NLP for Question Answering

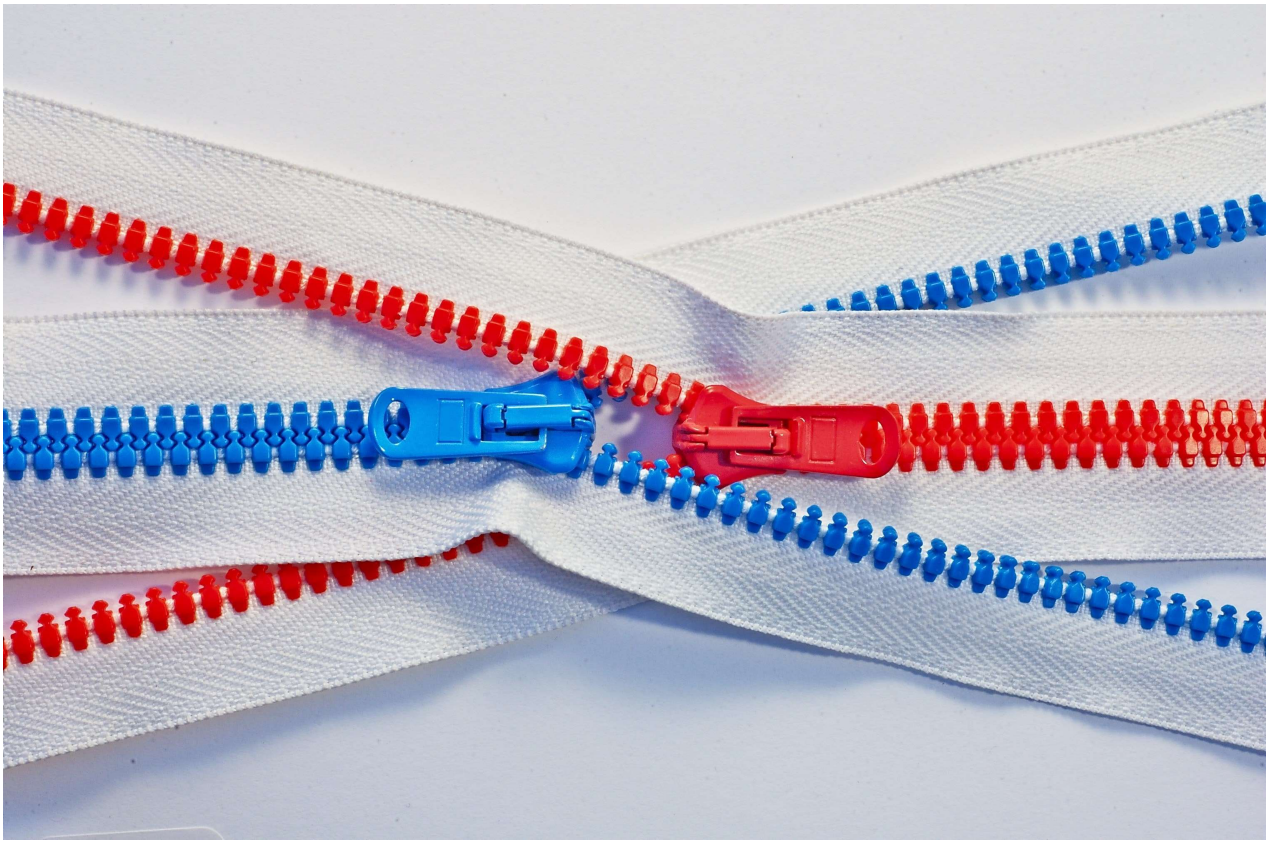
Evaluating QA: Metrics, Predictions, and the Null Response

A deep dive into computing QA predictions and when to tell BERT to zip it!

Jun 9, 2020 • 31 min read

no answer null threshold bert distilbert exact match F1 robust predictions

View On GitHub launch binder Open in Colab



Sometimes BERT needs to zip it.

In our last post, Building a QA System with BERT on Wikipedia, we used the HuggingFace framework to train BERT on the SQuAD2.0 dataset and built a simple QA system on top of the Wikipedia search engine. This time, we'll look at how to assess the quality of a BERT-like model for Question Answering. We'll cover what metrics are used to quantify quality, how to evaluate a model using the Hugging Face framework, and the importance of the "null response" (questions that don't have answers) for both improved performance and more realistic QA output. By the end of this post, we'll have implemented a more robust answering method for our QA system.

Note: Throughout this post we'll be using a distilBERT model fine-tuned on SQuAD2.0 by a member of the NLP community; this model can be found here in the HF repository. Additionally, much of the code in this post is inspired by the HF `squad_metrics.py` script.

