Advanced NLP Project

Implementing Rank QA

Team ALP, No. 19

- **T. H. Arjun, 2019111012**
- **Arvindh A, 2019111010**

Motivation

In IRE and NLP an Open Question Answering system tries to answer a query by extracting a set of relevant documents from a corpus and then using a Machine Comprehension Model to extract top spans from each document and then gives the best span as the answer. But till the model presented in the paper[1] we have for the project, the paradigm was considered split into two independent processes, firstly extracting the relevant documents and then secondly a NN for machine comprehension extracts the best span as the answer. The motivation of the paper is to use features extracted from the two steps and use them in the third step of re-ranking of spans to extract the best span. **As part of the project, we implemented RankQA and DrQA (Retriever & Reader) from scratch. We use the whole Wikipedia dump and train the models on the Open Question Answering task on the questions in the SQuAD dataset. We perform various experiments including an ablation study on the models.**

Architecture

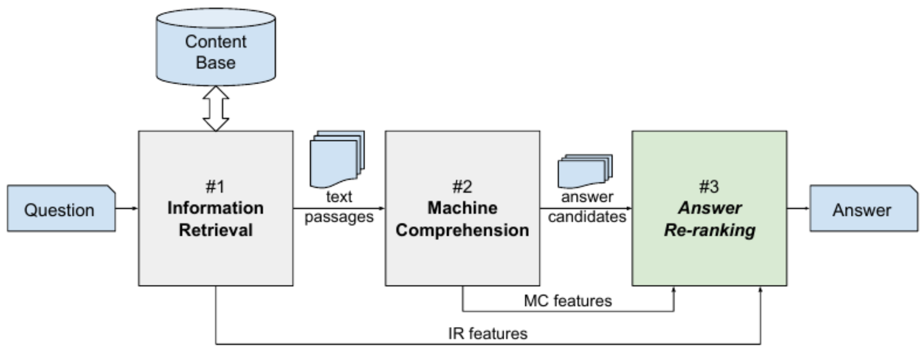


Figure 1 - Overall Architecture (credits: the paper)

The architecture proposed by the paper (Figure 1) uses three modules:

- ## Module 1: Information Retrieval

  Firstly, the query is received by this Module which it uses to extract the top n documents from the content base. The paper implements the architecture proposed by Chen.et.al(2017)[2] known as DrQA. The module functions by first calculating the TF-IDF vector for a question and for all documents. Then the documents are ranked with respect to cosine similarity and top k documents are returned.

- ## Module 2: Machine Comprehension

  Given the k passages from the previous module, this module predicts the best spans for an answer. The paper proposes the architecture of Chen. et.al(2017)[2] DrQA for this.
  Let us consider paragraph p having $p_1, p_2, p_3, p_4, \ldots p_l$, that is l tokens and question $q$containing $q_1, q_2, q_3, \ldots q_m$ with $m$tokens. The task is to predict the start and end points from the context that is the paragraph here. The inputs while training are the question, context/paragraph and the truth values.
  Firstly, the context and question are embedded differently. For this the paper [2] proposes using the features:
  - Exact match: that is if the token exactly matches in the lemma, or lowercase to one
  - Token features : Includes POS, NER and TF of context tokens
  - Aligned question embedding f_{align}

$$f_{align} = \sum_j a_{i,j} E(q_j)$$

$$a_{i,j} = \frac{\exp\left(\alpha(\mathbf{E}(p_i)) \cdot \alpha(\mathbf{E}(q_j))\right)}{\sum_{j'} \exp\left(\alpha(\mathbf{E}(p_i)) \cdot \alpha(\mathbf{E}(q_{j'}))\right)}$$

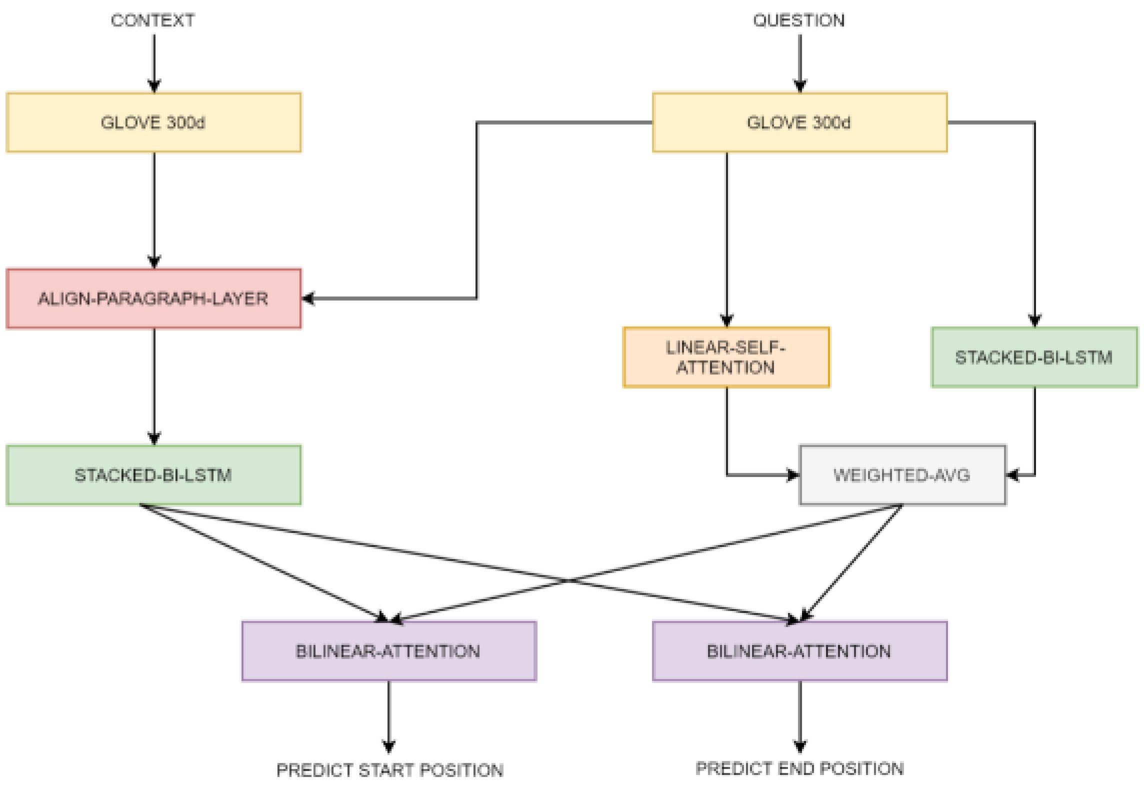


Figure 2: DrQA Architecture

Where $\alpha$is a linear layer with ReLU activation. The intuition here is that this helps the model to learn what portion of the context is more important or relevant with respect to the question. That is because here the tokens from question and context are multiplied and hence capturing their similarities. Quoting the paper, "these features add soft alignments between similar but non-identical words (e.g., car and vehicle)"[2]. So now the encoding is made by concatenating the features: glove embedding, f_{align}, exact match, POS, NER, TF. This is now passed through 3 layers of stacked Bi-LSTMs. Then the architecture involves concatenating the final output of all layers. Now for encoding the question, the glove embedding is taken for each token. Which is then passed through a Linear Self Attention, which is just that it is multiplied by a weight matrix and softmax is taken on top of it followed by a weighted average.

$$b_j = \frac{\exp(\mathbf{w} \cdot \mathbf{q}_j)}{\sum_{j'} \exp(\mathbf{w} \cdot \mathbf{q}_{j'})} \qquad q = \sum_j b_j q_j$$

The first equation above shows a weighted matrix multiplication followed by a softmax. The second equation is the weighted average for all tokens. The intuition is to learn which tokens are important to prediction.

