scientific reports

Check for updates

OPEN

Translating synthetic natural language to database queries with a polyglot deep learning framework

Adrián Bazaga, Nupur Gunwant & Gos Micklem✉

The number of databases as well as their size and complexity is increasing. This creates a barrier to use especially for non-experts, who have to come to grips with the nature of the data, the way it has been represented in the database, and the specific query languages or user interfaces by which data are accessed. These difficulties worsen in research settings, where it is common to work with many different databases. One approach to improving this situation is to allow users to pose their queries in natural language. In this work we describe a machine learning framework, Polyglotter, that in a general way supports the mapping of natural language searches to database queries. Importantly, it does not require the creation of manually annotated data for training and therefore can be applied easily to multiple domains. The framework is polyglot in the sense that it supports multiple different database engines that are accessed with a variety of query languages, including SQL and Cypher. Furthermore Polyglotter supports multi-class queries. Good performance is achieved on both toy and real databases, as well as a human-annotated WikiSQL query set. Thus Polyglotter may help database maintainers make their resources more accessible.

Natural language querying (NLQ) is a known problem in information retrieval[1,2]. It allows questions to be formed without knowledge of database-specific logical languages such as SQL or Cypher. In principle this can ease data access for non-expert users. To this end, several approaches to build NLQ systems have been proposed. Recent surveys[1,3,4], segmented them into five approaches: keyword-based, pattern-based, syntax-based, grammar-based, and, more recently, connectionist-based.

In keyword-based systems, the approach has two stages, a first stage where keywords present in an input query are extracted, and a second stage where keywords are matched against metadata available in the underlying database. For instance, Blunschi et al.[5] described Search over DAta Warehouse (SODA), which generates SQL queries from natural language (NL) queries over a business-related database. The system processes an input NL query through a series of steps. First, the keywords in the query are matched against all the possible entries in the database metadata. Second, by means of a heuristic, each result is scored and the process continues with the top N results. A third step identifies the tables used by each result and their relationships before being passed to the fourth step, which is responsible for finding, from the original query, the needed filters over the tables and columns. Last, the gathered information from the previous steps is combined to generate a SQL query that takes into account possible join patterns by looking at foreign keys and inheritance patterns in the schema. A drawback of this approach is it requires hand-crafting the patterns that are used to translate from a keyword-based input to a SQL query for the specific modelling of the target database, which becomes a bottleneck when trying to use it across a variety of databases.

In pattern-based systems, the capabilities of keyword-based approaches are extended by including NL patterns for processing queries, alleviating issues such as aggregation operations. For instance, Shah et al.[6] described NLKBIDB, a NL to SQL query interface that uses NL patterns to fix syntactically incorrect queries, and a keyword-based approach to obtain the corresponding facts from the schema before carrying out the conversion. The system uses lexical analysis to tokenize the NL query and syntax analysis to parse the lexicons. If the input query is syntactically valid, then the lexicons are analyzed by a semantic analyzer using a domain ontology before generating a SQL query. Invalid queries are converted into SQL by applying a set of rules. An issue that arises from this approach is the need for a knowledge expert to refine the rules used to convert syntactically invalid queries into SQL sentences, as well as the need to use hand-crafted natural language rules for generating the

Department of Genetics, University of Cambridge, Cambridge, UK. ✉email: gm263@cam.ac.uk

SQL queries for syntactically valid queries. This makes it difficult to adapt for new use cases. In[21] ATHENA++ is presented to cope with nested SQL queries. In this approach linguistic patterns from NL queries are combined with domain reasoning using ontologies to enable nested query detection and generation. In addition, a new benchmark dataset (FIBEN) is introduced, containing 237 distinct complex SQL queries on a database with 152 tables. This approach tries to translate a NL query to OQL using the domain ontology, and then the OQL query is translated to SQL by using the mappings between the ontology and the database. It appears that one potential limitation of such approach is the requirement to have both a domain ontology and a defined mapping of that ontology to the database.

Syntax-based and grammar-based systems[7] share similar methodologies. In both approaches, the NL query is parsed using linguistic rules to produce a syntax tree. Then, for syntax-based systems, the concepts in the tree are mapped to a query in the target query language (e.g. SQL). This has a variety of issues. First, a given query may have different parse trees, which after mapping may produce different queries. Another issue is deciding which concepts from the query to map and which ones not. To alleviate this, grammar-based systems exploit domain knowledge captured by the grammar with the aim of reducing ambiguity when mapping queries. However, this method requires knowledge of the domain to build an effective parsing grammar, making it hard to adapt to new domains.

In contrast to approaches treating items in a language as symbols (symbolic approaches), relying on theoretical foundations in linguistics, connectionist approaches[8], also known as computational intelligence-based approaches, try to learn statistical patterns in the data and/or distributed representations of it, allowing for a richer linguistic variability. These methods can be divided into traditional machine learning and more recent deep learning techniques. The latter have become widely used in natural language processing (NLP) tasks, achieving state-of-the-art results[9,10]. The main difference between these methods and traditional machine learning is that their objective is to learn a distributed representation (in the form of real-valued vectors) of the data, without the need for the feature engineering stage that earlier approaches required.

Deep learning methodologies cover key NLP applications[11,12], including part-of-speech tagging[13], named entity recognition[14] and machine translation[15]. In such systems the problem usually has been posed as an end-to-end neural semantic parsing problem[16]. A specific case is the supervised sequence-to-sequence machine translation[17] problem, in which the algorithm is trained to construct a representation encoding the input sentences in one domain (e.g. natural language) and decode them into output sentences in a different domain (e.g. a database specific language, such as SQL).

For instance, the Seq2SQL system[18] is a seq2seq-based deep neural network for translating NL questions to their corresponding SQL queries, using a reinforcement learning policy to learn the conditions of the SQL query. The output space of the softmax function that predicts each token in the SQL query is limited by leveraging a priori knowledge of the database schema, thus reducing the number of invalid queries. The model is evaluated by running the predicted SQL query on the database and comparing the result obtained against the ground truth. The authors generated a dataset to train and evaluate their model, named WikiSQL, comprising 80,654 manually annotated example questions, and their equivalent SQL queries and results across 24,241 tables from Wikipedia. Despite achieving a highly accurate model, the system is intended to work on a single table at a time, with the system already knowing on which table it has to run the query as well as the schema describing the table. With a similar approach, Xiaojun et al.[19] describes SQLNet, which employs a sketch-based scheme containing a dependency graph, and a column attention mechanism to synthesize the query from the sketch. In a similar fashion to Seq2SQL, the SQLNet handles translation of NL queries to SQL over a single database table, taking a pair of the input table schema and the NL query. Thus this approach only provided single table queries based on SQL.

