DECEMBER 8, 2020

University *of*
New Haven

QUESTION ANSWER GENERATION

Shreya Gopal Sundari
Venu Korada
Prof. Muhammad Aminul Islam

Contents

1. Introduction

Question answering (QA) is a computer science discipline within the fields of information retrieval and natural language processing (NLP), which is concerned with building systems that automatically answer questions posed by humans in a natural language. A question answering implementation, usually a computer program, may construct its answers by querying a structured database of knowledge or information, usually a knowledge base. More commonly, question answering systems can pull answers from an unstructured collection of natural language documents.

Some examples of natural language document collections used for question answering systems include: a local collection of reference texts, internal organization documents and web pages, compiled newswire reports, a set of Wikipedia pages, a subset of World Wide Web pages etc. Question answering research attempts to deal with a wide range of question types including: fact, list, definition, How, Why, hypothetical, semantically constrained, and cross-lingual questions.

Closed-domain question answering deals with questions under a specific domain (for example, medicine or automotive maintenance), and can exploit domain-specific knowledge frequently formalized in ontologies. Alternatively, closed-domain might refer to a situation where only a limited type of questions are accepted, such as questions asking for descriptive rather than procedural information. Question answering systems in the context of machine reading applications have also been constructed in the medical domain, for instance related to Alzheimer's disease.

Open-domain question answering deals with questions about nearly anything and can only rely on general ontologies and world knowledge. On the other hand, these systems usually have much more data available from which to extract the answer. **Multimodal** question answering uses multiple modalities of user input to answer questions, such as text and images.

2. Dataset Details

The dataset for this work is the Stanford Question Answering Dataset (SQuAD), a new reading comprehension dataset consisting of 100,000+ questions posed by crowdworkers on a set of Wikipedia articles, where the answer to each question is a segment of text from the corresponding reading passage.

The dataset is to understand the types of reasoning required to answer the questions, leaning heavily on dependency and constituency trees. To address the need for a large and high-quality reading comprehension dataset, the Question Answering Dataset v1.1 (SQuAD) is freely available at https://stanford-qa.com.

In contrast to prior datasets, SQuAD does not provide a list of answer choices for each question. Rather, systems must select the answer from all possible spans in the passage, thus needing to cope with a fairly large number of candidates.

To assess the difficulty of SQuAD, a strong logistic regression model is bulit, which achieves an F1 score of 51.0%, a significant improvement over a simple baseline (20%). However, human performance (86.8%) is much higher, indicating that the dataset presents a good challenge problem for future research.

Number of Instances: 87599

Features:

- title: Name of the University
- context: Context of the question
- question: Asked Question
- answer: Given Answer
- answer_start: Answer start
- answer_end: Answer End

3. Importing Libraries & Loading Data

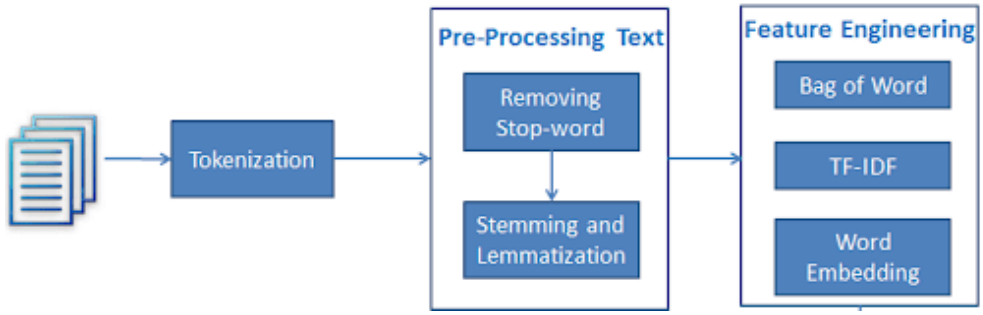
Initially, all the basic necessary libraries like numpy, pandas, torch, torchtext, torchtext.data etc., are imported into Jupyter Notebook. When using the Google Colab environment, check the version of torchtext. By default, the version of torchtext in the Colab is 0.3.0. So, to access all the functionality of torchtext, upgrade the version of it to 0.4.0 or greater. If any other libraries are required in the future, they can be imported accordingly.

