Text Classification Competition
Team: West Coasters

Table of Contents

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

Overview

The project is about contextual text classification, where the task is to classify response to a tweet being sarcastic or not. It is a Natural Language Processing problem. Though the models in this document will talk only about Sarcasm detection, the same code/models can be used for any binary classification of NLP problems, where there are only two label classes. The same solution can be applied to problems like:

- Spam detection, where text has to be labeled either as Spam or Not Spam
- Classifying product reviews to be helpful or not
- Classifying movie reviews to be positive or negative

Classification is a task that requires Machine Learning algorithms/models that can be trained to label from the training dataset and can predict based on the learning. As input to the project, we are given training dataset which we used for training the models we created. As there are many machine learning algorithms that can perform classification, in our initial proposal we have considered exploring Linear Regression, Decision Tree, SVM and, RNN. From the comments we received on proposal, we decided to explore more on neural networks.

Neural Networks

A neural network is a series of algorithms that try to learn the relation between the input data and the input label during training and based on the learning predict the labels for test inputs. Neural networks can adapt to changing input; so, the neurons generate the best possible results without needing to redesign the output criteria.

On a lower-level neural networks are nothing but simple combination of mathematical operations. Each network consists of sequence of layers I which data passes through. Each unit in these networks is called a neuron. These neurons pass the data from one layer to neurons in next layer. There are some default weights to the connections between the neurons which can be modified while defining the model. Extra bias also could be built around these connections and neurons. Each neuron holds only a single numeric value. The data starts at the input layer and is transformed as it passes through subsequent layers. For more information on neural networks refer https://arxiv.org/pdf/1901.05639.pdf.

For the assignment we experimented with following models:

1. Multichannel CNN
2. Bert
3. DNN Classifier
4. Bidirectional LSTM

Following section will cover more details on our experiments. Code for every model follows the steps as:

1. Load dataset, loads dataset from training and test files
2. Split dataset, split dataset into training and evaluation datasets
3. Preprocessing, convert data in the format expected by model
4. Define model, create multi-layer neural network either using pre-existing classifiers or using
5. Training, train the model with training datasets
6. Evaluation, evaluate the model using evaluation dataset
7. Prediction, predict the labels for test dataset

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

Models

Multichannel CNN

Model overview

CNN are considered to be one of the model types, which are used for the text classification. During feasibility study our team found a couple of articles, which recommend the use of CNN for text classification, and specifically for the sentiment classification.

Some examples of the CNN for text classification:

- Very Deep Convolutional Networks for Text Classification, Alexis Conneau, Holger Schwenk, Yann Le Cun, Lo ̈ıc Barrault, 2017. The work claims that the error rate on the test set was not higher than 37%. The work uses the text representation in the smallest possible parts - characters. The model, used in this work, contains 29 layers, including convolutional layers of different sizes, and max-pooling layers. The researchers designed it, being inspired by the success of convolutional network architecture in computer vision.
- The article at https://machinelearningmastery.com/develop-n-gram-multichannel-convolutional-neural-network-sentiment-analysis/ by Jason Brownlee, describing multichannel convolutional model. This model was operating on words as a lexical unit, and showed 87.5% of accuracy on the test set. The model has different kernel sizes in the different convolutional channels, letting it read the text in those channels with different n-gram sizes. Due to high declared accuracy, and due to its logic of getting info simultaneously from different n-grams for the same piece of text, this model looked promising to try in our sarcasm text classification project.

As for our team the neural networks, and the CNN architecture, was new, we have spent a good piece of time to study how it works, what can be the hyperparameters which we will need to change to tune it, and how to supply the data to this model.

The code, which was produced in the result, takes two files:

- Jasonl training file, which can contain any number of lines (in real file 5000 lines) with following fields in each: “labels”, “context”, “response”. It is considered that field “labels” and “response” have str type, and field “context” is a list of 2 str type entries.
- Jasonl test file, which can contain any number of lines (in real file 1800 lines) with following fields in each: “id”, “context”, “response”. It is considered that field “id” and “response” have str type, and field “context” is a list of 2 str type entries.

It outputs the txt file, where on each line it writes the id of the observation, and the predicted label name for it (“SARCASM” or “NOT_SARCASM”). The number of lines and id numbers correspond to the jasonl test input file.

Model code

First, we create two functions, which import the data from jsonl file to a tuple of 3 lists: “labels”, “context”, “response” for training file, and “id”, “context”, “response” for test file. Since the “context” field has the type list of strings in both files, we convert it to the string type right inside those two functions:

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

For training data:

```
#Function to create a train tuple of lists of Labels/Context/Response from json
def getdatajson(path):
    with open(path, "r") as file_data:
        file_r = file_data.read()
        response = []
        context = []
        labels = []
        for line in file_r.splitlines():
            document = json.loads(line)
            labels.append(document["label"])
            intermediate = str(" ".join(document["context"]))
            context.append(intermediate)
            response.append(document["response"])

    return labels, context, response
```

For test data:

```
#Function to create a test tuple of lists of Index/Context/Response from json
def getdatatest(path):
    with open(path, "r") as file_data:
        file_r = file_data.read()
        test_cont = []
        test_resp = []
        test_idx = []
        for line in file_r.splitlines():
            document = json.loads(line)
            intermediate = str(" ".join(document["context"]))
            test_cont.append(intermediate)
            test_resp.append(document["response"])
            test_idx.append(document["id"])
    return test_idx, test_cont, test_resp
```

