Natural Language Processing for Data Extraction

Abstract

Retrieving account information or payment details in a heavy banking and account transaction environment is extremely difficult and inefficient unless the underlying transaction details are well known. Banking information extraction can be retrieved only for individual static scenarios which requires technical knowledge and programing skills to retrieve transactions details and retrieve/update account information. There is no intelligent solution which can handle dynamic scenarios to retrieve transactions details and retrieve/update account information.

In this work, natural language processing is used for data extraction from a stored database for handling the dynamic scenarios . A NLP solution was developed where an user can enter a specific natural language query and the machine learning models process the query by identifying the intent and key words to retrieve a structured information from the database as a result for the input query. A POC solution was built for a customer and the solution is planned to be delivered as an integrated application for SAP solution.

Solution Overview

The process flow diagram shown below explains an overview of the solution.

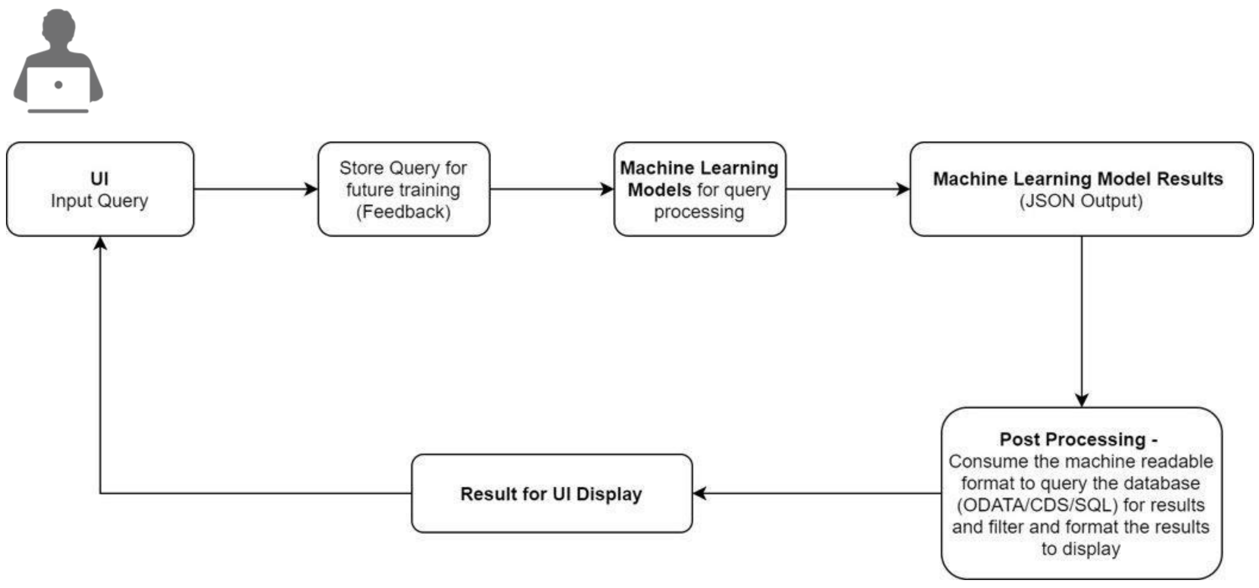

UI
Input Query
Store Query for future training (Feedback)
Machine Learning Models for query processing
Machine Learning Model Results (JSON Output)
Post Processing - Consume the machine readable format to query the database (ODATA/CDS/SQL) for results and filter and format the results to display
Result for UI Display

Figure 1 Flow diagram for the end-to-end processing of the unstructured text query

The user enters the query in the user interface (UI) and the query is sent to the machine learning (ML) models for processing. The ML model processes the query with an aim to identify the intent and the key words required to execute the query results. The machine learning output is used for post processing to generate the required structured format to query the database. The post processing step may have an ODATA, a CDS or a SQL query. The results from the database are obtained in a structure of either a json or xml format and may be further processed before displaying it to the user.

The query entered by the user is stored in the database and shall be used in the future for re-training and analysis.

