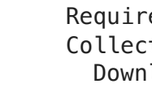
Natural Language Inference With BERT

For this homework, we will work on (NLI)[https://nlp.stanford.edu/projects/snli/].

The task is, give two sentences: a premise and a hypothesis, to classify the relation between them. We have three classes to describe this relationship.

1. Entailment: the hypothesis follows from the fact that the premise is true
2. Contradiction: the hypothesis contradicts the fact that the premise is true
3. Neutral: There is not relationship between premise and hypothesis

See below for examples



Prereqs

```
! pip install transformers datasets tqdm
```

```
Collecting transformers
  Downloading transformers-4.35.2-py3-none-any.whl (7.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.9/7.9 MB 28.9 MB/s eta 0:00:00
Collecting datasets
  Downloading datasets-2.15.0-py3-none-any.whl (521 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 521.2/521.2 kB 40.3 MB/s eta 0:00:00
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (4.66.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from transformers) (3.13.1)
Collecting huggingface-hub<1.0,>=0.16.4 (from transformers)
  Downloading huggingface_hub-0.19.4-py3-none-any.whl (311 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 311.7/311.7 kB 39.6 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (1.23.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers) (23.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (2023.6.3)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers) (2.31.0)
Collecting tokenizers<0.19,>=0.14 (from transformers)
  Downloading tokenizers-0.15.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.8/3.8 MB 60.9 MB/s eta 0:00:00
Collecting safetensors>=0.3.1 (from transformers)
  Downloading safetensors-0.4.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 38.9 MB/s eta 0:00:00
Requirement already satisfied: pyarrow>=8.0.0 in /usr/local/lib/python3.10/dist-packages (from datasets) (9.0.0)
Collecting pyarrow-hotfix (from datasets)
  Downloading pyarrow_hotfix-0.5-py3-none-any.whl (7.8 kB)
Collecting dill<0.3.8,>=0.3.0 (from datasets)
  Downloading dill-0.3.7-py3-none-any.whl (115 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 115.3/115.3 kB 11.0 MB/s eta 0:00:00
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from datasets) (1.5.3)
Requirement already satisfied: xxhash in /usr/local/lib/python3.10/dist-packages (from datasets) (3.4.1)
Collecting multiprocess (from datasets)
  Downloading multiprocess-0.70.15-py310-none-any.whl (134 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 12.1 MB/s eta 0:00:00
Requirement already satisfied: fsspec[http]<=2023.10.0,>=2023.1.0 in /usr/local/lib/python3.10/dist-packages (from datasets) (2023.6.0)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-packages (from datasets) (3.8.6)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (23.1.0)
Requirement already satisfied: charset-normalizer<4.0,>=2.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (3.3.2)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (6.0.4)
Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (4.0.3)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.9.2)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.3.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.16.4->transformers) (4.5.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2023.7.22)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas->datasets) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->datasets) (2023.3.post1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas->datasets) (1.16.0)
Installing collected packages: safetensors, pyarrow-hotfix, dill, multiprocess, huggingface-hub, tokenizers, transformers, datasets
Successfully installed datasets-2.15.0 dill-0.3.7 huggingface-hub-0.19.4 multiprocess-0.70.15 pyarrow-hotfix-0.5 safetensors-0.4.0 tokenizers-0.15.0 transformers-4.35.2
```

```
# Imports for most of the notebook
import torch
from transformers import BertModel
from transformers import AutoTokenizer
from typing import Dict, List
import random
from tqdm.autonotebook import tqdm


print(torch.cuda.is_available())
# device = torch.device("cpu")
# TODO: Uncomment the below line if you see True in the print statement
device = torch.device("cuda:0")
```

```
True
```

First let's load the Stanford NLI dataset from the huggingface datasets hub using the datasets package

Explore the dataset!

```
from datasets import load_dataset
dataset = load_dataset("snli")
print("Split sizes (num_samples, num_labels):\n", dataset.shape)
print("\nExample:\n", dataset['train'][0])
```

Downloading builder script: 100% 3.82k/3.82k [00:00<00:00, 217kB/s]

Downloading metadata: 100% 1.90k/1.90k [00:00<00:00, 129kB/s]

Downloading readme: 100% 14.1k/14.1k [00:00<00:00, 738kB/s]

Downloading: 100% 1.93k/1.93k [00:00<00:00, 118kB/s]

Each example is a dictionary with the keys: (premise, hypothesis, label).

Data Fields

- premise: a string used to determine the truthfulness of the hypothesis
- hypothesis: a string that may be true, false, or whose truth conditions may not be knowable when compared to the premise
- label: an integer whose value may be either 0, indicating that the hypothesis entails the premise, 1, indicating that the premise and hypothesis neither entail nor contradict each other, or 2, indicating that the hypothesis contradicts the premise.

Create Train, Validation and Test sets

```
from datasets import load_dataset
from collections import defaultdict

def get_snli(train=10000, validation=1000, test=1000):
    snli = load_dataset('snli')
    train_dataset = get_even_datapoints(snli['train'], train)
    validation_dataset = get_even_datapoints(snli['validation'], validation)
    test_dataset = get_even_datapoints(snli['test'], test)

    return train_dataset, validation_dataset, test_dataset

def get_even_datapoints(datapoints, n):
    random.seed(42)
    dp_by_label = defaultdict(list)
    for dp in tqdm(datapoints, desc='Reading Datapoints'):
        dp_by_label[dp['label']].append(dp)

    unique_labels = [0, 1, 2]

    split = n//len(unique_labels)

    result_datapoints = []

    for label in unique_labels:
        result_datapoints.extend(random.sample(dp_by_label[label], split))

    return result_datapoints

train_dataset, validation_dataset, test_dataset = get_snli()
```

Reading Datapoints: 100% 550152/550152 [00:41<00:00, 18828.97it/s]

Reading Datapoints: 100% 10000/10000 [00:00<00:00, 17131.92it/s]

Reading Datapoints: 100% 10000/10000 [00:00<00:00, 17390.18it/s]

```
## sub set stats
from collections import Counter

# num sample stats
print(len(train_dataset), len(validation_dataset), len(test_dataset))

# label distribution
print(Counter([t['label'] for t in train_dataset]))
print(Counter([t['label'] for t in validation_dataset]))
print(Counter([t['label'] for t in test_dataset]))

# We have a perfectly balanced dataset
```

```
9999 999 999
Counter({0: 3333, 1: 3333, 2: 3333})
Counter({0: 333, 1: 333, 2: 333})
Counter({0: 333, 1: 333, 2: 333})
```

We want a function to load samples from the huggingface dataset so that they can be batched and encoded for our model.

Now let's reimplement our tokenizer using the huggingface tokenizer.

Notice that our **call** method (the one called when we call an instance of our class) takes both a premise batch and a hypothesis batch.

The HuggingFace BERT tokenizer knows to join these with the special sentence seperator token between them. We let HuggingFace do most of the work here for making batches of tokenized and encoded sentences.

```
# Nothing to do for this class!

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

    def __init__(self, model_name='prajjwal1/bert-small'):
        """Initializes the tokenizer

        Args:
            pad_symbol (Optional[str], optional): The symbol for a pad. Defaults to "<P>".
        """
        self.hf_tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model_name = model_name

    def get_sep_token(self,):
        return self.hf_tokenizer.sep_token

    def __call__(self, prem_batch: List[str], hyp_batch: List[str]) -> List[List[str]]:
        """Uses the huggingface tokenizer to tokenize and pad a batch.

        We return a dictionary of tensors per the huggingface model specification.

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

        Returns:
            Dict: The dictionary of token specifications provided by HuggingFace
        """
        # The HF tokenizer will PAD for us, and additionally combine
        # The two sentences deimited by the [SEP] token.
        enc = self.hf_tokenizer(
            prem_batch,
            hyp_batch,
            padding=True,
            return_token_type_ids=False,
            return_tensors='pt'
        )

        return enc


# HERE IS AN EXAMPLE OF HOW TO USE THE BATCH TOKENIZER
tokenizer = BatchTokenizer()
x = tokenizer(*[["this is the first premise", "This is the second premise"], ["This is first hypothesis", "This is the second hypothesis"]])
print(x)
tokenizer.hf_tokenizer.batch_decode(x["input_ids"])
```

```
(…)wal1/bert-small/resolve/main/config.json: 100%  286/286 [00:00<00:00, 13.8kB/s]

(…)jjwal1/bert-small/resolve/main/vocab.txt: 100%  232k/232k [00:00<00:00, 2.52MB/s]

{'input_ids': tensor([[  101,  2023,  2003,  1996,  2034, 18458,   102,  2023,  2003,  2034,
         10744,   102,     0],
        [  101,  2023,  2003,  1996,  2117, 18458,   102,  2023,  2003,  1996,
          2117, 10744,   102]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}
['[CLS] this is the first premise [SEP] this is first hypothesis [SEP] [PAD]',
 '[CLS] this is the second premise [SEP] this is the second hypothesis [SEP]']
```

We can batch the train, validation, and test data, and then run it through the tokenizer

```
def generate_pairwise_input(dataset: List[Dict]) -> (List[str], List[str], List[int]):
    """
    TODO: group all premises and corresponding hypotheses and labels of the datapoints
    a datapoint as seen earlier is a dict of premis, hypothesis and label
    """
    premises = []
    hypothesis = []
    labels = []
    for x in dataset:
        premises.append(x['premise'])
        hypothesis.append(x['hypothesis'])
        labels.append(x['label'])

    return premises, hypothesis, labels


train_premises, train_hypotheses, train_labels = generate_pairwise_input(train_dataset)
validation_premises, validation_hypotheses, validation_labels = generate_pairwise_input(validation_dataset)
test_premises, test_hypotheses, test_labels = generate_pairwise_input(test_dataset)


def chunk(lst, n):
    """Yield successive n-sized chunks from lst."""
    for i in range(0, len(lst), n):
        yield lst[i:i + n]

def chunk_multi(lst1, lst2, n):
    for i in range(0, len(lst1), n):
        yield lst1[i: i + n], lst2[i: i + n]

batch_size = 16

# Notice that since we use huggingface, we tokenize and
# encode in all at once!
tokenizer = BatchTokenizer()
train_input_batches = [b for b in chunk_multi(train_premises, train_hypotheses, batch_size)]
# Tokenize + encode
train_input_batches = [tokenizer(*batch) for batch in train_input_batches]
```

