Aid Escalating Internet Coverage

Evaluation - 3 Code

Importing Dependencies

Lets import all necessary librarires: -

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error, mean_absolute_error
import nltk
import re
import json
from cleantext import clean
from tqdm import tqdm
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings("error")
warnings.filterwarnings("ignore", category=DeprecationWarning)
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from sklearn.feature_extraction.text import TfidfVectorizer
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error, mean_absolute_error
import nltk
import re
import json
from cleantext import clean
from tqdm import tqdm
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings("error")
warnings.filterwarnings("ignore", category=DeprecationWarning)
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from sklearn.feature_extraction.text import TfidfVectorizer
import seaborn as sns
from sklearn.decomposition import PCA
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from sklearn.pipeline import make_pipeline
from sklearn import svm
```

```
warnings.filterwarnings("ignore", category=FutureWarning)
from pyexpat import model
from sklearn.ensemble import RandomForestRegressor
import xgboost
import shap
import statsmodels.api as sm
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from urllib.parse import urlparse
from sklearn.impute import SimpleImputer
```

Let's read our train and test csv files

In [ ]:
```
df_train = pd.read_csv("./train.csv")
df_test = pd.read_csv("./test.csv")
df_train.head()
```

Out[ ]:

	link	link_id	page_description	alchemy_category	a
0	http://www.cbc.ca/stevenandchris/2012/11/peggy...	7426	{"url":"cbc ca stevenandchris 2012 11 peggy ks...	arts_entertainment	
1	http://www.instructables.com/id/Vegan-Baked-Po...	8430	{"title":"Vegan Potato Spinach Balls Fat Free ...	recreation	
2	http://www.oled-info.com/toshiba-shows-ultra-t...	3469	{"title":"Toshiba shows an ultra thin flexible...	business	
3	http://www.collegehumor.com/videos/playlist/64...	1326	{"url":"collegehumor videos playlist 6472556 e...	arts_entertainment	
4	http://sports.yahoo.com/nba/blog/ball_dont_lie...	3580	{"title":"Shaq admits to taking performance en...	sports	

5 rows × 27 columns

Preprocessing : -

Quite a few '?' values in the dataset, lets replace that with NaN.

In [ ]:
```
for column in df_train.columns:
    l = []
    l.append(column)
    df_train[df_train[l] == '?'] = np.nan

for col in df_test.columns:
    l = []
    l.append(col)
    df_test[df_test[l] == '?'] = np.nan
```

Lets check for NULL values

In [ ]:
```
df_train.isna().sum()
```

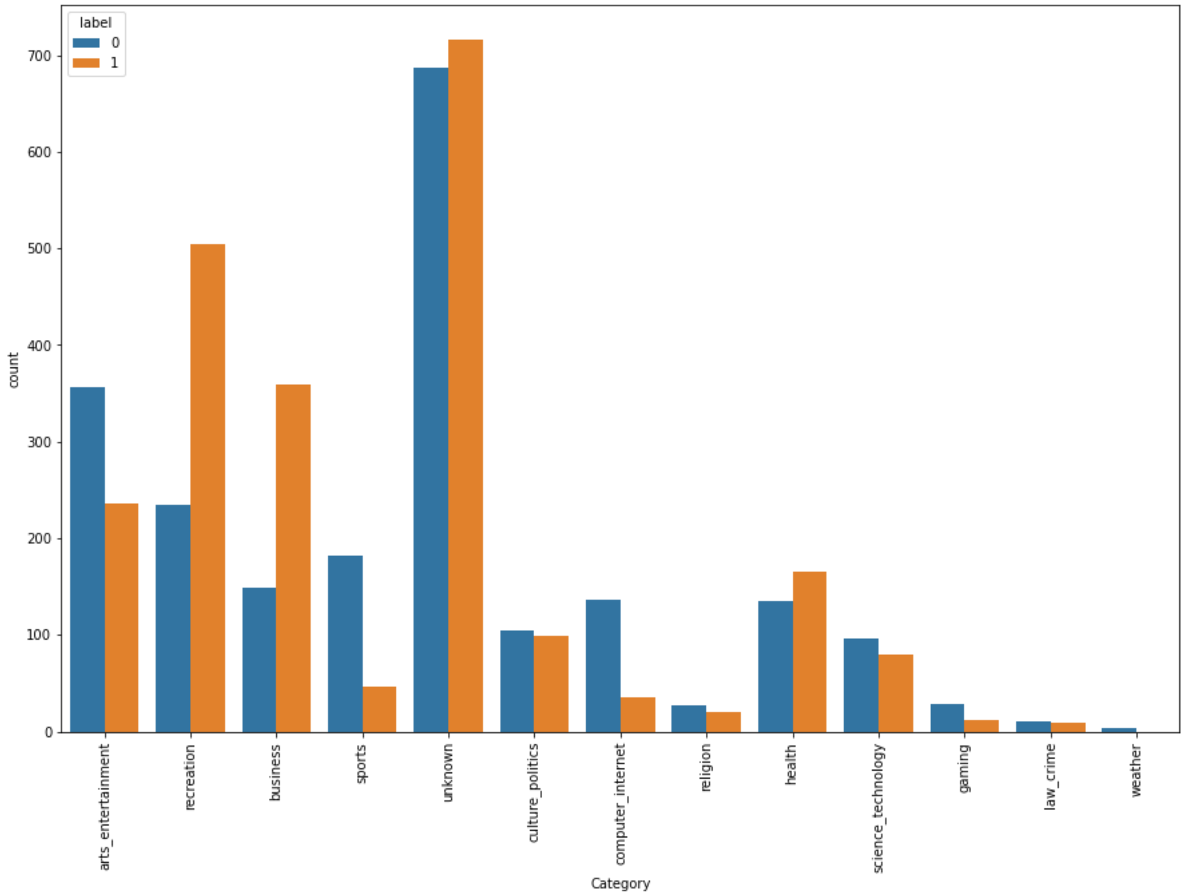
Out[ ]:
```
link                                    0
link_id                                 0
page_description                        0
alchemy_category                     1397
alchemy_category_score               1397
avg_link_size                           0
common_word_link_ratio_1                0
common_word_link_ratio_2                0
common_word_link_ratio_3                0
common_word_link_ratio_4                0
compression_ratio                       0
embed_ratio                             0
frame_based                             0
frame_tag_ratio                         0
has_domain_link                         0
html_ratio                              0
image_ratio                             0
is_news                              1688
lengthy_link_domain                     0
link_word_score                         0
news_front_page                       727
non_markup_alphanumeric_characters      0
count_of_links                          0
number_of_words_in_url                  0
parametrized_link_ratio                 0
spelling_mistakes_ratio                 0
label                                   0
dtype: int64
```

For the alchemy_category feature, which is categorical, before one hot encoding, lets replace the null values with 'unknown' and see correlation of each of its possible values with the label, which will help us decide whether it is an important feature or not.

In [ ]:
```
df_train['alchemy_category'] = df_train['alchemy_category'].replace(np.nan, 'unknow
df_test['alchemy_category'] = df_test['alchemy_category'].replace(np.nan, 'unknown
print(df_train['alchemy_category'].unique())
plt.figure(figsize=(15,10))
sns.countplot(x=df_train['alchemy_category'],hue=df_train['label'])
plt.xlabel('Category')
plt.xticks(rotation=90)
```

```
['arts_entertainment' 'recreation' 'business' 'sports' 'unknown'
 'culture_politics' 'computer_internet' 'religion' 'health'
 'science_technology' 'gaming' 'law_crime' 'weather']
```

Out[ ]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]),
 [Text(0, 0, 'arts_entertainment'),
  Text(1, 0, 'recreation'),
  Text(2, 0, 'business'),
  Text(3, 0, 'sports'),
  Text(4, 0, 'unknown'),
  Text(5, 0, 'culture_politics'),
  Text(6, 0, 'computer_internet'),
  Text(7, 0, 'religion'),
  Text(8, 0, 'health'),
  Text(9, 0, 'science_technology'),
  Text(10, 0, 'gaming'),
  Text(11, 0, 'law_crime'),
  Text(12, 0, 'weather')])
```



Clearly, there are few specific categories that have significant contribution to the output.

Therefore, let's one-hot encode our categorical non-textual feature, alchemy_category. And lets replace the non-categorical NULL values with their mean.

In [ ]:
```
df_train['alchemy_category_score'] = df_train['alchemy_category_score'].astype(flo
df_test['alchemy_category_score'] = df_test['alchemy_category_score'].astype(float
df_train['alchemy_category_score'].fillna(value=df_train['alchemy_category_score']
df_test['alchemy_category_score'].fillna(value=df_test['alchemy_category_score'].m

df_train['is_news'] = df_train['is_news'].astype(float)
df_test['is_news'] = df_test['is_news'].astype(float)
df_train['is_news'].fillna(value=df_train['is_news'].mean(), inplace=True)
df_test['is_news'].fillna(value=df_test['is_news'].mean(), inplace=True)

df_train['news_front_page'] = df_train['news_front_page'].astype(float)
df_test['news_front_page'] = df_test['news_front_page'].astype(float)
df_train['news_front_page'].fillna(value=df_train['news_front_page'].mean(), inpla
df_test['news_front_page'].fillna(value=df_test['news_front_page'].mean(), inplace

df_train = pd.get_dummies(df_train, columns = ['alchemy_category'])
df_test = pd.get_dummies(df_test, columns = ['alchemy_category'])
```

Feature Selection : -

This is a very important step. With appropriate selection of important features, and discarding of not-important features, we can find a model with the best possible accuracy.