Solution Components

The solution components which were used for processing the query and extracting the results from the database is shown in Figure 2 and are listed as follows,

1. Text Pre-processing
2. Machine Learning Model component
    a. Named Entity Recognizer
    b. Annotators
    c. Lexical Rules
3. Rules Classes
4. Semantic Parsing
5. Model Scoring
6. Post Processing
    a. Indent Mapping
    b. Fall back approach
7. User Interface Display

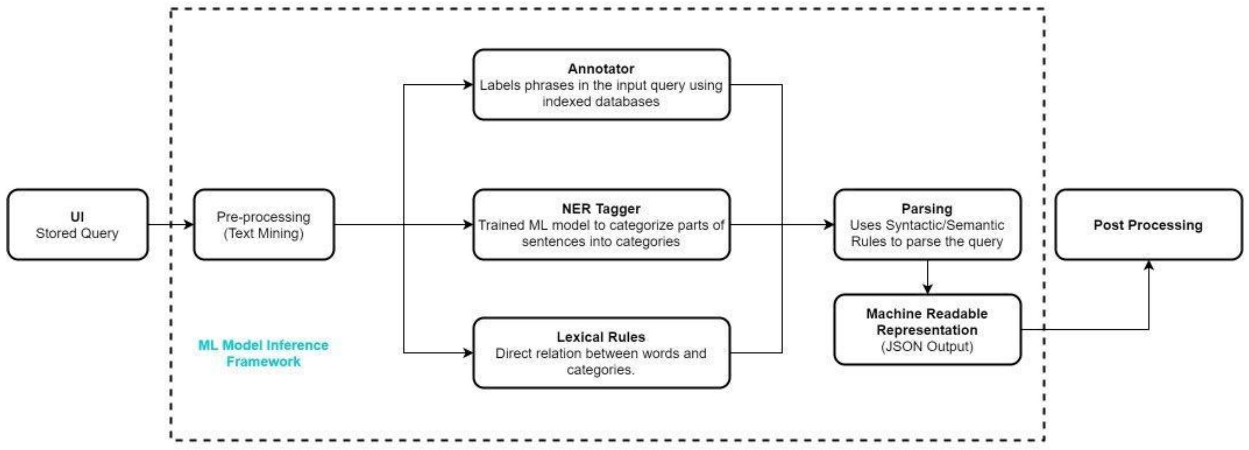

UI
Stored Query
Pre-processing
(Text Mining)
Annotator
Labels phrases in the input query using indexed databases
NER Tagger
Trained ML model to categorize parts of sentences into categories
Lexical Rules
Direct relation between words and categories.
Parsing
Uses Syntactic/Semantic Rules to parse the query
Machine Readable Representation
(JSON Output)
Post Processing
ML Model Inference Framework

Figure 2 Block diagram of ML Model Inference Framework to process unstructured text and provide recommendations/results.

In the Figure 1, the block diagram of the proposed ML model is presented. The UI interface receives the user query and stores it in the HANA database. It also forwards the query to the Machine Learning Model. The first step in processing a query by an ML model is preprocessing.

1. Text Preprocessing Query

Since the input query is written in natural language it may not have a fixed structure or form. The users may input unwanted characters or wrong spelling while writing the query. Also, the same query can be expressed in multiple ways using synonyms.

In order to deal with these following techniques are used to preprocessing:

1. **Removing unwanted characters:** The unwanted characters (@, ~) and the punctuation marks are removed from the query except the underscore and hyphen characters.
2. **Using spelling models:** The query is passed through pretrained spelling models to identify potential spelling mistakes and correct them.
3. **Converting to lowercase:** The entire query is converted to lowercase to make it easy to further process the query by the ML model.

Though all the above steps help us refine the query for further processing, the same query can still be written in multiple ways by use of synonyms. The sentences which are similar (in intent) yet are expressed differently (by use of synonyms) need to be classified under similar scenarios. This is handled in the ML model using the lexical rules and will be described in detail in the next section.

2. Machine Learning Model Components

The ML model tries to perform 2 tasks on the query.

