Programming Assignment (20 points)

In this assignment, you will solve an irony detection task: given a tweet, your job is to classify whether it is ironic or not.

You will implement a new classifier that does not rely on feature engineering as in previous homeworks. Instead, you will use pretrained word embeddings downloaded from using the `irony.py` script as your input feature vectors. Then, you will encode your sequence of word embeddings with an (already implemented) LSTM and classify based on its final hidden state.

In [1]:
```
# This is so that you don't have to restart the kernel everytime you edit hmm.py

%load_ext autoreload
%autoreload 2
```

Data

We will use the dataset from SemEval-2018: https://github.com/Cyvhee/SemEval2018-Task3

In [2]:
```
from irony import load_datasets

train_sentences, train_labels, test_sentences, test_labels, label2i = load_datasets()

# TODO: Split train into train/dev
```

```
2022-10-27 23:56:53.844715: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

Baseline: Naive Bayes

We have provided the solution for the Naive Bayes part from HW2 in bayes.py

There are two implementations: NaiveBayesHW2 is what was expected from HW2. However, we will use a more effecient implementation of it that uses vector operations to calculate the probabilities. Please go through it if you would like to

In [3]:
```
from irony import run_nb_baseline

run_nb_baseline()
```

```
Vectorizing Text: 100%|██████████| 3834/3834 [00:00<00:00, 5003.24it/s]
Vectorizing Text: 100%|██████████| 3834/3834 [00:00<00:00, 7145.00it/s]
Vectorizing Text: 100%|██████████| 784/784 [00:00<00:00, 5248.19it/s]
Baseline: Naive Bayes Classifier
F1-score Ironic: 0.6402966625463535
Avg F1-score: 0.6284487265300938
```

Task 1: Implement avg_f1_score() in util.py. Then re-run the above cell (2 Points)

So the micro F1-score for the test set of the Ironic Class using a Naive Bayes Classifier is **0.64**

Logistic Regression with Word2Vec (Total: 18 Points)

Unlike sentiment, Irony is very subjective, and there is no word list for ironic and non-ironic tweets. This makes hand-engineering features tedious, therefore, we will use word embeddings as input to the classifier, and make the model automatically extract features aka learn weights for the embeddings

Tokenizer for Tweets

Tweets are very different from normal document text. They have emojis, hashtags and bunch of other special character. Therefore, we need to create a suitable tokenizer for this kind of text.

Additionally, as described in class, we also need to have a consistent input length of the text document in order for the neural networks built over it to work correctly.

Task 2: Create a Tokenizer with Padding (5 Points)

Our Tokenizer class is meant for tokenizing and padding batches of inputs. This is done before we encode text sequences as torch Tensors.

Update the following class by completing the todo statements.

In [4]:
```
from typing import Dict, List, Optional, Tuple
from collections import Counter

import torch
import numpy as np
import spacy


class Tokenizer:
    """Tokenizes and pads a batch of input sentences."""

    def __init__(self, pad_symbol: Optional[str] = "<PAD>"):
        """Initializes the tokenizer

        Args:
            pad_symbol (Optional[str], optional): The symbol for a pad. Defaults to "<PAD>".
        """
        self.pad_symbol = pad_symbol
        self.nlp = spacy.load("en_core_web_sm")

    def __call__(self, batch: List[str]) -> List[List[str]]:
        """Tokenizes each sentence in the batch, and pads them if necessary so
        that we have equal length sentences in the batch.

        Args:
            batch (List[str]): A List of sentence strings

        Returns:
            List[List[str]]: A List of equal-length token Lists.
        """
        batch = self.tokenize(batch)
        batch = self.pad(batch)

        return batch

    def tokenize(self, sentences: List[str]) -> List[List[str]]:
        """Tokenizes the List of string sentences into a Lists of tokens using spacy tokenizer.

        Args:
            sentences (List[str]): The input sentence.

        Returns:
            List[str]: The tokenized version of the sentence.
        """
        # TODO: Tokenize the input with spacy.
        # TODO: Make sure the start token is the special <SOS> token and the end token
        #       is the special <EOS> token
        tokenized_sentences = []
        for sentence in sentences:
            tokenized_sentence = ['<SOS>']
            sentence = self.nlp(sentence)
            for token in sentence:
                tokenized_sentence.append(token.text)
            tokenized_sentence.append('<EOS>')
            tokenized_sentences.append(tokenized_sentence)
        return tokenized_sentences

    def pad(self, batch: List[List[str]]) -> List[List[str]]:
        """Appends pad symbols to each tokenized sentence in the batch such that
        every List of tokens is the same length. This means that the max length sentence
        will not be padded.

        Args:
            batch (List[List[str]]): Batch of tokenized sentences.

