Sentiment Analysis Report

Group 17

June 10, 2024

Abstract

Sentiment analysis task with financial statements in Vietnamese. Sentiment analysis can be applied to financial news articles, earnings reports, and other corporate communications to understand how the market is reacting to a company's performance or announcements. This can help investors and analysts identify potential opportunities or risks.

1 Dataset

1. **Description**:

   - Dataset consists of *Vietnamese statements* be crawled from articles, news on Internet, also be labeled sentimental states including *positive*, *negative* and *neutral*.
     (using *Scrapy* framework for crawling)

2. **Quantity**:

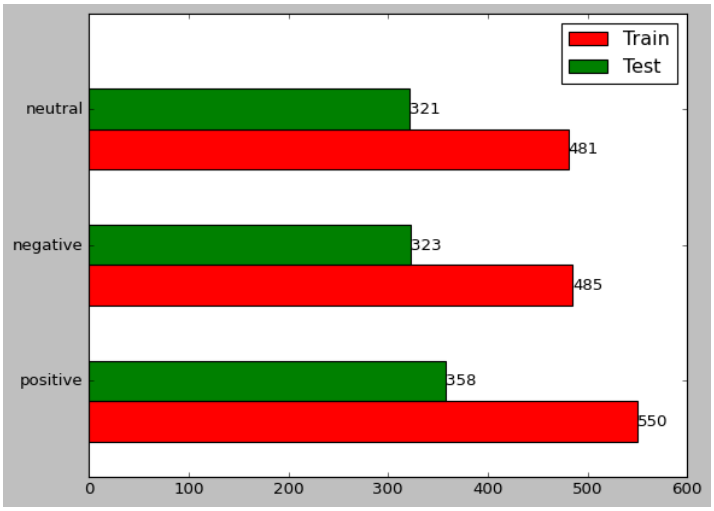


Figure 1: Collected samples

   - Around 2500 samples (be updated)
   - The number of sample in three classes are relatively equaling (this is important to ensure accurate evaluation)
   - After crawling, these are split into 3 plain text files for each class, then convert them to .json / .csv files.
   - Merging them into a file then split it into 2 files for training and testing section (also in .json / .csv files).

2 Model Analysis

***Purpose*:**

- Analyze architecture of trained models
- Show obtained results
- Compare the trained models

2.1 Bidirectional Long Short-Term Memory

1. **Preprocessing**:
   - Use one-hot encoding for labels
   - Shuffle training set for generalizatin
   - Set batching and prefetching for each dataset
2. **Architecture**:

Layer (type)	**Output Shape**	**Param #**
text_vectorization (TextVectorization)	(None, None)	0
embedding (Embedding)	(None, None, 128)	291712
bidirectional (Bidirectional)	(None, None, 128)	98816
dropout (Dropout)	(None, None, 128)	0
bidirectional_1 (Bidirectional)	(None, 64)	41216
dropout_1 (Dropout)	(None, 64)	0
dense (Dense)	(None, 32)	2080
dense_1 (Dense)	(None, 3)	99
Total params:		433923 (1.66 MB)
Trainable params:		433923 (1.66 MB)
Non-trainable params:		0 (0.00 Byte)

Table 1: Trained Bidirectional LSTM model summary

1. **text_vectorization** (*keras.layers.TextVectortization*):
   - Text Normalization: transform uppercase to lowercase, removing punctuation -> minimize vocabulary size.
   - Tokenization: tokenize each document to a sequence of token (word or n-gram).
   - Sequence Encoding: index each token to a numerical representation, build vocabulary then apply it to each sequence.
   - Padding or Truncation: based on the frequency of sequence length, choose the fixed length for sequences to avoid losing sentence meaning.
   - Out-of-Vocabulary: assign <UNK> token to non-present words or exclude them from sequence.

   Note: this layer do not include any parameters. You can solve this phase outside the model.
2. **embedding** (*keras.layer.Embedding*):
   Purpose: using for representing each sequence as a 2D vector (None, 128) (128: the dimension of each word)
   Mechanism: This built-in layer uses Word2Vec or GloVe mechanism to capture semantic relationships between each word with its neighbor words.