Prerequisites

- a basic understanding of Transformers and PyTorch

- a basic understanding of Transformer outputs (logits) and softmax
- a Transformer fine-tuned on SQuAD2.0
- the SQuAD2.0 dev set

Answering questions is complicated

Quantifying the success of question answering is a tricky task. When you or I ask a question, the correct answer could take multiple forms. For example, in our previous post, BERT answered the question, "Why is the sky blue?" with "Rayleigh scattering," but another answer would be:

> *The Earth's atmosphere scatters short-wavelength light more efficiently than that of longer wavelengths. Because its wavelengths are shorter, blue light is more strongly scattered than the longer-wavelength lights, red or green. Hence the result that when looking at the sky away from the direct incident sunlight, the human eye perceives the sky to be blue.*

Both of these answers can be found in the Wikipedia article Diffuse Sky Radiation and both are correct. However, we've also had a model answer the same question with "because its wavelengths are shorter," which is close - but not really a correct answer; the sky itself doesn't have a wavelength. This answer is missing too much context to be useful. What if we'd asked a question that couldn't be answered by the Diffuse Sky Radiation page? For example: "Could the sky ever be green?" If you read that Wiki article you'll see there probably isn't a sure-fire answer to this question. What should the model do in this case?

How should we judge a model's success when there are multiple correct answers, even more incorrect answers, and potentially no answer available to it at all? To properly assess quality, we need a labeled set of questions and answers. Let's turn back to the SQuAD dataset.

The SQuAD2.0 dev set

The SQuAD dataset comes in two flavors: SQuAD1.1 and SQuAD2.0. The latter contains the same questions and answers as the former, but also includes additional questions that cannot be answered by the accompanying passage. This is intended to create a more realistic question answering task. The ability to identify unanswerable questions is much more challenging for Transformer models, which is why we focused on the SQuAD2.0 dataset rather than SQuAD1.1.

SQuAD2.0 consists of over 150k questions, of which more than 35% are unanswerable in relation to their associated passage. For our last post, we fine-tuned on the train set (130k examples); now we'll focus on the dev set, which contains nearly 12k examples. Only about half of these examples are answerable questions. In the following section, we'll look at a couple of these examples to get a feel for them.

(Use the hidden cells below to get set up, if needed.)

Show Code

Show Code

Show Code

Load the dev set using HF data processors

Hugging Face provides the Processors library for facilitating basic processing tasks with some canonical NLP datasets. The processors can be used for loading datasets and converting their examples to features for direct use in the model. We'll be using the SQuAD processors.

```
from transformers.data.processors.squad import SquadV2Processor

# this processor loads the SQuAD2.0 dev set examples
processor = SquadV2Processor()
examples = processor.get_dev_examples("./data/squad/", filename="dev-v2.0.json")
print(len(examples))
```

```
100%|██████████| 35/35 [00:05<00:00,  6.71it/s]
```

```
11873
```

While `examples` is a list, most other tasks we'll work with use a unique identifier - one for each question in the dev set.

```
# generate some maps to help us identify examples of interest
qid_to_example_index = {example.qas_id: i for i, example in enumerate(examples)}
qid_to_has_answer = {example.qas_id: bool(example.answers) for example in examples}
answer_qids = [qas_id for qas_id, has_answer in qid_to_has_answer.items() if has_answer]
no_answer_qids = [qas_id for qas_id, has_answer in qid_to_has_answer.items() if not has_ans
```

```
def display_example(qid):
    from pprint import pprint

    idx = qid_to_example_index[qid]
    q = examples[idx].question_text
    c = examples[idx].context_text
    a = [answer['text'] for answer in examples[idx].answers]

    print(f'Example {idx} of {len(examples)}\n---------------------')
    print(f"Q: {q}\n")
    print("Context:")
    pprint(c)
    print(f"\nTrue Answers:\n{a}")
```

A positive example

Approximately 50% of the examples in the dev set are questions that have answers contained within their corresponding passage. In these cases, up to five possible correct answers are provided (questions and answers were generated and identified by crowd-sourced workers). Answers must be direct excerpts from the passage, but we can see there are several ways to arrive at a "correct" answer.

```
display_example(answer_qids[1300])
```

```
Example 2548 of 11873
---------------------
Q: Where on Earth is free oxygen found?

Context:
```

```
("Free oxygen also occurs in solution in the world's water bodies. The "
 'increased solubility of O\n'
 '2 at lower temperatures (see Physical properties) has important implications '
 'for ocean life, as polar oceans support a much higher density of life due to '
 'their higher oxygen content. Water polluted with plant nutrients such as '
 'nitrates or phosphates may stimulate growth of algae by a process called '
 'eutrophication and the decay of these organisms and other biomaterials may '
 'reduce amounts of O\n'
 '2 in eutrophic water bodies. Scientists assess this aspect of water quality '
 "by measuring the water's biochemical oxygen demand, or the amount of O\n"
 '2 needed to restore it to a normal concentration.')

True Answers:
['water', "in solution in the world's water bodies", "the world's water bodies"]
```

A negative example

The other half of the questions in the dev set do not have an answer in the corresponding passage. These questions were generated by crowd-sourced workers to be related and relevant to the passage, but unanswerable by that passage. There are thus no True Answers associated with these questions, as we see in the example below.