Now for prediction, the paper trains two classifiers, $P_{start}(i)$ and $P_{end}(i)$ which scores the probability of i being start or end respectively. $P_{start}(i) \times P_{end}(j)$ will be the probability of that part of paragraph $(i \rightarrow j)$ being the answer.

$$\begin{aligned} P_{start}(i) &\propto \exp\left(\mathbf{p}_i \mathbf{W}_s \mathbf{q}\right) \\ P_{end}(i) &\propto \exp\left(\mathbf{p}_i \mathbf{W}_e \mathbf{q}\right) \end{aligned}$$

- ## Module 3: Rank QA (Answer Re-Ranking)

This is the main contribution of the paper which is answer re-ranking and integrating into the full QA pipeline. First, features from modules 1 and 2 are extracted. Features from module 1 include

- Document-question similarity
- Paragraph-question similarity
- Paragraph length
- Question length
- Question start indicator variable (e. g., “what”, “who”, “when”, etc.), etc.

while features from module 2 include

- Original score and rank of the answer candidate
- The rank of the answer candidate
- POS tags
- NER tags

Next, identical answers are aggregated and the feature vector is appended with aggregation features such as the number of occurrences, mean, max, and sum of similarity scores and ranking scores, rank, etc. Then the answers suggested by module 2 are re-ranked based on the above-generated features. The Re-Ranking is done by two feed-forward NN with a Relu Activation in the hidden layer as shown below.

$$f(x_i) = \mathrm{ReLU}(x_i A^T + b_1)\, B^T + b_2$$

The paper also proposes using a custom loss which is a combination of ranking loss from Burges et. al(2005)[3] and an additional penalty in terms of Regularisation which are given below.

$$\mathcal{L}_{\text{rank}}(x_i, x_j) = \Big[y_i - \sigma\left(f(x_i) - f(x_j)\right) \Big]^2$$

$$\mathcal{L}_{\text{reg}} = \|A\|_1 + \|B\|_1 + \|b_1\|_1 + \|b_2\|_1$$

The intuition behind the loss function is really simple. This module tries to assign a new score for each of the answer candidates. It subsamples a pair of data points (a, b) in such a way that $y_a \neq y_b$. If $y_b = 1$, then the original pair is reversed to create (b, a). $f(a)$, $f(b)$ are the new scores calculated. When the above pair is fed through the architecture proposed, the following cases can occur

- $y_a = 1$ & $y_b = 0$

  The loss function becomes,

  $$L = [1 - \sigma(f(a) - f(b))]^2$$

  This forces $f(a) \gg f(b)$ ultimately assigning a an higher score than b.
- $y_a = 0$ & $y_b = 1$

  The loss function becomes,

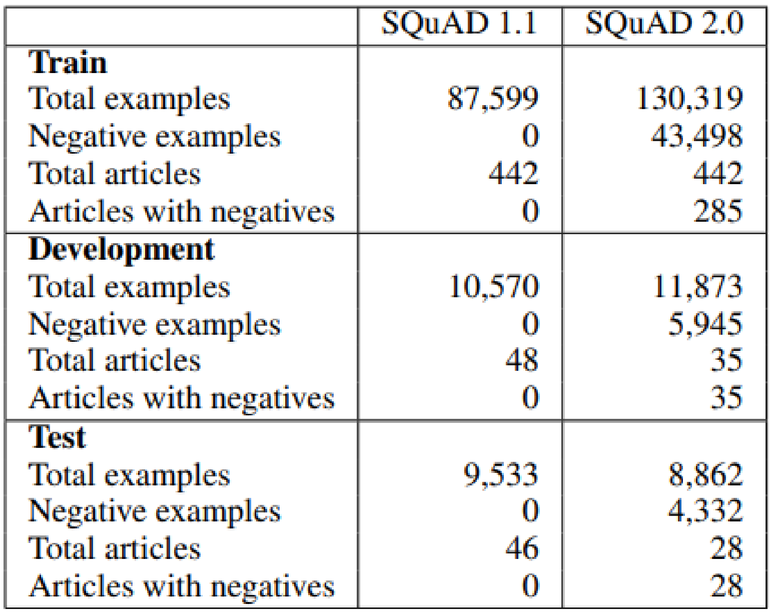
  $$L = [1 - \sigma(f(b) - f(a))]^2$$

  This forces $f(b) \gg f(a)$ ultimately assigning b an higher score than a.

So, this is how the model learns its objective - To assign a higher score to questions which have correct answers.

Dataset

We use the SQUADopen dataset in our implementation.



	SQuAD 1.1	SQuAD 2.0
Train		
Total examples	87,599	130,319
Negative examples	0	43,498
Total articles	442	442
Articles with negatives	0	285
Development		
Total examples	10,570	11,873
Negative examples	0	5,945
Total articles	48	35
Articles with negatives	0	35
Test		
Total examples	9,533	8,862
Negative examples	0	4,332
Total articles	46	28
Articles with negatives	0	28

Figure 3

Experiments

Module 1

We implemented Module 1 using the Gensim library for the TFIDF Vectorization. We indexed the whole wikipedia dump of 19GB. We release the dumb and the searchable index as part of this project. We faced a bit of computation issues in doing so but we overcame them using multiprocessing. We also released an SQL dataset of the wikipedia dump.

Module 2

We implemented DRQA Reader (Module 2) in pytorch. We used the same parameters and model architecture as discussed in the DRQA [2] paper. We trained the model with a decaying learning rate and stopped when it reached a threshold as mentioned in the paper. The training loss, validation loss and learning rate across epochs for DrQA is as shown in Figure 4.

Module 3

We ran the SQuAD Dataset on the entire module 1 and module 2 and extracted the features as mentioned in the paper. Due to computational issues we restricted ourselves to 4 candidate answers instead of the 40 described in the original paper. We see that this restriction might have affected the results compared to the original paper. We trained our models with early stopping and used the loss function as described in the paper with the same parameters. The training loss, validation loss over epochs for all the experiments and models are shown in figures below.

Results

In our experiments we see an accuracy increase of about 5.5% from DrQA to RankQA output. This shows that the model is really good for re-ranking and the intuition of connecting the retriever and reader outputs with a third model is very useful. We also perform an ablation study by removing some features in RankQA and we observe the following:

1. The RankQA model suggested in the paper performs the best
2. The features such NER and POS are very crucial for the prediction as we see a very sudden drop and no learning when removing them.
3. The futures such as document similarity scores and others related to module 1 are not so important to the prediction as the performance does not get hit when removing them.

4. We also see that aggregated features provide a bit of help to the prediction as the performance reduces when removing them.
5. We then plot the accuracies of all the models including DrQA and RankQA across each question category. We see no specific trend other than the fact that there is an increase due to reranking.

