Using Deep Learning to Predict Clickbait: The Results Will SHOCK You

Anupama Warkhandkar

Cory Boyko

Shubham Sharma

Ouloide Yannick Goue

Harrisburg University of Science and Technology

Author Note

Correspondence concerning this article should be addressed to Anupama Warkhandkar at awarkhandkar@my.harrisburgu.edu, Cory Boyko at cboyko@my.harrisburgu.edu, Shubham Sharma at ssharma2@my.harrisburgu.edu, Ouloide Goue at OGoue@my.harrisburgu.edu

Abstract

This research shows that deep neural networks can outperform classic machine learning algorithms for predicting whether an article is clickbait or not just from the article title. Clickbait is a common form of an article where the title uses highly polar and misleading language in order to get more people to click on the article. The articles then provide little relevant information or information that is much less dire than the title led one to believe. Previous research has used classic machine learning algorithms like support vector machines, decision trees, and random forests. Through the use of neural net architectures like dense networks, convolutional neural networks, and recurrent neural networks along with GloVe embeddings, models can have an improved accuracy. Simple RNN, LSTM, CNN and LSTM, and Bidirectional and GRU models can each produce a better accuracy than past research. Simple DNN does not produce a better accuracy.

Keywords: Neural Network, NLP, deep learning, clickbait, RNN

Using Deep Learning to Predict Clickbait: The Results Will SHOCK You

Clickbait is an increasingly more common type of article where the title makes it look like the article contains something incredibly interesting or information that the reader needs to know. The title is misleading, because the article itself is about either very general information, speculative, or is nowhere near as consequential as the title made it sound. An example of this would be if the title said, “You won’t believe what X said about her co-star!!!” and then the article will be about a simple interview where they said the costar was great to work with. The title made it sound scandalous, but the article was just about a generic statement.

These articles are commonly used to get website visitors to click on the article which can increase the site’s internal metrics or may often lead to increased ad revenue. They exist in all types of articles from sports to entertainment to genuine news. Though possibly frustrating, clickbait does have its pros and cons. Clickbait articles do often provide insights occasionally about their topics that are still of interest to readers. They also help make the article more noticeable by having an interesting title. From a company standpoint, they are quite useful as they do bring more traffic to their site which may separate them from competitors. The increased ad revenue helps as well. The cons are that people can become annoyed with the articles as they felt they were misled or lied to. This can lead to people becoming annoyed with the brand or distrustful of the site. That can lead to less people returning to the site which would harm their metrics and ad revenue.

The goal of this paper was to classify articles as clickbait based just on their title. To do this, deep learning was specifically utilized as it doesn’t need as much pre-processing and has been shown to have better performance in other areas. Deep learning has been shown to excel in text cases as well. Though clickbait is not

necessarily bad, it is still important to let the reader know, so their expectations can be properly managed.

Literature Review

The past work where this research came from was called “Stop Clickbait: Detecting and Preventing Clickbait in Online News Media” by Abhijnan Chakraborty, Bhargavi Paranjape, Sourya Kakarla, and Niloy Ganguly. It was published in 2016 for the International Conference on Advances in Social Networks Analysis and Mining (ASONAM). The article titles that they used (and were subsequently used for this paper) came from Wikinews, BuzzFeed, Upworthy, ViralNova, Scoopwhoop, and ViralStories. From there, each of the articles’ titles were then checked by a group of 6 volunteers to confirm whether they should have been classified as clickbait or not. The final set that they used consisted of 15,000 titles with 7,500 in each category. The models naturally needed numeric variables, so they used the following features: title length, length of words, length of syntactic dependencies, stop words, hyperbolic words, slang, punctuation, common clickbait phrases, subject of article, determiners, parts of speech, N-grams, and syntactic N-grams. To find the best model for predicting clickbait, they tested SVM (with radial basis function as the kernel), decision trees, and random forests. They used the accuracy from a previously made browser extension called “Downworthy”, which claimed to accurately classify clickbait 76% of the time, as their baseline accuracy. The best model with an accuracy of 93% was the SVM model.

Dataset Description

The dataset consisted of a total of 32,000 article titles. There were 16,001 non-clickbait articles and 15,999 clickbait articles. Accompanying each article is a binary flag of 1 or 0 denoting whether they were a clickbait article (1) or not (0). No other variables were in the files leaving all analysis to be based off of just the article titles. The titles themselves vary in length and style in an attempt to emulate all possible ways clickbait may exist on the internet.

