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.

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

%load_ext autoreload
%autoreload 2
```

Data

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

```
from irony import load_datasets
from sklearn.model_selection import train_test_split

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

text_train, text_dev, label_train, label_dev =
train_test_split(train_sentences, train_labels, test_size = 0.2,
stratify = train_labels)

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

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

```
from irony import run_nb_baseline

run_nb_baseline()

Vectorizing Text: 100%|
██████████████████████████████████████████████████████████| 3834/3834
[00:00<00:00, 9574.52it/s]
Vectorizing Text: 100%|
██████████████████████████████████████████████████████████| 3834/3834
[00:00<00:00, 11618.50it/s]
Vectorizing Text: 100%|
████████████████████████████████████████████████████████████| 784/784
[00:00<00:00, 15521.70it/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.

```python
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
        # raise NotImplementedError
        tokenized_sentences = []

        for sentence in sentences:
            tokens = ["<SOS>"] + [token.text for token in
self.nlp(sentence)] + ["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
        # raise NotImplementedError
        max_len = max(len(sentence) for sentence in batch)
        padded_batch = [sentence + [self.pad_symbol] * (max_len - len(sentence)) for sentence in batch]

        return padded_batch

# create the vocabulary of the dataset: use both training and test sets here

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

all_data = train_sentences + test_sentences
my_tokenizer = Tokenizer()

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

with open('vocab.txt', 'w', encoding = 'utf-8') 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.

```
# Dowload the gloVe vectors for Twitter tweets. This will download a file called glove.twitter.27B.zip
# ! python -m wget https://nlp.stanford.edu/data/glove.twitter.27B.zip

# unzip glove.twitter.27B.zip
# if there is an error, please download the zip file again
# Manually unzipped based on piazza post
# ! unzip glove.twitter.27B.zip

# Let's see what files are there:
# 4 different files
# ! ls . | grep "glove.*.txt"

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

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

Creating a custom Embedding Layer

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

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

```
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", encoding='utf8') 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
            # raise NotImplementedError
            if word in vocab:
                word2i[word] = len(word2i)
                vector = torch.FloatTensor([float(weight) for weight
in weights])
                vectors.append(vector)

    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

```
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)

```
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
    # raise NotImplementedError
    std_dev = dim ** -0.5
    return torch.randn(num_embeddings, dim) / std_dev


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
    # raise NotImplementedError
    word2i = glove_word2i.copy()
    embeddings = glove_embeddings.clone()
    for oov in oovs:
        if oov not in word2i:
            word2i[oov] = len(word2i)

    new_rows = intialize_new_embedding_weights(len(oovs), 
embeddings.size(1))
    embeddings = torch.cat([embeddings, new_rows], dim=0)
    return word2i, embeddings

def make_batches(sequences: List[str], batch_size: int) -> 
List[List[str]]:
    """Yield batch_size chunks from sequences."""
    # TODO
    # raise NotImplementedError
    for i in range(0, len(sequences), batch_size):
        yield sequences[i:i+ batch_size]


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

# We make batches now and use those.
batch_tokenized = []
# 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))


glove_word2i, glove_embeddings = read_pretrained_embeddings(
    embeddings_path,
    vocab_path
)

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

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

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

Encoding words to integers: DO NOT EDIT

```
# 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)

```
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#torc
h.nn.Embedding.from_pretrained
        self.embeddings =
torch.nn.Embedding.from_pretrained(embeddings_tensor)#, freeze=False,
padding_idx=self.pad_idx)
        # 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
        """

        if(symbols.dim() == 1):
            symbols = symbols.unsqueeze(0)

        # 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")
        lens = torch.maximum(lens, torch.tensor(1, dtype=lens.dtype))
        packed = torch.nn.utils.rnn.pack_padded_sequence(
            embedded, lens, batch_first=True, enforce_sorted=False
        )
        # -> batch_size x seq_len x encoder_dim, (h0, c0).
        packed_outs, (H, C) = self.lstm(packed)
        encoded, _ = torch.nn.utils.rnn.pad_packed_sequence(
            packed_outs,
            batch_first=True,
            padding_value=self.pad_idx,
            total_length=None,
        )
        # Now we have the representation of each token encoded by the
LSTM.
        encoded, (H, C) = self.lstm(embedded)