Note: In this case, the question is a trick -- the numbers are reoriented in a way that no longer holds true. Will the model pick up on that?

```
display_example(no_answer_qids[1254])
```

```
Example 2564 of 11873
---------------------
Q: What happened 3.7-2 billion years ago?

Context:
("Free oxygen gas was almost nonexistent in Earth's atmosphere before "
 'photosynthetic archaea and bacteria evolved, probably about 3.5 billion '
 'years ago. Free oxygen first appeared in significant quantities during the '
 'Paleoproterozoic eon (between 3.0 and 2.3 billion years ago). For the first '
 'billion years, any free oxygen produced by these organisms combined with '
 'dissolved iron in the oceans to form banded iron formations. When such '
 'oxygen sinks became saturated, free oxygen began to outgas from the oceans '
 '3–2.7 billion years ago, reaching 10% of its present level around 1.7 '
 'billion years ago.')

True Answers:
[]
```

Metrics for QA

There are two dominant metrics used by many question answering datasets, including SQuAD: exact match (EM) and F1 score. These scores are computed on individual question+answer pairs. When multiple correct answers are possible for a given question, the maximum score over all possible correct answers is computed. Overall EM and F1 scores are computed for a model by averaging over the individual example scores.

Exact Match

This metric is as simple as it sounds. For each question+answer pair, if the *characters* of the model's prediction exactly match the characters of (one of) the True Answer(s), EM = 1, otherwise EM = 0. This is a strict all-or-nothing metric; being off by a single character results in a score of 0. When assessing against a negative example, if the model predicts any text at all, it automatically receives a 0 for that example.

F1

F1 score is a common metric for classification problems, and widely used in QA. It is appropriate when we care equally about precision and recall. In this case, it's computed over the individual *words* in the prediction against those in the True Answer. The number of shared words between the prediction and the truth is the basis of the F1 score: precision is the ratio of the number of shared words to the total number of words in the *prediction*, and recall is the ratio of the number of shared words to the total number of words in the *ground truth*.

$$F_1 = \frac{2}{\text{recall}^{-1} + \text{precision}^{-1}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Thanks Wikipedia

Let's see how these metrics work in practice. We'll load up a fine-tuned model (this one, to be precise) and its tokenizer, and compare our predictions against the True Answers.

Load a Transformer model fine-tuned on SQuAD 2.0

```
tokenizer = AutoTokenizer.from_pretrained("twmkn9/distilbert-base-uncased-squad2")
model = AutoModelForQuestionAnswering.from_pretrained("twmkn9/distilbert-base-uncased-squac
```

The following `get_prediction` method is essentially identical to what we used last time in our simple QA system.

```
def get_prediction(qid):
    # given a question id (qas_id or qid), load the example, get the model outputs and gene
    question = examples[qid_to_example_index[qid]].question_text
    context = examples[qid_to_example_index[qid]].context_text

    inputs = tokenizer.encode_plus(question, context, return_tensors='pt')

    outputs = model(**inputs)
    answer_start = torch.argmax(outputs[0])  # get the most likely beginning of answer with
    answer_end = torch.argmax(outputs[1]) + 1

    answer = tokenizer.convert_tokens_to_string(tokenizer.convert_ids_to_tokens(inputs['inp

    return answer
```

Below are some functions we'll need to compute our quality metrics.

```
# these functions are heavily influenced by the HF squad_metrics.py script
def normalize_text(s):
    """Removing articles and punctuation, and standardizing whitespace are all typical text
    import string, re

    def remove_articles(text):
        regex = re.compile(r"\b(a|an|the)\b", re.UNICODE)
        return re.sub(regex, " ", text)

    def white_space_fix(text):
```

```
        return " ".join(text.split())

    def remove_punc(text):
        exclude = set(string.punctuation)
        return "".join(ch for ch in text if ch not in exclude)

    def lower(text):
        return text.lower()

    return white_space_fix(remove_articles(remove_punc(lower(s))))

def compute_exact_match(prediction, truth):
    return int(normalize_text(prediction) == normalize_text(truth))

def compute_f1(prediction, truth):
    pred_tokens = normalize_text(prediction).split()
    truth_tokens = normalize_text(truth).split()

    # if either the prediction or the truth is no-answer then f1 = 1 if they agree, 0 other
    if len(pred_tokens) == 0 or len(truth_tokens) == 0:
        return int(pred_tokens == truth_tokens)

    common_tokens = set(pred_tokens) & set(truth_tokens)

    # if there are no common tokens then f1 = 0
    if len(common_tokens) == 0:
        return 0

    prec = len(common_tokens) / len(pred_tokens)
    rec = len(common_tokens) / len(truth_tokens)

    return 2 * (prec * rec) / (prec + rec)

def get_gold_answers(example):
    """helper function that retrieves all possible true answers from a squad2.0 example"""

    gold_answers = [answer["text"] for answer in example.answers if answer["text"]]

    # if gold_answers doesn't exist it's because this is a negative example -
    # the only correct answer is an empty string
    if not gold_answers:
        gold_answers = [""]

    return gold_answers
```