Methods

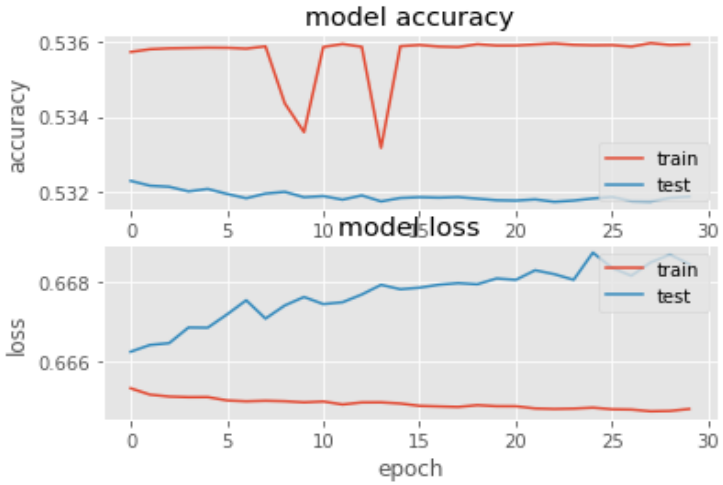
To improve on previous research, we chose to use deep learning to create the models. We tested a variety of types of models to find the best one. These included a simple DNN (only with dense layers), CNN, and a variety of recurrent neural networks such as simple RNN, LSTM, bidirectional, and GRU. Other layers and aspects were used such as dropout and early stopping. Previous research used various types of feature extraction, but for this research only word embeddings were used. Instead of attempting to create a new set of word embeddings or using something simple like a document-term matrix, GloVe pre-trained embeddings, created by Stanford, were used. The embeddings that were used were from the Wikipedia + Gigaword 5 set that was trained on 6 billion texts, consisted of 400,000 tokens, and transformed responses into 50 dimensions. Using pre-trained models often leads to increased performance as they can better represent relationships between similar texts. For the embeddings, only the 10,000 most common tokens were used. The data was then separated to create a 70-30 split for train and test.

Results

Simple DNN

A simple DNN model was built with GloVe embedding of 50 dimensions and the hidden layers used relu activation. The model was trained with a batch size of 128, 30 epochs, and Adam optimizer. 20% of the data was used as validation during training. Because of the binary nature of the dataset, “binary_crossentropy” was used as the loss function and “accuracy” as the metrics. An accuracy of 52.39% was observed.

Below learning curves display how accuracy and loss changed for training and test data sets, as the model execution progressed from zero to 30th epoch. Since the plot for training accuracy is above test accuracy, the model has overfitting issues. Also, model loss is on an increasing trend as compared to training dataset. Even though building and testing simple DNN model was less complicated, it looks like there is scope for a better model with improved accuracy.



Simple RNN

Following the simple DNN model, a simple RNN model with 32 layers was built. The model used the GloVe embedding with 50 dimensions. The model was trained with a batch size of 128, 50 epochs, and Adam optimizer with 20% of the data used as validation during

training. Because of the binary nature of the dataset, “binary_crossentropy” was used as the loss function and “accuracy” as the metrics. It follows from the evaluation of the model an improved accuracy compared to the DNN model. In fact, a 97.14% accuracy was observed for the validation data.

In order to check for model overfit or underfit, two learning curves were plotted: loss and accuracy. These are shown in the figure below. It appears from the graph that the model moderately overfit. It is also observed that after between 10 and 20 epochs, not much has changed in the curves. Thus, it could be suggested that the model could have been stopped after about 15 epochs.

