Improving user experience by ANN prediction and NLP chatbot

Diana Bratić[1], Kristina Puhanić[1], Denis Jurečić[1]

[1]University of Zagreb, Faculty of Graphic Arts, Getaldićeva 2, 10 000 Zagreb, Croatia
diana.bratic@grf.unizg.hr

Abstract

As communication between service providers and users increases daily due to the large supply and demand for services, the application of artificial intelligence in business domains can greatly facilitate and improve characteristics such as the speed and efficiency of communication through the implementation of NLP chatbots presented in this work. This work aims to investigate the consequences of NLP chatbot implementation and the reliability of ANN prediction in terms of user experience, and to determine whether the developed NLP chatbot application matches the ingenuity in understanding communication in terms of algorithm reliability. It also aims to verify whether differences in the improvement of the user experience by NLP chatbots can be detected through a comparative analysis and what impact the speed of NLP chatbots has on the user experience compared to a chatbot without NLP function.

The programming of the application and the simulation were done with the *Python* programming language in the *PyCharm* integrated development environment using the artificial neural network model and libraries such as *PyTorch*, NLTK and *Tkinter*. The creation of the experimental part consisted of the creation of an NLP chatbot application, the creation of a simple chatbot represented by a simulation display, and the carrying out a comparative analysis between an NLP chatbot and a rule-based chatbot.

The proposed model, primarily the principle itself, can be used for other domains that could significantly utility from the implementation of NLP chatbots, and not only for the domains highlighted in this paper.

Keywords: NLP, chatbot, AI, user experience, PyTorch

1. INTRODUCTION

The influence of artificial intelligence or AI in the business domain can greatly facilitate and improve parameters such as the speed and efficiency of communication through the implementation of an NLP chatbot that has the ability to communicate with the user through text or audio methods.

Therefore, in this paper, a comparative analysis of the NLP chatbot and the rule-based chatbot, i.e., the rule-based chatbot that does not use artificial intelligence, is performed. Both chatbots were programmed in the *Python* programming language using the *PyCharm* integrated development environment. To create the NLP chatbot application, it was necessary to collect the data needed for training, perform *tokenization* and word stem reduction, and create the *bag-of-words* model. It also used *PyTorch*, an open-source machine learning framework based on the *Python* programming language and the *Torch* library. It was tried to implement a neural network model (*Artificial Neural Network - ANN*) using the *Feed-Forward* model rules to obtain a neural network with two hidden layers. In the last step, the quality of

reliability of the algorithm was investigated by prediction. To create NLP chatbot applications, *Tkinter* was used, a standard *Python* library used to create graphical user interfaces for work applications. The aim of the work was to investigate the capabilities of NLP chatbot implementation and the reliability of ANN predictions in the user experience related to a rule-based chatbot. It was investigated whether the ANN prediction enables the user experience improvement through the implementation of an NLP chatbot in terms of the reliability of the algorithm, what effect the speed of the NLP chatbot has on the user experience in terms of the rule-based chatbot, whether the simulation of the NLP chatbot achieves ingenuity in understanding communication in terms of the reliability algorithm, and whether you affect the application of NLP chatbots to user experience improvement in terms of rule-based chatbots.

2. THEORETICAL BACKGROUND

Users of various online services are overwhelmed by what is available. Therefore, user satisfaction is a top priority for online businesses, and it can best be provided through chatbot customer support [1, 2]. Today, chatbots are expected to be at the level of human understanding in the communication process. Therefore, in addition to the already well-known rule-based chatbots, NPL-based chatbots are increasingly being integrated.

Artificial intelligence algorithms make human language understandable to machines. Natural Language Processing (NLP) is defined as the automatic software manipulation of natural language, such as speech and text. NLP is used to understand the structure and meaning of human language by analysing various aspects such as syntax, semantics, pragmatics, and morphology [3].

An AI chatbot is software that communicates with a person through written language, i.e., a computer program that simulates human communication. It is often embedded in websites or other digital applications to answer customer queries without the need for human agents. AI chatbots are often used in customer service [4].

Rule-based chatbots are also called decision tree bots [5]. Like flowcharts, rule-based chatbots map conversations by predicting what a user might ask and how the chatbot should respond. These chatbots do not learn through interactions, only perform scenarios for which they have been trained, and cannot answer questions outside of the defined rules [6, 7].