In the following cell, we start by computing EM and F1 for our first example - the one that has several True Answers associated with it.

```
prediction = get_prediction(answer_qids[1300])
example = examples[qid_to_example_index[answer_qids[1300]]]

gold_answers = get_gold_answers(example)

em_score = max((compute_exact_match(prediction, answer)) for answer in gold_answers)
f1_score = max((compute_f1(prediction, answer)) for answer in gold_answers)

print(f"Question: {example.question_text}")
print(f"Prediction: {prediction}")
print(f"True Answers: {gold_answers}")
print(f"EM: {em_score} \t F1: {f1_score}")
```

```
Question: Where on Earth is free oxygen found?
Prediction: water bodies
True Answers: ['water', "in solution in the world's water bodies", "the world's water bodies"]
EM: 0    F1: 0.8
```

We see that our prediction is actually quite close to some of the True Answers, resulting in a respectable F1 score. However, it does not exactly match any of them, so our EM score is 0.

Let's try with our negative example now.

```
prediction = get_prediction(no_answer_qids[1254])
example = examples[qid_to_example_index[no_answer_qids[1254]]]

gold_answers = get_gold_answers(example)

em_score = max((compute_exact_match(prediction, answer)) for answer in gold_answers)
f1_score = max((compute_f1(prediction, answer)) for answer in gold_answers)

print(f"Question: {example.question_text}")
print(f"Prediction: {prediction}")
print(f"True Answers: {gold_answers}")
print(f"EM: {em_score} \t F1: {f1_score}")
```

```
Question: What happened 3.7-2 billion years ago?
Prediction: [CLS] what happened 3 . 7 - 2 billion years ago ? [SEP] free oxygen gas was almost nonexist
```

```
True Answers: ['']
EM: 0    F1: 0
```

Wow. Both our metrics are zero, because this model does not correctly assess that this question is unanswerable! Even worse, it seems to have catastrophically failed, including the entire question as part of the answer. In a later section, we'll explicitly dig into why this happens, but for now, it's important to note that we got this answer because we simply extracted start and end tokens associated with the maximum score (we took an `argmax` of the model output in `get_prediction`) and this lead to some unintended consequences.

Now that we've seen the basics of computing QA metrics on a couple of examples, we need to assess the model on the entire dev set. Luckily, there's a script for that.

Evaluating a model on the SQuAD2.0 dev set with HF

The same `run_squad.py` script we used to fine-tune a Transformer for question answering can also be used to evaluate the model. (You can grab the script here or run the hidden cell below.)

Show Code

Below are the arguments needed to properly evaluate a fine-tuned model for question answering on the SQuAD dev set. Because we're using SQuAD2.0, it is **crucial** to include the `--version_2_with_negative` flag!

```
!python run_squad.py  \
    --model_type distilbert   \
    --model_name_or_path twmkn9/distilbert-base-uncased-squad2  \
    --output_dir models/distilbert/twmkn9_distilbert-base-uncased-squad2 \
    --data_dir data/squad   \
    --predict_file dev-v2.0.json   \
    --do_eval   \
    --version_2_with_negative \
```

```
    --do_lower_case  \
    --per_gpu_eval_batch_size 12   \
    --max_seq_length 384   \
    --doc_stride 128
```

Refer to our last post for more details on what these arguments mean and what this script does. For our immediate purposes, running the cell above will produce the following output in the `--output_dir` directory:

- `predictions_.json`
- `nbest_predictions_.json`
- `null_odds_.json`

(We'll go over what these are later on.) Additionally, an overall `Results` dict will be displayed to the screen. If you run the above cell, the last line of output should display something like the following:

```
Results = {
    # a) scores averaged over all examples in the dev set
    'exact': 66.25958056093658,
    'f1': 69.66994428499025,
    'total': 11873,  # number of examples in the dev set

    # b) scores averaged over only positive examples (have answers)
    'HasAns_exact': 68.91025641025641,
    'HasAns_f1': 75.74076391627662,
    'HasAns_total': 5928, # number of positive examples

    # c) scores averaged over only negative examples (no answers)
    'NoAns_exact': 63.61648444070648,
    'NoAns_f1': 63.61648444070648,
    'NoAns_total': 5945, # number of negative examples

    # d) given probabilities of no-answer for each example, what would the best scores and
    'best_exact': 66.25958056093658,
    'best_exact_thresh': 0.0,
    'best_f1': 69.66994428499046,
    'best_f1_thresh': 0.0
}
```

The first three blocks of the `Results` output are pretty straightforward. EM and F1 scores are reported over a) the full dev set, b) the set of positive examples, and c) the set of negative examples. This can provide some insight into whether a model is performing adequately on both answer and no-answer questions. (This particular model is pretty bad at no-answer questions).

