NLP Chatbot for booking and retrieving train tickets

Andris Kokins

Computer Science, University of Nottingham, psyak19@nottingham.ac.uk

CCS CONCEPTS • Human-centered computing~Human computer interaction (HCI)~Interaction paradigms~Natural language interfaces • Computing methodologies~Artificial intelligence~Natural language processing • Information systems~Information retrieval

Additional Keywords and Phrases: NLP, AI, Chatbot, Classification

1 INTRODUCTION

The process of booking and managing tickets remains a mundane task for many travellers, often involving complex web interfaces, or inconvenient visits to physical ticket booths at the station, involving interactions with customer service. These traditional methods are old fashioned and out of date.

Digital Conductor is an intelligent chatbot that streamlines train ticket management through natural language processing (NLP). By allowing users to interact in plain English. The system simplifies common tasks like booking tickets, retrieving reservation details and question answering. The chatbot leverages modern NLP techniques including intent matching, named entity recognition and contextual understanding to provide an intuitive booking experience for the user.

2 CHATBOT ARCHITECTURE

The main functionality of the chatbot controlled by the intent matching system within *main.py*,

Figure 1 shows an overview of how the system works, although in code it is all within a *while true* loop so after each intent has been identified and executed the whole process starts again. This method allows for a modular approach to the system where each matched intent can be its own function. The system will continue to match user's every response unless it explicitly mentions *quit*, *stop* or *exit*.