DrQA

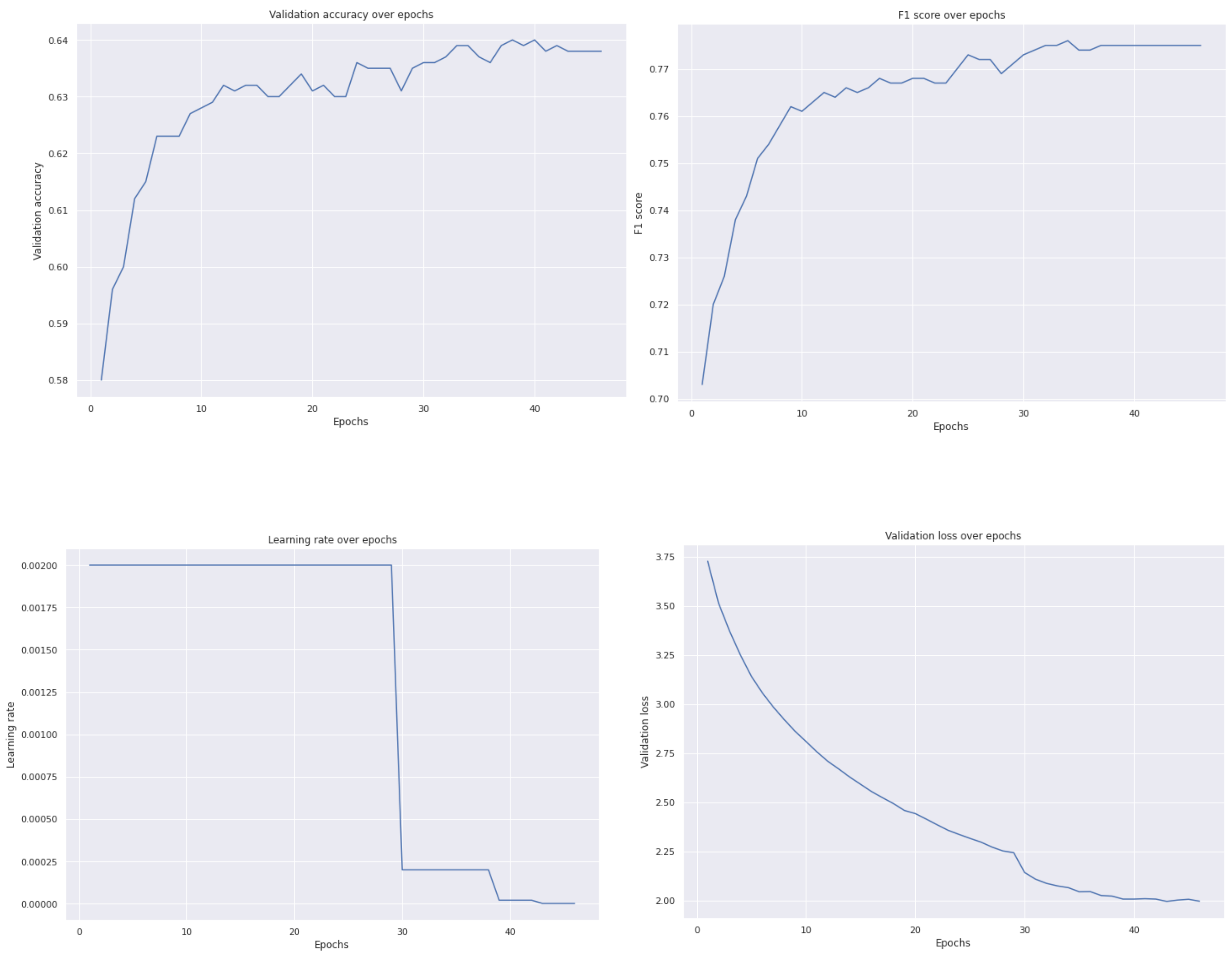
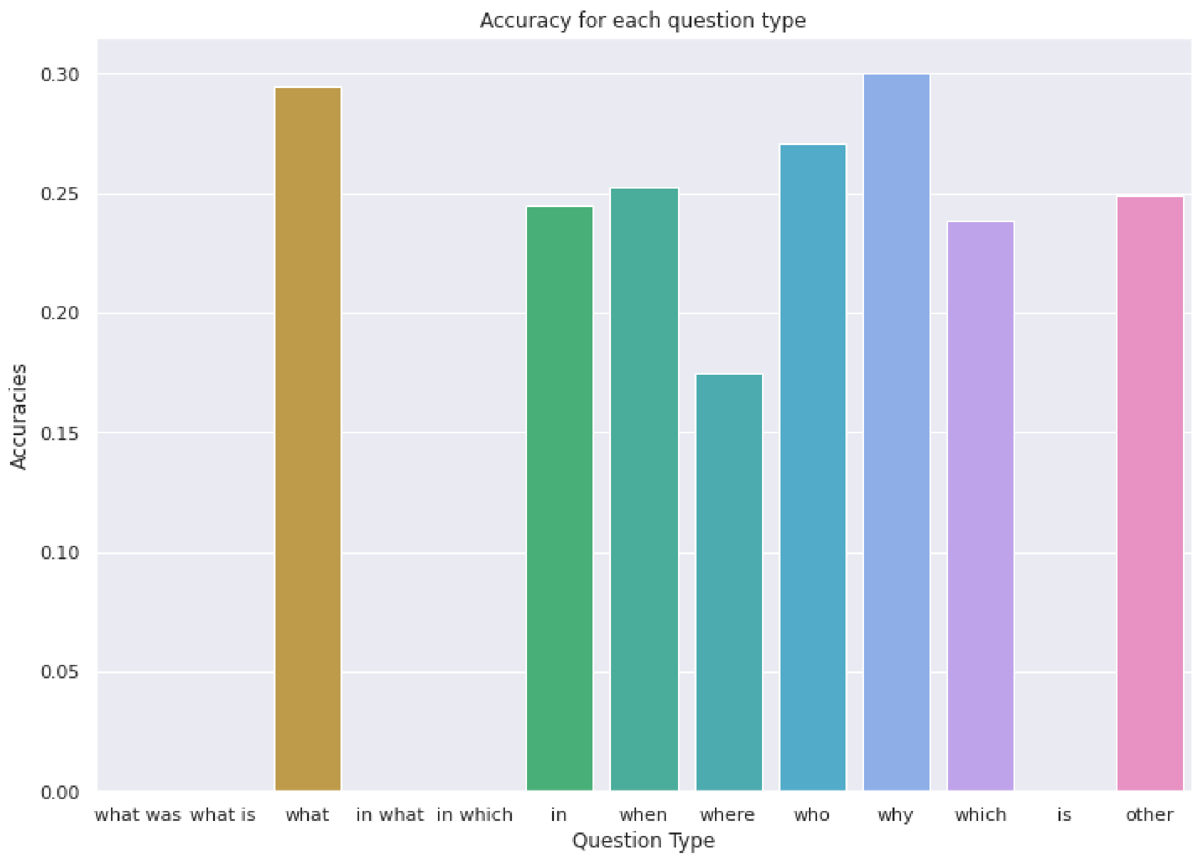
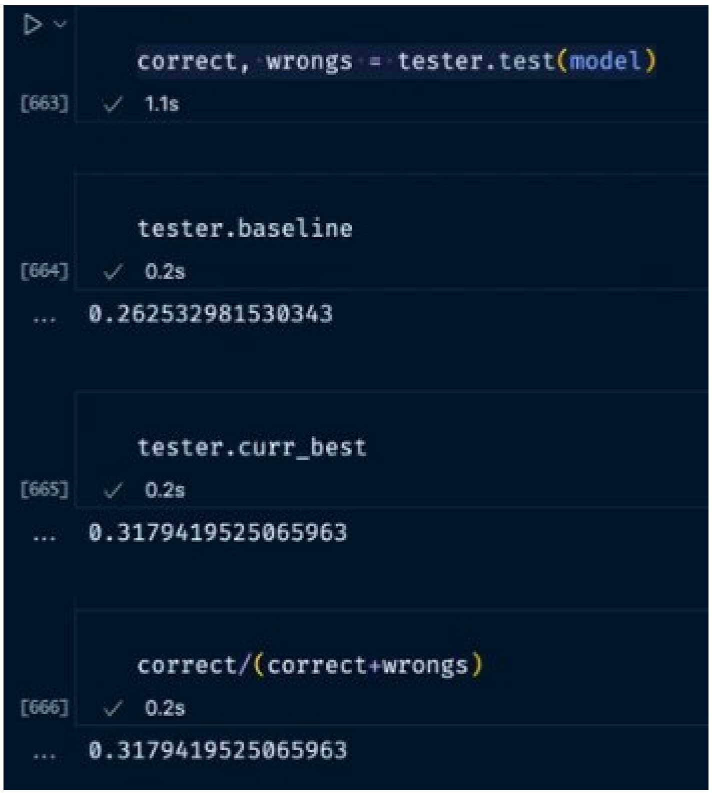