   For more: https://arxiv.org/pdf/1301.3781

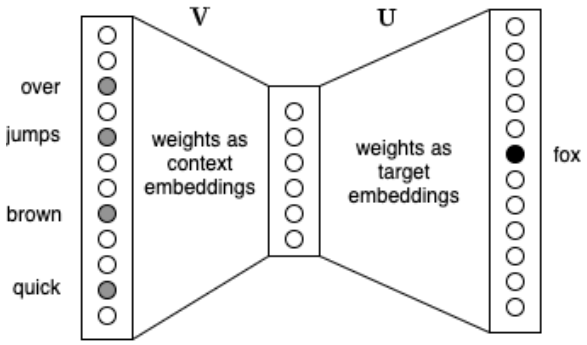
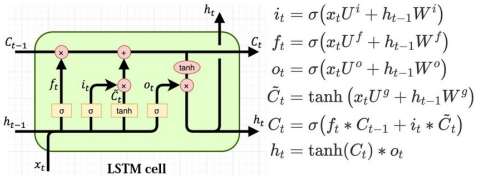


Figure 2: cbow word2vec.

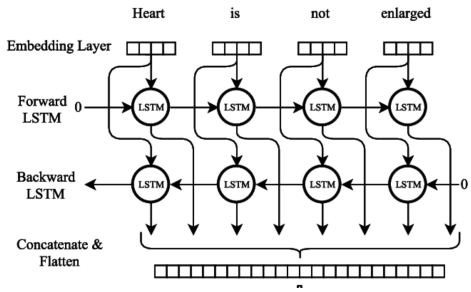
3. **bidirectional** (*keras.layers.Bidirectional*):
   Description: stands for Long Short-Term Memory, and it is a type of recurrent neural network (RNN) cell that is designed to address the vanishing gradient problem that can occur in traditional RNNs.
   General archiecture:

   i. Forget Gate (f): The forget gate determines what information from the previous cell state should be remembered or forgotten. This allows the LSTM to selectively retain relevant information from previous time steps.
   ii. Input Gate (i): The input gate controls what new information from the current input and previous hidden state should be added to the cell state.
   iii. Cell State (C): The cell state is the "memory" of the LSTM, which is updated at each time step based on the information from the forget gate and the input gate.
   iv. Output Gate (o): The output gate determines what part of the cell state and the current input should be used to compute the new hidden state.
   v. Hidden State (h): The hidden state is the output of the LSTM cell, which is a filtered version of the cell state, controlled by the output gate.

   Output: (None, None, 128) (batch size, sequence length, total dimension)
   For more: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43905.pdf



(a) LSTM cell



(b) Bidirectional LSTM architecture

4. **dropout**, **dropout_1** (*keras.layers.Dropout*):
   Avoid overfit by deactivate subset of neurons.
5. **bidirectional_1** (*keras.layers.Bidirectional*): the second Bidirectional LSTM layer aggregates the sequence information into a fixed-size 64-dimensional vector, instead of producing a sequence of vectors.
6. **dense**, **dense_1** (*keras.layers.Dense*): fully connected layers to sequentially reduce dimension output. The last layer use 'softmax' function to calculate distribution of 3 classes

3. **Metric Score**:
   Few parameters, short training time. low accuracy

	Precision	Recall	F1-Score	Support
positive	0.69	0.62	0.66	358
negative	0.75	0.54	0.63	323
neutral	0.49	0.68	0.57	321
accuracy			0.62	1002
macro avg	0.64	0.62	0.62	1002
weighted avg	0.65	0.62	0.62	1002

Table 2: Bidirectional LSTM result

2.2 Bidirectional Gated Recurrent Unit (GRU)

1. **Preprocessing**:
   Similar to Bidirecitonal LSTM model

2. **Architecture**:
   In Bidirectional GRU, we use the same general architecture as Bidirectionl LSTM, replace 2 bidirectional layers to *keras.layers.GRU* architecture.

