Only you can see this message

This story's distribution setting is off. Learn more

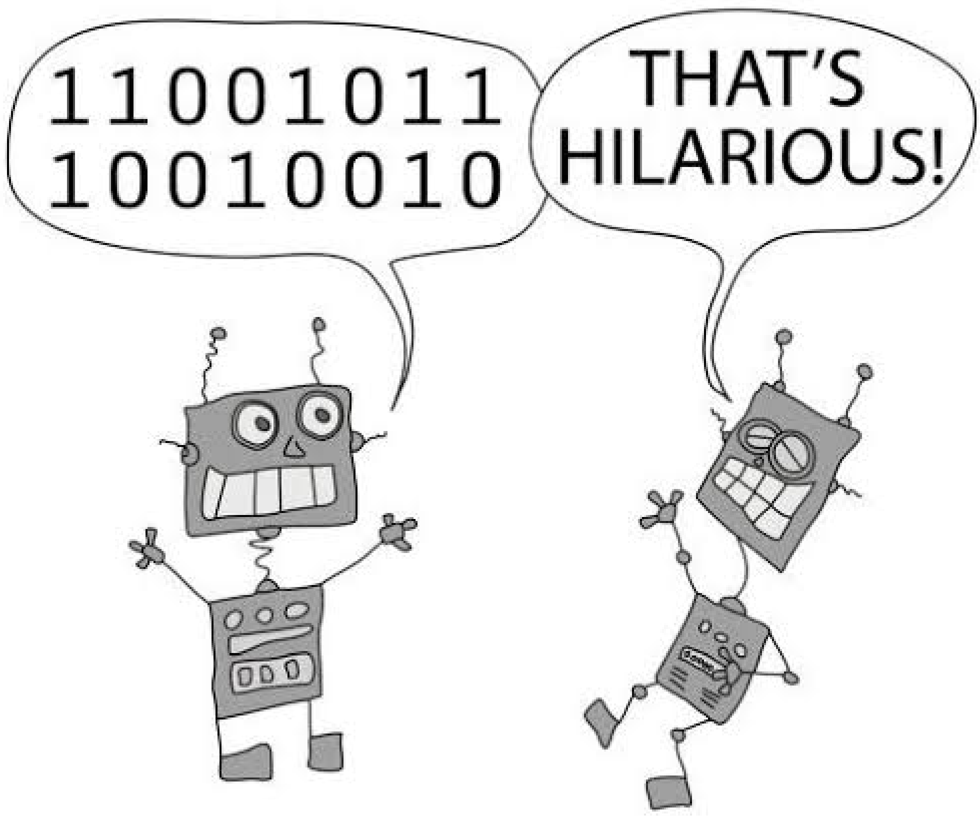
Humor Detection with a BERT Regressor

T

Theoliao

Apr 25 · 4 min read

Humor (or funniness) detection and generation have always been a challenge in NLP. This could be helpful to build many useful tools. Advising AIs could bring cleverer substitutes for text editing, and text generators might produce more human text. Chatbots and translation AIs might also become more humorous if they “understand” more about humor.



A robot telling joke (Pinterest)

In this article, we try to take it an easy way. That is, to evaluate the level of humor with a regressor trained on some manually graded data. BERT(Bidirectional Encoder Representations from Transformers), as a powerful NLP model with a high-quality pre-trained language model, is used to build such a regressor. Though still far from“understanding” humor, humor can be detected to some extent by a BERT regressor.