However, what's going on with the last block? This portion of the output is not useful unless we supply the evaluation method with additional information. For that, we'll need to dig deeper into the evaluation process - because it turns out that we need to compute more than just a prediction for an answer; we must also compute a prediction for NO answer and we must score both predictions!

The following section will dive into the technical details of computing robust predictions on SQuAD2.0 examples, including how to score an answer and the null answer, as well as how to determine which one should be the "correct" prediction for a given example. Feel free to skip to the next section for the punchline. (For those of you considering building your own QA system, we found this information to be invaluable for understanding the inner workings of prediction and assessment.)

[Optional] Computing predictions

> **Note:** The code in the following section is an under-the-hood dive into the HF `compute_predictions_logits` method in their `squad_metrics.py` script.

When the tokenized question+context is passed to the model, the output consists of two sets of logits: one for the start of the answer span, the other for the end of the answer span. These logits represent the likelihood of any given token being the start or end of the answer. Every token passed to the model is assigned a logit, including special tokens (e.g., [CLS], [SEP]), and tokens corresponding to the question itself.

Let's walk through the process using our last example (Q: What happened 3.7-2 billion years ago?).

```
inputs = tokenizer.encode_plus(example.question_text, example.context_text, return_tensors=
start_logits, end_logits = model(**inputs)
```

```
# look at how large the logit is in the [CLS] position (index 0)!
# strong possibility that this question has no answer... but our prediction returned an ans
start_logits
```

```
tensor([[  6.4914,  -9.1416,  -8.4068,  -7.5684,  -9.9081,  -9.4256, -10.1625,
          -9.2579, -10.0554,  -9.9653,  -9.2002,  -8.8657,  -9.1162,   0.6481,
          -2.5947,  -4.5072,  -8.1189,  -6.5871,  -5.8973, -10.8619, -11.0953,
         -10.2294,  -9.3660,  -7.6017, -10.8009, -10.8197,  -6.1258,  -8.3507,
          -4.2463, -10.0987, -10.2659,  -8.8490,  -6.7346,  -8.6513,  -9.7573,
          -5.7496,  -5.5851,  -8.9483,  -7.0652,  -6.1369,  -5.7810,  -9.4366,
          -8.7670,  -9.6743,  -9.7446,  -7.7905,  -7.4541,  -1.5963,  -3.8540,
          -7.3450,  -8.1854,  -9.5566,  -8.3416,  -8.9553,  -8.3144,  -6.4132,
          -4.2285,  -9.4427,  -9.5111,  -9.2931,  -8.9154,  -9.3930,  -8.2111,
          -8.9774,  -9.0274,  -7.2652,  -7.4511,  -9.8597,  -9.5869,  -9.9735,
          -7.0526,  -9.7560,  -8.7788,  -9.5117,  -9.6391,  -8.6487,  -9.5994,
          -7.8213,  -5.1754,  -4.3561,  -4.3913,  -7.8499,  -7.7522,  -8.9651,
          -3.5229,  -0.8312,  -2.7668,  -7.9180, -10.0320,  -8.7797,  -4.5965,
          -5.9465,  -9.9442,  -3.2135,  -5.0734,  -8.3462,  -7.5366,  -3.7073,
          -7.0968,  -4.3325,  -1.3691,  -4.1477,  -5.3794,  -7.6138,   1.3183,
          -3.4190,   3.1457,  -3.0152,  -0.4102,  -2.4606,  -3.5971,   6.4519,
          -0.5654,   0.9829,  -1.6682,   3.3549,  -4.7847,  -2.8024,  -3.3160,
          -0.5868,  -0.9617,  -8.1925,  -4.3299,  -7.3923,  -5.0875,  -5.3880,
          -5.3676,  -3.0878,  -4.3427,   4.3975,   1.8860,  -5.4661,  -9.1565,
          -3.6369,  -3.5462,  -4.1448,  -2.0250,  -2.4492,  -8.7015,  -7.3292,
          -7.7616,  -7.0786,  -4.6668,  -4.4089,  -9.1182]],
       grad_fn=<SqueezeBackward1>)
```

In our simple QA system, we predicted the best answer by selecting the start and end tokens with the largest logits, but that's not very robust. In fact, the original BERT paper suggested considering any sensible start+end combination as a possible answer to the question. These combinations would then be scored, and the one with the highest score would be considered the best answer. A possible (candidate) answer is scored as the sum of its start and end logits.

Note: This reflects how a basic span extraction classifier works. The raw hidden layer from the model is passed through a `Linear` layer and then fed to a `CrossEntropyLoss` for each class. In span extraction, there

are two classes: the beginning of the span and the end of the span. The span loss is computed as the sum of the `CrossEntropyLoss` for the start and end positions. The probability of an answer span is the probability of a given start token S and an end token E: P(S and E) = P(S)P(E), because the start and end tokens are treated as being independent. Thus summing the start and end logits is equivalent to a product of their softmax probabilities.

