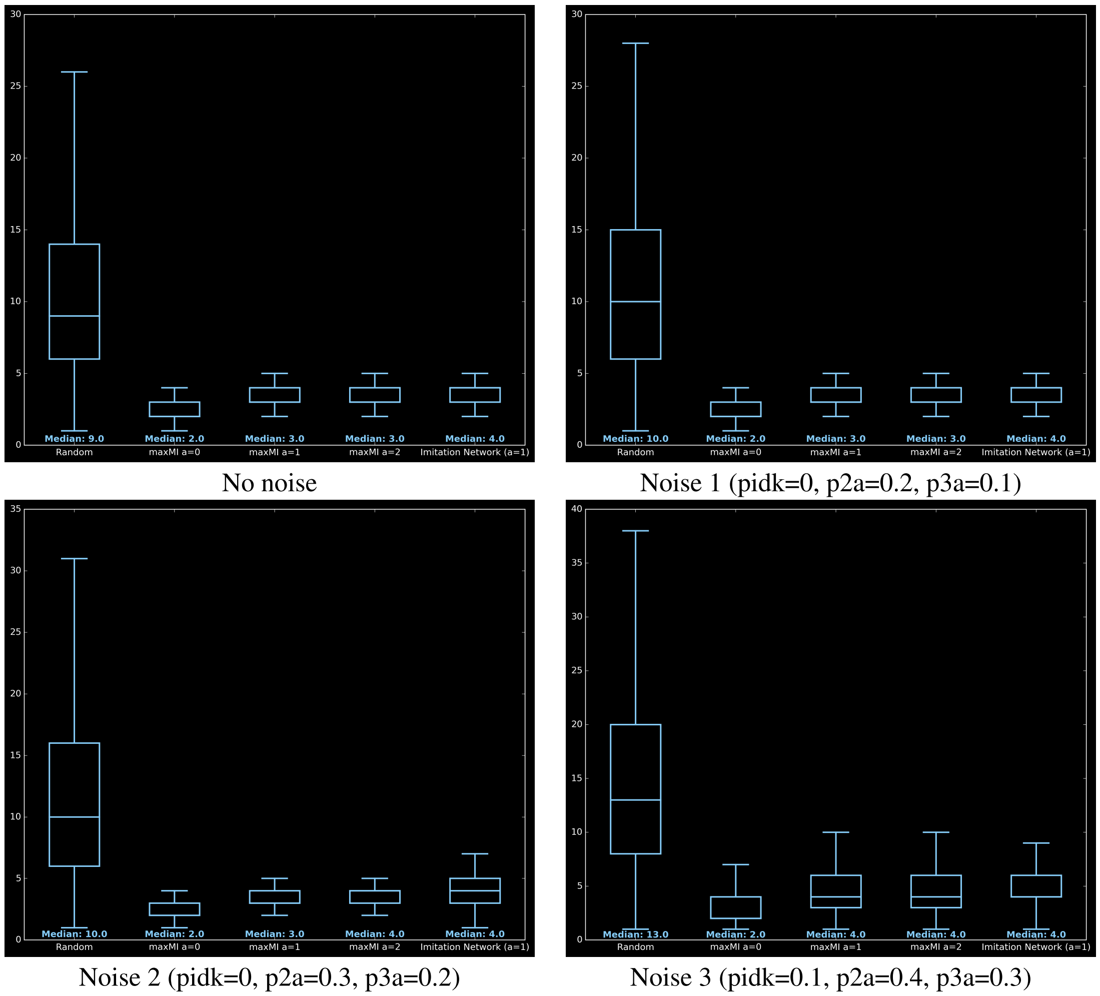


Figure 3: Boxplots of the number of question asked per user in 4 different noise settings for our greedy algorithm and our Imitation Network. a refers to the hyperparameter used to give more importance to questions used by a real previous user (from the historical traffic data). Test set had size 1000 products. Top whisker of the boxplot is defined as Q3+1.5(Q3-Q1) and the lower whisker as Q3-1.5(Q3-Q1). Outliers are not depicted for clarity of the graphs.

- Until you have reached the test set size:
  - Sample one user (i.e. one target product and the corresponding answers to questions representing the knowledge of the simulated user) according to the chosen noise setting.
  - Run the Greedy or DAgger algorithm to get the predicted list of questions to ask to the simulated user. Save the questions, answers and number of questions asked to the users.
  - Repeat the procedure for the random baseline

Recall that the main problem with the greedy algorithm was the running time. Getting the full list of question for one product took up to 60 mins (in the worse case where the user only knows very few answers). On a standard laptop, finding the next question could take up to 2 mins (which is way too long to be used interactively). On the contrary, once DAgger was trained obtained the list of question from the saved model was nearly done instantly.