Yin et al.[20] described Neural Enquirer, a neural network architecture for predicting a ranked list of possible answers to a NL question, Q, over a SQL database table, T. The system embeds both Q and T into a distributed representation before passing it through a pipeline of executors that derive a probability distribution over the table entries as a ranked list of possible answers to the original NL query. In order to evaluate the system, the authors present an approach to generate synthetic question-table-answer triples on which the models are trained. In this generated dataset, each query involves a single table randomly sampled from an Olympic Games database. The natural language for the query is generated using NL templates to generate 4 different types of queries. The system reports an overall accuracy of 99.9% on a synthetic dataset containing 100,000 query-table-answer triples. Like Seq2SQL and SQLNet, this system does not consider searching over multiple tables with a single query, and similarly is engineered to work only with SQL databases.

In an effort to alleviate the requirement to have a manually annotated dataset to train a NL to database (DB) interface, Weir et al.[22] developed DBPal, a deep neural network for learning natural language interfaces to SQL databases that synthesizes a large collection of pairs of NL queries and their corresponding SQL statements from a given database schema to train the model. The authors employ data augmentation by means of paraphrasing to make the model more robust to linguistic variation, and devise a manually crafted dataset of 300 queries to assess the robustness of the model, where the same NL query is written with multiple linguistic variants.

In this paper, we propose a polyglot NL-to-query system, Polyglotter, to address some of the limitations in the approaches described above. Unlike most other systems, we support questions over multiple tables in a single query. We create synthetic data for training from the data models of each of the supported database engines. These synthetic datasets are then used to train a sequence-to-sequence-based algorithm. In order to allow Polyglotter to handle queries for a variety of database engines, we introduce an abstract representation for user queries in the form of a graph. This contains the classes, attributes and constraints, and acts as an intermediate representation that facilitates support for multiple database languages, e.g. SQL or Cypher. This makes it easier to re-use our system in new contexts. Here, classes represent the types of entities that the database stores, with attributes being the properties of these entities. Constraints within a query are used to select the subset of objects in the database that are of interest for that query.

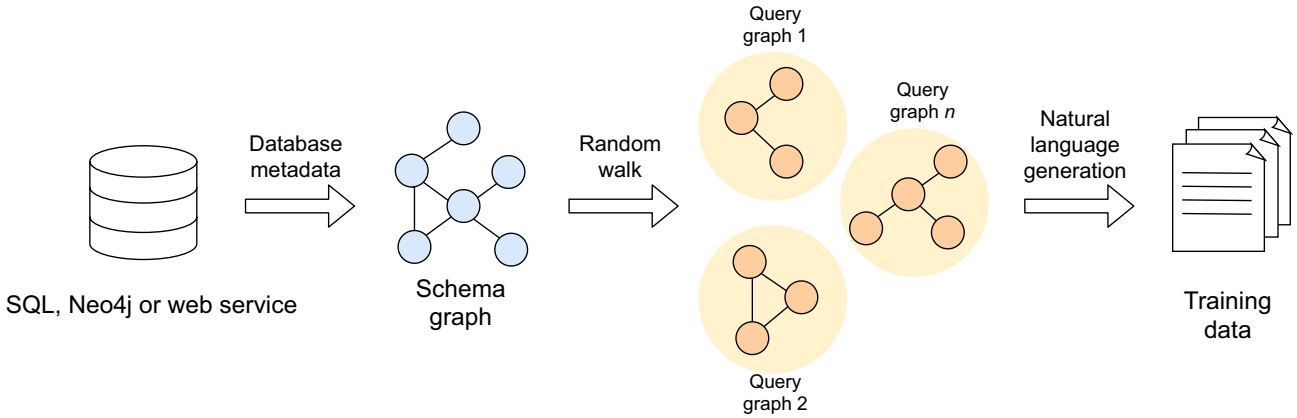

Figure 1. Overview of the random query generation method used by Polyglotter to obtain training datasets for the sequence-to-sequence models. Classes and attributes are added through a random walk, and constraint operators and values are also selected randomly.

This graph can be decomposed into pairs (a class with any one of its attributes) and triples (a class with any one of its attributes and a constraint on that attribute) that forms a convenient target for machine learning. Note that a constraint is made up of an operator that makes a comparison and a value. The trained system is able, given a user query in natural language, to provide multiple suggestions in the specified database-specific logical language. We assess the performance of our system in three widely used database back-ends: MySQL, Neo4j and web service-based data storage, loaded with real and synthetic data from multiple domains: biology, business and product line management. With the above developments we hope to provide a generally useful system (1) that provides a simple and data-less way to build natural language interfaces supporting multi-table queries, (2) which supports multiple database engines and (3) that can be extended to support other database systems.

Synthesizing training data using random query generation

To generate the training dataset for a specific database backend, we derive a schema graph abstraction that represents the entities existing in the database, as well as the relations between them. Then we perform random walks on it, as described below, to emit multiple subgraphs describing different queries over the database. These are further transformed to synthetic natural language to build the training set. Hence, our training data is not made of human-produced queries but computationally-based queries that resemble human questions. Figure 1 provides an overview of the steps followed by the query generation procedure.

We start by gathering the database metadata in order to collect four elements: the different entities existing in the database, the number of instances within them, their relationships and the attributes that describe each of them. To this extent, our implementation already provides an extensible set of metadata gatherers for several popular database engines: MySQL, Neo4j and web services/APIs. In the case of web services-based databases, we specifically provide an example implementation for HumanMine[23], an open-source biological data warehouse for *Homo sapiens* data.

In the schema graph, each node represents a class (e.g. a table in the case of MySQL), and nodes are connected if there is a corresponding class-class relationship (e.g. foreign keys in the case of MySQL). Also, each of the nodes is enriched with the attributes that it contains (e.g. the column names in the case of a node corresponding to a MySQL table).

Once the database schema graph has been obtained, the query generation process performs random walks over it each of which selects a contiguous subset of the graph, which can be represented as a query. The random walk procedure is parametric, allowing the system maintainer to tune the complexity of the generated queries as described below. The number of query graphs to produce is controlled by the parameter *N*.

The starting node for a random walk is chosen randomly following a uniform distribution over classes. Queries can be enlarged by selecting which attributes of the class to report and also, independently, by applying constraints to the values that these attributes may take, as well as by adding further classes. The attribute and constraint choice probabilities (*attributeChoiceProbability* and *constraintChoiceProbability* within the code) control the likelihood of adding an attribute or constraint to the query for the current class. The constraint logic is picked randomly by rolling a dice over a distribution of possible logical operators. Since we are using a model that uses attention to align and copy the values given in a user query, the values of the synthetic data set constraints are randomly generated and typing is not checked. Similarly, the graph traversal probability parameter determines the chance of adding another class to the query. This is done by considering those directly connected nodes that have not already been added to the query. The *cap_classes* parameter sets a limit on the number of classes allowed in each query.