3. NLP CHATBOT APPLICATION DEVELOPMENT

The Czech company *JetBrains'* integrated development environment, *PyCharm*, and the *Python* programming language were used to program the application and set up chatbots, and simulations of conversations between chatbots were created. It was especially important to study parameters such as the speed of the NLP chatbot and its ingenuity in understanding how to communicate with the user [8, 9].

To create the NLP chatbot application, training data was first collected, then natural language processing models were created, trained, and implemented.

The work utilized *PyTorch*, an enhanced deep-learning tensor library. Applications that use both the GPU and CPU can use this library, which is based on *Python* and *Torch*. A *Python* package called the NLTK (*Natural Language Toolkit*) library is also used in NLP and houses tools that let computers comprehend human language.

3.1. Data collection

There are a few tags that have been chosen as illustrations of samples that exhibit the NLP pre-processing flow steps depicted in Figure 1. They are *tokenization*, *stemming*, *lower stemming*, and the *bag-of-words* model.

Tokenization, or the division of text data into words, terms, sentences, symbols, or other meaningful parts, is a requirement for a computer to comprehend any text. *Stemming* or root word reduction is another crucial stage in the NLP preparation sequence for data collection. *Bag-of-words* or abbreviated BoW is a textual representation of word occurrence in a document that comprises a measure of the frequency of known words as well as a dictionary of known terms. The model simply cares about recognized terms appearing in the document, not about where they appear [10]. After installing the *nltk* package, it was feasible to run the *bag-of-words model* whose program code is shown in Figure 2. with an example for the sentence *"Good day, what services do you offer?*

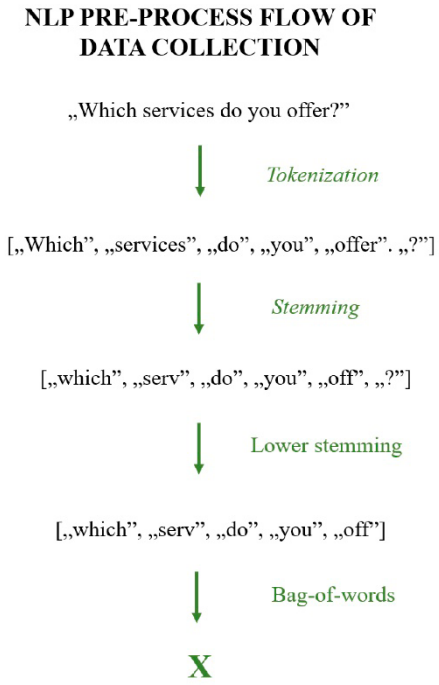

Figure 1: NLP pre-process flow of data collection
Source: own

```
15  def bag_of_words(tokenized_sentence, all_words):
16
17      tokenized_sentence = [stem(w) for w in tokenized_sentence]
18
19      bag = np.zeros(len(words), dtype=np.float32)
20      for idx, w in enumerate(words):
21          if w in tokenized_sentence:
22              bag[idx] = 1
23
24      return bag
25
26  sentence = ["Good", "afternoon", "which", "services", "do", "you", "offer", "?"]
27  words = ["good", "afternoon", "how", "are", "you", "see", "you"]
28  bag = bag_of_words(sentence, words)
29  print(bag)
```

Figure 2: Program code of Bag-of-words model
Source: own

3.2. Training set script

By executing the program code from the supplied script, a training script named *train.py* was constructed in order to build an NLP chatbot, and a script named *data.pth* was created as a result. Furthermore, a *json* script containing all the necessary tags along with the answers is imported into the *train.py* script. UTF, which stands for *Unicode Transformation Format*, and 8 indicate that the encoding uses 8-bit values. Then the *ChatDataset* class was created, which contained the *x* and *y* training set data. Figure 3 shows the program code of the *ChatDataset* class.

```
class ChatDataset(Dataset):

    def __init__(self):
        self.n_samples = len(X_train)
        self.x_data = X_train
        self.y_data = y_train

    # support indexing such that dataset[i] can be used to get i-th sample
    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    # we can call len(dataset) to return the size
    def __len__(self):
        return self.n_samples
```

Figure 3: Program code of the ChatDataset class
Source: own

3.3. ANN model

The implementation of the neural network model (*Artificial Neural Network - ANN*) was developed in the script *model.py*. The *PyTorch* library module had to be included in the script, and the *NeuralNet* class had to be defined. A *__init__* function was defined in the class with indexes *self*, *input_size*, *hidden_size*, and *num_classes*. The *forward* function, which converts input values back into output values, is then defined. The *reverse* function determines the gradient of the input values with respect to a given scalar value using the gradient of the output values for that scalar value as input. The *NeuralNet* class's programming code is displayed in Figure 4.

