Paper SAS4429-2020

NLP with BERT: Sentiment Analysis Using SAS® Deep Learning and DLPy

Doug Cairns and Xiangxiang Meng, SAS Institute Inc.

ABSTRACT

A revolution is taking place in natural language processing (NLP) as a result of two ideas. The first idea is that pretraining a deep neural network as a language model is a good starting point for a range of NLP tasks. These networks can be augmented (layers can be added or dropped) and then fine-tuned with transfer learning for specific NLP tasks. The second idea involves a paradigm shift away from traditional recurrent neural networks (RNNs) and toward deep neural networks based on Transformer building blocks. One architecture that embodies these ideas is Bidirectional Encoder Representations from Transformers (BERT). BERT and its variants have been at or near the top of the leaderboard for many traditional NLP tasks, such as the general language understanding evaluation (GLUE) benchmarks. This paper provides an overview of BERT and shows how you can create your own BERT model by using SAS® Deep Learning and the SAS DLPy Python package. It illustrates the effectiveness of BERT by performing sentiment analysis on unstructured product reviews submitted to Amazon.

INTRODUCTION

Providing a computer-based analog for the conceptual and syntactic processing that occurs in the human brain for spoken or written communication has proven extremely challenging. As a simple example, consider the abstract for this (or any) technical paper. If well written, it should be a concise summary of what you will learn from reading the paper. As a reader, you expect to see some or all of the following:

- Technical context and/or problem
- Key contribution(s)
- Salient result(s)

If you were tasked to create a computer-based tool for summarizing papers, how would you translate your expectations as a reader into an implementable algorithm? This is the type of problem that the field of natural language processing (NLP) addresses. NLP encompasses a wide variety of issues in both spoken and written communication. The field is quite active, because many NLP problems do not have solutions that approach human performance.

Historically, NLP practitioners focused on solutions with problem-specific, handcrafted features that relied on expert knowledge. There was a shift starting in the early 2000s (Bengio et al. 2003) to data-driven, neural network–based solutions that learned features automatically. During this shift, a key idea emerged: training a neural network to function as a language model is a good foundation for solving a range of NLP problems (Collobert et al. 2011). This neural network could either provide context-sensitive features to augment a task-specific solution (Peters et al. 2018) or be fine-tuned to solve a specific NLP problem (Radford 2018; Devlin et al. 2018). Both approaches were extremely successful and led to speculation (Ruder 2018) that an NLP "ImageNet" moment was at hand. The sense of the speculation was that neural network–based NLP solutions were approaching or exceeding human performance, as they had in the field of image processing. The neural network advances in image processing were inspired by the ImageNet Large Scale Visual Recognition Challenge (image-net.org 2012); hence the notion of an "ImageNet" moment.

Until recently, most neural network–based NLP approaches focused on recurrent neural networks (RNNs). Unlike other types of neural networks, RNNs consider data ordering, so an RNN is well suited for text or audio data, where order matters. For each element in a sequence, the output of an RNN depends on the current element as well as state information. The state information is a function of the sequence element(s) previously observed and is updated for each new element. This enables the RNN to "remember" and thus learn how sequential data evolve over time. The state calculation also makes an RNN difficult to efficiently implement, so training can be extremely time-consuming.

In 2017, the dominance of the RNN approach was challenged by the Transformer architecture (Vaswani et al. 2017). The Transformer is based on an attention mechanism. You can think of an attention mechanism as an adaptive weighting scheme. The output of an attention mechanism for sequence element n is a weighted sum of all the input sequence elements. The "adaptive" part refers to the fact that weights are trained for each sequence position. Attention (Transformer) differs from recurrence (RNN) in that all sequence elements are considered simultaneously. This approach has both performance and implementation advantages.

Bidirectional Encoder Representations from Transformers (BERT) combines language model pretraining and the Transformer architecture to achieve impressive performance on an array of NLP problems. Subsequent sections present an overview of BERT and a tutorial on how to build and train a BERT model using SAS Deep Learning actions and DLPy.