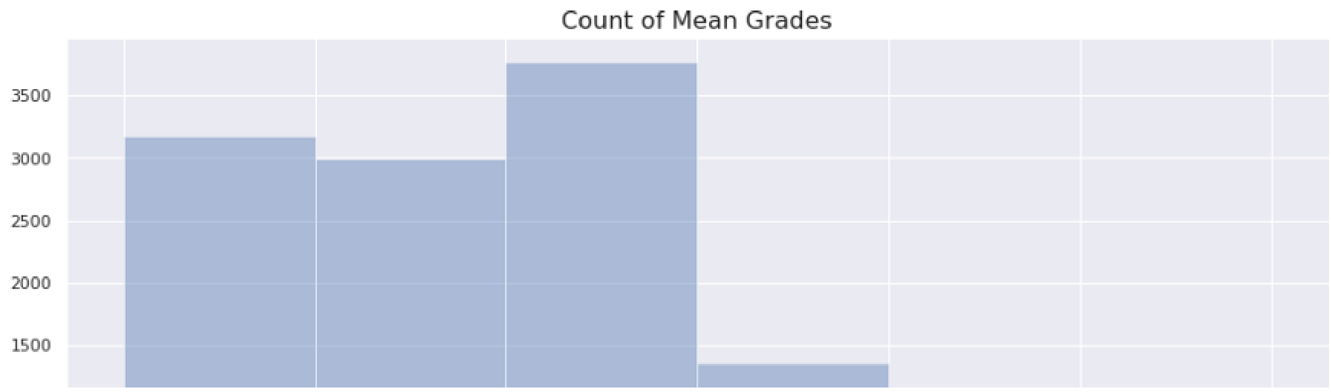
Data

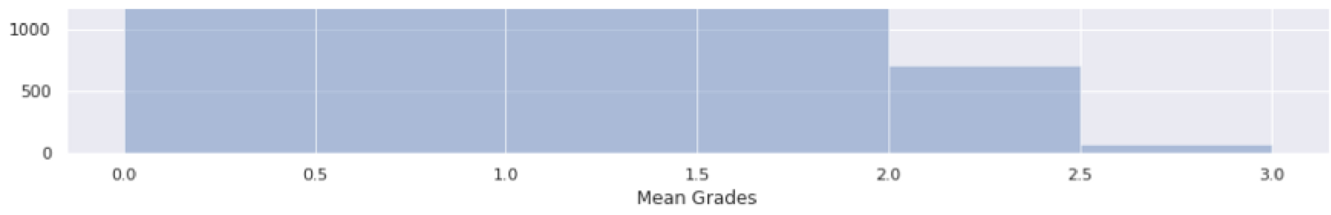
Humicroedit, a dataset introduced in “President Vows to Cut <Taxes> Hair”: Dataset and Analysis of Creative Text Editing for Humorous Headlines, contains collected news headlines with specified words edited to be funnier. Data can be obtained from Headline Humor Dataset — Rochester CS.

The dataset includes a training set and a validation set both with original headlines, edits, grades, and mean grades as shown below. There are 9652 observations in the training set and 2419 in the validation set.

	id	original	edit	grades	meanGrade
0	1723	Thousands of gay and bisexual <men/> convicted...	swans	22100	1.0
1	12736	Special <prosecutor/> appointed to Trump Russia	chef	21100	0.8
2	12274	Spanish police detain man and search Ripoll ad...	squad	21000	0.6
3	8823	N.Y. Times <reprimands/> reporter for sharing ...	applauds	32210	1.6

As we can see, grades here are 5-digit numbers representing 5 manually labeled integer grades from 0 to 3. And what we are interested most is the mean grades. The overall distribution of the mean grades can be visualized as follows.





For the input of the regressor, we can use the finalized edited headline, which can be obtained by replacing the edited words.

```python
import pandas as pd

train_df = pd.read_csv('./data/train.csv')
val_df = pd.read_csv('./data/dev.csv')

train_df['text'] = train_df.apply(lambda x:x['original'].replace(x['original'][x['original'].fir
val_df['text'] = val_df.apply(lambda x:x['original'].replace(x['original'][x['original'].find('<
```

1.py hosted with ♡ by GitHub view raw

We can also use sentence pairs that combine edited headlines with the original text. For instance, we can add a sentence in the form of “From <original word> to <edited word>”. Remember to add the “[SEP]” token to separate two sentences.

```python
train_df['old'] = train_df.apply(lambda x:x['original'][x['original'].find('<')+1:x['original'].
val_df['old'] = val_df.apply(lambda x:x['original'][x['original'].find('<')+1:x['original'].finc

train_df['text2'] = train_df.apply(lambda x:x['text'] + ' [SEP] From '+x['old'] + ' to '+x['edit
val_df['text2'] = val_df.apply(lambda x:x['text'] + ' [SEP] From '+x['old'] + ' to '+x['edit'] ,
```

2.py hosted with ♡ by GitHub view raw

Train a BERT Regressor

Next, we are going to train a BERT regressor with the preprocessed text as input and the mean grades as output. With the transformers package from the Hugging Face Library installed and GPU used, we refer to the notebook BERT Fine-Tuning Sentence Classification v3 for detailed usages of the BERT model here.