The SQuAD data is shown below:

	title	context	question	answer	answer_start	answer_end
0	University_of_Notre_Dame	Architecturally, the school has a Catholic cha...	To whom did the Virgin Mary allegedly appear i...	Saint Bernadette Soubirous	515	541
1	University_of_Notre_Dame	Architecturally, the school has a Catholic cha...	What is in front of the Notre Dame Main Building?	a copper statue of Christ	188	213
2	University_of_Notre_Dame	Architecturally, the school has a Catholic cha...	The Basilica of the Sacred heart at Notre Dame...	the Main Building	279	296
3	University_of_Notre_Dame	Architecturally, the school has a Catholic cha...	What is the Grotto at Notre Dame?	a Marian place of prayer and reflection	381	420
4	University_of_Notre_Dame	Architecturally, the school has a Catholic cha...	What sits on top of the Main Building at Notre...	a golden statue of the Virgin Mary	92	126

SQuAD data

4. Text Processing



Text Processing Steps

A process of transforming text into something an algorithm can digest is text processing. This includes:

- tokenizing the data
- removing the punctuation
- removing the stopwords

- stemming
- lemmatization

As of now, we are only going to tokenize the data and work without removing the punctuation or stop words and apply any other text processing methods. Tokenization is a way of separating a piece of text into smaller units called tokens. Here, tokens can be either words, characters, or subwords.

For tokenization, two classes are defined one is a simple tokenizer, SimpleTokenizer and other is Spacy tokenizer, SpacyTokenizer. The code for them is shown below:

```
class SimpleTokenizer(Tokenizer):
    ALPHA_NUM = r'[\p{L}\p{N}\p{M}]+'
    NON_WS = r'[^\p{Z}\p{C}]'

    def __init__(self, **kwargs):
        """
        Args:
            annotators: None or empty set (only tokenizes).
        """
        self._regexp = regex.compile(
            '(%s)|(%s)' % (self.ALPHA_NUM, self.NON_WS),
            flags=regex.IGNORECASE + regex.UNICODE + regex.MULTILINE
        )
        if len(kwargs.get('annotators', {})) > 0:
            logger.warning('%s only tokenizes! Skipping annotators: %s' %
                           (type(self).__name__, kwargs.get('annotators')))
        self.annotators = set()

    def tokenize(self, text):
        data = []
        matches = [m for m in self._regexp.finditer(text)]
        for i in range(len(matches)):
            # Get text
            token = matches[i].group()

            # Get whitespace
            span = matches[i].span()
            start_ws = span[0]
            if i + 1 < len(matches):
                end_ws = matches[i + 1].span()[0]
            else:
                end_ws = span[1]

            # Format data
            data.append((
                token,
                text[start_ws: end_ws],
                span,
            ))
        return Tokens(data, self.annotators)
```

