NExt ApplicationS of Quantum Computing


# D6.3: WP6 QNLP Pre-Alpha prototype

## Document Properties

| | |
|---|---|
| Contract Number | 951821 |
| Contractual Deadline | M12 (31/08/2021) |
| Dissemination Level | Public |
| Nature | Report |
| Editors | Antonio Villalpando, ICHEC<br>Kaspars Balodis, Tilde<br>Rihards Krišlauks, Tilde<br>Venkatesh Kannan, ICHEC |
| Authors | Antonio Villalpando, ICHEC<br>Kaspars Balodis, Tilde<br>Rihards Krišlauks, Tilde<br>Venkatesh Kannan, ICHEC |
| Reviewers | Vedran Dunjko, ULEI<br>Andrés Gómez, CESGA |
| Date | 25/08/2021 |
| Keywords | Quantum Natural Language Processing, prototype |
| Status | Final Review |
| Release | 1.0 |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 951821

## History of Changes

| Release | Date | Author, Organisation | Description of Changes |
|---|---|---|---|
| 0.1 | 23/07/2021 | Antonio Villalpando | Background, task explanation, mathematical representation, quantum module and results |
| 0.2 | 03/08/2021 | Antonio Villalpando, Kaspars Balodis, Rihards Krišlauks, Venkatesh Kannan | Submitted for First Review |
| 0.3 | 20/08/2021 | Antonio Villalpando, Kaspars Balodis, Rihards Krišlauks, Venkatesh Kannan | Submitted for Second Review |
| 1.0 | 25/08/2021 | Antonio Villalpando, Kaspars Balodis, Rihards Krišlauks, Venkatesh Kannan | Submitted for Final Review |
| | | | |

# Table of Contents

## 1. Executive Summary

The NEASQC project aims at demonstrating and advancing the capabilities of NISQ-era devices through the development of practically-relevant use-cases. Under the category of Symbolic AI and Graph Algorithmic algorithms, one of the use-cases that is being developed is for Quantum-enabled Natural Language Processing (QNLP). The objective of the QNLP use-case in NEASQC is for the investigation, development and comparison of existing methods in classical NLP with a QNLP approach for encoding and processing sentences in a hybrid classical-quantum workflow.

For this, deliverable D6.1 “QNLP design and specification” was presented in M6 with an overview of the background and existing approaches for classical NLP and quantum NLP along with a detailed illustration of the proposed QNLP software architecture solution and methods for testing and benchmarking the QNLP implementation.

Deliverable D6.3 “QNLP pre-alpha prototype” is the first version of the QNLP software (pre-alpha prototype) which is primarily aimed at assessment within the NEASQC project. D6.3 implements a first version of the modules for generating training datasets (composed of sentences of specific grammatical structures), quantum circuits for training using single sentences and whole datasets, and classical NLP approaches for evaluation. This report accompanying D6.3 provides an overview of the currently implemented modules along with access and usage details of the QNLP pre-alpha prototype.

## 2. Background and approach

Following the effectiveness of variational algorithms for NISQ devices, we intend to use methods for state preparation and encoding of corpus data, along a similar line to that of (Coecke et al., 2021) and (Meichanetzidis et al., 2020). Given the tensor-network-like relationships for describing sentence relationships, we can aim to take advantage of this formalism by representing the encoding quantum corpus state as a matrix-product state (MPS), with operations performed to evaluate transforms and contractions using matrix product operators (MPO), as discussed in (Biamonte & Bergholm, 2017)(Orús, 2014)(Huggins et al., 2019)(Bai et al., 2020).It is important to note that in this case, using high bond dimension states is essential to leverage quantum advantage.

Operations on the tensor network can be optimised to run well on both classical systems for verification, analysis and comparison of the methods. For this, we also make effective use of the variational algorithms that have been offering great promise for NISQ devices, given their tolerance towards noise. As such, by exploiting the state preparation capabilities of variational models, and with the representability of tensor network models, we expect one can prepare states to offer a large area of exploration and data representation methods, for NLP and beyond.

### 2.1. Pregroup grammars

A pregroup grammar is a mathematical model of natural language grammar introduced by Lambek in 1999. A pregroup is a structure $\mathcal{G} = (G, \leq, \cdot, l, r, 1)$ such that $(G, \cdot, 1)$ is a partially ordered monoid, $l$ and $r$ are unary operations on the elements of the algebra $G$, satisfying the inequalities:

$$a^l \cdot a \leq 1 \leq a \cdot a^l \text{ and } a \cdot a^r \leq 1 \leq a^r \cdot a,$$

for all $a \in G$. The elements $a^l$ and $a^r$ are called the left adjoint and the right adjoint, respectively, of $a$.

A pregroup grammar is a grammar formalism consisting of a lexicon of words, along with a set of types which generates a pregroup, and also a mapping that relates words to a set of types.

We have then a set of types (in this case $n$ and $s$), which generates the preordered algebra $G$ through the $(\leq, \cdot, l, r)$ operations. This gives as a result a set of categories, in the theoretical sense, that are mapped to parts of speech. For example, we have nouns assigned to category $n$, or transitive verbs assigned to category $n^r \cdot s \cdot n^l$.

#### 2.1.1. DisCoCat model

The DisCoCat model, introduced by (Coecke et al., 2010), uses pregroup grammar and category theory to combine the benefits of two approaches to linguistics: categorial grammar and distributional semantics.

Whereas in classical NLP the distributional approach has been successful, leading to complex models such as the Transformer (Vaswani et al., 2017), or RNNs, the DiscoCat model uses the structure of quantum mechanics to add the insight of categorical grammar to learning models, and not relying only in a distributional approach.

In the DiscoCat model, words are represented as quantum states. These states can be viewed as tensor following Penrose notation, their legs given by the categories they represent. The bond dimension is determined by the parameterization. Then, some contractions are done between the proper legs as shown in the diagrams, following pregroup grammar partial ordering rules, to form different sentence structures (Coecke et al., 2020) (Coecke et al., 2010).

This way, we can have sentences embedded as quantum states, and comparing sentences of different sizes and structure, as they will always be tensor states with legs corresponding to the dimensionality $q_s$ of the sentence category $s$.

