

University of Washington, Tacoma

TCSS 555: Final Project

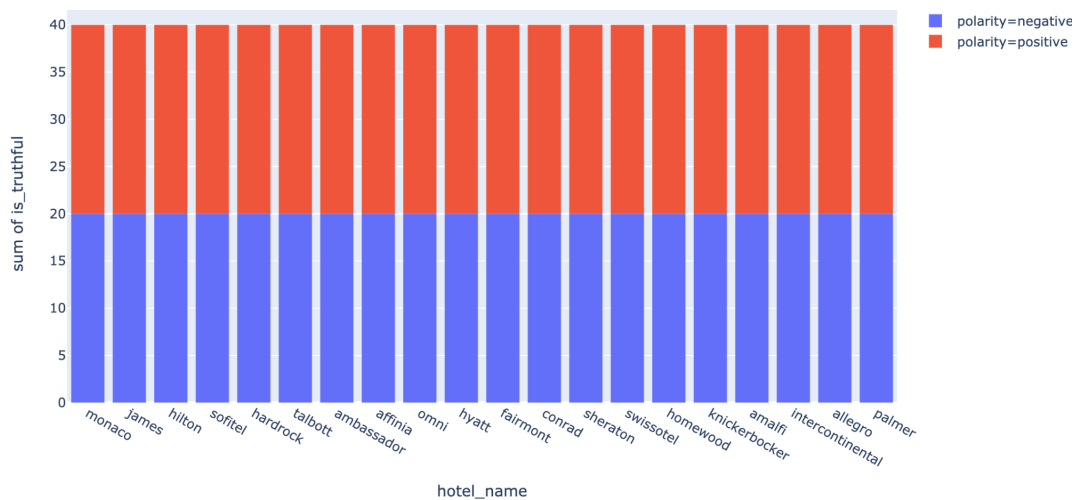
Opinion Spam Detection

Minzhi Qu, Sijin Huang, Guanchen Zhao

Data Analysis

Raw Data Analysis

First, we did raw data analysis with visualization tools called *Plotly Express*.



It shows that the data is evenly distributed in different hotels, and each hotel has an equal number of positive and negative, truthful and fake data.

Data Preprocess

Then we design some transformer to preprocess data:

1. *TextStemmer*: to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form.
2. *TextLemmatizer*: with the same function as *TextStemmer*, but the base form is human-readable.
3. *TextVectorizer*: create feature vector for each review based on most frequently occurring words

The following pictures show the execution of sample codes:

```

X = list(reviews['text'].copy())
y = list(reviews['is_truthful'].copy())

stemmer = TextStemmer()
X_stem = stemmer.fit_transform(X)

stem_df = pd.DataFrame({
    'text': X_stem,
    'is_truthful': y,
})

stem_df.head()

```

```

text is_truthful
0  when we got check and arriv at our room the fi... 0
1  the jame chicago is a stuffi uninvit hotel lf ... 0
2  We book a room at the hilton chicago for two n... 0
3  for a hotel rate with four diamond by aaa one ... 0
4  I wa veri disappoint with thi hotel the front ... 0

```

```

X = list(reviews['text'].copy())
y = list(reviews['is_truthful'].copy())

lemmatizer = TextLemmatizer()
X_lemma = lemmatizer.fit_transform(X)

lemma_df = pd.DataFrame({
    'text': X_lemma,
    'is_truthful': y,
})

lemma_df.head()

```

```

text is_truthful
0  get check arrive room first thing notice light... 0
1  jam chicago stuffy uninverting hotel look comfo... 0
2  book room hilton chicago two nights stay weeke... 0
3  hotel rat four diamonds aaa one would think hi... 0
4  disappoint hotel front desk clerk rude ; perso... 0

```

The result generated by stemmer is not human readable, so we choose to use *TextLemmatizer* to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form.