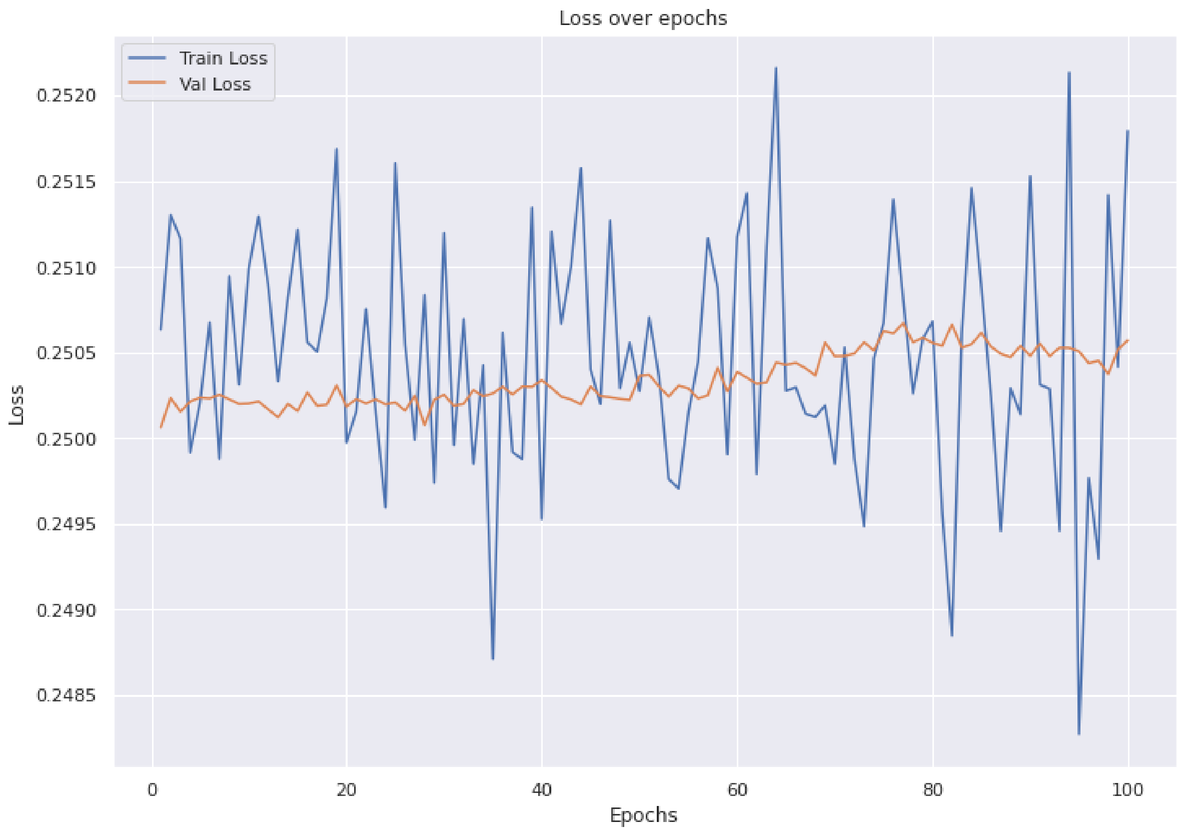
Figure 4 : Losses for DRQA

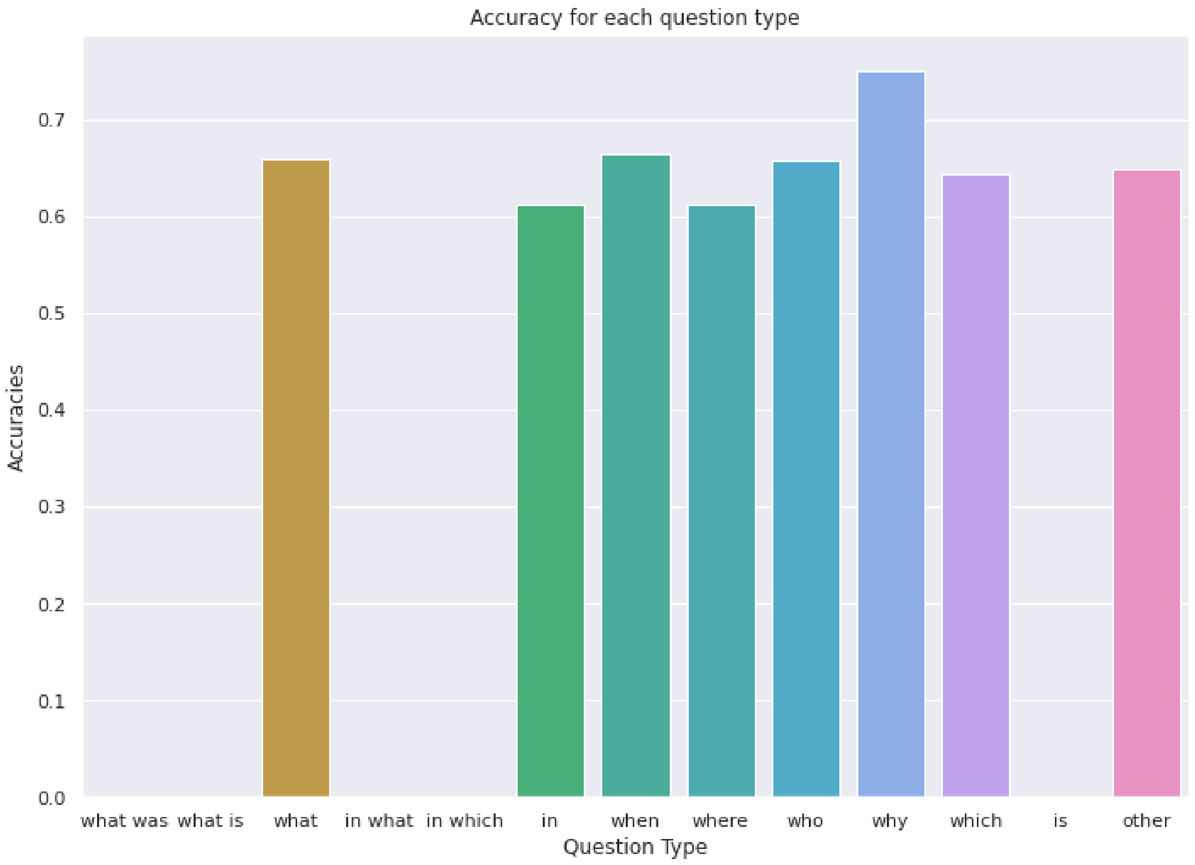


RankQA

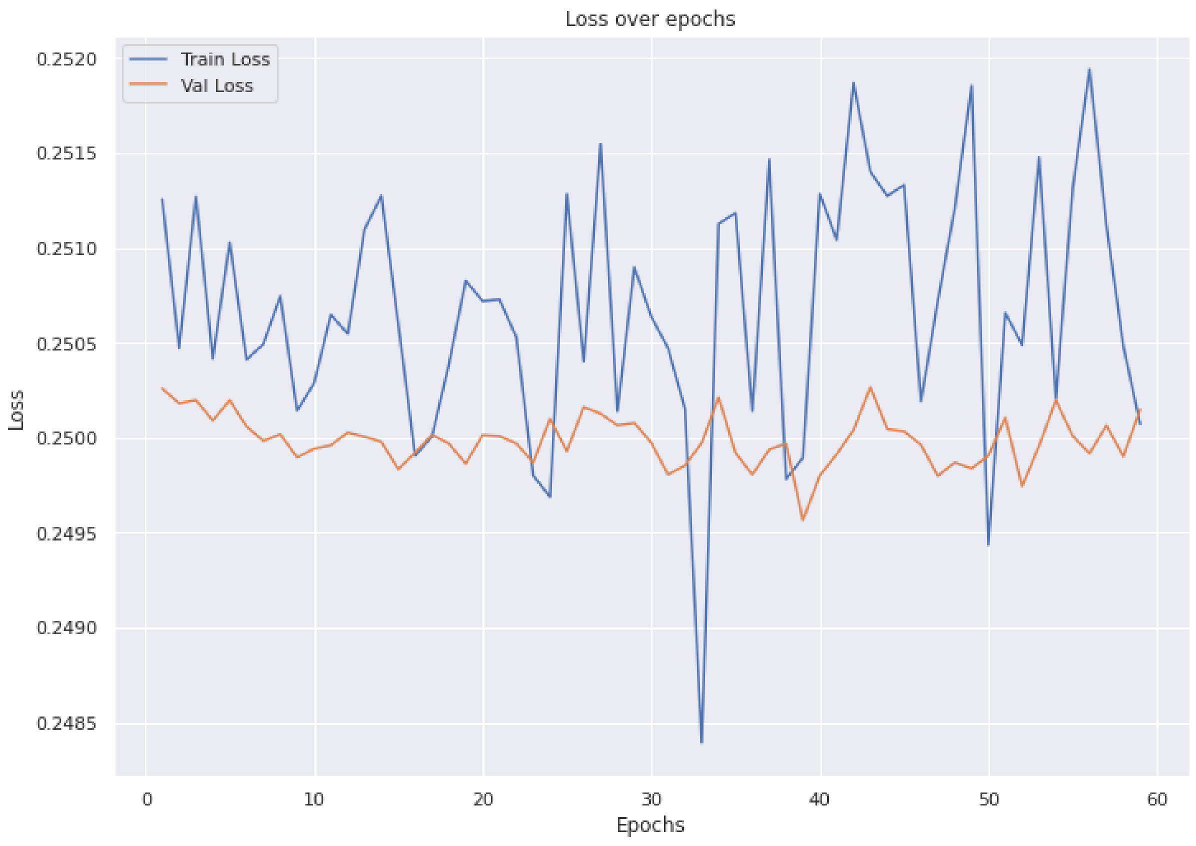

```
correct, wrongs = tester.test(model)
[663]  ✓ 1.1s

tester.baseline
[664]  ✓ 0.2s
...  0.262532981530343

tester.curr_best
[665]  ✓ 0.2s
...  0.3179419525065963

correct/(correct+wrongs)
[666]  ✓ 0.2s
...  0.3179419525065963
```