First, load the pre-trained BERT tokenizer and tokenize all the input text. Tokenize the text with added “[CLS]” and “[SEP]” tokens in the beginning and at the end as well as padding to form 32-word long embeddings. This can be similarly applied to combined sentence pairs *train_df[‘text2’]*.

```
# modified from notebook in https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FI

from transformers import BertTokenizer
import torch

# Load the BERT tokenizer.
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased",do_lower_case=True)

# Get the lists of sentences and their labels.
sentences = train_df['text'].values
labels = train_df['meanGrade'].values

input_ids = []
attention_masks = []

# For every sentence...
for sent in sentences:
    # `encode_plus` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = 32,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(labels)
```

Next, tokenize text in the validation set (and similarly the combined sentence pairs *val_df['text2']*) in the same way.

```
# modified from notebook in https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FIc

sentences_val = val_df['text'].values
labels_val = val_df['meanGrade'].values

# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids_val = []
attention_masks_val = []

# For every sentence...
for sent in sentences_val:
    # `encode_plus` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = 32,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    input_ids_val.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    attention_masks_val.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
input_ids_val = torch.cat(input_ids_val, dim=0)
attention_masks_val = torch.cat(attention_masks_val, dim=0)
labels_val = torch.tensor(labels_val)
```

Iterators for our dataset are created using the torch DataLoader class. This helps save memory since not all data need to be loaded with an iterator. The batch size is set as 32 here.

```python
# modified from notebook in https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FIc

from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler

train_dataset = TensorDataset(input_ids, attention_masks, labels)
val_dataset = TensorDataset(input_ids_val, attention_masks_val, labels_val)

# The DataLoader needs to know our batch size for training, so we specify it
# here. For fine-tuning BERT on a specific task, the authors recommend a batch
# size of 16 or 32.
batch_size = 32

# Create the DataLoaders for our training and validation sets.
# We'll take training samples in random order.
train_dataloader = DataLoader(
            train_dataset,  # The training samples.
            sampler = RandomSampler(train_dataset), # Select batches randomly
            batch_size = batch_size # Trains with this batch size.
        )

# For validation the order doesn't matter, so we'll just read them sequentially.
validation_dataloader = DataLoader(
            val_dataset, # The validation samples.
            sampler = SequentialSampler(val_dataset), # Pull out batches sequentially.
            batch_size = batch_size # Evaluate with this batch size.
        )
```

Next, we need to load the pre-trained BERT model as a regressor. Here we use the *BertForSequenceClassification* class and set the number of labels to be 1, which actually makes it a regressor. The model is set to store double values for a regression task.

```python
# modified from notebook in https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FIc

from transformers import BertForSequenceClassification, AdamW, BertConfig

model = BertForSequenceClassification.from_pretrained(
    'bert-base-uncased', # Use the 12-layer BERT model, with an uncased vocab.
    num_labels = 1,
    output_attentions = False, # Whether the model returns attentions weights.
    output_hidden_states = False, # Whether the model returns all hidden-states.
)

# Tell pytorch to run this model on the GPU.
model.cuda()
```

```
model.cuda()

model = model.double()
```

6.py hosted with ♡ by GitHub view raw

Parameters are set as follows. Here we use a learning rate of 2e-5, adam episilon of 1e-8, and 4 epochs.

```
# modified from notebook in https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FIc

from transformers import get_linear_schedule_with_warmup

optimizer = AdamW(model.parameters(),
                  lr = 2e-5, # args.learning_rate - default is 5e-5,
                  eps = 1e-8 # args.adam_epsilon  - default is 1e-8.
                )

# Number of training epochs. The BERT authors recommend between 2 and 4.
epochs = 4

# Total number of training steps is [number of batches] x [number of epochs].
# (Note that this is not the same as the number of training samples).
total_steps = len(train_dataloader) * epochs

# Create the learning rate scheduler.
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0, # Default value in run_glue.p
                                            num_training_steps = total_steps)
```

7.py hosted with ♡ by GitHub view raw

Create a function to format time for later use.

```
# modified from notebook in https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FIc

import time
import datetime

def format_time(elapsed):
    '''
    Takes a time in seconds and returns a string hh:mm:ss
    '''
    # Round to the nearest second.
    elapsed_rounded = int(round((elapsed)))

```

```
    # Format as hh:mm:ss
    return str(datetime.timedelta(seconds=elapsed_rounded))
```