- Intent Classification
- Entity extraction

To accomplish this task, the first step is to categorize each word in the sentence. These initial categories may not be the final entity classes that we are trying to extract. There are various methods that work simultaneously towards categorizing each word in the query. These methods are described in the following section.

a) Lexical Rules

A lexical rule is a form of syntactic rule used within many theories of natural language syntax. In this method we define certain words, that if found in the query, can directly be classified into a corresponding category. In other words, it a direct many to many relation from words to categories. This trivial task is fundamental for the downstream processes.

Eg:

$YESTERDAY	<-	yesterday
$RECEIVE	<-	receive

In this example the words ‘yesterday’ and ‘receive’ on the RHS are instructed by the lexical rules to be directly classified as $YESTERDAY and $RECEIVE respectively. There can exist multiple rules for the same word, say $REL_DATE <- yesterday. All the overlaps occurring due to such rules are handled in later stages.

These rules exist as a repository of known words that are expected to have a certain known semantic behavior in the query.

b) Annotators

Annotators are capable of enclosing logic for categorizing words. They can use databases to recognize words or regular expressions to recognize certain patterns and put words into corresponding categories. We can broadly categorize the annotators used into 2 categories:

1. Database based Annotators
2. Regular Expression based Annotators

The Database based annotators use domain specific dataset of entities to recognize entities that appear in a query. The domain database is an exhaustive collection of entities that can appear in the input queries. Each of these entities in the database is enriched with multiple string representations for the way it might appear in a query. These entities are indexed based on these text representations and the phrases in the query that match these indexed representations are categorized based on type of the matched entity.

Regular expression based annotator on the other hand are used for categorizing entities like alphanumeric codes that may appear in the query. Examples of such entities may include bank account numbers, employee IDs, equipment model numbers etc.

We in our application use the following annotators

1. TokenAnnotator: A regular expression based annotator that classifies every word it comes across as $TOKEN. As explained earlier every word can belong to multiple categories, hence, this annotator ensures that a word that was not classified by any of the methods mentioned in this section will always have a $TOKEN category to fall back to. This category is handled in the later stages in such a way that it does not interrupt the extraction of entities and intents from other meaningful categories.
2. AlphanumericAnnotator: A regular expression based annotator that classifies any word containing essentially a number and optionally any of the characters in the set {-,_, a to z, A to Z} into the category of $ALPHANUMERIC. For example, in the query ‘show the last transaction for bank account number 876309798’ the token ‘876309798’ will be categorized as $ALPHANUMERIC.
3. EntityAnnotator: This is a database based annotator that requires that the domain specific master database tables be available. It uses this data in the manner described above to categorize phrases appearing in the query that represent entities into their corresponding entity types. For example, in the query ‘What are the benefits of having a premium red card?’ the phrase ‘premium red card’ must be classified as $CREDIT_CARD or in the query ‘What is the interest rate for education loan?’, the phrase ‘education loan’ must be recognized and categorized as $LOAN.

c) Named Entity Recognizer

The last component in this step of assigning preliminary categories to every word or phrase is a named entity recognizer (NER). A pretrained, BERT-based model is employed for this task. The NER model is capable of recognizing the following categories of words or phrases:

PERSON, NORP, FACILITY, ORGANIZATION, GPE, LOCATION, PRODUCT, EVENT, WORK OF ART, LAW, LANGUAGE, DATE, TIME, PERCENT, MONEY, QUANTITY, ORDINAL, CARDINAL.

The reported F1 score of the model we use is 88.6. However, we only use 3 of the above categories for our task, namely, PERSON, DATE and MONEY. Based on the labels generated by the NER model, we append these categories to the preliminary categories of the words or phrases in the sentence. The following sections will explain how these categories are maintained, how they are used to finally extract useful entities from the query and also to determine the intent of the query.