Simple Tokenizer

```
class SpacyTokenizer(Tokenizer):

    def __init__(self, **kwargs):
        """
        Args:
            annotators: set that can include pos, lemma, and ner.
            model: spaCy model to use (either path, or keyword like 'en').
        """
        model = kwargs.get('model', 'en')
        self.annotators = copy.deepcopy(kwargs.get('annotators', set()))
        nlp_kwargs = {'parser': False}
        if not any([p in self.annotators for p in ['lemma', 'pos', 'ner']]):
            nlp_kwargs['tagger'] = False
        if 'ner' not in self.annotators:
            nlp_kwargs['entity'] = False
        self.nlp = spacy.load(model, **nlp_kwargs)

    def tokenize(self, text):
        # We don't treat new lines as tokens.
        clean_text = text.replace('\n', ' ')
        tokens = self.nlp.tokenizer(clean_text)
        if any([p in self.annotators for p in ['lemma', 'pos', 'ner']]):
            self.nlp.tagger(tokens)
        if 'ner' in self.annotators:
            self.nlp.entity(tokens)

        data = []
        for i in range(len(tokens)):
            # Get whitespace
            start_ws = tokens[i].idx
            if i + 1 < len(tokens):
                end_ws = tokens[i + 1].idx
            else:
                end_ws = tokens[i].idx + len(tokens[i].text)

            data.append((
                tokens[i].text,
                text[start_ws: end_ws],
                (tokens[i].idx, tokens[i].idx + len(tokens[i].text)),
                tokens[i].tag_,
                tokens[i].lemma_,
                tokens[i].ent_type_,
            ))

        # Set special option for non-entity tag: '' vs 'O' in spaCy
        return Tokens(data, self.annotators, opts={'non_ent': ''})
```

Spacy Tokenizer

The other text processing methods are coded under a class for the further usage. They are slice (to return the list of tokens), untokenized (returns the original text with whitespace reinserted), lemmas (returns a list of the lemmatized text of each token), ngrams (returns a list of all ngrams from length 1 to n).

Next, we are going to dive into the model building and training.

5. Dense Passage Retrieval (DPR)

Open-domain question answering relies on efficient passage retrieval to select candidate contexts. Although reducing open-domain QA to machine reading is a very reasonable strategy, a huge performance degradation is often observed in practice, indicating the needs of improving retrieval. The retrieval can be practically implemented using dense representations alone, where embeddings are learned from a small number of questions and passages by a simple dual-encoder framework.

Retrieval in open-domain QA is usually implemented using TF-IDF or BM25, which matches keywords efficiently with an inverted index and can be seen as representing the question and context in high dimensional, sparse vectors (with weighting). Conversely, the dense, latent semantic encoding is complementary to sparse representations by design.

Dense encodings are also **learnable** by adjusting the embedding functions, which provides additional flexibility to have a task-specific representation. With special in-memory data structures and indexing schemes, retrieval can be done efficiently.

However, it is generally believed that learning a good dense vector representation needs a large number of labeled pairs of question and contexts. By leveraging the now standard BERT pretrained model and a dual-encoder architecture, we can develop the right training scheme using a relatively small number of question and passage pairs.

Given a collection of M text passages, the goal of dense passage retriever (DPR) is to index all the passages in a low-dimensional and continuous space, such that it can retrieve efficiently the top k passages relevant to the input question for the reader at run-time. Dense passage retriever (DPR) uses a dense encoder EP (·) which maps any text passage to a d dimensional real-valued vectors and builds an index for all the M passages that we will use for retrieval.

At run-time, DPR applies a different encoder EQ(·) that maps the input question to a d-dimensional vector, and retrieves k passages of which vectors are the closest to the question vector. We define the similarity between the question and the passage using the dot product of their vectors:

$$\text{sim}(q, p) = E_Q(q)^\intercal E_P(p).$$

Although in principle the question and passage encoders can be implemented by any neural networks, in this work we use two independent BERT networks. Hugging Face Bert encoder class, HFBertEncoder and BertTensorizer classes are defined for the BERT networks.