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

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

Evaluation

```
def predict(model: torch.nn.Module, dev_sequences:
List[torch.Tensor]):
    preds = []
    # TODO: Get the predictions for the dev_sequences using the model
    # raise NotImplementedError
    preds = []

    model.eval()

    with torch.no_grad():
        for symbols in dev_sequences:
            output = model(symbols)

            _, predicted_labels = torch.max(output, 2)
            preds.append(predicted_labels.tolist())

    return preds
```

Training

```
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()
            preds = model(features).squeeze(1)
            labels = labels.unsqueeze(0)
            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)

        # due to my previous code, the dimension is not completely
correct, flat it out
        flat_preds = []
        for element in preds:
            flat_preds.append(element[0][0])
        dev_f1 = f1_score(flat_preds, dev_labels, label2i['1'])
        dev_avg_f1 = avg_f1_score(flat_preds, dev_labels,
list(label2i.keys()))
        print(f"Dev F1 {dev_f1}")
        print(f"Avf Dev F1 {dev_avg_f1}")

    # Return the trained model
    return model

# TODO: Load the model and run the training loop
#       on your train/dev splits. Set and tweak hyperparameters.
epochs = 20

input_dim = 50
hidden_dim = 256
output_size = 2
dropout_val = 0.3
LR = 0.005

tokenizer = Tokenizer()

# Tokenize and encode training data
text_train_tokenized = tokenizer(text_train)
text_train_encoded = encode_sentences(text_train_tokenized, word2i)
label_train_encoded = encode_labels(label_train)

# Tokenize and encode dev data
text_dev_tokenized = tokenizer(text_dev)
text_dev_encoded = encode_sentences(text_dev_tokenized, word2i)
label_dev_encoded = encode_labels(label_dev)

model = IronyDetector(
    input_dim=input_dim,
    hidden_dim=hidden_dim,
    embeddings_tensor=embeddings,
    pad_idx=word2i["<PAD>"],
    output_size=output_size,
)


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


# training_loop(
#     epochs,
#     text_train_encoded,
#     label_train_encoded,
#     text_dev_encoded,
#     label_dev_encoded,
#     optimizer,
#     model,
# )

# Tokenize and encode test data
text_test_tokenized = tokenizer(test_sentences)
text_test_encoded = encode_sentences(text_test_tokenized, word2i)
label_test_encoded = encode_labels(test_labels)

training_loop(
    epochs,
    text_train_encoded,
    label_train_encoded,
    text_test_encoded,
    label_test_encoded,
    optimizer,
    model,
)
```

```
Training...

C:\Users\linsh\AppData\Local\Temp\ipykernel_130784\587731332.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):
```

{"model_id":"a7f28bc304f44e06bdbf5b4bbece8dd5","version_major":2,"version_minor":0}

```
epoch 0, loss: 0.7454035960436645
Evaluating dev...
Dev F1 0.5680365296803653
Avf Dev F1 0.28401826484018267
```

{"model_id":"0fb571e1f6064eb8b6f304e14d8ccc66","version_major":2,"version_minor":0}

```
epoch 1, loss: 0.7520697910786841
Evaluating dev...
Dev F1 0.5685557586837293
Avf Dev F1 0.28638758398321484
```

{"model_id":"33d1c99ef8114146ae7015495ab3837c","version_major":2,"version_minor":0}

```
epoch 2, loss: 0.7851492331642931
Evaluating dev...
Dev F1 0.5680365296803653
Avf Dev F1 0.28401826484018267
```

{"model_id":"502f12b6585144b5812784d363688515","version_major":2,"version_minor":0}

```
epoch 3, loss: 0.7776615808365358
Evaluating dev...
Dev F1 0.5680365296803653
Avf Dev F1 0.28401826484018267
```

{"model_id":"9e9b202f7dbb4bf89c38e010bd695682","version_major":2,"version_minor":0}

```
epoch 4, loss: 0.7767891655534865
Evaluating dev...
Dev F1 0.5680365296803653
Avf Dev F1 0.28401826484018267
```