Let's batch the labels, ensuring we get them in the same order as the inputs

```
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])


train_label_batches = [b for b in chunk(train_labels, batch_size)]
train_label_batches = [encode_labels(batch) for batch in train_label_batches]
```

Now we implement the model. Notice the TODO and the optional TODO (read why you may want to do this one.)

```
class NLIClassifier(torch.nn.Module):
    def __init__(self, output_size: int, hidden_size: int, model_name='prajjwal1/bert-small'):
        super().__init__()
        self.output_size = output_size
        self.hidden_size = hidden_size

        # Initialize BERT, which we use instead of a single embedding layer.
        self.bert = BertModel.from_pretrained(model_name)

        # TODO [OPTIONAL]: Updating all BERT parameters can be slow and memory intensive.
        # Freeze them if training is too slow. Notice that the learning
        # rate should probably be smaller in this case.
        # Uncommenting out the below 2 lines means only our classification layer will be updated.

        for param in self.bert.parameters():
            param.requires_grad = False

        self.bert_hidden_dimension = self.bert.config.hidden_size

        # TODO: Add an extra hidden layer in the classifier, projecting
        #      from the BERT hidden dimension to hidden size. Hint: torch.nn.Linear()

        self.hidden_layer = torch.nn.Linear(self.bert_hidden_dimension, self.hidden_size)

        # TODO: Add a relu nonlinearity to be used in the forward method
        #      https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

        self.relu = torch.nn.ReLU()

        self.classifier = torch.nn.Linear(self.hidden_size, self.output_size)
        self.log_softmax = torch.nn.LogSoftmax(dim=2)

    def encode_text(
        self,
        symbols: Dict
    ) -> torch.Tensor:
        """Encode the (batch of) sequence(s) of token symbols BERT.
            Then, get CLS represenation.

        Args:
            symbols (Dict): The Dict of token specifications provided by the HuggingFace tokenizer

        Returns:
            torch.Tensor: CLS token embedding
        """
        # First we get the contextualized embedding for each input symbol
        # We no longer need an LSTM, since BERT encodes context and
        # gives us a single vector describing the sequence in the form of the [CLS] token.
        encoded_sequence = self.bert(**symbols)
        # TODO: Get the [CLS] token
        #      The BertModel output. See here: https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertModel
        #      and check the returns for the forward method.
        # We want to return a tensor of the form batch_size x 1 x bert_hidden_dimension
        # print(encoded_sequence.last_hidden_state.shape)
        cls_token_embedding = encoded_sequence.last_hidden_state[:, 0, :]
        # Return only the first token's embedding from the last_hidden_state. Hint: using list slices
        return cls_token_embedding.unsqueeze(1)

    def forward(
        self,
        symbols: Dict,
    ) -> torch.Tensor:
        """_summary_

        Args:
            symbols (Dict): The Dict of token specifications provided by the HuggingFace tokenizer

        Returns:
            torch.Tensor: _description_
        """
        encoded_sents = self.encode_text(symbols)
        output = self.hidden_layer(encoded_sents)
        output = self.relu(output)
        output = self.classifier(output)
        return self.log_softmax(output)


# For making predictions at test time
def predict(model: torch.nn.Module, sents: torch.Tensor) -> List:
    logits = model(sents.to(device)).to(device)
    return list(torch.argmax(logits.cpu(), axis=2).squeeze().numpy())
```

Evaluation metrics: Macro F1

```
import numpy as np
from numpy import sum as t_sum
from numpy import logical_and


def precision(predicted_labels, true_labels, which_label=1):
    """
    Precision is True Positives / All Positives Predictions
    """
    pred_which = np.array([pred == which_label for pred in predicted_labels])
    true_which = np.array([lab == which_label for lab in true_labels])
    denominator = t_sum(pred_which)
    if denominator:
        return t_sum(logical_and(pred_which, true_which))/denominator
    else:
        return 0.


def recall(predicted_labels, true_labels, which_label=1):
    """
    Recall is True Positives / All Positive Labels
    """
    pred_which = np.array([pred == which_label for pred in predicted_labels])
    true_which = np.array([lab == which_label for lab in true_labels])
    denominator = t_sum(true_which)
    if denominator:
        return t_sum(logical_and(pred_which, true_which))/denominator
    else:
        return 0.


def f1_score(
    predicted_labels: List[int],
    true_labels: List[int],
    which_label: int
):
    """
    F1 score is the harmonic mean of precision and recall
    """
    P = precision(predicted_labels, true_labels, which_label=which_label)
    R = recall(predicted_labels, true_labels, which_label=which_label)

    if P and R:
        return 2*P*R/(P+R)
    else:
        return 0.


def macro_f1(
    predicted_labels: List[int],
    true_labels: List[int],
    possible_labels: List[int],
    label_map=None
):
    converted_prediction = [label_map[int(x)] for x in predicted_labels] if label_map else predicted_labels
    scores = [f1_score(converted_prediction, true_labels, l) for l in possible_labels]
    # Macro, so we take the uniform avg.
    return sum(scores) / len(scores)
```