3. Unary, Binary and N-ary Rules

Lexical rules described in a previous section is a special case of the Rules class. Other types of Rules include unary, binary and N-ary rules.

a) Unary Rules

These are almost the same as lexical rules with the slight modification that the RHS is also a category. A unary rule specifies a single category in RHS that can be said to belong to another parent category in the LHS.
These are some example

$REL_DATE	<-	$YESTERDAY
$DATE	<-	$REL_DATE
$DATE	<-	$NER_DATE

In the above examples all the intermediate categories on the RHS are said to belong to a parent category of $DATE.

b) Binary Rules

In this variation, the RHS can have two categories that if appear side by side in the query, can be said to be children of the category on LHS. This makes possible modelling semantic relations between words in the query.
As an example, consider a family of rules

$TO	<-	to	(lexical)
$FROM	<-	from	(lexical)
$TODATE	<-	$TO $DATE	(binary)
$FROMDATE	<-	$FROM $DATE	(binary)

In this example the last two rules are binary rules. This rule says that if a $TO category appears right before the $DATE category (described in previous sections), the expression contained by both these categories can be said to belong to the category $TODATE. Similarly, for $FROMDATE. As a result of this example, we have extracted a window of dates that was mentioned in the query.

c) N-ary Rules

This is a more general variation of the above rules. An N-ary rule has more than two elements in the RHS. These rules by themselves cannot be utilized by the parsing algorithm. But they are essential for effective modelling of a domain. To grant us this freedom of using such rules despite the above-mentioned limitation, a trick can be used. We break these n-ary rules into smaller binary rules that when put together can produce the same parses as allowed by the n-ary rule.

To binarize an n-ary rule, we combine all elements except the first one on the RHS and assign a new parent category to the rest. This creates a new rule from the original rule and the RHS of the original rule can be rewritten using the new parent category in place of all the categories except the first one in the RHS.

This example will make this process clearer.
Say we have an n-ary rule

```
$Z      <-      $A  $B  $C  $D
```

This has four elements on its RHS.
We create a new category $Z_$A and break the original rule into the following two rules.

```
$Z_$A   <-      $B  $C  $D
$Z      <-      $A  $Z_$A
```

This process can convert an n-ary rule with any length of the RHS, by repeated breaking, into a family of binary rules.

4. Parsing Algorithm

The parsing algorithm brings together all the components described above to achieve entity extraction and intent classification. Certain helper data structures were employed for this purpose. These are described below:

1. **Rule**: A data structure that can store all types of rules described in the previous sections. It consists of 3 attributes, namely ‘lhs’, ‘rhs’ and ‘sem’. We have already talked about what ‘lhs’ and ‘rhs’. Briefly, ‘lhs’ holds the parent category name and the ‘rhs’ holds the sequential list of children categories or words. The third attribute, ‘sem’ holds a function that is responsible for producing the semantic representation of the query. This semantic representation is the target of our parsing algorithm.
2. **Parse**: This structure imitates the role of a tree node. A parse object holds the parent, children and the semantics of a node in the tree that is formed during parsing of a query. The children of a parse object are other parse objects. All the parses generated during a parse are stored in the chart object.
3. **Chart**: This is a 2-dimensional matrix of size NxN where N is the number of words or tokens in the query. In this matrix the cell[i, j] holds all the Parse objects that are generated for part of the query that starts with the i^{th} token and ends with the j^{th} token. As a result, the cell[0, N] will hold all the parses that are generated for the whole query and these can be regarded as being the root nodes for the trees that are generated during the parsing algorithm. Moreover, the semantics of the Parse objects residing in this cell must essentially contain our target representation (semantic representation) of the query.

This algorithm is an example of the chart parsing algorithm that can be put more broadly into dynamic programming. This relies on all the data structures described above. Given a set of rules, this tries to find the set of parses for the input query that are allowed by the rules.
This is how the chart parsing algorithm works:

- It splits the input query into a sequence of tokens
- Create a chart that can hold a list of possible parses for every span of the input query in the appropriate cells.

- Iterate over all possible spans of the query, starting from the smallest up until the span containing the entire query.
- For every span of the query, try to apply all the rules, annotators and the NER model, to see if a parse can be generated and if so, add the parse to the chart.

The following section describes how each of the components tries to create a parse for a given span.

Annotators

Given a span, every annotator calls a annotate function that takes as input the tokens contained in the span and tries to return a list of categories for the span. If the list is non-empty, a Parse object is built for every item in the list and added to the chart.