In our experiments we used the following values for the parameters: a uniformly random starting node selection, multi-class queries, 0.25 and 0.05 for the attribute and constraint choice probabilities respectively and 0.5 for the graph traversal probability. We limited the maximum number of classes in each query by taking into account the size of each database schema. For Neo4j, MySQL and HumanMine with schemas comprising a total

of 5, 8 and 170 classes respectively, we capped the number of classes possible in a query to a maximum of 3, 4, or 5 respectively. We found these values provided a reasonable diversity of queries that looked realistic to a human.

Finally, once N query graphs have been generated, each of them is translated into an equivalent query in English, and also into the target pairs and triples that the model will need to predict. Attributes from classes in the query, are modelled as *Class* − *Attribute* pairs. Similarly, constraints over an attribute in a class are modelled as *Class* − *Attribute* − *Constraint* triples. Note that the Constraint is itself a pair composed of Constraint logic and Constraint Value. The pairs and triples for a given target query are separated by the special token "; ".

We do not programmatically validate the automatically generated natural language queries. However, we note two points regarding this issue: first, as we generate the queries by walking over the schema of the database, including class-class connectivity, the generated queries are always valid with respect to the database model. Second, the natural language is generated using base templates written by humans, and while the provided set appear to be sufficient, there is nothing to stop users adding further such templates. Through the use of such skeletons, the generated queries make sense to human users.

The translation to synthetic natural language from a query graph is carried out by going over each of the graph's elements and expressing them in English. Native speakers would naturally choose between a variety of ways of doing this: for instance they might choose different starting points, different orders and prioritise different elements, for instance dealing with all the classes before describing attributes or constraints, and make use of synonyms when describing constraints.

Therefore we have tried to include these and other approaches in order to increase the diversity of the generated language: the system randomly chooses one of six different styles of sentence, where each variation alters the order in which the different types of element (class, attribute, constraint) are dealt with. For example, one variation iterates over the classes, and with each describes the corresponding attributes and constraints; another begins with the attribute(s), followed by all the class(es) and finally all the constraint(s); a third one specifies first the constraints, then the attributes and then the classes. Further variability is generated by starting the conversion at different locations in the graph, and by picking synonyms of verbs and constraint operators randomly from a pre-defined list. As an example, from a query graph built out of one node (hence representing a class/table - e.g. "*genes*"), annotated with the attribute "*name*", a constraint for attribute "*length*" with constraint logic ">" and constraint value "9800", one form of synthetic English that could be generated is: *"give name of genes such that length > 9800"*.

We exploit the FastText word embedding[30] during training of the model in order to capture more natural language variability. This word embedding captures in a high-dimensional space the semantics of 1 million words extracted from a 2017 version of Wikipedia. This has two benefits: (1) during training, the limited vocabulary used when synthesising language is placed in the context of more general but similar English; (2) when used in production, this word embedding mediates between the diversity present in user queries and the Transformer model.

In the above ways we try to capture some of the diversity of expression that an English speaker would use (see Supplementary Table 1). Finally, the generated sentences are stored paired with their attributes, classes and constraints using a valid format for OpenNMT[24,25,27].

In contrast, in order to generate a database query from a query graph, we use a recursive process that fills a template using the attributes, classes and constraints from a given query graph. We provide a query template for each of the proposed underlying databases (SQL, Neo4j and HumanMine) that are supported. As an example for the Cypher database language, from a query graph built out of one node "*Proteins*", annotated with the attribute "*identifier*", a constraint for attribute "*symbol*" with constraint logic "=" and constraint value "*IGF*1", the resulting Cypher after replacing the attributes, classes and constraints in the base query template would be *MATCH (p:Proteins) WHERE p.symbol = 'IGF1' RETURN p.identifier*. In the case of queries comprising more than one class (or table), we make use of the generated minimum spanning tree of the query classes on the database schema: this allows the recursive process to identify the joining attributes between the pairs of classes and include each as a constraint in the generated query.

Development of a neural sequence-to-sequence model with an attention mechanism

We constructed a deep learning-based sequence-to-sequence model to predict the set of classes, attributes and constraints from a given user query, in the form of the pairs and triples as defined above. In this work we utilize the recently proposed Transformer architecture[28] to train a model with such capabilities. Apart from overall higher performance in this kind of task, one of the main advantages of this model, in contrast to the older Recursive Neural Network (RNN)-based models, is the highly parallelizable training phase, making training on large corpora more efficient.

The overall objective of a sequence-to-sequence model is to learn a latent representation (embedding) of an input language, in order to be capable of rebuilding the equivalent sentence in a different language or vocabulary from the learned embedding. The part of the model responsible for encoding a sentence from the input language is called the encoder, whereas the part that transforms the embedding to a sentence in the target language is called the decoder. The model uses the embedding to represent the semantic meaning of each word, as well as learning the most relevant elements in the input to be looked at as each word is examined in turn by using a self-attention mechanism.

The Transformer model improves upon the self-attention mechanism by introducing multi-headed attention, so learning a set of attention matrices instead of one. This provides two main benefits: first, it extends the ability of the model to focus on different positions, by helping to prevent the attention being dominated by the word translated at each timestep. Second, each of the attention sets is initialized with different random values,