Training loop.

```
def training_loop(
    num_epochs,
    train_features,
    train_labels,
    dev_sents,
    dev_labels,
    optimizer,
    model,
):
    print("Training...")
    loss_func = torch.nn.NLLLoss()
    batches = list(zip(train_features, train_labels))
    random.shuffle(batches)
    for i in range(num_epochs):
        losses = []
        for features, labels in tqdm(batches):
            # Empty the dynamic computation graph
            optimizer.zero_grad()
            preds = model(features.to(device)).squeeze(1)
            loss = loss_func(preds, labels.to(device))
            # 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...")
        all_preds = []
        all_labels = []
        for sents, labels in tqdm(zip(dev_sents, dev_labels), total=len(dev_sents)):
            pred = predict(model, sents)
            all_preds.extend(pred)
            all_labels.extend(list(labels.cpu().numpy()))

        dev_f1 = macro_f1(all_preds, all_labels, [0, 1, 2])
        print(f"Dev F1 {dev_f1}")

    # Return the trained model
    return model


# You can increase epochs if need be
epochs = 40

# TODO: Find a good learning rate and hidden size
LR = 0.0001
hidden_size = 10

possible_labels = set(train_labels)
model = NLIClassifier(output_size=len(possible_labels), hidden_size=hidden_size)
```

```
model.to(device)
optimizer = torch.optim.AdamW(model.parameters(), LR)

batch_tokenizer = BatchTokenizer()

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]

# Tokenize + encode
validation_input_batches = [batch_tokenizer(*batch) for batch in validation_input_batches]
validation_batch_labels = [b for b in chunk(validation_labels, batch_size)]
validation_batch_labels = [encode_labels(batch) for batch in validation_batch_labels]

training_loop(
    epochs,
    train_input_batches,
    train_label_batches,
    validation_input_batches,
    validation_batch_labels,
    optimizer,
    model,
)
```

```
Training...
```
100% 625/625 [00:07<00:00, 87.28it/s]
```
epoch 0, loss: 1.097312852668762
Evaluating dev...
```
100% 63/63 [00:00<00:00, 109.09it/s]
```
Dev F1 0.40836113835216586
```
100% 625/625 [00:06<00:00, 115.30it/s]
```
```
epoch 1, loss: 1.0728248240470886
Evaluating dev...
```
100% 63/63 [00:00<00:00, 119.80it/s]
```
Dev F1 0.43575302162301854
```
100% 625/625 [00:05<00:00, 114.52it/s]
```
```
epoch 2, loss: 1.0527944693565368
Evaluating dev...
```
100% 63/63 [00:00<00:00, 127.26it/s]
```
Dev F1 0.43842869505546433
```
100% 625/625 [00:06<00:00, 84.64it/s]
```
```
epoch 3, loss: 1.0374208548545838
Evaluating dev...
```
100% 63/63 [00:00<00:00, 96.55it/s]
```
Dev F1 0.4481222188173606
```
100% 625/625 [00:05<00:00, 115.46it/s]
```
```
epoch 4, loss: 1.0250985927581786
Evaluating dev...
```
100% 63/63 [00:00<00:00, 125.63it/s]
```
Dev F1 0.46905523666548105
```
100% 625/625 [00:05<00:00, 104.00it/s]
```
```
epoch 5, loss: 1.0153348851203918
Evaluating dev...
```
100% 63/63 [00:00<00:00, 103.31it/s]
```
Dev F1 0.4579154223732537
```
100% 625/625 [00:06<00:00, 121.53it/s]
```
```
epoch 6, loss: 1.0078289861679077
Evaluating dev...
```
100% 63/63 [00:00<00:00, 126.80it/s]
```
Dev F1 0.48763826122372084
```
100% 625/625 [00:05<00:00, 117.71it/s]
```
```
epoch 7, loss: 1.000646111011505
Evaluating dev...
```
100% 63/63 [00:00<00:00, 122.60it/s]
```
Dev F1 0.49600474082283297
```
100% 625/625 [00:05<00:00, 96.46it/s]
```
```
epoch 8, loss: 0.9948511468887329
Evaluating dev...
```
100% 63/63 [00:00<00:00, 125.45it/s]
```
Dev F1 0.5008522962974137
```
100% 625/625 [00:06<00:00, 118.01it/s]
```
```
epoch 9, loss: 0.9896235171318054
Evaluating dev...
```
100% 63/63 [00:00<00:00, 127.25it/s]
```
Dev F1 0.5055315592659991
```
100% 625/625 [00:05<00:00, 122.72it/s]
```
```
epoch 10, loss: 0.9849632378578186
Evaluating dev...
```
100% 63/63 [00:00<00:00, 131.09it/s]
```
Dev F1 0.5089966732987328
```
100% 625/625 [00:06<00:00, 82.73it/s]
```
```
epoch 11, loss: 0.9807702584266662
Evaluating dev...
```
100% 63/63 [00:00<00:00, 101.09it/s]
```
Dev F1 0.5115116317352455
```
100% 625/625 [00:05<00:00, 124.14it/s]
```
```
epoch 12, loss: 0.976937613773346
Evaluating dev...
```
100% 63/63 [00:00<00:00, 126.03it/s]
```
Dev F1 0.5134668202463019
```
100% 625/625 [00:05<00:00, 102.92it/s]
```
```
epoch 13, loss: 0.9734506892204284
Evaluating dev...
```
100% 63/63 [00:00<00:00, 107.27it/s]
```
Dev F1 0.517433780975072
```
100% 625/625 [00:06<00:00, 114.60it/s]
```
```
epoch 14, loss: 0.9701675902366638
Evaluating dev...
```
100% 63/63 [00:00<00:00, 122.86it/s]
```
Dev F1 0.5187565519005393
```
100% 625/625 [00:05<00:00, 114.82it/s]
```
```
epoch 15, loss: 0.9670367488861084
Evaluating dev...
```
100% 63/63 [00:00<00:00, 124.02it/s]
```
Dev F1 0.5205251459625017
```
100% 625/625 [00:06<00:00, 86.04it/s]
```
```
epoch 16, loss: 0.9641344951629639
Evaluating dev...
```
100% 63/63 [00:00<00:00, 111.90it/s]
```
Dev F1 0.5214686795398046
```
100% 625/625 [00:06<00:00, 120.23it/s]
```
```
epoch 17, loss: 0.9613689749717712
Evaluating dev...
```
100% 63/63 [00:00<00:00, 121.42it/s]
```
Dev F1 0.5215130160548425
```
100% 625/625 [00:05<00:00, 112.65it/s]
```
```
epoch 18, loss: 0.9587447636604309
Evaluating dev
```

```
Evaluating dev...
```
100% 63/63 [00:00<00:00, 126.88it/s]
```
Dev F1 0.5236197370243185
```
100% 625/625 [00:06<00:00, 84.59it/s]
```
epoch 19, loss: 0.9562762427330017
Evaluating dev...
```
100% 63/63 [00:00<00:00, 92.26it/s]
```
Dev F1 0.5227544666077187
```
100% 625/625 [00:05<00:00, 109.63it/s]
```
epoch 20, loss: 0.9538764311790466
Evaluating dev...
```
100% 63/63 [00:00<00:00, 124.26it/s]
```
Dev F1 0.5216824453023359
```
100% 625/625 [00:05<00:00, 100.09it/s]
```
epoch 21, loss: 0.9516332839012146
Evaluating dev...
```
100% 63/63 [00:00<00:00, 110.66it/s]
```
Dev F1 0.5236656535488264
```
100% 625/625 [00:06<00:00, 114.43it/s]
```
epoch 22, loss: 0.9494729369163514
Evaluating dev...
```
100% 63/63 [00:00<00:00, 124.65it/s]
```
Dev F1 0.5217824106204634
```
100% 625/625 [00:06<00:00, 80.58it/s]
```
epoch 23, loss: 0.94742669506073
Evaluating dev...
```
100% 63/63 [00:00<00:00, 96.82it/s]
```
Dev F1 0.524698969843478
```
100% 625/625 [00:07<00:00, 78.22it/s]
```
epoch 24, loss: 0.9453948566436767
Evaluating dev...
```
100% 63/63 [00:00<00:00, 87.10it/s]
```
Dev F1 0.5256169787277479
```
100% 625/625 [00:05<00:00, 124.96it/s]
```
epoch 25, loss: 0.9434526980400085
Evaluating dev...
```
100% 63/63 [00:00<00:00, 130.63it/s]
```
Dev F1 0.5277655843934727
```
100% 625/625 [00:05<00:00, 93.10it/s]
```
epoch 26, loss: 0.9415931610107422
Evaluating dev...
```
100% 63/63 [00:00<00:00, 106.13it/s]
```
Dev F1 0.5266499596912614
```
100% 625/625 [00:06<00:00, 120.49it/s]
```
epoch 27, loss: 0.9397716620445251
Evaluating dev...
```
100% 63/63 [00:00<00:00, 123.42it/s]
```
Dev F1 0.5255751872214796
```
100% 625/625 [00:05<00:00, 112.66it/s]
```
epoch 28, loss: 0.9380079201698304
Evaluating dev...
```
100% 63/63 [00:00<00:00, 126.27it/s]
```
Dev F1 0.5246047439854541
```
100% 625/625 [00:06<00:00, 99.51it/s]
```
epoch 29, loss: 0.9363240198135376
Evaluating dev...
```
100% 63/63 [00:00<00:00, 102.41it/s]
```
Dev F1 0.5249031702620811
```
100% 625/625 [00:06<00:00, 114.98it/s]
```
epoch 30, loss: 0.9347326590538025
Evaluating dev...
```
100% 63/63 [00:00<00:00, 125.40it/s]
```
Dev F1 0.5268454283782212
```
100% 625/625 [00:05<00:00, 114.78it/s]
```
epoch 31, loss: 0.9331026349067688
Evaluating dev...
```
100% 63/63 [00:00<00:00, 125.49it/s]
```
Dev F1 0.526944043456965
```
100% 625/625 [00:06<00:00, 88.98it/s]
```
epoch 32, loss: 0.9315242610931397
Evaluating dev...
```
100% 63/63 [00:00<00:00, 120.23it/s]
```
# TODO: Get a final macro F1 on the test set.
# You should be able to mimic what we did with the validaiton set.
batch_tokenizer = BatchTokenizer()

test_input_batches = [b for b in chunk_multi(test_premises, test_hypotheses, batch_size)]

# Tokenize + encode
test_input_batches = [batch_tokenizer(*batch) for batch in test_input_batches]
test_batch_labels = [b for b in chunk(test_labels, batch_size)]
test_batch_labels = [encode_labels(batch) for batch in test_batch_labels]

all_preds = []
all_labels = []
# test_premises, test_hypotheses, test_labels = generate_pairwise_input(test_dataset)
# print(test_premises)
# print(test_input_batches)
# print(test_batch_labels)
for sents, labels in tqdm(zip(test_input_batches, test_batch_labels), total=len(test_input_batches)):
    pred = predict(model, sents)
    all_preds.extend(pred)
    all_labels.extend(list(labels.cpu().numpy()))

dev_f1 = macro_f1(all_preds, all_labels, [0, 1, 2])
print(f"Test F1 {dev_f1}")
```

```
100%                                                63/63 [00:00<00:00, 111.09it/s]
Evaluating dev...
```

```
import torch

# Assume 'model' is your PyTorch model
torch.save(model.state_dict(), 'bert-snli.pth')
```

```
    epoch 38, loss: 0.9230571282386078
```

Evaluation of Pretrained Model

Huggingface also hosts models which users have already trained on SNLI -- we can load them here and evaluate their performance on the validation and test set.

These models include the BertModel which we were using before, but also the trained weights for the classifier layer. Because of this, we'll use the standard HuggingFace classification model instead of the classifier we used above, and modify the training and prediction functions to handle this correctly.

Try and find the differences between the training loop. One addition is the new usage of "label_map". Why may this be necessary?

```
          (word_embeddings): Embedding(30522, 512, padding_idx=0)
```

```
def class_predict(model, sents):

    with torch.inference_mode():

        logits = model(**sents.to(device)).logits
        predictions = torch.argmax(logits, axis=1)
        return predictions

def prediction_loop(model, dev_sents, dev_labels, label_map=None):
    print("Evaluating...")
    all_preds = []
    all_labels = []
    for sents, labels in tqdm(zip(dev_sents, dev_labels), total=len(dev_sents)):
        pred = class_predict(model, sents).cpu()
        all_preds.extend(pred)
        all_labels.extend(list(labels.cpu().numpy()))

    # print(all_preds)
    # print(all_labels)

    dev_f1 = macro_f1(all_preds, all_labels, possible_labels=set(all_labels), label_map = label_map)
    print(f"F1 {dev_f1}")

def class_training_loop(
    num_epochs,
    train_features,
    train_labels,
    dev_sents,
    dev_labels,
    optimizer,
    model,
    label_map=None
):
    print("Training...")
    loss_func = torch.nn.CrossEntropyLoss()
    batches = list(zip(train_features, train_labels))
    random.shuffle(batches)
    for i in range(num_epochs):
        losses = []
        for features, labels in tqdm(batches):
            # Empty the dynamic computation graph
            optimizer.zero_grad()
            logits = model(**features.to(device)).logits
            # preds = torch.argmax(logits, axis=1)
            loss = loss_func(logits, 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...")
    all_preds = []
    all_labels = []
    for sents, labels in tqdm(zip(dev_sents, dev_labels), total=len(dev_sents)):
        pred = predict(model, sents).cpu()
        all_preds.extend(pred)
        all_labels.extend(list(labels.numpy()))

    all_preds
    dev_f1 = macro_f1(all_preds, all_labels, possible_labels=set(all_labels), label_map = label_map)
    print(f"Dev F1 {dev_f1}")

    # Return the trained model
    return model
```

Now we can load a model and re-tokenize our data.

```
## TODO: Get the label_map
## For the snli dataset  0=Entailment, 1=Neutral, 2=Contradiction
label_map = {0: 2, 1: 0, 2: 1}
```

```
from transformers import  BertForSequenceClassification, BertTokenizer


model_name = 'textattack/bert-base-uncased-snli'
tokenizer_model_name = 'textattack/bert-base-uncased-snli' # This is sometimes different from model_name, but should normally be the same

model =  BertForSequenceClassification.from_pretrained(model_name)

model.to(device)

batch_tokenizer = BatchTokenizer(model_name = tokenizer_model_name)

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]

# Tokenize + encode
validation_input_batches = [batch_tokenizer(*batch) for batch in validation_input_batches]
validation_batch_labels = [b for b in chunk(validation_labels, batch_size)]
validation_batch_labels = [encode_labels(batch) for batch in validation_batch_labels]

prediction_loop(model, validation_input_batches, validation_batch_labels, label_map=label_map)
```

```
(…)se-uncased-snli/resolve/main/config.json: 100%  630/630 [00:00<00:00, 17.5kB/s]
pytorch_model.bin: 100%  438M/438M [00:01<00:00, 249MB/s]
(…)base-uncased-snli/resolve/main/vocab.txt: 100%  232k/232k [00:00<00:00, 2.62MB/s]
Evaluating...
100%  63/63 [00:02<00:00, 23.46it/s]
F1 0.6671987159037203
```

To complete this section, find 2 other BERT-based models which have been trained on natural language inference and evaluate them on the dev set. Note the scores for each model and any high-level differences you can find between the models (architecture, sizes, training data, etc.)