Next, as we have decided to use the response and context jointly, as an array of concatenated strings (first response, then context), we have made a function to concatenate it both for training and for test dataset. Result of this function is a numpy array of strings where each string is concatenated response and context:

```
def combine_vectors(vector1, vector2):
    array_data2 = np.array(vector1, dtype=str)
    array_data3 = np.array(vector2, dtype=str)
    Features = np.vstack((array_data3, array_data2))
    combined_features = []
    for item in Features.T:
        summed = str(" ".join(map(str, [item[0], item[1]])))
        combined_features.append(summed)
    return combined_features
```


Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

Now, let's apply all those functions to the corresponding input files:

```
#Applying function to get the training data file:

data = getdatajson(filename)

#Applying function to get the test data file:

test_data = getdatatest(test_file)
```

```
train_features = combine_vectors(data[1], data[2])
test_features = combine_vectors(test_data[1], test_data[2])
```

In terms of format, as the "labels" array is array of strings, and the tensorFlow neural network accepts only integer labels type, I'm going to transform the labels to numpy array of integers:

```
#Creating Label vector and transferring labels into integer
Labels_num = []
for item in data[0]:
    if item.endswith("NOT_SARCASM"):
        Labels_num.append(0)
    else:
        Labels_num.append(1)
Labels_num = np.array(Labels_num)
```

Next, as we will need to get a training and validation set from the training file which we have, I have splitted the arrays of labels and features for the training set to training and validation sets using the scikit learn library function:

```
X_train, X_val, y_train, y_val = sk.train_test_split(train_features,Labels_num.T,test_size=0.2, random_state = 42)
```

Next, I will transform all the resulting arrays to tensors, as the next text processing functions from the TensorFlow Text Vectorization library will require tensors as an input:

```
tensor_train = tf.constant(X_train)
tensor_val = tf.constant(X_val)
tensor_label_train = tf.constant(y_train)
tensor_label_val = tf.constant(y_val)

#Converting test data to tensor:
tensor_test = tf.constant(test_features)
```

Next step will be the text processing. In order to feed the model, using the words as a lexical units, we will need to cut all the punctuation signs, end of line and other text file tags, html tags, and put all the text in the lower case. The function for that is taken from a TensorFlow tutorial https://www.tensorflow.org/tutorials/text/word_embeddings.

```
def custom_standardization(input_data):
  lowercase = tf.strings.lower(input_data)
  stripped_html = tf.strings.regex_replace(lowercase, '<br />', ' ')
  return tf.strings.regex_replace(stripped_html, '[%s]' % re.escape(string.punctuation), '')
```

Following same tutorial, now we will need to vectorize the text. This operation will normalize, split, and map strings to integers as CNN requires integer input. The strings, which will be less than "sequence_length" parameter

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

will be padded, the ones which are longer - will be cut. The vocabulary of the tokenized text will have a number of words equal to parameter “max_features”. At the end, we are adapting the vectorization layer to the training set, which will be further used to vectorize the test and validation set.

```
max_features = 12000
sequence_length =300

vectorize_layer = TextVectorization(
    standardize=custom_standardization,
    max_tokens=max_features,
    output_mode='int',
    output_sequence_length=sequence_length)

vectorize_layer.adapt(tensor_train)
```

Now, we are vectorizing all the datasets. The shape of each dataset will now become (N of strings, 250).

```
train_dataset = vectorize_layer(tf.expand_dims(tensor_train, -1))
val_dataset = vectorize_layer(tf.expand_dims(tensor_val, -1))
test_dataset = vectorize_layer(tf.expand_dims(tensor_test, -1))
```

Next, we now can feed the data into the neural network, which is described at https://machinelearningmastery.com/develop-n-gram-multichannel-convolutional-neural-network-sentiment-analysis/.

This neural network has 3 inputs, to all 3 inputs we will supply the training dataset during training, and validation/test datasets during validation and testing.

Each of the input then is converted to Embeddings. The Embedding layer as per TensorFlow tutorial, takes the integer-encoded vocabulary and looks up the embedding vector for each word-index. These vectors are learned as the model trains. The vectors add a dimension to the output array. The resulting dimensions are: (batch, sequence, embedding)

After each of 3 inputs, it goes convolutional layer with different kernel sizes: 4, 6, 8. Those sizes were recommended in the article, and during many experiments with this model it turned out that they are really the most optimal kernel channel sizes. So, this is one of the small number of parameters left unchanged after experiments.

After each convolutional layer goes Dropout layer Dropout is a regularization technique, where every neuron can be switched of during one of the training steps, but activated back at the next step. The ratio of neurons, which will be deactivated on each step (dropout rate) is another parameter, which we haven’t changed compared to the published model, as rate of 0.5 was proven to give the best result.

After Dropout layer, a pooling layer goes. In the model it is used MaxPooling as it is a most efficient technique for this type of data.

Before merging all the outputs of the channel and passing it to the Dense layer, we are flattening it by the Flatten layer to have only a single dimension of the output for each channel.

After merging, the data comes through one Dense hidden layer with hyperparameters which we attempted to tune as well, and an Output layer, which outputs 1 number which can be afterwards classified as “SARCASM” if the number is above 0.5, or “NOT_SARCASM” if the number is below 0.5. The threshold comes from the nature of sigmoid function, which outputs the numbers between 0 and 1.