{"model_id":"2ee8582007e240bb867449c52221f2d6","version_major":2,"version_minor":0}

```
epoch 5, loss: 0.7769248262667232
Evaluating dev...
Dev F1 0.5722171113155474
Avf Dev F1 0.30274057228979034
```

{"model_id":"d72cbd22a20f4e2992b64f4c913c2704","version_major":2,"version_minor":0}

```
epoch 6, loss: 0.7573146273825047
Evaluating dev...
Dev F1 0.601010101010101
Avf Dev F1 0.59689680308237
```

{"model_id":"1b29e85dbc154e09ba8c12f585dc8391","version_major":2,"version_minor":0}

```
epoch 7, loss: 0.7408549033542187
Evaluating dev...
Dev F1 0.5889101338432123
Avf Dev F1 0.3825776722855908
```

```
{"model_id":"b8d8196d1d734b1aafd5b21de9a32aa4","version_major":2,"vers
ion_minor":0}

epoch 8, loss: 0.7272581803157834
Evaluating dev...
Dev F1 0.40704500978473585
Avf Dev F1 0.560192325138347

{"model_id":"7aba7b02cd334972b30db9410aa44ecf","version_major":2,"vers
ion_minor":0}

epoch 9, loss: 0.746205808693512
Evaluating dev...
Dev F1 0.5473411154345006
Avf Dev F1 0.5547245100384548

{"model_id":"fd9f585e065f4f91b989c2eabdb88ace","version_major":2,"vers
ion_minor":0}

epoch 10, loss: 0.722943169026331
Evaluating dev...
Dev F1 0.560398505603985
Avf Dev F1 0.5494802985536265

{"model_id":"f4243756272341078 29f157a512fcde0","version_major":2,"vers
ion_minor":0}

epoch 11, loss: 0.7387737606779694
Evaluating dev...
Dev F1 0.4258289703315881
Avf Dev F1 0.547587851999965

{"model_id":"94d184d799eb473883041ab90ec00aef","version_major":2,"vers
ion_minor":0}

epoch 12, loss: 0.7390346753638056
Evaluating dev...
Dev F1 0.5495118549511855
Avf Dev F1 0.5849791942205986

{"model_id":"7e6946c905ad4cc8a55f6856367e1798","version_major":2,"vers
ion_minor":0}

epoch 13, loss: 0.7287901155534466
Evaluating dev...
Dev F1 0.40661157024793393
Avf Dev F1 0.5169091080730843

{"model_id":"1f76fda15ff946e7b2d318b22652f570","version_major":2,"vers
ion_minor":0}
```

```
epoch 14, loss: 0.7305173016754369
Evaluating dev...
Dev F1 0.38966202783300197
Avf Dev F1 0.55069955851744
```

{"model_id":"5c6b66e534004b9e8362a64874526a3a","version_major":2,"version_minor":0}

```
epoch 15, loss: 0.7227549278122662
Evaluating dev...
Dev F1 0.10899182561307902
Avf Dev F1 0.41835935993393325
```

{"model_id":"5c816cf831944064b6ec26636e3f4b60","version_major":2,"version_minor":0}

```
epoch 16, loss: 0.7385619107997006
Evaluating dev...
Dev F1 0.45407279029462744
Avf Dev F1 0.5681060217870716
```

{"model_id":"ebfcec410f614f389cc826c332c8fcd6","version_major":2,"version_minor":0}

```
epoch 17, loss: 0.7377002371262084
Evaluating dev...
Dev F1 0.49087893864013266
Avf Dev F1 0.586372111807113
```

{"model_id":"30c12d784d8e4ae587daab2b8edeb17f","version_major":2,"version_minor":0}

```
epoch 18, loss: 0.7097178135528179
Evaluating dev...
Dev F1 0.3230088495575221
Avf Dev F1 0.5244076505852127
```

{"model_id":"709971522b604878be04c027c79e4aee","version_major":2,"version_minor":0}

```
epoch 19, loss: 0.7486619811868256
Evaluating dev...
Dev F1 0.48682170542635655
Avf Dev F1 0.5641042438291046
```

```
IronyDetector(
  (embeddings): Embedding(17302, 50)
  (dropout_layer): Dropout(p=0.3, inplace=False)
  (lstm): LSTM(50, 256, num_layers=2, batch_first=True, dropout=0.3,
bidirectional=True)
  (classifier): Linear(in_features=512, out_features=2, bias=True)
  (log_softmax): LogSoftmax(dim=2)
)
```

Written Assignment (30 Points)

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

The task involves irony detection in tweets. Given a tweet, the goal is to classify whether it is ironic or not. This is a binary classification problem where the model needds to distinguish between ironic and non-ironic statements. Irony detection is valuable in various NLP tasks such as sentiment analysis, which could be used to improve user understanding in different settings such as social media analysis.