If you don't have access to a GPU, inference may be slow, particularly for larger models. In this case, take a sample of the validation set; the size should be large enough such that all labels are covered, and a score will still be meaningful, but also so that inference doesn't take more than 3-5 minutes.

```
from transformers import  BertForSequenceClassification, BertTokenizer


model_name = 'boychaboy/SNLI_bert-base-uncased'
tokenizer_model_name = 'boychaboy/SNLI_bert-base-uncased' # This is sometimes different from model_name, but should normally be the same

model =  BertForSequenceClassification.from_pretrained(model_name)

model.to(device)

batch_tokenizer = BatchTokenizer(model_name = tokenizer_model_name)

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]

# Tokenize + encode
validation_input_batches = [batch_tokenizer(*batch) for batch in validation_input_batches]
validation_batch_labels = [b for b in chunk(validation_labels, batch_size)]
validation_batch_labels = [encode_labels(batch) for batch in validation_batch_labels]

prediction_loop(model, validation_input_batches, validation_batch_labels, label_map=None)
```

```
(…)rt-base-uncased/resolve/main/config.json: 100%  825/825 [00:00<00:00, 57.6kB/s]
pytorch_model.bin: 100%  438M/438M [00:01<00:00, 260MB/s]
(…)cased/resolve/main/tokenizer_config.json: 100%  285/285 [00:00<00:00, 18.3kB/s]
(…)bert-base-uncased/resolve/main/vocab.txt: 100%  232k/232k [00:00<00:00, 1.33MB/s]
(…)base-uncased/resolve/main/tokenizer.json: 100%  466k/466k [00:00<00:00, 1.84MB/s]
(…)sed/resolve/main/special_tokens_map.json: 100%  112/112 [00:00<00:00, 8.07kB/s]
Evaluating...
100%  63/63 [00:02<00:00, 23.65it/s]
F1 0.6709028992295
```

```
from transformers import  BertForSequenceClassification, BertTokenizer


model_name = 'emrecan/bert-base-multilingual-cased-snli_tr'
tokenizer_model_name = 'emrecan/bert-base-multilingual-cased-snli_tr' # This is sometimes different from model_name, but should normally be the same

model =  BertForSequenceClassification.from_pretrained(model_name)

model.to(device)

batch_tokenizer = BatchTokenizer(model_name = tokenizer_model_name)

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]

# Tokenize + encode
validation_input_batches = [batch_tokenizer(*batch) for batch in validation_input_batches]
validation_batch_labels = [b for b in chunk(validation_labels, batch_size)]
validation_batch_labels = [encode_labels(batch) for batch in validation_batch_labels]

prediction_loop(model, validation_input_batches, validation_batch_labels, label_map={0:0, 1:1, 2:2})
```

```
(…)l-cased-snli_tr/resolve/main/config.json: 100%  1.11k/1.11k [00:00<00:00, 59.3kB/s]
pytorch_model.bin: 100%  712M/712M [00:17<00:00, 42.1MB/s]
(…)li_tr/resolve/main/tokenizer_config.json: 100%  341/341 [00:00<00:00, 21.5kB/s]
(…)ual-cased-snli_tr/resolve/main/vocab.txt: 100%  996k/996k [00:00<00:00, 8.40MB/s]
(…)ased-snli_tr/resolve/main/tokenizer.json: 100%  1.96M/1.96M [00:00<00:00, 11.5MB/s]
(…)_tr/resolve/main/special_tokens_map.json: 100%  112/112 [00:00<00:00, 5.43kB/s]
Evaluating...
100%  63/63 [00:03<00:00, 22.02it/s]
F1 0.20654891478929152
```

Written Assignment

1. Describe the task and what capability is required to solve it.

2. How does the method of encoding sequences of words in our model differ here, compared to the word embeddings in HW 4. What is different? Why benefit does this method have?

3. Discuss your results. Did you do any hyperparameter tuning? Did the model solve the task?

1.

The task for this part of the assignment is Natural Language Inference (NLI) also known as Recognizing Textual Entailment (RTE). The goal is to determine the relationship between two given sentences: a premise and a hypothesis. We are looking at three relationships:

- Entailment: The hypothesis logically follows from the fact that the premise is true.
- Contradiction: The hypothesis contradicts the fact that the premise is true.
- Neutral: There is no significant relationship between the premise and the hypothesis.

The technical capabilities needed by the model are:

- Transfer Learning: We are training out model on top of a pretrained BERT model while freezing some layers
- Word Encoding: We are converting the words from strings to a integer vector to represent its meaning
- Fine Tuning: We have to fine tune the hyperparameters to obtain the optimal model

2.

The major difference in encoding between the two parts of assignments is the incorporation of context in the Bert Tokenizer. While we use Word2Vec in the other part that only allows for each word to be represented by a different vector however the word itself only has one vector regardless of the context. Meanwhile the BERT tokenizer takes into account the context to create unique embedding for the word. This leads to the same word having different embedding depending on the context. This method allows for a better understanding of the sense of the word as the context can change the meaning of a word. For example, we could use "beam" in multiple ways, it could be used to refer to a metalic component used in construction such as "Lift the beam carefully, make sure it does not fall", or it could be used to refer to an laser, such that "The Death Star's laser beam obliterated Alderan". The sense of beam is very different in both the sentences which is better encapsulated through the BERT tokeniser.

3.

We have obtained a macro f1 score of 0.53 on the test set. I tuned the hyperparameters to obtain this as the best case. Upon the model having too large a hidden layer or a large learning rate the model seems to get stuck at an f1 score of 0.167 due to only printing 0. However upon fine tuning the model I was able to get past that the model showed marked improvement since then. However it does stop improving near a f1 score of 0.5.

I also looked into other models, while the model provided to us gave an f1 score of 0.667, the model I found by boychaboy got an f1 score of 0.67 while the model by emrecan only gave an f1 score of 0.2. The model given had a different label map than ours and hence I had to pass that, while the model by boychaboy had the same label map. Overall these models perform better that mine, this could be due to them having more resources and time to train the model, while also using the entire bert model instead of freezing some of it like I did due to limited resources.

```
print(model)

    BertForSequenceClassification(
      (bert): BertModel(
        (embeddings): BertEmbeddings(
          (word_embeddings): Embedding(119547, 768, padding_idx=0)
          (position_embeddings): Embedding(512, 768)
          (token_type_embeddings): Embedding(2, 768)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (encoder): BertEncoder(
          (layer): ModuleList(
```

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 [35]:
```
# This is so that you don't have to restart the kernel everytime you edit hmm.py

%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Data

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

In [1]:
```
from irony import load_datasets
from sklearn.model_selection import train_test_split

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

# TODO: Split train into train/dev
train_sentences, dev_sentences, train_labels, dev_labels = train_test_split(train_sentences, train_labels, test_size=0.2, random_state=42)

print("Train set size:", len(train_sentences))
print("Dev set size:", len(dev_sentences))
print("Test set size:", len(test_sentences))
```

```
Train set size: 3067
Dev set size: 767
Test set size: 784
```

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 [2]:
```
from irony import run_nb_baseline

run_nb_baseline()
```

```
Vectorizing Text: 100%|██████████| 3834/3834 [00:00<00:00, 13717.27it/s]
Vectorizing Text: 100%|██████████| 3834/3834 [00:00<00:00, 20228.34it/s]
Vectorizing Text: 100%|██████████| 784/784 [00:00<00:00, 25013.38it/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 [2]:
```
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:
            tokens = self.nlp(sentence)
            tokens = [token.text for token in tokens]
            tokens.insert(0, "<SOS>")
            tokens.append("<EOS>")
            tokenized_sentences.append(tokens)

        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
        max_length = max(len(s) for s in batch)

        padded_batch = []
        for sentence in batch:
            padded_sentence = sentence.copy()
            padded_sentence.extend([self.pad_symbol] * (max_length - len(padded_sentence)))
            padded_batch.append(padded_sentence)

        return padded_batch
```

In [3]:
```
# 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 + dev_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 [5]:
```
# 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
```

```
--2023-11-14 13:20:13--  https://nlp.stanford.edu/data/glove.twitter.27B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.twitter.27B.zip [following]
--2023-11-14 13:20:13--  https://downloads.cs.stanford.edu/nlp/data/glove.twitter.27B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1520408563 (1.4G) [application/zip]
Saving to: ‘glove.twitter.27B.zip.1’

glove.twitter.27B.z  10%[=>                  ] 146.11M  1.66MB/s    eta 10m 47s^C
```

In [6]:
```
# unzip glove.twitter.27B.zip
# if there is an error, please download the zip file again

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

```
Archive:  glove.twitter.27B.zip
replace glove.twitter.27B.25d.txt? [y]es, [n]o, [A]ll, [N]one, [r]ename: ^C
```

In [7]:
```
# Let's see what files are there:

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

```
glove.twitter.27B.100d.txt
glove.twitter.27B.200d.txt
glove.twitter.27B.25d.txt
glove.twitter.27B.50d.txt
```

In [4]:
```
# 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

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

In [ ]:
```
```

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 [5]:
```
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(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
                vector_tensor = torch.tensor([float(weight) for weight in weights])
                vectors.append(vector_tensor)
                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 [6]:
```
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 [7]:
```
from torch.nn.init import xavier_uniform_

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
    weights = torch.FloatTensor(num_embeddings, dim).normal_()
    xavier_uniform_(weights)
    return weights


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
    new_word2i = {word: i + len(glove_word2i) for i, word in enumerate(oovs)}
    glove_word2i.update(new_word2i)

    oov_embeddings = intialize_new_embedding_weights(len(oovs), dim=glove_embeddings.size(1))
    new_embeddings = torch.cat((glove_embeddings, oov_embeddings), dim=0)


    return glove_word2i, new_embeddings
```