Following functions generate dense embeddings to the text:

```
def gen_ctx_vectors(ctx_rows: List[Tuple[object, str, str]], model: nn.Module, tensorizer: Tensorizer,
                    insert_title: bool = True) -> List[Tuple[object, np.array]]:
    n = len(ctx_rows)
    bsz = args.batch_size
    total = 0
    results = []
    for j, batch_start in enumerate(range(0, n, bsz)):

        batch_token_tensors = [tensorizer.text_to_tensor(ctx[1], title=ctx[2] if insert_title else None) for ctx in
                               ctx_rows[batch_start:batch_start + bsz]]

        ctx_ids_batch = move_to_device(torch.stack(batch_token_tensors, dim=0),args.device)
        ctx_seg_batch = move_to_device(torch.zeros_like(ctx_ids_batch),args.device)
        ctx_attn_mask = move_to_device(tensorizer.get_attn_mask(ctx_ids_batch),args.device)
        with torch.no_grad():
            _, out, _ = model(ctx_ids_batch, ctx_seg_batch, ctx_attn_mask)
        out = out.cpu()

        ctx_ids = [r[0] for r in ctx_rows[batch_start:batch_start + bsz]]

        assert len(ctx_ids) == out.size(0)

        total += len(ctx_ids)

        results.extend([
            (ctx_ids[i], out[i].view(-1).numpy())
            for i in range(out.size(0))
        ])

        if total % 10 == 0:
            logger.info('Encoded passages %d', total)

    return results
```

Dense Embeddings Generation Function

Here, Encoder model wrappers are based based on HuggingFace Transformer code. A BiEncoder training pipeline component is coded which can be used to initiate or resume training and validate the trained model using either binary classification's negative log likelihood(NLL) loss or average rank of the question's gold passages across dataset provided pools of negative passages.

Before going to the BiEncoder class, the following functions are defined:

```
def dot_product_scores(q_vectors: T, ctx_vectors: T) -> T:
    """
    calculates q->ctx scores for every row in ctx_vector
    :param q_vector:
    :param ctx_vector:
    :return:
    """
    # q_vector: n1 x D, ctx_vectors: n2 x D, result n1 x n2
    r = torch.matmul(q_vectors, torch.transpose(ctx_vectors, 0, 1))
    return r
```

Dot Product Function

```
def cosine_scores(q_vector: T, ctx_vectors: T):
    # q_vector: n1 x D, ctx_vectors: n2 x D, result n1 x n2
    return F.cosine_similarity(q_vector, ctx_vectors, dim=1)
```

Cosine Similarity Function

The BiEncoder training class has the following methods:

```
def run_train(self, ):
    args = self.args
    upsample_rates = None
    if args.train_files_upsample_rates is not None:
        upsample_rates = eval(args.train_files_upsample_rates)

    train_iterator = self.get_data_iterator(args.train_file, args.batch_size,
                                            shuffle=True,
                                            shuffle_seed=args.seed, offset=self.start_batch,
                                            upsample_rates=upsample_rates)

    logger.info("  Total iterations per epoch=%d", train_iterator.max_iterations)
    updates_per_epoch = train_iterator.max_iterations // args.gradient_accumulation_steps
    total_updates = max(updates_per_epoch * (args.num_train_epochs - self.start_epoch - 1), 0) + \
                    (train_iterator.max_iterations - self.start_batch) // args.gradient_accumulation_steps
    logger.info(" Total updates=%d", total_updates)
    warmup_steps = args.warmup_steps
    scheduler = get_schedule_linear(self.optimizer, warmup_steps, total_updates)

    if self.scheduler_state:
        logger.info("Loading scheduler state %s", self.scheduler_state)
        scheduler.load_state_dict(self.scheduler_state)

    eval_step = math.ceil(updates_per_epoch / args.eval_per_epoch)
    logger.info("  Eval step = %d", eval_step)
    logger.info("***** Training *****")

    for epoch in range(self.start_epoch, int(args.num_train_epochs)):
        logger.info("***** Epoch %d *****", epoch)
        self._train_epoch(scheduler, epoch, eval_step, train_iterator)

    if args.local_rank in [-1, 0]:
        logger.info('Training finished. Best validation checkpoint %s', self.best_cp_name)
```