To solve the problem of implementing this model in quantum computers, we rely on variational algorithms and MyQLM simulator as a backend for the code (Atos, 2021). These algorithms, which are leading the NISQ era, are widely used given the current limitations of quantum devices in terms of qubits and circuit

depth (Bharti et al., 2021). They perform the optimization part on classical computers, and use the quantum machines only to prepare and measure the quantum states (Cerezo et al., 2020).

## 3. Proposed NLP task and implementation

In the QNLP pre-alpha prototype, we implement sentence classification into true or false categories which is also used as the benchmarking task. In this approach, we are also able to select the word that fits best a sentence with a missing word, as we can compare the level of truthfulness of the sentences when we input a set of words. This second example is expected to be implemented for the alpha prototype (D6.7 in M20). The training and test datasets consist of short simple sentences stating facts about some simple domain, in this case the animal kingdom. We have adapted the dataset used by (Toumi & Koziell-Pipe, 2021) but supplement it with additional false sentences. The true and false sentences are correspondingly labeled. We keep the dataset size relatively small — on the order of 100 sentences.

The motivation for using a manually created dataset for the pre-alpha prototype is twofold:

1. It is unlikely that we will be able to filter some existing large datasets, containing a broader class of sentences, down to something that's suitable for ingestion by the pre-alpha prototype.
2. It is hard to randomly generate a natural-looking synthetic datasets that wouldn't invalidate the use of pre-trained word embeddings.

### 3.1. Mathematical representation of the problem

Following the philosophy of the DisCoCat model, sentences can be represented graphically as shown in Figure 1 for a transitive sentence consisting of NOUN-TVERB-NOUN:

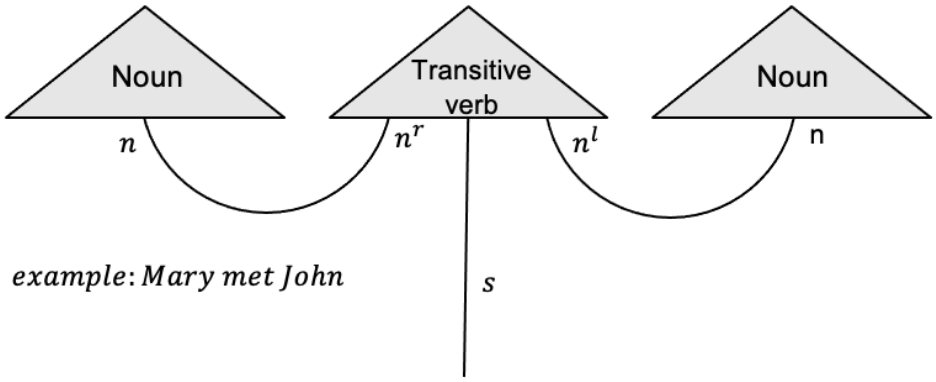



***Figure 1**: Diagram for transitive sentence*

While in Figure 1 a simple diagram is shown, Figure 2 illustrates the model for more complex sentences (an intransitive sentence with a preposition) with a different structure and parts of speech. It can be seen how the states involved have more tensor legs, the number of contractions needed increase and the word categories found are more complex for this second example. As will be clear later when we show how these diagrams are translated to quantum circuits, the number of qubits needed for the sentence increase as the sentences are longer and more complex. The number of parameters to train becomes large if we use several qubits for each category and have an extensive vocabulary involved. Also, creating these diagrams from the sentences will require coding pregroup grammar rules to detect the contractions automatically.

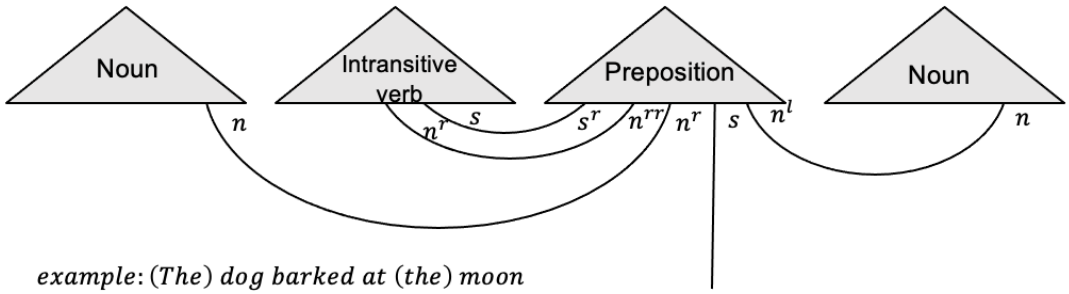



***Figure 2**: Diagram for intransitive sentence with a preposition*

Here, the triangles pointing upwards are quantum states or $kets$, and the lines are correspond to tensor legs. A triangle upside down would be a quantum effect or a $bra$. The cups, which we refer to as contractions, are the result of inverting the triangles corresponding to nouns, and applying projective post-processing to get the desired quantum states. Figures 3 and 4 illustrate these objects in terms of the circuit model of quantum computing (Meichanetzidis et al., 2020).

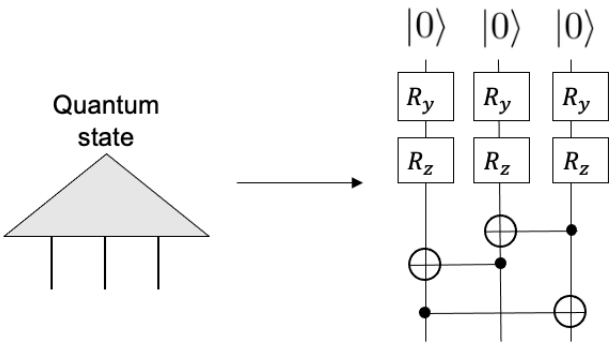



***Figure 3**: The quantum state Ansatz for the DisCoCat model used in our implementation.*

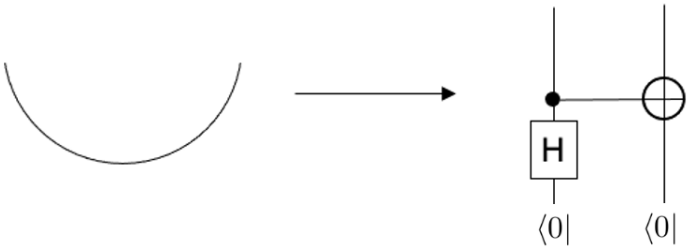



***Figure 4**: A cup or contraction in terms of quantum gates.*

## 3.2. Quantum circuits

With the information above, the quantum circuits for different sentences can be built. Following the Ansatz presented in Section 3.1, a quantum circuit embedding the transitive sentence shown in Figure 1 would look as shown in Figure 5.