In [8]:
```
def make_batches(sequences: List[str], batch_size: int) -> List[List[str]]:
    """Yield batch_size chunks from sequences."""
    # TODO
    batches = []
    current_batch = []

    for sequence in sequences:
        current_batch.append(sequence)

        if len(current_batch) == batch_size:
            batches.append(current_batch)
            current_batch = []

    if current_batch:
        batches.append(current_batch)

    return batches

def make_label_batches(labels: List[int], batch_size: int) -> List[List[int]]:
    """Yield batch_size chunks from labels."""
    batches = []
    current_batch = []

    for label in labels:
        current_batch.append(label)
        if len(current_batch) == batch_size:
```

```
            batches.append(current_batch)
            current_batch = []

    if current_batch:
        batches.append(current_batch)

    return batches


# TODO: Set your preferred batch size
batch_size = 8
tokenizer = Tokenizer()

# We make batches now and use those.
batch_tokenized = []
batch_labels=make_label_batches(train_labels, batch_size)
# 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):
    batch_tokenized.append(tokenizer(batch))


# We make batches now and use those.
dev_batch_tokenized = []
dev_batch_labels=make_label_batches(dev_labels, batch_size)
# 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(dev_sentences, batch_size):
    dev_batch_tokenized.append(tokenizer(batch))

# We make batches now and use those.
test_batch_tokenized = []
test_batch_labels=make_label_batches(test_labels, batch_size)
# 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(test_sentences, batch_size):
    test_batch_tokenized.append(tokenizer(batch))

glove_word2i, glove_embeddings = read_pretrained_embeddings(
    embedding_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 [9]:
```
# 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])
```

Modeling (7 Points)

In [10]:
```
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.Embedding.from_pretrained
        self.embeddings = torch.nn.Embedding.from_pretrained(
            embeddings_tensor, padding_idx=pad_idx, freeze=True
        )
        # 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).
        # print(embedded.shape)
        # print(packed)
        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:
        # print("starting")
        encoded_sents = self.encode_text(symbols)
        # print("encoded")
        output = self.classifier(encoded_sents)
        return self.log_softmax(output)
```

Evaluation

In [11]:
```
def predict(model: torch.nn.Module, dev_sequences: List[torch.Tensor]):
    preds = []
    # TODO: Get the predictions for the dev_sequences using the model
    # Set the model to evaluation mode
    model.eval()
    with torch.no_grad():
        for batch in dev_sequences:
            predictions = model(batch)
            # print(predictions)
            _, predicted_labels = torch.max(predictions, 2)
            predicted_labels = predicted_labels.squeeze().tolist()
            preds.extend(predicted_labels)
    # print(preds)
    return preds
```

Training

In [33]:
```
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)
    for i in range(num_epochs):
        losses = []
        for features, labels in tqdm(batches):
            # Empty the dynamic computation graph
            optimizer.zero_grad()
            # print(features.shape)
            preds = model(features).squeeze(1)
            # print(preds)
            # print(labels)
            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)
        # print(preds)
        # print(dev_labels)
        dev_f1 = f1_score(preds, dev_labels, label2i['1'])
        dev_avg_f1 = avg_f1_score(preds, dev_labels, [0, 1])
        print(f"Dev F1 {dev_f1}")
        print(f"Avf Dev F1 {dev_avg_f1}")

    # Return the trained model
    return model
```

In [13]:
```
encoded_train_data   = []
encoded_train_labels = []
encoded_dev_data     = []
encoded_dev_labels   = []
encoded_test_data    = []
encoded_test_labels  = []


for x in batch_tokenized:
    encoded_train_data.append(encode_sentences(x, word2i))

for x in batch_labels:
    # print(len(x))
    encoded_train_labels.append(encode_labels(x))
```

```
for x in dev_batch_tokenized:
    encoded_dev_data.append(encode_sentences(x, word2i))

for x in test_batch_tokenized:
    encoded_test_data.append(encode_sentences(x, word2i))

encoded_dev_labels = [int(x) for x in dev_labels]
encoded_test_labels = [int(x) for x in test_labels]

# print(len(encoded_train_labels))
# print(encoded_dev_labels)
# print(train_data[0].shape)
```

In [36]:
```
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
LR=0.001
model = IronyDetector(
    input_dim=50,
    hidden_dim=10,
    embeddings_tensor=embeddings,
    pad_idx=-1,
    output_size=2,
)

optimizer = torch.optim.Adam(model.parameters(), LR)
trained_model = training_loop(
    num_epochs=20,
    train_features=encoded_train_data,
    train_labels=encoded_train_labels,
    dev_features=encoded_dev_data,
    dev_labels=encoded_dev_labels,
    optimizer=optimizer,
    model=model,
)
```

```
Training...
/var/folders/jd/yz20r3xj2x710fkjwd2jwntm0000gn/T/ipykernel_41791/4214819134.py:22: 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%|          | 0/384 [00:00<?, ?it/s]
epoch 0, loss: 0.6942745245372256
Evaluating dev...
Dev F1 0.6799307958477508
Avf Dev F1 0.350547408505886
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 1, loss: 0.6934807274180154
Evaluating dev...
Dev F1 0.6804478897502153
Avf Dev F1 0.34290491002256074
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 2, loss: 0.692899682559073
Evaluating dev...
Dev F1 0.6747404844290658
Avf Dev F1 0.34001574486003555
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 3, loss: 0.6900365498537818
Evaluating dev...
Dev F1 0.6468781985670419
Avf Dev F1 0.5137443057467166
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 4, loss: 0.6852958591965338
Evaluating dev...
Dev F1 0.6125654450261779
Avf Dev F1 0.6140749303052968
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 5, loss: 0.6610435633144031
Evaluating dev...
Dev F1 0.598404255319149
Avf Dev F1 0.6061074985035643
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 6, loss: 0.6471000308326135
Evaluating dev...
Dev F1 0.6180904522613065
Avf Dev F1 0.6030831665100571
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 7, loss: 0.6324807313115647
Evaluating dev...
Dev F1 0.6339066339066339
Avf Dev F1 0.6100088725088726
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 8, loss: 0.6216871701957037
Evaluating dev...
Dev F1 0.6763717805151175
Avf Dev F1 0.6127568730968723
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 9, loss: 0.608595403454577
Evaluating dev...
Dev F1 0.6765039727582293
Avf Dev F1 0.6200284029181652
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 10, loss: 0.6015849360264838
Evaluating dev...
Dev F1 0.6727272727272728
Avf Dev F1 0.6161801501251043
  0%|          | 0/384 [00:00<?, ?it/s]
```

```
epoch 11, loss: 0.5910641442363461
Evaluating dev...
Dev F1 0.6786570743405275
Avf Dev F1 0.6478999657416924
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 12, loss: 0.5768006108701229
Evaluating dev...
Dev F1 0.6675031367628608
Avf Dev F1 0.6539686647179297
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 13, loss: 0.5628116114530712
Evaluating dev...
Dev F1 0.667515923566879
Avf Dev F1 0.6595256520371111
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 14, loss: 0.5524635118975615
Evaluating dev...
Dev F1 0.6888361045130642
Avf Dev F1 0.6551116938750292
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 15, loss: 0.5398838041583076
Evaluating dev...
Dev F1 0.6801007556675063
Avf Dev F1 0.6684287562121316
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 16, loss: 0.5290557606689011
Evaluating dev...
Dev F1 0.6871921182266009
Avf Dev F1 0.6676957821049903
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 17, loss: 0.5225802829954773
Evaluating dev...
Dev F1 0.6840148698884758
Avf Dev F1 0.6666291680941692
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 18, loss: 0.5134091453704362
Evaluating dev...
Dev F1 0.6578599735799208
Avf Dev F1 0.6622633201232937
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 19, loss: 0.5032367866563922
Evaluating dev...
Dev F1 0.664928292046936
Avf Dev F1 0.664928292046936
```

In [46]:

```python
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
LR=0.001
model = IronyDetector(
    input_dim=50,
    hidden_dim=30,
    embeddings_tensor=embeddings,
    pad_idx=-1,
    output_size=2,
)

optimizer = torch.optim.Adam(model.parameters(), LR)
trained_model = training_loop(
    num_epochs=20,
    train_features=encoded_train_data,
    train_labels=encoded_train_labels,
    dev_features=encoded_dev_data,
    dev_labels=encoded_dev_labels,
    optimizer=optimizer,
    model=model,
)
```

```
Training...
/var/folders/jd/yz20r3xj2x710fkjwd2jwntm0000gn/T/ipykernel_41791/4214819134.py:22: 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%|          | 0/384 [00:00<?, ?it/s]
epoch 0, loss: 0.6938428458136817
Evaluating dev...
Dev F1 0.6747404844290658
Avf Dev F1 0.34001574486003555
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 1, loss: 0.6925782101849715
Evaluating dev...
Dev F1 0.6509433962264152
Avf Dev F1 0.43517633946341855
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 2, loss: 0.6856470398294429
Evaluating dev...
Dev F1 0.6701791359325606
Avf Dev F1 0.5675682004449127
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 3, loss: 0.6560120962870618
Evaluating dev...
Dev F1 0.6287978863936592
Avf Dev F1 0.6335752623731488
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 4, loss: 0.6400787997990847
Evaluating dev...
Dev F1 0.6403061224489794
Avf Dev F1 0.6321530612244898
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 5, loss: 0.6206971986684948
Evaluating dev...
Dev F1 0.6792929292929293
Avf Dev F1 0.668487434996869
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 6, loss: 0.6030462741230925
Evaluating dev...
Dev F1 0.6543535620052771
Avf Dev F1 0.6583623480129479
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 7, loss: 0.5836598464520648
Evaluating dev...
Dev F1 0.6734434561626429
Avf Dev F1 0.6647003090719505
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 8, loss: 0.5705294870616248
Evaluating dev...
Dev F1 0.6476683937823835
Avf Dev F1 0.6453565066024778
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 9, loss: 0.554148993610094
Evaluating dev...
Dev F1 0.6623376623376623
Avf Dev F1 0.661011763106004
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 10, loss: 0.5365692067425698
Evaluating dev...
Dev F1 0.6732186732186732
Avf Dev F1 0.6518871143871143
```