This model depicts a network of *Feed Forward* neurons in two hidden layers. Figure 5 shows the *Feed Forward* model, which represents the simplest form of a neural network since the information is processed in a single direction. Even though the data may go through several hidden nodes, they always move in one direction and never backward. A network of neurons is generally viewed in its simplest form as a one-layer perceptron. In this model, a series of entries are added to the layer and multiplied by the weights. Each value is then

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.l2 = nn.Linear(hidden_size, hidden_size)
        self.l3 = nn.Linear(hidden_size, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        # no activation and no softmax at the end
        return out
```

Figure 4: Program code of the NeuralNet class
Source: own

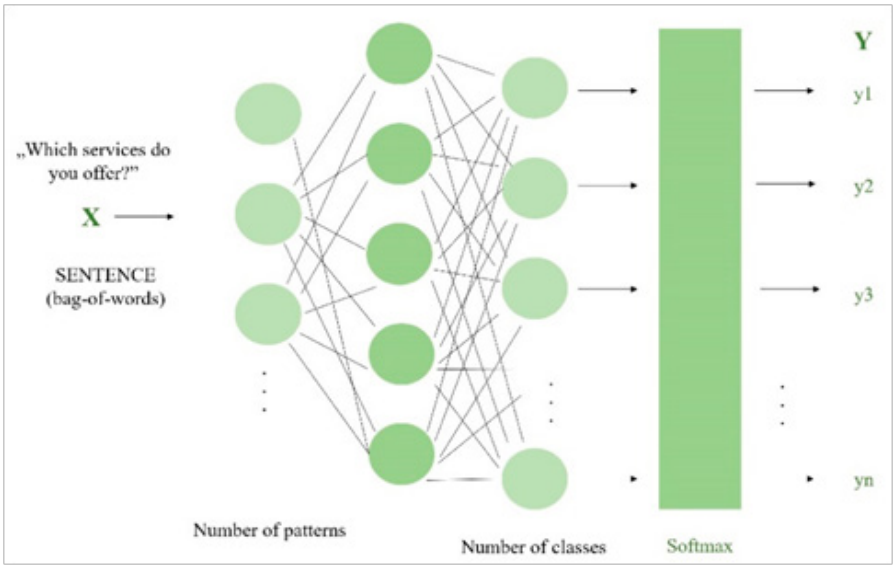

Figure 5: Graphic representation of Feed Forward neural network
Source: own

added together to obtain the sum of the weighted entry values. If the sum of the values is above a certain threshold, usually set to zero, the value produced is often 1, while if the sum falls below the threshold, the output value is -1. The single-layer perceptron is an important model of advanced neural networks and is often used in classification tasks. A vector of numbers is transformed into a vector of probabilities via the mathematical operation known as *Softmax*, where the probability of each value is inversely proportional to the relative scale of each value in the vector.

Saving and loading the model

The *train.py* script created a dictionary where various parameters were to be recorded in order to save data. A model is saved as a key, where *model_state* is named in the form *model.state_dict()*. In addition, it is required to store tags, all of the words that were initially gathered in the form of the expression *all_words*, as well as *input_size*, *output_size*, and *hidden_size* in the dictionary. Then the name of the file is defined, which in this case was the name *data.pth*, and the *torch* module is called to save the specified file. In this manner, *serialization* - the process of transforming a data object into a format that permits data storage or transmission, is accomplished. The *data.pth* script is created when the program code is executed. The model saving and loading program code is shown in Figure 6 together with the data storage dictionary.