        Returns:
            List[List[str]]: Batch of padded tokenized sentences.
        """
        # TODO: For each sentence in the batch, append the special <P>
        #       symbol to it n times to make all sentences equal length
        pad_max = max([len(l) for l in batch])
        for sentence in batch:
            sentence.extend(['<PAD>'] * (pad_max - len(sentence)))
        return batch
```

In [5]:
```
# create the vocabulary of the dataset: use both training and test sets here

SPECIAL_TOKENS = ['<UNK>', '<PAD>', '<SOS>', '<EOS>']

all_data = train_sentences + test_sentences
my_tokenizer = Tokenizer()

tokenized_data = my_tokenizer.tokenize(all_data)
vocab = sorted(set([w for ws in tokenized_data + [SPECIAL_TOKENS] for w in ws]))

with open('vocab.txt', 'w') as vf:
    vf.write('\n'.join(vocab))
```

Embeddings

We use GloVe embeddings https://nlp.stanford.edu/projects/glove/. But these do not necessarily have all of the tokens that will occur in tweets! Hoad the GloVe embeddings, pruning them to only those words in vocab.txt. This is to reduce the memory and runtime of your model.

Then, find the out-of-vocabulary words (oov) and add them to the encoding dictionary and the embeddings matrix.

In [6]:
```
# Dowload the gloVe vectors for Twitter tweets. This will download a file called glove.twitter.27B.zip

# ! wget https://nlp.stanford.edu/data/glove.twitter.27B.zip
```

In [7]:
```
# unzip glove.twitter.27B.zip
# if there is an error, please download the zip file again

# ! unzip glove.twitter.27B.zip
```

In [8]:
```
# Let's see what files are there:

# ! ls . | grep "glove.*.txt"
```

In [9]:
```
# For this assignment, we will use glove.twitter.27B.50d.txt which has 50 dimensional word vectors
# Feel free to experiment with vectors of other sizes

embeddings_path = 'glove.twitter.27B.50d.txt'
vocab_path = "./vocab.txt"
```

Creating a custom Embedding Layer

Now the GloVe file has vectors for about 1.2 million words. However, we only need the vectors for a very tiny fraction of words -> the unique words that are there in the classification corpus. Some of the next tasks will be to create a custom embedding layer that has the vectors for this small set of words

Task 2: Extracting word vectors from GloVe (3 Points)

In [32]:
```
from typing import Dict, Tuple

import torch


def read_pretrained_embeddings(
    embeddings_path: str,
    vocab_path: str
) -> Tuple[Dict[str, int], torch.FloatTensor]:
    """Read the embeddings matrix and make a dict hashing each word.

    Note that we have provided the entire vocab for train and test, so that for practical purposes
    we can simply load those words in the vocab, rather than all 27B embeddings

    Args:
        embeddings_path (str): _description_
        vocab_path (str): _description_

    Returns:
        Tuple[Dict[str, int], torch.FloatTensor]: _description_
    """
    word2i = {}
    vectors = []

    with open(vocab_path, encoding='utf8') as vf:
        vocab = set([w.strip() for w in vf.readlines()])

#     print('vocab we have', vocab)
    print(f"Reading embeddings from {embeddings_path}...")
    with open(embeddings_path, "r") as f:
        i = 0
        for line in f:
            word, *weights = line.rstrip().split(" ")
            # TODO: Build word2i and vectors such that
            #       each word points to the index of its vector,
            #       and only words that exist in `vocab` are in our embeddings
            if word in vocab:
                word2i[word] = i
                vectors.insert(i, torch.Tensor([float(x) for x in weights]))
                i += 1
    return word2i, torch.stack(vectors)
```

Task 3: Get GloVe Out of Vocabulary (oov) words (0 Points)

The task is to find the words in the Irony corpus that are not in the GloVe Word list

In [11]:
```
def get_oovs(vocab_path: str, word2i: Dict[str, int]) -> List[str]:
    """Find the vocab items that do not exist in the glove embeddings (in word2i).
    Return the List of such (unique) words.

    Args:
        vocab_path: List of batches of sentences.
        word2i (Dict[str, int]): _description_

    Returns:
        List[str]: _description_
    """
    with open(vocab_path, encoding='utf8') as vf:
        vocab = set([w.strip() for w in vf.readlines()])

    glove_and_vocab = set(word2i.keys())
    vocab_and_not_glove = vocab - glove_and_vocab
    return list(vocab_and_not_glove)
```

Task 4: Update the embeddings with oov words (3 Points)

In [12]:
```
def intialize_new_embedding_weights(num_embeddings: int, dim: int) -> torch.FloatTensor:
    """xavier initialization for the embeddings of words in train, but not in gLove.