BERT OVERVIEW

This overview presents BERT from the perspective of an NLP practitioner—that is, someone primarily interested in taking a pretrained BERT model and using transfer learning to fine-tune it for a specific NLP problem. The key considerations for an NLP practitioner are the input data representation, the model architecture, and keys for successful transfer learning, all of which are discussed in the following sections.

INPUT REPRESENTATION

Neural networks cannot operate directly on raw text data, so a standard practice in NLP is to tokenize the text (that is, split the text into meaningful phrase, word, or subword units) and then replace each token with a corresponding numeric embedding vector. BERT follows this standard practice but does so in a unique manner. There are three related representations required by BERT for any text string. The first is the tokenized version of the text. The second is the position of the token within the text string, which is something BERT inherits from the Transformer. The third is whether a given token belongs to the first sentence or the second sentence. This last representation makes sense only if you understand the BERT training objectives. A BERT model is trained to perform two simultaneous tasks:

- *Masked language model*. A fraction of tokens is masked (replaced by a special token), and then those tokens are predicted during training.
- *Next sentence prediction (NSP)*. Two sentences are combined, and a prediction is made as to whether the second sentence follows the first sentence.

The NSP task requires an indication of token/sentence association; hence the third representation. Both training objectives require special tokens ([CLS], [SEP], and [MASK]) that indicate classification, separation, and masking, respectively. In practice, this means that BERT text input is decomposed to three related values for each token. Figure 1 shows the three values for all tokens in this simple example:

"What is the weather forecast? Rain is expected."

Text	What is the weather forecast? Rain is expected.													
Token	[CLS]	what	is	the	weather	fore	##cast	?	[SEP]	rain	is	expected	.	[SEP]
Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Segment	A	A	A	A	A	A	A	A	A	B	B	B	B	B

Figure 1: BERT input for example

The example illustrates several important points. First, a classification token ([CLS]) begins every tokenized string, and separation tokens ([SEP]) conclude every sentence. Second, some words (such as *forecast*) can be split as part of the tokenization process. This is normal and follows the rules of the WordPiece model that BERT employs. Finally, you see that tokens in the first sentence are associated with segment A, while tokens in the second sentence are associated with segment B. In the case of a single sentence, all tokens are associated with segment A.

ARCHITECTURE

A BERT model has three main sections, as shown in Figure 2. The lowest layers make up the embedding section, which is composed of three separate embedding layers followed by an addition layer and a normalization layer. The next section is the Transformer encoder section, which typically consists of N encoder blocks connected sequentially (that is, output of encoder block 1 connects to input of encoder block 2, . . . , output of encoder block $N-1$ connects to input of encoder block N). Figure 2 shows $N=1$ for ease of illustration. The final section is customized for a specific task and can consist of one or more layers.

Embedding

The embedding section maps the three input values associated with each token in the tokenized text to a corresponding embedding vector. The embedding vectors are then summed and normalized (Ba, Kiros, and Hinton 2016). Here, the term *maps* refers to the process of using the token, position, or segment value as a key to extract the corresponding embedding vector from a dictionary or lookup table.

The token embedding layer maps token input values to a WordPiece embedding vector. The token embedding table/dictionary contains slightly more than 30,000 entries. The position embedding layer maps the position input values to a position embedding vector. Note that the position input value can be no greater than 512, so the position embedding table/dictionary contains 512 entries. The position value restriction also limits the length of the raw text string. The segment embedding layer maps the segment input value to one of two segment embedding vectors. All embedding vectors are of dimension D, where D is typically 768 or greater.