Then we construct a pipeline to preprocess raw data into feature vectors.

```

from sklearn.pipeline import Pipeline

X = list(reviews['text'].copy())
y = list(reviews['is_truthful'].copy())

preprocessPipeline = Pipeline([
    ("lemmatize text", TextLemmatizer()),
    ("text to feature", TextVectorizer(sizeOfVocabulary=100)),
])

most_common_voc, X_preprocessed = preprocessPipeline.fit_transform(X)

print(X_preprocessed.toarray())
print(most_common_voc)

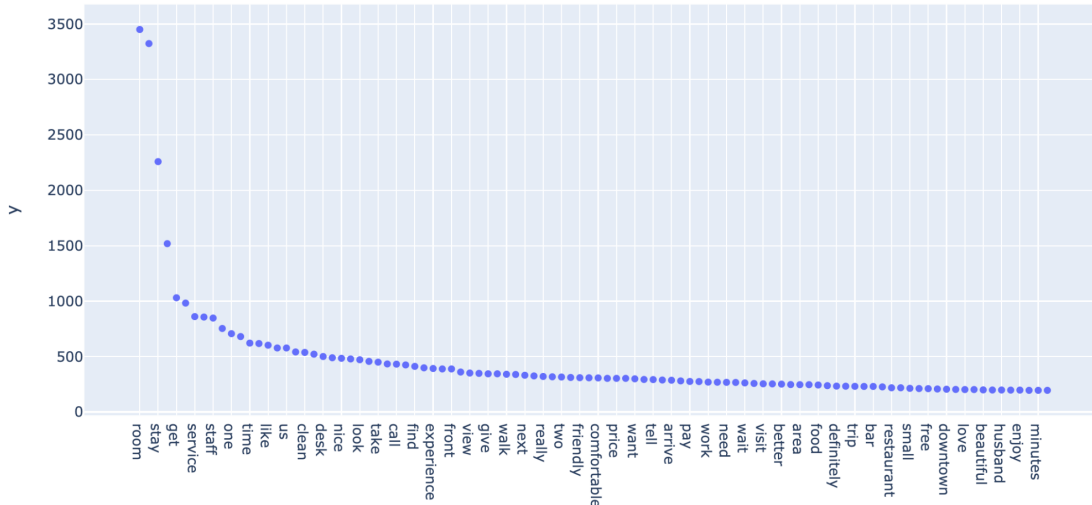
```

```

[[ 3  0  0 ...  0  0 30]
 [ 1  3  1 ...  0  0 19]
 [ 2  2  2 ...  0  0 37]
 ...
 [ 1  0  0 ...  0  0 13]
 [ 4  0  3 ...  0  0 47]
 [ 2  0  1 ...  0  0 35]]
[('room', 3451), ('hotel', 3324), ('stay', 2259), ('chicago', 1519), ('get', 1031), ('would', 983), ('service', 861),

```

The following picture shows the distribution of frequently occurring words.