```
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 11, loss: 0.5210452313379695
Evaluating dev...
Dev F1 0.6847058823529412
Avf Dev F1 0.6464465084279326
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 12, loss: 0.5001246128231287
Evaluating dev...
Dev F1 0.6544980443285527
Avf Dev F1 0.6544980443285527
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 13, loss: 0.48283204056012136
Evaluating dev...
Dev F1 0.6658259773013872
Avf Dev F1 0.6541005729961726
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 14, loss: 0.46917635495386395
Evaluating dev...
Dev F1 0.6028169014084508
Avf Dev F1 0.6302919458498565
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 15, loss: 0.4392756793143538
Evaluating dev...
Dev F1 0.5963431786216596
Avf Dev F1 0.6238094993958845
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 16, loss: 0.41534043656429276
Evaluating dev...
Dev F1 0.6214099216710183
Avf Dev F1 0.6219028775021758
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 17, loss: 0.42207200499251485
Evaluating dev...
Dev F1 0.6331125827814569
Avf Dev F1 0.6387642503124229
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 18, loss: 0.4080371934978757
Evaluating dev...
Dev F1 0.5933429811866858
Avf Dev F1 0.6300048239266762
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 19, loss: 0.3749270719029785
Evaluating dev...
Dev F1 0.6658905704307334
Avf Dev F1 0.620352692622774
```

In [49]:
```python
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
LR=0.001
model = IronyDetector(
    input_dim=50,
    hidden_dim=30,
    embeddings_tensor=embeddings,
    pad_idx=-1,
    output_size=2,
)

optimizer = torch.optim.Adam(model.parameters(), LR)
trained_model = training_loop(
    num_epochs=10,
    train_features=encoded_train_data,
    train_labels=encoded_train_labels,
    dev_features=encoded_dev_data,
    dev_labels=encoded_dev_labels,
    optimizer=optimizer,
    model=model,
)
```

```
Training...
/var/folders/jd/yz20r3xj2x710fkjwd2jwntm0000gn/T/ipykernel_41791/4214819134.py:22: 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%|          | 0/384 [00:00<?, ?it/s]
epoch 0, loss: 0.6940850719499091
Evaluating dev...
Dev F1 0.47092198581560285
Avf Dev F1 0.5104911497232417
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 1, loss: 0.6924297859271368
Evaluating dev...
Dev F1 0.4109149277688604
Avf Dev F1 0.5040304605913457
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 2, loss: 0.6791984674055129
Evaluating dev...
Dev F1 0.6432160804020101
Avf Dev F1 0.6291961160817638
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 3, loss: 0.6482229790029427
Evaluating dev...
Dev F1 0.6871310507674144
Avf Dev F1 0.6506979853545951
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 4, loss: 0.6281536080253621
Evaluating dev...
Dev F1 0.6842105263157895
Avf Dev F1 0.652993515306892
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 5, loss: 0.6117409110690156
Evaluating dev...
Dev F1 0.6916764361078546
Avf Dev F1 0.6527398333255865
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 6, loss: 0.5966584676643834
Evaluating dev...
Dev F1 0.6968641114982578
Avf Dev F1 0.6545241805633935
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 7, loss: 0.5797571902318547
Evaluating dev...
Dev F1 0.6925714285714285
Avf Dev F1 0.6421885974420116
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 8, loss: 0.5640131044977655
Evaluating dev...
Dev F1 0.6783625730994152
Avf Dev F1 0.6366776046645823
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 9, loss: 0.5370935885778939
Evaluating dev...
Dev F1 0.6736842105263158
Avf Dev F1 0.6313929152778854
```

In [51]:
```python
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
LR=0.001
```

```
model = IronyDetector(
    input_dim=50,
    hidden_dim=30,
    embeddings_tensor=embeddings,
    pad_idx=-1,
    output_size=2,
)

optimizer = torch.optim.Adam(model.parameters(), LR)
trained_model = training_loop(
    num_epochs=20,
    train_features=encoded_train_data,
    train_labels=encoded_train_labels,
    dev_features=encoded_dev_data,
    dev_labels=encoded_dev_labels,
    optimizer=optimizer,
    model=model,
)
```

```
Training...
/var/folders/jd/yz20r3xj2x710fkjwd2jwntm0000gn/T/ipykernel_41791/4214819134.py:22: 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%|          | 0/384 [00:00<?, ?it/s]
epoch 0, loss: 0.6937297157322367
Evaluating dev...
Dev F1 0.10859728506787329
Avf Dev F1 0.37389571213100625
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 1, loss: 0.6920515773817897
Evaluating dev...
Dev F1 0.1655773420479303
Avf Dev F1 0.4046491361402442
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 2, loss: 0.6856572475129118
Evaluating dev...
Dev F1 0.5955882352941176
Avf Dev F1 0.5679891037194822
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 3, loss: 0.6580821888831755
Evaluating dev...
Dev F1 0.5964912280701754
Avf Dev F1 0.6097210238711532
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 4, loss: 0.6346381864665697
Evaluating dev...
Dev F1 0.6043360433604336
Avf Dev F1 0.618750936253081
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 5, loss: 0.6179168701249486
Evaluating dev...
Dev F1 0.6233062330623306
Avf Dev F1 0.6370300009532758
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 6, loss: 0.6038264304709932
Evaluating dev...
Dev F1 0.6335570469798657
Avf Dev F1 0.643774721208564
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 7, loss: 0.5874790140272429
Evaluating dev...
Dev F1 0.6082036775106082
Avf Dev F1 0.6366290455267671
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 8, loss: 0.5650565198933085
Evaluating dev...
Dev F1 0.5909752547307133
Avf Dev F1 0.6296080523948726
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 9, loss: 0.5466500500915572
Evaluating dev...
Dev F1 0.5945165945165946
Avf Dev F1 0.6301952770442664
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 10, loss: 0.5280219468210513
Evaluating dev...
Dev F1 0.5852941176470589
Avf Dev F1 0.6275416724066676
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 11, loss: 0.49763943266589195
Evaluating dev...
Dev F1 0.5692541856925419
Avf Dev F1 0.623281596837149
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 12, loss: 0.4732037619687617
Evaluating dev...
Dev F1 0.5590062111801243
Avf Dev F1 0.6199525437923094
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 13, loss: 0.45647360630876693
Evaluating dev...
Dev F1 0.611032531824611
Avf Dev F1 0.639252662526574
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 14, loss: 0.43309890376015875
Evaluating dev...
Dev F1 0.5878962536023054
Avf Dev F1 0.6237100315630575
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 15, loss: 0.4273408875257398
Evaluating dev...
Dev F1 0.6321381142098274
Avf Dev F1 0.6387323093456307
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 16, loss: 0.39748152310494334
Evaluating dev...
Dev F1 0.6014388489208633
Avf Dev F1 0.6356419512780717
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 17, loss: 0.38849402737105265
Evaluating dev...
Dev F1 0.6140602582496413
Avf Dev F1 0.646337178109289
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 18, loss: 0.3776361171427804
Evaluating dev...
Dev F1 0.6524633821571237
Avf Dev F1 0.6595650244118952
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 19, loss: 0.3684615839156322
Evaluating dev...
Dev F1 0.6346153846153847
Avf Dev F1 0.6522952853598015
```

In [38]:
```
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
```

```
LR=0.001
model = IronyDetector(
    input_dim=50,
    hidden_dim=40,
    embeddings_tensor=embeddings,
    pad_idx=-1,
    output_size=2,
)

optimizer = torch.optim.Adam(model.parameters(), LR)
trained_model = training_loop(
    num_epochs=20,
    train_features=encoded_train_data,
    train_labels=encoded_train_labels,
    dev_features=encoded_dev_data,
    dev_labels=encoded_dev_labels,
    optimizer=optimizer,
    model=model,
)
```

```
Training...
/var/folders/jd/yz20r3xj2x710fkjwd2jwntm0000gn/T/ipykernel_41791/4214819134.py:22: 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%|          | 0/384 [00:00<?, ?it/s]
epoch 0, loss: 0.6945561696775258
Evaluating dev...
Dev F1 0.19789473684210526
Avf Dev F1 0.4190606828686447
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 1, loss: 0.6923871333710849
Evaluating dev...
Dev F1 0.2846441947565543
Avf Dev F1 0.4513220973782771
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 2, loss: 0.6772466158339133
Evaluating dev...
Dev F1 0.6877256317689532
Avf Dev F1 0.43775952949950003
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 3, loss: 0.6685426131201287
Evaluating dev...
Dev F1 0.6298076923076922
Avf Dev F1 0.5955306267806266
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 4, loss: 0.6445985662285239
Evaluating dev...
Dev F1 0.6681415929203539
Avf Dev F1 0.5959755583649389
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 5, loss: 0.6383125393961867
Evaluating dev...
Dev F1 0.6619385342789598
Avf Dev F1 0.6231204299301775
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 6, loss: 0.6282755623882016
Evaluating dev...
Dev F1 0.6626936829558999
Avf Dev F1 0.6277497191757917
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 7, loss: 0.6153369500146558
Evaluating dev...
Dev F1 0.6782407407407408
Avf Dev F1 0.6316576838032062
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 8, loss: 0.613708144131427
Evaluating dev...
Dev F1 0.6813441483198146
Avf Dev F1 0.635754041373022
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 9, loss: 0.5948142981699979
Evaluating dev...
Dev F1 0.6681922196796339
Avf Dev F1 0.6143991401428472
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 10, loss: 0.576184056738081
Evaluating dev...
Dev F1 0.6792009400705052
Avf Dev F1 0.6397468829195865
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 11, loss: 0.560420515636603
Evaluating dev...
Dev F1 0.6777251184834122
Avf Dev F1 0.6417611099663438
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 12, loss: 0.5401005334764098
Evaluating dev...
Dev F1 0.6782810685249709
Avf Dev F1 0.633345586268429
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 13, loss: 0.5269098900801813
Evaluating dev...
Dev F1 0.6748878923766816
Avf Dev F1 0.6115872483690261
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 14, loss: 0.5055468052159995
Evaluating dev...
Dev F1 0.6725082146768895
Avf Dev F1 0.5955133665977039
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 15, loss: 0.500869333879867
Evaluating dev...
Dev F1 0.6619385342789598
Avf Dev F1 0.6231204299301775
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 16, loss: 0.47087839581460383
Evaluating dev...
Dev F1 0.6427688504326328
Avf Dev F1 0.6220740803887302
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 17, loss: 0.45076603651978076
Evaluating dev...
Dev F1 0.6611957796014067
Avf Dev F1 0.6184099309167093
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 18, loss: 0.43130545209472376
Evaluating dev...
Dev F1 0.6600741656365884
Avf Dev F1 0.6403819104045011
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 19, loss: 0.40622873009609367
Evaluating dev...
Dev F1 0.6699266503667483
Avf Dev F1 0.6464158391498546
```

In [39]:
```
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
LR=0.0005
model = IronyDetector(
    input_dim=50,
    hidden_dim=30,
    embeddings_tensor=embeddings,
    pad_idx=-1,
    output_size=2,
)

optimizer = torch.optim.Adam(model.parameters(), LR)
trained_model = training_loop(
    num_epochs=40,
    train_features=encoded_train_data,
    train_labels=encoded_train_labels,
    dev_features=encoded_dev_data,
    dev_labels=encoded_dev_labels,
    optimizer=optimizer,
    model=model,
)
```