8.py hosted with ♡ by GitHub view raw

Finally, we can train the regressor. A specific seed value is set so that the result might be reproduced. After every, the RMSE(Root Mean Squared Error) score on the validation set is evaluated to show the performances.

```
# modified from notebook in https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FI

import random
from sklearn.metrics import mean_squared_error

# This training code is based on the `run_glue.py` script here:
# https://github.com/huggingface/transformers/blob/5bfcd0485ece086ebcbed2d008813037968a9e58/ex

# Set the seed value all over the place to make this reproducible.
seed_val = 42

random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)

# We'll store a number of quantities such as training and validation loss,
# validation accuracy, and timings.
training_stats = []

# Measure the total training time for the whole run.
total_t0 = time.time()

# For each epoch...
for epoch_i in range(0, epochs):

    # # ========================================
    # #               Training
    # # ========================================

    # Perform one full pass over the training set.

    print("")
    print('======== Epoch {:} / {:} ========'.format(epoch_i + 1, epochs))
    print('Training...')

    # Measure how long the training epoch takes.
```

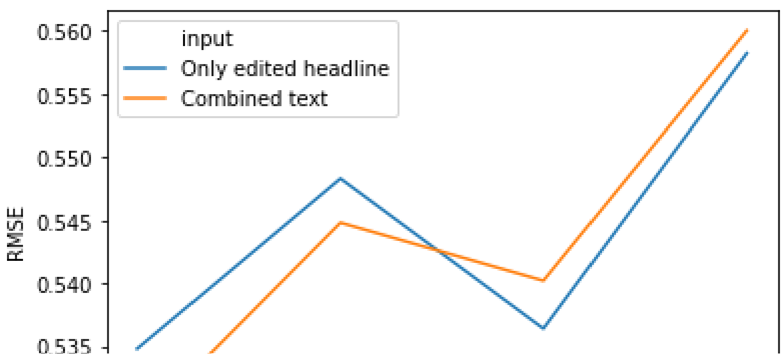
```
    t0 = time.time()

    # Reset the total loss for this epoch.
    total_train_loss = 0

    # Put the model into training mode. Don't be mislead--the call to
    # `train` just changes the *mode*, it doesn't *perform* the training.
    # `dropout` and `batchnorm` layers behave differently during training
    # vs. test (source: https://stackoverflow.com/questions/51433378/what-does-model-train-do-
    model.train()

    # For each batch of training data...
    for step, batch in enumerate(train_dataloader):

        # Progress update every 40 batches.
        if step % 40 == 0 and not step == 0:
            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)

            # Report progress.
            print('  Batch {:>5,}  of  {:>5,}.    Elapsed: {:}.'.format(step, len(train_datalo

        # Unpack this training batch from our dataloader.
        #
        # As we unpack the batch, we'll also copy each tensor to the GPU using the
        # `to` method.
        #
        # `batch` contains three pytorch tensors:
        #   [0]: input ids
        #   [1]: attention masks
        #   [2]: labels
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)

        # Always clear any previously calculated gradients before performing a
        # backward pass. PyTorch doesn't do this automatically because
        # accumulating the gradients is "convenient while training RNNs".
        # (source: https://stackoverflow.com/questions/48001598/why-do-we-need-to-call-zero-gr
        model.zero_grad()

        # Perform a forward pass (evaluate the model on this training batch).
        # The documentation for this `model` function is here:
        # https://huggingface.co/transformers/v2.2.0/model_doc/bert.html#transformers.BertForS
        # It returns different numbers of parameters depending on what arguments
        # arge given and what flags are set. For our useage here, it returns
        # the loss (because we provided labels) and the "logits"--the model
```

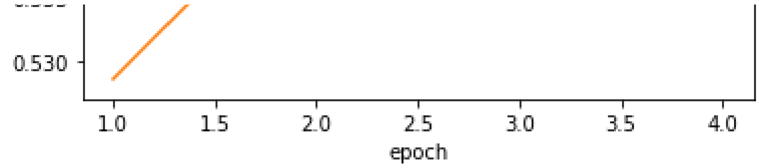
```
        # outputs prior to activation.
        loss, logits = model(b_input_ids,
                             token_type_ids=None,
                             attention_mask=b_input_mask,
                             labels=b_labels)

        # Accumulate the training loss over all of the batches so that we can
        # calculate the average loss at the end. `loss` is a Tensor containing a
        # single value; the `.item()` function just returns the Python value
        # from the tensor.
        total_train_loss += loss.item()

        # Perform a backward pass to calculate the gradients.
        loss.backward()

        # Clip the norm of the gradients to 1.0.
        # This is to help prevent the "exploding gradients" problem.
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # Update parameters and take a step using the computed gradient.
        # The optimizer dictates the "update rule"--how the parameters are
        # modified based on their gradients, the learning rate, etc.
        optimizer.step()

        # Update the learning rate.
        scheduler.step()

    # Calculate the average loss over all of the batches.
    avg_train_loss = total_train_loss / len(train_dataloader)

    # Measure how long this epoch took.
    training_time = format_time(time.time() - t0)

    print("")
    print("  Average training loss: {0:.2f}".format(avg_train_loss))
    print("  Training epcoh took: {:}".format(training_time))

    # ========================================
    #               Validation
    # ========================================
    # After the completion of each training epoch, measure our performance on
    # our validation set.

    print("")
    print("Running Validation...")

    t0 = time.time()