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

```
length = sequence_length
vocab_size = max_features
# channel 1
inputs1 = layers.Input(shape=(length,))
embedding1 = layers.Embedding(vocab_size, 100)(inputs1)
conv1 = layers.Conv1D(filters=64, kernel_size=4, activation='tanh', kernel_initializer = "lecun_normal")(embedding1)
drop1 = layers.Dropout(0.5)(conv1)
pool1 = layers.MaxPooling1D(pool_size=2)(drop1)
print(pool1.shape)
flat1 = layers.Flatten()(pool1)
```

```
# channel 2
inputs2 = layers.Input(shape=(length,))
embedding2 = layers.Embedding(vocab_size, 100)(inputs2)
conv2 = layers.Conv1D(filters=64, kernel_size=6, activation='tanh', kernel_initializer = "lecun_normal")(embedding2)
drop2 = layers.Dropout(0.5)(conv2)
pool2 = layers.MaxPooling1D(pool_size=2)(drop2)
flat2 = layers.Flatten()(pool2)
# channel 3
inputs3 = layers.Input(shape=(length,))
embedding3 = layers.Embedding(vocab_size, 100)(inputs3)
conv3 = layers.Conv1D(filters=64, kernel_size=8, activation='tanh', kernel_initializer = "lecun_normal")(embedding3)
drop3 = layers.Dropout(0.5)(conv3)
pool3 = layers.MaxPooling1D(pool_size=2)(drop3)
flat3 = layers.Flatten()(pool3)
```

```
# merge
merged = layers.concatenate([flat1, flat2, flat3])
# interpretation
dense1 = layers.Dense(10, activation='relu', kernel_initializer = "he_normal")(merged)
outputs = layers.Dense(1, activation='sigmoid')(dense1)
model1 = keras.Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
```

Next, we just compile, fit the model, then evaluate it and predict the labels for the test dataset:

```
# compile model
model1.compile(loss=losses.BinaryCrossentropy(from_logits=True), optimizer='nadam', metrics=['accuracy'])



print("defining model")
# fit model
model1.fit([train_dataset,train_dataset,train_dataset], tensor_label_train, epochs=10, batch_size=16)
# evaluate model on training dataset
loss, acc = model1.evaluate([val_dataset,val_dataset,val_dataset], tensor_label_val, verbose=0)
print('Train Accuracy: %f' % (acc * 100))
```

```
#Predicting the labels for test dataset
predictions = model1.predict([test_dataset, test_dataset, test_dataset])
```

After that, we are writing the results of the labels predicting to the file:

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

```
test_idx = test_data[0]
#writing the results in the file
fin = open("/home/marina/PycharmProjects/pythonProject1/ClassificationCompetition_private/answer.txt", "w")
idx = 0
print("Process predictions")
for item in predictions:
    fin.write("{},{}{}".format(test_idx[idx], "SARCASM" if item >0.5 else "NOT_SARCASM",  "\n" if idx < len(test_idx) - 1 else ""))
    idx += 1
fin.close()
```

During work on the model, we have evaluated the impact of the following hyperparameters and architecture changes on the final prediction results:

1. Changing kernel size. We have tried increasing each of items in the range, decreasing of each item in the range and trying to make broader the range of the kernel sizes (i.e. to make instead of (4, 6, 8) sizes (3, 6, 9). The best configuration stays (4, 6, 8) as proposed in the article. The F1 on the test set drops on 0.5 as minimum in any other configuration.
2. Dropout rate. We have checked 0.3 dropout rate, and the result was worse than 0.5
3. Activation function. The best model results were achieved with tanh function in the convolutional channels, relu in the dense layer and sigmoid in the output layer. That gave several % of gain in F1.
4. For better result, we have added kernel initializer. By default, TensorFlow (and the article) use Glorot initialization, but with the activation functions which we finally chosen it was not working in the optimal way. The best result was achieved with “lecun_normal” initializer in the channels, and “he_normal” in the dense layer.
5. Number of filters. Gain for at least 5% in F1 was achieved by changing it to 64 from 32.
6. We have attempted to increase the number of convolutional layers in each channel to 2. 1-st layer went with 32 filters, and 2-nd layer - with 64 filters. The number of kernels stayed the same in both layers. This change resulted in a decrease in F1 by 10%, so we dropped this change.
7. We have attempted to change the parameters “max_features” and “sequence_length”, as the real vocabulary of the training set is closer to 25 000, and maximum sequence length closer to 1000. But the training of the CNN failed already above 300 for the sequence length, as probably the size of the lines were too much different and in case of sequence length equal to the maximum string size the number of padded elements prevented to do effective prediction. Same thing happened to vocabulary. So we have left 12000 for vocabulary and the 300 for the length. With all the modifications above, we were able to reach F1 0.681775259678942 on the test set.
8. Another significant change, which was attempted, was to drop “context” both for the test and training set, and train the same model on the “response” field. Due to the reasons stated in point 7 above, we have used for this option max_features = 10 000 and sequence_length = 50. The number of epochs was increased to 15. The F1 result for this configuration was 0.677625570776255
9. Different optimizers were tried. The best result was given by “nadam” optimizer
10. It was attempted to use only response for fitting the model, but it appeared that it gave the similar result to using response+context. Also, it was attempted to feed the different channels of the model by response and context separately, it did not give the gain in performance as well.