```
117  data = {
118  "model_state": model.state_dict(),
119  "input_size": input_size,
120  "output_size": output_size,
121  "hidden_size": hidden_size,
122  "all_words": all_words,
123  "tags": tags
124  }
125
126  FILE = "data.pth"
127  torch.save(data, FILE)
128
129  print(f'training complete. file saved to {FILE}')
```

Figure 6: Program code of the data storage dictionary, as well as saving and loading the model
Source: own

Chat Implementation

The phrase "*import random*" was written in the *chat.py* script, which attempts to provide a random selection of potential answers. The constructed *Feed Forward* neural network model, the *PyTorch* library package, the *json* script, the *bag-of-words*, and the tokenizer from the *nltk_utils* script are then imported

into the script. The implementation of the dialogue starts after all essential imports have been made. The *bot_name = "Code"* expression was used to first give the chatbot *Code* a name. The program code specifies that the discussion will finish if the user types "*quit*" because the user's sentences serve as inputs. Otherwise, the conversation goes on. It attempts to tokenize the input sentences using the syntax *sentence = tokenize(sentence)*, after which a *bag-of-words* is generated using the syntax *X = bag_of_words(sentence, all_words)*. Then it is reshaped using the formula *X = X.reshape(1, X.shape[0])*, where *0* denotes the number of columns and *1* denotes the number of rows. Additionally, *X* is changed into *torch* using the formula *X = torch.from_numpy()*. An array of negative integers is used to index a grid of items that are all of the same types. The rank of the array is its total number of dimensions, and its shape is a set of integers that determines how big the array is along each dimension. Prediction is enabled by the provided equation *predicted = torch.max(output, dim=1)* from Figure 7. By the using the formula *tag = tags[(predicted.item())*, where *predicted.item()* stands in for the class label, an attempt is made to read the tags and establish their prediction. A loop was made over all of the intents from the *json* script since an attempt was made to identify a match for the *tag*, and it would be discovered that the loop contained all of the intents. If the tag matches the intent with the key tag, the response and bot name are printed, according to the expression *if tag == intent["tag"]:* written in the loop. It was also crucial to verify that if the tag's probability was high enough, *softmax* would be used in the formula *probs = torch.softmax(output, dim = 1)*, where the predicted item index's real probability was equal to 0. The solutions are printed if the probability is high enough, or larger than 0.75. However, the program outputs "*I don't understand, can you explain?*" when the probability is lower.

```
    output = model(X)
    _, predicted = torch.max(output, dim=1)

    tag = tags[predicted.item()]

    probs = torch.softmax(output, dim=1)
    prob = probs[0][predicted.item()]
    if prob.item() > 0.75:
        for intent in intents['intents']:
            if tag == intent["tag"]:
                return random.choice(intent['responses'])

    return "I don't understand, can you please explain?"
```

Figure 7: Prediction program code
Source: own

NLP chatbot app

Although an application was constructed to test the ANN chatbot, the procedure of constructing an application that incorporates GUI and Tkinter is so complex that it is not described here due to its excessive complexity.

3.4. Creating a rule-based chatbot

A relatively straightforward method was employed to construct this chatbot, which uses the identified keywords to print responses.

Responses script

The *random* module was first imported into the *long_responses.py* script, which was then produced. The foundation function *random*, which creates a random float uniformly in the semi-open range *[0.0, 1.0)*, is the basis for almost all of the module's functions. The functions provided by this module are bound methods of a hidden instance of the *random.Random* class. The *Random* class can also be subclassed if a different base generator is to be used. The *unknown* function, which is defined in the *long_responses.py* script and contains all the required responses, is shown in Figure 8 which displays the brief answer's software code.

```
7   def unknown():
8       response = ["Could you please express yourself differently?",
9                   "I don't understand, explain it differently.",
10                  "What does that mean?",
11                  "I don't get it, can you rephrase what you wrote?"][
12          random.randrange(4)]
13      return response
```

Figure 8: Program code of short answers
Source: own

Program code for long answers is also created in this script, but it is defined through a separate variable and is referenced in the *main.py* script through such a variable. Figure 9 shows the program code of long answers.

