NLP in the Financial Market — Sentiment Analysis

How much the latest NLP models outperform traditional models — From Lexicon approach to BERT for a sentiment analysis task on financial texts

Yuki Takahashi Oct 3, 2020 · 9 min read ★

Photo by Markus Spiske on Unsplash

Deep learning in Computer Vision has been successfully adopted in a variety of applications since a pioneer CNN called AlexNet on ImageNet in 2012. On the contrary, NLP has been behind in terms of the deep neural network utilisation. A lot of applications which claim the use of AI often use some sort of rule-based algorithm and traditional machine learning rather than deep neural networks. In 2018 saw a state-of-the-art (STOA) model called BERT outperformed human scores in some NLP tasks. Here, I apply several models for a sentiment analysis task to see how useful they are in the financial market where I'm from. The code is in jupyter notebook and available in git repo.

1. Introduction

NLP tasks can be broadly categorised as follows.

1. Text Classification — filtering spam emails, categorising documents

3. Text Meaning — topic modeling, search, question answering

4. Sequence to Sequence —machine translation, text summarisation, Q&A

5. Dialog Systems

Different approaches will be required for different tasks and in most case are the combination of multiple NLP techniques. When developing a bot, the backend logic is often a rule based search engine and ranking algorithms to form a natural communication.

There is a good reason for this. The language has grammar and word orders, which could be better handled by a rule-based approach, while machine learning approach could learn collocations and word similarities better. Vectorisation techniques such as word2vec, bag-of-word help the model to express a text in mathematical way. The most famous examples are:

```
King - Man + Woman = Queen

Paris - France + UK = London
```

The first example describes the gender relationship and the second captures the concept of capital city. In these approaches, however, the context are not captured as the same word is always represented by the same vector in any text, which is not true in many cases. Recurrent Neural Network (**RNN**) architecture, which uses the previous information from input sequence and handles time series data, performed well in capturing and remembering the context. One of the typical architecture is Long short-term memory (**LSTM**), which is composed of input gate, output gate and forget gate to overcome the varnishing gradient problem of RNN. The are many improved models based on LSTM, such as bidirectional LSTM to capture the context not only from preceding words but also from backwards. These were good for some specific tasks but not quite in practical applications.

encoder-decoder stack based on attention mechanism. Bidirectional Encoder Representations from Transformers (**BERT**), is a masked language model with multiple encoder stack by Google in 2018, which achieved STOA in GLUE, SQuAD and SWAG benchmarks with a big improvement. There are a number of articles and blogs explaining the architecture, such as the one by Jay Alammar.

Working in the financial industry, I struggled to see the sufficient robust performance in our past R&D of machine learning models on NLP for the production use in trading systems over the past few years. Now that BERT based models are getting matured and easy to use thanks to Huggingface implementation and many pre-trained models have been made public. My goal is to see if this latest development in NLP reaches a good level to use in my domain. In this post, I compare different models on a rather simple task of the sentiment analysis on financial texts as a baseline to judge if it's worth trying another R&D in a real solution.

Models compared here are:

1. Rule based approach using Lexicon
2. Traditional Machine Learning approach using Tfidf
3. LSTM as a Recurrent Neural Network architecture
4. BERT (and ALBERT)

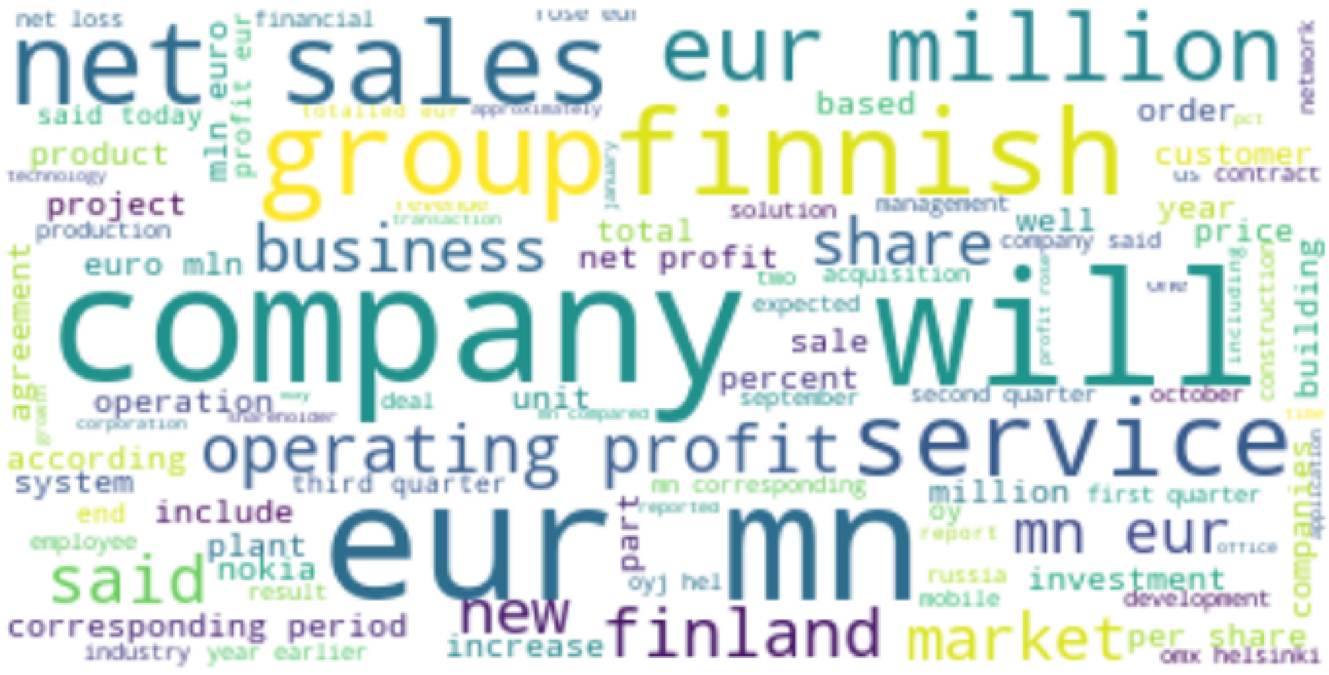
2. Input Data

I took the following two inputs to represent different languages in the industry for the sentiment analysis task.

2. Tweets from Stocktwits as trader informal talk

I will write another post for the latter, so focus on the former data here. It is an example of texts containing more formal financial domain specific languages and I used FinancialPhraseBank by Malo et al. (2014), which consist of 4845 hand labeled headline texts by 16 persons and provided with agree level. I used 75% agreed labels with 3448 texts as training data.