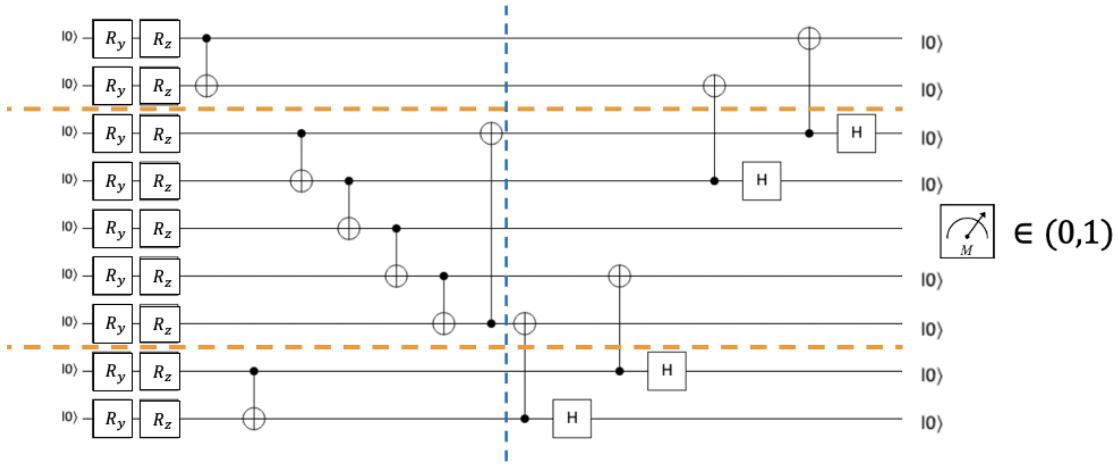



***Figure 5**: Quantum circuit for a transitive sentence.*

Here, the part of the circuit to the left of the blue dashed line corresponds to word preparation (or the triangles in DisCoCat diagrams), and the part to the right corresponds to cup contractions. The orange dashed lines separate words.Let $q_x$ be the number of qubits we use for a certain category $x$. This will be the dimensionality of that leg, and it will define how many qubits are needed to encode a word. The image above corresponds to a choice of $q_n = 2$ and $q_s = 1$.

All the qubits are then measured. We are interested in the output states where all qubits are 0 but the one/s corresponding to the $s$ leg, as is derived for the shape of the cups in terms of gates. This or these qubits are the ones carrying the truth value of the sentence, being a *False* sentence when the value is $0$ and a *True* sentence for value $1$. This output is then compared with the label of the sentence within the dataset.

## 3.3. Training pipeline

For training the model parameters we rely on the `SciPy` package. Currently, Atos `myQLM` does not support custom cost functions for model training. The `ScipyMinimizePlugin` is a particular implementation of the `Optimizer` class which is capable of optimizing the energy or expected value of an observable given a set of parameters in a quantum circuit.

The goal of our implementation is different, as we are selecting specific states and calculating the probability of occurrence of these states against others. As `QLM` Optimizer is using `SciPy` under the hood, this different approach to the learning aspect could eventually open a possibility to contributing to `QLM` development including custom cost functions for datasets.

Cross entropy is used for the cost function, as it is commonly used for binary classification tasks. 'False' sentences will be labelled as $0$, and 'True' sentences labelled as $1$.

Let $y_i$ be the truth value of the $i^{th}$ sentence, and $\hat{y}_i$ be the probability of the sentence qubit being $0$ in the quantum circuit for that sentence. Then, we define the cost function $c(y, \hat{y})$ as:

$$c(y,\hat{y}) \begin{cases} y = False \rightarrow c = -\log(\hat{y}) \\ y = True \rightarrow c = -\log(1-\hat{y}) \end{cases}$$

For a dataset consisting of $N$ sentences, the loss function $\mathcal{L}$ corresponding to the dataset will be:

$$\mathcal{L} = \frac{\sum_i c(y_i, \hat{y}_i)}{N}$$

The objective then is to use a variational method to find the angles that can be used to encode the words in a way that when running the quantum circuits for a set of sentences, their truth value can be guessed. The `"COBYLA"` optimizer has been used for this task, as it has been found to provide

the best convergence among all the methods available in `SciPy`. `tol` and `rhoberg` parameters of `scipy.optimize.minimize` are adjusted to yield the best convergence, although for larger datasets it is not always achieved.

# 4. Pre-alpha prototype implementation

For D6.3, the focus is on developing a working pipeline to show a proof-of-concept implementation of the tasks described in Section 3, and in the subsequent prototypes will focus on the QNLP modules that were outlined on (Villalpando et al., 2020). We believe that the C4 diagram is a good starting point and a guideline for the development, but the final structure will be determined by the evolution and necessities of the package.

## 4.1. GitHub repository

For the internal review, the code for the QNLP pre-alpha prototype is hosted in ICHEC Git repository. The following link is for the current tag proposed for this release: https://git.ichec.ie/neasqc-qnlp/wp6_libtemplate/-/tree/9959575b8909e193af3c334c642e1c7825379e4a/. This can be downloaded for a local copy as zip file or using

```
git clone https://git.ichec.ie/neasqc-qnlp/wp6_libtemplate.git
```

in a terminal. The files will be then downloaded in the terminal directory. If Git is not installed in your computer, you may download it from the following link: https://git-scm.com/downloads.

After the review the code for the QNLP pre-alpha prototype will be hosted in the NEASQC GitHub repository, which still needs to be populated and will be made public at the end of the review process.