    Args:
        num_embeddings (int): _description_
        dim (int): _description_

    Returns:
        torch.FloatTensor: _description_
    """
#     TODO: Initialize a num_embeddings x dim matrix with xiavier initiialization
#           That is, a normal distribution with mean 0 and standard deviation of dim^-0.5
    w = torch.empty(num_embeddings, dim)
    return torch.nn.init.xavier_normal_(w)


def update_embeddings(
    glove_word2i: Dict[str, int],
    glove_embeddings: torch.FloatTensor,
    oovs: List[str]
) -> Tuple[Dict[str, int], torch.FloatTensor]:
    # TODO: Add the oov words to the dict, assigning a new index to each

    # TODO: Concatenate a new row to embeddings for each oov
    #       initialize those new rows with `intialize_new_embedding_weights`

    # TODO: Return the tuple of the dictionary and the new embeddings matrix
    index = len(glove_word2i)
    row = 0
    for word in oovs:
        glove_word2i[word] = index
        index += 1
        row += 1
    x = intialize_new_embedding_weights(row, glove_embeddings.size(dim=1))
    glove_embeddings = torch.cat((glove_embeddings, x), 0)
    return  glove_word2i, glove_embeddings
```

In [33]:
```
glove_word2i, glove_embeddings = read_pretrained_embeddings(
    embeddings_path,
    vocab_path
)

# Find the out-of-vocabularies
oovs = get_oovs(vocab_path, glove_word2i)

# Add the oovs from training data to the word2i encoding, and as new rows
# to the embeddings matrix
word2i, embeddings = update_embeddings(glove_word2i, glove_embeddings, oovs)
```

```
Reading embeddings from glove.twitter.27B.50d.txt...
```

Encoding words to integers: DO NOT EDIT

In [14]:
```
# Use these functions to encode your batches before you call the train loop.

def encode_sentences(batch: List[List[str]], word2i: Dict[str, int]) -> torch.LongTensor:
    """Encode the tokens in each sentence in the batch with a dictionary

    Args:
        batch (List[List[str]]): The padded and tokenized batch of sentences.
        word2i (Dict[str, int]): The encoding dictionary.

    Returns:
        torch.LongTensor: The tensor of encoded sentences.
    """
    UNK_IDX = word2i["<UNK>"]
    tensors = []
    for sent in batch:
        tensors.append(torch.LongTensor([word2i.get(w, UNK_IDX) for w in sent]))

    return torch.stack(tensors)


def encode_labels(labels: List[int]) -> torch.FloatTensor:
    """Turns the batch of labels into a tensor

    Args:
        labels (List[int]): List of all labels in the batch

    Returns:
        torch.FloatTensor: Tensor of all labels in the batch
    """
    return torch.LongTensor([int(l) for l in labels])
```

In [30]:
```
import random
vocab_path = "./vocab.txt"


def make_batches(sequences: List[str], batch_size: int) -> List[List[str]]:
    """Yield batch_size chunks from sequences."""
    # TODO
    samples = sequences
    batches = []
    for i in range(0, len(samples), batch_size):
        batches.append(samples[i:i+batch_size])
    return batches

# TODO: Set your preferred batch size
batch_size = 16
tokenizer = Tokenizer()

# We make batches now and use those.
encode_tokenized_batch_train_sentences = []
encode_tokenized_batch_train_labels = []
encode_tokenized_batch_dev_sentences = []
encode_tokenized_batch_dev_labels = []

# Note: Labels need to be batched in the same way to ensure
# We have train sentence and label batches lining up.
for batch in make_batches(train_sentences, batch_size):
    encode_tokenized_batch_train_sentences.append(encode_sentences(tokenizer(batch), word2i))

for batch in make_batches(train_labels, batch_size):
    encode_tokenized_batch_train_labels.append(encode_labels(batch))

for batch in make_batches(test_sentences, batch_size):
    encode_tokenized_batch_dev_sentences.append(encode_sentences(tokenizer(batch), word2i))

for batch in make_batches(test_labels, batch_size):
    encode_tokenized_batch_dev_labels.append(encode_labels(batch))
```

Modeling (7 Points)

In [16]:
```
import torch


# Notice there is a single TODO in the model
class IronyDetector(torch.nn.Module):
    def __init__(
        self,
        input_dim: int,
        hidden_dim: int,
        embeddings_tensor: torch.FloatTensor,
        pad_idx: int,
        output_size: int,
        dropout_val: float = 0.3,
    ):
        super().__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.pad_idx = pad_idx
        self.dropout_val = dropout_val
        self.output_size = output_size
        # TODO: Initialize the embeddings from the weights matrix.
        #       Check the documentation for how to initialize an embedding layer
        #       from a pretrained embedding matrix.
        #       Be careful to set the `freeze` parameter!
        #       Docs are here: https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html#torch.nn.Embed
        self.embeddings = torch.nn.Embedding.from_pretrained(embeddings_tensor)
        # Dropout regularization
        # https://jmlr.org/papers/v15/srivastava14a.html
        self.dropout_layer = torch.nn.Dropout(p=self.dropout_val, inplace=False)
        # Bidirectional 2-layer LSTM. Feel free to try different parameters.
        # https://colah.github.io/posts/2015-08-Understanding-LSTMs/
        self.lstm = torch.nn.LSTM(
            self.input_dim,
            self.hidden_dim,
            num_layers=2,
            dropout=dropout_val,
            batch_first=True,
            bidirectional=True,
        )
        # For classification over the final LSTM state.
        self.classifier = torch.nn.Linear(hidden_dim*2, self.output_size)
        self.log_softmax = torch.nn.LogSoftmax(dim=2)

    def encode_text(
        self,
        symbols: torch.Tensor
    ) -> torch.Tensor:
        """Encode the (batch of) sequence(s) of token symbols with an LSTM.
            Then, get the last (non-padded) hidden state for each symbol and return that.