```
Training...
/var/folders/jd/yz20r3xj2x710fkjwd2jwntm0000gn/T/ipykernel_41791/4214819134.py:22: 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%|          | 0/384 [00:00<?, ?it/s]
epoch 0, loss: 0.6943441530068716
Evaluating dev...
Dev F1 0.6787252368647718
Avf Dev F1 0.3393626184323859
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 1, loss: 0.6936233635060489
Evaluating dev...
Dev F1 0.6754850088183422
Avf Dev F1 0.3777425044091711
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 2, loss: 0.6873549267183989
Evaluating dev...
Dev F1 0.6920222634508348
Avf Dev F1 0.48197604400611915
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 3, loss: 0.6564421626583984
Evaluating dev...
Dev F1 0.6908396946564886
Avf Dev F1 0.512086513994911
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 4, loss: 0.6424897058556477
Evaluating dev...
Dev F1 0.6748329621380846
Avf Dev F1 0.6078567326413693
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 5, loss: 0.6312897702058157
Evaluating dev...
Dev F1 0.6881959910913139
Avf Dev F1 0.6239722093821349
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 6, loss: 0.6228371031271914
Evaluating dev...
Dev F1 0.6803652968036531
Avf Dev F1 0.6274166909550181
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 7, loss: 0.6134684411032746
Evaluating dev...
Dev F1 0.657243816254417
Avf Dev F1 0.6162131490031209
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 8, loss: 0.6083549187363436
Evaluating dev...
Dev F1 0.6795454545454546
Avf Dev F1 0.6241763969974979
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 9, loss: 0.5956786267925054
Evaluating dev...
Dev F1 0.674364896073903
Avf Dev F1 0.6261046037255742
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 10, loss: 0.5865982608714452
Evaluating dev...
Dev F1 0.661137440758294
Avf Dev F1 0.6233223435675528
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 11, loss: 0.5723031014204025
Evaluating dev...
Dev F1 0.6674816625916871
Avf Dev F1 0.6437966972176312
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 12, loss: 0.5617928159190342
Evaluating dev...
Dev F1 0.6609124537607891
Avf Dev F1 0.6402764205180156
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 13, loss: 0.5511471744005879
Evaluating dev...
Dev F1 0.6552147239263804
Avf Dev F1 0.6321970698908675
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 14, loss: 0.5452267568325624
Evaluating dev...
Dev F1 0.664179104477612
Avf Dev F1 0.647158045389491
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 15, loss: 0.5340734493608276
Evaluating dev...
Dev F1 0.6626065773447016
Avf Dev F1 0.6370536393034869
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 16, loss: 0.5201232327381149
Evaluating dev...
Dev F1 0.645077720207254
Avf Dev F1 0.6427488338569078
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 17, loss: 0.5153530118792938
Evaluating dev...
Dev F1 0.6394904458598726
Avf Dev F1 0.6308266648524997
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 18, loss: 0.5036351312883198
Evaluating dev...
Dev F1 0.6420382165605095
Avf Dev F1 0.6334356636874644
  0%|          | 0/384 [00:00<?, ?it/s]
```

```
epoch 19, loss: 0.49824195582186803
Evaluating dev...
Dev F1 0.6440251572327045
Avf Dev F1 0.6305376124458515
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 20, loss: 0.4791678633579674
Evaluating dev...
Dev F1 0.6559006211180124
Avf Dev F1 0.6379640279801311
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 21, loss: 0.4706101923366077
Evaluating dev...
Dev F1 0.6400996264009963
Avf Dev F1 0.622375394595847
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 22, loss: 0.47425352776190266
Evaluating dev...
Dev F1 0.6617647058823529
Avf Dev F1 0.6386817958381124
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 23, loss: 0.46822046015101176
Evaluating dev...
Dev F1 0.6059782608695652
Avf Dev F1 0.6212848697831534
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 24, loss: 0.44750038771114004
Evaluating dev...
Dev F1 0.6408977556109725
Avf Dev F1 0.6237275663300765
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 25, loss: 0.44107726600486785
Evaluating dev...
Dev F1 0.6119791666666667
Avf Dev F1 0.61147261205396
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 26, loss: 0.4361387670893843
Evaluating dev...
Dev F1 0.6296774193548388
Avf Dev F1 0.6257741510476433
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 27, loss: 0.4225133314030245
Evaluating dev...
Dev F1 0.648854961832061
Avf Dev F1 0.6399355023064048
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 28, loss: 0.4125850993635443
Evaluating dev...
Dev F1 0.6217616580310881
Avf Dev F1 0.6192797791467777
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 29, loss: 0.41082997907263535
Evaluating dev...
Dev F1 0.6296774193548388
Avf Dev F1 0.6257741510476433
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 30, loss: 0.3860329113861856
Evaluating dev...
Dev F1 0.6545893719806763
Avf Dev F1 0.6247451109195167
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 31, loss: 0.3897501607813562
Evaluating dev...
Dev F1 0.6376440460947503
Avf Dev F1 0.630907016407269
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 32, loss: 0.3864018041349482
Evaluating dev...
Dev F1 0.640926640926641
Avf Dev F1 0.636183267623162
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 33, loss: 0.38398419456401217
Evaluating dev...
Dev F1 0.6233766233766234
Avf Dev F1 0.6218977357720814
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 34, loss: 0.38805157617510605
Evaluating dev...
Dev F1 0.579250720461095
Avf Dev F1 0.6158158364210238
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 35, loss: 0.3949122618456992
Evaluating dev...
Dev F1 0.6122448979591837
Avf Dev F1 0.6277745140609435
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 36, loss: 0.3508025266540547
Evaluating dev...
Dev F1 0.6082191780821918
Avf Dev F1 0.626248892523683
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 37, loss: 0.36728754169113625
Evaluating dev...
Dev F1 0.6054794520547945
Avf Dev F1 0.623635248415457
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 38, loss: 0.3283821191192449
Evaluating dev...
Dev F1 0.6038251366120219
Avf Dev F1 0.6211145633184798
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 39, loss: 0.33259853486864205
Evaluating dev...
Dev F1 0.6262886597938144
Avf Dev F1 0.6218514539074613
```

In [40]:

```python
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
LR=0.0002
model = IronyDetector(
    input_dim=50,
    hidden_dim=30,
    embeddings_tensor=embeddings,
    pad_idx=-1,
    output_size=2,
)

optimizer = torch.optim.Adam(model.parameters(), LR)
trained_model = training_loop(
    num_epochs=40,
    train_features=encoded_train_data,
    train_labels=encoded_train_labels,
    dev_features=encoded_dev_data,
    dev_labels=encoded_dev_labels,
    optimizer=optimizer,
    model=model,
)
```

```
Training...
/var/folders/jd/yz20r3xj2x710fkjwd2jwntm0000gn/T/ipykernel_41791/4214819134.py:22: 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%|          | 0/384 [00:00<?, ?it/s]
epoch 0, loss: 0.6936515443958342
Evaluating dev...
Dev F1 0.024449877750611245
Avf Dev F1 0.3348916055419723
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 1, loss: 0.6933069848455489
Evaluating dev...
Dev F1 0.1733615221987315
Avf Dev F1 0.40242062914837606
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 2, loss: 0.6928726594584683
Evaluating dev...
Dev F1 0.28037383177570097
Avf Dev F1 0.44749422319515775
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 3, loss: 0.6918516674389442
Evaluating dev...
Dev F1 0.39616613418530355
Avf Dev F1 0.489933287356969
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 4, loss: 0.6858313850437602
Evaluating dev...
Dev F1 0.5488505747126438
Avf Dev F1 0.5870744520341262
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 5, loss: 0.6551861487484226
Evaluating dev...
Dev F1 0.6236842105263157
Avf Dev F1 0.6270875832993336
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 6, loss: 0.6347719120482603
Evaluating dev...
Dev F1 0.6305170239596469
Avf Dev F1 0.6175527090108626
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 7, loss: 0.6245104381038497
Evaluating dev...
Dev F1 0.6501240694789082
Avf Dev F1 0.6313807160581355
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 8, loss: 0.61723533916908
Evaluating dev...
Dev F1 0.6641883519206939
Avf Dev F1 0.6457117825628229
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 9, loss: 0.6110102492384613
Evaluating dev...
Dev F1 0.6683291770573566
Avf Dev F1 0.6524705994576401
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 10, loss: 0.605168957884113
Evaluating dev...
Dev F1 0.6666666666666667
Avf Dev F1 0.6526268115942029
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 11, loss: 0.5994717145028213
Evaluating dev...
Dev F1 0.6666666666666667
Avf Dev F1 0.6512050932241928
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 12, loss: 0.5942669921399405
Evaluating dev...
Dev F1 0.6608260325406758
Avf Dev F1 0.6460592747737393
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 13, loss: 0.5882055079176401
Evaluating dev...
Dev F1 0.660759493670886
Avf Dev F1 0.6502722199537225
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 14, loss: 0.5825850896459693
Evaluating dev...
Dev F1 0.6675031367628608
Avf Dev F1 0.6539686647179297
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 15, loss: 0.5776580246165395
Evaluating dev...
Dev F1 0.6666666666666667
Avf Dev F1 0.6497716894977169
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 16, loss: 0.5708541844893867
Evaluating dev...
Dev F1 0.6674846625766871
Avf Dev F1 0.6452861421367441
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 17, loss: 0.5658879306089754
Evaluating dev...
Dev F1 0.6514575411913814
Avf Dev F1 0.641165012206429
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 18, loss: 0.5599011534359306
Evaluating dev...
Dev F1 0.6556962025316456
Avf Dev F1 0.6450524023410916
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 19, loss: 0.5567491130592922
Evaluating dev...
Dev F1 0.6448230668414154
Avf Dev F1 0.6466657487255066
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 20, loss: 0.5498994651328152
Evaluating dev...
Dev F1 0.636604774535809
Avf Dev F1 0.6426613616268788
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 21, loss: 0.5487481813955432
Evaluating dev...
Dev F1 0.6457516339869281
Avf Dev F1 0.6466729561352066
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 22, loss: 0.5434192798954124
Evaluating dev...
Dev F1 0.6284953395472702
Avf Dev F1 0.6360867502334052
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 23, loss: 0.534736147771279
Evaluating dev...
Dev F1 0.62787550744249
Avf Dev F1 0.6409817788784777
  0%|          | 0/384 [00:00<?, ?it/s]
```