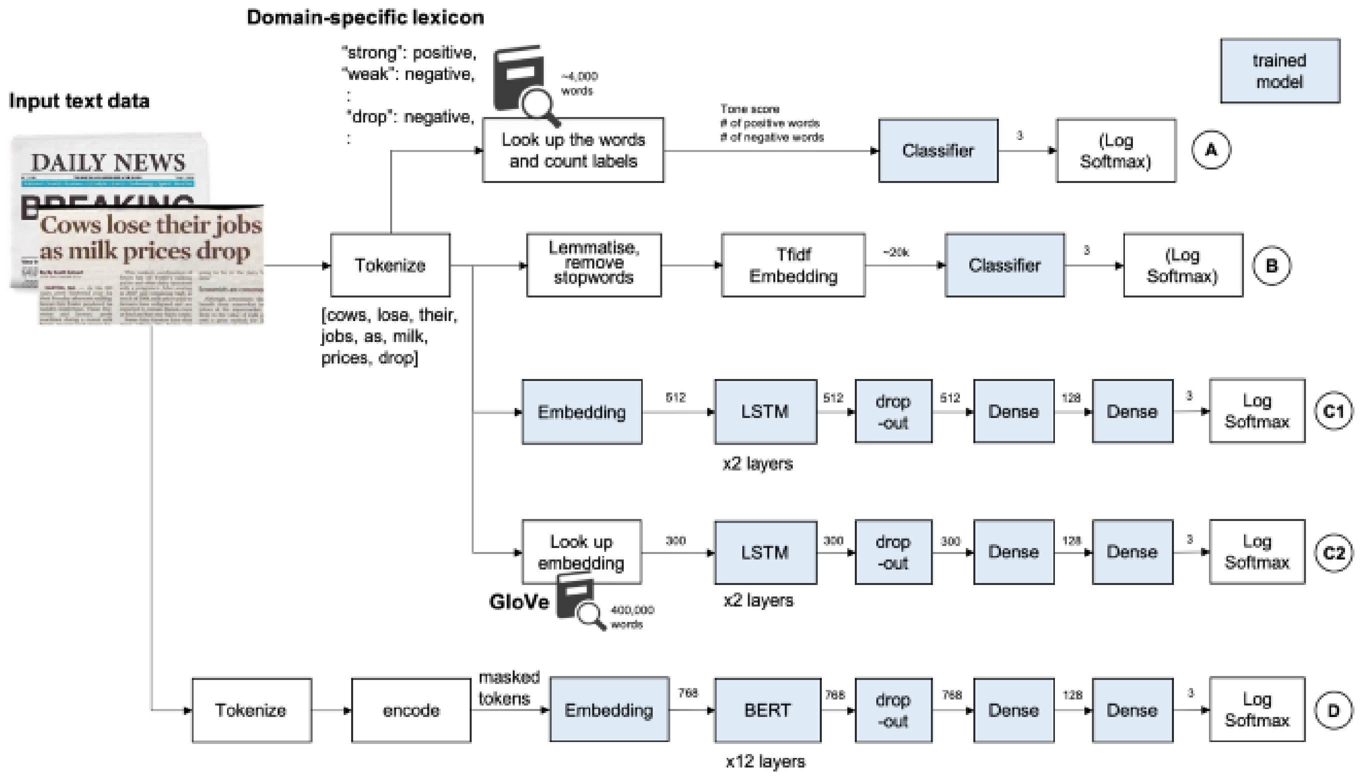
```
## Input text samples

positive "Finnish steel maker Rautaruukki Oyj ( Ruukki ) said on July
7 , 2008 that it won a 9.0 mln euro ( $ 14.1 mln ) contract to supply
and install steel superstructures for Partihallsforbindelsen bridge
project in Gothenburg , western Sweden."

neutral "In 2008 , the steel industry accounted for 64 percent of the
cargo volumes transported , whereas the energy industry accounted for
28 percent and other industries for 8 percent."

negative "The period-end cash and cash equivalents totaled EUR6 .5 m
, compared to EUR10 .5 m in the previous year."
```



Word Cloud for the first input training text, image by author

3. Models

Here are the four models I compared the performance.



NLP Models evaluated, Image by Author

A. Lexicon-based Approach

Creating domain specific dictionaries is a traditional approach and simple yet strong in some cases where the source is from a particular person or media. Loughran and McDonald Sentiment Word Lists. This list contains more than 4k words which appears on financial statements with sentiment labels. Note: This data requires the license to use for commercial application. Please check their website before use.

```
## Sample

negative: ABANDON
negative: ABANDONED
constraining: STRICTLY
```

for this kind of approach. Words such as not, no, don't, etc. change the meaning of negative words to positive and here I simply flip the sentiment if one of negate words occurring within three words preceding a positive words.

Then, the tone score is defined as follows and fed into classifiers along the positive/negative counts.

```
tone_score = 100 * (pos_count — neg_count) / word_count
```

14 different classifier were trained with default parameters, then used grid search cross validation for hyper-parameter tuning of Random Forest.

```python
classifiers = []
classifiers.append(("SVC", SVC(random_state=random_state)))
classifiers.append(("DecisionTree", DecisionTreeClassifier(random_state=random_state)))
classifiers.append(("AdaBoost", AdaBoostClassifier(DecisionTreeClassifier(random_state=random_st
classifiers.append(("RandomForest", RandomForestClassifier(random_state=random_state, n_estimato
classifiers.append(("ExtraTrees", ExtraTreesClassifier(random_state=random_state)))
classifiers.append(("GradientBoosting", GradientBoostingClassifier(random_state=random_state)))
classifiers.append(("MultipleLayerPerceptron", MLPClassifier(random_state=random_state)))
classifiers.append(("KNeighboors", KNeighborsClassifier(n_neighbors=3)))
classifiers.append(("LogisticRegression", LogisticRegression(random_state = random_state)))
classifiers.append(("LinearDiscriminantAnalysis", LinearDiscriminantAnalysis()))
classifiers.append(("GaussianNB", GaussianNB()))
classifiers.append(("Perceptron", Perceptron()))
classifiers.append(("LinearSVC", LinearSVC()))
classifiers.append(("SGD", SGDClassifier()))

cv_results = []
for classifier in classifiers :
    cv_results.append(cross_validate(classifier[1], X_train, y=Y_train, scoring=scoring, cv=kfol
```

finphrase_14clf.py hosted with ♡ by GitHub view raw

```python
# Use Random Forest Classifier
rf_clf = RandomForestClassifier()
```

```
                'min_samples_split': [1, 3, 5, 10],
                'min_samples_leaf': [1, 2, 3, 5],
                'max_features': [1, 2, 3],
                'max_depth': [None],
                'criterion': ['gini'],
                'bootstrap': [False]}

model = GridSearchCV(rf_clf, param_grid=param_grid, cv=kfold, scoring=scoring, verbose=verbose,
model.fit(X_train, Y_train)
rf_best = model.best_estimator_
```