        Args:
            symbols (torch.Tensor): The batch size x sequence length tensor of input tokens

        Returns:
            torch.Tensor: The final hiddens tate of the LSTM, which represents an encoding of
                the entire sentence
        """
        # First we get the embedding for each input symbol
        embedded = self.embeddings(symbols)
        embedded = self.dropout_layer(embedded)
        # Packs embedded source symbols into a PackedSequence.
        # This is an optimization when using padded sequences with an LSTM
        lens = (symbols != self.pad_idx).sum(dim=1).to("cpu")
        packed = torch.nn.utils.rnn.pack_padded_sequence(
            embedded, lens, batch_first=True, enforce_sorted=False
        )
        # -> batch_size x seq_len x encoder_dim, (h0, c0).
        packed_outs, (H, C) = self.lstm(packed)
        encoded, _ = torch.nn.utils.rnn.pad_packed_sequence(
            packed_outs,
            batch_first=True,
            padding_value=self.pad_idx,
            total_length=None,
        )
        # Now we have the representation of eahc token encoded by the LSTM.
        encoded, (H, C) = self.lstm(embedded)

        # This part looks tricky. All we are doing is getting a tensor
        # That indexes the last non-PAD position in each tensor in the batch.
        last_enc_out_idxs = lens - 1
        # -> B x 1 x 1.
        last_enc_out_idxs = last_enc_out_idxs.view([encoded.size(0)] + [1, 1])
        # -> 1 x 1 x encoder_dim. This indexes the last non-padded dimension.
        last_enc_out_idxs = last_enc_out_idxs.expand(
            [-1, -1, encoded.size(-1)]
        )
        # Get the final hidden state in the LSTM
        last_hidden = torch.gather(encoded, 1, last_enc_out_idxs)
        return last_hidden

    def forward(
        self,
        symbols: torch.Tensor,
    ) -> torch.Tensor:
        encoded_sents = self.encode_text(symbols)
        output = self.classifier(encoded_sents)
        return self.log_softmax(output)
```

Evaluation

In [17]:
```
def predict(model: torch.nn.Module, dev_sequences: List[torch.Tensor]):
    preds = []
    # TODO: Get the predictions for the dev_sequences using the model
    for dev_sequence in dev_sequences:
        preds.append(torch.argmax(model(dev_sequence).squeeze(1), dim = 1))
    return preds
```

Training

In [18]:
```
from tqdm import tqdm_notebook as tqdm

import random
from util import avg_f1_score, f1_score


def training_loop(
    num_epochs,
    train_features,
    train_labels,
    dev_features,
    dev_labels,
    optimizer,
    model,
):
    print("Training...")
    loss_func = torch.nn.NLLLoss()
    batches = list(zip(train_features, train_labels))
    random.shuffle(batches)
    dev_labels = np.vstack(ten.numpy() for ten in dev_labels).flatten()
    for i in range(num_epochs):
        losses = []
        for features, labels in tqdm(batches):
            # Empty the dynamic computation graph
            optimizer.zero_grad()
            preds = model(features).squeeze(1)
            loss = loss_func(preds, labels)
            # Backpropogate the loss through our model
            loss.backward()
            optimizer.step()
            losses.append(loss.item())

        print(f"epoch {i}, loss: {sum(losses)/len(losses)}")
        # Estimate the f1 score for the development set
        print("Evaluating dev...")
        preds = predict(model, dev_features)
        preds= np.vstack(ten.numpy() for ten in preds).flatten()
        dev_f1 = f1_score(preds, dev_labels, label2i['1'])
        dev_avg_f1 = avg_f1_score(preds, dev_labels, list(label2i.values()))
        print(f"Dev F1 {dev_f1}")
        print(f"Avf Dev F1 {dev_avg_f1}")

    # Return the trained model
    return model
```

In [31]:
```
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
model = IronyDetector(
        input_dim=50,
        hidden_dim=25,
        embeddings_tensor=embeddings,
        pad_idx=word2i['<PAD>'],
        output_size=2
)

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

model = training_loop(num_epochs = 10,
                      train_features = encode_tokenized_batch_train_sentences,
                      train_labels = encode_tokenized_batch_train_labels,
                      dev_features = encode_tokenized_batch_dev_sentences ,
                      dev_labels = encode_tokenized_batch_dev_labels,
                      optimizer = optimizer,
                      model = model)
```