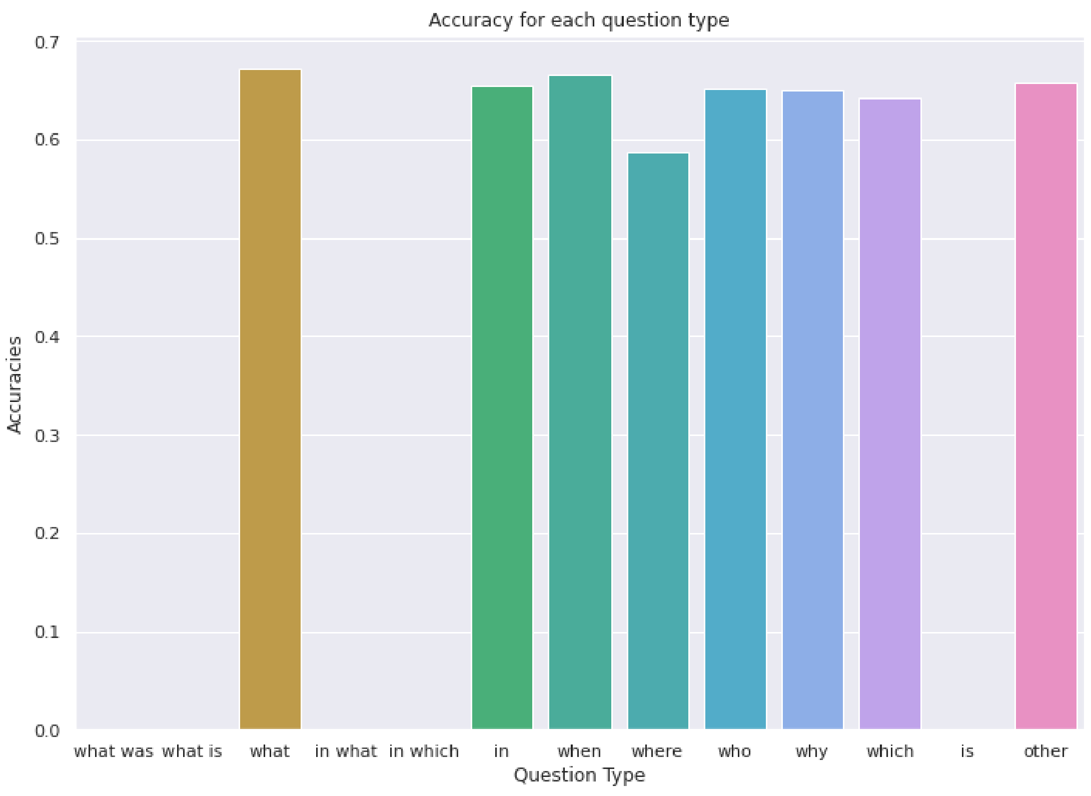
tester.baseline denotes the baseline accuracy (DrQA)
test.curr_best denotes the RankQA accuracy


Loss over epochs
Train Loss
Val Loss
0.2520
0.2515
0.2510
0.2505
0.2500
0.2495
0.2490
0.2485
Loss
0
20
40
60
80
100
Epochs

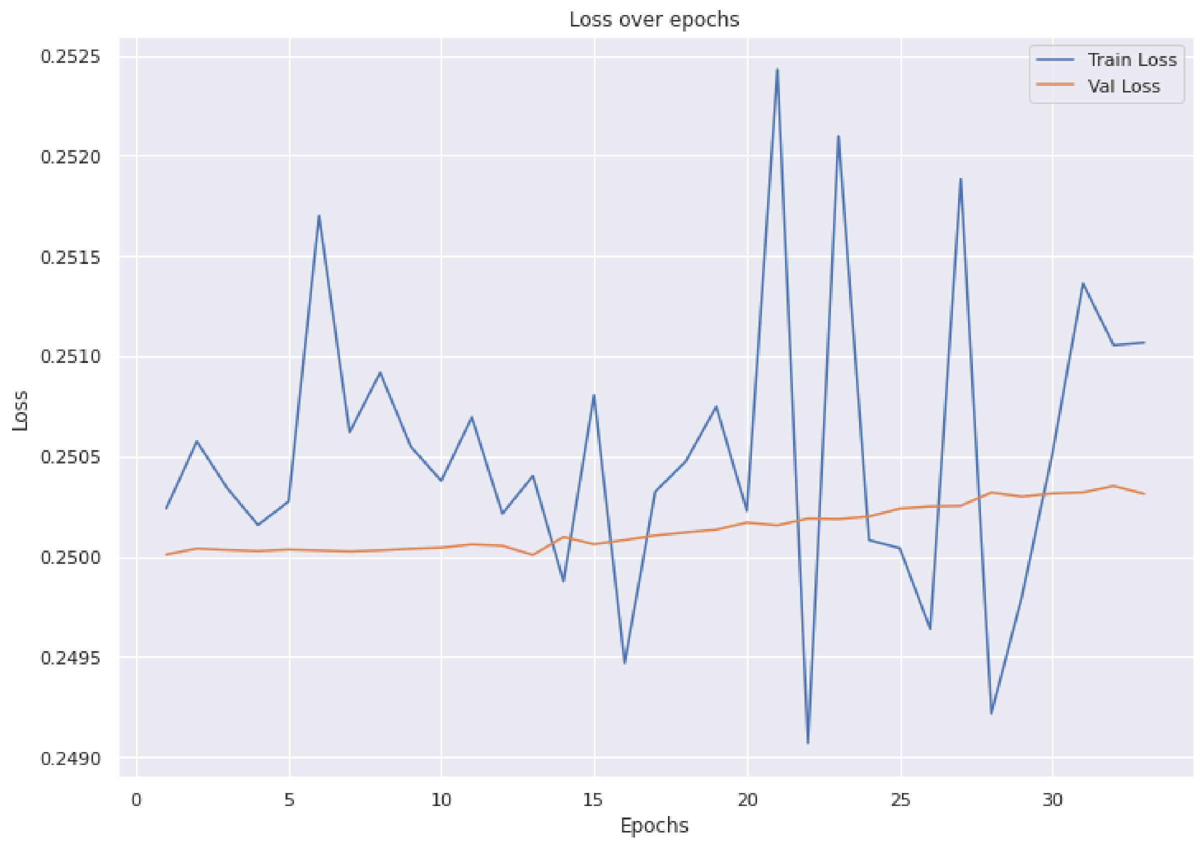

Accuracy for each question type
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Accuracies
what was
what is
what
in what
in which
in
when
where
who
why
which
is
other
Question Type