Layer (type)	**Output Shape**	**Param #**
text_vectorization (TextVectorization)	(None, None)	0
embedding (Embedding)	(None, None, 128)	291712
bidirectional (Bidirectional)	(None, None, 128)	74496
dropout (Dropout)	(None, None, 128)	0
bidirectional_1 (Bidirectional)	(None, 64)	31104
dropout_1 (Dropout)	(None, 64)	0
dense (Dense)	(None, 32)	2080
dense_1 (Dense)	(None, 3)	99
Total params:		399491 (1.52 MB)
Trainable params:		399491 (1.52 MB)
Non-trainable params:		0 (0.00 Byte)

Table 3: Trained Bidirectional GRU model summary

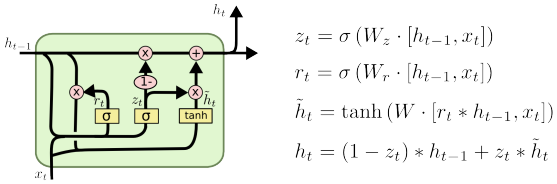


Figure 4: GRU cell architecture

For more: https://arxiv.org/pdf/1701.05923

1. Reset Gate (r): The reset gate determines how much of the past information (from the previous hidden state) should be forgotten or remembered.
2. Update Gate (z): The update gate controls how much of the previous hidden state and the current input should be used to compute the new hidden state. It determines the balance between the previous hidden state and the current input.

3. Candidate Hidden State ($\bar{h}$): The candidate hidden state is a new hidden state that is computed based on the current input and the previous hidden state. It represents the information the cell wants to remember.
4. Hidden State (h): The hidden state is the output of the GRU cell, which is a weighted combination of the previous hidden state and the candidate hidden state, controlled by the update gate.

Feature	**GRU**	**LSTM**
Architecture	Simpler, two gates	More complex, three gates
Memory Mechanism	Single hidden state	Separate cell state and hidden state
Performance	Faster, more efficient	More powerful, better long-term dependencies
Overfitting	Less prone	More prone

Table 4: Comparison of GRU and LSTM

3. **Metric Score**:
The results are statistically better than Bidirectional LSTM model but trivial.

	Precision	Recall	F1-score	Support
positive	0.70	0.68	0.69	358
negative	0.72	0.67	0.70	323
neutral	0.56	0.61	0.58	321
accuracy			0.65	1002
macro avg	0.66	0.65	0.65	1002
weighted avg	0.66	0.65	0.66	1002

Table 5: Bidirectional GRU Metric Score

2.3 XLM-Roberta pre-trained model

Description: XLM-RoBERTa (Cross-lingual Masked Language Model with RoBERTa) is a state-of-the-art language model developed by Facebook AI Research (FAIR). It is an extension of the RoBERTa (Robustly Optimized BERT Pretraining Approach) model, which is a more robust and improved version of the BERT (Bidirectional Encoder Representations from Transformers) model.
For more information: https://arxiv.org/pdf/1911.02116

1. **Pre-processing**:
useing the pre-trained XLMR-LARGE tokenizer including:

   - Wordpiece Tokenization: The XLMR-LARGE tokenizer uses a Wordpiece-based tokenization algorithm (BPE), similar to the one used by the BERT model.
   For more information about BPE tokenizer:
   https://arxiv.org/pdf/2309.08715
   - Padding and Truncation: The tokenizer pads or truncates the input sequence to a fixed length, as required by the XLMR-LARGE model. Padding is done by adding a special [PAD] token to the sequence, while truncation removes tokens from the end of the sequence.
   - Numerical Representation: The tokenizer converts the tokens in the input sequence to numerical token IDs, which are the indices of the tokens in the model's vocabulary.

2. **Architecture**:
Break down the different components of this model:

   - *RobertaEncoder*:

– This is the main encoder component of the RobertaModel, responsible for processing the input text.
– It consists of a TransformerEncoder, which is the core of the Transformer architecture.

```
RobertaModel(
  (encoder): RobertaEncoder(
    (transformer): TransformerEncoder(
      (token_embedding): Embedding(250002, 768, padding_idx=1)
      (layers): TransformerEncoder(
        (layers): ModuleList(
          (0-11): 12 x TransformerEncoderLayer(
            (self_attn): MultiheadAttention(
              (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
            )
            (linear1): Linear(in_features=768, out_features=3072, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
            (linear2): Linear(in_features=3072, out_features=768, bias=True)
            (norm1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (norm2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout1): Dropout(p=0.1, inplace=False)
            (dropout2): Dropout(p=0.1, inplace=False)
          )
        )
      )
      (positional_embedding): PositionalEmbedding(
        (embedding): Embedding(514, 768, padding_idx=1)
      )
      (embedding_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (head): RobertaClassificationHead(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (out_proj): Linear(in_features=768, out_features=3, bias=True)
    (activation_fn): ReLU()
  )
)
```