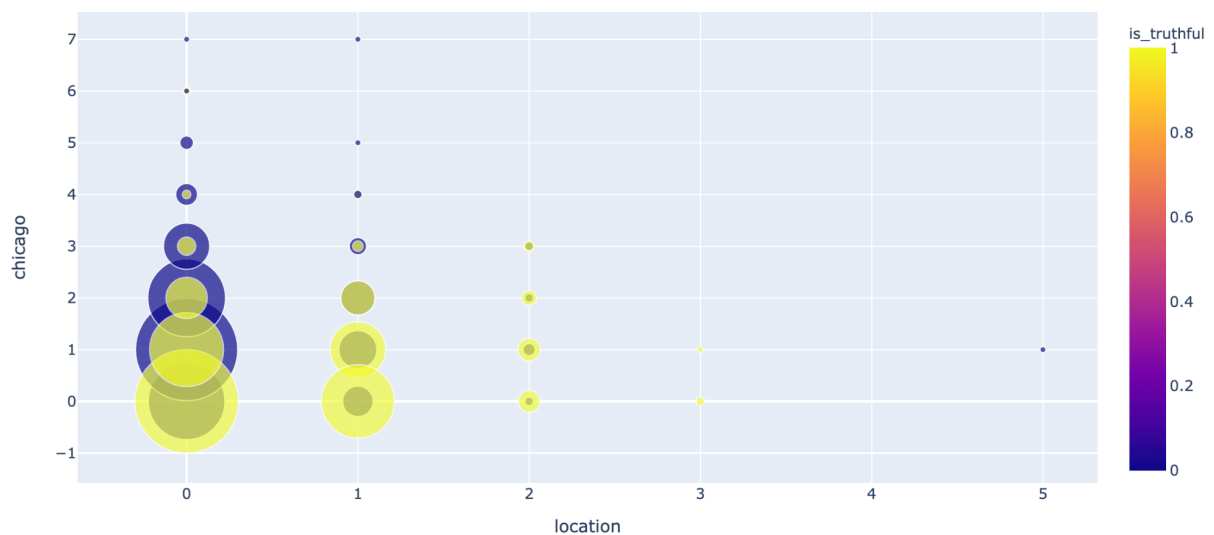


To get a better understanding of feature vectors, we calculate the correlation coefficient between feature vectors and labels.

```
[14] corr_matrix = data_df.corr()
      corr_matrix["is_truthful"].sort_values(ascending=False)

is_truthful    1.000000
location       0.246782
floor          0.190081
small          0.170599
great          0.156438
...
look           -0.121371
hotel          -0.130839
visit          -0.143806
experience     -0.146691
chicago      -0.337182
Name: is_truthful, Length: 102, dtype: float64
```

It is shown that some high frequent words have relatively high correlation with labels. We visualize two keywords 'location' and 'chicago' as shown in figure below:



It can be seen that it's hard to distinguish between truthful reviews from deceptive ones with only two vectors, even though the 1000 words vector has a relatively good performance to do that job.

Sentiment Analysis

We made a simple test which shows that the classification accuracy will be higher if all data belong to one sentiment category - positive and negative.

We use the same way to preprocess data with the same sentiment.

```
corr_matrix = data_df_pos.corr()  
corr_matrix["is_truthful"].sort_values(ascending=False)
```

```
is_truthful    1.000000  
location       0.283261  
floor          0.247632  
bathroom       0.219070  
michigan       0.216349  
...  
experience     -0.148978  
husband        -0.160058  
amaze          -0.181791  
visit          -0.193109  
chicago       -0.333119  
Name: is_truthful, Length: 102, dtype: float64
```

```
corr_matrix = data_df_neg.corr()  
corr_matrix["is_truthful"].sort_values(ascending=False)
```

```
is_truthful    1.000000  
location       0.211325  
great          0.170688  
floor          0.163444  
small          0.152139  
...  
finally        -0.165084  
expect         -0.169879  
seem           -0.189681  
smell          -0.238975  
chicago       -0.353530  
Name: is_truthful, Length: 102, dtype: float64
```

It shows that the most frequent occurring words are different in the dataset with different sentiment, and the correlation goes higher.

Therefore, we decided to design a sentiment classifier first.

We use the default sentiment classifier in *nlk*, the result is shown as follows:

```
[ ] # test for positive reviews
total = len(X_pos)
cnt_error = 0
for text in X_pos:
    if is_positive(text) is False:
        cnt_error += 1

print(f"Positive accuracy: {(total - cnt_error) / total:.2%}")

# test for negative reviews
cnt_error = 0
for text in X_neg:
    if is_positive(text) is True:
        cnt_error += 1

print(f"Negative accuracy: {(total - cnt_error) / total:.2%}")

Positive accuracy: 99.50%
Negative accuracy: 49.38%
```

Clearly, it is sensitive to positive reviews but less sensitive to negative. We are going to design a new sentiment classifier based on default mode.

We analyze the most frequent words for positive and negative dataset respectively,

```
[ ] print(top_positive_words)

{'flawless', 'award', 'alternative', 'definetly', 'pump', 'recomend', 'rex', 'dear', 'mag', '
[ ] print(top_negative_words)

{'mold', 'ignored', 'calls', 'deliver', 'refused', 'okay', 'standing', 'peeling', 'claimed',
```

We assume that a review is positive if it has more positive words than negative. Based on that assumption, we design a new feature as a 4-tuple (positive word counter, positive index, negative word counter, negative index).

Finally, we trained a **sentiment classifier** with random forest model, which has accuracy of 93.4%

```
[ ] cvres = grid_search.cv_results_
    for accuracy, params in zip(cvres['mean_test_score'], cvres['params']):
        print(accuracy, params)

0.9293750000000001 {'max_features': 1, 'n_estimators': 20}
0.9318750000000001 {'max_features': 1, 'n_estimators': 30}
0.93125 {'max_features': 1, 'n_estimators': 40}
0.9331250000000001 {'max_features': 1, 'n_estimators': 50}
0.9337499999999999 {'max_features': 2, 'n_estimators': 20}
0.930625 {'max_features': 2, 'n_estimators': 30}
0.928125 {'max_features': 2, 'n_estimators': 40}
0.928125 {'max_features': 2, 'n_estimators': 50}
0.928125 {'max_features': 3, 'n_estimators': 20}
0.9262499999999999 {'max_features': 3, 'n_estimators': 30}
0.9293750000000001 {'max_features': 3, 'n_estimators': 40}
0.9237500000000001 {'max_features': 3, 'n_estimators': 50}
0.9262499999999999 {'max_features': 4, 'n_estimators': 20}
0.921875 {'max_features': 4, 'n_estimators': 30}
0.9231250000000001 {'max_features': 4, 'n_estimators': 40}
0.9275 {'max_features': 4, 'n_estimators': 50}
```