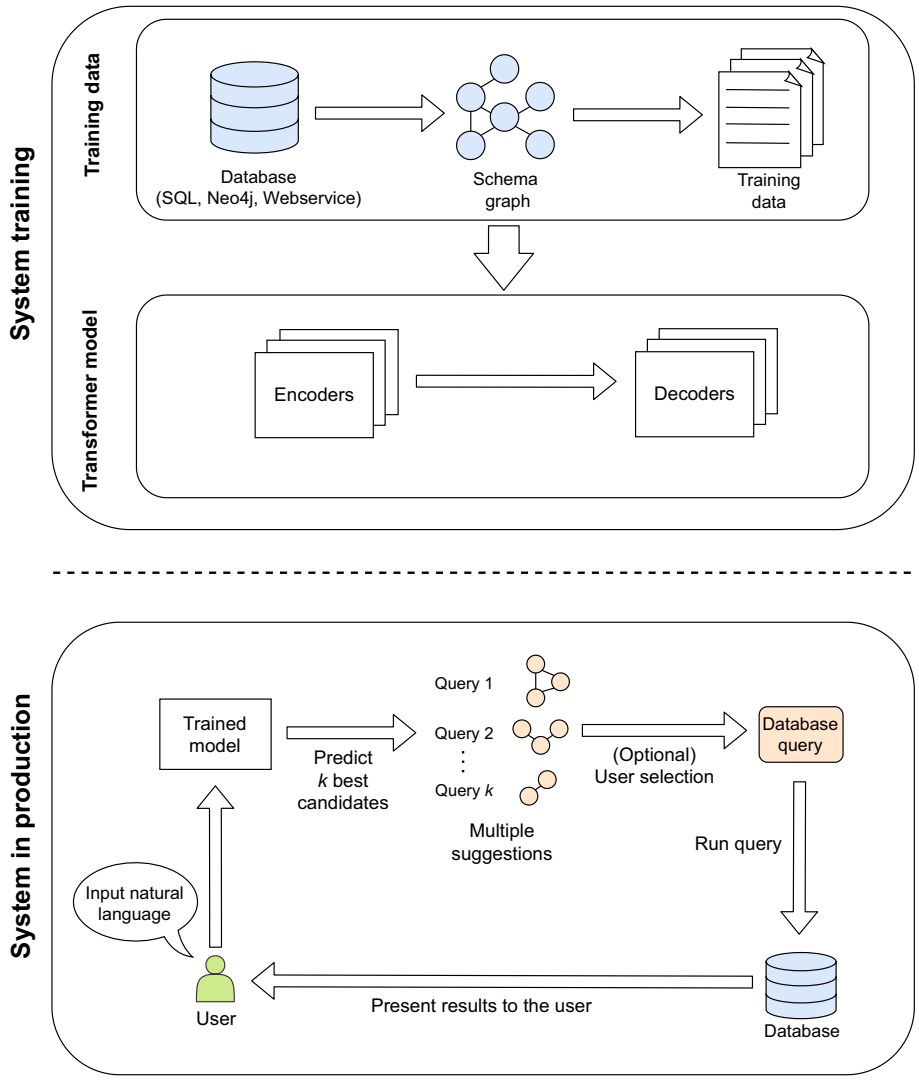

Figure 2. Graphical summary of the overall workflow for Polyglotter.

providing different representation subspaces when used during the projection of the input embeddings. For the model architecture, we found the original Transformer[28] parameters to be the most suitable for our problem through an ablation study (Supplementary Figure 1): we use a Transformer with 6 layers in each of the encoder and decoder, with a hidden size of 2048 for the inner position-wise fully connected feed-forward network. The dimensionality of the input and output is 512. We employ 8 parallel attention (heads) layers. We initialize the model weights using Glorot. The batch size during training is set to 4096 with a gradient accumulation based on four batches. We choose Adam[26] as the optimizer and set the learning rate to 2. The weight decay method is set to Noam.

Figure 2 provides a graphical summary of the overall workflow for Polyglotter, using the Transformer as the translation model. First, training data was generated once from the target database using the random query generation procedure described in "Synthesizing training data using random query generation". Then, the model was trained with early stopping with the same procedure as for the original Transformer implementation, including the model's hyperparameters.

The random walk procedure creates a bias against longer queries, yet these are of interest. Therefore we normalised the training set by the number of classes, setting a limit of five classes for HumanMine as this is greater than queries typically run on the database. Thus, for instance, the data set of size 100,000 for HumanMine was made up of five sets of 20,000 queries, with each set having a distinct number of classes from one to five. The class size cap was set according to the complexity of the database schema, at five for HumanMine (170 classes), four for MySQL (8 classes) and three for Neo4j (5 classes).

The datasets generated above were then divided into training, validation and test sets, comprising 60%, 20% and 20% of the total observations, respectively. Duplications of identical natural language queries were removed from the testing set. We note that there are a variety of ways that a single query graph can be expressed as natural language and it is desirable that the system is capable of predicting correctly in all those cases. Therefore, we also evaluated the fraction of target query graphs in the testing set that were also in the training set for HumanMine

(Supplementary Figure 2) by considering the classes, attributes and constraints. Given that there are 170 classes, it is unsurprising that there is 21% target graph overlap when the dataset size is 1,000,000 and queries containing only one class ($nc = 1$) are considered. This drops to 8.6% for $nc = 2$ and there is minimal overlap for $nc = 3, 4, 5$ (2%, 0.4%, 0.07% respectively). We note that for the simpler databases (MySQL: 8 classes; Neo4j: 5 classes) high test performance is achieved with datasets of only 5000 samples (see below) and at this dataset size, unsurprisingly we see overlaps in the range of 20–30% for $nc = 1, 2$ and 6–7% for $nc = 3$.

The queries with the largest class count (5, 4, 3 for HumanMine, MySQL and Neo4j respectively) were kept in the testing set and never seen by the model during training, as a way of assessing the capability of the model to generalize to longer queries.

Once trained, and on receiving an input natural language query, the model is used to predict a set of k query graphs as follows. The model directly predicts class-attribute pairs and class-attribute-constraint triples. In order to ensure the connectivity of these predictions, a minimum spanning tree of the predicted classes is obtained based on the database schema. Together with the predicted attributes and constraints this yields a query graph. Beam search was used to generate the top k predictions for a given query[29]. In our evaluation described in the following sections, we employ three values for k: $k = 1, 3, 5$, corresponding to the top 1, top 3 and top 5 query graph suggestions, respectively.

If given a choice in production, an end-user can choose which of these query graph predictions should be translated into a database-specific query that can be executed. We provide example implementations for transforming query graphs into equivalent database queries for the MySQL and Neo4j database engines, and also for HumanMine web services. It is also possible to translate the predicted query graph back into synthetic natural language, as described in "Synthesizing training data using random query generation". This can be used to help users select which of a number of alternative predictions is closest to what they want, and also to provide feedback to users who are trying to compose a query with any other query building tools.

Evaluation of queries predicted from natural language

We compared how the performance of Polyglotter varied with training dataset size in order to see whether there is evidence of saturation i.e. a point where increasing the number of instances available for training does not provide meaningful improvements in model accuracy. In order to assess the generalization performance we report the global accuracy by comparing the predicted query graphs against the known true query graph. We define the global accuracy over the testing set as the fraction of total testing set for which the predicted query graphs match exactly the ground truth for all the pairs and triples that comprise the query (i.e. predicting accurately all the attributes, classes and constraints).