Figure 5: XLM-Roberta fine-tuned model

- *TransformerEncoder*:
  1. The TransformerEncoder contains multiple layers of TransformerEncoderLayer, which are the building blocks of the Transformer.
  2. Each TransformerEncoderLayer consists of:
     2.1. MultiheadAttention: This is the self-attention mechanism that allows the model to attend to different parts of the input sequence.
     2.2. Linear1 and Linear2: These are the feedforward neural network layers that follow the attention mechanism.
     2.3. Norm1 and Norm2: These are layer normalization layers that help stabilize the training process.
     2.4. Dropout1 and Dropout2: These are dropout layers that help prevent overfitting.
  3. The token_embedding is the learnable embedding layer that converts the input tokens into a dense representation.
  4. The positional_embedding adds positional information to the token embeddings, as the Transformer model is not inherently aware of the position of the tokens in the sequence.
  5. The embedding_layer_norm and dropout layers are applied to the combined token and positional embeddings before passing them to the Transformer encoder layers.
- *RobertaClassificationHead*:
  This is the classification head of the RobertaModel, which can be used for various downstream tasks, such as text classification.
  – It consists of a Dense layer, a Dropout layer, an out_proj linear layer, and an activation_fn (ReLU) layer.
  – The classification head takes the final hidden state of the Transformer encoder and produces the output logits f.

3. **Metric score**:
   Acceptable score, but still low in sentiment analysis task.

	f1-score	precision	recall	support
positive	0.8187	0.8589	0.7821	358
negative	0.8239	0.8371	0.8111	323
neutral	0.7076	0.6667	0.7539	321
accuracy	-	-	0.7824	1002
macro avg	0.7834	0.7875	0.7824	1002
weighted avg	0.7848	0.7903	0.7824	1002

2.4 PhoBert pre-trained model

Description: Pre-trained PhoBERT models are the state-of-the-art language models for Vietnamese. Two PhoBERT versions of "base" and "large" are the first public large-scale monolingual language models pre-trained for Vietnamese. PhoBERT pre-training approach is based on RoBERTa which optimizes the BERT pre-training procedure for more robust performance.
For more information: https://aclanthology.org/2020.findings-emnlp.92.pdf

1. **Pre-processing**:
   using *transformers.AutoTokenizer.from_pretrained("vinai/phobert-base")* to pre-process, including:

   - **Preprocessing**: The tokenizer will handle basic text preprocessing tasks, such as removing HTML tags, handling punctuation, and normalizing the input text (e.g., converting to lowercase).
   - **Tokenization**: The tokenizer will split the preprocessed text into tokens. Since it's using the PhoBERT model, the tokenization strategy is likely based on a subword algorithm, such as Byte-Pair Encoding (BPE) or SentencePiece. This allows the tokenizer to handle rare or unknown words more effectively by breaking them down into smaller subword units.
   - **Vocabulary Lookup**: The tokenizer maintains a vocabulary of all the unique tokens that the PhoBERT model was trained on. It will look up the index or ID of each token in the input text within this vocabulary. *Note: Encoding: The sequence of token IDs is then encoded as a numeric tensor, which can be used as input to the PhoBERT model. This encoding process may include padding or truncating the sequence to a fixed length, adding special tokens (e.g., start/end of sequence tokens), and handling out-of-vocabulary (OOV) tokens.

     ***Note**: When the model generates output, the tokenizer can be used to convert the predicted token IDs back into human-readable text by looking up the corresponding tokens in the vocabulary.

2. **Architecture**:

   This model is an instance of the RobertaForSequenceClassification class, which is a Transformer-based model used for sequence classification tasks. Break down the different components of the model:

   - RobertaModel: This is the core Transformer model, which is responsible for encoding the input sequence and producing the final representation.