BiEncoder Training Method

```
def validate_and_save(self, epoch: int, iteration: int, scheduler):
    args = self.args
    # for distributed mode, save checkpoint for only one process
    save_cp = args.local_rank in [-1, 0]

    if epoch == args.val_av_rank_start_epoch:
        self.best_validation_result = None

    if epoch >= args.val_av_rank_start_epoch:
        validation_loss = self.validate_average_rank()
    else:
        validation_loss = self.validate_nll()

    if save_cp:
        cp_name = self._save_checkpoint(scheduler, epoch, iteration)
        logger.info('Saved checkpoint to %s', cp_name)

        if validation_loss < (self.best_validation_result or validation_loss + 1):
            self.best_validation_result = validation_loss
            self.best_cp_name = cp_name
            logger.info('New Best validation checkpoint %s', cp_name)
```

BiEncoder Training Loss & Save Method

```
def validate_nll(self) -> float:
    logger.info('NLL validation ...')
    args = self.args
    self.biencoder.eval()
    data_iterator = self.get_data_iterator(args.dev_file, args.dev_batch_size, shuffle=False)

    total_loss = 0.0
    start_time = time.time()
    total_correct_predictions = 0
    num_hard_negatives = args.hard_negatives
    num_other_negatives = args.other_negatives
    log_result_step = args.log_batch_step
    batches = 0
    for i, samples_batch in enumerate(data_iterator.iterate_data()):
        biencoder_input = BiEncoder.create_biencoder_input(samples_batch, self.tensorizer,
                                                           True,
                                                           num_hard_negatives, num_other_negatives, shuffle=False)

        loss, correct_cnt = _do_biencoder_fwd_pass(self.biencoder, biencoder_input, self.tensorizer, args)
        total_loss += loss.item()
        total_correct_predictions += correct_cnt
        batches += 1
        if (i + 1) % log_result_step == 0:
            logger.info('Eval step: %d , used_time=%f sec., loss=%f ', i, time.time() - start_time, loss.item())

    total_loss = total_loss / batches
    total_samples = batches * args.dev_batch_size * self.distributed_factor
    correct_ratio = float(total_correct_predictions / total_samples)
    logger.info('NLL Validation: loss = %f. correct prediction ratio  %d/%d ~  %f', total_loss,
                total_correct_predictions,
                total_samples,
                correct_ratio
                )
    return total_loss
```

BiEncoder Training Negative Log Likelihood Loss Method

A class is defined to do passage retrieving over the provided index and question encoder called DenseRetriver. Few of the function of this calss are as follows:

```
    def generate_question_vectors(self, questions: List[str]) -> T:
        n = len(questions)
        bsz = self.batch_size
        query_vectors = []

        self.question_encoder.eval()

        with torch.no_grad():
            for j, batch_start in enumerate(range(0, n, bsz)):

                batch_token_tensors = [self.tensorizer.text_to_tensor(q) for q in
                                       questions[batch_start:batch_start + bsz]]

                q_ids_batch = torch.stack(batch_token_tensors, dim=0).cuda()
                q_seg_batch = torch.zeros_like(q_ids_batch).cuda()
                q_attn_mask = self.tensorizer.get_attn_mask(q_ids_batch)
                _, out, _ = self.question_encoder(q_ids_batch, q_seg_batch, q_attn_mask)

                query_vectors.extend(out.cpu().split(1, dim=0))

                if len(query_vectors) % 100 == 0:
                    logger.info('Encoded queries %d', len(query_vectors))

        query_tensor = torch.cat(query_vectors, dim=0)

        logger.info('Total encoded queries tensor %s', query_tensor.size())

        assert query_tensor.size(0) == len(questions)
        return query_tensor
```

Question Vector Generator Method

```
def get_top_docs(self, query_vectors: np.array, top_docs: int = 100) -> List[Tuple[List[object], List[float]]]:
    """
    Does the retrieval of the best matching passages given the query vectors batch
    :param query_vectors:
    :param top_docs:
    :return:
    """
    time0 = time.time()
    results = self.index.search_knn(query_vectors, top_docs)
    logger.info('index search time: %f sec.', time.time() - time0)
    return results
```

