General Report on Improvised 15- Models
Team Fake

Github repo: https://github.com/Jothish2283/Debunkathon

Datasets:
https://zenodo.org/record/4561253/files/WELFake_Dataset.csv?download=1

The dataset consists of 3 columns ["title", "text", "label"]
And the dataset is well-balanced. Not much preprocessing is needed.

```
[14] news_dataset.head()
```

	Unnamed: 0	title	text	label
0	0	LAW ENFORCEMENT ON HIGH ALERT Following Threat...	No comment is expected from Barack Obama Membe...	1
1	1		Did they post their votes for Hillary already?	1
2	2	UNBELIEVABLE! OBAMA'S ATTORNEY GENERAL SAYS MO...	Now, most of the demonstrators gathered last ...	1
3	3	Bobby Jindal, raised Hindu, uses story of Chri...	A dozen politically active pastors came here f...	0
4	4	SATAN 2: Russia unvelis an image of its terrif...	The RS-28 Sarmat missile, dubbed Satan 2, will...	1

```
[15] news_dataset.label.value_counts() #balanced dataset

1    37106
0    35028
Name: label, dtype: int64
```

Preprocessing the data

Using nltk package for filtering stopwords, and stemming.

nltk(filtering stopwords, stemming)_explained:
https://realpython.com/nltk-nlp-python/#stemming

Stemming: It is the process of reducing a word to its root-word
example: actor, actress, acting-->act(root-word)

```
[9] port_stem = PorterStemmer()
```

```
cachedStopWords = stopwords.words('english')
```

```
[11] def stemming(content):
         stemmed_content = re.sub('[^a-zA-Z]',' ',content)# re :searching paragraph or text, sub :substitute
         stemmed_content = stemmed_content.lower()# converting all letters to lower case
         stemmed_content = stemmed_content.split()# splitting to converted to list
         stemmed_content = [port_stem.stem(word) for word in stemmed_content if not word in cachedStopWords]
         stemmed_content = ' '.join(stemmed_content)# joining the words
         return stemmed_content
```

Vectorizer: For converting the textual data to numerical data

- We use TfidfVectorizer [term frequency-inverse documentary frequency vectorizer]
- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

```
# converting the textual data t
vectorizer = TfidfVectorizer()#
vectorizer.fit(X)

X = vectorizer.transform(X)
```

ML_Models:

1. **LR [Linear Regression]**
    1.1. Model Architecture

```
model_lr = LogisticRegression()
model_lr.fit(X_train,Y_train)
```

    1.2. Results:

```
{'Accuracy': 0.9486379704720316,
 'Precision': 0.943088352348103,
 'Recall': 0.9579571486322598,
 'f1_score': 0.9504646032488802}
```

    1.3. External Link
    LR_explained:https://towardsdatascience.com/linear-regression-explained-1b36f97b7572

2. **KNN [K-Nearest Neighbors]**
    2.1. Model Architecture:

```
model_knn = KNeighborsClassifier()
model_knn.fit(X_train,Y_train)
```

    2.2. Results:

```
{'Accuracy': 0.6894711305191654,
 'Precision': 0.6302595446895207,
 'Recall': 0.9587656650047164,
 'f1_score': 0.760555852485302}
```

    2.3. External Links
    KNN_Explained:https://towardsdatascience.com/k-nearest-neighbors-knn-explained-cbc31849a7e3

3. **SVM [Simple Vector Machine]**

3.1. Model Architecture:

```
model_svm = svm.SVC(kernel='linear')
model_svm.fit(X_train,Y_train)
```

3.2. Results:

```
{'Accuracy': 0.9591737714008456,
 'Precision': 0.9569288389513109,
 'Recall': 0.9640210214256839,
 'f1_score': 0.9604618379539505}
```

3.3. External Links

SVM_Explained:https://towardsdatascience.com/support-vector-machine-simply-explained-fee28eba5496

4. **DT [Decision Trees]**

4.1. Model Architecture:

```
model_dt= tree.DecisionTreeClassifier(max_depth = 64)
model_dt.fit(X_train, Y_train)
```

4.2. Results:

```
{'Accuracy': 0.9373397102654745,
 'Precision': 0.9268963710205685,
 'Recall': 0.9533755558550061,
 'f1_score': 0.9399495150790488}
```

