VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

NATURAL LANGUAGE PROCESSING (CO3085)

Assignment

Sentiment Classification

Advisor: Quản Thành Thơ
Students: Đậu Gia Kiên - 1952799

HO CHI MINH CITY, NOVEMBER 2023

Contents

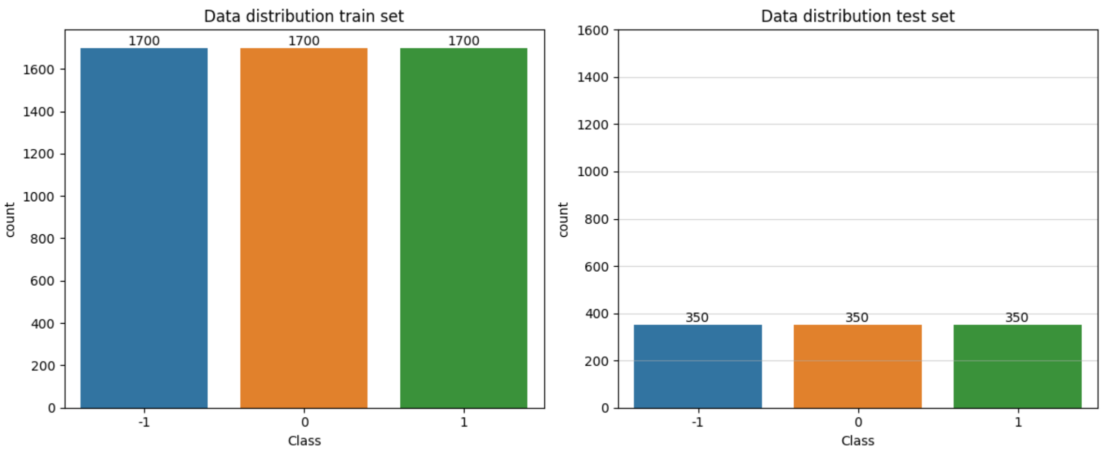
1 Introduction

In this assignment, I have conduct many experiment to come up with the best result for the given dataset using CNN, LSTM and CNN + LSTM. I use mainly pytorch library for this assigment

2 Data prepocressing

2.1 Data cleaning

In this sentiment classification task, the dataset consist of reviews about apple watch, and there are 3 class of data (negative, neutral, positive). Training data & testing data have 5100 and 1050 records respectively and is evenly distributed into 3 classes. The first step is clean the data. Data



Hình 1: Data distribution of train and test set

is very inconsistent accross 2 sets. My strategy of cleaning text is remove all links, numbers and common special characters and punctuation as well as lower case all text. It's a regret that i can't lemmatize all words into the original form due to lack of instant available resources that i can find, so i skipped this step.

Without lemmatization, the removal of stopwords isn't viable. In Vietnamese a single word may have a lot of meaning. However due to the inconsistency of the dataset (for example acronyms like 'a', 'dt', 'dc', 'đc', 'k', 'không', 'e') removing stopwords defined correctly in a dictionary may not remove all apperance of stopwords in the dataset, which actually lead to worse result in my experiment.

However, i decide to do some manual lemmatization, bringing acronyms and teencodes to its original form myself using excel. I also experiment with Circumflex Accent Marks removal however the result was worse due to absence of Vietnamese word features (mất or mắt is mat after Circumflex Accent Mark removed)

2.2 Vectorzing words

For vectorizing, I've used pyvi and torchtext library

```
from torchtext.vocab import build_vocab_from_iterator
import torchtext.transforms as T

def tokenize(data):
    #function to yield each token for the vocab input
    for x in data:
        yield ViTokenizer.tokenize(x.lower()).split()

vocab = build_vocab_from_iterator(
    tokenize(pd.concat([df_train['Data'], df_test['Data']])),
    min_freq=1,
    specials= ['<pad>', '<sos>', '<eos>', '<unk>'],
    special_first=True
)
#refer to
#https://pytorch.org/text/stable/vocab.html#build-vocab-from-iterator
#for more info
vocab.set_default_index(vocab['<unk>'])
```