finphrase_rf_train.py hosted with ♡ by GitHub view raw

B. Traditional Machine Learning on Tfidf vector

The inputs were tokenized by NLTK word_tokenize(), then lemmatised and stop words were removed. Then fed into TfidfVectorizer and classified by Logistic Regression and Random Forest Classifier.

```
### Logistic Regression
pipeline1 = Pipeline([
    ('vec', TfidfVectorizer(analyzer='word')),
    ('clf', LogisticRegression())])

pipeline1.fit(X_train, Y_train)

### Random Forest with Grid Search CV
pipeline2 = Pipeline([
    ('vec', TfidfVectorizer(analyzer='word')),
    ('clf', RandomForestClassifier())])

param_grid = {'clf__n_estimators': [10, 50, 100, 150, 200],
              'clf__min_samples_leaf': [1, 2],
              'clf__min_samples_split': [4, 6],
              'clf__max_features': ['auto']
             }

model = GridSearchCV(pipeline2, param_grid=param_grid, cv=kfold, scoring=scoring, verbose=verbos
model.fit(X_train, Y_train)
tfidf_best = model.best_estimator_
```

C. LSTM — a Recurrent Neural Network

As LSTM is designed to remember long-term memory which expresses the context, used a custom tokenizer to extract alphabetical letters as they are without lemmatisation or stop word removal. Then the inputs were fed into an embedding layer, then two lstm layer. To avoid overfitting, apply dropout as is often the case, then fully-connected layers and finally take the log softmax.

```
class TextClassifier(nn.Module):
  def __init__(self, vocab_size, embed_size, lstm_size, dense_size, output_size, lstm_layers=2,
    """
    Initialize the model
    """
    super().__init__()
    self.vocab_size = vocab_size
    self.embed_size = embed_size
    self.lstm_size = lstm_size
    self.dense_size = dense_size
    self.output_size = output_size
    self.lstm_layers = lstm_layers
    self.dropout = dropout

    self.embedding = nn.Embedding(vocab_size, embed_size)
    self.lstm = nn.LSTM(embed_size, lstm_size, lstm_layers, dropout=dropout, batch_first=False)
    self.dropout = nn.Dropout(dropout)

    if dense_size == 0:
      self.fc = nn.Linear(lstm_size, output_size)
    else:
      self.fc1 = nn.Linear(lstm_size, dense_size)
      self.fc2 = nn.Linear(dense_size, output_size)

    self.softmax = nn.LogSoftmax(dim=1)

def init_hidden(self, batch_size):
    """
    Initialize the hidden state
    """
    weight = next(self.parameters()).data
    hidden = (weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_(),
def forward(self, nn_input_text, hidden_state):
    """
    Perform a forward pass of the model on nn_input
    """
    batch_size = nn_input_text.size(0)
    nn_input_text = nn_input_text.long()
    embeds = self.embedding(nn_input_text)
    lstm_out, hidden_state = self.lstm(embeds, hidden_state)
    # Stack up LSTM outputs, apply dropout
    lstm_out = lstm_out[-1,:,:]
    lstm_out = self.dropout(lstm_out)
    # Dense layer
    if self.dense_size == 0:
      out = self.fc(lstm_out)
    else:
      dense_out = self.fc1(lstm_out)
      out = self.fc2(dense_out)
    # Softmax
    logps = self.softmax(out)

    return logps, hidden_state
```

finphrase_lstm_model.py hosted with ♡ by GitHub view raw

As alternative, also tried GloVe word embedding from Stanford, which is an unsupervised learning algorithm for obtaining vector representations for words. Here, took the pre-trained on Wikipedia and Gigawords with 6B tokens, 400k vocab size and 300 dimentional vectors. Around 90% of words in our vocab were found in this GloVe vocab and the rest are initialised randomly.

D. BERT (and also ALBERT as an alternative of BERT model)

I used pytorch implementation of BERT model by transformers from Huggingface. Now (v3) they provide tokenizer as well as encoder which generate text ids, pad masks and segment ids that can be directly used in their BertModel and no custom implementation was necessary for a standard training process.

option without enough budget for computation resource and also no enough data, so I used pre-trained models and fine tuned. The pre-trained models used as as follows:

- BERT: bert-base-uncased
- ALBERT: albert-base-v2

Training process with the pre-trained bert looks like this.