Figure 3 shows the test set performance of Polyglotter across different generated dataset sizes ($N = 1000$ to 1,000,000) from the Neo4j, MySQL and HumanMine databases. As expected, we find that model accuracy increases with the size of the training corpus and that less complex databases require smaller training datasets to achieve a given level of performance. The impact of considering multiple predictions was examined by allowing any of the top 1, 3 or 5 predictions from each model to match the corresponding test, with overall performance increasing with the number of predictions considered.

In the case of HumanMine, substantial increases in accuracy are achieved until $N = 25{,}000$ (86.7% test set accuracy with $k = 1$), and more marginal but useful benefits are observed with larger datasets. For instance, increasing from $N = 25{,}000$ to $N = 1,000,000$ (90.3%) provides a gain of just 3.6%, whereas from $N = 10{,}000$ (71.6%) and $N = 25{,}000$ provides a gain of 15% in performance. For the other two databases we see similar trends, but in this case most of the performance has been achieved by $N = 5000$ (MySQL 88.7%; Neo4j 95.4%). For these two databases, we note that from $N = 5000$ the overlap of the testing and training dataset target query graphs is of slightly higher degree compared to that of HumanMine at $N = 1{,}000{,}000$. This is not surprising given that, for MySQL and Neo4j, the number of classes in the database schema is much lower (8 and 5 classes, respectively) than for HumanMine (170 classes). This means that the complexity of the set of queries is lower. For completeness we show the accuracies achieved up to $N = 1{,}000{,}000$ for these databases (Fig. 3).

As expected, increasing the number of predictions to three or five gives progressively better performance, with most of the improvement coming from considering three choices rather than just one. For HumanMine ($N = 1{,}000{,}000$), relatively high performance is achieved with accuracies of 90.3%, 98.3% and 98.9% for the top 1, top 3 and top 5 choices, respectively. Correspondingly, at $N = 5000$, the MySQL model reached 88.7%, 93.7% and 94.3%, and the Neo4j model achieved 95.4%, 97.3% and 97.9%. Across the three databases we note that the performance is not inversely related to the difficulty of the learning problem, represented by the complexity of the corresponding data model i.e. by the cardinality of the set of attributes and classes used in the database model. Thus the performance on the most complex database (HumanMine) is intermediate to the two simpler ones with one choice, but better than either of them with 3 or 5 choices.

To assess the complexity of the query graph prediction problem, we evaluated separately how precise the models are at guessing each of the different components of the query graph i.e. the classes, attributes and constraints. Figure 3 (right column) shows for each database the test set performance across the different generated datasets for each of these three components. From these experiments we note that predicting the constraints being expressed in the synthetic natural language queries was harder than predicting the classes and attributes. i.e. HumanMine 88.5% ($N = 1{,}000{,}000$); MySQL 88.7% and Neo4j 95.4% for $N = 5000$ (Fig. 3, second column, rows 1–3 respectively). Two possible explanations include, first, since the vocabulary that has to be predicted in the case of the constraints is infinite, it is more difficult for the model to achieve precise predictions. Second, in order to predict a constraint correctly, the class, attribute, constraint logic and constraint value must all be correct and this is harder than simple predicting class-attribute pairs.

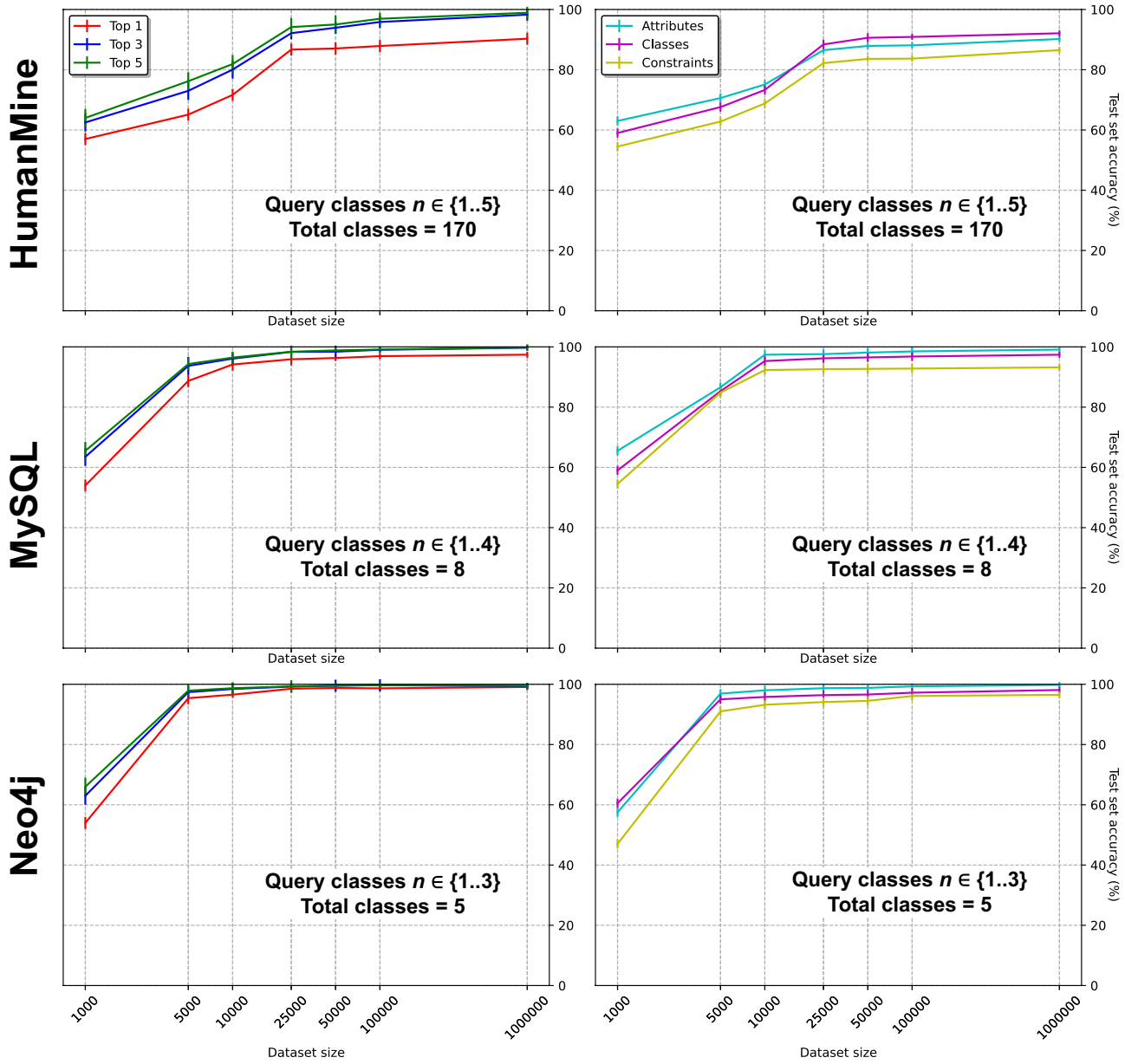

Figure 3. Test set performance as a function of training dataset size for HumanMine, MySQL and Neo4j (rows 1, 2 and 3 respectively). Left: overall test set performance allowing any of the top 1, 3 or 5 predictions from each model to match the corresponding test. Right: test set performance separately for each of the elements forming a query (attributes, classes and constraints) when using just the top prediction. The error bars show the standard deviation across ten independent datasets.