```
Training...
/var/folders/31/yzh9j02x7bxd463cl1x0_21h0000gn/T/ipykernel_73734/2389766617.py:20: FutureWarning: arrays to stack must be passed as a "sequence" type such as list or tuple. Support for non-sequence iterables such as generators is deprecated as of NumPy 1.16 and will raise an error in the future.
  dev_labels = np.vstack(ten.numpy() for ten in dev_labels).flatten()
/var/folders/31/yzh9j02x7bxd463cl1x0_21h0000gn/T/ipykernel_73734/2389766617.py:23: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for features, labels in tqdm(batches):
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 0, loss: 0.6887192375957966
Evaluating dev...
Dev F1 0.6471600688468158
Avf Dev F1 0.7197299837648465
/var/folders/31/yzh9j02x7bxd463cl1x0_21h0000gn/T/ipykernel_73734/2389766617.py:37: FutureWarning: arrays to stack must be passed as a "sequence" type such as list or tuple. Support for non-sequence iterables such as generators is deprecated as of NumPy 1.16 and will raise an error in the future.
  preds= np.vstack(ten.numpy() for ten in preds).flatten()
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 1, loss: 0.6603087313473225
Evaluating dev...
Dev F1 0.6978102189781021
Avf Dev F1 0.7316910664539436
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 2, loss: 0.6463813333461682
Evaluating dev...
Dev F1 0.6870229007633587
Avf Dev F1 0.7312441995602117
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 3, loss: 0.637993273884058
Evaluating dev...
Dev F1 0.6489028213166145
Avf Dev F1 0.7040213031314255
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 4, loss: 0.6271532665938139
Evaluating dev...
Dev F1 0.6827067669172934
Avf Dev F1 0.7245206038351694
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 5, loss: 0.6164442876974742
Evaluating dev...
Dev F1 0.6407766990291262
Avf Dev F1 0.7035462442514052
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 6, loss: 0.6142691244681676
Evaluating dev...
Dev F1 0.664624808575804
Avf Dev F1 0.7126402731403609
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 7, loss: 0.6071960693846147
Evaluating dev...
Dev F1 0.6867469879518071
Avf Dev F1 0.7283292461882929
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 8, loss: 0.5958467165629069
Evaluating dev...
Dev F1 0.6737481031866465
Avf Dev F1 0.7186122254107049
  0%|          | 0/240 [00:00<?, ?it/s]
epoch 9, loss: 0.583369650443395
Evaluating dev...
Dev F1 0.6779661016949152
Avf Dev F1 0.7252724958964238
```

Written Assignment (30 Points)

1. Describe what the task is, and how it could be useful.

1. implement average f1 score, it can minimize the biase for using f1-score only, espcially for imbalanced dataset
2. Create a Tokenizer with Padding, tokenizing and padding batches of inputs sentences, pad make every list of tokens is the same length, make model easier.
3. Extracting word vectors from GloVe, we load the created vocabulary, and get embeddings from the existing 1.2 million words from GloVe file, so we have generated our embeddings based on our own vocabulary.
4. Update the embeddings with oov words, xavier initialization for the embedidings of words in train, and add the oov words to the dict, assigning a new index to each. So we have a dictionary for word to index and a torch tensor containing the embeddings.