I will be following the post: - https://towardsdatascience.com/feature-selection-techniques-for-classification-and-python-tips-for-their-application-10c0ddd7918b

The techniques I have used for analysing feature importance will be
1.) Unsupervised methods (PCA)
2.) Univariate Filtering technique (Logistic Regression)
3.) Wrapper methods (Forward and Backward selection)
4.) Tree based models to find feature importance (with xgboost)

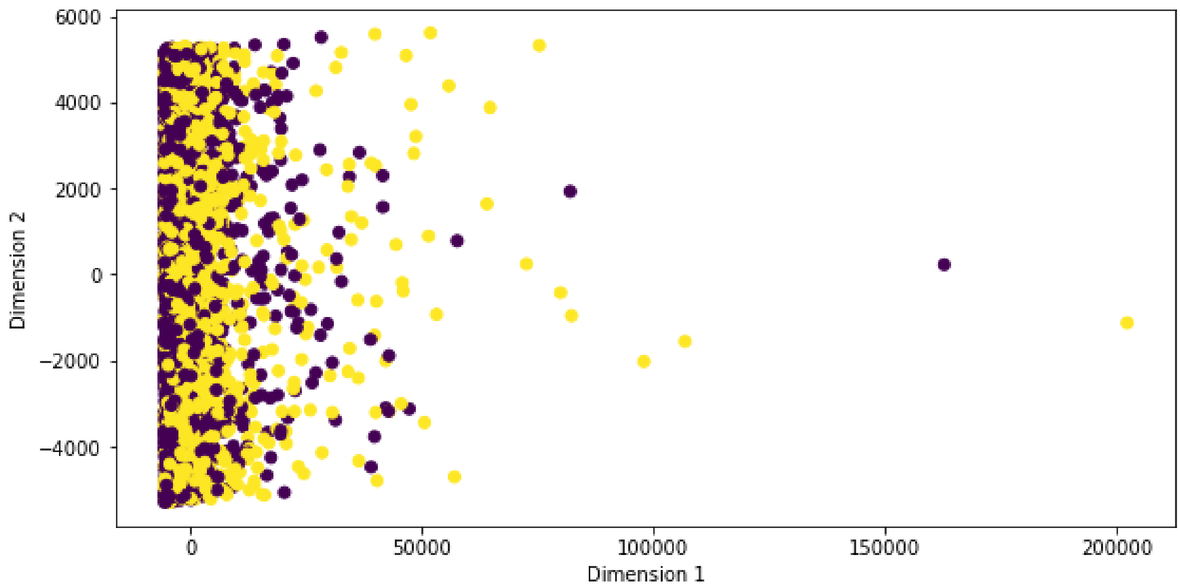
Each technique will be elaborated by me when I perform it.
Also, the "url" and "page_description" textual features are not included in this section, as we will perform NLP on them to analyse their importance later.

1.) Unsupervised feature groupingwith PCA. Lets see whether the class labels are somewhat serperable if PCA is applied on the Dataset.

In [ ]:
```
plt.figure(figsize=(10,5))
cols = list(df_train.columns)
cols.remove('page_description')
cols.remove('link')
cols.remove('label')
X_PCA = df_train.loc[:, cols].values
Y_PCA = df_train.loc[:, ['label']].values
X_PCA = PCA().fit_transform(X_PCA)

plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.scatter(X_PCA[:,0], X_PCA[:,1], c=Y_PCA)
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x2387b8e54f0>
```

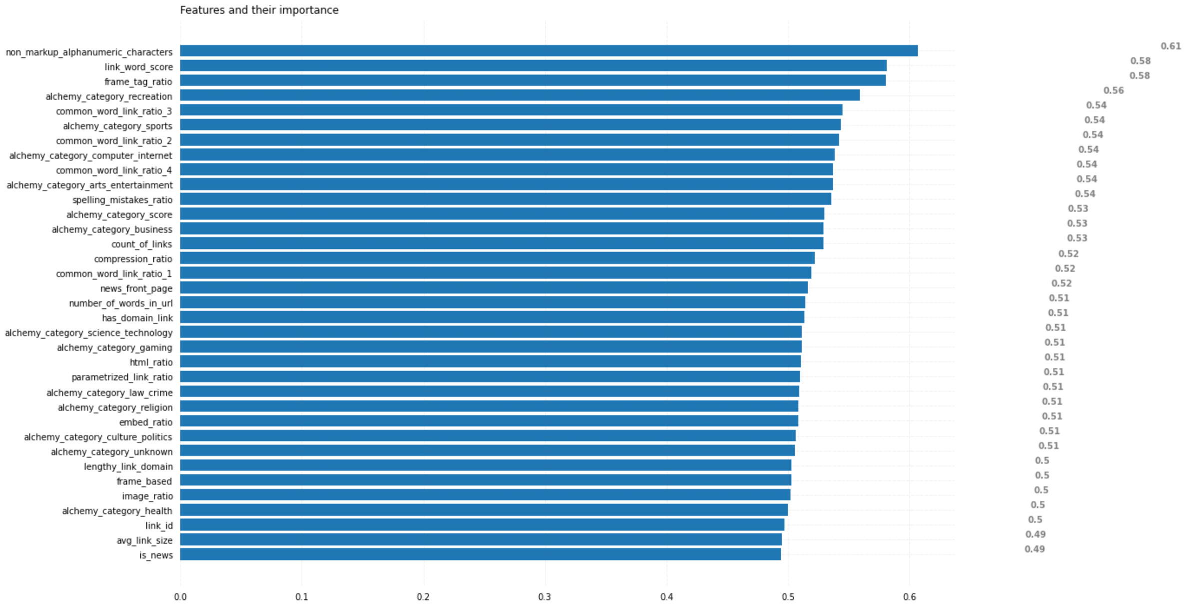


Therefore, we can see the data is somewhat random. It is not serperable on the Y-axis or the X-axis.

2.) Univariate Filtering technique.
I will use a Logistic Regression model to fit each feature with the class label. This is not the best method for multiple features as it completely sidelines covariance and multi-variable models accuracy, but it does help in finding few features, if they exist, that have very high correlation with the class label.

In [ ]:
```
Scores = []
for feature in df_train.columns:
        if feature != "link" and feature != "page_description" and feature != "alcl
                model = LogisticRegression(solver='saga')
                X = df_train[feature].to_numpy()
                Y = df_train["label"].to_numpy()
                scale = StandardScaler()
                X = scale.fit_transform(X.reshape(-1, 1))
                x_train, x_test, y_train, y_test = train_test_split(X, Y, shuffle=
                try:
                        model.fit(x_train.reshape(-1, 1), y_train)
                        predictions = model.predict(x_test.reshape(-1, 1))
                        roc_auc_score_feature = roc_auc_score(y_test, predictions)
                        # print(feature, "ROC-AUC score is", roc_auc_score_feature,
                        Logistic_regression_score = model.score(x_test.reshape(-1,
                        # print(feature, Logistic_regression_score, roc_auc_score_
                        Scores.append((Logistic_regression_score + roc_auc_score_fe
                except:
                        # print(feature, "Could not converge")
                        Scores.append(0)
df_columns = df_train.columns.to_list()
df_columns.remove("link")
df_columns.remove("page_description")
df_columns.remove("label")
list_new = []
for i in range(len(df_columns)):
        if Scores[i] != 0:
                list_new.append([df_columns[i], Scores[i]])
list_new = np.array(sorted(list_new, key=lambda x:x[1]))
# plt.barh(list_new[:,0], list_new[:,1])
# plt.show()

fig, ax = plt.subplots(figsize =(16, 12))
ax.barh(list_new[:,0], [float(i) for i in list_new[:,1]])
for s in ['top', 'bottom', 'left', 'right']:
    ax.spines[s].set_visible(False)
ax.xaxis.set_ticks_position('none')
ax.yaxis.set_ticks_position('none')
ax.xaxis.set_tick_params(pad = 5)
ax.yaxis.set_tick_params(pad = 5)
ax.grid(b = True, color ='grey',
        linestyle ='-.', linewidth = 0.5,
        alpha = 0.2)
# ax.invert_yaxis()
for i in ax.patches:
    plt.text(i.get_width()+0.2, i.get_y()+0.5,
             str(round((i.get_width()), 2)),
             fontsize = 10, fontweight ='bold',
             color ='grey')
ax.set_title('Features and their importance',
             loc ='left')
plt.show()
```



Takeaways: -

There are no standout features with a roc_score more than 0.75. Highest is 0.6 infact. Therefore, we must do some sort of multivariable filtering to see what model and what selection of features gives the highest accuracy.

3.) Wrapper functions.

I will be using the 3rd party library, mlxtend to perform its feature selection functions, by using Exhaustive forward searching, Sequential forward, and backward searching, with and without floating flag selection.

In [ ]:
```
LR = LogisticRegression()
SVM = svm.SVC(kernel='rbf')
clf = make_pipeline(StandardScaler(), SVM)
X_FS = df_train.loc[:, cols].values
Y_FS = df_train.loc[:, ['label']].values

# EFS_LR = EFS(LR, min_features = 1, max_features = len(cols), scoring='roc_auc', p
# EFS_LR.fit(X_FS, Y_FS.ravel())
# print('Best accuracy score: %.2f' % EFS_LR.best_score_)
# print('Best subset (indices):', EFS_LR.best_idx_)
# print('Best subset (corresponding names):', EFS_LR.best_feature_names_)

SFS_LR = SFS(LR, k_features=(1, len(cols)), forward=True, floating=False, scoring=
SFS_LR.fit(X_FS, Y_FS.ravel())
print('\nSequential Forward Selection:')
print(SFS_LR.k_feature_idx_)
print('CV Score:')
print(SFS_LR.k_score_)
```

```
Sequential Forward Selection:
(1, 3, 4, 5, 9, 10, 11, 12, 16, 17, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 3
0, 31, 33, 34, 35)
CV Score:
0.7031114327047892
```

Takeaways: -

The output shows SFS search gave a maximum roc score of 0.70, when it chose features with the index list shown in the output.

The model used was Logistic Regression, as a SVM with a linear kernel, a SVM with a RBF kernel gave very low roc scores, and a RandomForestRegressor had a close but slightly

smaller roc score.
Exhaustive forward search had extremly slow convergence, so it is commented.
SBS didn't converge, therefore, its code has been removed. And neither did it converge with the floating=True flag.

4.) Tree based model feature importance.
I have used an XGBoost model for our data.

In [ ]:
```
XGBoost_Model = xgboost.train({"learning_rate": 0.01}, xgboost.DMatrix(df_train[co
shap.initjs()
model_explainer = shap.TreeExplainer(XGBoost_Model)
shap_values = model_explainer.shap_values(df_train[cols])

shap.summary_plot(shap_values, df_train[cols], plot_type="Bar")
```

Takeaways: -
This modelling technique for feature selection is used to solve the slow convergence or no convergence issues of step 3.
The problem is, the model punishes features with high covariance among themselves.
Therefore, we can see some features match our expected importance from step 3, and some do not.
But due to the covariance penalisation issue, we use the result from step 3 moving forward.

Our SFS_cols list stores the features which SFS used to get the highest roc score.