### 4.1.1. Installation guide and dependencies

Instructions for setting up a Python environment to be able to run the developed software are provided in the `README.md` file in the project's repository. The instructions describe software dependency and language model installation for the project.

### 4.1.2. C4 diagram and developed modules

Figure 6 presents the proposed software for the QNLP use-case. The modules under development in D6.3 (the pre-alpha prototype) are `DataPreparation` and `Quantum`. A first version of `SoftwareTesting` and `Benchmark` will be available following the delivery of D6.3 to test both the Atos GitHub integration library repository template and the `.json` file according to the methodology proposed for application-centric benchmarks in Task 3.3.

### 4.1.3. Dataset generation

Dataset generation is implemented in the `DataPreparation` module. It defines 112 sentences which can optionally be randomized, split into training and test datasets and output to JSON. This functionality is implemented in the following submodules:

- **gen_animal_dataset.py**: It defines the dataset and provides a command line interface to outputing it.
  - **function** `generate_dataset(seed)` : Generates a list of sentences, sentence types and their truth values. Sentences consist of facts about animals.
    - Parameters:
      - **seed**: *Optional[int] = None* if a seed is provided, the returned list is shuffled; the seed is used to initialize the random number generator
    - Return: *list[tuple[str, str, bool]]* the dataset – a list of sentence, sentence type, sentence truth value tuples
- **gen_train_test_split_json.py**: provides a command line interface to randomize and split the dataset into train and test subsets; outputs the result as JSON.

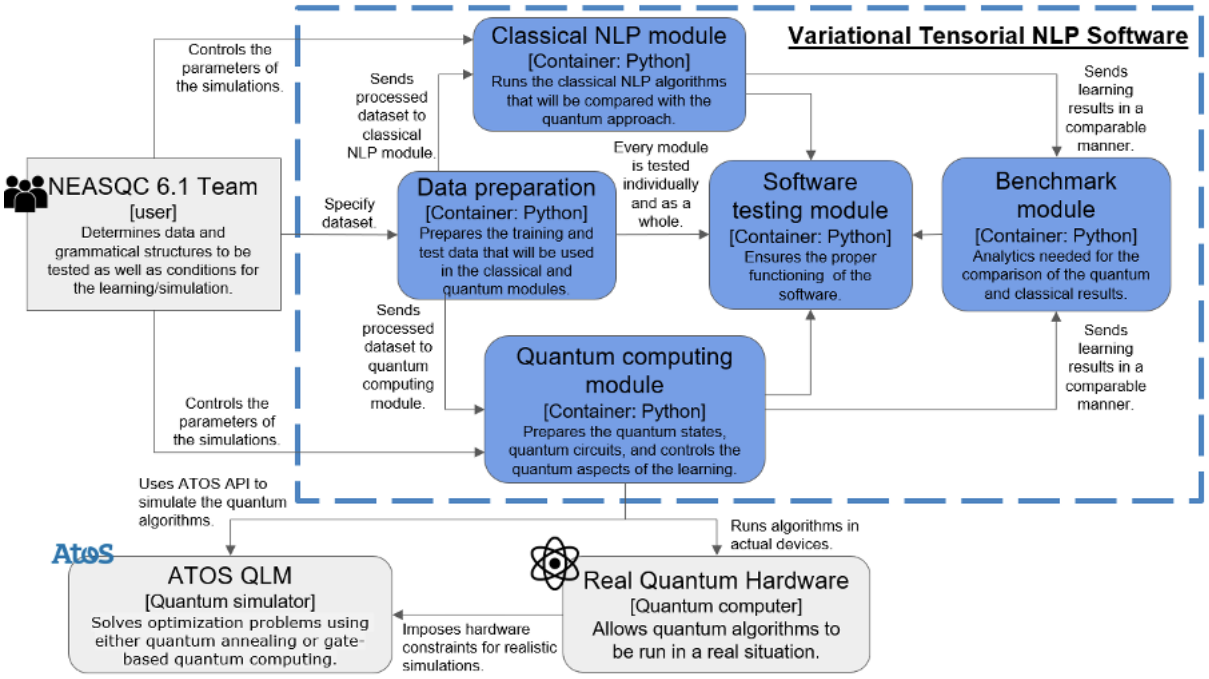



***Figure 6**: C4 architecture of the package*

### 4.1.4. Quantum Computing

The `Quantum` module, corresponding to Quantum Computing module in the diagram, has the goal of receiving the files generated by the data preparation module and processing the data to write the corpus as quantum states and train a model to solve the selected NLP task, in this case assert the veracity of a set of sentences. The full model optimization is coded under this module. The classical optimization of the circuit and the implementation of the DisCoCat model, despite not being fully quantum related, are implemented here. The following components have been developed for the quantum computing module:

- **dictionary.py**: It contains all the grammar-related information. The categories and words supported, the number of qubits used for the model, and the assignation of word to categories are here.
  - **class** `PartOfSpeech(part,cats)` : An object containing the information about the parts of speech and how to map them to a set of categories for the DisCoCat model.
    - Parameters:
      - **parts**: *list[str]* Parts of speech supported. Examples are `"NOUN"`, `"TVERB"`, `"PREP"`,...
      - **cats**: *list[str]* Categories for each part of speech. Example: `["nr", "s", "nl"]`
  - **class** `QuantumDict(qn,qs)` : An object containing the words used in the model and useful information about word properties.
  - Parameters:
    - **qn**: *int* Number of qubits per $n$ category.
    - **qs**: *int* Number of qubits per $s$ category.
    - **partsOfSpeech**: *obj:* `PartOfSpeech` The parts of speech associated with the words in the dictionary.
    - **dictionary**: *dict* A dictionary of with the words in the dataset. It can contain either `QuantumWord` objects or regular words. Word parameters and number of qubits are included in this dictionary.

    - **nlp**: *Obj: Spacy language model* The English NLP model used to parse the tokens if provided
  - **class** `QuantumWord(token,wordstring, wordtype)`
  - Parameters:
    - **token**: *Obj:* `Spacy` *token* Spacy token of the word we want to create.
    - **wordstring**: *str* String of the word we want to create. Provide instead of a `Spacy` token.
    - **wordtype**: *str* Part of speech corresponding to the word. Provide if a string is given, not a token.