5. Make_batches, we batch train/dev set and labels along with tokenizer and encode methods. So the model can directly use these inputs
6. Get the predictions for the dev_sequences using the model, we use torch argmax to get prediction result because the model forward method return a softmax result so we need torch argmax to determine index 0 or 1 for the result.
7. Load the model and run the training loop on your train/dev splits. Set and tweak hyperparameters. Set epochs as 10 for feasible time, the parameter we can change is learning rate and batch size, please see question 5 for result table.

2. Describe, at the high level, that is, without mathematical rigor, how pretrained word embeddings like the ones we relied on here are computed. Your description can discuss the Word2Vec class of algorithms, GloVe, or a similar method.

Word2Vec is a simple neural network with a single hidden layer, and like all neural networks, it has weights, and during training, its goal is to adjust those weights to reduce a loss function. It takes as its input a large corpus of words and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space.

3. What are some of the benefits of using word embeddings instead of e.g. a bag of words?

1. They retain semantic similarity
2. They have dense vectors
3. They have a constant vector size
4. Their Vector representations are absolute
5. They have multiple embedding models

4. What is the difference between Binary Cross Entropy loss and the negative log likelihood loss we used here (`torch.nn.NLLLoss`)?

The cross_entropy combines nn.LogSoftmax() and nn.NLLLoss() in one single class. If we use NLLLoss we need to use softmax manually to transform the output.

Use NLLLoss if two-dimensional input encodes log-likelihood, it essentially performs the masking step followed by mean reduction. Use CrossEntropyLoss if two-dimensional input encodes raw prediction values that need to be activated using the softmax function

5. Show your experimental results. Indicate any changes to hyperparameters, data splits, or architectural changes you made, and how those effected results.

---	loss	Dev F1	Avf Dev F1
lr = 0.01 & batch = 8	0.5908	0.5997	0.6355
lr = 0.01 & batch = 16	0.5690	0.5749	0.6381
lr = 0.01 & batch = 4	0.6312	0.5594	0.6532
lr = 0.001 & batch = 8	0.5723	0.5932	0.6291
lr = 0.001 & batch = 16	0.5833	**0.6779**	**0.7252**
lr = 0.001 & batch = 4	**0.5553**	0.6177	0.6721

Fake News Detection in Politifact texts using WorldNews Glove Vectors and MultiClass Logistic Regression

This task contains three sections: the state-of-the-art result for the dataset; the model that achieve those results; how I would improve on the results.

There are five papers will be referenced for state-of-the-art results in this task:

1. Guo, Z., Schlichtkrull, M., & Vlachos, A. (2022). A survey on automated fact-checking. Transactions of the Association for Computational Linguistics, 10, 178-206.
2. Tariq Alhindi, Savvas Petridis, and Smaranda Muresan. 2018. Where is Your Evidence: Improving Fact-checking by Justification Modeling. In Proceedings of the First Workshop on Fact Extraction and VERification (FEVER), pages 85–90, Brussels, Belgium. Association for Computational Linguistics.