In [ ]:
```
SFS_cols = [cols[col_id] for col_id in SFS_LR.k_feature_idx_]
print(SFS_cols)
```

```
['alchemy_category_score', 'common_word_link_ratio_1', 'common_word_link_ratio_2',
'common_word_link_ratio_3', 'frame_based', 'frame_tag_ratio', 'has_domain_link',
'html_ratio', 'link_word_score', 'news_front_page', 'non_markup_alphanumeric_chara
cters', 'count_of_links', 'parametrized_link_ratio', 'spelling_mistakes_ratio', 'a
lchemy_category_arts_entertainment', 'alchemy_category_business', 'alchemy_categor
y_computer_internet', 'alchemy_category_culture_politics', 'alchemy_category_gamin
g', 'alchemy_category_health', 'alchemy_category_law_crime', 'alchemy_category_rec
reation', 'alchemy_category_religion', 'alchemy_category_sports', 'alchemy_categor
y_unknown', 'alchemy_category_weather']
```

news_front_page had quite a few missing values, therefore, we will drop the column.
frame_based has only 1 value, 0. Therefore, we will drop that column.
There are not many weather alchemy category types, hence, we will drop it.

In [ ]:
```
SFS_cols.remove('news_front_page')
SFS_cols.remove('frame_based')
SFS_cols.remove('alchemy_category_weather')
SFS_cols_categorical = ['has_domain_link', 'alchemy_category_arts_entertainment',
SFS_cols_not_categorical = ['alchemy_category_score', 'common_word_link_ratio_1',
print(SFS_cols)
```

```
['alchemy_category_score', 'common_word_link_ratio_1', 'common_word_link_ratio_2',
'common_word_link_ratio_3', 'frame_tag_ratio', 'has_domain_link', 'html_ratio', 'l
ink_word_score', 'non_markup_alphanumeric_characters', 'count_of_links', 'parametr
ized_link_ratio', 'spelling_mistakes_ratio', 'alchemy_category_arts_entertainmen
t', 'alchemy_category_business', 'alchemy_category_computer_internet', 'alchemy_ca
tegory_culture_politics', 'alchemy_category_gaming', 'alchemy_category_health', 'a
lchemy_category_law_crime', 'alchemy_category_recreation', 'alchemy_category_relig
ion', 'alchemy_category_sports', 'alchemy_category_unknown']
```

NLP

Now lets shift our attention to the textual columns, "link" and "page_description". Lets preprocess those cloumns, starting with "page_description"

In [ ]:
```
CLEANR = re.compile('<.*?>')

def preprocess_text(text, html=True, clean_all=True, extra_spaces=True, stemming=Fa
    if html:
        text = re.sub(CLEANR, ' ', text)

    txt_list = []
    [txt_list.append(x) for x in text.split() if x not in txt_list]
    text = ' '.join(txt_list)
```

```
    text = clean(text, clean_all=clean_all, extra_spaces=extra_spaces, stemming=st
    return text

l_train = []
for i in df_train.page_description.values:
    st = ''
    txt = json.loads(i)
    if 'title' in txt.keys():
        if type(txt['title']) != type(None):
            st += txt['title']
    l_train.append(st)
df_train['text_title'] = l_train

l_test = []
for i in df_test.page_description.values:
    st = ''
    txt = json.loads(i)
    if 'title' in txt.keys():
        if type(txt['title']) != type(None):
            st += txt['title']
    l_test.append(st)
df_test['text_title'] = l_test

l_train = []
for i in df_train.page_description.values:
    st = ''
    txt = json.loads(i)
    if 'url' in txt.keys():
        if type(txt['url']) != type(None):
            st += txt['url']
    l_train.append(st)
df_train['text_url'] = l_train

l_test = []
for i in df_test.page_description.values:
    st = ''
    txt = json.loads(i)
    if 'url' in txt.keys():
        if type(txt['url']) != type(None):
            st += txt['url']
    l_test.append(st)
df_test['text_url'] = l_test

l_train = []
for i in df_train.page_description.values:
    st = ''
    txt = json.loads(i)
    if 'body' in txt.keys():
        if type(txt['body']) != type(None):
            st += txt['body']
    l_train.append(st)
df_train['text_body'] = l_train

l_test = []
for i in df_test.page_description.values:
    st = ''
    txt = json.loads(i)
    if 'body' in txt.keys():
        if type(txt['body']) != type(None):
            st += txt['body']
    l_test.append(st)
df_test['text_body'] = l_test
```

In [ ]:
```
l = []
for i in tqdm(df_train.text_title, total=len(df_train)):
    try:
        l.append(preprocess_text(i))
    except:
        l.append("not found")
        pass
df_train.text_title = l

l = []
for i in tqdm(df_test.text_title, total=len(df_test)):
    try:
        l.append(preprocess_text(i))
    except:
        l.append("not found")
        pass
df_test.text_title = l

l = []
for i in tqdm(df_train.text_url, total=len(df_train)):
    try:
        l.append(preprocess_text(i))
    except:
        l.append("not found")
        pass
df_train.text_url = l

l = []
for i in tqdm(df_test.text_url, total=len(df_test)):
    try:
        l.append(preprocess_text(i))
    except:
        l.append("not found")
        pass
df_test.text_url = l

l = []
for i in tqdm(df_train.text_body, total=len(df_train)):
    try:
        l.append(preprocess_text(i))
    except:
        l.append("not found")
        pass
df_train.text_body = l

l = []
for i in tqdm(df_test.text_body, total=len(df_test)):
    try:
        l.append(preprocess_text(i))
    except:
        l.append("not found")
        pass
df_test.text_body = l
```

```
100%|██████████| 4437/4437 [00:01<00:00, 2282.22it/s]
100%|██████████| 2958/2958 [00:01<00:00, 2341.55it/s]
100%|██████████| 4437/4437 [00:01<00:00, 2342.98it/s]
100%|██████████| 2958/2958 [00:01<00:00, 2322.76it/s]
100%|██████████| 4437/4437 [00:05<00:00, 819.08it/s]
100%|██████████| 2958/2958 [00:03<00:00, 802.82it/s]
```

In [ ]:
```
text_list = []
for text in df_train.text_title:
```

```
    text = re.sub("[^a-zA-Z]", " ", text)
    text = nltk.word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    text = [w for w in text if not w in stop_words]
    lemma = nltk.WordNetLemmatizer()
    text = [lemma.lemmatize(word) for word in text]
    text = " ".join(text)
    text_list.append(text)
df_train.text_title = text_list

text_list_test = []
for text in df_test.text_title:
    text = re.sub("[^a-zA-Z]", " ", text)
    text = nltk.word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    text= [w for w in text if not w in stop_words]
    lemma = nltk.WordNetLemmatizer()
    text = [lemma.lemmatize(word) for word in text]
    text = " ".join(text)
    text_list_test.append(text)
df_test.text_title = text_list_test

text_list = []
for text in df_train.text_url:
    text = re.sub("[^a-zA-Z]", " ", text)
    text = nltk.word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    text = [w for w in text if not w in stop_words]
    lemma = nltk.WordNetLemmatizer()
    text = [lemma.lemmatize(word) for word in text]
    text = " ".join(text)
    text_list.append(text)
df_train.text_url = text_list

text_list_test = []
for text in df_test.text_url:
    text = re.sub("[^a-zA-Z]", " ", text)
    text = nltk.word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    text= [w for w in text if not w in stop_words]
    lemma = nltk.WordNetLemmatizer()
    text = [lemma.lemmatize(word) for word in text]
    text = " ".join(text)
    text_list_test.append(text)
df_test.text_url = text_list_test

text_list = []
for text in df_train.text_body:
    text = re.sub("[^a-zA-Z]", " ", text)
    text = nltk.word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    text = [w for w in text if not w in stop_words]
    lemma = nltk.WordNetLemmatizer()
    text = [lemma.lemmatize(word) for word in text]
    text = " ".join(text)
    text_list.append(text)
df_train.text_body = text_list

text_list_test = []
for text in df_test.text_body:
    text = re.sub("[^a-zA-Z]", " ", text)
    text = nltk.word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    text= [w for w in text if not w in stop_words]
```

```
    lemma = nltk.WordNetLemmatizer()
    text = [lemma.lemmatize(word) for word in text]
    text = " ".join(text)
    text_list_test.append(text)
df_test.text_body = text_list_test
```

Now we have successfully parsed the page_description feature, preprocessed the text the stored them in the text_title, text_url, text_body columns newly made.

Now lets parse the link column and store it in the parsed_link column

In [ ]:
```
from urllib.parse import urlparse

l = []
for i in range(0, len(df_train)):
        url = df_train['link'][i]
        parsed_url = urlparse(url)
        scheme = parsed_url.scheme
        netloc = parsed_url.netloc
        path = parsed_url.path
        params = parsed_url.params
        query = parsed_url.query
        fragment = parsed_url.fragment

        parsed_url_str = " ".join([scheme, netloc, path, params, query, fragment])
        parsed_url_str = preprocess_text(parsed_url_str)
        parsed_url_str = re.sub("[^a-zA-Z]", " ", parsed_url_str)
        parsed_url_str = nltk.word_tokenize(parsed_url_str)
        stop_words = set(stopwords.words('english'))
        parsed_url_str = [w for w in parsed_url_str if not w in stop_words]
        lemma = nltk.WordNetLemmatizer()
        parsed_url_str = [lemma.lemmatize(word) for word in parsed_url_str]
        parsed_url_str = " ".join(parsed_url_str)
        l.append(parsed_url_str)
df_train['parsed_link'] = l

l = []
for i in range(0, len(df_test)):
        url = df_test['link'][i]
        parsed_url = urlparse(url)
        scheme = parsed_url.scheme
        netloc = parsed_url.netloc
        path = parsed_url.path
        params = parsed_url.params
        query = parsed_url.query
        fragment = parsed_url.fragment

        parsed_url_str = " ".join([scheme, netloc, path, params, query, fragment])
        parsed_url_str = preprocess_text(parsed_url_str)
        parsed_url_str = re.sub("[^a-zA-Z]", " ", parsed_url_str)
        parsed_url_str = nltk.word_tokenize(parsed_url_str)
        stop_words = set(stopwords.words('english'))
        parsed_url_str = [w for w in parsed_url_str if not w in stop_words]
        lemma = nltk.WordNetLemmatizer()
        parsed_url_str = [lemma.lemmatize(word) for word in parsed_url_str]
        parsed_url_str = " ".join(parsed_url_str)
        l.append(parsed_url_str)
df_test['parsed_link'] = l
```

In [ ]:
```
df_train.head()
```

Out[ ]:

	link	link_id	page_description	alchemy_category_sc
0	http://www.cbc.ca/stevenandchris/2012/11/peggy...	7426	{"url":"cbc ca stevenandchris 2012 11 peggy ks...	0.471
1	http://www.instructables.com/id/Vegan-Baked-Po...	8430	{"title":"Vegan Potato Spinach Balls Fat Free ...	0.885
2	http://www.oled-info.com/toshiba-shows-ultra-t...	3469	{"title":"Toshiba shows an ultra thin flexible...	0.716
3	http://www.collegehumor.com/videos/playlist/64...	1326	{"url":"collegehumor videos playlist 6472556 e...	0.562
4	http://sports.yahoo.com/nba/blog/ball_dont_lie...	3580	{"title":"Shaq admits to taking performance en...	0.893

5 rows × 43 columns

Model Ensemble

Now lets try a few models and test their accuracy.