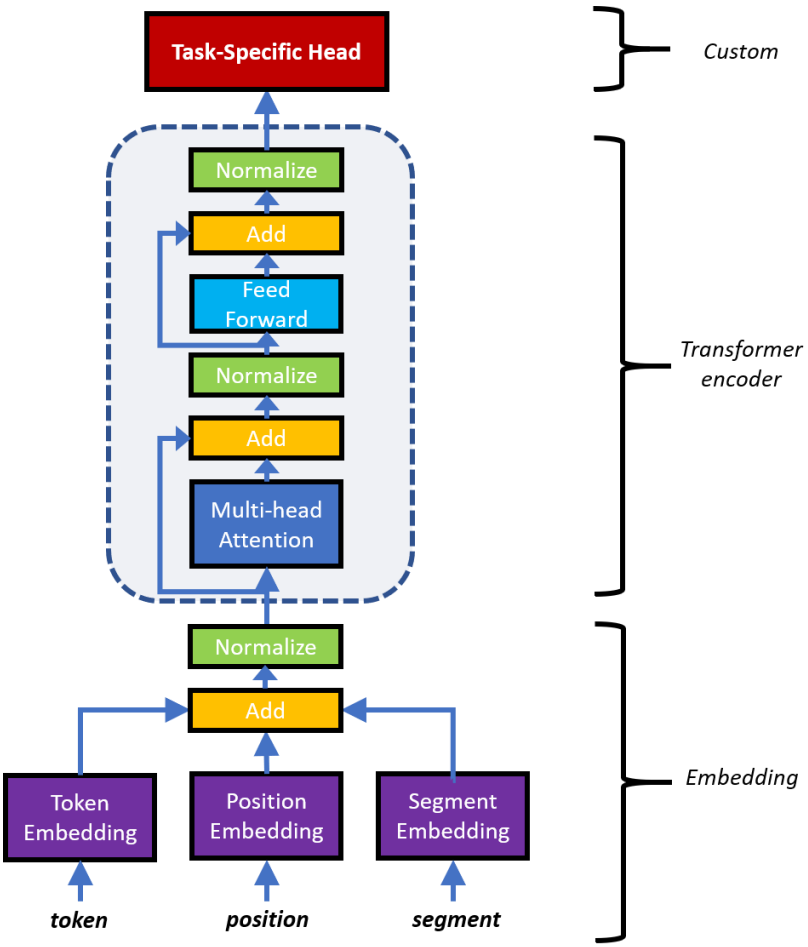

Figure 2: Simplified BERT model

Transformer Encoder

The encoder block consists of layers already encountered in the embedding section—namely, addition and normalization. There are also two composite layers, called feedforward and multi-head attention. The feedforward layer consists of two back-to-back fully connected (or dense) layers, where each input neuron is connected to all output neurons. Multi-head attention, shown in Figure 3, is more complex; there are three features of this composite layer to highlight.

First, you can see that multi-head attention requires three inputs, shown in the figure as $\mathbf{X_Q}$, $\mathbf{X_K}$, and $\mathbf{X_V}$. These are matrices, where each row is a vector of dimension *D* that corresponds to a token from a tokenized text string. The first row represents the first token in the string, the second row represents the second token, and so on. In Transformer terminology, $\mathbf{X_Q}$, $\mathbf{X_K}$, and $\mathbf{X_V}$ are the *query*, *key*, and *value* inputs, respectively. Whereas multi-head attention requires three inputs, Figure 2 shows only a single input to the encoder. This is because the BERT encoder uses self-attention (that is, $\mathbf{X_Q} = \mathbf{X_K} = \mathbf{X_V}$).

Second, notice that there are multiple attention "heads." An attention head projects the *query*, *key*, and *value* inputs to independent lower-dimensional subspaces. This is followed by scaled dot-product attention. BERT builds on the Transformer, and the rationale for multiple heads is given in a comment in the original Transformer paper (Vaswani et al. 2017): "Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this."