Method to Retrieves Top Docs

```
def validate(passages: Dict[object, Tuple[str, str]], answers: List[List[str]],
             result_ctx_ids: List[Tuple[List[object], List[float]]],
             workers_num: int, match_type: str) -> List[List[bool]]:
    match_stats = calculate_matches(passages, answers, result_ctx_ids, workers_num, match_type)
    top_k_hits = match_stats.top_k_hits

    logger.info('Validation results: top k documents hits %s', top_k_hits)
    top_k_hits = [v / len(result_ctx_ids) for v in top_k_hits]
    logger.info('Validation results: top k documents hits accuracy %s', top_k_hits)
    return match_stats.questions_doc_hits
```

Validation Method

Coded another Pipeline to train the reader model on top of the retriever results. A ReaderTrainer class is defined which is similar to the BiEncoder Training class and few of the methods included in it are as follows:

```
def run_train(self):
    args = self.args

    train_iterator = self.get_data_iterator(args.train_file, args.batch_size,
                                            True,
                                            shuffle=True,
                                            shuffle_seed=args.seed, offset=self.start_batch)

    num_train_epochs = args.num_train_epochs - self.start_epoch

    logger.info("Total iterations per epoch=%d", train_iterator.max_iterations)
    updates_per_epoch = train_iterator.max_iterations // args.gradient_accumulation_steps
    total_updates = updates_per_epoch * num_train_epochs - self.start_batch
    logger.info(" Total updates=%d", total_updates)

    warmup_steps = args.warmup_steps
    scheduler = get_schedule_linear(self.optimizer, warmup_steps=warmup_steps,
                                    training_steps=total_updates)
    if self.scheduler_state:
        logger.info("Loading scheduler state %s", self.scheduler_state)
        scheduler.load_state_dict(self.scheduler_state)

    eval_step = args.eval_step
    logger.info("  Eval step = %d", eval_step)
    logger.info("***** Training *****")

    global_step = self.start_epoch * updates_per_epoch + self.start_batch

    for epoch in range(self.start_epoch, int(args.num_train_epochs)):
        logger.info("***** Epoch %d *****", epoch)
        global_step = self._train_epoch(scheduler, epoch, eval_step, train_iterator, global_step)

    if args.local_rank in [-1, 0]:
        logger.info('Training finished. Best validation checkpoint %s', self.best_cp_name)

    return
```

ReaderTrainer Training Method

```
def _calc_loss(self, input: ReaderBatch) -> torch.Tensor:
    args = self.args
    input = ReaderBatch(**move_to_device(input._asdict(), args.device))
    attn_mask = self.tensorizer.get_attn_mask(input.input_ids)
    questions_num, passages_per_question, _ = input.input_ids.size()

    if self.reader.training:
        # start_logits, end_logits, rank_logits = self.reader(input.input_ids, attn_mask)
        loss = self.reader(input.input_ids, attn_mask, input.start_positions, input.end_positions,
                           input.answers_mask)

    else:
        # TODO: remove?
        with torch.no_grad():
            start_logits, end_logits, rank_logits = self.reader(input.input_ids, attn_mask)

        loss = compute_loss(input.start_positions, input.end_positions, input.answers_mask, start_logits,
                            end_logits,
                            rank_logits,
                            questions_num, passages_per_question)
    if args.n_gpu > 1:
        loss = loss.mean()
    if args.gradient_accumulation_steps > 1:
        loss = loss / args.gradient_accumulation_steps

    return loss
```