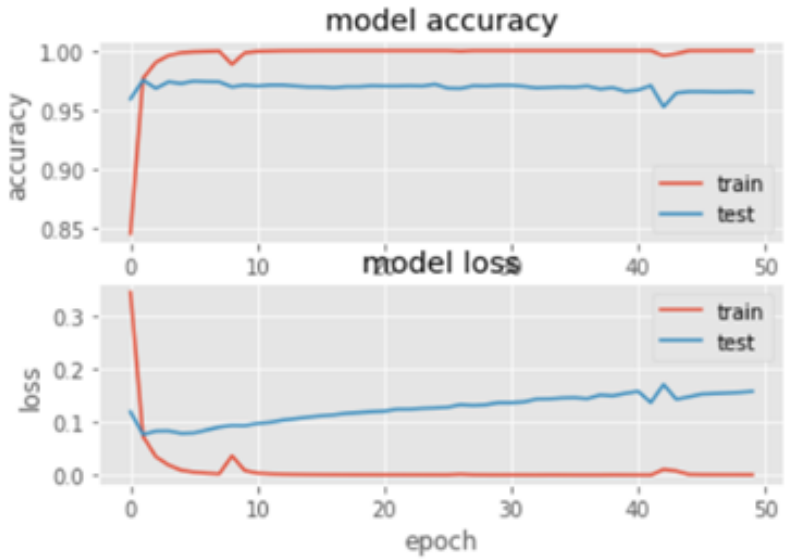


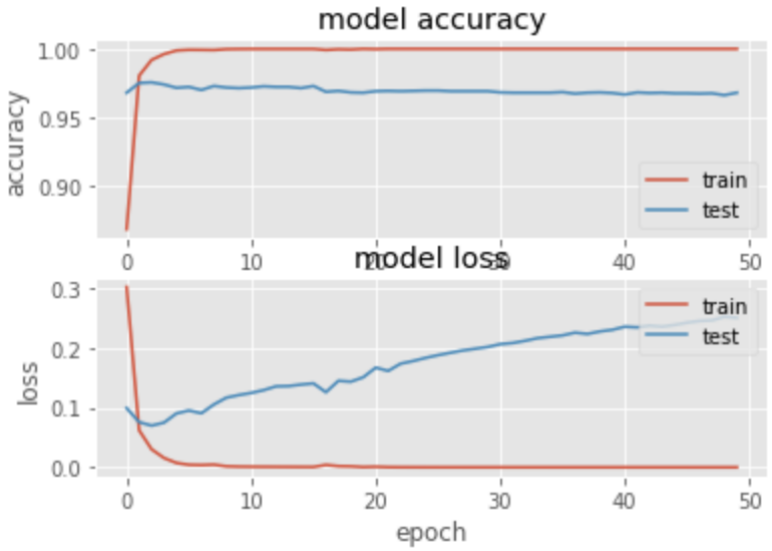
Figure: Learning curves for model loss and model accuracy

LSTM

LSTM stands for Long-Short Term Memory. LSTM models are a special kind of Recurrent Neural Networks (RNNs) used in deep learning. In the LSTM model we built for our project, we have used GloVe embeddings with 50 dimensions. In this model, we have used an LSTM layer with 32 units. While training this model, we have implemented a batch size of 128 units and ‘adam’ optimizer. We ran this model for a total of 50 epochs.