     It consists of several key components:

     – RobertaEmbeddings: This handles the input representation, which is a combination of word embeddings, position embeddings, and token type embeddings.
     – RobertaEncoder: This is the main Transformer encoder, which consists of 12 Transformer layers, each with a self-attention mechanism, an intermediate dense layer with a GELU activation, and a final output layer.
   - RobertaClassificationHead: This is a task-specific head that is added on top of the RobertaModel to perform the sequence classification task. It consists of: A dense layer that projects the final hidden state (of size 768) to a new representation of size 768. A dropout

```
[12]: RobertaForSequenceClassification(
        (roberta): RobertaModel(
          (embeddings): RobertaEmbeddings(
            (word_embeddings): Embedding(64001, 768, padding_idx=1)
            (position_embeddings): Embedding(258, 768, padding_idx=1)
            (token_type_embeddings): Embedding(1, 768)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (encoder): RobertaEncoder(
            (layer): ModuleList(
              (0-11): 12 x RobertaLayer(
                (attention): RobertaAttention(
                  (self): RobertaSelfAttention(
                    (query): Linear(in_features=768, out_features=768, bias=True)
                    (key): Linear(in_features=768, out_features=768, bias=True)
                    (value): Linear(in_features=768, out_features=768, bias=True)
                    (dropout): Dropout(p=0.1, inplace=False)
                  )
                  (output): RobertaSelfOutput(
                    (dense): Linear(in_features=768, out_features=768, bias=True)
                    (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
                    (dropout): Dropout(p=0.1, inplace=False)
                  )
                )
                (intermediate): RobertaIntermediate(
                  (dense): Linear(in_features=768, out_features=3072, bias=True)
                  (intermediate_act_fn): GELUActivation()
                )
                (output): RobertaOutput(
                  (dense): Linear(in_features=3072, out_features=768, bias=True)
                  (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
                  (dropout): Dropout(p=0.1, inplace=False)
                )
              )
            )
          )
        )
        (classifier): RobertaClassificationHead(
          (dense): Linear(in_features=768, out_features=768, bias=True)
          (dropout): Dropout(p=0.1, inplace=False)
          (out_proj): Linear(in_features=768, out_features=3, bias=True)
        )
      )
```

Figure 6: PhoBERT fine-tuned model

layer to regularize the model. An output projection layer that maps the 768-dimensional representation to the desired number of classes (in this case, 3).

The details of each component:

- RobertaEmbeddings:
  1. The word embeddings are learned from a vocabulary of 64,001 tokens, with each embedding having a size of 768.
  2. The position embeddings have a maximum sequence length of 258, and each position embedding also has a size of 768.
  3. The token type embeddings have a single embedding (i.e., 1 dimension), which is also 768-dimensional.
  4. The embeddings are passed through a layer normalization and a dropout layer before being fed into the RobertaEncoder.
- RobertaEncoder: The encoder consists of 12 Transformer layers, each with the following components:
  5. RobertaAttention: This handles the self-attention mechanism, with separate linear layers for the query, key, and value projections, followed by a dropout layer.
  6. RobertaSelfOutput: This takes the output of the self-attention layer and applies a dense layer, layer normalization, and dropout.
  7. RobertaIntermediate: This is the intermediate dense layer with a GELU activation function.

8. RobertaOutput: This takes the output of the intermediate layer and applies another dense layer, layer normalization, and dropout.

The output of the 12 Transformer layers is the final representation of the input sequence.

- RobertaClassificationHead:

9. This head takes the final representation from the RobertaModel and applies a dense layer to project it to a 768-dimensional vector.
10. A dropout layer is then applied for regularization.
11. Finally, an output projection layer maps the 768-dimensional vector to the desired number of classes (3 in this case).

***NOTE**: Both fine-tuning methods are based on BERT which is based on the Transformer architecture and is designed to pre-train deep bidirectional representations from unlabeled text, which can then be fine-tuned for a wide range of natural language processing (NLP) tasks.
For more information about Transformer: https://arxiv.org/pdf/1706.03762

3. **Metric Score**:
Over 80% in all categories, still need to improve.
(possibly because of lack of data: around 1500 for training section)

	Precision	Recall	F1-score	Support
positive	0.8446	0.8045	0.8240	358
negative	0.8827	0.8390	0.8603	323
neutral	0.7090	0.7819	0.7437	321
accuracy			0.8084	1002
macro avg	0.8121	0.8085	0.8094	1002
weighted avg	0.8135	0.8084	0.8100	1002