ReaderTrainer Loss Calculation Method

```
def _save_predictions(self, out_file: str, prediction_results: List[ReaderQuestionPredictions]):
    logger.info('Saving prediction results to  %s', out_file)
    with open(out_file, 'w', encoding="utf-8") as output:
        save_results = []
        for r in prediction_results:
            save_results.append({
                'question': r.id,
                'gold_answers': r.gold_answers,
                'predictions': [{
                    'top_k': top_k,
                    'prediction': {
                        'text': span_pred.prediction_text,
                        'score': span_pred.span_score,
                        'relevance_score': span_pred.relevance_score,
                        'passage_idx': span_pred.passage_index,
                        'passage': self.tensorizer.to_string(span_pred.passage_token_ids)
                    }
                } for top_k, span_pred in r.predictions.items()]
            })
        output.write(json.dumps(save_results, indent=4) + "\n")
```

ReaderTrainer Prediction Method

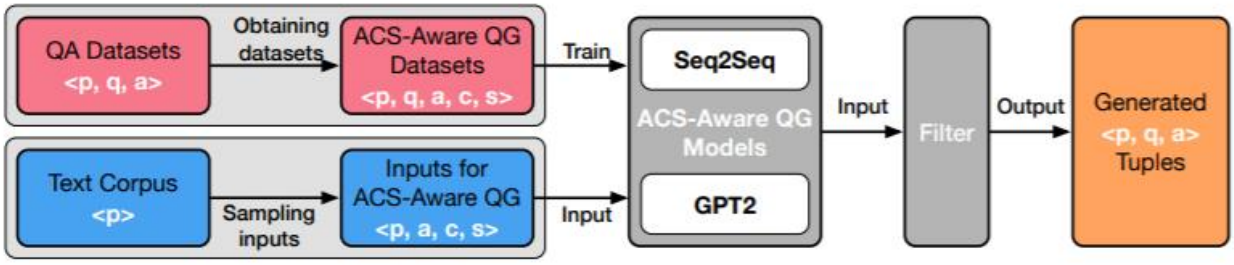
The DPR Top-20 & Top-100 retrieval accuracy on test sets, measured as the percentage of top 20/100 retrieved passages that contain the answer trained on the SQuAD dataset are 62% and 75%. The DPR gave an End-to-end QA (Exact Match) Accuracy of 28% on the SQuAD v1.1 dataset.

6. Answer-Clue-Style-aware Question Generation (ACS-QG)

The ability to ask questions is important in both human and machine intelligence. Learning to ask questions helps knowledge acquisition, improves question-answering and machine reading comprehension tasks, and helps a chatbot to keep the conversation flowing with a human. Existing question generation models are ineffective at generating a large amount of high-quality question-answer pairs from unstructured text, since given an answer and an input passage, question generation is inherently a one-to-many mapping.

Answer-Clue-Style-aware Question Generation (ACS-QG), aims at automatically generating high-quality and diverse question-answer pairs from unlabeled text corpus at scale by imitating the way a human asks questions. The system consists of:

- an information extractor, which samples from the text multiple types of assistive information to guide question generation.
- neural question generators, which generate diverse and controllable questions, leveraging the extracted assistive information.
- a neural quality controller, which removes low-quality generated data based on text entailment.



ACS-QG Architecture

A separate content function words method is defined which does the following:

- merge stop words, function words/phrases lists to get FUNCTION_WORDS_LIST
- get chunks of spacy_doc

- for each word or chunk, if it is inside the FUNCTION_WORDS_LIST, we tag the word or whole chunk as function words. The rest are content words.