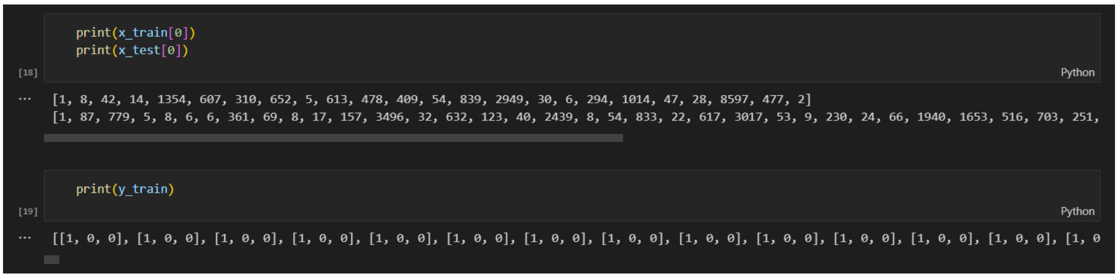
Then, I proceed to do a transform on both train and test set to convert all words available into the input vector

```
def getTransform(vocab):
    """
    Create transforms based on given vocabulary. The returned transform is applied to
    sequence
    of tokens.
    """
    text_tranform = T.Sequential(
        ## converts the sentences to indices based on given vocabulary
        T.VocabTransform(vocab=vocab),
        ## Add <sos> at beginning of each sentence. 1 because the index for <sos> in
    vocabulary is
        # 1 as seen in previous section
        T.AddToken(1, begin=True),
        ## Add <eos> at beginning of each sentence. 2 because the index for <eos> in
    vocabulary is
        # 2 as seen in previous section
        T.AddToken(2, begin=False)
    )
    return text_tranform
def applyTransform(text):
    """
    Apply transforms to sequence of tokens in a sequence pair
    """
    return getTransform(vocab)(ViTokenizer.tokenize(text.lower()).split())

x_train = df_train['Data'].apply(applyTransform)
x_test = df_test['Data'].apply(applyTransform)
```

After that, i onehot encode the labels into arrays size of 3.

```
y_train = []
y_test = []
label_train = df_train['Class'].values
```

```
label_test = df_test['Class'].values
for x in label_train:
    if x == -1:
        y_train.append([1,0,0])
    elif x == 0:
        y_train.append([0,1,0])
    elif x == 1:
        y_train.append([0,0,1])

for y in label_test:
    if y == -1:
        y_test.append([1,0,0])
    elif y == 0:
        y_test.append([0,1,0])
    elif y == 1:
        y_test.append([0,0,1])
```



Hình 2: Vectorize input and Labels

2.3 Data preparation

In order to train data with pytorch `nn.Module` (generic class for all models, I need to define a `Dataset` and `DataLoader` object as well as collate function to batchify data.

```
class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        """This function is called everytime we iterate through the dataloader"""
        return self.data.iloc[idx], self.labels[idx]

def custom_collate_fn(batch):
    #function take in a Dataset object
    data, labels = zip(*batch)

    # Sort sequences by length (from longest to shortest)
```

```
    sorted_data, sorted_labels = zip(*sorted(zip(data, labels), key=lambda x: len(x[0]),
     reverse=True))
    # Pad sequences to the length of the longest sequence
    padded_data = pad_sequence([torch.tensor(seq) for seq in sorted_data], batch_first=
    True)
    padded_labels = torch.tensor(sorted_labels).float()

    # Create a mask for the padded elements
    mask = (padded_data != 0).float()

    return padded_data.to(device), mask, padded_labels.to(device)

torch.manual_seed(12)
trainset = CustomDataset(x_train, y_train)
testset = CustomDataset(x_test, y_test)

# Create a DataLoader for your dataset
train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=True, collate_fn=
    custom_collate_fn)
test_loader = DataLoader(testset, batch_size=batch_size, shuffle=True, collate_fn=
    custom_collate_fn)
```