4.3. External Links

DT_Explained:https://towardsdatascience.com/decision-trees-explained-3ec41632ceb6

5. XGBOOST

5.1. Model Architecture:

```
model_xgboost = XGBClassifier()
model_xgboost.fit(X_train, Y_train)
```

5.2. Results:

```
{'Accuracy': 0.9412213211339849,
 'Precision': 0.9270955165692008,
 'Recall': 0.9613259668508287,
 'f1_score': 0.9439005027785128}
```

5.3. External Links

XGBOOST_Explained:https://towardsdatascience.com/https-medium-com-vishalmorde-xgboost-algorithm-long-she-may-rein-edd9f99be63d

6. LGBM [LightGBM]

6.1. Model Architecture:

```
model_lgbm = LGBMClassifier()
model_lgbm.fit(X_train, Y_train)
```

6.2. Results:

```
{'Accuracy': 0.9645109863450475,
 'Precision': 0.956039603960396,
 'Recall': 0.9758792615550465,
 'f1_score': 0.9658575620165377}
```

6.3. External Links

LGBM_Explained:https://towardsdatascience.com/what-makes-lightgbm-lightning-fast-a27cf0d9785e

7. RF [RANDOM FOREST]

7.1. Model Architecture:

```
model_rf = RandomForestClassifier(n_estimators= 10, criterion="entropy")
model_rf.fit(X_train,Y_train)
```

7.2. Results:

```
{'Accuracy': 0.901296180772163З,
 'Precision': 0.9153622385371936,
 'Recall': 0.8904460315321385,
 'f1_score': 0.9027322404371585}
```

7.3. External Links

RF_Explained:https://towardsdatascience.com/understanding-random-forest-58381e0602d2

8.Ensemble

8.1 Model Architecture:

```
def ensemble(predictions):
  return np.mean(predictions, axis=0)
```

```
ensemble_test_prediction = ensemble(predictions)
```

Here we are calculating the mean of the above all ML algorithms and rounding it as 1 if the mean is greater than or equal to 0.5 or else 0.

8.2 Results:

```
{'Accuracy': 0.9628474388299716,
 'Precision': 0.9484173505275498,
 'Recall': 0.981134617976014,
 'f1_score': 0.9644986090872962}
```

8.3 External Links

Ensemble_Explained:https://towardsdatascience.com/ensemble-method

DL_Models:

1. MLP [Multi-Layer Perceptron]

1.1. Model Architecture:

```
model_1.summary()

Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1)]               0

 text_vectorization (TextVec  (None, 1994)             0
 torization)

 embedding (Embedding)       (None, 1994, 128)         2560000

 dense (Dense)               (None, 1994, 64)          8256

 global_max_pooling1d (Globa  (None, 64)               0
 lMaxPooling1D)

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 2,570,369
Trainable params: 2,570,369
Non-trainable params: 0
_________________________________________________________________
```

1.2. Text_vectorization:

It is a layer created to tokenize/vectorize the words in the data. Ie, It will represent each word as a number since a machine can only understand numbers and not letters or words.

1.3. Embedding:

It represents how a word is learned/interpreted by the computer by looking at the examples. It is initially randomly initialized but with training, these values get updated and the computer better understands the meaning and the use of the word.

1.4. Global-max Pooling:

It is used to bring down the dimensionality of the intermediate data.
*here the extra dimension was created due to the embedding layer.

1.5. Results:

```
[26] model_1_results

{'Accuracy': 0.9759343462348896,
 'Precision': 0.9716819833297713,
 'Recall': 0.981753400993306,
 'f1_score': 0.9766917293233084}
```

2. CNN [Convelutional Neural Network]

2.1. Model Architecture

```
[40] model_2.summary()

Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 1)]               0

 text_vectorization (TextVec  (None, 1994)             0
 torization)

 embedding_3 (Embedding)     (None, 1994, 128)         2560000

 conv1d_2 (Conv1D)           (None, 1992, 64)          24640

 conv1d_3 (Conv1D)           (None, 1990, 32)          6176

 global_max_pooling1d_2 (Glo  (None, 32)               0
 balMaxPooling1D)

 dense_5 (Dense)             (None, 1)                 33

=================================================================
Total params: 2,590,849
Trainable params: 2,590,849
Non-trainable params: 0
_________________________________________________________________
```