Model 1 (ROC SCORE 0.87634)

I have used the Tfidf vectorizer in the the textual data pipeline and a linear classifier is used at the end of the ColumnTransformer.

In [ ]:

```
vectorizer1 = TfidfVectorizer()
vectorizer2 = TfidfVectorizer()
vectorizer3 = TfidfVectorizer()

column_transformer = ColumnTransformer([('tfidf1', vectorizer1, 'text_title'), ('t
pipe = Pipeline([('tfidf', column_transformer), ('logistic_regression', LogisticRe
pipe.fit(df_train[['text_title', 'text_url', 'text_body']], df_train['label'])
predictions = pipe.predict_proba(df_test[['text_title', 'text_url', 'text_body']])
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('submission_pipeline_no_url_LR.csv')
pred_df.head()
```

```
[0.89196903 0.24151221 0.43480787 ... 0.25224833 0.41929032 0.30737405]
```

Out[ ]:

link_id	label
4049	0.891969
3692	0.241512
9739	0.434808
1548	0.741270
5574	0.969916

This gave a roc score of 0.87634 on kaggle, which is better than our evaluation 1's submission as we have counted the body as well.

Model 2 (ROC SCORE 0.87634)

On inspection, our textual data, has some single letter data, and also some words not there in the nltk dictionary.

In [ ]:

```
nltk.download('words')
words = set(nltk.corpus.words.words())
l_title = []
l_url = []
l_body = []
for i in range(0, len(df_train)):
        # if i % 500 == 0:
        #       print(i, df_train['text_title'][i], df_train['text_url'][i], df_tra
        l_title.append(' '.join([w for w in df_train['text_title'][i].split() if le
        l_url.append(' '.join([w for w in df_train['text_url'][i].split() if len(w
        l_body.append(' '.join([w for w in df_train['text_body'][i].split() if len

        # l_title[len(l_title) - 1] = " ".join(w for w in nltk.wordpunct_tokenize(
        # l_url[len(l_url) - 1] = " ".join(w for w in nltk.wordpunct_tokenize(l_ur
        # l_body[len(l_body) - 1] = " ".join(w for w in nltk.wordpunct_tokenize(l_b
df_train['text_title'] = l_title
df_train['text_url'] = l_url
df_train['text_body'] = l_body

l_title = []
l_url = []
l_body = []
for i in range(0, len(df_test)):
        # if i % 500 == 0:
        #       print(i, df_train['text_title'][i], df_train['text_url'][i], df_tra
        l_title.append(' '.join([w for w in df_test['text_title'][i].split() if le
        l_url.append(' '.join([w for w in df_test['text_url'][i].split() if len(w)
        l_body.append(' '.join([w for w in df_test['text_body'][i].split() if len(w

        # l_title[len(l_title) - 1] = " ".join(w for w in nltk.wordpunct_tokenize(
        # l_url[len(l_url) - 1] = " ".join(w for w in nltk.wordpunct_tokenize(l_ur
        # l_body[len(l_body) - 1] = " ".join(w for w in nltk.wordpunct_tokenize(l_b
df_test['text_title'] = l_title
df_test['text_url'] = l_url
df_test['text_body'] = l_body
```

```
[nltk_data] Downloading package words to
[nltk_data]     C:\Users\abhin\AppData\Roaming\nltk_data...
[nltk_data]   Package words is already up-to-date!
```

In [ ]:
```
pipe = Pipeline([('tfidf', column_transformer), ('logistic_regression', LogisticRe
# pipe = Pipeline([('tfidf', column_transformer), ('svm', svm.SVC(kernel="rbf"))])
# pipe = Pipeline([('tfidf', column_transformer), ('random_forest', RandomForestRe
pipe.fit(df_train[['text_title', 'text_url', 'text_body']], df_train['label'])
predictions = pipe.predict_proba(df_test[['text_title', 'text_url', 'text_body']])
# predictions = pipe.predict(df_test[['text_title', 'text_url', 'text_body']])
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('submission_pipeline_no_url_LR_no_single_letters.csv')
pred_df.head()
```

```
[0.89196903 0.24151221 0.43480787 ... 0.25224833 0.41929032 0.30737405]
```

Out[ ]:

	label
link_id	
4049	0.891969
3692	0.241512
9739	0.434808
1548	0.741270
5574	0.969916

The roc score was unaltered on removing 1 letter words. (87.643) The roc score droppedon removing non-vocabulary words, therefore, that section has been commented.

Model 3 (ROC SCORE 0.87504)

Before, we excluded the parsed_url part, now we will include it in the column transformer and see the results.

In [ ]:
```
vectorizer1 = TfidfVectorizer()
vectorizer2 = TfidfVectorizer()
vectorizer3 = TfidfVectorizer()
vectorizer4 = TfidfVectorizer()

column_transformer = ColumnTransformer([('tfidf1', vectorizer1, 'text_title'), ('t
pipe = Pipeline([('tfidf', column_transformer), ('logistic_regression', LogisticRe

df_local_train, df_local_test = train_test_split(df_train, shuffle=True, test_size
pipe.fit(df_local_train[['text_title', 'text_url', 'text_body', 'parsed_link']], d
predictions = pipe.predict(df_local_test[['text_title', 'text_url', 'text_body', '
roc_auc_score_pipeline = roc_auc_score(df_local_test['label'], predictions)
print(roc_auc_score_pipeline)

pipe.fit(df_train[['text_title', 'text_url', 'text_body', 'parsed_link']], df_trai
predictions = pipe.predict_proba(df_test[['text_title', 'text_url', 'text_body', '
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('submission_pipeline_with_url_LR.csv')
pred_df.head()
```

```
0.8151614940454709
[0.91092243 0.23591797 0.38782026 ... 0.21605735 0.42444132 0.27184582]
```

Out[ ]:

	label
link_id	
4049	0.910922
3692	0.235918
9739	0.387820
1548	0.766096
5574	0.977518

Model 4: - (Best Model so far, ROC SCORE 0.88243)

To increase our accuracy, what I will do now is exclude the pipelined structure and create my own.
I will combine all the columns, text_title, text_url, text_body, parsed_link, into 1 column called Complete_Textual_Data.
Now I will build ONE Tfidf Vectorizer which fits the the entire vocabulary of BOTH the train and test data and then perform transformations on the components.

In [ ]:

```
l = []
for i in range(0, len(df_train)):
        l.append(" ".join([df_train["text_title"][i], df_train["text_url"][i], df_
df_train['Complete_Textual_Data'] = l

l = []
for i in range(0, len(df_test)):
        l.append(" ".join([df_test["text_title"][i], df_test["text_url"][i], df_te
df_test['Complete_Textual_Data'] = l

SFS_cols.append('Complete_Textual_Data')

vectorizer = TfidfVectorizer()
df_local_train, df_local_test = train_test_split(df_train, shuffle=True, test_size
model = LogisticRegression()
vectorizer.fit_transform(df_train.Complete_Textual_Data.values).toarray()
X_train_tfidf = vectorizer.transform(df_local_train['Complete_Textual_Data']).toar
X_test_tfidf = vectorizer.transform(df_local_test['Complete_Textual_Data']).toarra

LR_model = LogisticRegression()
LR_model.fit(X_train_tfidf, df_local_train['label'])

predictions = LR_model.predict(X_test_tfidf)
roc_score = roc_auc_score(df_local_test['label'], predictions)
print(roc_score)

vectorizer = TfidfVectorizer()
model = LogisticRegression()
print(type(df_train.Complete_Textual_Data.values))
vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, df_

X_train = vectorizer.transform(df_train['Complete_Textual_Data'])
X_test = vectorizer.transform(df_test['Complete_Textual_Data'])

LR_model = LogisticRegression()
LR_model.fit(X_train, df_train['label'])
```

```
predictions = LR_model.predict_proba(X_test)[:, 1]
print(len(predictions))
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('submission_mypipeline_LR.csv')
pred_df.head()
```

```
0.7994149541770469
<class 'numpy.ndarray'>
2958
```

Out[ ]:

link_id	label
4049	0.883630
3692	0.228657
9739	0.417026
1548	0.528183
5574	0.915144

This gave a roc score of 88.23 on kaggle.

Model 5 (ROC SCORE 0.86407)

Now, I would include the non-textual data too in the model. My model includes a Tfidf Pipeline for textual data, a one-hot-encoder for categorical data, and a median imputer followed by standard scaling for non categorical features which is then fed to a linear classifier(SVM, RandomForestRegressor, LinearRegression) by a Column Transformer architecture.

In [ ]:

```
print(SFS_cols)
print(SFS_cols_categorical)
print(SFS_cols_not_categorical)
```

```
['alchemy_category_score', 'common_word_link_ratio_1', 'common_word_link_ratio_2', 'common_word_link_ratio_3', 'frame_tag_ratio', 'has_domain_link', 'html_ratio', 'l
ink_word_score', 'non_markup_alphanumeric_characters', 'count_of_links', 'parametr
ized_link_ratio', 'spelling_mistakes_ratio', 'alchemy_category_arts_entertainmen
t', 'alchemy_category_business', 'alchemy_category_computer_internet', 'alchemy_ca
tegory_culture_politics', 'alchemy_category_gaming', 'alchemy_category_health', 'a
lchemy_category_law_crime', 'alchemy_category_recreation', 'alchemy_category_relig
ion', 'alchemy_category_sports', 'alchemy_category_unknown', 'Complete_Textual_Dat
a']
['has_domain_link', 'alchemy_category_arts_entertainment', 'alchemy_category_busin
ess', 'alchemy_category_computer_internet', 'alchemy_category_culture_politics',
'alchemy_category_gaming', 'alchemy_category_health', 'alchemy_category_law_crim
e', 'alchemy_category_recreation', 'alchemy_category_religion', 'alchemy_category_
sports', 'alchemy_category_unknown']
['alchemy_category_score', 'common_word_link_ratio_1', 'common_word_link_ratio_2',
'common_word_link_ratio_3', 'frame_tag_ratio', 'html_ratio', 'link_word_score', 'n
on_markup_alphanumeric_characters', 'count_of_links', 'parametrized_link_ratio',
'spelling_mistakes_ratio']
```

In [ ]:

```
vectorizer = TfidfVectorizer()
numeric_transformer = Pipeline(steps=[("imputer", SimpleImputer(strategy="median")
column_transformer = ColumnTransformer(transformers=[('tfidf', vectorizer, 'Comple
# clf = Pipeline(steps=[("preprocessor", column_transformer), ("classifier", Logis
clf = Pipeline(steps=[("preprocessor", column_transformer), ('random_forest', Rand
```

```
clf.fit(df_train[SFS_cols], df_train['label'])

predictions = clf.predict(df_test[SFS_cols])
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('submission_pipeline_nontextual_RF.csv')
pred_df.head()
```

```
[0.94192242 0.28317085 0.30665191 ... 0.19393447 0.21960383 0.19138055]
```

Out[ ]:

	label
link_id	
4049	0.941922
3692	0.283171
9739	0.306652
1548	0.527948
5574	0.926341

This model could not converge with LR, SVM, had slow convergence with RF, and low Roc Score as well, and from later visualisations we can see that this is because the data along the non-textual-columns are not highly linearly seperable.