```
3   R_RECOMMENDATION = " Just check what you wrote on the internet! "
4   R_COMPETITORS = "I don't think about my competitors because I'm a chatbot!"
```

Figure 9: Program code of long answers
Source: own

Prediction

Two parameters were imported into a *main.py* script using the expressions *import re* and import *long_responses as long*. A *re* or regular expression specifies the set of strings that match it. After import, an infinite *while true* loop is created in the script, through which new answers can always be obtained. While it is true, the chatbot prints a specific text for a specific input parameter, therefore a *get_response* function is created in the loop that takes the input, i.e., the user's message, which it separates into two strings and moves all letters to a lower level. Before establishing an algorithm that checks all messages and finds the most suitable one, a *message_probability* function was created that calculates the probability that a message is a suitable message. Several parameters are placed in the function, and they are *user_message*, *recognized_words*, *single_response*, and *required_words*. The function also created a variable *message_certainty* which is equal to zero, and a boolean called *has_required_words* which is equal to True. To recognize text from the user's message, a *for* loop was developed.

When a word is contained in the recognized words parameter, message certainty equals one, indicating a high possibility that the statement is true. The percentage of truth was then calculated by setting the program code to equal the product of the expressions *float(message_certainty)* and *float(len(recognised_words))*. The chatbot should recognize the words that are inserted into the recognized words argument. A percentage from zero to one hundred is shown, depending on how many words in the sequence were identified. The program code is produced with the *has_required_words* parameter set to *False* in cases when the term is not included in the user's message, preventing the chance of improper sentence matching. Then a return statement is created as *return_statement* from the function that returns the truth of each sentence, compares them, and returns the best possible answer. Figure 10 shows the program code of the two described *for* loops.

```
    # Counts how many words are present in each predefined message
    for word in user_message:
        if word in recognised_words:
            message_certainty += 1

    # Calculates the percent of recognised words in a user message
    percentage = float(message_certainty) / float(len(recognised_words))

    # Checks that the required words are in the string
    for word in required_words:
        if word not in user_message:
            has_required_words = False
            break

    # Must either have the required words, or be a single response
    if has_required_words or single_response:
        return int(percentage * 100)
    else:
        return 0
```

Figure 10: Program code of two for loops
Source: own

Additionally, the *check_all_messages* function was created, which contains the *message* parameter. An empty dictionary is created in the function, which is equal to the variable *highest_prob_list*. Then a helper function named *response* was created with parameters *bot_response*, *list_of_words*, *single_response,* and *required_words*. The role of the function is to simplify the response of the creation, and it refers to the function *check_all_messages* via the keyword *nonlocal*. When working with variables inside of already-created functions when the variable should not be a part of the inner function, the *nonlocal* keyword is utilized. Moreover, a variable can be declared to be non-local by using the nonlocal keyword. A key was created in the *bot_response* parameter that takes the probability of the message with the parameters message, *list_of_words*, *single_response*, and *required_words*. Adding items to the dictionary is made simpler by this function, which makes it simpler to construct a key and enter a value in *message_probability*. Further, each response contains a key that denotes the chatbot's reaction to a particular term it identified in the user's message. The program code in Figure 11 was written to perform prediction testing. Through the expression *best_match = max(highest_prob_list, key=highest_prob_list)* the key with the highest value, and the *print(highest_prob_list)* expression is used to print the key. Then it was created to return unknown responses as *long. unknown()* if the *highest_prob_list* in the *best_match* parameter is less than one, meaning that if all matches are less than 1, the chatbot will output unknown responses. But otherwise, it will print any best match.

```
    best_match = max(highest_prob_list, key=highest_prob_list.get)
    print(highest_prob_list)
    print(f'Best match = {best_match} | Score: {highest_prob_list[best_match]}')

    return long.unknown() if highest_prob_list[best_match] < 1 else best_match
```

Figure 11: Program code for prediction tests
Source: own

4. RESULTS AND DISCUSSION

The NLP chatbot training process was repeated until at least some of the outcomes were acceptable. The last two training modules that produced the best outcomes are described. However, compared to the second training, the first case that will be discussed made a lot more faults. The outcomes of the first training session are displayed in Figure 12. The initial training consisted of 91 stemmed words, 37 patterns, and 10 tags. The difference between the true values and the anticipated values is measured by the *loss*. A neural network is trained with the intention of minimizing this *loss*. The similarity between actual and anticipated data increases with decreasing *loss*. The *final loss* in the scenario given was 0.0007.