- **sentence.py**: Through this file is processed all the information regarding the sentences in a dataset. Its methods are used to build the quantum circuits correctly according to sentence structure and the words involved.
  - **class** `Sentence(sentence,dataset, dictionary, label, stype)`
  - Parameters:
    - **sentence**: *str* The sentence string.
    - **dictionary**: *dict* A dictionary with the words in the sentence. These words have the relevant information for building the quantum circuits.
    - **qubitsarray**: *list* A list with the qubits needed for creating the circuit in a specific shape according to word distribution.
    - **categoriesarray**: *list* A list with the categories needed for creating the circuit in a specific shape according to word distribution.
    - **catqubits**: *list* a list with the qubits associated with each category.
    - **stype**: *int* A code to provide the sentence structure.
    - **label**: *int* The truth value of the sentence.
- **circuit.py**: It is an interface between Atos simulator backend and thee DisCoCat implementation for grammar. It takes care of building the circuits, submitting the jobs and processing the results.
  - **class** `CircuitBuilder(layers,parameterization, random)`
  - Parameters:
    - **layers**: *int* Number of parameterization layers in the circuits. Only `1` is implemented.
    - **parameterization**: *str* The shape of the Ansatz. Only 'Simple' is implemented.
    - **result**: *Obj:* `Result` A `QLM Result` object with the outcome of submitting the job to the backend.
    - **qlmprogram**: *Ob:* `Program` A `QLM Program` object with the circuit for a sentence.
    - **random**: *bool* Either we want to initialize parameters randomly or loading for a file. Only random = True is implemented.
- **optimizer.py**: It takes care of the classical optimization. `SciPy` is used under the hood to train the model, and the custom cost functions and criteria for the optimization is provided here.
  - **class** `ClassicalOptimizer(optimizer, tol maxiter)`
  - Parameters:
    - **optimizer**: *str* The name of the `SciPy` optimizer to use.
    - **tol**: *float* `tol` parameter for the `SciPy` optimizer.
    - **maxiter**: *int* The maximum number of iterations.
    - **itercost**: *list* A list that records the cost function value for each iteration.

    * **iteration**: *int* A index of the current iteration

- **loader.py**: Although not fully related with the quantum algorithms, it is needed to process the output files from the `DataPreparation` module. It creates `Pandas.DataFrame` objects so the dataset can be easily processed for the training.

# 5. Examples and Results

The modules and examples implemented in D6.3 are a first version of the targeted QNLP implementation. There is a large margin of improvement in the subsequent updates:

1. Refactoring the code to leverage the speedup of libraries such as `NumPy` or `Numba` is essential.
2. Refactoring the code for better maintainability in the future is also needed.
3. A more general implementation of pregroup grammars and category theory is recommended for better generalization, instead of providing the rules for building specific sentence structures.

For these reasons, we are not focusing on benchmarking or performing noisy simulations yet, and just give an overview of how the over algorithm and workflow is intended.

Two example workflows and their results are available in two notebooks under `misc/notebooks` directories. Here, it is considered that $q_n = q_s = 1$, as the number of parameters for the chosen Ansatz and implementation become large and the training is harder. These restriction can be changed when creating the `QuantumDict` object.

## 5.1. Quantum sentence training

Now, we discuss the results of training single sentences and datasets. These examples can be found in more detail in the Jupyter notebooks at `misc/notebooks`. We need to optimize the values of the rotations found in the Ansatz, that will be our model parameters so the truth values of sentences can be predicted running the corresponding quantum circuit. The cost function and other details are in section 3.

### 5.1.1. Single Sentence training

First, the training of a single sentence with the structure `NOUN-IVERB-PREP-NOUN` is shown. The chosen sentence is sentence is *"dog barking at cat"*, which corresponds to the structure in figure 2. For the number of qubits $q_n = q_s = 1$, which results in a total of $9$ qubits for building the sentence circuit. Figure 7 shows the evolution of the cost function where cross entropy was used.

The shape of the cost function may be different for different runs of the algorithms, as the seed for randomly generating the parameters is changing and the initial value of the cost function may vary. In any case, for one sentence convergence is fast and always successful.

### 5.1.2. Training of whole datasets

The datasets used for the training are generated using the `DataPreparation` module. The output is a `.json` file that is processed to create DataFrames. Several of them are available under the Datasets folder. This example can be seen in more detail in the notebook called `Dataset_ example.ipynb`.

`Expanded_Transitive_dataset.json` consists of sentences of type:

- `NOUN-TVERB-NOUN`
- `NOUN-IVERB`
- `NOUN-IVERB-PREP-NOUN`

A sample of the sentences shown in Figure 8.

Another dataset used in the example is available in `Expanded_Transitive_dataset.json`. For this, the vocabulary and possible sentences are slightly modified, trying to improve the performance of the algorithm making it easier to train the model. Only transitive sentence are included in this dataset. A sample of this dataset is shown in Figure 9.