SFS_new is the list of all important features used by our model, minus 'Complete_Textual_Data', as we usually include that in a Tfidf Pipeline which is seprate from how we deal with the other data.

In [ ]:

```
import copy
SFS_new = copy.copy(SFS_cols)
SFS_new.remove('Complete_Textual_Data')
print(SFS_new)
```

```
['alchemy_category_score', 'common_word_link_ratio_1', 'common_word_link_ratio_2', 'common_word_link_ratio_3', 'frame_tag_ratio', 'has_domain_link', 'html_ratio', 'l
ink_word_score', 'non_markup_alphanumeric_characters', 'count_of_links', 'parametr
ized_link_ratio', 'spelling_mistakes_ratio', 'alchemy_category_arts_entertainmen
t', 'alchemy_category_business', 'alchemy_category_computer_internet', 'alchemy_ca
tegory_culture_politics', 'alchemy_category_gaming', 'alchemy_category_health', 'a
lchemy_category_law_crime', 'alchemy_category_recreation', 'alchemy_category_relig
ion', 'alchemy_category_sports', 'alchemy_category_unknown']
```

Model 6 :- (ROC SCORE 0.88036)

Now, similar to how we dealt with Textual Data, I would use the same model as before, only this time I would remove 'Complete_Textual_Data' from out of my Pipeline and build my own implementation of it. The point is to make a TfidfVectorizer() that builds a vocabulary and dictionary jointly for both the test and train data and then for each individual component(train or test data) we would find their transofrmations by calling the transform() function on the vectorizer.

In [ ]:

```
column_transformer_no_text = ColumnTransformer(transformers=[('num', numeric_trans
# classifier_no_text = Pipeline(steps=[("preprocessor", column_transformer_no_text,
# classifier_no_text = Pipeline(steps=[("preprocessor", column_transformer_no_text,
classifier_no_text = Pipeline(steps=[("preprocessor", column_transformer_no_text),
```

```
classifier_no_text.fit(df_local_train[SFS_new], df_local_train['label'])

print(classifier_no_text.score(df_local_test[SFS_new], df_local_test['label']))

vectorizer = TfidfVectorizer()
vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, df_
X_local_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
X_local_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
X_test_tfidf = vectorizer.transform(df_test['Complete_Textual_Data'])
X_train_tfidf = vectorizer.transform(df_train['Complete_Textual_Data'])
model = LogisticRegression()
model.fit(X_local_train, df_local_train['label'])
print(model.score(X_local_test, df_local_test['label']))
predictions_classifier_no_text = classifier_no_text.predict_proba(df_local_test[SF
# print(predictions_classifier_no_text)
prediction_classifier_text = model.predict_proba(X_local_test)[:, 1]
# print(prediction_classifier_text)
predictions_train_no_text = classifier_no_text.predict_proba(df_local_train[SFS_ne
predictions_train_text = model.predict_proba(X_local_train)[:, 1]

# print(len(predictions_train_no_text))
# print(len(predictions_train_text))
integrating_model = LogisticRegression()
X_train = np.vstack((predictions_train_text, predictions_train_no_text)).T
# print(X_train)
integrating_model.fit(X_train, df_local_train['label'])
print(integrating_model.score(np.vstack((prediction_classifier_text, predictions_c
```

```
0.6569069069069069
0.7980480480480481
0.8063063063063063
```

In [ ]:

```
LR_model_text = LogisticRegression()
LR_model_text.fit(X_train_tfidf, df_train['label'])
LR_model_no_text = LogisticRegression()
LR_model_no_text.fit(df_train[SFS_new], df_train['label'])
probabilities_text = LR_model_text.predict_proba(X_train_tfidf)[:, 1]
probabilities_no_text = LR_model_no_text.predict_proba(df_train[SFS_new])[:, 1]
LR_model_main_pipeline = LogisticRegression()
LR_model_main_pipeline.fit(np.vstack((probabilities_text, probabilities_no_text)).

predictions = LR_model_main_pipeline.predict_proba(np.vstack((LR_model_text.predic
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('submission_mypipeline_nontextual_LR.csv')
pred_df.head()
```

```
[0.98550946 0.10029805 0.43590837 ... 0.14862474 0.15225852 0.12537356]
```

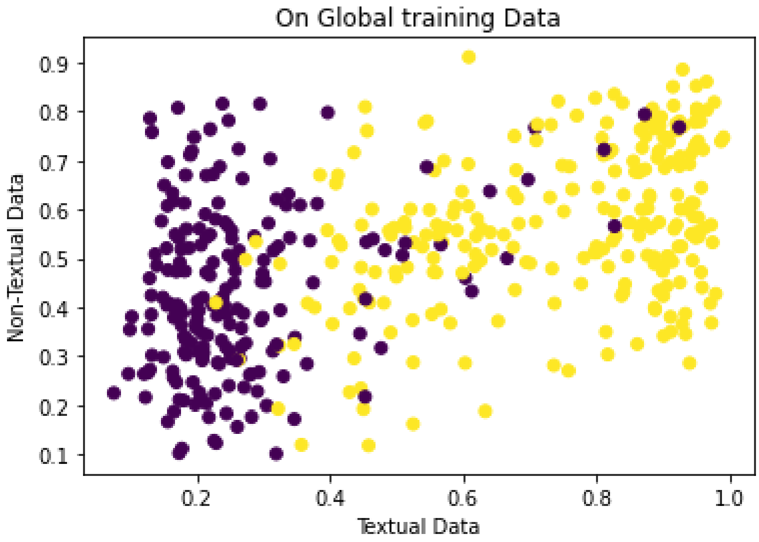
Out[ ]:

	label
link_id	
4049	0.985509
3692	0.100298
9739	0.435908
1548	0.627875
5574	0.983028

Now lets see some visualisations on the models: -

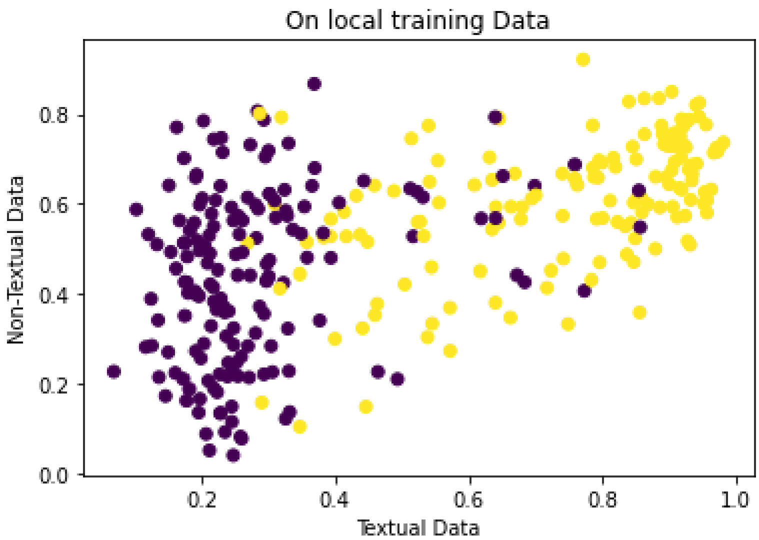
In [ ]:
```
plt.scatter(LR_model_text.predict_proba(X_train_tfidf)[:, 1][0::10], LR_model_no_te
plt.title('On Global training Data')
plt.xlabel('Textual Data')
plt.ylabel('Non-Textual Data')
```

Out[ ]:
```
Text(0, 0.5, 'Non-Textual Data')
```



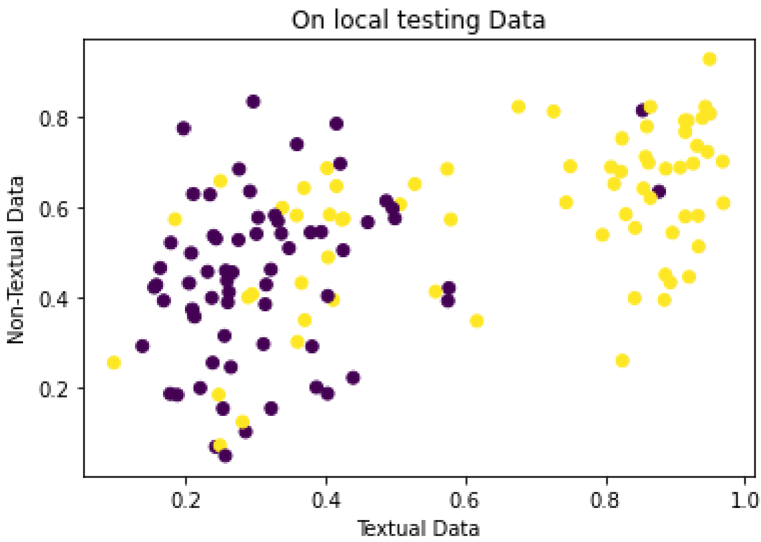
In [ ]:
```
plt.scatter(model.predict_proba(X_local_train)[:, 1][0::10], classifier_no_text.pre
plt.title('On local training Data')
plt.xlabel('Textual Data')
plt.ylabel('Non-Textual Data')
```

Out[ ]:
```
Text(0, 0.5, 'Non-Textual Data')
```



In [ ]:
```
plt.scatter(model.predict_proba(X_local_test)[:, 1][0::10], classifier_no_text.pre
plt.title('On local testing Data')
plt.xlabel('Textual Data')
plt.ylabel('Non-Textual Data')
```

Out[ ]:
```
Text(0, 0.5, 'Non-Textual Data')
```



As we can see, the textual_data classifier is able to segregate the data, while the data is not highly seperable with the non_textual features and inherently they get lower weights when we apply a linear classifier to it.