Lexical Rules

For the given part or span of the query, the algorithm looks for a lexical rule that has a matching RHS and generates a Parse object for every match. All these parses are added to the chart cell that corresponds to the given span.

Binary Rules

To generate parses for binary rules we iterate over the indices that can split the given span into two parts. For each split point, we retrieve all the parses that have already been generated for the two subspans produced by the split point. We look at each possible pair of parses from the two subspans and retrieve all the binary rules that can be used to combine them. For each rule, we build a Parse object and add it to the chart.

Unary Rules

Looks at the cell in the chart corresponding to the given span. If there exists a unary rule whose RHS is the same as the parent in any of the parses found in the cell, a new Parse object is created with the rule's LHS as the parent and the Parse object that has the matching parent as the child. This Parse object is then added to the same cell in the chart.

NER Tagger

This is run on the entire query at once and spans in the query that are labelled by the NER are directly added to the corresponding cells in the chart.

Generating Semantics

At this stage we have generated parses for the query (parses in the cell[0, query_length] of the chart). These parses still exist in the form of a tree representation of the various category elements in the input query. We need to, however, convert this representation into the structured key value pair output that is required by the model. In other words, Parse objects are required to be converted to corresponding semantic representations. We make use of the third attribute 'sem' of rules for this purpose. This attribute can store either a function or a string. If it holds a string value, the said string value becomes the semantic representation of the parse node that is built on the rule. On the other hand, if the 'sem' attribute of a rule is a function, the semantic representation of the parse node built on this rule is the value that is returned by the 'sem' function. The 'sem' function takes the semantic representations of the children of the parse node as inputs. Hence calling the semantic representation of the root parse node subsequently triggers the semantics of child parses and the structured output (semantic representation) for the entire query gets aggregated at the root node.
This example explains the process:
Query: show transactions for CRN number 12334.

Rules:

LHS		RHS	SEM	
$ROOT	<-	$QUERY_ELEMS	sem[0]	(unary rule)
$QUERY_ELEMS	<-	$QUERY_ELEM $QUERY_ELEMS	merge_dicts(sems)	(binary rule)
$QUERY_ELEM	<-	$INTENT	sems[0]	(unary rule)
$QUERY_ELEM	<-	$ENTITY	sems[0]	(unary rule)
$ENTITY	<-	$CRN_NUM	{“CRN_NUMBER”: sems[0]}	(unary rule)
$CRN_NUM	<-	$CRN $ALPHANUM	sems[1]	(binary rule)
$INTENT	<-	$PAYMENT_INFO	{“INTENT”: sems[0]}	(unary rule)
$PAYMENT_INFO	<-	$PAYMENT	“payment_info”	(unary rule)
$PAYMENT	<-	payments	null	(lexical rule)
$PAYMENT	<-	transactions	null	(lexical rule)
$CRN	<-	CRN	null	(lexical rule)
$ALPHANUMERIC	<-	12334	“12334”	(AlphanumericAnnotator)

In the above example the SEM column shows the “sem” attribute of the rules. The values that are enclosed in quotes are simply strings. All others are functions where the variable ‘sems’ is the input to the functions and holds a list of semantics of the children Parse objects. The function merge_dicts(sems) merges the dictionaries that may be members of the list held by sems.

The resulting semantic representation of this query will be
{
“INTENT”: “Payment_Info”
“CRN_NUMBER”: “12334”
}

5. Scoring Model

The parsing algorithm generates a set of all parses allowed by the rules. The challenge now remains to choose a parse that is the most accurate in terms of the target semantics that we require. This is where a scoring model is needed. The scoring model assigns every parse of the input query a score for it to be ranked. The parse with the maximum score is then selected as the final output.