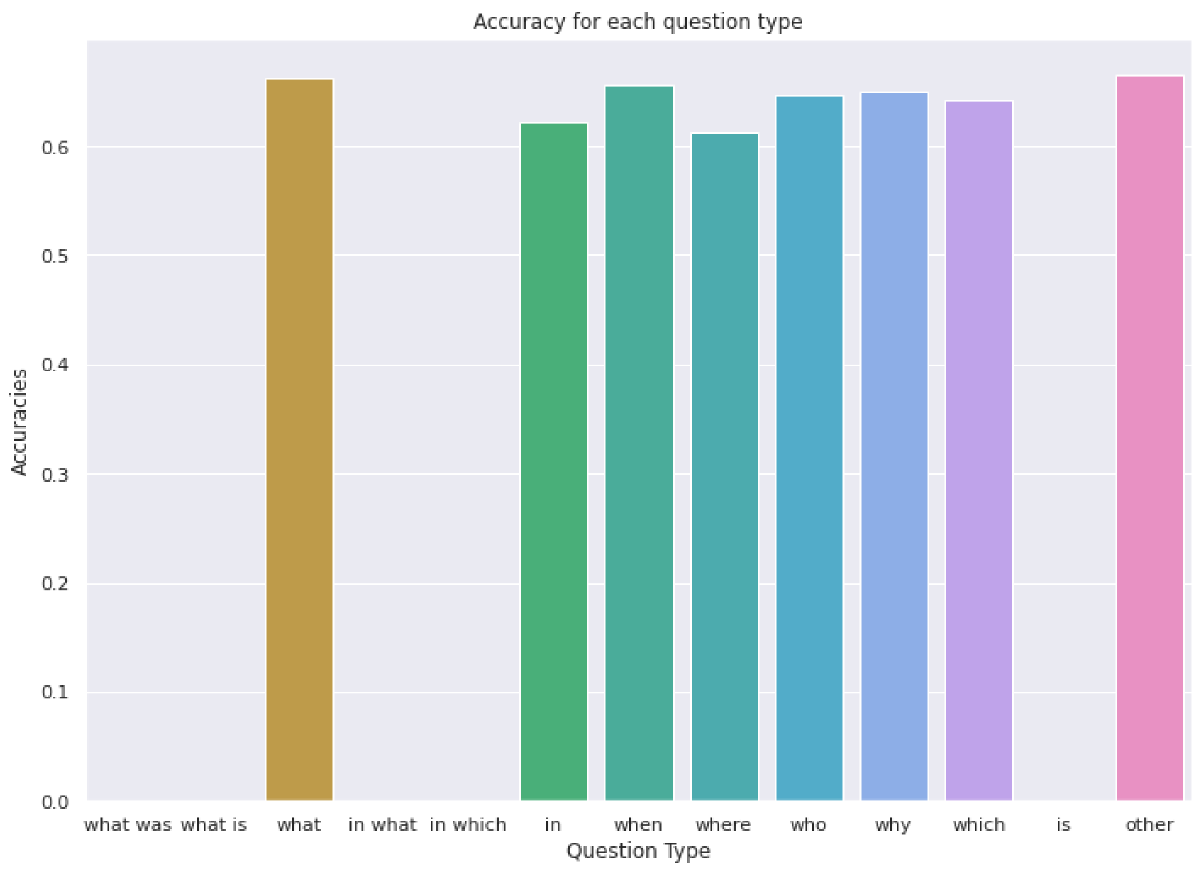
RankQA without Module 1 features



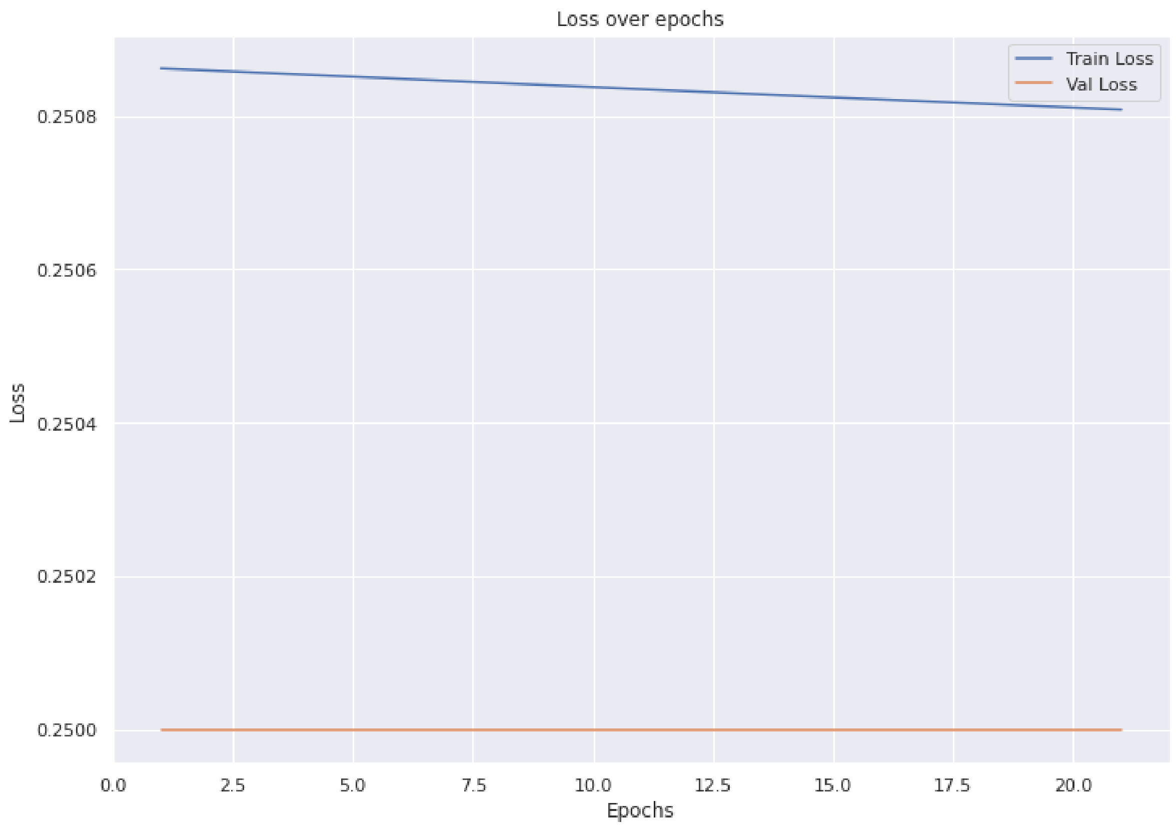


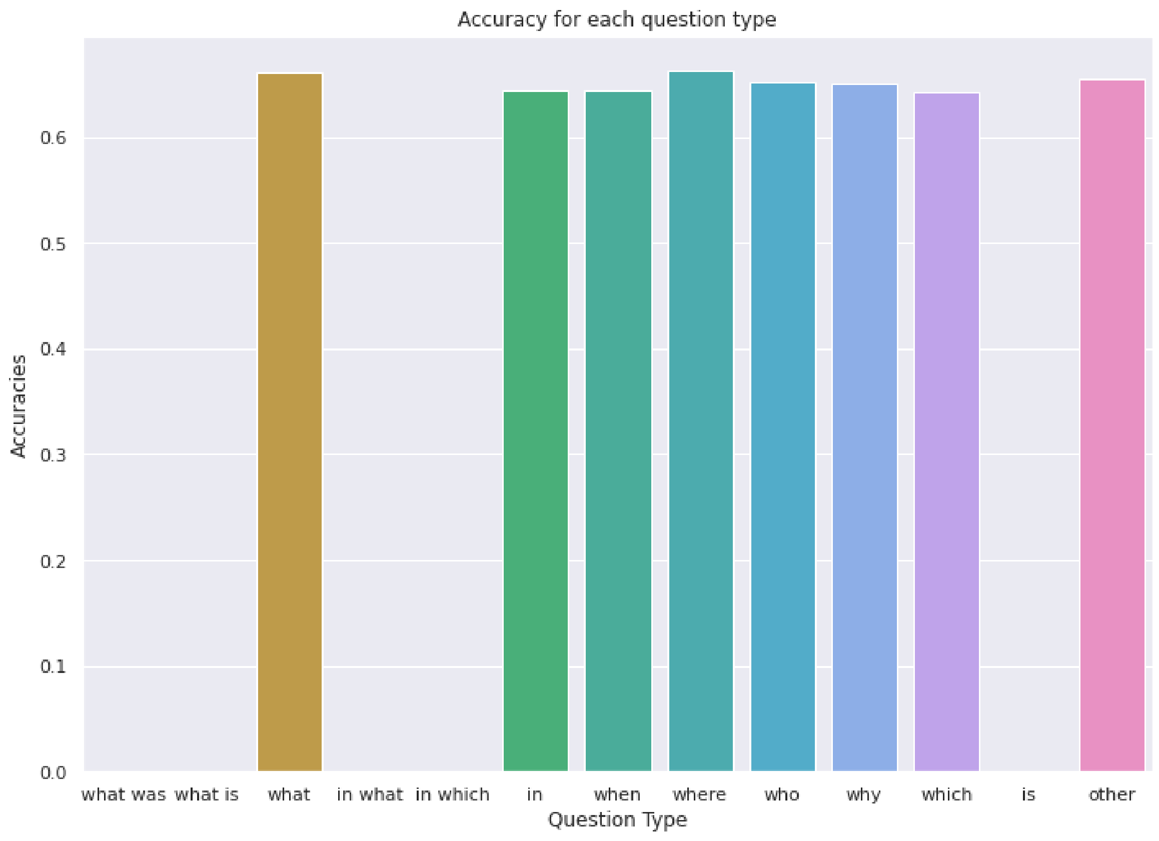
RankQA without aggregation





RankQA without POS, NER





Model	**Accuracy**
DrQA	26.25
RankQA	**31.79**
RankQA without Module 1 Features	31.53
RankQA without aggregation	31.46
RankQA without POS, NER	26.26

Table showing various experiments and their results.