```

```
        # Put the model in evaluation mode--the dropout layers behave differently
        # during evaluation.
        model.eval()

        # Tracking variables
        total_eval_accuracy = 0
        total_eval_loss = 0
        nb_eval_steps = 0

        y_pred = np.array([])
        y_true = np.array([])

        # Evaluate data for one epoch
        for batch in validation_dataloader:

            # Unpack this training batch from our dataloader.
            #
            # As we unpack the batch, we'll also copy each tensor to the GPU using
            # the `to` method.
            #
            # `batch` contains three pytorch tensors:
            #   [0]: input ids
            #   [1]: attention masks
            #   [2]: labels
            b_input_ids = batch[0].to(device)
            b_input_mask = batch[1].to(device)
            b_labels = batch[2].to(device)

            # Tell pytorch not to bother with constructing the compute graph during
            # the forward pass, since this is only needed for backprop (training).
            with torch.no_grad():

                # Forward pass, calculate logit predictions.
                # token_type_ids is the same as the "segment ids", which
                # differentiates sentence 1 and 2 in 2-sentence tasks.
                # The documentation for this `model` function is here:
                # https://huggingface.co/transformers/v2.2.0/model_doc/bert.html#transformers.Bert
                # Get the "logits" output by the model. The "logits" are the output
                # values prior to applying an activation function like the softmax.
                (loss, logits) = model(b_input_ids,
                                       token_type_ids=None,
                                       attention_mask=b_input_mask,
                                       labels=b_labels)

            # Accumulate the validation loss.
            total_eval_loss += loss.item()

            # Move logits and labels to CPU
```

```
        # Move logits and labels to CPU
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()
        y_pred = np.append(y_pred,logits)
        y_true = np.append(y_true,label_ids)

    rmse = mean_squared_error(y_true, y_pred, squared=False)
    print("  RMSE: {0:.4f}".format(rmse))

    # Calculate the average loss over all of the batches.
    avg_val_loss = total_eval_loss / len(validation_dataloader)

    # Measure how long the validation run took.
    validation_time = format_time(time.time() - t0)

    print("  Validation Loss: {0:.2f}".format(avg_val_loss))
    print("  Validation took: {:}".format(validation_time))

    # Record all statistics from this epoch.
    training_stats.append(
        {
            'epoch': epoch_i + 1,
            'Training Loss': avg_train_loss,
            'Valid. Loss': avg_val_loss,
            'Valid. RMSE.': rmse,
            'Training Time': training_time,
            'Validation Time': validation_time
        }
    )

```

help improve the performance.





To evaluate how good (or bad) the result is, we compare it with a model that always returns the mathematical expectation value of the mean grades based on the distribution on the training dataset. The mathematical expectation value is calculated to be approximately 0.9356, and the RMSE score of this baseline model on the validation dataset is around 0.58.

```
from collections import Counter
from sklearn.metrics import mean_squared_error

c = Counter()

for x in train_df['meanGrade']:
  c[x] += 1

c = {k:c[k]/len(train_df['meanGrade']) for k in c}

m_e = 0
for x in c:
  m_e += x * c[x]
# m_e = 0.9355712114933005

print(mean_squared_error(val_df['meanGrade'],[m_e]*len(val_df),squared=False))
# 0.5783998503042385
```

10.py hosted with ♡ by GitHub view raw

The RMSE of around 0.53 achieved by the BERT regressor is significantly better than the baseline score of around 0.58. The BERT model seems to be able to detect some humor from given text.

However, the performance is still not so satisfactory. One reason might come from the limited range and dense distribution of the actual mean grades in the dataset. For a better model, we might need some better datasets with some larger grade range and some more uniform distribution on the grades.

What’s Next

When applying the BERT model in this article, the regressor is directly trained from the given pre-trained language model (trained with Wikipedia text). The language model can be further pre-trained with news headlines which might help improve the performance.

Also, we might try other methods to add some extra features to the BERT model. Adding features like POS of the edited words and similarities between the edited and original words might help.

About Help Legal

Get the Medium app

Download on the App Store

GET IT ON Google Play