To mimic this behavior, we'll start by taking the *n* largest `start_logits` and the *n* largest `end_logits` as candidates. Any sensible combination of these start + end tokens is considered a candidate answer; however, several consistency checks must first be performed. For example, an answer wherein the end token falls before the start token should be excluded, because that just doesn't make sense. Candidate answers wherein the start or end tokens are associated with question tokens are also excluded, because the answer to the question should obviously not be in the question itself! It is important to note that the [CLS] token and its corresponding logits are not removed, because this token indicates the null answer.

```
def to_list(tensor):
    return tensor.detach().cpu().tolist()

# convert our start and end logit tensors to lists
start_logits = to_list(start_logits)[0]
end_logits = to_list(end_logits)[0]

# sort our start and end logits from largest to smallest, keeping track of the index
start_idx_and_logit = sorted(enumerate(start_logits), key=lambda x: x[1], reverse=True)
end_idx_and_logit = sorted(enumerate(end_logits), key=lambda x: x[1], reverse=True)

# select the top n (in this case, 5)
print(start_idx_and_logit[:5])
print(end_idx_and_logit[:5])
```

```
[(0, 6.491387367248535), (111, 6.451895713806152), (129, 4.397505760192871), (115, 3.354909658432007),
[(119, 6.33292293548584), (0, 6.084450721740723), (135, 4.417276382446289), (116, 4.3764214515686035),
```

The null answer token (index 0) is in the top five of both the start and end logit lists.

In order to eventually predict a text answer (or empty string), we need to keep track of the indexes which will be used to pull the corresponding token ids later on. We'll also need to identify which indexes correspond to the question tokens, so we can ensure we don't allow a nonsensical prediction.

```
start_indexes = [idx for idx, logit in start_idx_and_logit[:5]]
end_indexes = [idx for idx, logit in end_idx_and_logit[:5]]

# convert the token ids from a tensor to a list
tokens = to_list(inputs['input_ids'])[0]

# question tokens are defined as those between the CLS token (101, at position 0) and first
question_indexes = [i+1 for i, token in enumerate(tokens[1:tokens.index(102)])]
question_indexes
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Next, we'll generate a list of candidate predictions by looping through all combinations of the start and end token indexes, excluding nonsensical combinations. We'll save these to a list for the next step.

```
import collections

# keep track of all preliminary predictions
PrelimPrediction = collections.namedtuple(
    "PrelimPrediction", ["start_index", "end_index", "start_logit", "end_logit"]
)

prelim_preds = []
for start_index in start_indexes:
    for end_index in end_indexes:
        # throw out invalid predictions
        if start_index in question_indexes:
            continue
        if end_index in question_indexes:
            continue
        if end_index < start_index:
            continue
        prelim_preds.append(
            PrelimPrediction(
                start_index = start_index,
                end_index = end_index,
```

```
                start_logit = start_logits[start_index],
                end_logit = end_logits[end_index]
            )
        )
```

With a list of sensible candidate predictions, it's time to score them.

For a candidate answer, score = `start_logit` + `end_logit`. Below, we sort our candidate predictions by their score.

```
# sort preliminary predictions by their score
prelim_preds = sorted(prelim_preds, key=lambda x: (x.start_logit + x.end_logit), reverse=Tr
pprint(prelim_preds[:5])
```

```
[PrelimPrediction(start_index=0, end_index=119, start_logit=6.491387367248535, end_logit=6.332922935485
 PrelimPrediction(start_index=111, end_index=119, start_logit=6.451895713806152, end_logit=6.3329229354
 PrelimPrediction(start_index=0, end_index=0, start_logit=6.491387367248535, end_logit=6.08445072174072
 PrelimPrediction(start_index=0, end_index=135, start_logit=6.491387367248535, end_logit=4.417276382446
 PrelimPrediction(start_index=111, end_index=135, start_logit=6.451895713806152, end_logit=4.4172763824
```

Next we need to convert our preliminary predictions into actual text (or the empty string, if null). We'll keep track of text predictions we've seen, because different token combinations can result in the same text prediction and we only want to keep the one with the highest score (we're looping in descending score order). Finally, we'll trim this list down to the best 5 predictions.