To sum up, in the phase of preprocessing, we transform original text into feature vector. The features include the most frequent words and sentiment. And we also construct a sentiment classifier for distinguishing positive reviews from negative ones.

Model Setup and Training

Train-test split

We leveraged *sk-learn*'s built in *train_test_split* method to split review texts and deceptive labels into training set and validation set. 80% and 20% of the data are used in the training set and validation set, respectively.

```
from sklearn.model_selection import train_test_split
texts=df['text'].tolist()
labels=df['is_truthful'].tolist()
train_texts, val_texts, train_labels, val_labels = train_test_split(texts, labels, test_size=.2,
random_state=42)
```

Experiment settings

Train data, test data

DATA	SIZE
Train data	1280
Test data	320

Metrics

We evaluated the models mainly based on the metric of classification accuracy. The training and testing data are splitted previously, and different models would have different preprocessing and feature extraction pipelines.

Hardware used for training

We used Google Colab as the running environment for training 3 different models. The VM has the following hardware:

- 2-core Intel Xeon 2.2GHz CPU
- 13GB RAM
- 30GB HDD
- NVIDIA TESLA T4 GPU with 15GB GPU RAM

Proposed Models

We experimented with 3 different models: Logistic Regression, LSTM classifier and Bert classifier.

Logistic Regression

Logistic Regression is one of the traditional models for classification. For Logistic Regression, we fine tune different hyperparameters to improve performance of the classifier. Finally, we get the best accuracy of 84.68% with 'L2' penalty.

LSTM classifier

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture. For the LSTM classifier, we choose *tf.keras.layers.LSTM* to help us finish *LSTM* training. In terms of the clarity of the code and the convenience of model implementation, keras is really convenient. First, with the training data ready, we use *Tokenizer* to construct the input of *LSTM* model. Each batch will have a shape of (batch_size, sequence_length) because the padding is dynamic each batch will have a different length. Next create batches of these encoded strings. Use the *padded_batch* method to zero-pad the sequences to the length of the longest string in the batch. Then we can build the model, input the data and start training.

Bert classifier

For the Bert classifier, we take the last pooled hidden state as the representation of the whole review text. After the representation extraction, we stacked 2 fully connected layers with drop out and finally the softmax activation, which outputs the probabilities of 2 target classes. For training, we executed a transfer learning on the new data. By taking advantage of the pretrained model, the text representation could better capture the semantic features in the sentences.

Training and Fine Tuning

For logistic regression, we tried different sets of generated features. For LSTM and Bert based classifiers, we tried different network structures and optimizer hyperparameters.

Logistic Regression


```

GridSearchCV(cv=5, error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                           fit_intercept=True,
                                           intercept_scaling=1, l1_ratio=None,
                                           max_iter=100, multi_class='auto',
                                           n_jobs=None, penalty='l2',
                                           random_state=None, solver='lbfgs',
                                           tol=0.0001, verbose=0,
                                           warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'penalty': ['l1', 'l2', 'elasticnet']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)

```

```

[ ] X_train, X_test, y_train, y_test = train_test_split(X_preprocessed, y, test_size=0.2, random_state=40)
log_clf = LogisticRegression(solver="liblinear", penalty='l2', random_state=40)
score = cross_val_score(log_clf, X_train, y_train, cv=4)
print('Cross Validation Score: ', score.mean())

```

Cross Validation Score: 0.846875

We fine tune different hyperparameters to improve performance of the LR classifier. We got the best accuracy 84.68%.

LSTM classifier

After **Fit_on_texts**, **texts_to_sequences**, **pad_sequences**, we get the training data set **x_dataset**, as shown in the figure below.

```

array([[ 0,  0,  0, ..., 81, 1716,  2],
       [ 0,  0,  0, ..., 52,   5, 61],
       [ 0,  0,  0, ..., 233,  17, 72],
       ...,
       [ 0,  0,  0, ..., 78,  114,  2],
       [ 0,  0,  0, ..., 844,  78,  3],
       [ 0,  0,  0, ..., 36,   3,  4]], dtype=int32)

```

According to the keras document, we adjust the parameters and get the LSTM model. The details are as follows.