```
37 patterns
10 tags: ['Code Express', 'edukacija' 'men ... ]
91 unique stemmed words: [',', '0987654321' ... ]
91 10
Epoch [100/1000], Loss: 1.2487
Epoch [200/1000], Loss: 0.3267
Epoch [300/1000], Loss: 0.0386
Epoch [400/1000], Loss: 0.0084
Epoch [500/1000], Loss: 0.0042
Epoch [600/1000], Loss: 0.0038
Epoch [700/1000], Loss: 0.0013
Epoch [800/1000], Loss: 0.0011
Epoch [900/1000], Loss: 0.0015
Epoch [1000/1000], Loss: 0.0007
final loss: 0.0007
training complete. file saved to data.pth
```

Figure 12: Results of the first training session
Source: own

The target *loss* function is one of the main elements of training. This function specifies the objectives of the training as well as what is optimized. The *loss* function is an example of an objective function that is typically fairly generic and has some mathematical qualities that make it possible for popular training algorithms to successfully use it. The metrics actually achieved after training, such as response accuracy, click-through rate, and the proportion of questions the chatbot can effectively answer even when it doesn't, are different from these functions. Although it is occasionally possible to create objective functions that attempt to improve some metrics at the expense of others, these functions can only be verified once training is complete. The fundamental issue is that it is typically impossible to use the correct *loss* function. There are two causes for this: either the desired

function lacks the mathematical characteristics required to function well inside the current machine-learning ecosystem, or it is highly challenging to train. *Differentiability* is the mathematical quality that is typically sought after, however, there are many others as well. In the lack of the data required for training, a chatbot may respond to a user in an extremely inappropriate way, keep asking the same question, or, in the worst-case scenario, end communication on its own without the user's knowledge. However, the issue is the scarcity of training data. More tags in the *json* script lead to improved training results. Ten tags were created during the first training, and fifteen during the second (Figure 13). All of the tags, patterns, and responses created to create an NLP chatbot are contained in a script named *json*. JSON files contain lists of data and key-value pairs. However, it also provides for the computer to store a wide variety of data in the form of readable program code, with names serving as keys and values as the data associated with them. 43 patterns, 15 tags, 109 distinct stemmed words, and a *final loss* of 0.0004 were calculated after the *train.py* script's program code was run.