Model 7 (ROC SCORE 0.87407)

Now, I try to use a different pipeline approach, wherein instead of using 'Complete_Textual_Data', I use a LR on 'text_title', 'text_url', 'text_body' and 'parsed_link' individually, and use an integrating LR on top of the 4 LR's and check its roc score. The point of this technique is to see whether weight distribution between certain individual factors positively impacts the accuracy or not.

In [ ]:
```
vectorizer1 = TfidfVectorizer()
vectorizer2 = TfidfVectorizer()
vectorizer3 = TfidfVectorizer()
vectorizer4 = TfidfVectorizer()
vectorizer1.fit_transform(np.concatenate((df_train.text_title.values, df_test.text_
vectorizer2.fit_transform(np.concatenate((df_train.text_url.values, df_test.text_ur
vectorizer3.fit_transform(np.concatenate((df_train.text_body.values, df_test.text_b
vectorizer4.fit_transform(np.concatenate((df_train.parsed_link.values, df_test.pars

X_local_train1 = vectorizer1.transform(df_local_train['text_title'])
X_local_train2 = vectorizer2.transform(df_local_train['text_url'])
X_local_train3 = vectorizer3.transform(df_local_train['text_body'])
X_local_train4 = vectorizer4.transform(df_local_train['parsed_link'])
X_local_test1 = vectorizer1.transform(df_local_test['text_title'])
X_local_test2 = vectorizer2.transform(df_local_test['text_url'])
X_local_test3 = vectorizer3.transform(df_local_test['text_body'])
X_local_test4 = vectorizer4.transform(df_local_test['parsed_link'])

model1 = LogisticRegression()
model2 = LogisticRegression()
model3 = LogisticRegression()
model4 = LogisticRegression()
model1.fit(X_local_train1, df_local_train['label'])
model2.fit(X_local_train2, df_local_train['label'])
model3.fit(X_local_train3, df_local_train['label'])
model4.fit(X_local_train4, df_local_train['label'])

Integrating_Model_Local = LogisticRegression()
X_train_local_integrated = np.vstack((model1.predict_proba(X_local_train1)[:, 1], m
```

```
Integrating_Model_Local.fit(X_train_local_integrated, df_local_train['label'])

X_test_local_integrated = np.vstack((model1.predict_proba(X_local_test1)[:, 1], mo
print(Integrating_Model_Local.score(X_test_local_integrated, df_local_test['label'
```

```
0.8055555555555556
```

In [ ]:

```
X_main_train1 = vectorizer1.transform(df_train['text_title'])
X_main_train2 = vectorizer2.transform(df_train['text_url'])
X_main_train3 = vectorizer1.transform(df_train['text_body'])
X_main_train4 = vectorizer2.transform(df_train['parsed_link'])
X_main_test1 = vectorizer1.transform(df_test['text_title'])
X_main_test2 = vectorizer2.transform(df_test['text_url'])
X_main_test3 = vectorizer1.transform(df_test['text_body'])
X_main_test4 = vectorizer2.transform(df_test['parsed_link'])
model1 = LogisticRegression()
model2 = LogisticRegression()
model3 = LogisticRegression()
model4 = LogisticRegression()
model1.fit(X_main_train1, df_train['label'])
model2.fit(X_main_train2, df_train['label'])
model3.fit(X_main_train3, df_train['label'])
model4.fit(X_main_train4, df_train['label'])
Integrating_Model = LogisticRegression()
X_train_integrated = np.vstack((model1.predict_proba(X_main_train1)[:, 1], model2.
Integrating_Model.fit(X_train_integrated, df_train['label'])
X_test_integrated = np.vstack((model1.predict_proba(X_main_test1)[:, 1], model2.pr
predictions = Integrating_Model.predict_proba(X_test_integrated)[:, 1]
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('submission_Stanford_Pipeline_big_LR.csv')
pred_df.head()
```

```
[0.96258933 0.11789773 0.37800085 ... 0.26981564 0.40101902 0.17819508]
```

Out[ ]:

	label
link_id	
4049	0.962589
3692	0.117898
9739	0.378001
1548	0.959139
5574	0.998413

Model 8 (roc score 0.87892)

Another approach is too club-in all the textual data of page_description feature into one column and then along with 'parsed_link' feature, using the same pipeline as the previous model, see the results.

In [ ]:

```
l = []
for i in range(0, len(df_train)):
        l.append(" ".join([df_train['text_title'][i], df_train['text_url'][i], df_
df_train['parsed_page_description'] = l

l = []
for i in range(0, len(df_test)):
        l.append(" ".join([df_test['text_title'][i], df_test['text_url'][i], df_te
```

```
df_test['parsed_page_description'] = l

df_local_train, df_local_test = train_test_split(df_train, shuffle=True, test_size
vectorizer1 = TfidfVectorizer()
vectorizer2 = TfidfVectorizer()
vectorizer1.fit_transform(np.concatenate((df_train.parsed_page_description.values,
vectorizer2.fit_transform(np.concatenate((df_train.parsed_link.values, df_test.par

X_local_train1 = vectorizer1.transform(df_local_train['parsed_page_description'])
X_local_train2 = vectorizer2.transform(df_local_train['parsed_link'])
X_local_test1 = vectorizer1.transform(df_local_test['parsed_page_description'])
X_local_test2 = vectorizer2.transform(df_local_test['parsed_link'])

model1 = LogisticRegression()
model2 = LogisticRegression()
model1.fit(X_local_train1, df_local_train['label'])
model2.fit(X_local_train2, df_local_train['label'])

Integrating_Model_Local = LogisticRegression()
X_train_local_integrated = np.vstack((model1.predict_proba(X_local_train1)[:, 1], 
Integrating_Model_Local.fit(X_train_local_integrated, df_local_train['label'])

X_test_local_integrated = np.vstack((model1.predict_proba(X_local_test1)[:, 1], mo
print(Integrating_Model_Local.score(X_test_local_integrated, df_local_test['label'
```

```
0.8130630630630631
```

In [ ]:

```
X_main_train1 = vectorizer1.transform(df_train['parsed_page_description'])
X_main_train2 = vectorizer2.transform(df_train['parsed_link'])
X_main_test1 = vectorizer1.transform(df_test['parsed_page_description'])
X_main_test2 = vectorizer2.transform(df_test['parsed_link'])
model1 = LogisticRegression()
model2 = LogisticRegression()
model1.fit(X_main_train1, df_train['label'])
model2.fit(X_main_train2, df_train['label'])
Integrating_Model = LogisticRegression()
X_train_integrated = np.vstack((model1.predict_proba(X_main_train1)[:, 1], model2.
Integrating_Model.fit(X_train_integrated, df_train['label'])
X_test_integrated = np.vstack((model1.predict_proba(X_main_test1)[:, 1], model2.pr
predictions = Integrating_Model.predict_proba(X_test_integrated)[:, 1]
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('submission_Stanford_pipeline_small_LR.csv')
pred_df.head()
```

```
[0.97719105 0.06781706 0.43791654 ... 0.25765862 0.15169905 0.10314863]
```

Out[ ]:

	label
link_id	
4049	0.977191
3692	0.067817
9739	0.437917
1548	0.882614
5574	0.997273

HyperParamter Tuning Linear Classifiers

Note: - All approaches other than the best will be commented out. This will make the execution of the code faster too

In [ ]:
```
df_local_train, df_local_test = train_test_split(df_train, shuffle=True, test_size
from sklearn.model_selection import StratifiedKFold
```

TF-IDF Optimising

In [ ]:
```
# from sklearn.model_selection import StratifiedKFold
# seed = 7
# np.random.seed(seed)
# kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
# kfold_split = kfold.split(df_train, df_train['label'])

# cross_validation_scores = []
# for train_ids, test_ids in kfold_split:
#       df_local_train = df_train.iloc[list(train_ids)]
#       df_local_test = df_train.iloc[list(test_ids)]

#       vectorizer = TfidfVectorizer()
#       # vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.v
#       vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Da
#       X_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
#       Y_train = df_local_train['label']
#       X_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
#       y_test = df_local_test['label']
#       model = LogisticRegression()
#       model.fit(X_train, Y_train)
#       cross_validation_scores.append(model.score(X_test, y_test))
# print(cross_validation_scores)
# print(np.mean(cross_validation_scores))

# from sklearn.model_selection import StratifiedKFold
# seed = 7
# np.random.seed(seed)
# kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
# kfold_split = kfold.split(df_train, df_train['label'])

# cross_validation_scores = []
# for train_ids, test_ids in kfold_split:
#       df_local_train = df_train.iloc[list(train_ids)]
#       df_local_test = df_train.iloc[list(test_ids)]

#       vectorizer = TfidfVectorizer()
#       # vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.v
#       # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_
#       vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
#       X_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
#       Y_train = df_local_train['label']
#       X_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
#       y_test = df_local_test['label']
#       model = LogisticRegression()
#       model.fit(X_train, Y_train)
#       cross_validation_scores.append(model.score(X_test, y_test))
# print(cross_validation_scores)
# print(np.mean(cross_validation_scores))

# from sklearn.model_selection import StratifiedKFold
# seed = 7
# np.random.seed(seed)
```

```
# kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
# kfold_split = kfold.split(df_train, df_train['label'])

# cross_validation_scores = []
# for train_ids, test_ids in kfold_split:
#       df_local_train = df_train.iloc[list(train_ids)]
#       df_local_test = df_train.iloc[list(test_ids)]

#       vectorizer = TfidfVectorizer()
#       vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.va
#       # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_
#       # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
#       X_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
#       Y_train = df_local_train['label']
#       X_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
#       y_test = df_local_test['label']
#       model = LogisticRegression()
#       model.fit(X_train, Y_train)
#       cross_validation_scores.append(model.score(X_test, y_test))
# print(cross_validation_scores)
# print(np.mean(cross_validation_scores))
```

Not much difference, but when vocabulary (vectorizer.fit_transform()) is trained on entire data(train+test), performance is slightly better.