2.2. Results

```
model_2_results

{'Accuracy': 0.9845846733946989,
 'Precision': 0.9862524355921195,
 'Recall': 0.9836968257395811,
 'f1_score': 0.984972972972973}
```

*The same text vectorization and embedding layer are used

2.3. External Links

CNN_explained: [Convolutional Neural Networks, Explained | by Mayank Mishra | Towards Data Science]

3. RNN [Recurrent Neural Networks]

3.1. Model Architecture

```
model_3.summary()
```

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 1)]               0

 text_vectorization (TextVec  (None, 1994)             0
 torization)

 embedding_4 (Embedding)     (None, 1994, 128)         2560000

 lstm_2 (LSTM)               (None, 1994, 32)          20608

 gru_1 (GRU)                 (None, 1994, 32)          6336

 bidirectional_1 (Bidirectio  (None, 32)               6272
 nal)

 dense_6 (Dense)             (None, 1)                 33

=================================================================
Total params: 2,593,249
Trainable params: 2,593,249
Non-trainable params: 0
_________________________________________________________________
```

3.2. Results

```
model_3_results
```

```
{'Accuracy': 0.9660641011422868,
 'Precision': 0.9686757399006265,
 'Recall': 0.9653390742734123,
 'f1_score': 0.9670045287901661}
```

3.3. External Links

RNN_explained: [Recurrent Neural Networks (RNN) Explained — the ELI5 way | by Niranjan Kumar | Towards Data Science]

4. Transfer learning

4.1. Model Architecture

```
model_4.summary()
```

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None,)]                 0

 keras_layer (KerasLayer)    (None, 512)               256797824

 dense_8 (Dense)             (None, 64)                32832

 dense_9 (Dense)             (None, 32)                2080

 dense_10 (Dense)            (None, 1)                 33

=================================================================
Total params: 256,832,769
Trainable params: 34,945
Non-trainable params: 256,797,824
_________________________________________________________________
```

4.2. USE [Universal Sentence Encoder]

Use is a pertain embedding layer, it takes in the input as words and returns a vector embedding of dimension 3.

Link: [TensorFlow Hub (tfhub.dev)](https://tfhub.dev)

4.3. Results

```
model_4_results
```

```
{'Accuracy': 0.9184873017633359,
 'Precision': 0.9405137449301487,
 'Recall': 0.8986006458557588,
 'f1_score': 0.9190795992513486}
```

5. Custom + Transfer learning

5.1. Architecture

```
model_5.summary()
```

```
Model: "model_6"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_5 (InputLayer)           [(None, 1)]          0           []

 text_vectorization (TextVector  (None, 1994)        0           ['input_5[0][0]']
 ization)

 embedding_3 (Embedding)        (None, 1994, 128)    2560000     ['text_vectorization[3][0]']

 input_6 (InputLayer)           [(None,)]            0           []

 global_max_pooling1d_2 (Global  (None, 128)         0           ['embedding_3[0][0]']
 MaxPooling1D)

 keras_layer (KerasLayer)       (None, 512)          256797824   ['input_6[0][0]']

 concatenate (Concatenate)      (None, 640)          0           ['global_max_pooling1d_2[0][0]',
                                                                  'keras_layer[1][0]']

 dense_8 (Dense)                (None, 64)           41024       ['concatenate[0][0]']

 dense_9 (Dense)                (None, 32)           2080        ['dense_8[0][0]']

 dense_10 (Dense)               (None, 1)            33          ['dense_9[0][0]']

==================================================================================================
Total params: 259,400,961
Trainable params: 2,603,137
Non-trainable params: 256,797,824
__________________________________________________________________________________________________
```

5.2. Custom embedding + Transfer learning embedding

Custom embedding is the same embedding built and trained from scratch. Whereas the transfer learning embedding is a pre-trained embedding.

Both the embeddings are put together to yield better performance.

*Note we have to create dual input as done in code to fuse these embeddings together.