Install and execute code

In order to use the software, one will need to install/include following libraries:

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

```
import tensorflow as tf
import numpy as np
import pandas as pd
import json
import re
import string
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras import layers
from tensorflow.keras import losses
from tensorflow.keras import preprocessing
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
import sklearn.model_selection as sk
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow import keras
```

Also, related to the code, proposed above, the path to the files need to correspond to your real files path.

The code can be launched in any Python IDE, provided that the libraries are installed.

Statistics with CNN

Recall: 0.602761982128351
Precision: 0.8244444444444444
F1 measure: 0.6963866729235102

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

BERT

Model overview

The idea is to use Tensorflow and Keras to create a ML model and use BERT pre-training models to perform pre -training on the data, fine tune it, optimize the output and create the final model.
Here are the implementation steps taken into consideration:

- Import required libraries – Tensorflow, Keras, Numpy, Pandas, Jsonlines, Tensorflow - Hub
- Load the 'train.jsonl' file and read the file
- Create the train dataset from 'train.jsonl'.
- Split the dataset into train and validation datasets
- Load models from Tensorflow Hub
- Choose BERT models and determine the best fit to fine-tune
- Use BERT models to Pre-train the data
- Build own model by combining BERT with a classifier
- Train a model, including the preprocessing module, BERT encoder, data, and classifier
- Use an Optimizer like Adaptive Moments to fine-tune the model
- Run the test data and analyze data.

Model code

Code reference: https://www.tensorflow.org/tutorials/text/classify_text_with_bert

Load dataset

The code below will read the file asked to extract dataset from and provide context, response, labels in case of training dataset and context, response, tweet ids in case of test dataset.

```
def read_input_file(file_path, training=True):
    fin = open(file_path)
    data = fin.read()
    fin.close()
    tweets = [json.loads(jline) for jline in data.splitlines()]
    tweet_responses = [[item.get("response"), " ".join(item.get("context"))] for item in tweets]
    if training:
        # tweet_labels = [item.get("label") for item in tweets]
        tweet_labels = [0 if item.get("label") == "SARCASM" else 1 for item in tweets]
        return tweet_responses, tweet_labels
    tweet_ids = [item.get("id") for item in tweets]
    return tweet_ids, tweet_responses
```

Create Validation dataset

Read the train and test json files and load the output into a train dataset. This will be our input to the model. The dataset will have two columns – response and context.

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

```
def validation_set():
    autotune = tf.data.experimental.AUTOTUNE
    batch_size = 32
    seed = 42

    training_responses, training_labels = read_input_file('data/train.jsonl')
    test_tweet_ids, test_responses = read_input_file('data/test.jsonl', training=False)
    train_conversation, eval_conversations, train_labels, eval_labels = \
        sk.train_test_split(np.array(training_responses), np.array(training_labels), train_size=0.8)
    df = pd.DataFrame(data=train_conversation, columns=["response", "context"])
    training_label_series = pd.Series(train_labels)
    eval_df = pd.DataFrame(data=eval_conversations, columns=["response", "context"])
    eval_label_series = pd.Series(eval_labels)
    test_ds = pd.DataFrame(data=test_responses, columns=["response", "context"])
```

Fine tune dataset

Set the BERT model to fine-tune the data

```
    # BERT model to fine-tune
    bert_model_name = 'small_bert/bert_en_uncased_L-4_H-512_A-8'

    # BERT Models
    map_name_to_handle = {
        'bert_en_uncased_L-12_H-768_A-12':
            'https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/3',
        'bert_en_cased_L-12_H-768_A-12':
            'https://tfhub.dev/tensorflow/bert_en_cased_L-12_H-768_A-12/3',
        'bert_multi_cased_L-12_H-768_A-12':
            'https://tfhub.dev/tensorflow/bert_multi_cased_L-12_H-768_A-12/3',
        'small_bert/bert_en_uncased_L-2_H-128_A-2':
            'https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-2_H-128_A-2/1',
        'small_bert/bert_en_uncased_L-2_H-256_A-4':
            'https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-2_H-256_A-4/1',
        'small_bert/bert_en_uncased_L-2_H-512_A-8':
```

Pre-processing the dataset

The preprocessing model must be the one referenced by the documentation of the BERT model. Set the BERT preprocessing model and run it on test data

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

```
bert_model_name = 'https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4'
bert_premodel_name = 'https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/1'

tfhub_handle_encoder = map_name_to_handle[bert_model_name]
tfhub_handle_preprocess = map_model_to_preprocess[bert_premodel_name]

print('BERT model selected           : {tfhub_handle_encoder}')
print('Preprocess model auto-selected: {tfhub_handle_preprocess}')

# create preprocessing model
bert_preprocess_model = hub.KerasLayer(tfhub_handle_preprocess)
text_preprocessed = bert_preprocess_model(test_ds)

# use BERT on train model
bert_model = hub.KerasLayer(tfhub_handle_encoder)
bert_results = bert_model(text_preprocessed)
```