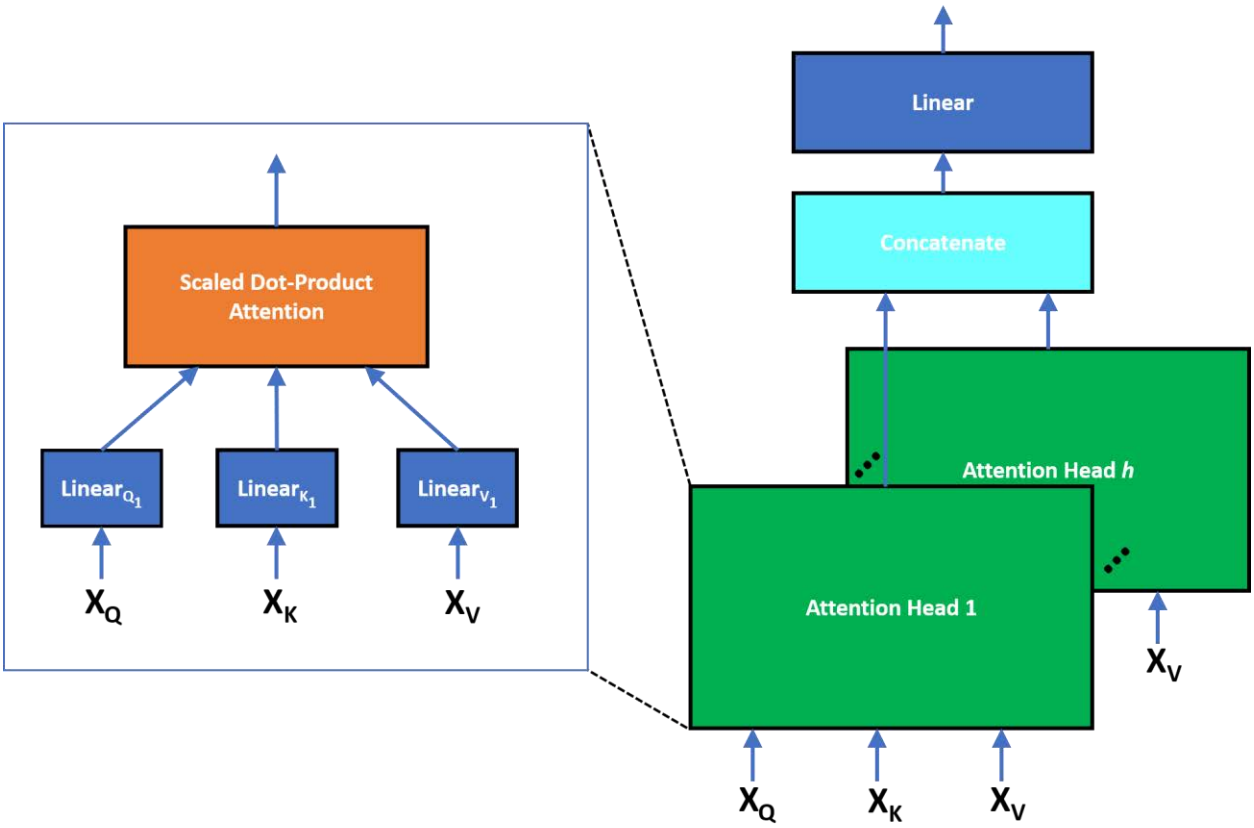

Figure 3: Multi-head attention

The final feature to highlight is the scaled dot-product attention mechanism. To gain some insight, consider the defining attention equation

$$Attention(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) = \frac{softmax\left(\mathbf{Q}_i \mathbf{K}_i^T\right)}{\sqrt{d}} \mathbf{V}_i$$

where the *softmax* operator applies to each row of the matrix product $\mathbf{Q}_i \mathbf{K}_i^T$. The scaling term d is the dimension of the subspace. The subspace dimension is equal to D/H, where H is the number of attention heads. For a vector $\mathbf{x}$, the *softmax* operator returns a vector, the mth element of that vector, given by

$$softmax(x_m) = \frac{e^{-x_m}}{\sum_j e^{-x_j}}$$

Now consider what the attention equation computes. For the ith attention head, matrices $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ are d-dimension subspace projections of matrices $\mathbf{X_Q}$, $\mathbf{X_K}$, and $\mathbf{X_V}$:

$$\mathbf{Q}_i = \mathbf{X_Q} \mathbf{W}_{\mathbf{Q}_i}$$
$$\mathbf{K}_i = \mathbf{X_K} \mathbf{W}_{\mathbf{K}_i}$$
$$\mathbf{V}_i = \mathbf{X_V} \mathbf{W}_{\mathbf{V}_i}$$