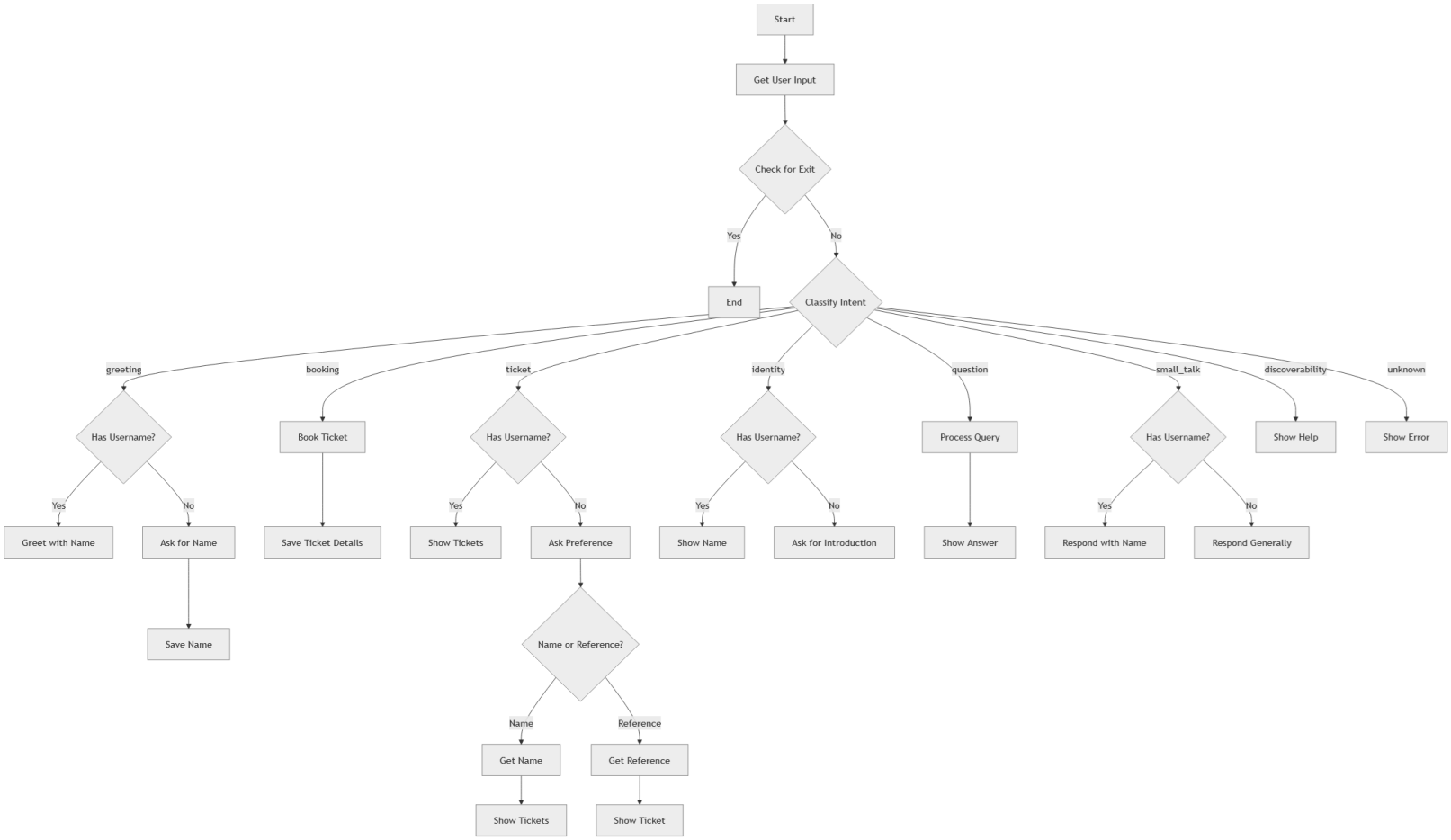

Figure 1: Flow diagram of the intent matching system. Made with: (https://mermaid.live/)

2.1 Intent Matching

Intent matching is at the core of this chatbot. The functionality of the intent matching system is to give a labelled intent based on the user's input. The current trained model in *main.py* can classify 6 user intents, the system will output one of the following: *greeting, booking, ticket, identity, small_talk, discoverability* or *unknown*. The most important intent to match for the bot is the *booking* intent, as this is where the chatbot recognises the user wants to book a ticket, so the appropriate function needs to be called. An example string that will activate the booking intent is "I'd like to book a train ticket from Nottingham to London.". The system is able to handle capitalisation and punctuation within a given input to appropriately process the intent.

2.1.1 Implementation

The technique behind the intent matching is a rudimentary search engine index. A term-document matrix and cosine similarity are employed to find the best-matching documents. Table 1 illustrates an example term-document matrix, where Doc0 and Doc1 represent indexed documents. Given a query (Doc2), the cosine similarity is computed against every document in the matrix to identify the best match.

In this example, the highest similarity score (0.56) occurs between Doc2 and Doc1, as shown by the equation below the table. Each row in the intent matching matrix corresponds to a sample phrase, such as "Good Morning," and is associated with its respective intent.[1] Queries are processed by generating a list of similarity scores, ranked in descending order, with the highest-ranked intent returned as the match.

Table 1: Example term-document matrix

Term	book	london	nottingham	ticket	train
Doc0	0.707107	0.00000	0.00000	0.707107	0.000000
Doc1	0.391484	0.66284	0. 00000	0.391484	0.504107
Doc2	0.391484	0.00000	0. 66284	0.391484	0.504107

$$\text{Cosine}(x, y) = \frac{x \cdot y}{|x||y|} \qquad (1)$$

$$\frac{0.560643}{1.000000 \times 1.000000} = 0.560643$$

The intent model is loaded in *main.py* using *load_model from utils.py* which checks if there is an existing pickle file, which is loaded otherwise a new model is built using *build_index.* Figure 2 shows how the index is built. Line 86 gets the rows from the specified column[2], e.g. given a DataFrame with question-and-answer columns – the model will be trained on the questions column. Line 90 preprocesses each row/question. Preprocessing includes, lowercasing, removing punctuation, tokenisation and lemmatisation, while stopword removal is optional. Line 93 initialises a term frequency-inverse document frequency (tf-idf) matrix with the supplied *settings* dictionary. Settings are used to specify whether to

1All intents can be found in data/intents.csv
[2] preprocess() function can be found in utis.py

use the default *word* or *char_wb analyser*. With the latter being used for spell checking city names and the former for everything else.

```
84  def build_index(data: DataFrame, column: str, stopwords: bool, settings: dict = {}):
85      # Get all rows from the specified column
86      questions = data.loc[:, column]
87
88      # Apply preprocessing to each question in the corpus
89      # do not remove stops words for intent matrix
90      questions = questions.apply(lambda q: preprocess(q, remove_stopwords=stopwords))
91
92      # Create a TF-IDF term-document matrix
93      vectorizer = TfidfVectorizer(**settings)
94      tfidf_matrix = vectorizer.fit_transform(questions)
95
96      return tfidf_matrix, vectorizer
```