5.3. Results

```
model_5_results
```

```
{'Accuracy': 0.9836974603526671,
 'Precision': 0.9830326460481099,
 'Recall': 0.9853606027987083,
 'f1_score': 0.9841952478228148}
```

6. **Ensemble**

6.1. Architecture

```
%%time
pred_probs=[]

for i in range(3):

  model_1_ensemble.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

  history_1_ensemble=model_1_ensemble.fit(train_dataset,
                          epochs=5,
                          callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_accuracy", patience=5, restore_best_weights=True),
                                     tf.keras.callbacks.ReduceLROnPlateau(monitor="val_accuracy", patience=2)],
                          validation_data=(test_dataset),
                          verbose=0)
  y_preds=model_1_ensemble.predict(test_dataset).squeeze()
  pred_probs.append(y_preds.tolist())

  model_2_ensemble.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

  history_2_ensemble=model_2_ensemble.fit(train_dataset,
                          epochs=5,
                          callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_accuracy", patience=2, restore_best_weights=True),
                                     tf.keras.callbacks.ReduceLROnPlateau(monitor="val_accuracy", patience=1)],
                          validation_data=(test_dataset),
                          verbose=0)
  y_preds=model_2_ensemble.predict(test_dataset).squeeze()
  pred_probs.append(y_preds.tolist())
```

6.2. Ensemble explained

Ensemble combines the knowledge gained by different learning algorithms ie. it pools together knowledge from the different experiments and models run.

6.3. Results

```
model_6_results= calculate_metrics(test_labels, ensemble_preds.round())
model_6_results
```

```
{'Accuracy': 0.9879671731174449,
 'Precision': 0.990065867616888,
 'Recall': 0.9865504626640843,
 'f1_score': 0.988305039073026}
```

6.4. External Links

[Ensemble learning - Wikipedia]()

State-of-the-art model

Bert, is considered one of the finest models for nlp application it can take on text generation, text classification, question and answers, and many more.

We have implemented a tesorflow_hub version of it(pre-processor + model)

Architecture:

```
model_bert.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to                     
==================================================================================================
 input_2 (InputLayer)           [(None,)]            0           []                               
                                                                                                  
 keras_layer_2 (KerasLayer)     {'input_mask': (Non  0           ['input_2[0][0]']                
                                e, 128),                                                          
                                 'input_type_ids':                                                
                                (None, 128),                                                      
                                 'input_word_ids':                                                
                                (None, 128)}                                                      
                                                                                                  
 keras_layer_3 (KerasLayer)     {'default': (None,   109482241   ['keras_layer_2[0][0]',          
                                768),                             'keras_layer_2[0][1]',          
                                 'pooled_output': (               'keras_layer_2[0][2]']          
                                None, 768),                                                       
                                 'encoder_outputs':                                               
                                 [(None, 128, 768),                                               
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768),                                                
                                 (None, 128, 768)],                                               
                                 'sequence_output':                                               
                                 (None, 128, 768)}                                                
                                                                                                  
 dense_3 (Dense)                (None, 64)           49216       ['keras_layer_3[0][13]']         
                                                                                                  
 dense_4 (Dense)                (None, 32)           2080        ['dense_3[0][0]']                
                                                                                                  
 dense_5 (Dense)                (None, 1)            33          ['dense_4[0][0]']                
                                                                                                  
==================================================================================================
Total params: 109,533,570
Trainable params: 51,329
Non-trainable params: 109,482,241
__________________________________________________________________________________________________
```

Results:

```
model_bert_results= calculate_metrics(test_labels, y_preds.round())
model_bert_results
```

```
{'Accuracy': 0.9164910724187646,
 'Precision': 0.9246479630226808,
 'Recall': 0.9144254278728606,
 'f1_score': 0.9195082843399252}
```

Note:

1. Beautiful infographics [both charts and tables] have been provided at the conclusion section of both modules [ML_Approach, DL_Approach] in the colab for a quick glance.
2. Tensorboard results are also shared at the end of the colab with a dedicated section.
3. Training curves are plotted right after the models and can be viewed.

PS: A lot of genuine hard work and effort has been put to make this report and the colab notebook... hope you like it 🙂.