Project Deliverables Delivered

- Implemented the DrQA paper (Module 1 and 2 from scratch)
- Got its output for Wikipedia Dump including the features for the dataset
- Implemented the Paper RankQA (Module 3) from scratch
- Ran the dataset over module 3 and tested performance
- Conducted an Ablation study
- Released all the data files required for rerunning the experiments

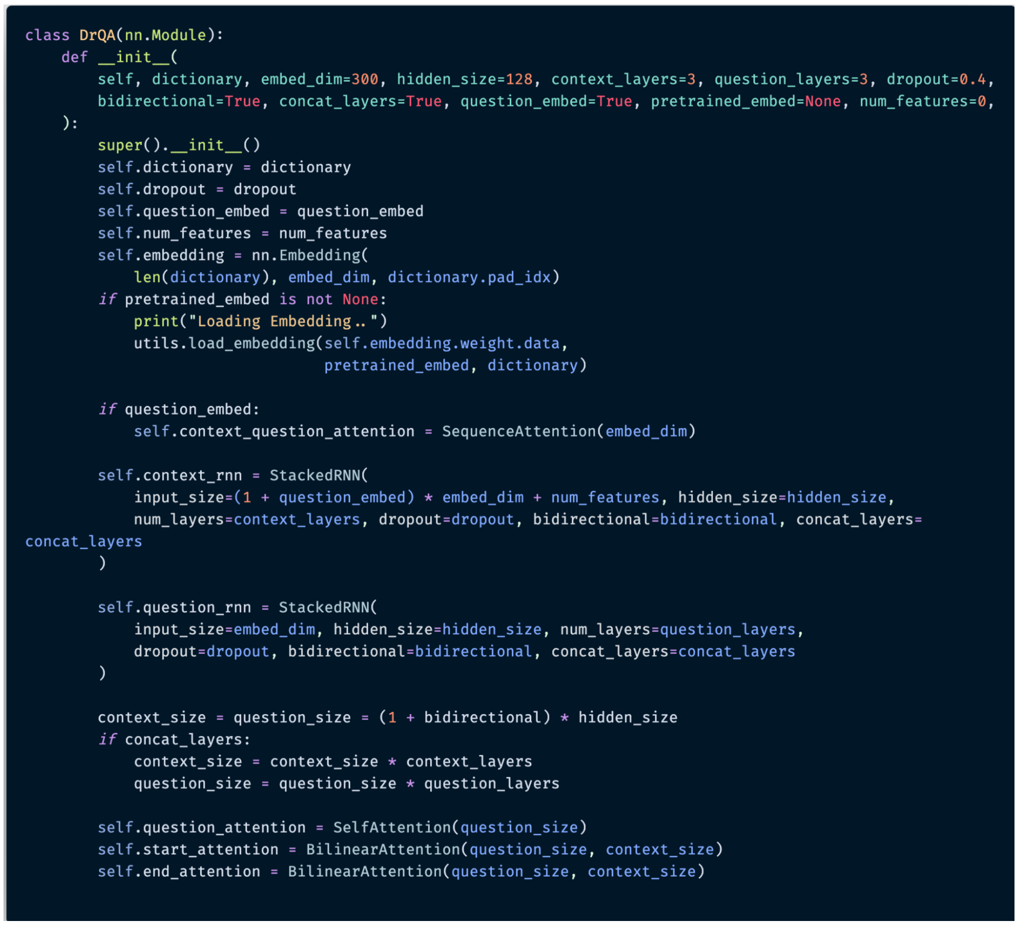
Limitations

- The training was a very compute heavy process, we had to parse over 16GB of dump and vectorize it
- Due to computational limitations, we were forced to adapt a small $k$value which hugely affected the final accuracy and findings

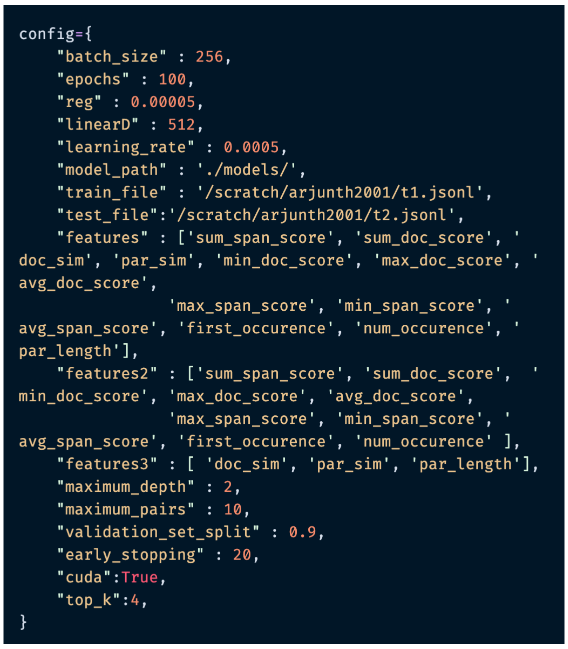
References

1. Kratzwald, B., Eigenmann, A., & Feuerriegel, S. (2019). Rankqa: Neural question answering with answer re-ranking. *arXiv preprint arXiv:1906.03008*.
2. Chen, D., Fisch, A., Weston, J., & Bordes, A. (2017). Reading Wikipedia to answer open-domain questions. *arXiv preprint arXiv:1704.00051*.
3. Burges, Chris, et al. "Learning to rank using gradient descent." Proceedings of the 22nd international conference on Machine learning. ACM, 2005