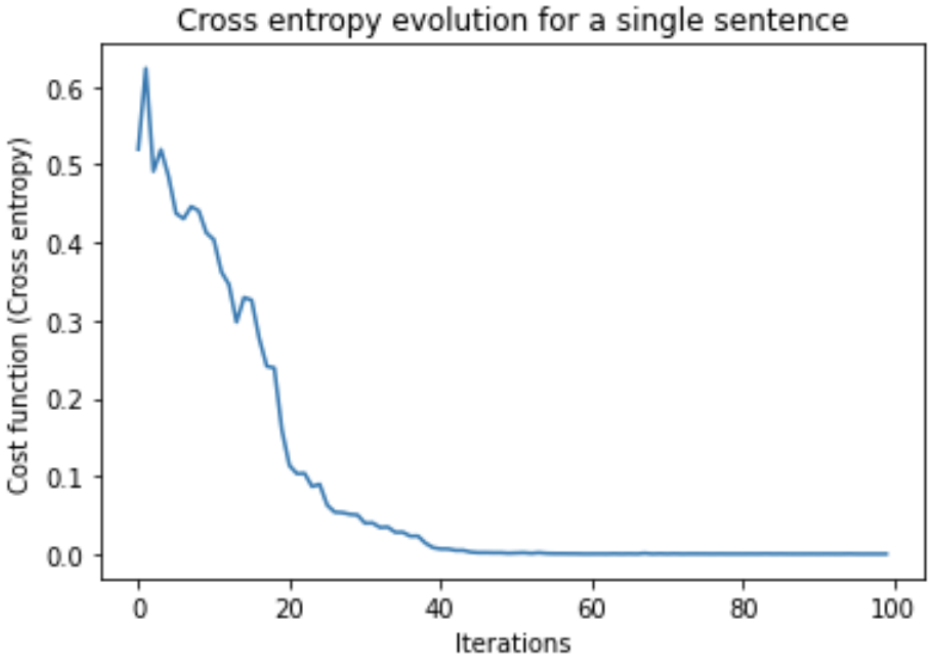



***Figure 7**: Evolution of the cost function with the iterations. At the end of the training, the sentence has a high probability of yielding a "False" result, as was expected*

| | sentence | sentence_type | truth_value | sentence_type_code |
|---|---|---|---|---|
| **0** | cat chases | NOUN-IVERB | True | 3 |
| **1** | mouse meows | NOUN-IVERB | False | 3 |
| **2** | chicken eats fox | NOUN-TVERB-NOUN | False | 0 |
| **3** | mouse flees krill | NOUN-TVERB-NOUN | False | 0 |
| **4** | dog runs | NOUN-IVERB | True | 3 |
| **5** | fish swims in water | NOUN-IVERB-PREP-NOUN | True | 1 |
| **6** | fox bites krill | NOUN-TVERB-NOUN | False | 0 |
| **7** | dog chases cat | NOUN-TVERB-NOUN | True | 0 |
| **8** | mouse eats dog | NOUN-TVERB-NOUN | False | 0 |
| **9** | cat runs after dog | NOUN-IVERB-PREP-NOUN | False | 1 |

***Figure 8**: DataFrame of the full dataset with all the sentence types supported.*

Once the model have been trained, we update the parameters and see if the sentences labeling is correct. It is found that for the training dataset most of the sentences are correctly assigned the `True` or `False` category, but we are not that successful with the test dataset.

The difference between the evolution of the cost function for each dataset shows the big impact that the chosen sentences have in the performance of the model. This suggests that it is worth further investigating how to create big consistent datasets that can be efficiently trained.

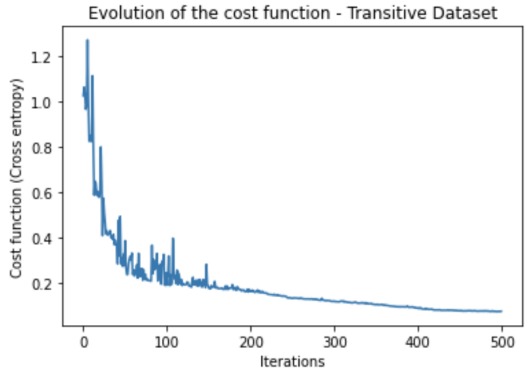
This point is illustrated in Figure 11, where is displayed how well the algorithm performed in assigning the label to the sentences in the training and test datasets, respectively.

This may be a case of overfitting due to a series of circumstances, that include:

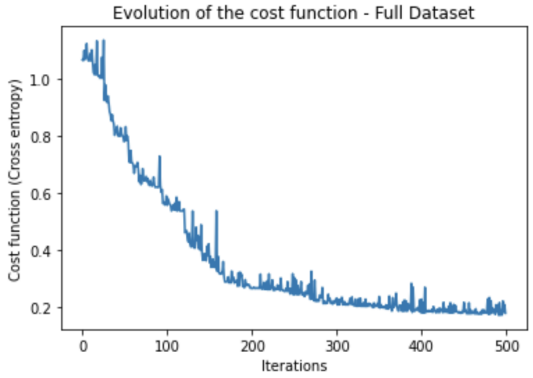
- the high dimensionality of the model parameters vector for the small number of words considered,

| | sentence | sentence_type | truth_value | sentence_type_code |
|---|---|---|---|---|
| **0** | dog eats bone | NOUN-TVERB-NOUN | True | 0 |
| **1** | mouse eats fish | NOUN-TVERB-NOUN | False | 0 |
| **2** | cat eats bone | NOUN-TVERB-NOUN | True | 0 |
| **3** | bird flees cat | NOUN-TVERB-NOUN | True | 0 |
| **4** | dog flees mouse | NOUN-TVERB-NOUN | False | 0 |

***Figure 9**: DataFrame with transitive sentences and modified vocabulary.*

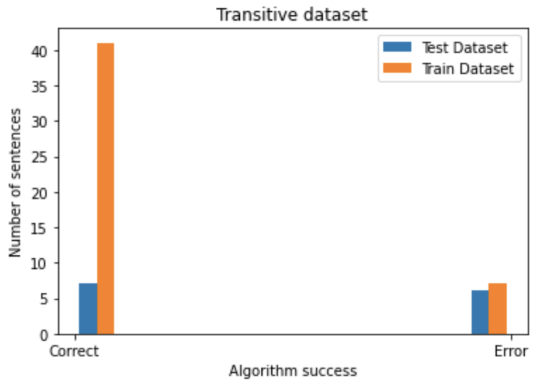



*(a) Evolution of the cost function for the transitive only dataset.*

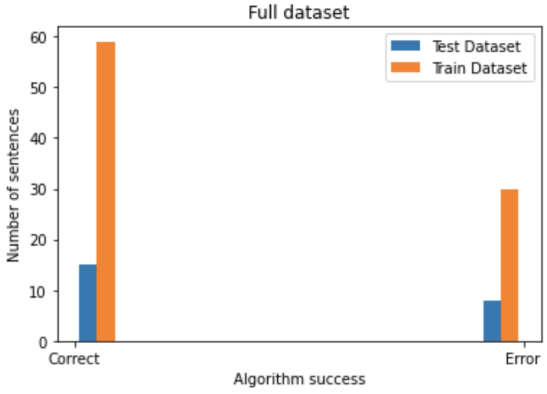



*(b) Evolution of the cost function for the full dataset*

***Figure 10**: Cost evolution for two sentences dataset*




*(a) Algorithm success for the transitive only dataset.*




*(b) Algorithm success for the full dataset.*

***Figure 11**: Algorithm success for two different sentence datasets.*

- the quality of the dataset considered,
- the chosen Ansatz, or
- the classical optimizer used and its fine tuning.