We also observed that predicting the attributes (HumanMine 90.2% for $N = 1{,}000{,}000$; MySQL 86.6%; and Neo4j 96.9% for $N = 5000$) was more difficult than predicting the classes (HumanMine 92.1% for $N = 1{,}000{,}000$; MySQL 87.3%; Neo4j 97.4% for $N = 5000$). This is probably because, with the random query generation parameters used, the number of attributes in a query is generally larger than the number of classes, posing a greater prediction challenge. We note that HumanMine has 103 unique attributes (1003 in total as some, such as "Name" or "Primary Identifier", are used many times), while the MySQL and Neo4j examples have 59 and 42 attributes respectively, all of which are unique to the given example.

The number of classes in a query is one proxy for query complexity. In order to understand how this affects performance in the most complex case examined, we compared the influence of training dataset size on the performance of queries containing different numbers of HumanMine classes (see Fig. 4). As expected, queries with the smallest number of classes ($nc = 1$ and $nc = 2$) performed consistently better than the rest of the queries, achieving test set accuracies of 98.6% and 97.4% respectively, for $N = 1{,}000{,}000$. Also, it's worth noting that from $N = 50{,}000$, accuracies of over 90% are already achieved for such cases. For queries covering 3, 4 and 5 classes, the accuracies achieved are noticeably lower: 88.7%, 84.8% and 78.6%, respectively. Note that although queries containing five classes were included in the overall dataset, they were excluded from the training set.

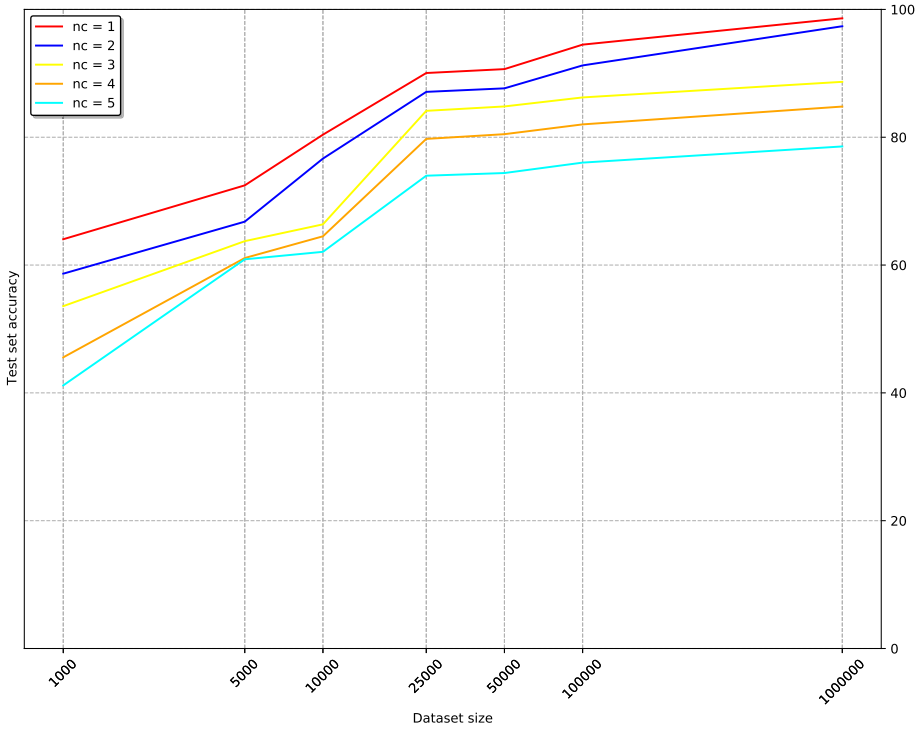

Figure 4. HumanMine performance as a function of number of classes in the query, as the number of training items varies, with $k = 1$. Note that queries containing five classes were absent from the training set.

Thus this gives a measure of the ability of the model trained on equal weighting of classes 1–4 data to generalise to five classes.

In terms of training time, we note that, even for the most complex database (HumanMine) and the largest dataset size (N = 1,000,000), training took around seventeen hours on a single low-end GPU system (workstation with a NVIDIA GeForce RTX 2070 GPU; Supplementary Figure 3). Therefore, for database schemas of the complexity considered here, training does not require special resources.

Finally, given that evaluating a machine learning method using a computer-generated dataset could lead to over-optimistic performance, we evaluated our framework on the WikiSQL[18] dataset. This dataset comprises 80,654 hand-annotated examples of questions and SQL queries distributed across 24,241 tables from Wikipedia. We kept the same source sentences (natural language queries) as they are in WikiSQL, but adapted the target to be the corresponding tuples of the query graphs. Since our framework is not able to deal with aggregation queries, we removed these from the training data, but they were taken into account for the accuracy calculations to allow a fair comparison. Training with the same architecture as the other experiments demonstrates reasonable performance on WikiSQL (76.5% validation, 73.4% test) when including the aggregation queries. If these are removed then the performance increases to 85.2% and 82.1% for validation and test sets respectively. In addition, we tested the effectiveness of training on synthetically generated data, as if no human-annotated data were available. For this, we trained Polyglotter on synthetic data generated from the WikiSQL schema, and tested the performance of the trained model on the real WikiSQL test set. Even though the training process was made up of only rule-generated language, the model achieved a test set accuracy of 61.3% when including aggregation queries and 69.4% when not, which we believe is reasonable performance when compared to the 73.4% and 82.1% achieved respectively when trained on the WikiSQL human-annotated dataset.

Furthermore, any single WikiSQL task takes as input both a natural language query and the columns of the relevant table, and from that the model generates the corresponding SQL query. In contrast, Polyglotter takes a less highly specified input, requiring only a natural language query, and from that, for SQL target databases, predicts which table or tables are needed to answer the query as well as the columns, constraints and query logic, which is a harder problem. For instance, in the case of SQL databases, it performs well (78.6%) on queries that require the correct five tables out of 170 to be identified, as well as identifying the correct components, constraints and logic to be applied to those tables.