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.

Pretrained word embeddings are computed using algorithms like Word2Vec, GloVe, or similar methods. These embeddings capture the semantic relationships between words based on the different senses. For example, Word2Vec is based on the idea that words appearing in similar contexts have similar meanings. Therefore, using the skip-gram algorithm puts words that appear together as positive examples. Similarly, GloVe aims to capture word relationships based on co-occurrence probabilities.

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

Word embeddings capture the semantic relationships between words, while bag of words usually lacks the semantic understanding. Additionally, word embedding can represent words in continuous vector space, which may reduce the high dimensionality of one-hot encoded bag-of-words. Lastly, the embeddings will consider the context of words, preserving information about how words are related to each other in the context.

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

Binary Cross Entropy Loss, like the name suggest are commonly used for binary classification programs, it measures the difference between predicted probabilities and the actual labels. On the other hand, NNL Loss is used when the model outputs log probabilities. It is more suitable for multiclass classification. In this context, it is applied to sequence to sequecne model where each word is treated as a class.

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

Epochs	LR	Hidden Size	Loss	F1 Score	Average F1 Score
20	0.01	64	0.7041157071	0.671065033	0.4666107406
20	0.005	64	0.5170861719	0.6364846871	0.6439128416
20	0.001	64	0.1032263312	0.6456692913	0.6479641793
20	0.001	128	0.07466707891	0.6438896189	0.6466537357
20	0.005	128	0.5772615606	0.6833855799	0.5791277986
20	0.005	256	0.7922084286	0.6749335695	0.384380365
20	0.001	256	0.07517395022	0.6641975309	0.6442534616
40	0.001	128	0.03310638676	0.6837387964	0.677858774
40	0.001	256	0.05707819213	0.6421319797	0.632057947
20	0.001	512	0.09581331675	0.595862068965517	0.6168432718

First, I started by randomly assigning values for each of the hyperparameters. Then I started fine-tuning them based on the training and dev data. The above are some of the sample runs during the experiment.

Learning Rate (LR)

I tested various learning rates and their effects on the F1 and average F1 scores. Surprisingly, the F1 score has been consistent, around 0.6 to 0.7; however, there is a variation in the Average F1 score; therefore, I decided that the learning rate will be around 0.001 to 0.005. Given enough time, I would perform a gid search from 0.001 to 0.005 to identify the most optimal learning rate.

Hidden Size

I tested different hidden sizes. Surprisingly, it did not heavily affect the F1 or the average F1 score. One of the test cases for 256 hidden size had the worst average F1 score. However, rerunning the test yields a different result. Therefore, at least in this task, hidden size does not contribute heavily to the F1 scores.

Epochs

The further examine why the F1 scores are similar, I started increasing the epochs to see how the scores are changing over time. It was clear how the score would suddenly decrease despite the loss still decreasing. Some of the potential explanations could be overfitting or being present in local optimal.

Other hyperparameters

I used the default value in the other hyperparameters as shown in other code portions. For example, the output_size = 2 and dropout_val = 0.3.

Data Split

Similar to past homework, I used the sklearn built-in, train_test_split(), to split the data into 80-20 splits.

Other Changes

I had to make various code changes to make the program executable and produce results. However, there should be no architectural change to the code itself.

Final Discussion

In the end, without any training on the test data, we obtained 0.48682170542635655 F1 score and 0.5641042438291046 average F1 score. This could indicate that the model is overfitting on the training data. This means we will likely need to do another random start to identify some other starting point for the hyperparameters. However, even though this model trains faster than part A, it still takes considerable time, making additional test work for the future.