Figure 2: Build Index Function (utils.py)

2.1.2 Justification

As shown in Figure 2 the code to create a search index is very short and simple, all while being very effective at matching the user's input to an intent which will be further discussed in the evaluation section 4 below. The principle of matching a query to an index is utilised in three parts within this chatbot: intent classification, question answering, and city/town name matching. For each task, an index is built using its own corpus, allowing for a modular approach and quick iteration of new features. For example, a new intent can be added by simply including a few sample phrases in the *intents.csv* corpus.

2.2 Query Similarity

Each search index [3]can be queried using the same technique using cosine similarity(1). Given a query string and the relevant search index, the cosine similarity is calculated between the query and each document in the index, returning a descending order of cosine similarity, ranging from 0 (no match) to 1 (perfect match). The top item in the list is the best matching document.

For example, given a query string: "What is my name?" and the intent index, the query similarity will return "Intent: identity". Question answering example: given a query string "Are pets allowed?", firstly the intent detects that's it's a question and proceeds to use the question answering index to find the best matching question in the database and returning an appropriate response, in this case "Small pets are allowed if kept in a suitable carrier." City name example given a query string "notingham" (misspelling of Nottingham) returns "Nottingham" with a similarity score of 0.77.

3 Search index, tf-idf matrix and search matrix refer to the same term-document matrix as shown in Table 1

2.2.1Implementation

Figure 3 shows the query similarity function which utilises *cosine_similarity* from *sklearn.metrics.pairwise.* Firstly, on line 59 the provided query string is transformed into a tf-idf vector which is then used by *cosine_similarity* to calculate the similarities between the query vector and the search matrix. Finally, the similarity scores are paired with the corresponding label and returned as a sorted DataFrame by the similarity score in descending order.

```
57  def query_similarity(query, tfidf_matrix, vectorizer):  5 usages  psyak19 *
58      # Transform the query to match the TF-IDF representation
59      query_vector = vectorizer.transform([query])
60
61      # Calculate cosine similarities
62      similarity_scores = cosine_similarity(query_vector, tfidf_matrix).flatten()
63
64      # Create results DataFrame
65      results = pd.DataFrame({
66          'Document': [f'Q{i + 1}' for i in range(tfidf_matrix.shape[0])],
67          'Cosine Similarity': similarity_scores
68      })
69      return results.sort_values(by='Cosine Similarity', ascending=False)
```

Figure 3: Query Similarity Function (utils.py)

2.2.2 Justification

Cosine similarity is a preferred approach in calculating similarity in vector space, as it not affected by the size of the vocabulary as much as Euclidean distance, therefore allowing to compare, a short query phrase against a whole sentence and still find the one that is most relevant regardless of size differences.

Similarly, the reason tf-idf term weighting is used as compared to raw frequency, is to eliminate the impact of common words that appear a lot in a corpus e.g. “the”, “is”, “and”.

2.3 Preprocessing

Preprocessing is a critical step in NLP systems to clean and standardize text data, making it more suitable for analysis. This chatbot incorporates preprocessing in both the model training phase and during user interaction. In the training phase, each string in the corpus is pre-processed as illustrated in Figure 2. Preprocessing in this system involves three primary tasks: punctuation removal, stop word removal (optional), and lemmatization.

2.3.1Implementation

The preprocessing pipeline begins by converting the input string to lowercase to ensure uniformity (line 23). Next, a custom punctuation list is constructed by excluding full stops from the default string.punctuation list and appending additional special characters, such as curly quotes and hyphens (line 26). A translation table is then generated (line 29) and applied to the string to remove all unwanted punctuation (line 30).