Model: "sequential_10"

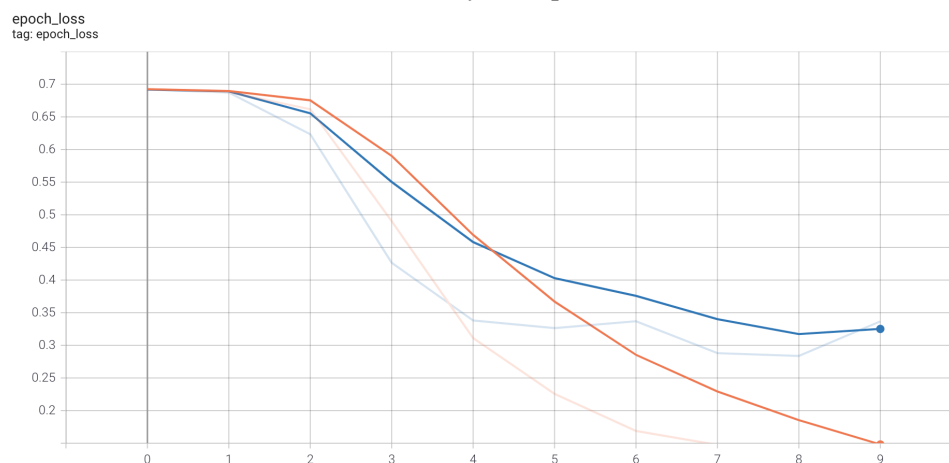
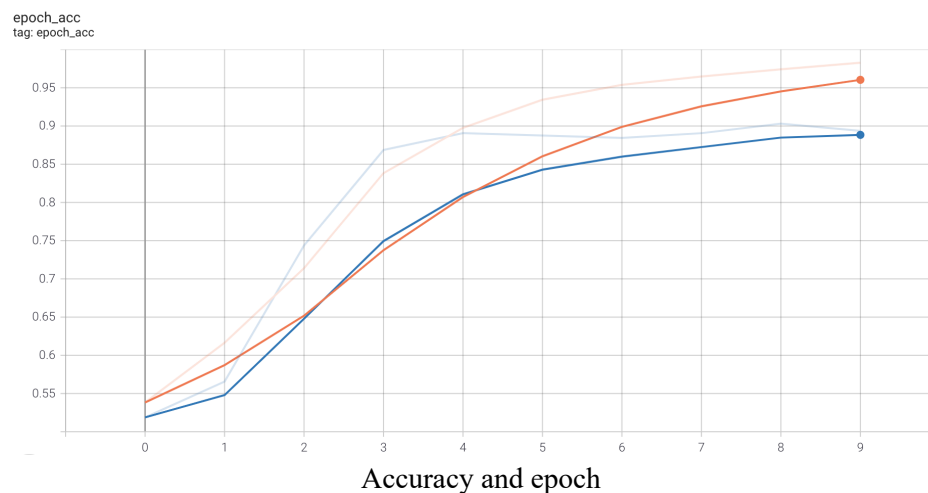
Layer (type)	Output Shape	Param #
embedding_10 (Embedding)	(None, None, 200)	400000
module_wrapper_30 (ModuleWra	(None, 64)	67840
module_wrapper_31 (ModuleWra	(None, 64)	4160
module_wrapper_32 (ModuleWra	(None, 1)	65
Total params: 472,065		
Trainable params: 472,065		
Non-trainable params: 0		

Then we fit the model with 10 epochs, this is the output of each epoch.