The WikiSQL dataset covering 24,241 tables is arguably more diverse than the 170 class HumanMine dataset that was the most complex we generated, and we observed (Fig. 3) that performance was lower on more complex datasets. WikiSQL's 80,654 examples are most similar in size to the 100,000 example HumanMine dataset, which achieved 87.8% accuracy. Thus we would expect Polyglotter to achieve less than this on WikiSQL. The fact that performance only drops to ~75% demonstrates that the synthetic data approach we use to train Polyglotter's model is empirically sufficient to deal decently with actual human-generated datasets. This confirmed our belief that the synthetic data generation procedure captures sufficient language diversity that it can train a relatively effective model.

Conclusions and future work

We have described a flexible framework, Polyglotter, for building natural language interfaces to multiple database types. In Polyglotter, we abstract the concepts of query and database schema in the form of graphs. We build a schema graph from any supported database engine, where the nodes are the original classes in the database, connected by edges depicting the class-class relationships, and with further edges to the attributes belonging to each class. User queries are abstracted into a labelled subgraph of the schema graph, containing attributes, classes and constraints. These query graphs can in turn be represented in the form of class-attribute pairs and class-attribute-constraint triples. During training, the Transformer model learns how to predict each of the pairs and triples necessary to reconstruct the query graph of the original synthetic natural language query. In this way Polyglotter can translate the query graph abstraction to any supported database-specific query language. We provide support for two widely used engines (MySQL, Neo4j), and an example implementation for web service-based back-ends. The framework is easily extensible to further database engines.

Given that Polyglotter utilises a graph as an abstraction to represent queries, it can be used as a polyglot mediator, where a natural language query can be transformed into equivalent queries over multiple databases whether or not they use the same underlying engine. Thus Polyglotter has the potential to be an effective tool for cross-database querying and aggregation. In support of this we demonstrated that Polyglotter is capable of achieving good performance on three benchmark databases with different engines, with accuracies of 90.3%, 88.7% and 95.4% in the HumanMine, MySQL and Neo4j databases, respectively and 98.3%, 93.7% and 97.4% if the top three predictions are considered. These are assessed under conditions of limited overlap between the training and test datasets ($N = 1{,}000{,}000$ for HumanMine, $N = 5000$ for the others). Thus, perhaps surprisingly, performance is not closely related to the complexities of the underlying databases and is achieved without taking into account the typing of the individual attributes - doing so might yield modest improvements at the cost of a harder learning task.

As noted above on the real-world HumanMine database schema of 170 classes and over 1000 non-unique attributes, Polyglotter performance is 90.3% when considering one prediction but increases to 98.28% when considering the top three predictions. This demonstrates the value of giving users a choice of predictions before the query is executed, so that corrections can be made if needed, and exploits the inherently probabilistic nature of the Transformer model to provide these choices. Note that the predicted queries can be presented to the end user after translation back into synthetic natural language, and that this kind of natural language feedback can also be generated when the user is attempting to use conventional query building tools. Such feedback should help users who are learning to use query interfaces to determine whether they are moving in the intended direction.

The framework we describe does not come without drawbacks. The current random query generation process does not have the expressive power of the underlying database languages, such as SQL or Cypher. Solving this completely would be a difficult task, but it should be relatively straightforward, for instance, to extend the query generator to include aggregation operators. Another extension to the system could be the capability to understand queries about the database structure itself. Nevertheless, we believe that Polyglotter covers many of the types of queries that are useful in real scenarios.

Although we use a pre-trained FastText word embedding[30] to increase the chance of capturing semantic and lexical similarities across potential questions that have the same meaning, our current procedure to generate the training data from the database schema is based on a simple language model, where the main source of variability comes from permuting the different elements in the query. We try to alleviate this limitation a little by the use of synonyms in the question templates. Hence, there is an opportunity to improve overall performance and utility through improvements to the synthetic natural language generation procedure, and, for instance, generative adversarial network models[31], or transfer learning[32] might be options worth exploring.

However we also note that humans are good at understanding dialects, variations in speech and broken natural language. Therefore the quality of the generated natural language, whether for training or as an explanation for end-users, may not be so important in practice, i.e. it is not necessary to generate a perfect dataset for it nevertheless to be useful. When we asked biologists to read synthetic queries generated for the HumanMine biological database, they agreed that while the generated language was not perfect English, it was nevertheless understandable. The performance of Polyglotter on the WikiSQL dataset of human-generated queries was consistent with this utility. Here we compared a model trained on the human-annotated query set with one trained on synthetic data. Unsurprisingly, performance was higher when the human-annotated query set was used for training. However, the performance on the synthetically trained model was decent, demonstrating that the synthetic language generation approach captures sufficient natural language diversity to be useful.

A feature of query generation is that it can be done automatically in response to any database schema, whereas it is not realistic to expect that human-generated datasets can be generated on demand as easily. In addition, we note that an advantage of the current system is that it is relatively straightforward to add support for other languages, or to use an alternative way of generating the synthetic language. We hope that the ability to train Polyglotter without the need for an annotated dataset will be a useful feature in practice, and one that will increase the opportunities for Polyglotter to provide natural language interfaces to a wider audience of database maintainers, supporting queries across multiple tables.

Finally, we believe that a separate study is needed to explore the effectiveness of Polyglotter as assessed by human users, and plan to recruit a small cohort of biologists to provide feedback on how effective Polyglotter is in the context of the most complex biological database studied, HumanMine.

Data availability

The datasets used in this study are available at https://github.com/AdrianBZG/Polyglotter.

Code availability

The code developed for Polyglotter is publicly available at the Github repository https://github.com/AdrianBZG/Polyglotter under an Open Source Initiative (https://opensource.org/licenses) approved licence, LGPL 2.1.