Following punctuation removal, the string is tokenised into an array of tokens. Stop words are optionally removed at this stage, depending on the context. For the final step, each token is tagged with a part-of-speech (POS) label, which is used by the lemmatizer to reduce words to their canonical dictionary form. The processed tokens are then returned as a cleaned string ready for use in downstream tasks.

```
21  def preprocess(raw_text: str, remove_stopwords: bool):  10 usages  psyak19 *
22      lemmatizer = WordNetLemmatizer()
23      raw_text = raw_text.lower()
24
25      # Define custom punctuation
26      custom_punctuation = string.punctuation.replace( __old: '.',  __new: '') + "’" + "-" + "‘" + "-"
27
28      # Remove punctuation
29      translator = str.maketrans('', '', custom_punctuation)
30      raw_text = raw_text.translate(translator)
31
32      # Tokenize text
33      tokens = nltk.word_tokenize(raw_text)
34
35      # Remove stopwords if specified
36      if remove_stopwords:
37          stop_words = set(stopwords.words('english'))
38          tokens = [word for word in tokens if word not in stop_words]
39
40      # POS tagging and lemmatization
41      pos_map = {'ADJ': 'a', 'ADV': 'r', 'NOUN': 'n', 'VERB': 'v'}
42      pos_tags = nltk.pos_tag(tokens, tagset='universal')
43      processed_tokens = [
44          lemmatizer.lemmatize(word, pos_map.get(tag, 'n')) for word, tag in pos_tags
45      ]
46
47      return ' '.join(processed_tokens)
```

Figure 4: Preprocessing Function (utils.py)

2.3.2 Justification

Preprocessing ensures the language is disambiguated, irrelevant details are removed, and the vocabulary is reduced to its most meaningful elements. This is particularly important for scalability, as larger corpora lead to exponentially growing search indexes and increased processing times.

However, not all preprocessing steps are uniformly applied across all use cases. For instance, the intent model retains stop words because some intents rely heavily on them. An example is the identity intent—such as the phrase “How are you?”—which would become meaningless if all stop words were removed, resulting in an empty string after processing.

2.4 Ticket Booking

Ticket booking is the main feature of the chatbot where the user interacts with the chatbot to book their train journey, within the set parameters. The booking process supports both experienced users, who might have used the bot previously as well as novices. As shown in Figure 5, a perfect input can smoothly flow through the process and book a ticket. The

first check is *attemps < 1 && actual_cities < 2*, is a required check due to the implementation of the system. If the user's original input is "I want to book a ticket", the intent matcher cannot distinguish between a 'request' and full booking input i.e. "I want to travel from Nottingham to London tomorrow at 10:35". Therefore, to accommodate for this, if it is the user's first attempt and they didn't specify two cities, it will simply re-prompt with instructions. However, on their second attempt, *book_ticket_stepped()* is called which will walk the user step by step on how to book a ticket until they confirm it is correct.

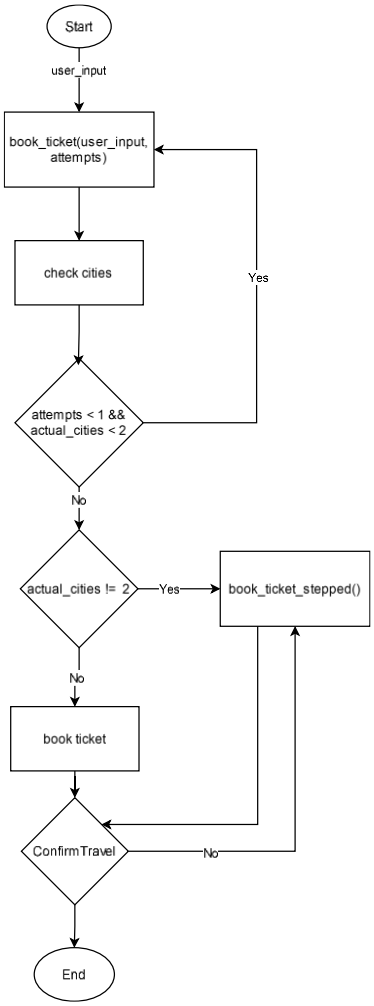

Figure 5: Flow diagram of ticket booking process (ticket_booking.py)