We can find that the accuracy rate is gradually rising, and finally close to the 88% level.

```
64/64 [=====] - 26s 192ms/step - loss: 0.6930 - acc: 0.5023 - val_loss: 0.6923 - val_acc: 0.5375
Epoch 2/10
64/64 [=====] - 11s 170ms/step - loss: 0.6898 - acc: 0.6356 - val_loss: 0.6878 - val_acc: 0.6250
Epoch 3/10
64/64 [=====] - 11s 170ms/step - loss: 0.6776 - acc: 0.7424 - val_loss: 0.6330 - val_acc: 0.7500
Epoch 4/10
64/64 [=====] - 11s 176ms/step - loss: 0.5430 - acc: 0.8503 - val_loss: 0.3607 - val_acc: 0.8875
Epoch 5/10
64/64 [=====] - 11s 176ms/step - loss: 0.3237 - acc: 0.8975 - val_loss: 0.3244 - val_acc: 0.8938
Epoch 6/10
64/64 [=====] - 11s 175ms/step - loss: 0.2062 - acc: 0.9432 - val_loss: 0.2943 - val_acc: 0.8938
Epoch 7/10
64/64 [=====] - 11s 172ms/step - loss: 0.1910 - acc: 0.9427 - val_loss: 0.3083 - val_acc: 0.9000
Epoch 8/10
64/64 [=====] - 11s 177ms/step - loss: 0.1565 - acc: 0.9513 - val_loss: 0.2887 - val_acc: 0.9094
Epoch 9/10
64/64 [=====] - 11s 174ms/step - loss: 0.0897 - acc: 0.9788 - val_loss: 0.3200 - val_acc: 0.8813
Epoch 10/10
64/64 [=====] - 11s 173ms/step - loss: 0.0730 - acc: 0.9819 - val_loss: 0.3259 - val_acc: 0.9062
```

Finally, with the help of *tensorboard*, we get the accurate change image.



With the increase of epoch, the accuracy of training set and validation set is increasing, and the loss is decreasing.

Bert classifier


```


▶ from torch.utils.data import DataLoader
  from transformers import DistilBertForSequenceClassification, AdamW

  device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

  model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased')
  model.to(device)

```

Downloading: 100%  442/442 [10:25<00:00, 1.42s/B]

Downloading: 100%  268M/268M [00:06<00:00, 42.6MB/s]

```

  Training accuracy: 100.0
  Loss: 0.8125727058200027
  validation accuracy: 90.3125
  Loss: 0.0009199929058922862
  Training accuracy: 100.0
  Loss: 0.7896947193644337
  validation accuracy: 91.25
  Loss: 1.0187692202621292
  Training accuracy: 93.828125
  Loss: 1.8158201947103407
  validation accuracy: 85.0
  Loss: 0.31303251200006343
  Training accuracy: 97.96875
  Loss: 0.7742305390129331
  validation accuracy: 90.625

```

The final results of 3 models are shown in the following table.

Model	Training Set	Validation Set
LR	90.31%	84.68%
LSTM	95.58%	88.28%
Bert	99.77%	91.56%

The time consumed for training is shown in the following table:

Model	Training Time
LR	0.1min
LSTM	2.5min
Bert	17min

Model Performance Comparison

Surprisingly, the LR model achieves an acceptable accuracy of 84.68%. It seems like the lexical and semantic and syntax features are impressively effective for spam detection. The time cost for training LR is also the shortest.

Also, both LSTM and Bert could achieve an accuracy around 90% without needing heavily extracting human designed features with expert-knowledge support.

The Bert has a slightly better performance than LSTM, which demonstrates the better semantic comprehension capability than LSTM. With more parameters and cross-domain linguistic knowledge learnt from pretraining on a huge corpus, it is not surprising that Bert could perform better than LSTM. But the large number of parameters leads to a much longer training time than the other two models.

Comparison with related research

Kennedy et al. reported several non-neural and neural models' performances on OpSpam dataset. Our results demonstrate a match to the reported performance.

Kennedy, Stefan, et al. "Fact or factitious? Contextualized opinion spam detection." *arXiv preprint arXiv:2010.15296* (2020).

Model	Performance(Paper)	Performance(Ours)
LR	85.6%	84.68%
LSTM	87.6%	88.28%
Bert	89.1%	91.56%

Our implemented models are competitive with the state-of-the-art performance of OpSpam dataset. The comparison demonstrates the effectiveness of our designed features and the correctness of our modeling.

Conclusion

By implementing data analysis, data cleaning, preprocessing, feature extraction and modeling, we successfully trained spam detection models with a compatible performance with SOTA results. Our results demonstrate the effectiveness of machine learning on finding linguistic patterns and ability to contribute to the human commercial society.

Individual Contribution

Minzhi Qu: Raw data analysis, data cleaning, sentiment analysis and feature extraction

Sijin Huang: Data reorganizing, data analysis, Bert Model establishment and training

Guanchen Zhao: Data analysis, Potential Models, LSTM Model training