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 50)          500000
_________________________________________________________________
lstm (LSTM)                  (None, 32)                10624
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 33
=================================================================
Total params: 510,657
Trainable params: 510,657
Non-trainable params: 0
_________________________________________________________________
```

After training the above model, we get an accuracy of 97.23% on testing data. The model starts to overfit after reaching 12 epochs and this is also evident in the figure plotted for accuracy vs number of epochs to check the learning rate of the model. We can also check the loss value of the model in the plot below. We get a loss of 0.142 while we train the model for 50 epochs. We can also see that the loss value increases with the number of epochs on test data while on training dataset, the value of loss approaches 0 within the first 10 epochs.

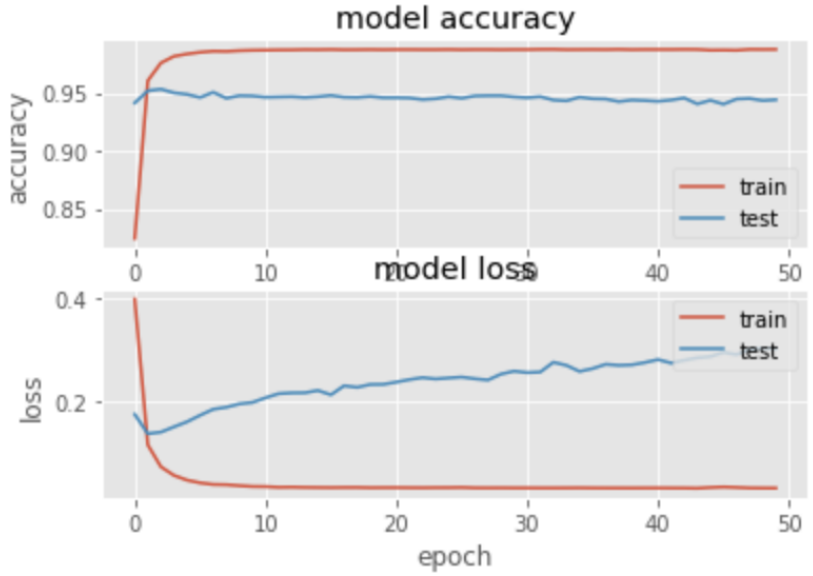


CNN + LSTM

In the next part of the project, we have combined CNN and LSTM in the same model and trained the data using this model. CNN stands for Convolutional Neural Networks and LSTM, as we discussed above, stands for Long-Short Term Memory. While we combine the two models here, CNN is used to extract the features from the dataset and LSTM is used to learn long-term dependencies.

In the CNN+LSTM model we built, we have used GloVe embeddings data with 50 dimensions. Here we use a LSTM layer with 32 units and a 1-dimensional convolutional layer using a 'relu' activation method. Also, we have used a batch size of 128 units and 'adam' optimizer and ran the model for 50 epochs.

After training the model we built above, we get an accuracy of 93.91%. The model starts to overfit after 15 epochs and we can use a dropout model to avoid the problem of overfitting. The loss in the model we built is 0.3. Also, the loss value keeps increasing as we increase the number of epochs. The advantage in building a model like this is that the complexity level is not very high and follows the same structure as other ones. It also helps in combining the advantages of both CNN and LSTM on the dataset.



Bidirectional + GRU

The final model that was tested consisted of bidirectional GRU layers. Bidirectional layers were used since text does not have an ordered sequence and features can exist in any part of the title. GRU layers were tested since they are similar to LSTM, though considered less complex, and often can outperform LSTM. The final model consisted of 3 GRU layers each with ReLu activation. The first layer had 30 nodes, the second had 50 nodes, and the third had 100 nodes. Each of the layers had 10% dropout and 50% recurrent dropout. The model was trained with a batch size of 1000 (due to training time) and started with 100 epochs, but due to early stopping with patience set to 10, converged at 48 epochs. 20% of the data was used as validation while training. The final model had an overall accuracy of 97.80% for the training data, 97.28% on the validation data, and 97.30% on the test data. The test data also had a true positive rate of 97.19% and a true negative rate of 97.39%.

Discussion

Overall, the best model was the model with bidirectional and GRU layers. This matches previous research as bidirectional and GRU are known to perform very well for text data. The simple RNN and LSTM model also performed very well with similar accuracies to the GRU model. The CNN + LSTM model also outperformed the accuracy research, though just barely. The simple DNN was the only model that had a poor accuracy, though this is consistent with previous understanding as dense layers don't perform well with dense layers alone.

Each of the models also had pros and cons. The simple DNN had a very simple model with a quick train time, but with a very poor accuracy that was close to a 50-50 guess. The simple RNN had a high accuracy and the lowest complexity of the RNNs tested, but had a long train time. The LSTM model also had a high accuracy, but had a long train time and was very easy to overfit. The CNN and LSTM model provided a good accuracy and combined the strengths of RNN and convolutional layers, but the model had a long train time and had a lower accuracy, showing that the combination of layers actually harmed the model as LSTM alone produced a better model. The bidirectional and GRU model had the best accuracy, but took a very long train time even with using a GPU.

Most importantly, this research showed that neural networks can outperform classic machine learning models with significantly less work. The past research needed to extract many features from the data whereas this research shows that with pre-trained embeddings and just a few layers, the same or better accuracy can be achieved. In summary, a model with any recurrent neural network architecture is better than other machine learning models.

Future Work

Previous Research used their clickbait models in practice by deploying them as a browser plug in. It would be interesting to see how these models would perform in a real time setting. The models have high accuracies, but also have high run times for classification. The

run times may be a deterior to people using them even with the high accuracy. In those cases the different models should be tested to find the best tradeoff of accuracy and run time.

The dataset had a large amount of article titles, but the non-clickbait titles came from a single news source. The clickbait titles came from a variety of sources, but the sources were mostly lesser known and unheard of websites. A lot of more commonly known sites also use clickbait, so it'd be interesting to test the models on articles from those sites. Sites with more resources are possibly better at "hiding" their clickbait articles by making them blend in with their regular articles.

Models such as these can also possibly be used as a basis to classify other types of articles such as those with high bias. Clickbait are the extreme example of articles that are misleading, but in today's world outlets are starting to use the same high polarity language to get people to click on their articles. These articles provide real information, but are often biased and use deceptive language to click on them. The architectures used here can be used as a basis for those models.

References

A. Chakraborty, B. Paranjape, S. Kakarla and N. Ganguly, "Stop Clickbait: Detecting and preventing clickbaits in online news media," 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), 2016, pp. 9-16, doi: 10.1109/ASONAM.2016.7752207.

Pennington, J. (n.d.). *GloVe: Global Vectors for Word Representation*. GloVe: Global Vectors for Word Representation. Retrieved June 17, 2021, from https://nlp.stanford.edu/projects/glove/

Appendix

Sample Code 1. Preprocessing Code Used by All