2.4.1 Implementation

The process begins in *book_ticket from ticket_booking.py*, there are two arrays – *potential_cities[]* and *actual_cities[]*. The former array is used to store any tokens that are one of the following: *('NN', 'NNS', 'NNP', 'NNPS', 'VB', 'JJ')*, this is because the nltk's named entity recognition (NER) can only recognise locations that are properly capitalised and spelled correctly. Once all the potential cities are found, they are processed by the *confidence()* function. Which utilises a search

index trained on named cities in the UK[4]. The search index returns a similarity score, which is used as part of three-tiered confidence level. High at >=0.7, medium at >=0.5 and low confidence at <0.5. At the first tier, the top match is returned but at the next level, the user could have misspelled a city name e.g. “Londan” instead of “London”, this returns a confidence level of 0.56 and asks the asks the user if they meant to say whatever the top matched city is, if the answer is yes, it will return the correct spelling, otherwise the user is asked to write the name of the city.

The final step is to extract the time, *get_time()* function utilises a dateparser library[5] to extract a date from within a string, with support for multiple formats e.g. “Tomorrow at 10:30” or “15/01/2025 at 10pm”.

Once the departure, destination and time is gathered from the user, the chatbot asks the user to confirm the booking, if the answer is yes, it will save the ticket in the database otherwise it will call *book_ticket_stepped().*

3 CONVERSATIONAL DESIGN

Right at the start of the conversation the chatbot is introduced in a friendly manner with a welcoming message – “Digital Conductor: Hello, how can I help you today?”. First key thing to mention is instead of a generic bot name i.e. “Chatbot” or “AI Chat, the bot is appropriately named as a “Digital Conductor”, hinting that the bot’s functionality is something to do with trains. Throughout the conversation the system from first person as “Digital Conductor” to keep it as human-like and conversational.

3.1 Personalisation

The first part of personalisation is when the user first greets the chatbot. The chatbot will ask for the user’s name and save it in a database, so the next time the same user comes back and says their name instead of saying “Digital Conductor: Hello {name}, I will make sure to remember you next time.” The bot will welcome them back with “Digital Conductor: Welcome back {name}! How can I help you today?”

In addition to this, when the user asks for their train bookings, the chatbot will first use the user’s name if it is not available it will ask for the user to introduce themselves or alternatively provide a reference number.

3.2 Confirmations

As mentioned in 2.4.1 above, the *confidence()* function in *ticket_booking.py* uses three tiered confidence, with the first tier being an implicit confirmation (without printing a message to avoid over-confirmation), the next tier asks the user for an explicit confirmation for the corrected spelling and the third tier ignores the input with the assumption the user didn’t mean to say that or there was no close match in the corpus.

3.3 Error Handling

Most errors are handled gracefully. For example, in *main.py,* unknown intents simply print “Digital Conductor: Sorry, I didn't understand that. Please try again.”, without blaming the user for not knowing the available intents and instead encourage them to their prompt again.

Similarly, if the intent is matched to a question but there is no matching response in the database, the chatbot will respond with “Digital Conductor: Sorry, I don't have the knowledge to answer that, please try again...”, which lets user know the bot doesn’t know and once again offers them to try again.

4 Full list of cities can be found in data/uk_cities.csv
5 More information about the dateparser library - https://dateparser.readthedocs.io/en/latest/

Meanwhile, in *ticket_booking.py* the main error handling is the *book_ticket_stepped()* function that walks the user step by step through the booking process with examples – “Digital Conductor: Enter your departure city, note this needs to be a city within the UK, e.g. London”. As well giving an example prompt – “Digital Conductor: For example, 'I want to travel from London to Manchester tomorrow at 10:00'”

4 EVAULATION

4.1 Response Classifier

The performance of the response classifier was evaluated using a dataset split into 75% training and 25% testing data, with a random state of 42 for reproducibility. The evaluation metrics include accuracy, F1-score, and a confusion matrix to analyse the classifier's performance across different intents.

The classifier achieved excellent accuracy (92.73%) and an F1-score (93.10%), suggesting reliable performance in distinguishing between intents. The confusion matrix reveals that the classifier successfully identified all positive cases without false positives, though a small number of false negatives were observed. These could be attributed to ambiguous queries or overlapping features between classes.