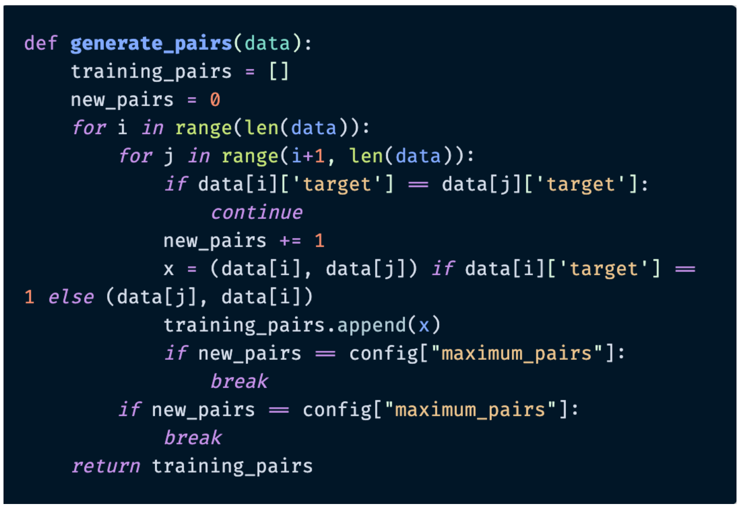
Appendix



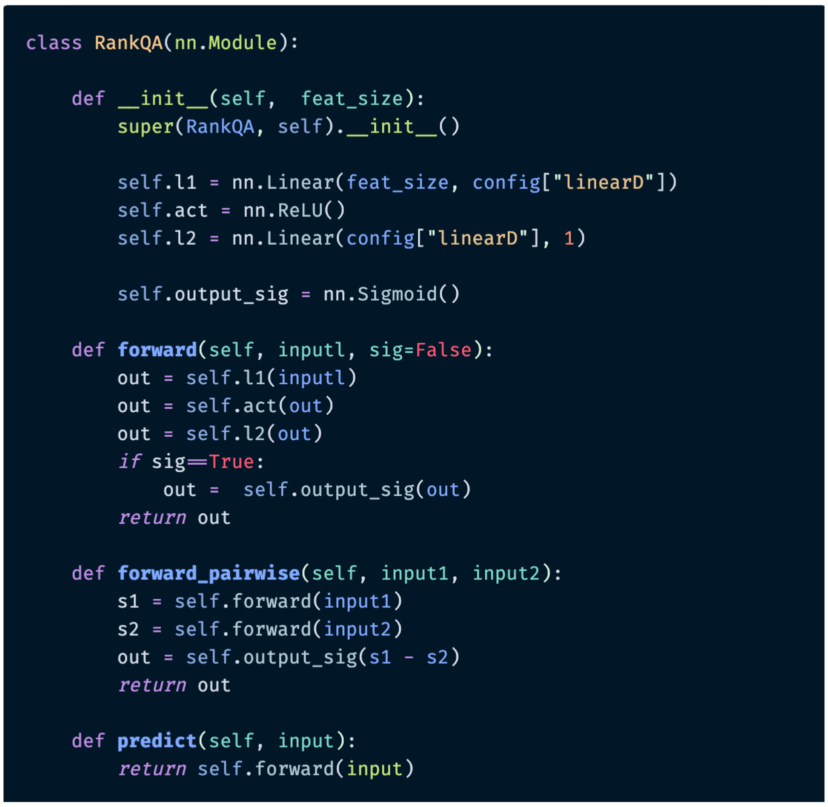
```
class DrQA(nn.Module):
    def __init__(
        self, dictionary, embed_dim=300, hidden_size=128, context_layers=3, question_layers=3, dropout=0.4,
        bidirectional=True, concat_layers=True, question_embed=True, pretrained_embed=None, num_features=0,
    ):
        super().__init__()
        self.dictionary = dictionary
        self.dropout = dropout
        self.question_embed = question_embed
        self.num_features = num_features
        self.embedding = nn.Embedding(
            len(dictionary), embed_dim, dictionary.pad_idx)
        if pretrained_embed is not None:
            print("Loading Embedding..")
            utils.load_embedding(self.embedding.weight.data,
                                 pretrained_embed, dictionary)

        if question_embed:
            self.context_question_attention = SequenceAttention(embed_dim)

        self.context_rnn = StackedRNN(
            input_size=(1 + question_embed) * embed_dim + num_features, hidden_size=hidden_size,
            num_layers=context_layers, dropout=dropout, bidirectional=bidirectional, concat_layers=
concat_layers
        )

        self.question_rnn = StackedRNN(
            input_size=embed_dim, hidden_size=hidden_size, num_layers=question_layers,
            dropout=dropout, bidirectional=bidirectional, concat_layers=concat_layers
        )

        context_size = question_size = (1 + bidirectional) * hidden_size
        if concat_layers:
            context_size = context_size * context_layers
            question_size = question_size * question_layers

        self.question_attention = SelfAttention(question_size)
        self.start_attention = BilinearAttention(question_size, context_size)
        self.end_attention = BilinearAttention(question_size, context_size)
```



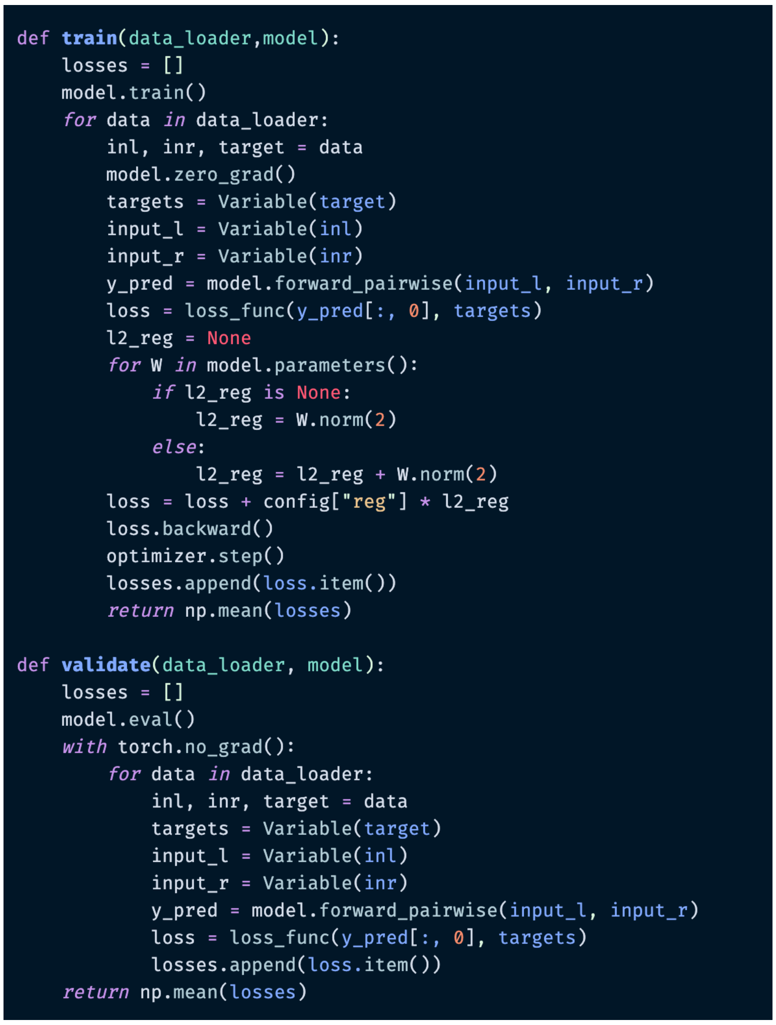
```
config={
    "batch_size" : 256,
    "epochs" : 100,
    "reg" : 0.00005,
    "linearD" : 512,
    "learning_rate" : 0.0005,
    "model_path" : './models/',
    "train_file" : '/scratch/arjunth2001/t1.jsonl',
    "test_file":'/scratch/arjunth2001/t2.jsonl',
    "features" : ['sum_span_score', 'sum_doc_score', '
doc_sim', 'par_sim', 'min_doc_score', 'max_doc_score', '
avg_doc_score',
                'max_span_score', 'min_span_score', '
avg_span_score', 'first_occurence', 'num_occurence', '
par_length'],
    "features2" : ['sum_span_score', 'sum_doc_score',  '
min_doc_score', 'max_doc_score', 'avg_doc_score',
                'max_span_score', 'min_span_score', '
avg_span_score', 'first_occurence', 'num_occurence' ],
    "features3" : [ 'doc_sim', 'par_sim', 'par_length'],
    "maximum_depth" : 2,
    "maximum_pairs" : 10,
    "validation_set_split" : 0.9,
    "early_stopping" : 20,
    "cuda":True,
    "top_k":4,
}
```



```
def generate_pairs(data):
    training_pairs = []
    new_pairs = 0
    for i in range(len(data)):
        for j in range(i+1, len(data)):
            if data[i]['target'] == data[j]['target']:
                continue
            new_pairs += 1
            x = (data[i], data[j]) if data[i]['target'] ==
1 else (data[j], data[i])
            training_pairs.append(x)
            if new_pairs == config["maximum_pairs"]:
                break
        if new_pairs == config["maximum_pairs"]:
            break
    return training_pairs
```



```python
class RankQA(nn.Module):

    def __init__(self,  feat_size):
        super(RankQA, self).__init__()

        self.l1 = nn.Linear(feat_size, config["linearD"])
        self.act = nn.ReLU()
        self.l2 = nn.Linear(config["linearD"], 1)

        self.output_sig = nn.Sigmoid()

    def forward(self, inputl, sig=False):
        out = self.l1(inputl)
        out = self.act(out)
        out = self.l2(out)
        if sig==True:
            out =  self.output_sig(out)
        return out

    def forward_pairwise(self, input1, input2):
        s1  = self.forward(input1)
        s2  = self.forward(input2)
        out = self.output_sig(s1 - s2)
        return out

    def predict(self, input):
        return self.forward(input)
```



```python
def train(data_loader,model):
    losses = []
    model.train()
    for data in data_loader:
        inl, inr, target = data
        model.zero_grad()
        targets = Variable(target)
        input_l = Variable(inl)
        input_r = Variable(inr)
        y_pred = model.forward_pairwise(input_l, input_r)
        loss = loss_func(y_pred[:, 0], targets)
        l2_reg = None
        for W in model.parameters():
            if l2_reg is None:
                l2_reg = W.norm(2)
            else:
                l2_reg = l2_reg + W.norm(2)
        loss = loss + config["reg"] * l2_reg
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
        return np.mean(losses)

def validate(data_loader, model):
    losses = []
    model.eval()
    with torch.no_grad():
        for data in data_loader:
            inl, inr, target = data
            targets = Variable(target)
            input_l = Variable(inl)
            input_r = Variable(inr)
            y_pred = model.forward_pairwise(input_l, input_r)
            loss = loss_func(y_pred[:, 0], targets)
            losses.append(loss.item())
    return np.mean(losses)
```