```
epoch 24, loss: 0.5272699839745959
Evaluating dev...
Dev F1 0.6174863387978142
Avf Dev F1 0.6341795783764632
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 25, loss: 0.5160589215229265
Evaluating dev...
Dev F1 0.6137566137566137
Avf Dev F1 0.6192176384978441
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 26, loss: 0.5104261147983683
Evaluating dev...
Dev F1 0.6151797603195739
Avf Dev F1 0.6230432645786885
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 27, loss: 0.5041782893046426
Evaluating dev...
Dev F1 0.5991440798858773
Avf Dev F1 0.6309045729561439
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 28, loss: 0.5019977771444246
Evaluating dev...
Dev F1 0.6276595744680851
Avf Dev F1 0.6348016542417152
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 29, loss: 0.48662144139719504
Evaluating dev...
Dev F1 0.6302083333333334
Avf Dev F1 0.6297255765883377
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 30, loss: 0.4777635795374711
Evaluating dev...
Dev F1 0.5860597439544808
Avf Dev F1 0.6179396192696592
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 31, loss: 0.46976009851399186
Evaluating dev...
Dev F1 0.6025459688826026
Avf Dev F1 0.6313818115271537
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 32, loss: 0.4633721882322182
Evaluating dev...
Dev F1 0.6291834002677376
Avf Dev F1 0.6386069479102348
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 33, loss: 0.45079884070825454
Evaluating dev...
Dev F1 0.6275033377837116
Avf Dev F1 0.636044662522429
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 34, loss: 0.44379197229864076
Evaluating dev...
Dev F1 0.6235138705416117
Avf Dev F1 0.6283592518731224
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 35, loss: 0.47399495776820305
Evaluating dev...
Dev F1 0.6328125
Avf Dev F1 0.6323331429503916
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 36, loss: 0.43231003888649866
Evaluating dev...
Dev F1 0.6287978863936592
Avf Dev F1 0.6335752623731488
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 37, loss: 0.43624345415931504
Evaluating dev...
Dev F1 0.6459627329192545
Avf Dev F1 0.6275081154308207
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 38, loss: 0.4264866710388257
Evaluating dev...
Dev F1 0.6042553191489362
Avf Dev F1 0.6338526294176526
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 39, loss: 0.40785057270356145
Evaluating dev...
Dev F1 0.6251655629139073
Avf Dev F1 0.6309396492361579
```

In [41]:

```python
# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
LR=0.001
model = IronyDetector(
    input_dim=50,
    hidden_dim=20,
    embeddings_tensor=embeddings,
    pad_idx=-1,
    output_size=2,
)

optimizer = torch.optim.Adam(model.parameters(), LR)
trained_model = training_loop(
    num_epochs=20,
    train_features=encoded_train_data,
    train_labels=encoded_train_labels,
    dev_features=encoded_dev_data,
    dev_labels=encoded_dev_labels,
    optimizer=optimizer,
    model=model,
)
```

```
Training...
/var/folders/jd/yz20r3xj2x710fkjwd2jwntm0000gn/T/ipykernel_41791/4214819134.py:22: 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%|          | 0/384 [00:00<?, ?it/s]
epoch 0, loss: 0.6934188908586899
Evaluating dev...
Dev F1 0.014999999999999998
Avf Dev F1 0.333778659611993
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 1, loss: 0.6921619878460964
Evaluating dev...
Dev F1 0.005025125628140704
Avf Dev F1 0.32821678816618305
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 2, loss: 0.6807978416327387
Evaluating dev...
Dev F1 0.429042904290429
Avf Dev F1 0.528099038352111
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 3, loss: 0.6749804703673968
Evaluating dev...
Dev F1 0.6324786324786325
Avf Dev F1 0.6057498057498057
```

```
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 4, loss: 0.6522554437008997
Evaluating dev...
Dev F1 0.6572769953051644
Avf Dev F1 0.6145622513182714
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 5, loss: 0.6436936287985494
Evaluating dev...
Dev F1 0.6279949558638084
Avf Dev F1 0.6149421472976262
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 6, loss: 0.6345205695834011
Evaluating dev...
Dev F1 0.6367137355584083
Avf Dev F1 0.6309396492361579
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 7, loss: 0.6228063829864064
Evaluating dev...
Dev F1 0.6538461538461539
Avf Dev F1 0.6217948717948718
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 8, loss: 0.6104259507264942
Evaluating dev...
Dev F1 0.6713286713286712
Avf Dev F1 0.6270844540075309
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 9, loss: 0.5969245807112505
Evaluating dev...
Dev F1 0.6744457409568261
Avf Dev F1 0.631166740493184
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 10, loss: 0.583554625705195
Evaluating dev...
Dev F1 0.6721120186697782
Avf Dev F1 0.6285227744752141
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 11, loss: 0.5715893000209084
Evaluating dev...
Dev F1 0.6560975609756098
Avf Dev F1 0.6305697888911662
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 12, loss: 0.5689084151526913
Evaluating dev...
Dev F1 0.674364896073903
Avf Dev F1 0.6261046037255742
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 13, loss: 0.5496181167351702
Evaluating dev...
Dev F1 0.6792873051224944
Avf Dev F1 0.6132285582216246
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 14, loss: 0.5372619533445686
Evaluating dev...
Dev F1 0.6726256983240223
Avf Dev F1 0.6070483734186621
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 15, loss: 0.5418133272711808
Evaluating dev...
Dev F1 0.667433831990794
Avf Dev F1 0.6164236829126902
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 16, loss: 0.5240917284972966
Evaluating dev...
Dev F1 0.6002691790040376
Avf Dev F1 0.6123975477826762
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 17, loss: 0.5077922656588877
Evaluating dev...
Dev F1 0.6651162790697674
Avf Dev F1 0.6189082879028363
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 18, loss: 0.4920735693109843
Evaluating dev...
Dev F1 0.6682297772567408
Avf Dev F1 0.6263322160879886
  0%|          | 0/384 [00:00<?, ?it/s]
epoch 19, loss: 0.49438708514207974
Evaluating dev...
Dev F1 0.6450809464508095
Avf Dev F1 0.6276020327329287
```

In [30]:
```
# This is so that you don't have to restart the kernel everytime you edit hmm.py

%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [52]:
```
preds = predict(model, encoded_test_data)
# print(preds)
# print(encoded_test_labels)
train_f1 = f1_score(preds, encoded_test_labels, 1)
print(f"Dev F1 {train_f1}")
train_f1 = avg_f1_score(preds, encoded_test_labels, [0, 1])
print(f"Avf Dev F1 {train_f1}")
```

```
Dev F1 0.5790297339593115
Avf Dev F1 0.6447355343639399
```

Written Assignment (30 Points)

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

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.

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

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

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

1.

Here the task is irony detection in text, this is a dificult task for models due to the model needing to be able to understand the hidden subtext in the text. This is very necesary for a model due to the prevalence of irony and sarcasm in human conversations. Recognition of irony would help models come closer to understanding human speech in a more holistic way.

2.

Pretrained word embeddings are vector representations of words in a continuous vector space. They capturing semantic relationships between words based on their contextual usage in a large corpus of text. Two of the most widely used methods for generating pretrained word embeddings are Word2Vec and GloVe (Global Vectors for Word Representation). I will explain how Word2Vec works further:

Word2Vec: Word2Vec is a designed to learn word embeddings by predicting the context of words in a given corpus. Given a target word, the model predicts the context words that are likely to appear around it. The objective is to maximize the probability of context words given the target word. This way we arbe able to understand how the word relates to other words by looking at its neighbours.

3.

Using word embeddings offers many advantages over bag-of-words representations. While BoW is a sparse method that does not take context into account, word embeddings provides us dense vector representations that capture semantic relationships and contextual information. They enable models to understand similarities between words, consider nuanced meanings based on context, and exhibit algebraic properties for word analogies. OVerall they offer more understanding into the word as compared to BoW and also allows for better models.

4.

The major difference between Binary Cross Entropy loss and Negative Log Likelihood is when they are used. While we use BCE loss in the case of binary classification, NLL loss is used for multi-class scenarios, especially in the context of sequence modeling.

5.

The table for all the models trained is bellow:

| Sno | Hidden Layer Size | Learning Rate | F1: Epoch 5 | F1: Epoch 10 | F1: Epoch 15 | F1: Epoch 20 | F1: Epoch 25 | F1: Epoch 30 | F1: Epoch 35 | F1: Epoch 40 | | :--- | :----: |:---: |:---: |:---: |:---: |:---: |:---: |:---: |:---: |:---: |:---: | | 1 | 10 | 0.001 | 0.60 | 0.62 | 0.66 | 0.66| | 2 | 20 | 0.001 | 0.61 | 0.63 | 0.62 | 0.63| | 3 | 30 | 0.001 | 0.62 | 0.62 | 0.64 | 0.65| | 4 | 30 | 0.0005 | 0.60 | 0.62 | 0.64 | 0.64| 0.62| 0.63 | 0.63|0.62| | 5 | 30 | 0.0002 | 0.58 | 0.65 | 0.65 | 0.64| 0.64| 0.64 | 0.63|0.63| | 6 | 40 | 0.001 | 0.60 | 0.64 | 0.62 | 0.65|

As we can see the model works best for a Hidden layer length of 30 and Learning rate of 0.001. This gave us an average f1 score of 0.644 on the test set. Overall a hidden layer lenght of 30 provides the highest f1 score while the optimal learning rate is 0.001 for this case.

This beats the naive Bayes Classifier in average F1 score, while the indivisual f1 score also reached near 0.69 in multipe runs.