Focusing on $softmax(\mathbf{Q}_i \mathbf{K}_i^T)\mathbf{V}_i$, notice that row m of matrix $\mathbf{Q}_i \mathbf{K}_i^T$ is the cross-correlation of the *query* subspace token at position m with all *key* subspace tokens for a given tokenized string. After the *softmax* operation, the row m cross-correlation values become a set of weights for combining all the *value* subspace tokens to create a new subspace token m representation. Here you see the concept of attention at work: the new token m is most influenced by (pays the most attention to) those *value* subspace tokens with large weights

and is least influenced by (generally ignores) those *value* subspace tokens with small weights.

Scaled dot-product attention concludes each attention head, but there is a final step that allows each head in the encoder to contribute to the new *D*-dimensional representation for each token. The encoder applies a fully connected layer to the concatenated output of all attention heads, mixing the token representations from the *value* subspaces for each head.

Task-Specific Layer(s)

The custom section of the BERT model is tailored to a task, so a general overview is difficult. However, because classification-type scenarios arise in many NLP situations, a simple example is possible. For classification scenarios, the task-specific head is just a fully connected layer, in which the number of output neurons is equal to the number of classes. If the classification task is something like sentiment analysis, then the classification decision uses the output of the fully connected layer associated with the [CLS] token. If the classification task is token-specific (such as named entity recognition), then the classification decision uses the output of the fully connected layer associated with the token(s) in question.

TRANSFER LEARNING

Transfer learning describes the following process:

1. Obtaining an appropriate model with pretrained parameters
2. Removing layer(s) specific to the original model objective
3. Adding layer(s) specific to the new model objective
4. Performing fine-tuning training to optimize model parameters

In general, steps 1 and 4 are the key steps in the process. There are multiple sources for pretrained BERT models, and any of them might be suitable for your application. One popular source is the HuggingFace *transformers* GitHub repository. There you will find many BERT pretrained models for a variety of scenarios (English-only, multilingual, and so on). You should choose the model that best matches your scenario. After selecting a BERT model, you must then fine-tune it with data specific to your problem. Like the original model training, the fine-tuning training requires the selection of an optimization algorithm along with associated training hyperparameters. This can be a time-consuming exercise, but fortunately, Devlin et al. (2018) provide some helpful suggestions. The Adam optimization algorithm (Kingma and Ba 2015) is recommended with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Also recommended are the hyperparameter settings shown in Table 1.

Hyperparameter	Setting
Dropout	0.1
Batch size	8, 16, 32
Learning rate	$2 \times 10^{-5}, 3 \times 10^{-5}, 5 \times 10^{-5}$
Number of epochs	2, 3, 4

Table 1: Recommended hyperparameter settings for fine-tuning

TUTORIAL: FINE-TUNING A BERT MODEL

You can use SAS Deep Learning and the SAS DLPy Python package to build and fine-tune a BERT model. To illustrate this process, consider performing sentiment analysis on an Amazon review data set with a BERT classification model. This tutorial example walks through the five steps you must perform and concludes with an evaluation of the trained model.

PREREQUISITES

This example assumes the following:

- The Amazon Fine Food Reviews data set from the Kaggle competition website is available on the client computer.
- You have a working understanding of Python.
- SAS Scripting Wrapper for Analytics Transfer (SWAT) is available on the client computer (clone, fork, or install from here).
- SAS DLPy is available on the client computer (clone, fork, or install from here).
- You have a working Python environment on the client computer that includes the *transformers* package from the HuggingFace GitHub repository and PyTorch. See the recommendations here for setting up a suitable Anaconda environment.
- A SAS® Viya® 3.5 server is set up and running.
- An active Viya session is running in the Python environment.