Listing 1: Create Dataset and DataLoader and collatefn

3 Training & Testing

After many testing and optimization, I've come up with the input which provide the best result. Batch size of 128 the decent mark for my GPU (NVIDIA Quadro M1000M/Release 2014). I've include both training and testing into my loop, so the program will evaluate after each epoch. For more details please check the files.

```
# Define the dimensions
input_dim = vocabsize # The dimension of your input data (e.g., vocabulary size)
hidden_dim = 512 # Size of the hidden layer
embedding_dim =  300
output_dim = 3  # Two classes: positive and negative
learning_rate = 0.001
epochs = 100
batch_size = 128
```

Listing 2: Model inputs

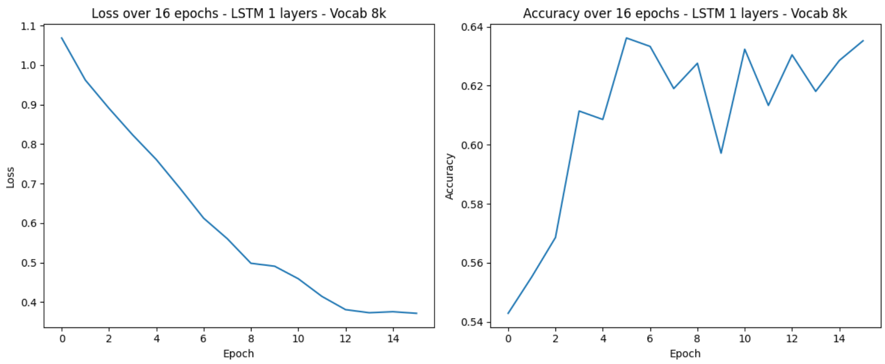
3.1 LSTM

The model architecture i've tried are:

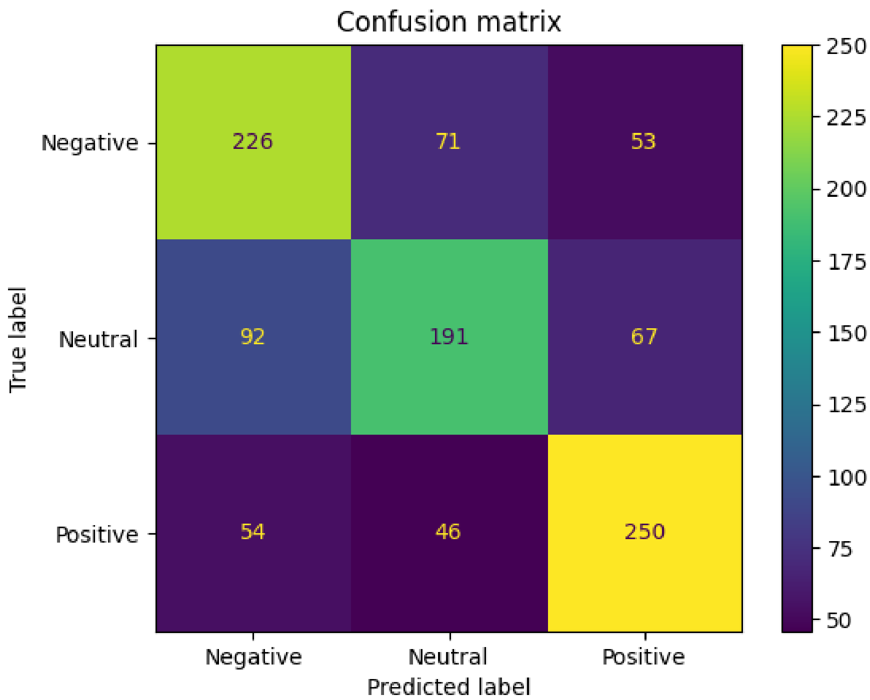
- embedding layer + 1 LSTM + 1 FC + 1 Dropout (best result)
- embedding layer + 1 LSTM + 3 FC + 1 Dropout (2nd best)
- embedding layer + 1 LSTM + 1 FC + 1 Dropout + Logsoftmax
- embedding layer + 3 LSTM + 1 FC + 1 Dropout + Logsoftmax