3. Hossain, Md. Muzakker & Awosaf, Zahin & Prottoy, Md. Salman & Alvy, Abu & Morol, Md. Kishor. (2022). Approaches for Improving the Performance of Fake News Detection in Bangla: Imbalance Handling and Model Stacking.
4. Upadhayay, B., & Behzadan, V. (2020, November). Sentimental LIAR: Extended Corpus and Deep Learning Models for Fake Claim Classification. In 2020 IEEE International Conference on Intelligence and Security Informatics (ISI) (pp. 1-6). IEEE.
5. H. E. Wynne and K. T. Swe, "Fake News Detection in Social Media using Two-Layers Ensemble Model," 2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), 2022, pp. 411-414, doi: 10.1109/ITC-CSCC55581.2022.9894967.

Background: This dataset is collected from fact-checking website PolitiFact through its API. It includes 12,836 human-labeled short statements, which are sampled from various contexts, such as news releases, TV or radio interviews, campaign speeches, etc. The labels for news truthfulness are fine-grained multiple classes: pants-fire, false, barely-true, half-true, mostly true, and true.

Existing data model and explanation:

Models	Valid.	Test
Majority	0.204	0.208
SVMs	0.258	0.255
Logistic Regress0ion	0.257	0.247
Bi-LSTMs	0.223	0.233
CNNs	0.260	0.270
Hybrid CNNs		
Text + Subject	0.263	0.235
Text + Speaker	**0.277**	0.248
Text + Job	0.270	0.258
Text + State	0.246	0.256
Text + Party	0.259	0.248
Text + Context	0.251	0.243
Text + History	0.246	0.241
Text + All	0.247	**0.274**

The paper introduces five baselines: a majority baseline, a regularized logistic regression classifier (LR), a support vector machine classifier (SVM), a bi-directional long short-term memory networks model (Bi-LSTMs), and a convolutional neural network model (CNNs).

SotA and Model Explanation:

Muzakker Hossain [3] introduce three feature extraction for improving the result: COUNT VECTORIZER, TF-IDF VECTORIZER, WORD EMBEDDING.

TABLE I. CLASSIFICATION METRICS WITH COUNT VECTORIZER

Classifiers	Performance evaluation			
	Accuracy	*Precision*	*Recall*	*Fscore*
SVM Linear	0.91	0.90	0.91	0.90
Logistic Regression	0.93	0.91	0.92	0.92
Decision Tree	0.82	0.80	0.81	0.80
Random Forest	0.88	0.94	0.79	0.86
XG-Boost	0.89	0.88	0.88	0.88
Gradient Boosting	0.89	0.89	0.88	0.88
Neural Network	0.94	0.94	0.93	0.93

TABLE II. CLASSIFICATION METRICS WITH TF-IDF VECTORIZER

Classifiers	Performance evaluation			
	Accuracy	*Precision*	*Recall*	*Fscore*
SVM Linear	0.94	0.93	0.93	0.93
Logistic Regression	0.93	0.93	0.91	0.92
Decision Tree	0.82	0.79	0.80	0.80
Random Forest	0.90	0.94	0.83	0.88
XG-Boost	0.89	0.89	0.88	0.88
Gradient Boosting	0.90	0.89	0.88	0.88
Neural Network	0.93	0.93	0.91	0.92

TABLE III. CLASSIFICATION METRICS WITH WORD EMBEDDING

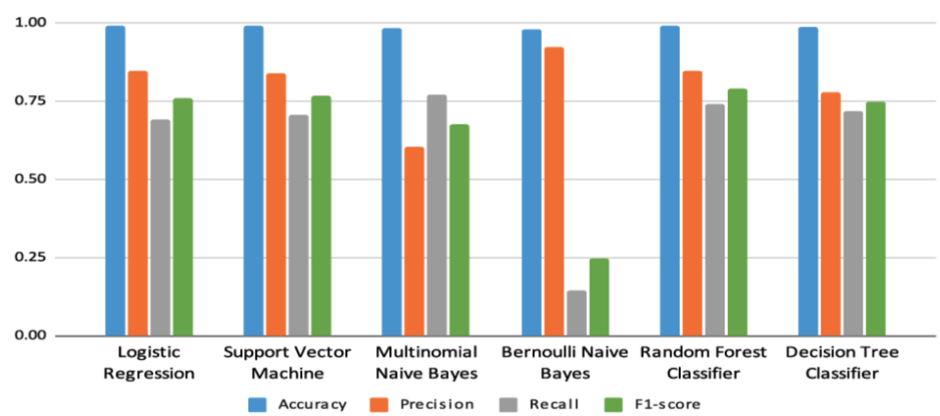
Classifiers	Performance evaluation			
	Accuracy	*Precision*	*Recall*	*Fscore*
SVM Linear	0.87	0.88	0.83	0.85
Logistic Regression	0.87	0.88	0.82	0.85
Decision Tree	0.73	0.64	0.69	0.69
Random Forest	0.84	0.85	0.79	0.82
XG-Boost	0.83	0.84	0.76	0.80
Gradient Boosting	0.83	0.84	0.76	0.80
Neural Network	0.90	0.92	0.86	0.89

Tariq Alhindi[2] extend the LIAR dataset by automatically extracting the justification from the fact-checking article used by humans to label a given claim.

Cond.	Model	Binary		Six-way	
		valid	test	valid	test
S	LR	0.58	0.61	0.23	0.25
	SVM	0.56	0.59	0.25	0.23
	BiLSTM	0.59	0.60	0.26	0.23
SJ	LR	0.68	0.67	0.37	0.37
	SVM	0.65	0.66	0.34	0.34