Feature Function
To achieve this type of scoring mechanism, the first requirement is to obtain features for every parse. To generate features for parses we use a feature function. One of the simplest implementations for a feature function is using rule features. A rule feature is simply the number of times a rule has been used to generate a parse. This can be explained using a simple example:
Query: from 1 July to 31 December
Parse: ($QUERY ($TODATE ($TO to) ($DATE 31 December)) ($FROMDATE ($FROM from) ($DATE 1 January)))
Features:

($TO, to)	:	1
($FROM, from)	:	1
($DATE, 31 December)	:	1
($DATE, 1 January)	:	1
($TODATE, $TO $DATE)	:	1
($FROMDATE, $FROM $DATE)	:	1
($QUERY, $TODATE $FROMDATE)	:	1

We then define a weights vector that stores the real-valued weights for each feature. The score of a parse is computed as the inner product between the feature vector and the weights vector. There exist other ways to generate parse features, but they are not described here.

The weights can be trained to ensure that the most accurate parse is has the maximum score.
We use a unary rule that lets a $TOKEN (tagged by the TokenAnnotator) belong to $Optional
$Optional <- $TOKEN

In our case, we set weights for all the rule features having LHS as "$Optional" to -1. This ensures that the parse that is selected does not leave out any of the meaningfully tagged words and uses maximum number of useful words to generate the semantic representation or the structured output.

ML Output

The output of the ML model is represented in a structured json format. It contains the fields extracted from the input query which would be required as input for retrieving information from the database. The fields extracted correspond to the columns of the data in the database. While building the filter for calling the OData we may directly use the values from the json format. Following this nomenclature makes the process of calling the OData easier and is described in detail in the next section. Apart from the information needed to process the query the json output also contains two additional fields namely Intent and Fall back code. The fall back code has a value zero in case the ML model was successful in identifying the intent of the query otherwise it is one. The intent specifies the task the query wants to achieve and helps us identify the OData to call to process the query.
The structure of the json as shown in Figure 2). All the values of the fields are NULL except the fields which are required by scenario and are provided in the input query.

```
{
FallBackCode : 0,
Intent : Provide_Info,
FirstName : NULL,
LastName : NULL,
TransactionID : NULL,
DateFrom : 1-12-2020,
DateTo : 31-12-2020,
ProgramID : NULL,
BankInfo : NULL,
UniquePersonID : ID12345,
AccountStatus : NULL,
StatusReason : NULL,
MathOperator : NULL,
}
```

Figure 3 The output of the ML model which is in the json format.

6. Intent – OData Mapping – Invention Novelty

The intent of each input query is identified and produced as the output of the ML model. Each of this intent is mapped to an OData link in the file. When the output of the ML model in json format is received, this file is referred to obtain the corresponding OData and the URL is built considering the format of the OData structure. Different OData structures were created for both calculation views and function modules. The process of building the filter and calling the final OData URL comes under post-processing and is described in next section.

7. Postprocessing – Creating OData link - Invention Novelty

The OData link has two parts: - the base URL and the filter. Base URL is obtained from the intent OData mapping and the filter is built from the json output. The filters are built using the entities extracted from the NLP query.
There are two kinds of OData URLs. The first is to display the results of querying the database while another is to perform some operations on the entries of the database.

1. **OData link for retrieval queries**: The OData URL is retrieved corresponding to the intent and the filter is appended to the URL. The filter comprises of all the non-Null values from the json output. The keys of the json object are according to the column names in the database. In case the keys are different from the column names in the database, an additional table is needed for mapping the keys of json output to the column names in the database.
2. **OData link for function queries**: There may be a need to apply some functions like sum, min, max or average for the entries in the database. The math_operator in the json output identifies the function and the query is built like the procedure described above.

Note: The CDS calculations via HANA calculation views to create the OData service is not a part of this document and excluded for the invention.

8. Postprocessing – Creating Sub-intent for OData link - Invention Novelty

Some intents can have a child intent called sub-intents which is used for secondary level Odata link mapping. Since the retrieval queries and function queries can have different Odata mapping for the same intent, hence, sub-intents will help to identify the underlying Odata mapping for different scenarios. For eg, A person can be recognized using identification number and/or name. In this case, the Odata mapping has to work for these two different mechanisms. Using sub-intent mapping, the engine can select which Odata mapping to be used based on the input query of identification number or name. In another example, for functional queries, an aggregation like sum, min or max, a sub-intent can be used for creating the Odata mapping under the parent intent.