The below function defines which words form the given input should belong in the answer:

```
def select_answers(context, processed_by_spacy=False):
    """
    Input a context, we select which part of the input words belonging to the answer.
    """
    # return a list of [(answer_text, answer_bio_ids)] tuples.
    tree = None
    try:
        tree = PARSER.parse(context)  # TODO: if the context is too long, it will cause error.
    except:
        pass
    if not processed_by_spacy:
        doc = NLP(context)
    else:
        doc = context
    token2idx, idx2token = get_token2char(doc)
    max_depth, node_num, chunklist = _navigate(tree)
    answer_chunks = _post(chunklist)
    answers = []
    for chunk in answer_chunks:
        label, leaves, st, ed = chunk
        # print('leaves={}\tst={}\ted={}\tlabel={}'.format(leaves, st, ed, label))
        try:
            char_st, char_ed = str_find(context, leaves)
            if char_st < 0:
                continue
            answer_text = context[char_st:char_ed + 1]
            st = idx2token[char_st]
            ed = idx2token[char_ed]
            answer_bio_ids = ['O'] * len(doc)
            answer_bio_ids[st: ed + 1] = ['I'] * (ed - st + 1)
            answer_bio_ids[st] = 'B'
            char_st = token2idx[st][0]
            char_ed = token2idx[ed][1]
            # print('answer_text={}\tchar_st={}\tchar_ed={}\tst={}\ted={}'.format(answer_text, char_st, char_ed, st, ed))
        except:
            continue
        answers.append((answer_text, char_st, char_ed, answer_bio_ids, label))

    return answers
```

Function to select the Words in Answer

A class is defined to train the ACS, Trainer which trains on the given dataset and has the following methods:

```
def _update_best_result(self, new_result, best_result):
    is_best = False
    # VARIABLE
    if (new_result["bleu4"] > best_result["bleu4"]):
        is_best = True
    for key in self.result_keys:
        best_result[key] = max(best_result[key], new_result[key])
    return best_result, is_best

def _result2string(self, result, result_keys):
    string = ""
    for key in result_keys:
        string += "_" + key + "_" + ("{:.5f}").format(result[key])
    return string
```

The main train method is as follows:

```
def train(self):
    patience = 0
    for epoch in range(self.start_epoch, self.args.epochs + 1):
        result = self._train_epoch(epoch)

        self.best_result, is_best = self._update_best_result(
            result, self.best_result)

        if self.args.use_early_stop:
            if (not is_best):
                patience += 1
                if patience > self.args.early_stop:
                    print("Perform early stop!")
                    break
            else:
                patience = 0

        if epoch % self.args.save_freq == 0:
            self._save_checkpoint(
                epoch, result, self.result_keys, is_best)  # !!!
    return self.model
```

ACS Train Method

```
        def eval(self, dataloader, eval_file, output_file):
            eval_dict = load(eval_file)
            result = self._valid(eval_dict, dataloader)
            print("eval: " + self._result2string(result, self.result_keys))
            if output_file is not None:
                with open(output_file, 'w', encoding='utf8') as outfile:
                    json.dump(result, outfile)
            return result
```

ACS Trainer Evaluation Method

```
def test(self, dataloader, output_file):
    result, example_sid, example_pid, example_ans_sent, example_answer_text, example_char_start, example_char_end = self._test(dataloader)
    if output_file is not None:
        with open(output_file, 'w', encoding='utf8') as outfile:
            for i in range(len(result)):
                q = result[i].rstrip()
                to_print = [
                    str(example_pid[i]),
                    str(example_sid[i]),
                    q,
                    str(example_ans_sent[i]),
                    str(example_answer_text[i]),
                    str(example_char_start[i]),
                    str(example_char_end[i])]

                outfile.write("\t".join(to_print).rstrip().replace("\n", "\\n") + "\n")
    return result
```

ACS Trainer Test Method

The other main function of this class is train_epoch class which trains the dataset on given number of epochs.

The evaluation results suggest that our system dramatically outperforms state-of-the-art neural question generation models in terms of the generation quality, while being scalable in the meantime. With models trained on a relatively smaller amount of data, we can generate 2.8 million quality-assured question-answer pairs from a million sentences.

7. References

[1] https://arxiv.org/pdf/2004.04906v3.pdf

[2] https://arxiv.org/pdf/2002.00748v2.pdf

[3] https://github.com/BangLiu/ACS-QG

[4] https://github.com/facebookresearch/DPR

[5] https://en.wikipedia.org/wiki/Question_answering

[6] https://github.com/huggingface/transformers

[7] https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html