Table 6 shows the performance comparison of our algorithms and the random baseline under four different noise settings. To get a more visual interpretation of the results we depicted the results as boxplots in Figure 3. First, we can see that our models significantly outperform the random baseline. Secondly, giving more weight to previously used filters i.e. making our algorithm more

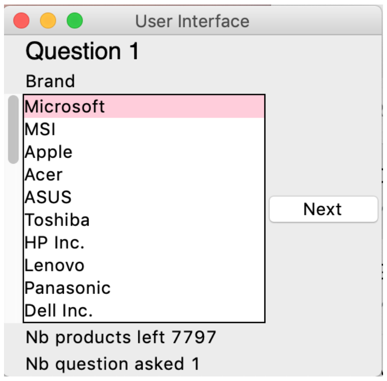
Table 2: Results of our experiment, given in terms of number of questions asked to the user before reaching the threshold of 50 products, for a test set of 1000 sampled products

Noise setting	MaxMI a=0	MaxMI a=1	MaxMI a=2	DAgger (a=1)	Random
No noise	Median: 2.0 Avg (std): 4.16 (10.5)	Median: 3.0 Avg (std): 9.13 (20.6)	Median: 3.0 Avg (std): 7.83 (19.2)	Median: 4.0 Avg (std): 10.56 (25.8)	Median: 9.0 Avg (std): 14.97 (19.7)
Noise 1 pidk=0, p2a=0.2, p3a=0.1	Median: 2.0 Avg (std): 4.03 (10.3)	Median: 3.0 Avg (std): 10.26 (24.4)	Median: 3.0 Avg (std): 9.99 (23.9)	Median: 4.0 Avg (std): 10.53 (25.0)	Median: 10.0 Avg (std): 16.53 (21.0)
Noise 2 pidk=0, p2a=0.3, p3a=0.2	Median: 2.0 Avg (std): 3.40 (5.9)	Median: 3.0 Avg (std): 10.89 (25.6)	Median: 4.0 Avg (std): 11.65 (26.6)	Median: 4.0 Avg (std): 12.77 (29.4)	Median: 10.0 Avg (std): 17.96 (23.1)
Noise 3 pidk=0.1, p2a=0.4, p3a=0.3	Median: 2.0 Avg (std): 5.94 (16.9)	Median: 4.0 Avg (std): 12.53 (28.0)	Median: 4.0 Avg (std): 13.20 (29.9)	Median: 4.0 Avg (std): 14.46 (31.5)	Median: 13.0 Avg (std): 21.55 (26.2)

real user-friendly does not have a huge negative impact on the simulated performance. Thirdly, our algorithms turn out to be not very noise sensitive which is preferable as in reality users' answers will not be perfect. Finally, we can see that the performance of our Imitation Network is very similar (nearly identical) to the performance of our optimal greedy algorithm.

7 User interface

To showcase our prototype model, we created a interactive user interface. The interface was constructed with the `Tkinter` Python library. This interface uses our algorithms (choice between MaxMI or DAgger) on the back-end to compute the next question to ask to the user. The user can choose one or several answers among those displayed, and the algorithm then computes the next best question to ask. Once the size of the set of possible products to choose becomes less than teh threshold, the interface stops asking questions and returns the list of ProductId's (the true name of products were not available in the dataset at hand for this project). Figure 4 shows how our interface looks like.



Question 1

Brand

Microsoft
MSI
Apple
Acer
ASUS
Toshiba
HP Inc.
Lenovo
Panasonic
Dell Inc.

Nb products left 7797

Nb question asked 1

Figure 4: Screenshot of the first question given by the greedy algorithm in our user interface

8 Conclusion

We build a fast interactive solution to return a selected subset of products to our user as fast as possible by asking him/her an optimally determined question, one at a time. After having implemented a greedy algorithm based on maximizing the mutual information between the questiosn and a target product, we also built a neural network capable of imitating this policy with an high accuracy (despite the high number of possible questions). The performance of the imitation network is comparable to the performance of its greedy teacher in various noise settings, and both algorithms outperform the random baseline in terms of total number of questions asked, and are robust against noisy answers. Finally, we implemented a Python interactive interface in order to demonstrate the usability of our solution.

References

Chen, Y. & Hassani H. S. & Karbasi, A. & Krause, A. (2015) Sequential Information Maximization: When is Greedy Near-optimal?. *JMLR: Workshop and Conference Proceedings*, **40**:1–26.

Chen, Y, & Hassani H. S. & Karbasi, A. & Krause, A. (2017) Near-optimal Bayesian active learning with correlated and noisy tests. *Electronic Journal of Statistics*, **11**:4969–5017.

Yue Y. & Le H. M. (2018) *Imitation Learning Tutorial ICML 2018* `https://www.reddit.com/r/MachineLearning/comments/90dcls/r_icml_2018_tutorial_on_imitation_learning/`

Daumé III H. (2017) A Course in Machine Learning

Attia A. & Dayan, S. (2018) Global overview of Imitation Learning

Ross, S. & Gordon, G. J. & Bagnell J. A. (2011) No-regret reductions for imitation learning and structured prediction. CoRR, abs/1011.0686, 2010.