## 5.2. Classical NLP approach

In order to have a baseline to compare against, a classical approach should also be implemented.

The envisaged classical approach for solving the chosen problem consists of two parts:

1. The sentences are vectorized in word embedding representation using some pretrained word embedding model.
2. A classical classifier, such as neural network, is trained on top of these embeddings.

Word embeddings can be generated in two different forms – on word level or on sentence level. The former approach outputs a vector for each word in a sentence. The latter outputs one vector for the whole sentence.

As a first approach, a sentence vector can be generated by simply taking the average of vectors of each of the words in the sentence. This approach has the obvious drawback that the sentence embedding does not depend on the word order.

The word embeddings can be static or contextual, meaning that either the vector for a word depends only on the given word, or it takes into account a larger context (in this case – the whole sentence) when generating embedding for a word, respectively.

The approach described above is implemented in a notebook in the `misc/notebooks/classical/` directory. Two different methods of vectorizing are implemented – using BERT (Devlin et al., 2019) and fastText (Bojanowski et al., 2017) word embedding models. BERT word embeddings are contextual, but fastText word embeddings are static. Currently, BERT vectorization can output word embeddings only for the whole sentence and fastText vectorization can output only vectors for the individual words.

For the classification part, a shallow feedforward neural network and a convolutional network are implemented. The architectures of the networks are chosen fairly arbitrary, since in this stage the focus is on establishing the processing pipeline. Additionally, a non-parametrized classifier which uses the label from the sentence of the training set which is the closest to the sentence which is currently being tested (i.e., 1-nearest neighbor algorithm) is implemented.

The results for the different classical algorithms can be seen in Figure 12.

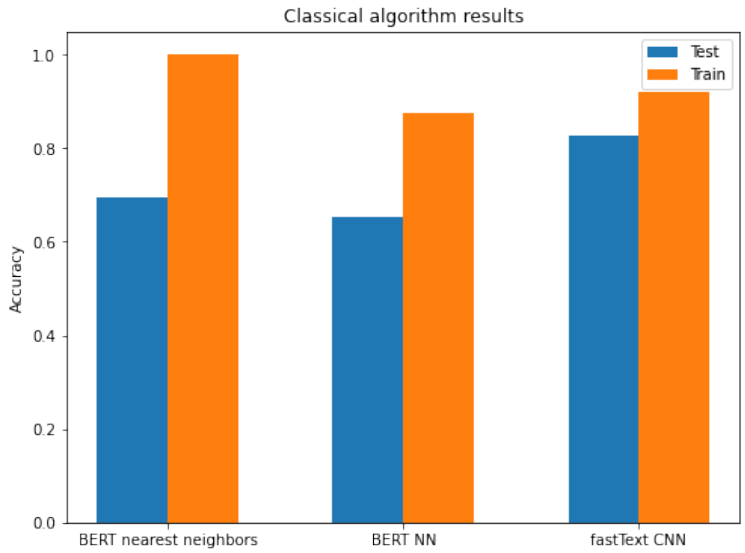



***Figure 12**: Train and test accuracy of the on the animal dataset. Results for the nearest neighbor, neural network (NN) and convolutional neural network (CNN) classifiers are shown. BERT/fastText indicates the vectorisation method used.*

For the sake of comparison, the accuracy of the quantum algorithm can be seen in the image below to contrast with the classical methods considered below.

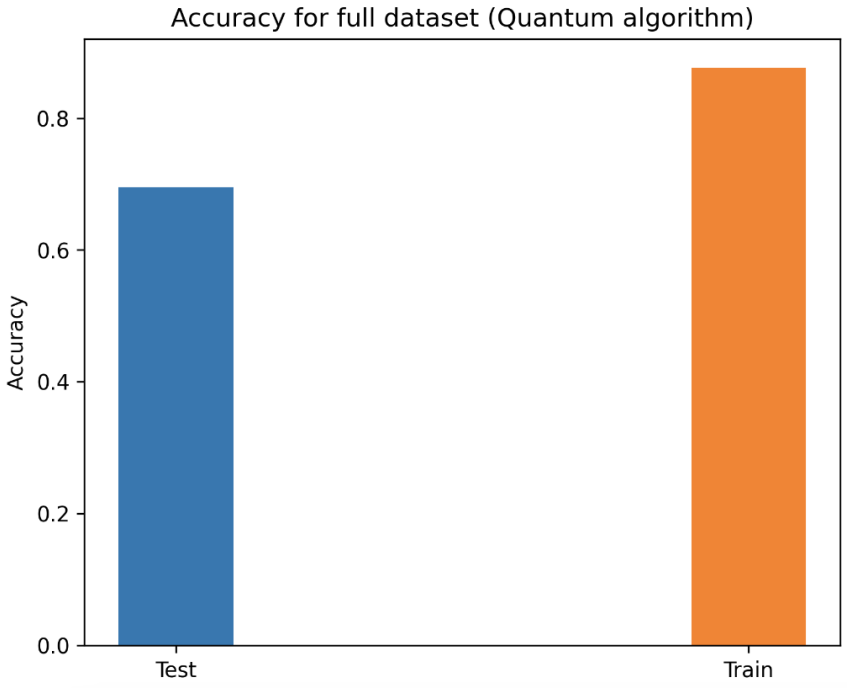



***Figure 13**: Accuracy for the train and test datasets using the quantum algortihm.*

# 6. Conclusions

While this is an early release, our first implementation of the most popular quantum NLP model, the DisCoCat model, is presented. Preliminary results are shown, that include quantum and classical approaches for solving the proposed task of true/false classification. This allowed for targeting complications that need to be solved, evaluating the tools available that we can apply specifically to this problem, and see which are the points that we can elaborate or improve for future releases.