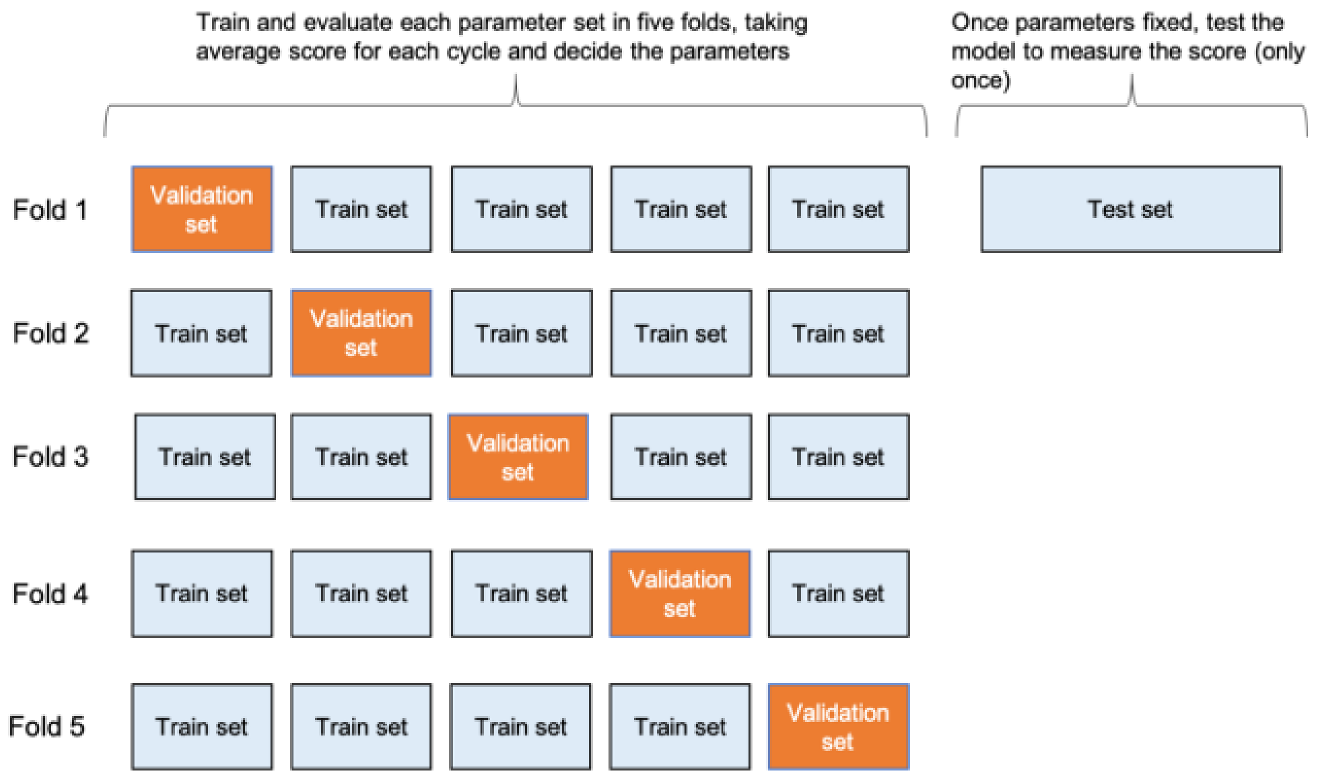
```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=3)

def train_bert(model, tokenizer)
  # Move model to GUP/CPU device
  device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
  model = model.to(device)

  # Load data into SimpleDataset (custom dataset class)
  train_ds = SimpleDataset(x_train, y_train)
  valid_ds = SimpleDataset(x_valid, y_valid)

  # Use DataLoader to load data from Dataset in batches
  train_loader = torch.utils.data.DataLoader(train_ds, batch_size=batch_size, shuffle=True)
  valid_loader = torch.utils.data.DataLoader(valid_ds, batch_size=batch_size, shuffle=False)

  # Optimizer and learning curve decay
  num_total_opt_steps = int(len(train_loader) * num_epochs)
  optimizer = AdamW_HF(model.parameters(), lr=learning_rate, correct_bias=False)
  scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=num_total_opt_steps*w

  # Set Train Mode
  model.train()

  # Tokenizer Parameter
  param_tk = {
    'return_tensors': "pt",
    'padding': 'max_length',
    'max_length': max_seq_length,
    'add_special_tokens': True,
    'truncation': True
  }
  best_f1 = 0.
  early_stop = 0
  train_losses = []
  valid_losses = []

  for epoch in tqdm(range(num_epochs), desc="Epoch"):
    # print('================     epoch {}     ==============='.format(epoch+1))
    train_loss = 0.

    for i, batch in enumerate(train_loader):
      # Input features and labels from batch and move to device
      x_train_bt, y_train_bt = batch
      x_train_bt = tokenizer(x_train_bt, **param_tk).to(device)
      y_train_bt = torch.tensor(y_train_bt, dtype=torch.long).to(device)

      # Reset gradient
      optimizer.zero_grad()

      # Feedforward prediction
      loss, logits = model(**x_train_bt, labels=y_train_bt)

      # Backward Propagation
      loss.backward()

      # Training Loss
      train_loss += loss.item() / len(train_loader)

      # Gradient clipping is not in AdamW anymore (so you can use amp without issue)
      torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)

      # Update Weights and Learning Rate
      optimizer.step()
      scheduler.step()

    train_losses.append(train_loss)

    # Move to Evaluation Mode
    model.eval()

    # Initialize for Validation
    val_loss = 0.
    y_valid_pred = np.zeros((len(y_valid), 3))
```

```
   # Input features and labels from batch and move to device
   x_valid_bt, y_valid_bt = batch
   x_valid_bt = tokenizer(x_valid_bt, **param_tk).to(device)
   y_valid_bt = torch.tensor(y_valid_bt, dtype=torch.long).to(device)
   loss, logits = model(**x_valid_bt, labels=y_valid_bt)
   val_loss += loss.item() / len(valid_loader)
 valid_losses.append(val_loss)

 # Calculate metrics
 acc, f1 = metric(y_valid, np.argmax(y_valid_pred, axis=1))

 # If improving, save the model. If not, count up for early stopping
 if best_f1 < f1:
  early_stop = 0
  best_f1 = f1
 else:
  early_stop += 1

 print('epoch: %d, train loss: %.4f, valid loss: %.4f, acc: %.4f, f1: %.4f, best_f1: %.4f, la
     (epoch+1, train_loss, val_loss, acc, f1, best_f1, scheduler.get_last_lr()[0]))

 if device == 'cuda:0':
  torch.cuda.empty_cache()

 # Early stop if it reaches patience number
 if early_stop >= patience:
  break

 # Back to Train Mode
 model.train()
return model
```

finphrase_bert.py hosted with ♡ by GitHub view raw

4. Evaluation

First, input data are split to train set and test set at 8:2. The test set is kept untouched until all parameters are fixed and used only once for each model. As the dataset is not large, cross validation is used to evaluate the parameter sets. In addition, to overcome


Train and evaluate each parameter set in five folds, taking average score for each cycle and decide the parameters
Once parameters fixed, test the model to measure the score (only once)
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Validation set
Train set
Test set

Cross Validation and Test set split, Image by Author

Evaluation is based F1 score as the input data are imbalanced, while the accuracy is also referred.

```python
def metric(y_true, y_pred):
    acc = accuracy_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred, average='macro')
    return acc, f1

scoring = {'Accuracy': 'accuracy', 'F1': 'f1_macro'}
refit = 'F1'
kfold = StratifiedKFold(n_splits=5)
```

finphrase_eval.py hosted with ♡ by GitHub view raw

Grid Search Cross Validation is used for Model A and B, whereas custom Cross Validation is performed for Deep Neural Network models of C and D.