9. Fall back code

If the ML model is unable to map the intent of the query to any of the known intents, the fall_back_code in the json output is set to 1.

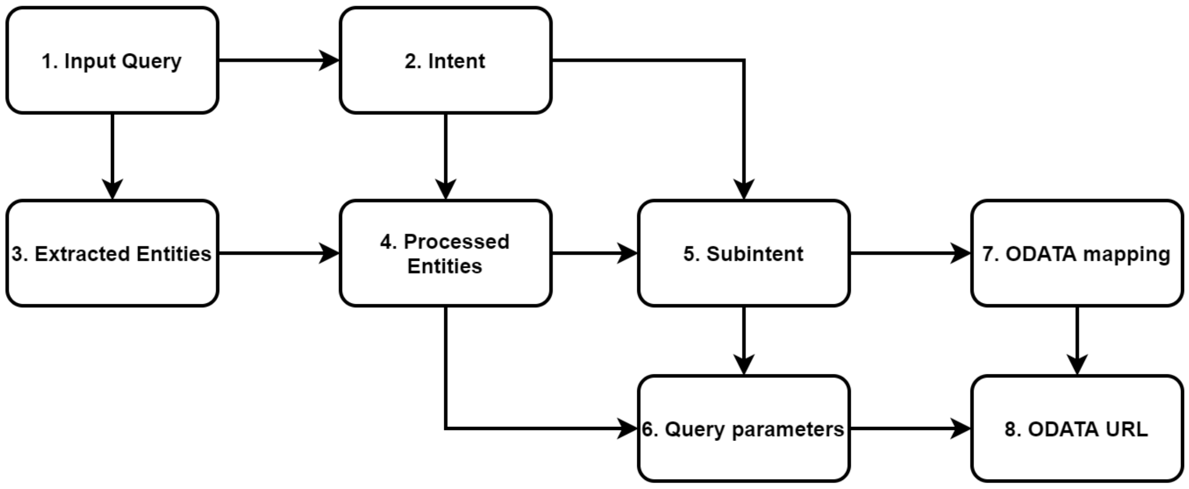
10. Display the result

The output of the OData URL call is in xml or json format.
The output is consumed by a UI where it is displayed in xml/json format or it may be converted to tabular format.
There may be two kinds of fall back situations. The first is when ML model is unable to understand the intent of the query and the second is when the URL is successfully built but cannot be connected to the database because of connection, authorization or other issues.

The first case can be identified by the ML json output where the fall_back_code signifies the success or failure of the ML model in processing the query. In case the ML model is successful in building the query but there are other connection issues can be identified as a different error code.

ER01	*"We could not understand your query. Please try again."*
ER02	*"Please check your connection, and try again."*

11. Additional Information – Flow Chart



Example:
Consider the input query,
"Show all transactions for William Johnson in the last month. His account number is 12348765 and postcode is 1201."

The intent and the entities extracted from this sentence are:

```
{
Intent: "Bank_Transactions",
Name: "William Johnson",
Date: "last month",
BankAccountNumber: 12348765,
Address: "postcode is 1201"
}
```

Based on the intent, the extracted entities are processed.
The intent Bank_Transactions only needs Name, Date and BankAccountNumber for querying information. Hence Address is discarded. Also, Date needs to be derived from the text and name needs to be split into first and last names. Hence, after processing, the intent and entities in cell 4 of the diagram look like this:

```
{
Intent: "Bank_Transactions",
FirstName: "William",
LastName: "Johnson",
```

FromDate: 2020-05-29,
ToDate: 2020-06-29,
BankAccountNumber:12348765
}

Based on these entities, the subintent is determined. In this case, due to the presence of BankAccountNumber, the subintent is “AccNumQuery”. Also, this subintent does not require a name field as BankAccountNumber is enough information. Hence the intent, subintent, and entities become:
{
Intent: “Bank_Transactions”,
Subintent: “AccNumQuery”,
FromDate: 2020-05-29,
ToDate: 2020-06-29,
BankAccountNumber:12348765
}

These entities can now be inserted in the ODATA URL to get the required information. Every (intent, subintent) pair is mapped to a base Odata URL and the processed entities are converted into arguments for the query. Based on the Odata URL, the syntax of the constructed arguments may vary. Example syntax: “(<key>=<value>)” or “(<key> eq <value>)”.