When comparing the classical and quantum approach, we do not expect to give a quantitative answer yet. In the notebooks can be seen that the classical algorithm is delivering a better accuracy score for the test dataset, as the success of the quantum approach when labelling these instance is little. Furthermore, slight differences to the dataset used in the classical part were introduced to investigate how the number of sentences, their structure, and the vocabulary used affect the performance of the quantum approach. In this sense, although we are not contrasting numerically these results, this is a first comparison to build the classical/quantum benchmark on.

## 6.1. Limitations

Quantum natural language processing is a fairly new field with no established paradigm. This opens up the possibility for innovation, but as a small research community the field is not as active as others.

Among the limitations and issues we have found when developing the package, are:

1. Programming pregroup grammars following category theory: In the current implementation, the structure of the sentence must be provided and the qubits contractions are calculated following rules that are structure-specific rather than general for any sentence.
2. Creating datasets that are coherent, with a balanced diversity of vocabulary and sentence structures, keeping a high number of sentences but at the same time not creating ambiguity with confusing labeling.
3. Running algorithms with higher number of qubits become unfeasible with MyQLM limitations. We must move to QLMaaS and to the actual simulator device to perform faster and useful simulations.
4. The QLM does not offer a huge number of options for training quantum circuits as other quantum machine learning platforms such as PennyLane or Tensorflow Quantum do.
5. Although for the pre-alpha prototype we were looking for a functional working code, we are conscious that refactoring the code would improve the performance dramatically.

## 6.2. Further work

While we follow the ideas proposed by the DisCoCat model, different approaches are possible that may have an impact in complex classical NLP algorithms. The application of quantum computing in Transformer models or recurrent neural networks are exciting topics that we would like to explore (Sipio, 2021) (Bausch, 2020). Next work items for the current implementation and to be solved in shorter-term include:

1. Automatically detect the proper DisCoCat diagram for any sentence, which may require a deeper understanding of category theory and how to bring that knowledge to actual code.
2. Not relying only in QLM plugins or `Scipy` but creating optimization methods from scratch that give more freedom for parameter bounds, tolerance, and so forth. Simultaneous Perturbation Stochastic Approximation (Bhatnagar, 2013) is one of the targets in that sense.
3. Exploring the effect of noise, gate fidelity and the number of shots in the performance.
4. Refactoring the coding dropping unnecessary methods, loops and any possible duplicity. Also, as has been stated above, incorporating libraries such as `NumPy` or `Numba` would improve the speed dramatically.

## List of Acronyms

| Term | Definition |
|---|---|
| **NEASQC** | NExt ApplicationS of Quantum Computing |
| **QLM** | Quantum Learning Machine |
| **RNN** | Recurrent Neural Networks |
| **QNLP** | Quantum Natural Language Processing |
| **NISQ** | Noisy Intermediate-Scale Quantum |
| **MPS** | Matrix Product State |
| **MPO** | Matrix Product Operator |
| **RNN** | Recurrent Neural Network |
| **DisCoCat** | Distributionak Compositional Categorial |

***Table 1**: Acronyms and Abbreviations*

## List of Figures

## Bibliography

Atos. (2021). *Myqlm documentation*. https://myqlm.github.io/

Bai, G., Yang, Y., & Chiribella, G. (2020). Quantum compression of tensor network states. *New Journal of Physics*, *22*(4), 043015.

Bausch, J. (2020). Recurrent quantum neural networks. *arXiv preprint arXiv:2006.14619*.

Bharti, K., Cervera-Lierta, A., Kyaw, T. H., Haug, T., Alperin-Lea, S., Anand, A., Degroote, M., Heimonen, H., Kottmann, J. S., Menke, T., et al. (2021). Noisy intermediate-scale quantum (nisq) algorithms. *arXiv preprint arXiv:2101.08448*.

Bhatnagar. (2013). *Stochastic recursive algorithms for optimization. simultaneous perturbation methods*. Springer.

Biamonte, J., & Bergholm, V. (2017). Tensor networks in a nutshell. *arXiv preprint arXiv:1708.00006*.

Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, *5*, 135–146.

Cerezo, M., Arrasmith, A., Babbush, R., Benjamin, S. C., Endo, S., Fujii, K., McClean, J. R., Mitarai, K., Yuan, X., Cincio, L., et al. (2020). Variational quantum algorithms. *arXiv preprint arXiv:2012.09265*.

Coecke, B., de Felice, G., Meichanetzidis, K., & Toumi, A. (2020). Foundations for near-term quantum natural language processing. *arXiv preprint arXiv:2012.03755*.

Coecke, B., de Felice, G., Meichanetzidis, K., & Toumi, A. (2021). How to make qubits speak. *arXiv preprint arXiv:2107.06776*.

Coecke, B., Sadrzadeh, M., & Clark, S. (2010). Mathematical foundations for a compositional distributional model of meaning. *arXiv preprint arXiv:1003.4394v1*.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171–4186.

Huggins, W., Patil, P., Mitchell, B., Whaley, K. B., & Stoudenmire, E. M. (2019). Towards quantum machine learning with tensor networks. *Quantum Science and technology*, *4*(2), 024001.

Meichanetzidis, K., Toumi, A., de Felice, G., & Coecke, B. (2020). Grammar-aware question-answering on quantum computers. *arXiv preprint arXiv:2012.03756*.

Orús, R. (2014). A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, *349*, 117–158.

Sipio, R. D. (2021). *Quantum-enhanced transformer neural network*. https : / / github . com / rdisipio / qtransformer

Toumi, A., & Koziell-Pipe, A. (2021). Functorial language models. *CoRR*, *abs/2103.14411*. https://arxiv.org/abs/2103.14411

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 5998–6008.

Villalpando, A., O'Riordan, L. J., Balodis, K., & Krišlauks, R. (2020). NEASQC D6.1 QNLP design and specification. https://www.neasqc.eu/wp-content/uploads/2021/04/NEASQC_D6.1_QNLP-design-and-formal-specification.pdf