Metric	Score
Accuracy	92.73%
F1-Score	93.10%

Table 2: Response Classifier Results

Actual / Predicted	Negative	Positive
Negative	24	4
Positive	0	27

Table 3: Response Classifier Confusion Matrix

4.2 Search index

The intent indexes were evaluated using Normalized Discounted Cumulative Gain (NDCG), [6]a widely used metric in information retrieval systems that measures the relevance of the returned results based on the ranking position.

[6] All scores were calculated using from sklearn.metrics import ndcg_score

4.2.1 Intent Index

Table 4 highlights the NDCG scores for a variety of prompts, with the index achieving an average score of 0.91. This demonstrates the system's ability to provide relevant results for most queries.

Prompt	Normalised Discounted Cumulative Gain
"Hello, how are you?"	0.99
"What can you do?"	0.77
"I'd like to book a train ticket"	1.0
"Show me my booked tickets"	1.0
"Are dogs allowed on the train?"	0.77

Table 4: Normalised Discounted Cumulative Gain for Intent Index

4.2.2 City Name Index

Table 5 shows NDCG scores for a variety of prompts, with a majority being misspellings of big cities. 4/5 searches found the correct spelling of the city names, except “notinham” which matched to Rainham closer than Nottingham which lowered the score significantly.

Prompt	Normalised Discounted Cumulative Gain
"London"	1.0
"Manchester"	1.0
"notinham"	0.49
"londan"	1.0
"burmingam"	1.0

Table 5: City Name Index NDCG

4.3 Performance

Performance was measured to evaluate the system’s ability to process user input from intent matching to saving the ticket in the database. This is a critical aspect of ensuring a smooth user experience for real-time chatbot systems, where response times directly influence user satisfaction.

The experiment was conducted on a machine with an AMD 57000X3D CPU, 32GB of RAM, and Python 3.12. Measurements were taken using *time.time()* to record timestamps after the first user input and once the ticket was saved to the database. Table 6 presents the results of four test runs, showing an average processing time of 2.285 seconds.

Run	Time (seconds)
1	2.88
2	2.12
3	2.08
4	2.06
	Average: 2.285

Table 6: Time taken to process input from intent matching to saving ticket

5 DISCUSSION

5.1 Evaluation Results

Looking back at section 4.2 above, the results of the tests seem good with an average score of 0.92 for the intent matching. These results are deceptive, since in reality, whenever a wrong intent was matched, there is no error correction or the ability for the user to choose the intent, the system simply picks the top one, which can be wrong, resulting in a very poor user experience. The fundamental problem is either in using a search index to solve the wrong problem or a search engine evaluation method for something that is not used as a search engine, since the user never sees what the ranked list is.

A possible solution could be to go through the top N possible intents and let the user choose one, but this would not be a very smart chatbot. Alternatively, the corpus can be refined more, removing redundant samples and replacing them with higher quality ones.

Context switching is something that's not part of this chatbot which is usually quite common to have, outside of the names and tickets database once the ticket booking process begins the user is not able to ask any questions, ask for help or leave without having to finish the task before proceeding. Even then, the user is forced to use the step by step booking process without receiving an explicit confirmation that the user wants or needs the additional help.

In addition to this, the chatbot takes the first recognised city as the departure without the ability to change or using context e.g. saying "To Nottingham from London" will classify Nottingham as the departure.

5.2 Reflection on the project

This project provides a solid foundation for an NLP chatbot capable of booking and retrieving train tickets. The system successfully implements fundamental features such as intent recognition, response generation, and a ticket database. However, several areas for improvement remain: The absence of context switching and flexibility during interactions limits the chatbot's usability in real-world scenarios. Intent recognition accuracy, while high in testing, requires further refinement to ensure reliability in diverse use cases. Introducing user feedback mechanisms and contextual understanding could elevate the chatbot's capabilities to a more sophisticated level.

Despite these limitations, the project demonstrates the potential of combining NLP techniques with structured data processing for practical applications. With further development, this chatbot could evolve into a robust and user-friendly conversational assistant.