For this scenario, the ODATA mapping is “BANK_TRANSACTIONS_ACC_NUM” for the intent “Bank_Transactions” and subintent “AccNumQuery” pair.

Based on the intent and subintent, the query parameters are generated. The information shown above translates to the following arguments:
“(TransDate ge datetime(‘2020-05-29’) and TransDate le datetime(‘2020-06-29’) and BankAccountNumber eq ‘12348765’)”

And the linked base URL for the given intent, subintent pair is
“/sap/opu/odata/get_data/BANK_TRANSACTIONS_ACC_NUM/”

Combining the query parameters and the base URL we get the final Odata URL as
“/sap/opu/odata/get_data/BANK_TRANSACTIONS_ACC_NUM/?$filter=(TransDate ge datetime(‘2020-05-29’) and TransDate le datetime(‘2020-06-29’) and BankAccountNumber eq ‘12348765’)&$format=json”

The ODATA URL can also be formed using “xml” instead of “json” format.

12. Additional Information – Other Areas

(a) Breadth of scenarios that a single deployment of the disclosed technology can handle
Here are some of the examples of the types of queries that can be handled by a single deployment of the system:

1. I want to view transactions of Person ID12345 done in last two months.

2. Please display the transactions done under program PJ123 between 21 Jan 2020 to 25 Feb 2020.
3. Display all the users whose account are disabled.
4. Display all the users which are enrolled in program PJ123 who have their account disabled.
5. State the reason the account of Peter Adams is disabled.
6. Show the bank info of Peter Adams.
7. Show the bank details of Person with ID12345.
8. What is the maximum amount paid to the Person ID12345 in last two months.
9. What is the minimum amount scheduled to be paid to Peter Adams in next three months.

All the above examples have the entities which map to one of the entries of the json format given in Figure 3.
Although the json entries are not exhaustive and can be increased according to our requirments. If we have more scenarios as given below:

1. Display the social security number of Peter Adams.
2. Display the sum of the transactions returned within last year from return id R1234.

In the first example an extra intent would be added to handle such queries. In the second example we are assuming that each return is labeled with an ID depending upon the person or post it was returned from. In such cases the json is expanded to contain additional attribute *ReturnID* which will be given the value R1234 and an additional intent for this scenario will be added.

Thus, even within a single deployment, the current solution is highly scalable to contain expanding requirements.

(b) Additional application areas beyond banking that the technology could be applied to

This technology can be applied to

- Any manufacturing, chemical and Oil and Gas industry which has process equipment’s,
  - Example scenarios,
    - What is the outlet temperature of boiler id 12345
    - Display the compressor ID number used in water treatment plant 2
    - What is the threshold limit inlet temperature for the heat exchanger in smelter plant
    - Display the average pressure outlet values of boiler id 12345
- Any retail industry to check the sales data
  - Example scenarios
    - What is the sales yesterday at the retail outlets in postal code 434567
    - What is the sales for last month at all the retail outlets in postal code 434567
    - Which product was sold in the retail id 12345 yesterday
    - What is the closing stock for the product 23456 at retail id 434567

- Any telecom industry to check network related data
    - Example scenarios
        - What is the frequency of the threshold limits exceeded in network id 123456
        - When was the last auto healing happened in the network id 123456
        - What is the maximum number of users on a network id for yesterday
        - What is the maximum sessions established on a network id for last two months
- Any Transport industry to check the route and locations
    - Example scenarios
        - What is the total tonnage transported by the vehicle id 234567
        - Where is the current location of the vehicle id 384865
        - How many vehicles are currently near postal code 345678
        - What was last month rent received for the vehicle id 3546478
- Any Real Estate industry to check the asset data
    - Example scenarios
        - What was rent received for the asset id for last one year
        - What is the asset id in the XY location which has 30000sqft
        - Show all the assets which has “vacate” status in the postal code 12345