Received: 12 May 2021; Accepted: 26 August 2021
Published online: 16 September 2021

References

1. Affolter, K., Stockinger, K. & Bernstein, A. A comparative survey of recent natural language interfaces for databases. *VLDB J.* **28**(5), 793–819 (2019).
2. Dar, H. S., Lali, M. I., Ul Din, M., Malik, K. M., & Bukhari, S. A. C. Frameworks for querying databases using natural language: A literature review (2019). arXiv:1909.01822.
3. Reshma, E. U. & Remya, P. C. A review of different approaches in natural language interfaces to databases. in *2017 International Conference on Intelligent Sustainable Systems (ICISS)*, 801–804 (IEEE, 2017).
4. Ozcan, F., Quamar, A., Sen, J., Lei, C. & Efthymiou, V. State of the art and open challenges in natural language interfaces to data. in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. SIGMOD '20 Series*, 2629–2636 (Association for Computing Machinery, 2020).
5. Blunschi, L., Jossen, C., Kossmann, D., Mori, M. & Stockinger, K. SODA: Generating SQL for business users. *Proc. VLDB Endow.* **5**(10), 932–943 (2012).
6. Shah, A., Pareek, J., Patel, H. & Panchal, N. NLKBIDB—Natural language and keyword based interface to database. in *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 1569–1576 (IEEE, 2013).
7. Wang, C., Xiong, M., Zhou, Q., & Yu, Y. PANTO: A Portable Natural Language Interface to Ontologies. in *The Semantic Web: Research and Applications*, Vol. 4519. Lecture Notes in Computer Science (eds Franconi, E. *et al.*), 473–487 (Springer, 2007). ISSN: 0302-9743, 1611-3349 Series.
8. Sinha, C., Reilly, R. G. & Sharkey, N. E. Connectionist approaches to natural language processing. *Am. J. Psychol.* **107**(2), 291 (1994).
9. Deng, L. & Liu, Y. (eds) *Deep Learning in Natural Language Processing* (Springer Singapore, 2018).
10. Young, T., Hazarika, D., Poria, S. & Cambria, E. Recent trends in deep learning based natural language processing [review article]. *IEEE Comput. Intell. Mag.* **13**(3), 55–75 (2018).
11. Yang, H., Luo, L., Chueng, L. P., Ling, D., Chin, F. Deep learning and its applications to natural language processing. in *Deep Learning: Fundamentals, Theory and Applications*, Vol. 2. Cognitive Computation Trends (eds Huang, K. *et al.*), 89–109 (Springer International Publishing, 2019).
12. Iacob, R. C. A., Brad, F., Apostol, E.-S., Truică, C.-O., Hosu, I. A., & Rebedea, T. Neural approaches for natural language interfaces to databases: A survey. in *Proceedings of the 28th International Conference on Computational Linguistics*, 381–395 (International Committee on Computational Linguistics, 2020).
13. Popov, A.. Deep learning architecture for part-of-speech tagging with word and suffix embeddings. in *Artificial Intelligence: Methodology, Systems, and Applications*, Vol. 9883. Lecture Notes in Computer Science (Dichev, C. & Agre, G. eds), 68–77 (Springer International Publishing, 2016).
14. Habibi, M., Weber, L., Neves, M., Wiegandt, D. L. & Leser, U. Deep learning with word embeddings improves biomedical named entity recognition. *Bioinformatics* **33**(14), i37–i48 (2017).
15. Bahdanau, D., Cho, K., & Bengio, Y. Neural machine translation by jointly learning to align and translate (2016). arXiv:1409.0473 [cs, stat].
16. Dong, L., & Lapata, M. Language to logical form with neural attention (2016). arXiv:1601.01280 [cs].
17. Sutskever, I., Vinyals, O. & Le, Q. V. Sequence to sequence learning with neural networks (2014). arXiv:1409.3215 [cs].
18. Zhong, V., Xiong, C. & Socher, R. Seq2SQL: Generating structured queries from natural language using reinforcement. *Learning* (2017). arXiv:1709.00103 [cs].
19. Xu, X., Liu, C. & Song, D. SQLNet: Generating structured queries from natural language without reinforcement. Learning (2017).arXiv:1711.04436 [cs].
20. Yin, P., Lu, Z., Li, H., & Ben, K. Neural enquirer: Learning to query tables in natural language. in *Proceedings of the Workshop on Human-Computer Question Answering*, 29–35 (Association for Computational Linguistics, 2016).
21. Sen, J. *et al.* ATHENA++: Natural language querying for complex nested SQL queries. in *Proc. VLDB Endow.* SIGMOD '20 Series, 2747–2759 (VLDB Endowment, 2020).
22. Weir, N. *et al.* DBPal: Weak supervision for learning a natural language interface to databases (2019). arXiv:1909.06182 [cs].
23. Kalderimis, A. *et al.* InterMine: Extensive web services for modern biology. *Nucleic Acids Res.* **42**, W468–W472. https://academic.oup.com/nar/article-pdf/42/W1/W468/17422978/gku301.pdf.
24. Klein, G., Kim, Y., Deng, Y., Senellart, J. & Rush, A. OpenNMT: Open-source toolkit for neural machine translation. in *Proceedings of ACL 2017, System Demonstrations*, 67–72 (Association for Computational Linguistics, 2017).
25. Klein, G. *et al.* OpenNMT: Neural machine translation toolkit. in *Proceedings of the 13th Conference of the Association for Machine Translation in the Americas (Volume 1: Research Track)*, 177–184 (Association for Machine Translation in the Americas, 2018).
26. Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization (2017). arXiv:1412.6980 [cs].
27. Klein, G., Hernandez, F., Nguyen, V. & Senellart, J.. The OpenNMT neural machine translation toolkit: 2020 edition. in *Proceedings of the 14th Conference of the Association for Machine Translation in the Americas (Vol. 1: Research Track)*, 102–109, Virtual (Association for Machine Translation in the Americas, 2020).
28. Vaswani, A. *et al.* Attention is all you need (2017). arXiv:1706.03762 [cs].
29. Freitag, M. & Al-Onaizan, Y. Beam search strategies for neural machine translation. in *Proceedings of the First Workshop on Neural Machine Translation* (2017).
30. Bojanowski, P., & Grave, E., & Joulin, A., & Mikolov, T. Enriching word vectors with subword information. arXiv:1607.04606.
31. Subramanian, S., Rajeswar, S., Dutil, F., Pal, C. & Courville, A. Adversarial generation of natural language. in *Proceedings of the 2nd Workshop on Representation Learning for NLP*, 241–251 (Association for Computational Linguistics, 2017).
32. Radford, A. *et al.* Language Models are Unsupervised Multitask Learners.

Acknowledgements

We thank the anonymous reviewers for their suggestions.

Author contributions

A.B. and G.M. designed the study. N.G. developed the initial version of the random query generator. A.B. further developed the random query generator, developed all other software including the training and prediction

pipelines. A.B. and G.M. devised the experiments. A.B. performed the experiments. A.B. and G.M. interpreted the results and wrote the manuscript. G.M. conceived and supervised the project.

Funding

This work was supported by the Wellcome Trust [208381]; Innovate UK [KTP011266]. For the purpose of Open Access, the author has applied a CC BY public copyright license to any Author Accepted Manuscript version arising from this submission.

Competing interests

The authors declare no competing interests.

Additional information

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1038/s41598-021-98019-3.

Correspondence and requests for materials should be addressed to G.M.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.