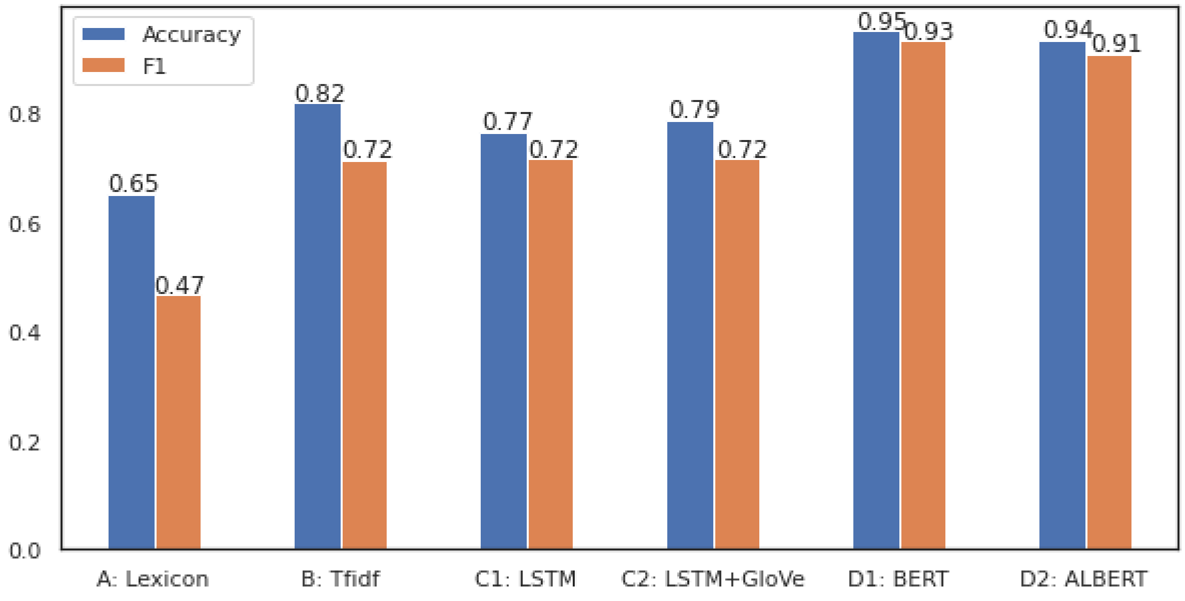
```python
# Stratified KFold
```

```python
for n_fold, (train_indices, valid_indices) in enumerate(skf.split(y_train, y_train)):
  # Model
  model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=3)

  # Input data for this fold
  x_train_fold = x_train[train_indices]
  y_train_fold = y_train[train_indices]
  x_valid_fold = x_train[valid_indices]
  y_valid_fold = y_train[valid_indices]

  # Do training
  train_bert(model, x_train_fold, y_train_fold, x_valid_fold, y_valid_fold)
```

finphrase_kfold.py hosted with ♡ by GitHub view raw

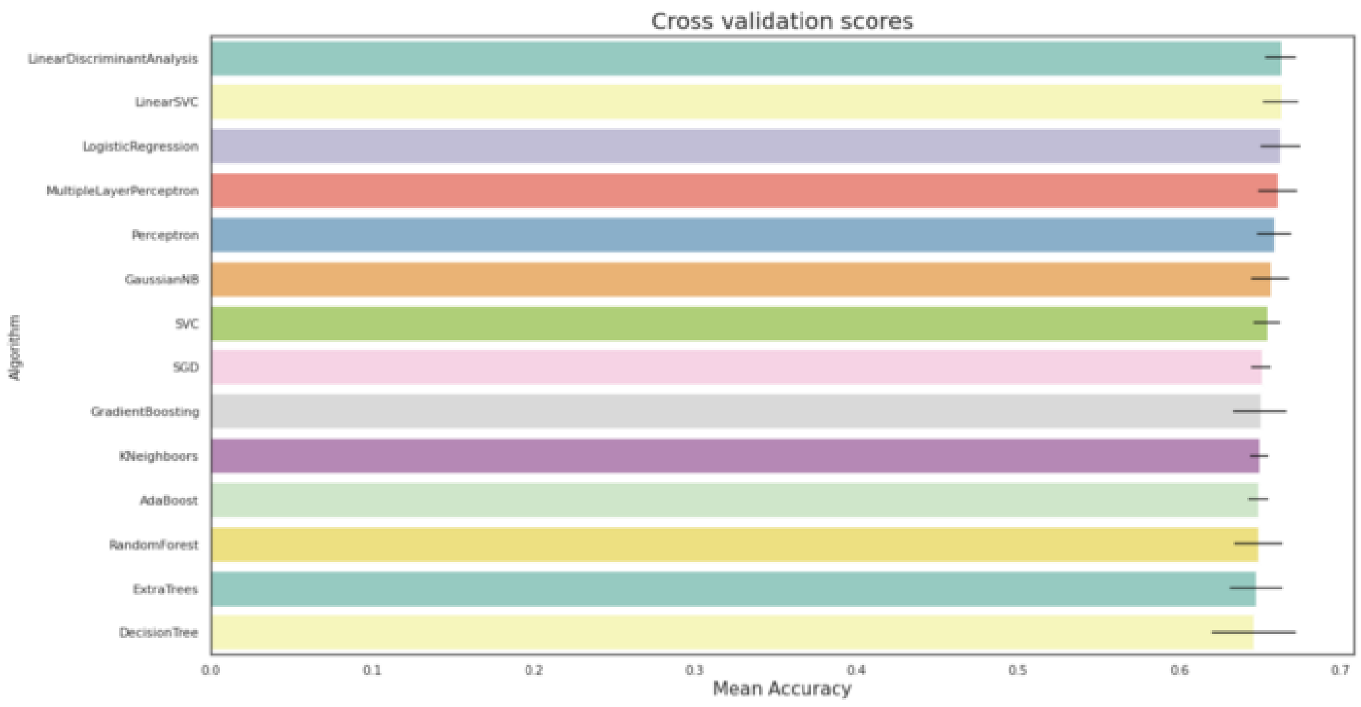
5. Result

Fine tuned BERT based models clearly outperform the other models after spending more or less similar time in hyper-parameter tuning.