Build Classifier model

Now the last step is to build the Classifier model and run the training dataset.
This is very simple fine-tuned model, with the preprocessing model, the selected BERT model, one Dense and a Dropout layer.

```
    # run model
    classifier_model = build_classifier_model(tfhub_handle_encoder, tfhub_handle_preprocess)
    bert_raw_result = classifier_model(tf.constant(text_preprocessed))
    print(tf.sigmoid(bert_raw_result))


def build_classifier_model(tfhub_handle_encoder, tfhub_handle_preprocess):
    """build a fine-tuned BERT model with one Dense and a Dropout layer"""
    text_input = tf.keras.layers.Input(shape=(), dtype=tf.string, name='text')
    preprocessing_layer = hub.KerasLayer(tfhub_handle_preprocess, name='preprocessing')
    encoder_inputs = preprocessing_layer(text_input)
    encoder = hub.KerasLayer(tfhub_handle_encoder, trainable=True, name='BERT_encoder')
    outputs = encoder(encoder_inputs)
    net = outputs['pooled_output']
    net = tf.keras.layers.Dropout(0.1)(net)
    net = tf.keras.layers.Dense(1, activation=None, name='classifier')(net)
    return tf.keras.Model(text_input, net)
```

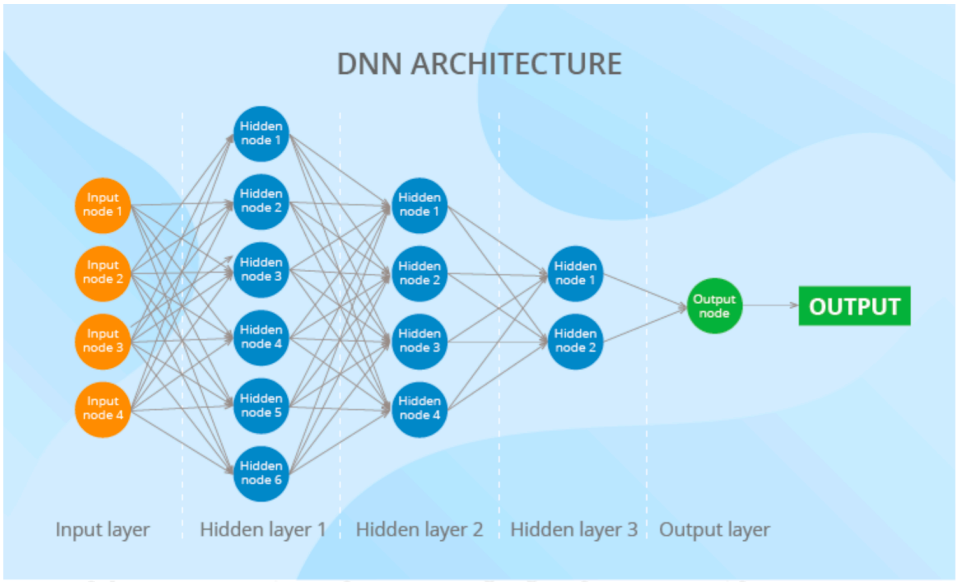
Note: Could not execute the program successfully due to too many library reference issues.

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

DNN Classifier

Model overview

A DNN is a collection of neurons organized in a sequence of multiple layers, where neurons receive as input the neuron activations from the previous layer, and perform a simple mathematical computation (e.g. a weighted sum of the input followed by a nonlinear activation). The term "deep" refers to the presence of multiple layers. The neurons of the network jointly implement a complex nonlinear mapping from the input to the output. This mapping between the input and output is learned from the data by adapting the weights of each neuron using a technique called error backpropagation. In simple words, DNN is a feedforward multilayer neural network architecture or it is an artificial neural network (ANN) with multiple layers between the input and the output layers. We trained it with a set of labeled data in order to perform classification on similar, unlabeled test data. In this experiment I used a pre-existing DNN classifier in tensorflow library.



More information on DNN classifiers at
https://papers.nips.cc/paper/2013/file/f7cade80b7cc92b991cf4d2806d6bd78-Paper.pdf.

Model code

Load dataset

The code below will read the file asked to extract dataset from and provide 1/context, 2/response, 3/labels in case of training dataset and 1/context, 2/response, 3/tweet ids in case of test dataset. Individual line in training and test datasets is a json object which is read and corresponding information is extracted.

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

```python
def read_input_file(file_path, training=True):
    # Open and read file contents
    fin = open(file_path)
    data = fin.read()
    fin.close()
    # Every line in file is a json object.
    # Irrespective of file to be read read every json object and extract response, context
    tweets = [json.loads(jline) for jline in data.splitlines()]
    tweet_responses = [[item.get("response"), " ".join(item.get("context"))] for item in tweets]
    # If training file is being read, need to read the labels for each tweet
    if training:
        # tweet_labels = [item.get("label") for item in tweets]
        tweet_labels = [0 if item.get("label") == "SARCASM" else 1 for item in tweets]
        return tweet_responses, tweet_labels
    # if the file type is not training, we will encounter this code path and will parse tweet ids
    tweet_ids = [item.get("id") for item in tweets]
    return tweet_ids, tweet_responses
```

Split dataset

While training the model I split the training dataset into training and evaluation in order to test for the local accuracy to build confidence in the model. Once the training dataset was available, I split it into training and evaluation datasets using scikit-learn's *train_test_split* API.

```python
# Split the dataset into training and evaluation datasets
train_conversation, eval_conversations, train_labels, eval_labels = \
    sk.train_test_split(np.array(training_responses), np.array(training_labels), train_size=0.8)
```