```
# keep track of all best predictions
BestPrediction = collections.namedtuple(  # pylint: disable=invalid-name
    "BestPrediction", ["text", "start_logit", "end_logit"]
)

nbest = []
seen_predictions = []
for pred in prelim_preds:

    # for now we only care about the top 5 best predictions
    if len(nbest) >= 5:
        break

    # loop through predictions according to their start index
    if pred.start_index > 0: # non-null answers have start_index > 0
```

```
        text = tokenizer.convert_tokens_to_string(
            tokenizer.convert_ids_to_tokens(
                tokens[pred.start_index:pred.end_index+1]
            )
        )
        # clean whitespace
        text = text.strip()
        text = " ".join(text.split())

        if text in seen_predictions:
            continue

        # flag this text as being seen -- if we see it again, don't add it to the nbest lis
        seen_predictions.append(text)

        # add this text prediction to a pruned list of the top 5 best predictions
        nbest.append(BestPrediction(text=text, start_logit=pred.start_logit, end_logit=pred

# and don't forget -- include the null answer!
nbest.append(BestPrediction(text="", start_logit=start_logits[0], end_logit=end_logits[0]))
```

The null answer is scored as the sum of the start_logit and end_logit associated with the [CLS] token.

At this point, we have a neat list of the top 5 best predictions for this question. The number of best predictions for each example is adjustable with the `--n_best_size` argument of the `run_squad.py` script. The `nbest` predictions for *every question* in the dev set are saved to disk under `nbest_predictions_.json` in `--output_dir`. (This is a great resource for digging into how a model is behaving.) Let's take a look at our `nbest` predictions.

```
pprint(nbest)
```

```
[BestPrediction(text='free oxygen began to outgas from the oceans', start_logit=6.451895713806152, end_
 BestPrediction(text='free oxygen began to outgas from the oceans 3 – 2 . 7 billion years ago , reachin
 BestPrediction(text='free oxygen began to outgas', start_logit=6.451895713806152, end_logit=4.37642145
 BestPrediction(text='free oxygen', start_logit=6.451895713806152, end_logit=4.125303268432617),
 BestPrediction(text='outgas from the oceans', start_logit=3.354909658432007, end_logit=6.3329229354858
 BestPrediction(text='', start_logit=6.491387367248535, end_logit=6.084450721740723)]
```

Our top prediction so far is "free oxygen began to outgas from the oceans," which is already a far cry better than what we originally predicted. This is because we have successfully excluded nonsensical predictions that would incorporate question tokens as part of the answer. However, we know it's still incorrect. Let's keep going.

The last step is to compute the null score -- more specifically, the difference between the null score and the best non-null score as shown below.

```
# compute the null score as the sum of the [CLS] token logits
score_null = start_logits[0] + end_logits[0]

# compute the difference between the null score and the best non-null score
score_diff = score_null - nbest[0].start_logit - nbest[0].end_logit

score_diff
```

```
-0.20898056030273438
```

This `score_diff` is computed for every example in the dev set and these scores are saved to disk in the `null_odds_.json`. Let's pull up the score stored for the example we're using and see how we did!

```
filename = model_dir + 'null_odds_.json'
null_odds = json.load(open(filename, 'rb'))

null_odds[example.qas_id]
```

```
-0.2090005874633789
```

We basically nailed it! (The full HF version contains a few more checks and some additional subtleties that could account for the slight differences in our `score_diff`.)

Using the null threshold

In the previous section we covered:

- how to generate more robust predictions (e.g., by excluding predictions that include question tokens in the answer),
- how to score a prediction as the sum of its start and end logits,
- and how to compute the score difference between the null prediction and the best text prediction.

The `run_squad.py` script performs all of these tasks for us and saves the score differences for every example in the `null_odds_.json`. With that, we can now start to make sense of the fourth block of the results output!

According to the original BERT paper,

> *We predict a non-null answer when s^i,j > s_null + τ , where the threshold τ is selected on the dev set to maximize F1.*

In other words, the authors are saying that one should predict a null answer for a given example if that example's score difference is above a certain threshold. What should that threshold be? How should we compute it? They give us a recipe:select the threshold that maximizes F1. Rather than rerunning `run_squad.py`, we can import the aptly-named method that computes SQuAD evaluation: `squad_evaluate`. (You can take a look at the code for yourself here.) To use `squad_evaluate` we'll need:

- the original examples (because that's where the True Answers are stored),
- `predictions_.json`,
- `null_odds_.json`,
- and a null threshold.

```
# load the predictions we generated earlier
filename = model_dir + 'predictions_.json'
preds = json.load(open(filename, 'rb'))

# load the null score differences we generated earlier
filename = model_dir + 'null_odds_.json'
null_odds = json.load(open(filename, 'rb'))
```

Let's re-evaluate our model on SQuAD2.0 using the `squad_evaluate` method. This method uses the score differences for each example in the dev set to determine thresholds that maximize either the EM score or the F1 score. It then recomputes the best possible EM score and F1 score associated with that null threshold.