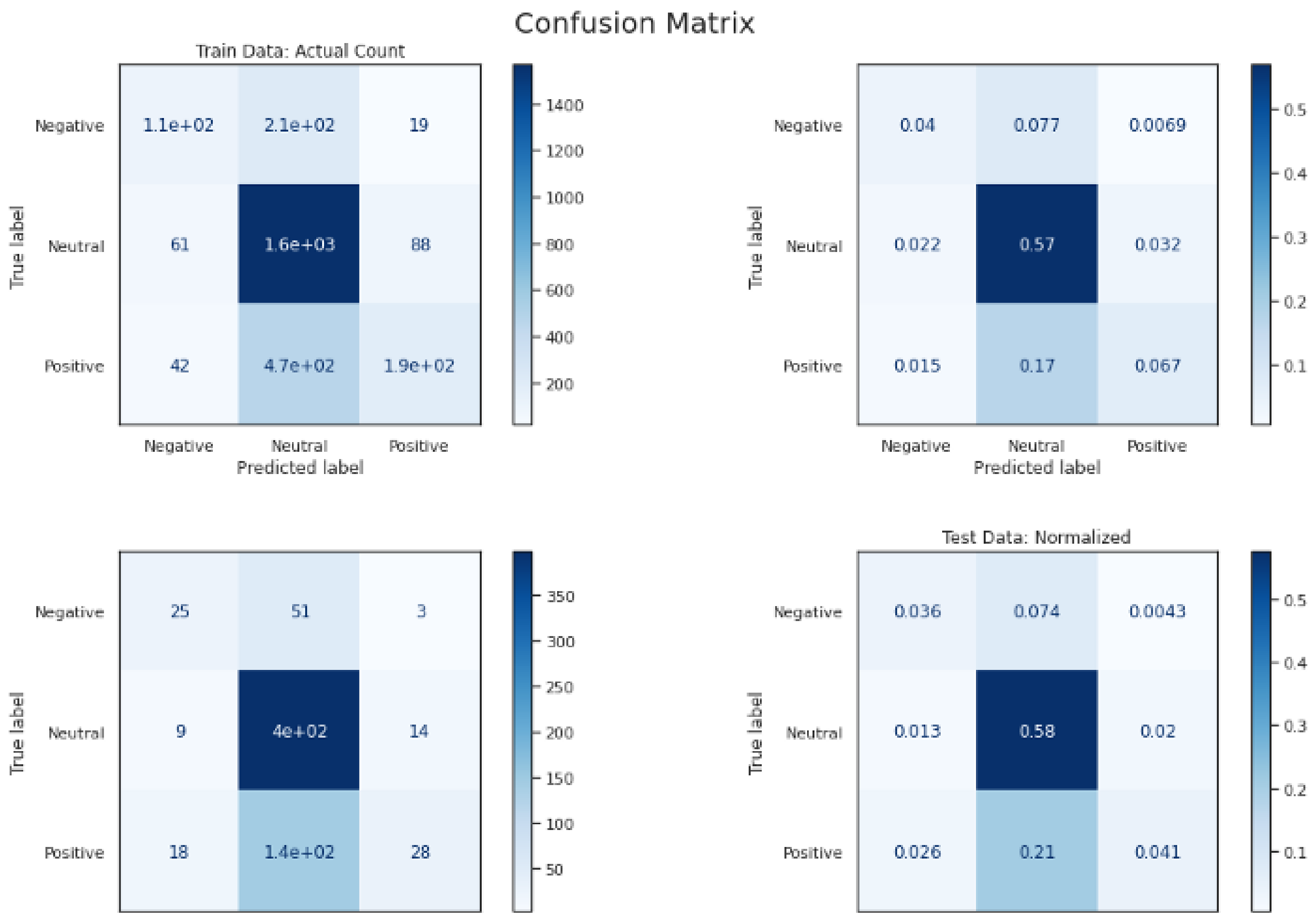


Evaluation Score of each model, Image by Author

Model A didn't perform well because the input was too simplified as tone score, which is a single value judging the sentiment, and the random forest model ended up labelling