In [ ]:

```
# vectorizer = TfidfVectorizer(max_df = 0.40)
# vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, d
# # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.v
# X_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
# X_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
# print(len(X_train.toarray()[0]))
```

The above code snippet is for seeing the length of the vocabulary. Through iterations, we see that for entire training data and test data with no max_df and min_df constraints, it is 73168

with min_df = 0.0001, it is 73168(original)

with min_df = 0.001, it is 12401

with min_df = 0.0025, it is 7156

with min_df = 0.005, it is 4532

with min_df = 0.01, it is 2662

with min_df = 0.02, it is 1513

with min_df = 0.03, it is 1000

In [ ]:

```
# from sklearn.model_selection import StratifiedKFold
# seed = 7
# np.random.seed(seed)
# kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
# kfold_split = kfold.split(df_train, df_train['label'])

# cross_validation_scores = []
# for train_ids, test_ids in kfold_split:
#       df_local_train = df_train.iloc[list(train_ids)]
#       df_local_test = df_train.iloc[list(test_ids)]

#       vectorizer = TfidfVectorizer(min_df=0.03)
#       vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.va
#       # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_
#       # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
```

```
#       X_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
#       Y_train = df_local_train['label']
#       X_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
#       y_test = df_local_test['label']
#       model = LogisticRegression()
#       model.fit(X_train, Y_train)
#       cross_validation_scores.append(model.score(X_test, y_test))
# print(cross_validation_scores)
# print(np.mean(cross_validation_scores))
```

Roc Scores: -

with min_df = 0.0001, length(73168)(original) = 0.8066

with min_df = 0.001, length(12401) = 0.8063

with min_df = 0.0025, length(7156) = 0.8036

with min_df = 0.005, length(4532) = 0.8030

with min_df = 0.01, length(2662) = 0.8052

with min_df = 0.02, length(1513) = 0.7996

with min_df = 0.03, length(1000) = 0.7958

Therefore, we should either use the entire vocabulary, or use min_df = 0.01

In [ ]:
```
# from sklearn.model_selection import StratifiedKFold
# seed = 7
# np.random.seed(seed)
# kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
# kfold_split = kfold.split(df_train, df_train['label'])

# cross_validation_scores = []
# for train_ids, test_ids in kfold_split:
#       df_local_train = df_train.iloc[list(train_ids)]
#       df_local_test = df_train.iloc[list(test_ids)]

#       vectorizer = TfidfVectorizer(max_df=0.50)
#       vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.va
#       # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_
#       # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
#       X_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
#       Y_train = df_local_train['label']
#       X_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
#       y_test = df_local_test['label']
#       model = LogisticRegression()
#       model.fit(X_train, Y_train)
#       cross_validation_scores.append(model.score(X_test, y_test))
# print(cross_validation_scores)
# print(np.mean(cross_validation_scores))
```

Therefore, we shouldn't use max_df either. It leads to a decrease of performance.

Logisitic Regression Optimising

In [ ]:
```
from sklearn.model_selection import RandomizedSearchCV
```

In [ ]:
```
# from sklearn.model_selection import StratifiedKFold
# seed = 7
# np.random.seed(seed)
# kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
# kfold_split = kfold.split(df_train, df_train['label'])
```

```
# cross_validation_scores = []
# # cross_validation_params = []
# for train_ids, test_ids in kfold_split:
#       df_local_train = df_train.iloc[list(train_ids)]
#       df_local_test = df_train.iloc[list(test_ids)]

#       vectorizer = TfidfVectorizer()
#       vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.va
#       # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_
#       # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
#       X_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
#       Y_train = df_local_train['label']
#       X_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
#       y_test = df_local_test['label']

#       solver = ['lbfgs', 'newton-cg', 'liblinear']
#       warm_start = [True, False]
#       C = np.arange(0, 1, 0.01)
#       random_grid ={
#               'warm_start' : warm_start,
#               'solver' : solver,
#               'C' : C,
#       }
#       # estimator = LogisticRegression(
#       # random_state = 1,
#       # penalty = 'l2'
#       # )
#       estimator = LogisticRegression()
#       random_estimator = RandomizedSearchCV(estimator = estimator,
#                                    param_distributions = random_grid,
#                                    n_iter = 10,
#                                    scoring = 'accuracy',
#                                    n_jobs = -1,
#                                    verbose = 1,
#                                    random_state = 1,
#                                   )
#       random_estimator.fit(X_train, Y_train)
#       best_estimator = random_estimator.best_estimator_
#       cross_validation_scores.append(best_estimator.score(X_test, y_test))
#       print(random_estimator.best_params_)
# print(cross_validation_scores)
# print(np.mean(cross_validation_scores))
```

Therefore, we can see it is making the performance better. So now we will make a submission file for it by first doing this for the whole data set.

BEST MODEL SO FAR

In [ ]:

```
vectorizer = TfidfVectorizer()
vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, df_
# vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.va
# vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
X_train = vectorizer.transform(df_train['Complete_Textual_Data'])
Y_train = df_train['label']
X_test = vectorizer.transform(df_test['Complete_Textual_Data'])
# y_test = df_test['label']

max_iter = range(100, 500)
solver = ['lbfgs', 'newton-cg', 'liblinear']
warm_start = [True, False]
C = np.arange(0, 1, 0.01)
random_grid ={
```

```
        'warm_start' : warm_start,
        'solver' : solver,
        'C' : C,
}
estimator = LogisticRegression()
random_estimator = RandomizedSearchCV(estimator = estimator,
                                      param_distributions = random_grid,
                                      n_iter = 10,
                                      scoring = 'accuracy',
                                      n_jobs = -1,
                                      verbose = 1,
                                      random_state = 1,
                                     )
random_estimator.fit(X_train, Y_train)
best_estimator = random_estimator.best_estimator_
# print(best_estimator.score(X_test, y_test))
print(random_estimator.best_params_)

predictions = best_estimator.predict_proba(X_test)[:, 1]
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('LR_HP_Tuned.csv')
pred_df.head()
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
{'warm_start': True, 'solver': 'newton-cg', 'C': 0.91}
[0.87727299 0.23241382 0.41605581 ... 0.29071209 0.28871812 0.27832926]
```

Out[ ]:

	label
link_id	
4049	0.877273
3692	0.232414
9739	0.416056
1548	0.527426
5574	0.912577

Random Forest Tuning

In [ ]:

```
# vectorizer = TfidfVectorizer(min_df=0.01)
# vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, d
# # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.v
# # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
# X_train = vectorizer.transform(df_train['Complete_Textual_Data'])
# Y_train = df_train['label']
# X_test = vectorizer.transform(df_test['Complete_Textual_Data'])
# # y_test = df_local_test['label']

# # Number of trees in Random Forest
# # rf_n_estimators = [int(x) for x in np.linspace(10, 20, 2)]
# rf_n_estimators = [10]
# # rf_n_estimators.append(1500)
# # rf_n_estimators.append(2000)

# # Maximum number of levels in tree
# # rf_max_depth = [int(x) for x in np.linspace(2, 4, 2)]
# rf_max_depth = [2, 4]
# # Add the default as a possible value
# rf_max_depth.append(None)
```

```
# # Number of features to consider at every split
# rf_max_features = ['auto', 'sqrt', 'log2']

# # # Criterion to split on
# rf_criterion = ['mse', 'mae']

# # Minimum number of samples required to split a node
# # rf_min_samples_split = [int(x) for x in np.linspace(2, 5, 2)]
# rf_min_samples_split = [2, 5]

# # Minimum number of samples required at each leaf node
# min_samples_leaf = [1, 2]

# # Minimum decrease in impurity required for split to happen
# rf_min_impurity_decrease = [0.0, 0.05, 0.1]

# # Method of selecting samples for training each tree
# rf_bootstrap = [True, False]

# # Create the grid
# rf_grid = {'n_estimators': rf_n_estimators,
#                         'max_depth': rf_max_depth,
#                         "criterion": rf_criterion,
#                         'max_features': rf_max_features,
#                         'bootstrap': rf_bootstrap}

# rf_base = RandomForestRegressor()
# random_estimator = RandomizedSearchCV(estimator = rf_base, param_distributions = 
#                                                                 n_iter = 5, cv = 5, verbos
#                                                                 n_jobs = -1)
# # random_estimator = GridSearchCV(estimator = rf_base, param_distributions = rf_
# #                                                               n_iter = 5, cv = 5, verbos
# #                                                               n_jobs = -1)
# random_estimator.fit(X_train, Y_train)
# best_estimator = random_estimator.best_estimator_
# # print(best_estimator.score(X_test, y_test))

# vectorizer = TfidfVectorizer(min_df=0.01)
# vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, 
# # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.
# # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
# X_train = vectorizer.transform(df_train['Complete_Textual_Data'])
# Y_train = df_train['label']
# X_test = vectorizer.transform(df_test['Complete_Textual_Data'])
# # y_test = df_local_test['label']

# rf = RandomForestRegressor(max_depth=10)
# rf.fit(X_train, Y_train)

# df_local_train, df_local_test = train_test_split(df_train, shuffle=True, test_si

# vectorizer = TfidfVectorizer()
# vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, 
# # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.
# # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
# X_train = vectorizer.transform(df_local_train['Complete_Textual_Data'])
# Y_train = df_local_train['label']
# X_test = vectorizer.transform(df_local_test['Complete_Textual_Data'])
# Y_test = df_local_test['label']

# rf = RandomForestRegressor()
# rf.fit(X_train, Y_train)
# # print(rf.score(X_test, Y_test))
```

```
# predictions = rf.predict(X_test)
# print(predictions)
# print(roc_auc_score(Y_test, predictions))

# vectorizer = TfidfVectorizer()
# vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, d
# # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.v
# # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
# X_train = vectorizer.transform(df_train['Complete_Textual_Data'])
# Y_train = df_train['label']
# X_test = vectorizer.transform(df_test['Complete_Textual_Data'])
# # y_test = df_local_test['label']

# rf = RandomForestRegressor()
# rf.fit(X_train, Y_train)

# predictions = rf.predict(X_test)
# print(predictions)

# pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
# pred_df.to_csv('RF_HP_1.csv')
# pred_df.head()

# vectorizer = TfidfVectorizer()
# vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, d
# # vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.v
# # vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
# X_train = vectorizer.transform(df_train['Complete_Textual_Data'])
# Y_train = df_train['label']
# X_test = vectorizer.transform(df_test['Complete_Textual_Data'])
# # y_test = df_local_test['label']

# # Number of trees in Random Forest
# # rf_n_estimators = [int(x) for x in np.linspace(10, 20, 2)]
# rf_n_estimators = [int(x) for x in np.linspace(10, 40, 4)]
# # rf_n_estimators.append(1500)
# # rf_n_estimators.append(2000)

# # Maximum number of levels in tree
# # rf_max_depth = [int(x) for x in np.linspace(2, 4, 2)]
# # rf_max_depth = [int(x) for x in np.linspace(2, 6, 2)]
# rf_max_depth = [2, 4, 6]
# # Add the default as a possible value
# rf_max_depth.append(None)

# # Number of features to consider at every split
# rf_max_features = ['auto', 'sqrt', 'log2']

# # # Criterion to split on
# rf_criterion = ['mse', 'mae']

# # Minimum number of samples required to split a node
# # rf_min_samples_split = [int(x) for x in np.linspace(2, 5, 2)]
# rf_min_samples_split = [2, 5]

# # Minimum number of samples required at each leaf node
# min_samples_leaf = [1, 2]

# # Minimum decrease in impurity required for split to happen
# rf_min_impurity_decrease = [0.0, 0.05, 0.1]

# # Method of selecting samples for training each tree
# rf_bootstrap = [True, False]
```

```
# min_samples_split = [2, 5]
# # Minimum number of samples required at each leaf node
# min_samples_leaf = [1, 2]
# # Create the grid
# rf_grid = {'n_estimators': rf_n_estimators,
#                        'max_depth': rf_max_depth,
#                        "criterion": rf_criterion,
#                        'max_features': rf_max_features,
#                        'min_samples_split': min_samples_split,
#                        'min_samples_leaf': min_samples_leaf,
#                        'min_impurity_decrease': rf_min_impurity_decrease,
#                        'bootstrap': rf_bootstrap}

# rf_base = RandomForestRegressor()
# random_estimator = RandomizedSearchCV(estimator = rf_base, param_distributions = 
#                                                         n_iter = 10, cv = 5, verbo
#                                                         n_jobs = -1)
# # random_estimator = GridSearchCV(estimator = rf_base, param_distributions = rf_g
# #                                                       n_iter = 5, cv = 5, verbose
# #                                                       n_jobs = -1)
# random_estimator.fit(X_train, Y_train)
# best_estimator = random_estimator.best_estimator_
# # print(best_estimator.score(X_test, y_test))

# print(random_estimator.best_params_)

# predictions = best_estimator.predict(X_test)
# print(predictions)
# pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
# pred_df.to_csv('RF_HP_2.csv')
# pred_df.head()

vectorizer = TfidfVectorizer()
vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, df_
# vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.va
# vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
X_train = vectorizer.transform(df_train['Complete_Textual_Data'])
Y_train = df_train['label']
X_test = vectorizer.transform(df_test['Complete_Textual_Data'])
# y_test = df_test['label']

rf = RandomForestRegressor()
rf.fit(X_train, Y_train)

predictions = rf.predict(X_test)
print(predictions)

pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('RF_HP_new.csv')
pred_df.head()
```

```
[1.   0.21 0.3  ... 0.05 0.27 0.08]
```