Preprocessing

The tensorflow DNNClassifier, can handle multiclass labels, multiple columns in input. There are various ways in which the dataset could be fed to the classifier; I tried feeding vectorized, cleaned data by applying tokenization, stemming, remove special characters, stop word removal, etc. but I got better accuracy with universal sentence encoder. For text representation I converted the dataset into *hub.text_embedding_column* using deep averaging encoder, one of Universal Sentence Encoder.

```python
# Start pre-processing
df = pd.DataFrame(data=train_conversation, columns=["response", "context"])
training_label_series = pd.Series(train_labels)

eval_df = pd.DataFrame(data=eval_conversations, columns=["response", "context"])
eval_label_series = pd.Series(eval_labels)

test_ds = pd.DataFrame(data=test_responses, columns=["response", "context"])

print("Creating text columns")
# Feeding text as columns - https://medium.com/engineering-zemoso/text-classification-bert-vs-dnn-b226497c9de7
os.environ['TFHUB_CACHE_DIR'] = 'tf_cache/'
TFHUB_URL = "https://tfhub.dev/google/universal-sentence-encoder/2"
embedded_text_response_column = hub.text_embedding_column(key="response", module_spec=TFHUB_URL)
embedded_text_context_column = hub.text_embedding_column(key="context", module_spec=TFHUB_URL)
```

Sentence Encoder encodes text into high dimensional vectors that can be used for text classification, semantic similarity, clustering, and other natural language tasks. The pre-trained Universal Sentence Encoder is publicly available in Tensorflow-hub. The model is trained and optimized for greater-than-word length text, such as sentences, phrases or short paragraphs. It is trained on a variety of data sources and a variety of tasks with the aim of dynamically accommodating a wide variety of natural language understanding tasks. The input is variable length English text and the output is a 512-dimensional vector. It comes with two variations i.e., one trained with Transformer encoder and the other one trained with Deep Averaging Network (DAN). The two have a trade-off of accuracy and computational resource requirement. While the one with a Transformer encoder has higher accuracy, it is computationally more intensive. The one with DAN encoding uses less memory with slight compromise of accuracy. We will use universal sentence encoder which is part of TensorFlow-hub to embed the sentences to train the DNN classifier. Reference - https://tfhub.dev/google/universal-sentence-encoder/2 and https://medium.com/engineering-zemoso/text-classification-bert-vs-dnn-b226497c9de7 .

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

My machine could not withstand the memory usage for transformer encoder, thus I used model trained with DAN for this assignment. Just like every tensorflow encoder, sentence encoder also pulls down the model locally in a THUB_CACHE_DIR environment variable, which is configured in the code. When you execute the code, tensorflow will bring down the model into *tf_cache* directory (configured in the code) under the execution path. For the first time execution of the model, creating cache will take a few minutes. In my experience it takes around 3-4 minutes to create cache.

Define model

For this experiment I used pre-existing classifier – *tensorflow.estimators.DNNClassifier* .

```
# DNN Classifier reference – https://www.tensorflow.org/tutorials/estimator/premade
# crelu activation whitepaper reference – https://arxiv.org/pdf/1603.05201.pdf
classifier = tf.estimator.DNNClassifier(
    feature_columns=[embedded_text_response_column, embedded_text_context_column],
    hidden_units=[64, 32, 16],
    dropout=0.2,
    n_classes=2,
    optimizer="SGD",
    activation_fn=tf.nn.crelu)
```

For hyperparameter tuning I added three hidden layers with additional

- dropout layer of 0.2 to avoid overfitting the dataset
- optimizer – SGD
- activation of crelu, using default learning rate, decay and, momentum for SGD optimizer. Further reading on crelu - https://arxiv.org/pdf/1603.05201.pdf

Training

After creating the model, I trained the model with training dataset which uses the input function, converting feature columns “response” and “context” into tensorflow datasets, as:

```
def input_fn(features, labels, training=True, batch_size=256):
    # Convert the inputs to DataSet
    dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))
    if training:
        dataset = dataset.shuffle(10000).repeat()
    return dataset.batch(batch_size)
```

Using the training dataset and above input function train the classifier in 5000 steps. In local testing the local accuracy stopped increasing after 5000 steps, thus chose the step count as 5000. The classifier creates regular checkpoints of the trained model which can be used for either

- using model for predictions
- performing more training. For instance, in first go I had experimented with 2000 steps, and after that used checkpointing directory to feed further for another 3000 steps. This way I was able train the model quicker without having to re-run the entire training for tunning step count.

Also, I added the dataset to shuffle for every step so as to avoid overfitting the dataset.

```
print("Created classifier, starting training")
classifier.train(input_fn=lambda: input_fn(df, training_label_series, training=True), steps=5000)
```

Due to the number of steps being 5000, model takes more time for training. In my local setup it took 15-20 minutes for training.