```
from transformers.data.metrics.squad_metrics import squad_evaluate

# the default threshold is set to 1.0 -- we'll leave it there for now
results_default_thresh = squad_evaluate(examples,
                                        preds,
                                        no_answer_probs=null_odds,
                                        no_answer_probability_threshold=1.0)

pprint(results_default_thresh)
```

```
OrderedDict([('exact', 66.25958056093658),
             ('f1', 69.66994428499025),
             ('total', 11873),
             ('HasAns_exact', 68.91025641025641),
             ('HasAns_f1', 75.74076391627662),
             ('HasAns_total', 5928),
             ('NoAns_exact', 63.61648444070648),
             ('NoAns_f1', 63.61648444070648),
             ('NoAns_total', 5945),
             ('best_exact', 68.36519834919565),
             ('best_exact_thresh', -4.189256191253662),
             ('best_f1', 71.1144383018176),
             ('best_f1_thresh', -3.767639636993408)])
```

The first three blocks have identical values as in our initial evaluation because they are based on the default threshold (which is currently 1.0). However, the values in the fourth block have been updated by taking into account the `null_odds` information. When a given example's `score_diff` is greater than the threshold, the prediction is flipped to a null answer which affects the overall EM and F1 scores.

Let's use the `best_f1_thresh` and run the evaluation once more to see a breakdown of our model's performance on `HasAns` and `NoAns` examples:

```
best_f1_thresh = -3.7676548957824707
results_f1_thresh = squad_evaluate(examples,
                                   preds,
                                   no_answer_probs=null_odds,
                                   no_answer_probability_threshold=best_f1_thresh)

pprint(results_f1_thresh)
```

```
OrderedDict([('exact', 68.31466352227744),
             ('f1', 71.11106931335648),
             ('total', 11873),
             ('HasAns_exact', 61.53846153846154),
             ('HasAns_f1', 67.13929250294865),
             ('HasAns_total', 5928),
             ('NoAns_exact', 75.07148864592094),
             ('NoAns_f1', 75.07148864592094),
             ('NoAns_total', 5945),
             ('best_exact', 68.36519834919565),
             ('best_exact_thresh', -4.189256191253662),
             ('best_f1', 71.1144383018176),
             ('best_f1_thresh', -3.767639636993408)])
```

When we used the default threshold of 1.0, we saw that our `NoAns_f1` score was a mere 63.6, but when we use the `best_f1_thresh`, we now get a `NoAns_f1` score of 75 - nearly a 12 point jump! The downside is that we lose some ground in how well our model correctly predicts `HasAns` examples. Overall, however, we see a net increase of a couple points in both EM and F1 scores. This demonstrates that computing null scores and properly using a null threshold significantly increases QA performance on the SQuAD2.0 dev set with almost no additional work.

Putting it all together

Below we present a new method that will select more robust predictions, compute scores for the best text predictions (as well as for the null prediction), and use these scores along with a null threshold to determine whether the question should be answered. As a bonus, this method also computes and returns the probability of the answer, which is often easier to interpret than a

logit score. Prediction probabilities depend on `nbest`, since they are computed with a softmax over the number of most likely predictions.

```
def get_robust_prediction(example, tokenizer, nbest=10, null_threshold=1.0):

    inputs = get_qa_inputs(example, tokenizer)
    start_logits, end_logits = model(**inputs)

    # get sensible preliminary predictions, sorted by score
    prelim_preds = preliminary_predictions(start_logits,
                                           end_logits,
                                           inputs['input_ids'],
                                           nbest)

    # narrow that down to the top nbest predictions
    nbest_preds = best_predictions(prelim_preds, nbest, tokenizer)

    # compute the probability of each prediction - nice but not necessary
    probabilities = prediction_probabilities(nbest_preds)

    # compute score difference
    score_difference = compute_score_difference(nbest_preds)

    # if score difference > threshold, return the null answer
    if score_difference > null_threshold:
        return "", probabilities[-1]
    else:
        return nbest_preds[0].text, probabilities[0]
```

Show Code

Will we now get the right answer (an empty string) for that tricky no-answer example we were working with?

```
print(example.question_text)
get_robust_prediction(example, tokenizer, nbest=10, null_threshold=best_f1_thresh)
```

```
What happened 3.7-2 billion years ago?
```

```
('', 0.34412444013709165)
```

Woohoo!! We got the right answer this time!!

Even if we didn't have the best threshold in place, our additional checks still allow us to output more sensible looking answers, rejecting predictions that include the question tokens.

```
print(example.question_text)
get_robust_prediction(example, tokenizer, nbest=10, null_threshold=1.0)
```

```
What happened 3.7-2 billion years ago?
```

```
('free oxygen began to outgas from the oceans', 0.42410620054269993)
```

And if it hadn't been a trick question, this would be the correct answer! (Seems like distilBERT could use some improvement in number understanding.)

Final Thoughts

Using a robust prediction method like the above will do more than allow a model to perform better on a curated dev set, though this is an important first step. It will also provide the model with a slightly better ability to refrain from answering questions that simply don't have an answer in the associated passage. This is a crucial feature for QA models, because it's not enough to get an answer if that answer doesn't make sense. We want our models to tell us something useful -- and sometimes that means telling us nothing at all.