```
43 patterns
15 tags: ['Analiza podataka', 'Code Express' ... ]
109 unique stemmed words: [',', '0987654321' ... ]
109 15
Epoch [100/1000], Loss: 1.5571
Epoch [200/1000], Loss: 0.2552
Epoch [300/1000], Loss: 0.0079
Epoch [400/1000], Loss: 0.0059
Epoch [500/1000], Loss: 0.0042
Epoch [600/1000], Loss: 0.0032
Epoch [700/1000], Loss: 0.0008
Epoch [800/1000], Loss: 0.0001
Epoch [900/1000], Loss: 0.0003
Epoch [1000/1000], Loss: 0.0004
final loss: 0.0004
training complete. file saved to data.pth
```

Figure 13: Results of the second training session
Source: own

Judging by the attitude of the user, that the NLP chatbot is faster in the communication process compared to a chatbot that does not use the NLP function was observed under certain conditions and is not entirely true but taking into account the larger amount of data with which the NLP chatbot is used and the average one with which the rule-based chatbot is used, surely the NLP chatbot is faster in understanding communication. Also is concluded that a comparative analysis shows differences in the improvement of the user experience using an NLP chatbot compared to a chatbot that does not use the NLP function. NLP chatbot, available 24 hours a day, collects more data, offers greater engagement, and enables more complex communications with users compared to a rules-based chatbot. However, the NLP chatbot does not encourage frustration in the user because it can answer more complex questions, so the user does not have to wait for a human agent and waste his time. Figure 14 shows a table of comparative analysis between AI chatbot and rule-based chatbot.

There are several characteristics according to which NLP and rule-based chatbots differ, and they have been identified through a comparative analysis. Some of them are the characteristics of functionality, mechanism, ease of setup, scalability, consumption, efficiency, and personal experience. Rule-based chatbots are driven by keywords, operate on

manually created rules, are quick to set up, are more difficult to scale as chatbots become more complex, have lower implementation costs - especially for simpler chatbots, require human intervention in the form of logical adjustments, and involve a lower personal experience. Unlike rule-based chatbots, NLP chatbots are context-driven, are trained using quality data, take more time to deploy, are easier to scale because they learn incrementally from user interaction, are more expensive to implement, require more complex and higher-quality initial training, but are easier to maintain over time without human intervention, and have a higher level of personal experience.

ANN enables the improvement of the user experience by implementing NLP chatbots which is evident when sending more comprehensive responses to the user after the second training, while through comparative analysis it was established that NLP chatbots can understand behavioural patterns and possess a wider range of decision-making skills in contrast from a rules-based chatbot. That the simulation achieves satisfactory resourcefulness in understanding communication about the reliability of the algorithm was confirmed by the second training of the model in the period when the user had no objections to the answers sent by the NLP chatbot, which was shown through the percentage of the probability of matching the best answer option through the parameter *best match*.

5. CONCLUSION

A model of an artificial neural network is able to implicitly recognize complicated nonlinear interactions between dependent and independent variables while processing a huge number of datasets. The use of ANN in chatbots offers several benefits because it can also identify all potential interactions between predictive variables. The conclusion that the context-driven, ANN-based NLP chatbot can understand the user adequately without contacting a human agent confirms the idea *that ANN facilitates the enhancement of the user experience through the usage of NLP chatbots*. The percentage of the chance of matching the best answer option through the best match parameter showed the validity of the claim that *the simulation achieves adequate resourcefulness in understanding communication about the reliability of the method*. The NLP chatbot's responses were received favourably by the user, who had no complaints.

Because NLP employs more data than rule-based chatbots, it can comprehend communication more quickly than both the category of short and long questions and both the categories of lengthy questions, rule-based chatbots. Furthermore, because it is keyword-driven, it is debatable whether the term understanding can be applied to a rule-based chatbot. But if you define comprehension as the frequency with which the chatbot provided accurate responses and made the user happy, then yes, it is possible.

The comparison investigation led to the conclusion that independent learning is where AI chatbots stand out from the competition. AI chatbots deal with complete consumer requests and offer services without involving a human 24 hours a day, seven days a week. They can be programmed to comprehend various tongues and dialects and, unlike rules-based chatbots, can personalize contact with users. Although the costs of implementing an NLP chatbot may be high and the initial training may be challenging and of high quality, NLP chatbots have many advantages over other types of chatbots, including context-based guidance, training from reputable sources, easier scaling because it continuously learns from user interaction, a more personalized experience than a chatbot based on rules, and the ability to maintain them over the long term without human intervention.

6. REFERENCES

[1] Zhu Y. et al. AI is better when I'm sure: The influence of certainty of needs on consumers' acceptance of AI chatbots. Journal of Business Research [Internet]. 2022 [cited 2022 Aug 8th];150(2022):642-652. DOI: 10.1016/j.jbusres.2022.06.044

[2] Chen Q., Gong Y. Classifying and measuring the service quality of AI chatbot in frontline service. Journal of Business Research [Internet]. 2022 [cited 2022 Aug 8th];145(2022):552-568. DOI: 10.1016/j.jbusres.2022.02.088

[3] Brownlee J. Machine Learning Mastery: What Is Natural Language Processing? [Internet]. 2019 [cited 2022 Aug 8th]. Available from: https://machinelearningmastery.com/natural-language-processing/

[4] CM.com. What is an AI chatbot? [Internet]. 2022 [cited 2022 Aug 9th]. Available from: https://www.cm.com/glossary/what-is-ai-chatbot/

[5] Thorat S. A., Jadhav J. D. A Review on Implementation Issues of Rule-based Chatbot Systems. Proceedings of the International Conference on Innovative Computing & Communications (ICICC) 2020 [Internet]. 2020 [cited 2022 Aug 10th];1-6. DOI:10.2139/ssrn.3567047

[6] Liu B., Mei C. (2021). Lifelong Knowledge Learning in Rule-based Dialogue Systems. The 35th AAAI Conference on Artificial Intelligence (AAAI-21) [Internet]. 2022 [cited 2022 Aug 8th];15058-15063. Available from: https://arxiv.org/abs/2011.09811

[7] Monkey Learn. Natural Language Proce ssing (NLP): What Is It & How Does it Work? [Internet]. 2021 [cited 2022 Aug 8th]. Available from: https://monkeylearn.com/natural-language-processing/

[8] Zhou M., Duan N. Progress in Neural NLP: Modeling, Learning, and Reasoning. Engineering [Internet]. 2020 [cited 2022 Aug 9th]; 6(3):275-290. https://doi.org/10.1016/j.eng.2019.12.014

[9] Alshemali B., Kalita J. Improving the Reliability of Deep Neural Networks in NLP: A Review. Knowledge-Based Systems [Internet]. 2020 [cited 2022 Aug 9th];191(2020):105210, https://doi.org/10.1016/j.knosys.2019.105210

[10] Fornell Haugeland I. K. et al. Understanding the user experience of customer service chatbots: An experimental study of chatbot interaction design. International Journal of Human-Computer Studies [Internet]. 2022 [cited 2022 Aug 8th];161(2022):102788, https://doi.org/10.1016/j.ijhcs.2022.102788