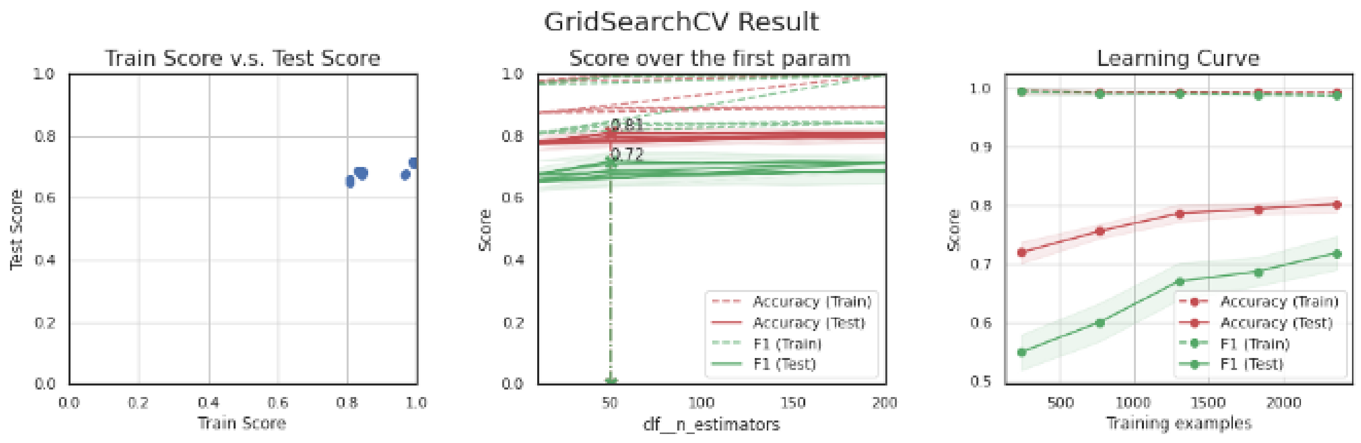


Accuracy of each classifier, Image by Author

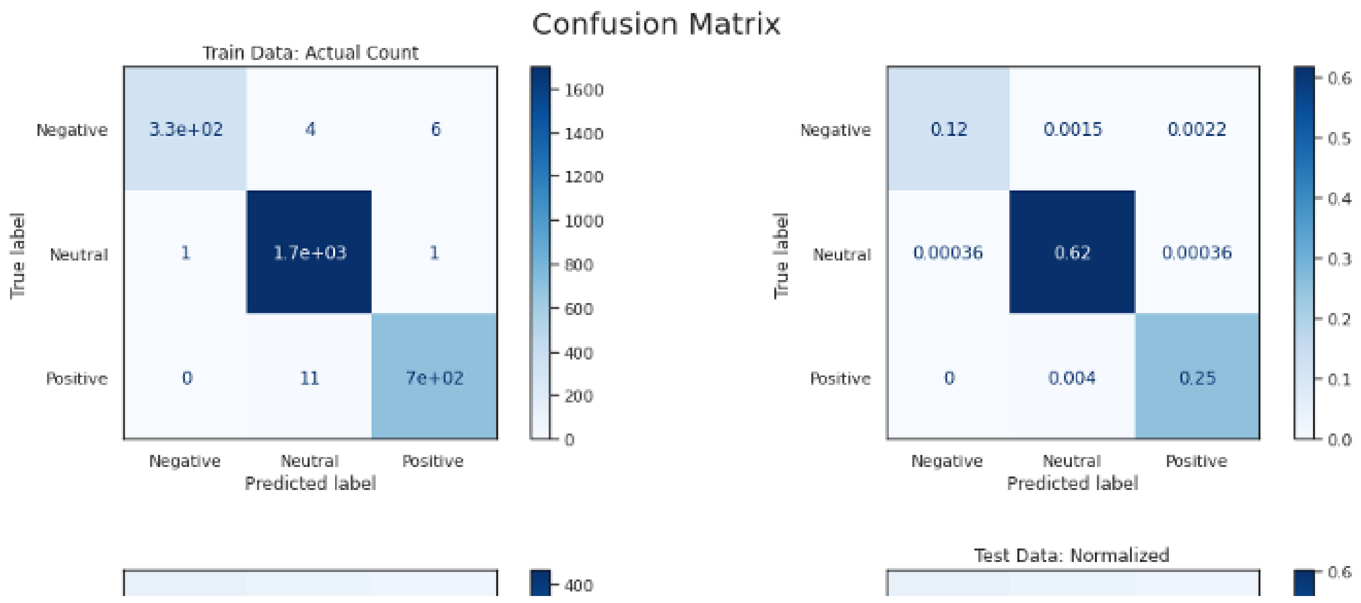


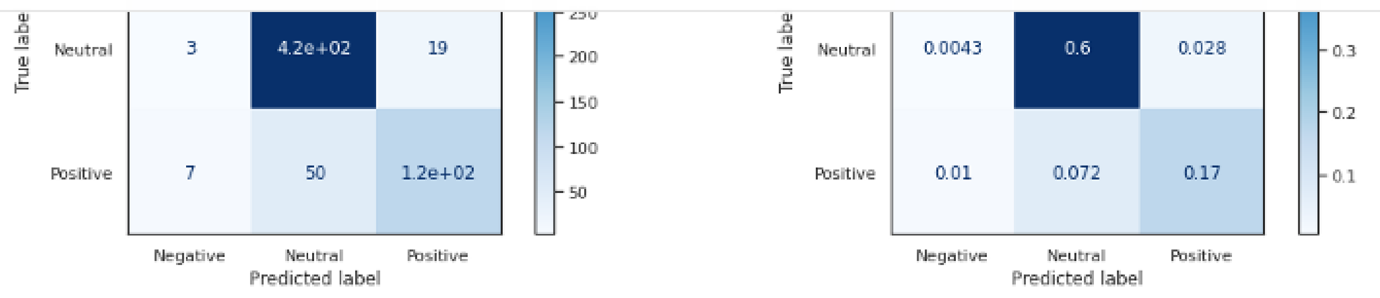
We didn't balance the input data using methods like under/over-sampling or SMOTE because it can rectify this issue but would deviate from the actual situation where the imbalance exists. Potential improvement to this model is to build a custom lexicon instead of L-M dictionary if the cost can be justified to build a lexicon per problem to solve. More complex negation could also improve the accuracy of the prediction.

Model B was muchbetter than the previous model, but it overfits to the training set with almost 100% of accuracy and f1 score and failed to be generalised. I tried to reduce the model complexity to avoid overfitting but it ended up lower score in validation set. The balancing data could help solve this issue or collect much more data.



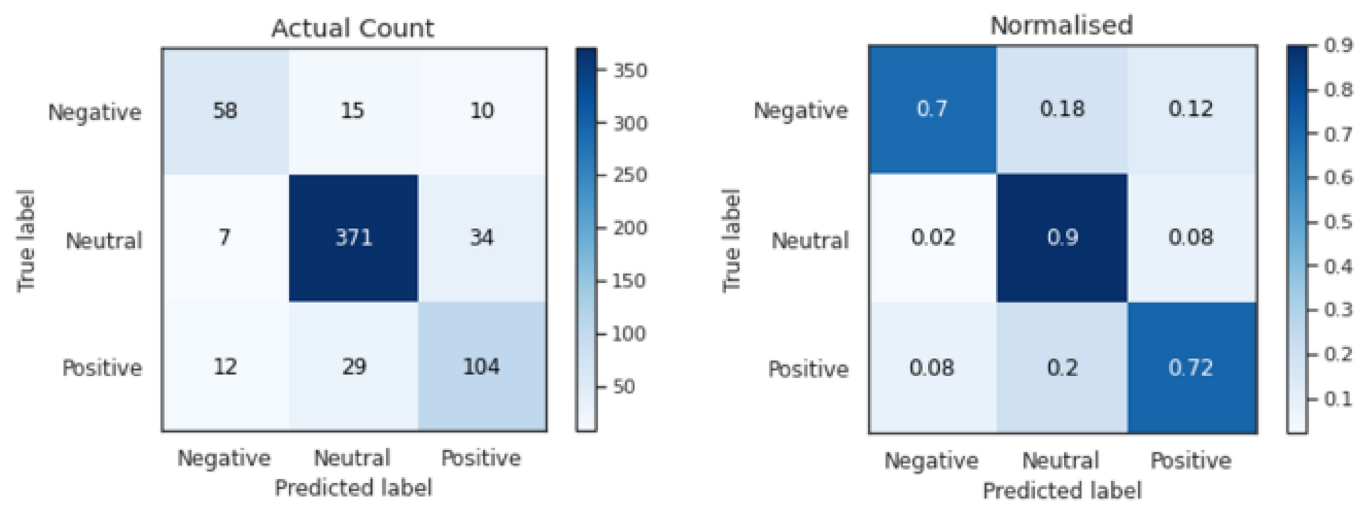
Grid Search Cross Validation result, Image by Author





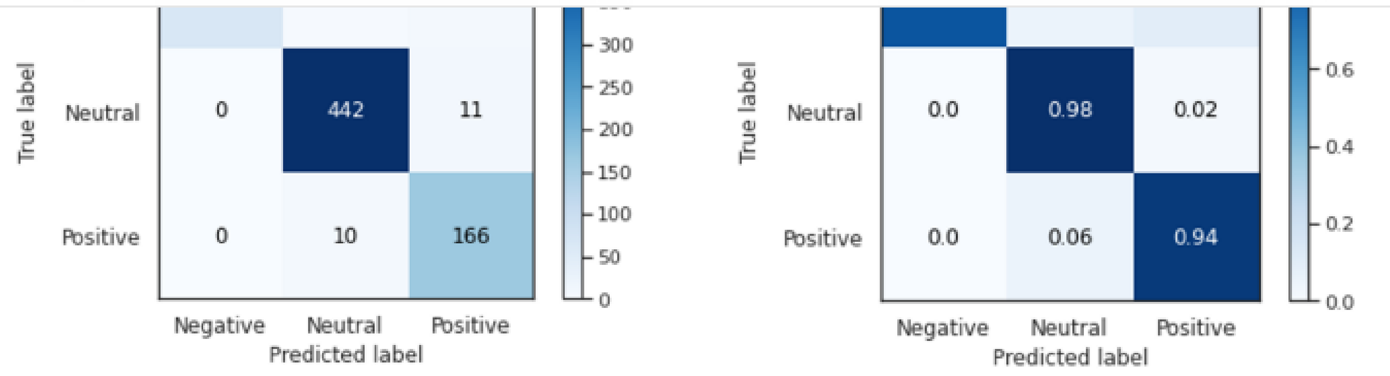
Confusion Matrix, Image by Author

Model C produced a similar result to the previous model but not improved much. In fact, the number of training data were insufficient to train the neural network from scratch and needed to a number of epochs, which tends to overfit. The pre-trained GloVe embedding does not improve the result. One possible improvement on this latter model is to train the GloVe using a bunch of text from similar domain such as 10K, 10Q financial statements instead of using pre-trained model from wikipedia.