```
from google.colab import drive
drive.mount('/content/gdrive')
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
import os
import pandas as pd
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense
import matplotlib.pyplot as plt
from keras import models
from keras.layers import SimpleRNN
from keras.layers import LSTM

data = pd.read_csv('/content/gdrive/MyDrive/ML2 Project/Clickbait and
Nonclickbait Data.csv')

data2 = data.values
texts = data2[:, 0]
labels = data2[:, 1]
labels = labels.astype(str).astype('int32')
max_features = 10000
maxlen = 100
training_samples = 22400
testing_samples = 9600
max_words = 10000

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
data = pad_sequences(sequences, maxlen=maxlen)
labels = np.asarray(labels)
```

```
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]
x_train = data[:training_samples]
y_train = labels[:training_samples]
x_test = data[training_samples: training_samples + testing_samples]
y_test = labels[training_samples: training_samples + testing_samples]

glove_dir = '/content/gdrive/MyDrive/ML2 Project'
embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.50d.txt'), encoding="utf8")
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Found %s word vectors.' % len(embeddings_index))

embedding_dim = 50
embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

Sample Code 2. Final GRU Model

```
model11 = Sequential()
model11.add(Embedding(max_features, 50))
model11.add(layers.GRU(30,
dropout=0.1,recurrent_dropout=0.5,return_sequences=True))
model11.add(layers.GRU(50,
activation='relu',dropout=0.1,recurrent_dropout=0.5,
return_sequences=True))
model11.add(layers.GRU(100,
activation='relu',dropout=0.1,recurrent_dropout=0.5))
```

```
model11.add(Dense(1, activation='sigmoid'))
model11.summary()

from keras.callbacks import ModelCheckpoint
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss',
patience= 10)
callbacks_list=
ModelCheckpoint('Models/Weights-{epoch:03d}--{val_loss:.5f}.hdf5',
monitor='val_loss', save_best_only = True)
callbacks = [early_stop, callbacks_list]


model11.layers[0].set_weights([embedding_matrix])
model11.layers[0].trainable = False
model11.compile(optimizer='adam',loss='binary_crossentropy',
metrics=['acc'])
history11 = model11.fit(x_train,
y_train,epochs=100,batch_size=1000,validation_split=0.2, callbacks =
callbacks)

model11.evaluate(x_test, y_test, verbose = True)
predictions2 = model11.predict_classes(x_test)
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test, predictions2))
```

Sample Code 3: Long-Short Term Memory (LSTM)

```
model4 = Sequential()
model4.add(layers.Embedding(max_features, 50))
model4.add(LSTM(32))
model4.add(Dense(1, activation='sigmoid'))
model4.summary()

model4.compile(optimizer='adam',loss='binary_crossentropy',
metrics=['acc'])
history4 = model4.fit(x_train,
y_train,epochs=50,batch_size=128,validation_split=0.2)
```

Sample Code 4: Convolutional Neural Network (CNN) + Long-Short Term Memory (LSTM)

```
model6 = Sequential()
model6.add(layers.Embedding(max_features, 50, input_length=maxlen))
model6.add(layers.Conv1D(32, 7, activation='relu'))
model6.add(layers.MaxPooling1D(5))
model6.add(LSTM(32))
model6.add(Dense(1, activation='sigmoid'))

model6.compile(optimizer='adam',loss='binary_crossentropy',
metrics=['acc'])
history6 = model6.fit(x_train,
y_train,epochs=50,batch_size=128,validation_split=0.2)
```