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

Evaluation

Now it's time to evaluate the model using the evaluation data which was split in earlier step. The result from evaluation helped me tune the parameters better locally.

```
print("Trained classifier, initiating evaluation")
result = classifier.evaluate(input_fn=lambda: input_fn(eval_df, eval_label_series, training=False))
```

Prediction

After evaluating on local to find steps where local accuracy stops increasing, I set the training steps to 5000 and start with prediction using

```
# Use trained classifier to predict for the test dataset
predictions = classifier.predict(input_fn=lambda: test_input_fn(test_ds))
```

Then store the predictions in answer.txt

```
# Create answer.txt for submission from predictions
fin = open("answer.txt", "w")
idx = 0
print("Process predictions")
for pred_dict in predictions:
    class_id = pred_dict['class_ids'][0]
    fin.write("{},{}{}".format(test_tweet_ids[idx], "SARCASM" if class_id == 0 else "NOT_SARCASM",
                               "\n" if idx < len(test_tweet_ids) - 1 else ""))
    idx += 1
fin.close()
```

Install and execute code

Install the required softwares

- Python 3.8
- tensorflow 2.3.1
- tensorflow-hub 0.10.0
- scikit-learn 0.23.2
- pandas 1.1.4
- numpy 1.18.5, if pandas is downloaded before this numpy will be downloaded already

Execute the program

Execute the python script as: "python dnn.py"

Disclaimer

The machine on which the code was built:
MacBook Pro (13-inch, 2019, Four Thunderbolt 3 ports)
Processor 2.8 GHz Intel Core i7
Memory 16 GB 2133 MHz LPDDR3
Graphics Intel Iris Plus Graphics 655 1536 MB

Statistics with DNN

Recall: 0.7733333333333333
Precision: 0.699497487437186
F1 measure: 0.7345646437994723

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

Bidirectional LSTM

Model overview

Long Short-Term Memory networks are extension of RNN, which are capable of learning order dependence in sequence predictions. The model is well known for applications like text classification, speech recognition, etc. Just like RNN, LSTM also uses context for making predictions. LSTM overcomes the shortcoming of RNN (RNN has limited ability to learn from range of contextual information) by remembering the context in short or long memory.

An LSTM layer consists of a set of recurrently connected blocks, known as memory blocks. These blocks can be thought of as a differentiable version of the memory chips in a digital computer. Each one contains one or more recurrently connected memory cells and three multiplicative units – the input, output and forget gates – that provide continuous analogues of write, read and reset operations for the cells. … The net can only interact with the cells via the gates.
- Alex Graves, et al., Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures, 2005.

In this part I created multilayer LSTM model with tensorflow-keras library.

Model code

Load dataset

The code below will read the file asked to extract dataset from and provide context, response, labels in case of training dataset and context, response, tweet ids in case of test dataset.

```
# Read training or test files
def read_input_file(file_path, training=True):
    # Open and read file contents
    fin = open(file_path)
    data = fin.read()
    fin.close()
    # Every line in file is a json object.
    # Irrespective of file to be read read every json object and extract response, context
    tweets = [json.loads(jline) for jline in data.splitlines()]
    # I could not figure out how to feed context and response separately in this model, thus concatenating the two
    tweet_responses = [clean_input(" ".join([item.get("response"), " ".join(item.get("context"))])) for item in tweets]
    # If training file is being read, need to read the labels for each tweet
    if training:
        # Convert label into numeric values, SARCASM as 0 and NOT_SARCASM as 1
        tweet_labels = [0 if item.get("label") == "SARCASM" else 1 for item in tweets]
        return tweet_responses, tweet_labels
    # if the file type is not training, we will encounter this code path and will parse tweet ids
    ids = [item.get("id") for item in tweets]
    return ids, tweet_responses
```

While parsing the data from file, it is supplied to clean_input function which removes emojis, htmls tags, non-alphanumeric characters, etc.; performs stemming and removes stop words

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

```
# Method to clean data while parsing from file
def clean_input(seq):
    # remove common regex like emojis, symbols, etc
    # reference https://stackoverflow.com/questions/33404752/removing-emojis-from-a-string-in-python
    regrex_pattern = re.compile(pattern="["
                                        u"\U0001F600-\U0001F64F"  # emoticons
                                        u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                                        u"\U0001F680-\U0001F6FF"  # transport & map symbols
                                        u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                                        "]+", flags=re.UNICODE)
    seq = regrex_pattern.sub(r'', seq)
    # remove all html tags
    words = nltk.word_tokenize(re.sub("<.*?>", " ", seq.lower()))
    # remove punctuations
    token_words = [w for w in words if w.isalpha()]
    # stemming
    stemmed_words = [stemming.stem(word) for word in token_words]
    # remove stop words
    clean_words = [w for w in stemmed_words if not w in stops]
    return " ".join(clean_words)
```

Split dataset

Once the training dataset is available, split it into training and evaluation datasets

```
    # Split the dataset into training and evaluation datasets
    train_responses, eval_responses, train_labels, eval_labels = \
        sk.train_test_split(np.array(training_responses), np.array(training_labels), train_size=0.8)
```

Preprocessing

In this experiment the model I created expects the dataset in numeric form, thus the preprocessing converts the text data into numeric form using tensorflow's TextVecorization. During text vectorization data is cleaned using *cutom_standardization* function

```
    # Start pre-processing
    tensor_train_labels = tf.constant(train_labels)
    tensor_eval_labels = tf.constant(eval_labels)
    max_features = 125000
    sequence_length = 500
    # Create TextVectorization object to vectorize the text dataset
    text_vector = TextVectorization(
        standardize=custom_standardization,
        max_tokens=max_features,
        output_mode='int',
        output_sequence_length=sequence_length)
```

Custom_standaradization function

```
# Method called during text vectorization to perform data cleanup
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    stripped_html = tf.strings.regex_replace(lowercase, '<br />', ' ')
    return tf.strings.regex_replace(stripped_html, '[%s]' % re.escape(string.punctuation), '')
```