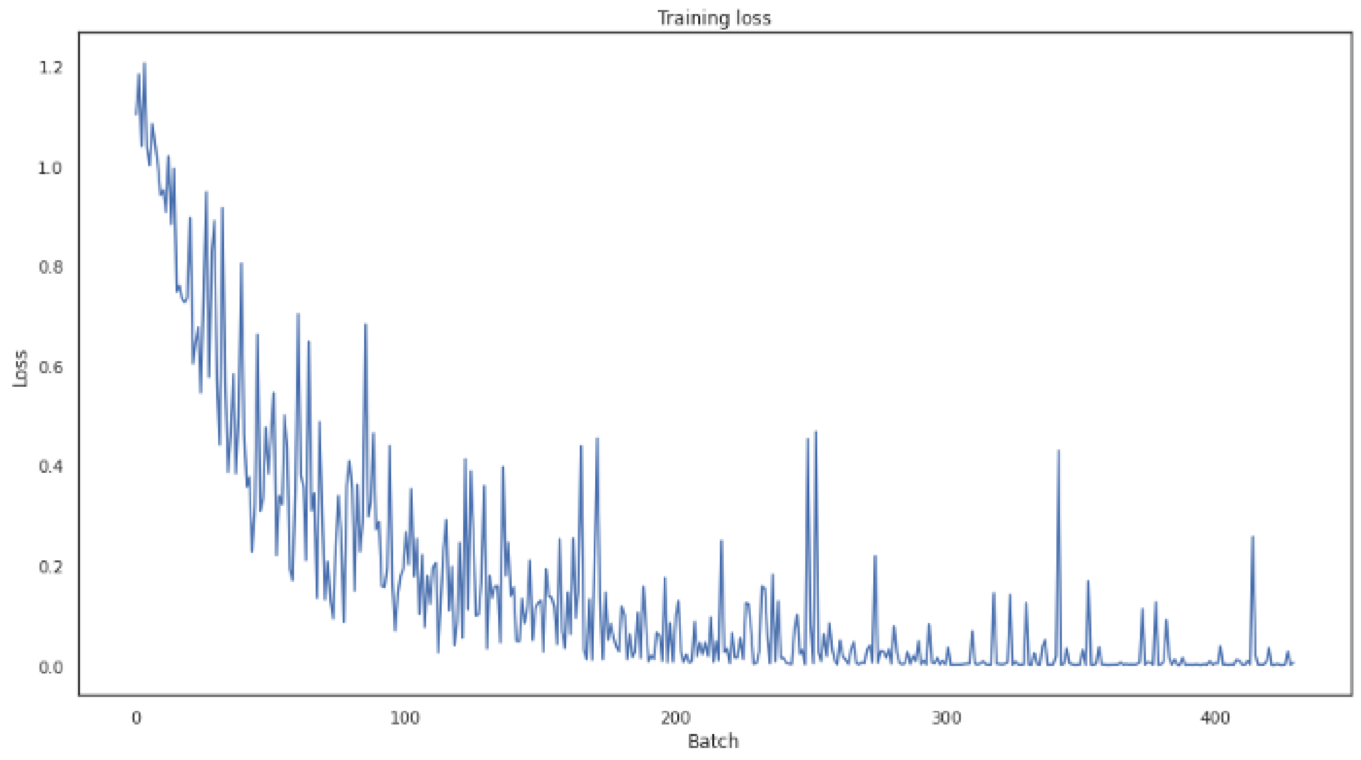


Confusion Matrix, Image by Author

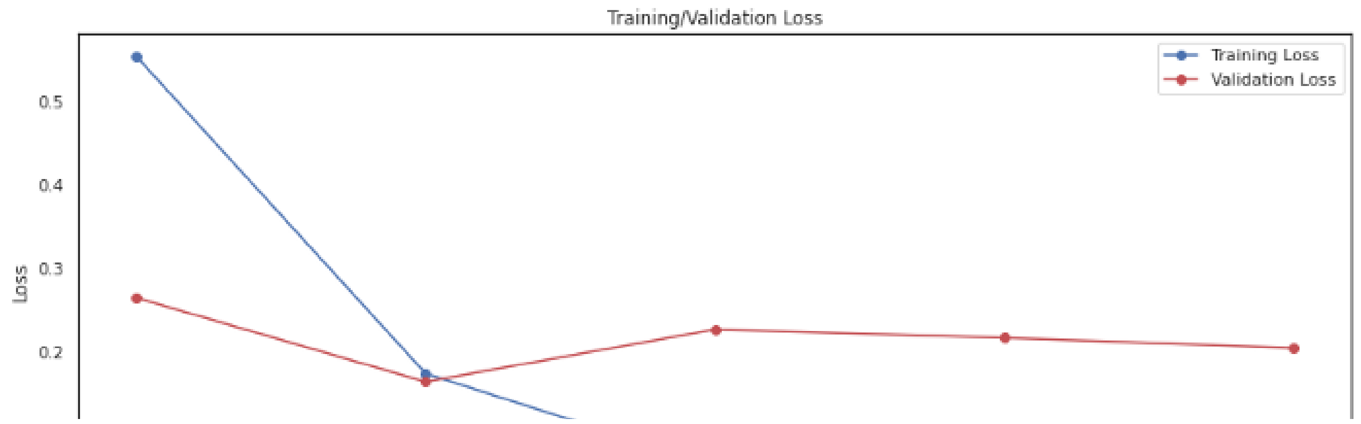
Model D performed quite well with more than 90% in both accuracy and f1 score in cross validation as well as final test. It correctly classify negative texts at 84% whereas positive texts at 94%, which could be due to the number of inputs but better to look closely to improve the performance further. This indicates the fine tuning of the pre-trained model performs well on this small data set thanks to transfer learning and the language model.



Confusion matrix on the final test set shows good performance, Image by Author



Loss for the training data reduced as the train proceeds, Image by Author



Loss for the validation data is not declining as does the training data, Image by Author

5. Conclusion

This experiment shows the potential of BERT based model application to my domain where previous models have failed to produce sufficient performance. The result is, however, not deterministic and based on a rather simple manual hyperparameter tuning on free-tier GPU and it could be different depending on the input data and tuning approaches.

It is also worth noting that in practical application, getting the proper input data is also quite important. The model cannot be trained well without data with good quality as often referred to as “garbage in, garbage out”.

I will cover these points next time. All the code used here are available in git repo.

Machine Learning NLP AI Sentiment Analysis Finance