- embedding layer + 3 LSTM + 1 Dropout
- embedding layer + 3 LSTM + 1 FC (worst)

Note that Logsoftmax is basically result of logging after taking softmax at the output. Logsoftmax results can be passed into `nn.CrossEntropyLoss` module and can be revert later by doing e^{output} to compare with the labels



Hình 3: 1 LSTM

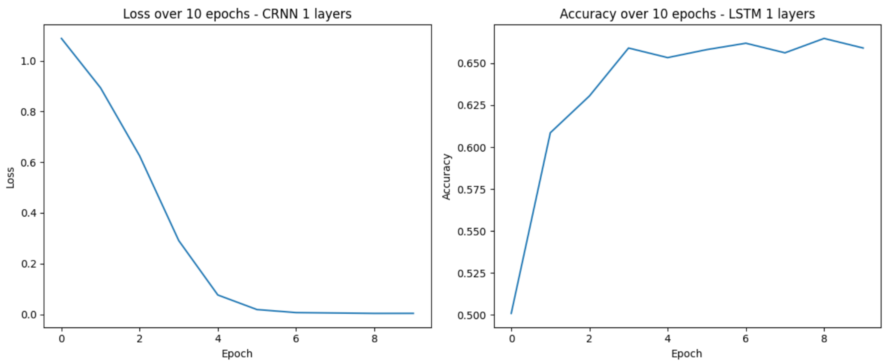


Hình 4: Confusion matrix for RNN

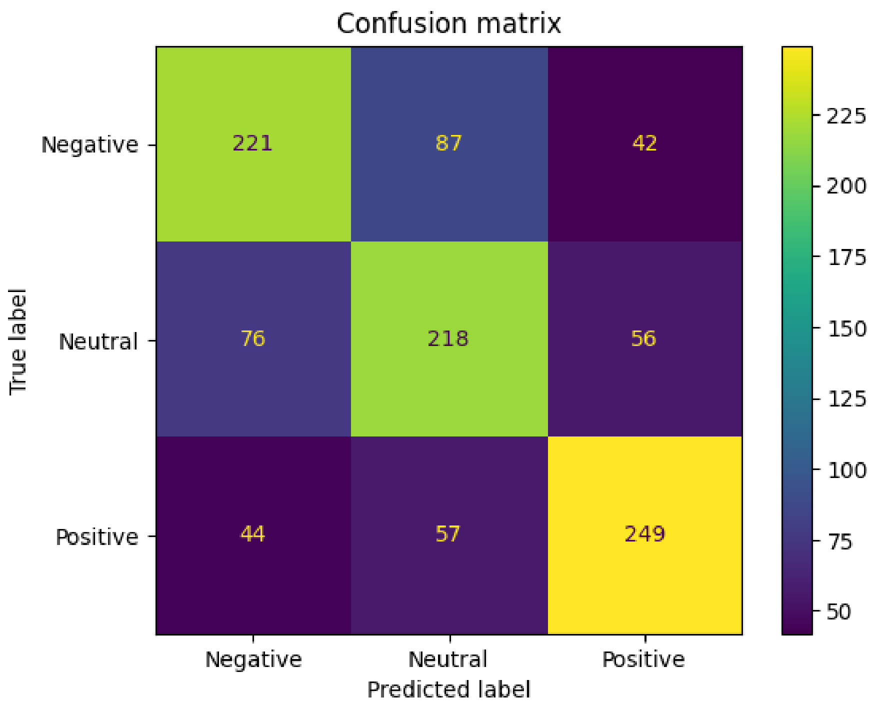
3.2 CNN + LSTM

The follow model architecture i've tried are:

- embedding layer + 1 CNN + 1 LSTM + 1 FC + 1 Dropout (best result)
- embedding layer + 3 CNN + 1 LSTM + 1 FC + 1 Dropout (2nd best)
- embedding layer + 3 CNN + 3 LSTM + 1 FC + 1 Dropout (worst)



Hình 5: 1 CNN + 1 LSTM

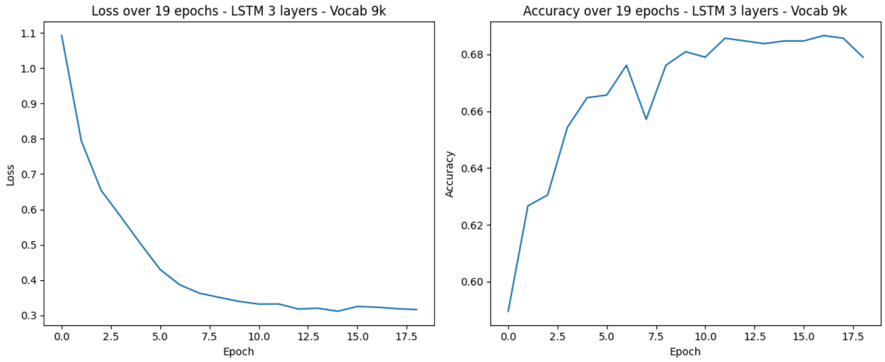


Hình 6: Confusion matrix for CRNN

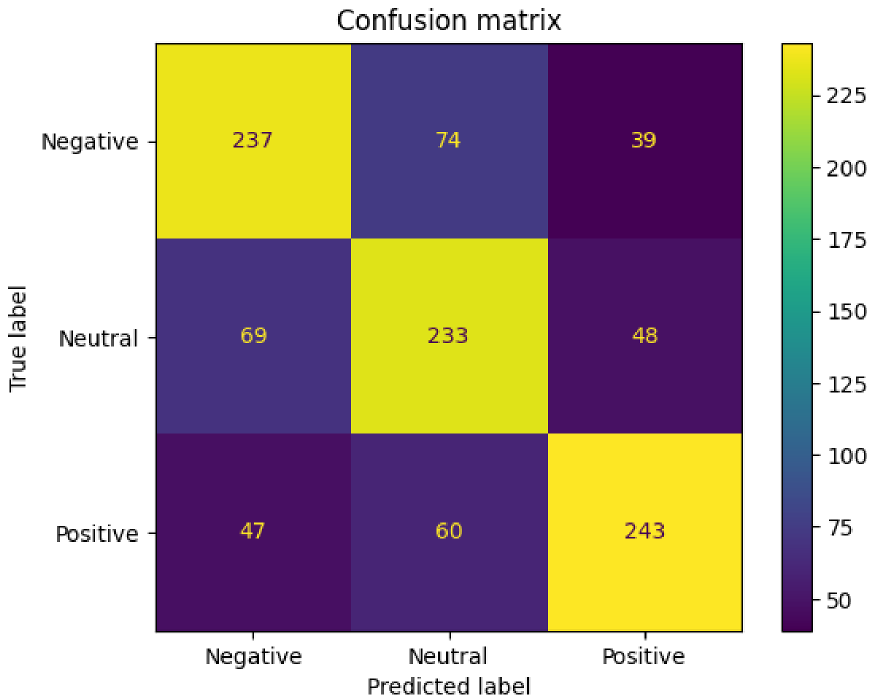
3.3 CNN

The follow model architecture i've tried are:

- embedding layer + 1 CNN + 1 FC + 1 Dropout (best result)
- embedding layer + 3 CNN + 1 FC + 1 Dropout (2nd best)



Hình 7: 1 CNN



Hình 8: Confusion matrix for CNN

References

[1] pytorch tutorial

[2] torchtext docs