For text vectorization process the training data has to be fed to the vectorization object for developing vocabulary. Given training file was split into 80% training and 20% evaluation datasets. The training dataset contains 1,250,241 trainable parameters. As neural networks are capable of learning high number of features, limiting the max features to 125000 and sequence length to be 500. The definition of text vector is then used for adapting the training dataset for building vocabulary:

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

```
# Adapt the training dataset for model to learn vocabulary
text_vector.adapt(train_responses)
```

To clean the dataset and bring it in expected input format by the model, apply text vectorization to all data sets

```
# Vectorize training dataset
train_dataset = text_vector(train_responses)
# Vectorize evaluation dataset
eval_dataset = text_vector(eval_responses)
# Vectorize test dataset
test_data_set = text_vector(test_tweets)
```

Define model

For this experiment I built model using tensorflow's keras APIs to build a sequential model. It has multiple layers:

1. Input layer which accepts the vocabulary from text vectorization step above
2. Bidirectional LSTM
3. Activation layer to adjust bias and weights in the network
4. Dropout layer to avoid overfitting the model to training dataset
5. Output layer, emitting the probability

```
# Build LSTM model
model = tf.keras.Sequential([
    # Define input layer taking vocabulary from adapt step above
    layers.Embedding(input_dim=len(text_vector.get_vocabulary()),
                     output_dim=64,
                     mask_zero=True),
    # Define bidirectional LSTM model
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    # Add hidden layer with activation tanh
    tf.keras.layers.Dense(32, activation='tanh'),
    # Add dropout
    layers.Dropout(0.2),
    # Add output layer which holds the prediction probability
    layers.Dense(1)])
```

For activations I tried multiple activations like linear (no activation), relu, softmax, tanh; I got better consistency with tanh.

The model defined with Keras APIs need to compiled for using it for training

```
# Compile model with adam optimizer
model.compile(loss=losses.BinaryCrossentropy(from_logits=True),
              optimizer='adam',
              metrics=tf.metrics.BinaryAccuracy(threshold=0.0))
```

The given dataset has high number of parameters; thus, we need to use an optimizer that supports high number of parameters and provide good results with them. One such optimizer is "adam" which is used in my model.

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. According to [Kingma et al., 2014] Adam: A Method for Stochastic Optimization, the method is "*computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters*".

Training and evaluation = fit

Keras APIs allow to train and evaluate the model with single method of fit. In my local testing the local accuracy stopped increasing after 4 epochs.

```
# In local testing, local_accuracy stops increasing after 4 epochs
epochs = 4
# Train and evaluate the model by asking model to fit to the training datasets
history = model.fit(train_dataset, tensor_train_labels, epochs=epochs,
                    validation_data=(eval_dataset, tensor_eval_labels),
                    verbose=2)
```

Prediction

After fitting the model to training dataset, performed prediction using the model to receive probability of a label for each record in test dataset.

```
# Predict test dataset
results = tf.sigmoid(model.predict(test_data_set))
```

Then store the predictions in answer.txt

```
# Create answer.txt for submission from predictions
fin = open("answer.txt", "w")
idx = 0
for x in np.nditer(results):
    fin.write("{},{}{}".format(tweet_ids[idx], "SARCASM" if x < 0.5 else "NOT_SARCASM", "\n" if idx < len(tweet_ids) - 1 else ""))
    idx += 1
fin.close()
```

Install and execute code

Install the required softwares

- Python 3.8
- Numpy 1.18.5
- Tensorflow 2.3.1
- Sklearn
- Nltk 3.5

Execute the program

Execute the python script as: “python lstm.py”

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2

Disclaimer
The machine on which the code was built:
MacBook Pro (13-inch, 2019, Four Thunderbolt 3 ports)
Processor 2.8 GHz Intel Core i7
Memory 16 GB 2133 MHz LPDDR3
Graphics Intel Iris Plus Graphics 655 1536 MB

Statistics with Bidirectional LSTM

Recall: 0.7022222222222222
Precision: 0.6147859922178989
F1 measure: 0.6556016597510372

Team Contribution

We as a team initially started to explore more on learning various Machine Learning algorithms and sharing the learning within the team. From our learnings we had identified that everyone in the team has to work on individual models to try out and share the results with the team. Original model distribution was:

Marina – CNN model
Tirthankar – Bert model
Savita – XLNet

We had regular meetings to evaluate the progress and share the learnings. From experimentation Savita could not get the XLNet working locally, thus decided to explore DNN Classifier and LSTM models. At the end the final distribution of the work was:

Marina – Mutichannel CNN model
Tirthankar – Bert model
Savita – DNN Classifier and Bidirectional LSTM model

At the end following are the results from leadership board:

Model	Precision	Recall	F1
Multichannel CNN	0.602761982128351	0.8244444444444444	0.6963866729235102
Bidirectional LSTM	0.6147859922178989	0.7022222222222222	0.6556016597510372
DNN Classifier	0.699497487437186	0.7733333333333333	0.7345646437994723

DNN classifier was able to cross the baseline consistently and screenshot from leadership board:

Project Artifacts

- Link to code base and documentation - https://github.com/violetta-ta/CourseProject
- For submission to Leaderboard different repository was used and the name on leadership board is - marina_polupanova
- Project tutorial - https://mediaspace.illinois.edu/media/t/1_uscvryhp

Marina Polupanova - marinap2
Savita Manghnani - savitam2
Tirthankar Bhakta - tbhakta2