Note that SAS Viya is a client-server architecture, and there are references to both client and server in the preceding list. For the purposes of this tutorial, consider the Viya client to be a desktop PC and the Viya server to be a separate computer.

STEP 1: CREATE BERT CLASSIFICATION MODEL

The first step is to create a DLPy model object that encapsulates the BERT classification model. Start by defining the BERT cache directory, and then instantiate an object of the *BERT_Model* class. The *BERT_Model* object looks in the cache directory for BERT model definition and parameter information. If this information is absent, it will be downloaded from the HuggingFace repository:

```
from dlpy.transformers import BERT_Model

cache_dir = 'path/to/your/cache-dir'

bert = BERT_Model(viya_conn,
                  cache_dir,
                  'bert-base-uncased',
                  2,
                  num_hidden_layers = 12,
                  max_seq_len = 256,
                  verbose = True)
```

Note the *viya_conn* variable. This is the SWAT connection to the active Viya session referred to earlier.

STEP 2: PREPARE DATA

The second step is to prepare your data for training. Begin by reading the data from the Amazon Fine Food Reviews data set into a Pandas DataFrame:

```
import pandas as pd
reviews = pd.read_csv('name-of-your-amazon-review-file',
                      header=0,
                      encoding='utf-8')
```

Then assign a numeric value to indicate positive or negative sentiment for each review. Since the number of stars associated with each review ranges from 1 to 5, filter out the neutral reviews (3 stars), and then assign a negative label to 1- and 2-star reviews and a positive label to 4- and 5-star reviews:

```
t_idx = reviews["Score"] != 3
inputs = reviews[t_idx]["Text"].to_list()
targets = reviews[t_idx]["Score"].to_list()

for ii,val in enumerate(targets):
    inputs[ii] = inputs[ii].replace("<br />","")
    if (val == 1) or (val == 2):   # negative reviews
        targets[ii] = 1
    elif (val == 4) or (val == 5): # positive reviews
        targets[ii] = 2
```

Finally, import the data preparation helper function. Invoking the helper function tokenizes the review data and creates the three input values (token, position, segment) associated with each token as well as the sentiment target for each review. The helper function also automatically creates a Viya table or tables containing the prepared data. The following invocation splits the reviews into a training set that has 80% of the overall data and a testing set that contains the remaining data:

```
from dlpy.transfomers.bert_utils import bert_prepare_data

num_tgt_var, train, test = bert_prepare_data(viya_conn,
                                             bert.get_tokenizer(),
                                             input_a=inputs,
                                             target=targets,
                                             train_fraction=0.8,
                                segment_vocab_size=bert.get_segment_size(),
                                classification_problem=bert.get_problem_type(),
                                             verbose=True)
```

STEP 3: CREATE SAS VIYA BERT MODEL

The third step is to create a SAS Deep Learning model that is the equivalent of the base BERT model plus a classification (fully connected) layer. The DLPy BERT model object provides a convenient function that performs this step for you:

```
bert.compile(num_target_var=num_tgt_var)
```

Note that this function does more than just create a BERT model. It also reads the trained parameters from the HuggingFace BERT model and saves them as an HDF5 file on the client computer. This file has a predefined structure that the SAS Deep Learning actions expect.

STEP 4: ATTACH MODEL PARAMETERS

The fourth step is to attach the trained model parameters stored in the HDF5 file to the BERT model. SAS Deep Learning actions read this HDF5 file, so it must be accessible by the Viya server. Because the client computer is assumed to be separate from the server computer, copy or move the HDF5 file to a location where it is visible to the server:

```
import os
from shutil import copyfile

server_dir = 'path/to/your/server-directory'

copyfile(os.path.join(cache_dir,'bert-base-uncased.kerasmodel.h5'),
         os.path.join(server_dir,'bert-base-uncased.kerasmodel.h5'))
```

This example assumes that even though the client and server computers are distinct, they share a common file system. If that weren't true, then some other means of moving the file from the client to the server (such as FTP) would be required. When the file has been successfully copied or moved, invoke the *load_weights()* function exposed by the DLPy BERT model object to attach parameters:

```
bert.load_weights(server_dir+'/bert-base-uncased.kerasmodel.h5',
                  num_target_var=num_tgt_var,
                  freeze_base_model=False)
```

The parameter *freeze_base_model* controls the training of the BERT model. It is set to False here; this allows training of all layers of the new BERT model. If you set it to True, then only the final classification layer could be trained. In that case, all other model parameters would be fixed.