	BiLSTM	0.70	0.68	0.34	0.31
	P-BiLSTM	0.69	0.67	0.36	0.35
$S^{+}M$	LR	0.61	0.61	0.26	0.25
	SVM	0.57	0.60	0.26	0.25
	BiLSTM	0.62	0.62	0.27	0.25
$S^{+}MJ$	LR	0.69	0.67	0.38	0.37
	SVM	0.66	0.66	0.35	0.35
	BiLSTM	0.71	0.68	0.34	0.32
	P-BiLSTM	0.70	0.70	0.37	0.36

Table 2: Classification Results

While originally intended as an auxiliary task to improve claim verification, these justifications have been used as explanations (Atanasova et al., 2020b). Recently, Kotonya and Toni (2020b) constructed the first dataset which explicitly includes gold explanations. These consist of fact-checking articles and other news items, which can be used to train natural language generation models to provide posthoc justifications for the verdicts.



Bibek Upadhayay [4] introduce two model to improve results using BERT-Base with feed-forward Neural Network and BERT-Base with CNN for Classification.

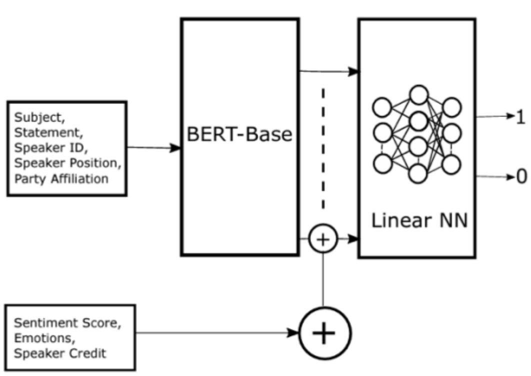


Fig. 2. BERT-Base with feed-forward component for classification

S.N.	Experiment	Accuracy	F1 Score Macro
1.	TEXT → [BB], BB_OP → [NN]	0.6882	0.5842
2.	TEXT+EMO → [BB], BB_OP → [NN]	0.6773	0.6352
3.	TEXT+EMO+ SPC → [BB], BB_OP → [NN]	0.6720	0.4021
4.	TEXT+EMO+ SPC+SEN → [BB], BB_OP → [NN]	0.6734	0.4097
5.	TEXT → [BB], BB_OP+EMP+ SPC+SEN → [NN]	0.6937	0.57234

TABLE II
BERT-BASE WITH FEED-FORWARD NN, ACCURACY AND F1 SCORE

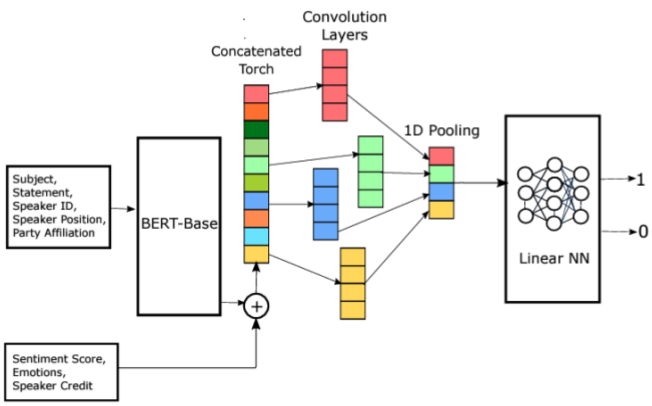


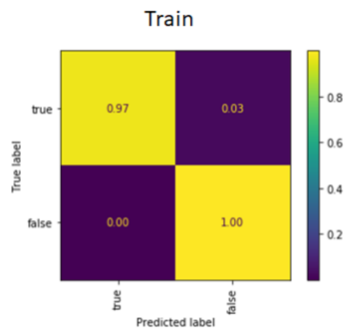
Fig. 3. BERT-Base with CNN for Classification

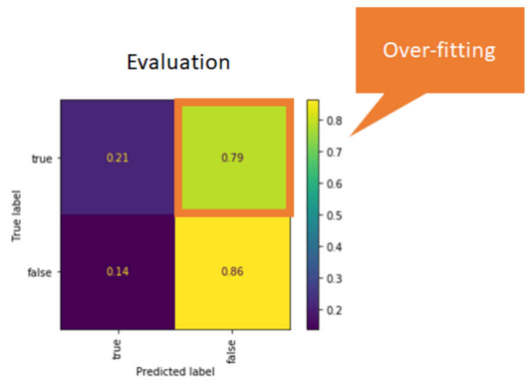
S.N.	Experiment	Accuracy	F1 Score Macro
1.	TEXT → [BB], BB_OP → [CNN]	0.6882	0.5308
2.	TEXT+EMO + SPC → [BB], BB_OP → [CNN]	0.5546	0.55641
3.	TEXT → [BB], BB_OP+ EMO → [CNN]	0.6554	0.608
4.	TEXT+SPC → [BB], BB_OP+ EMO → [CNN]	0.6890	0.6542
5.	TEXT → [BB], BB_OP+EMO +SPC → [CNN]	0.7000	0.6370
6.	TXT → [BB], BB_OP+EMO+ SPC+SEN → [CNN]	0.6992	0.6430

TABLE III
BERT-BASE + CNN, ACCURACY AND F1 SCORE

Improving on the result:

1. More baseline model:
    a. Multinomial Naïve Bayes
    b. Bernoulli Naïve Bayes
    c. Random Forest Classifier (fi-score: 0.58)



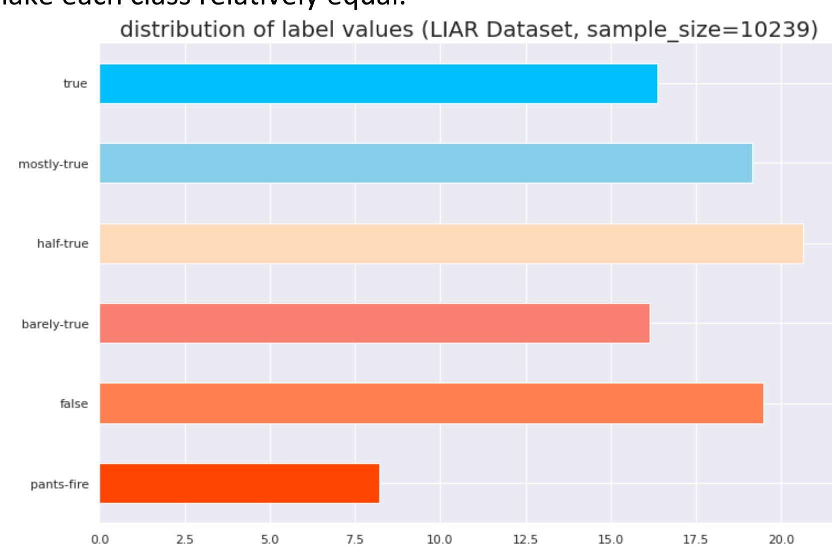


i. Over-fitting issue

d. Gradient boosting

e. RNN: recurrent neural networks

2. Oversampling/Undersampling for different class:
   It is clearly that the distribution of label values are not well balanced, especially the pants-fire label, so we need to oversampling pants-fire or Undersampling other class to make each class relatively equal.



a. Bootstrap or Random Oversampling: Generating new samples by randomly sampling current data samples with substitution

b. SMOTE: Synthetic Minority Oversampling Technique is generating samples from the minority class using k-nearest neighbor method

c. ADASYN: Adaptive Synthetic Sampling Method, more synthetic data is produced for minority class samples that are more difficult to learn than for minority class samples that are simpler to learn.
d. NearMiss: When two points in the distribution belong to different classes and are extremely close to each other, this technique eliminates the data points from the larger class, attempting to balance the distribution

3. More advanced NLP tools: part of speech analysis (Hidden Markov Model and Viterbi Algorithm), google BERT model, Named Entity Recognition with Conditional Random Fields