STEP 5: FINE-TUNE BERT CLASSIFICATION MODEL

The final step is to perform fine-tuning training. Recall the fine-tuning recommendations provided by Devlin et al. (2018). The Adam optimizer and a default set of hyperparameters are defined for you when you invoke the *load_weights()* function in step 4. If the defaults are acceptable, you can invoke the *fit()* function exposed by the DLPy BERT model object. If you want to override one or more of the defaults, you can call the *set_optimizer_parameters()* function before invoking *fit()*, as follows:

```
bert.set_optimizer_parameters(learning_rate=2e-5)

bert.fit(train,
         data_specs= bert.get_data_spec(num_tgt_var),
         optimizer=bert.get_optimizer(),
         text_parms=bert.get_text_parameters(),
         seed=12345,
         n_threads=32))
```

When the *fit()* function finishes executing, your BERT model is fine-tuned for sentiment analysis.

MODEL EVALUATION

You can now evaluate your fine-tuned sentiment analysis model by using the test data set. The *predict()* function exposed by the DLPy BERT model object provides a convenient method for this evaluation:

```
res = bert.predict(test,
                   text_parms=bert.get_text_parameters())


print(res['ScoreInfo'])
```

The results of the print function are shown in Figure 4. You can see that after three epochs of fine-tuning, the model achieves slightly better than 98.0% accuracy on the test data.

	Descr	Value
0	Number of Observations Read	105205
1	Number of Observations Used	105205
2	Misclassification Error (%)	1.994202
3	Loss Error	0.066576

Figure 4: Evaluation results

CONCLUSION

The field of natural language processing is undergoing a revolution, thanks to the ideas of language model pretraining and attention. BERT brings these concepts together to enable a powerful paradigm based on Transformer building blocks: simple fine-tuning using transfer learning provides state-of-the-art performance in many NLP tasks. This paper provides an NLP practitioner with an overview of key aspects of BERT and shows how to fine-tune and evaluate a BERT sentiment analysis model using SAS Deep Learning and the SAS DLPy Python package.

REFERENCES

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). "Layer Normalization." *arXiv:1607.06450v1*, 1–14.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). "A Neural Probabilistic Language Model." *Journal of Machine Learning Research* 3:1137–1155.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). "Natural Language Processing (Almost) from Scratch." *Journal of Machine Learning Research* 12:2493–2537.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." *arXiv:1810.04805v1*, 1–14.

Image-net.org. (2012). "ImageNet Large Scale Visual Recognition Challenge." Accessed December 9, 2019. http://image-net.org/challenges/LSVRC/.

Kingma, D. P., and Ba, J. (2015). "Adam: A Method for Stochastic Optimization." 3rd International Conference for Learning Representations. San Diego: ICLR.

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). "Deep Contextualized Word Representations." *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics,* 2227–2237. New Orleans: Association for Computational Linguistics.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). "Improving Language Understanding with Unsupervised Learning." *OpenAI Technical Report*.

Ruder, S. (2018). "NLP's ImageNet Moment Has Arrived." Accessed December 9, 2019, https://ruder.io/nlp-imagenet.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). "Attention Is All You Need." *Advances in Neural Information Processing Systems 30*. NIPS 2017, Long Beach, CA.

RECOMMENDED READING

- *The Illustrated Transformer* (blog)
- *BERT for Dummies—Step by Step Tutorial* (blog)
- *HuggingFace Quickstart* (Transformers documentation)

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors:

Doug Cairns
doug.cairns@sas.com

Xiangxiang Meng
xiangxiang.meng@sas.com