Out[ ]:

	label
link_id	
4049	1.00
3692	0.21
9739	0.30
1548	0.62
5574	0.97

Neural Networks

Note: - All approaches, but the best/best 2 have been commented

In [ ]:
```
df_train_with_validation, df_validation = train_test_split(df_train, test_size=0.2

import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
```

In [ ]:
```
def batch_generator_shuffle(X_data, y_data, batch_size):
    samples_per_epoch = X_data.shape[0]
    number_of_batches = samples_per_epoch/batch_size
    counter=0
    index = np.arange(np.shape(y_data)[0])
    np.random.shuffle(index)
    while 1:
        index_batch = index[batch_size*counter:batch_size*(counter+1)]
        X_batch = X_data[index_batch,:].toarray()
        y_batch = y_data[y_data.index[index_batch]]
        counter += 1
        yield np.array(X_batch),y_batch
        if (counter > number_of_batches):
            np.random.shuffle(index)
            counter=0

def batch_generator(X_data, y_data, batch_size):
    samples_per_epoch = X_data.shape[0]
    number_of_batches = samples_per_epoch/batch_size
    counter=0
    index = np.arange(np.shape(y_data)[0])
    while 1:
        index_batch = index[batch_size*counter:batch_size*(counter+1)]
        X_batch = X_data[index_batch,:].todense()
        # y_batch = y_data[y_data.index[index_batch]]
        y_batch = y_data[index_batch]
        counter += 1
        yield(np.array(X_batch),y_batch)
        if (counter > number_of_batches):
            counter=0
```

In [ ]:
```
vectorizer = TfidfVectorizer(max_features=500)
# vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, d
vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, df_
```

```
X_train = vectorizer.transform(df_train_with_validation['Complete_Textual_Data'])
X_validation = vectorizer.transform(df_validation['Complete_Textual_Data'])
X_test = vectorizer.transform(df_test['Complete_Textual_Data'])
X_train_2 = X_train.todense()
X_test_2 = X_test.todense()
X_validation_2 = X_validation.todense()

# vectorizer_new = TfidfVectorizer(max_features=500)
# vectorizer_new.fit_transform(np.concatenate((df_train.text_title.values, df_test
# X_train_new = vectorizer_new.transform(df_train_with_validation['text_title'])
# X_validation_new = vectorizer_new.transform(df_validation['text_title'])
# X_test_new = vectorizer_new.transform(df_test['text_title'])
# X_train_2_new = X_train_new.todense()
# X_test_2_new = X_test_new.todense()
# X_validation_2_new = X_validation_new.todense()
# X_train_main_new = vectorizer_new.transform(df_train['text_title'])
# X_train_main_2_new = X_train_main_new.todense()
```

In [ ]:
```
Tfidf_Vector_length = len(X_validation.toarray()[0])
print(type(X_test))
print(Tfidf_Vector_length)

# Tfidf_Vector_length_2 = len(X_validation_new.toarray()[0])
# print(type(X_test_new))
# print(Tfidf_Vector_length_2)
```

```
<class 'scipy.sparse.csr.csr_matrix'>
500
```

In [ ]:
```
# # def batch_generator(X_train, Y_train):

# #     while True:
# #          # samples_counter = 0
# #         for fl, lb in zip(X_train, Y_train):
# #             sam, lam = get_IQsamples(fl, lb)
# #             max_iter = sam.shape[0]
# #             sample = []  # store all the generated data batches
# #             label = [] # store all the generated label batches
# #             i = 0
# #             for d, l in zip(sam, lam):
# #                 sample.append(d)
# #                 label.append(l)
# #                 i += 1
# #                 if i == max_iter:
# #                     break
# #             sample = np.asarray(sample)
# #             label = np.asarray(label)
# #             yield sample, label

# model = Sequential()
# model.add(Dense(64, activation='relu', input_dim=Tfidf_Vector_length))
# model.add(Dense(1, activation='sigmoid'))
# model.compile(optimizer='adam',
#               loss='binary_crossentropy',
#               metrics=['accuracy'])

# # model.fit(batch_generator_shuffle(X_train, df_train_with_validation['label'], 1
# #                     epochs=5, validation_data=(X_validation, df_validation['lal
# #                     steps_per_epoch=X_train.shape[0]/1)
# generator_input = tf.data.Dataset.from_generator(batch_generator(X_train, df_tra
# # generator_input = batch_generator(X_train, df_train_with_validation['label'], 1
# model.fit(generator_input,
#                     epochs=5,
#                     steps_per_epoch=X_train.shape[0]/1, shuffle=True)
```

```
# # print(batch_generator(X_train, df_train_with_validation['label'], 1))
# # model.fit(batch_generator(), df_train_with_validation['label'], epochs=5, steps

# def build_model():
#     model = Sequential()
#     model.add(Dense(64, activation='relu', input_dim=Tfidf_Vector_length))
#     model.add(Dense(1, activation='sigmoid'))
#     model.compile(optimizer='adam',
#                   loss='binary_crossentropy',
#                   metrics=['accuracy'])
#     return model

# # warnings.filterwarnings("ignore", category=PendingDeprecationWarning)
# # def convert_sparse_matrix_to_sparse_tensor(X):
# #     coo = X.tocoo()
# #     indices = np.mat([coo.row, coo.col]).transpose()
# #     return tf.SparseTensor(indices, coo.data, coo.shape)

# from keras.wrappers.scikit_learn import KerasClassifier
# estimator = KerasClassifier(build_fn=build_model, epochs=2, batch_size=10)
# # estimator.fit(tf.sparse.reorder(convert_sparse_matrix_to_sparse_tensor(X_train,
# estimator.fit(X_train_2, df_train_with_validation['label'])

# predictions = estimator.predict(X_test_2)
# predictions=predictions.flatten()
# print(predictions)

# pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
# pred_df.to_csv('NN_Submission_5.csv')
# pred_df.head()

# def build_model2():
#     model = Sequential()
#     # model.add(Dense(64, input_dim=Tfidf_Vector_length, activation='relu'))
#     # model.add(Dropout(0.5))
#     # model.add(Dense(200, activation='relu'))
#     # model.add(Dropout(0.5))
#     # model.add(Dense(160, activation='relu'))
#     # model.add(Dropout(0.5))
#     # model.add(Dense(120, activation='relu'))
#     # model.add(Dropout(0.5))
#     # model.add(Dense(80, activation='relu'))
#     # model.add(Dropout(0.5))
#     model.add(Dense(4, input_dim=Tfidf_Vector_length, activation='relu'))
#     model.add(Dropout(0.3))
#     # model.add(Dense(200, activation='relu'))
#     # model.add(Dropout(0.5))
#     model.add(Dense(10, activation='relu'))
#     model.add(Dropout(0.3))
#     model.add(Dense(1, activation='sigmoid'))
#     model.compile(optimizer='adam',
#                   loss='binary_crossentropy',
#                   metrics=['accuracy'])
#     return model

# from keras.wrappers.scikit_learn import KerasClassifier
# estimator = KerasClassifier(build_fn=build_model2, epochs=10, batch_size=20)

# # from sklearn.model_selection import RepeatedKFold, cross_val_score
# # kfold= RepeatedKFold(n_splits=5, n_repeats=2)
# # results=cross_val_score(estimator, X_train_main_2_new, df_train['label'], cv=kf
# # results.mean()

# estimator.fit(X_train_2, df_train_with_validation['label'], validation_data=(X_va
```

```
# # predictions = estimator.predict_proba(X_test_2)[:, 1]
# # #predictions=predictions.flatten()
# # print(predictions)

# # pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
# # pred_df.to_csv('NN_Submission_9.csv')
# # pred_df.head()
```

Model 1: -

In [ ]:
```
import torch
import gc
torch.cuda.empty_cache()
gc.collect()
```

Out[ ]: 22

In [ ]:
```
# predictions = estimator.predict_proba(X_test_2)[:, 1]
# print(predictions)
# pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
# pred_df.to_csv('NN_Submission_11.csv')
# pred_df.head()
```

In [ ]:
```
def build_model3():
    model = Sequential()
    model.add(Dense(4, input_dim=Tfidf_Vector_length_2, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(10, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

feature_name = "Complete_Textual_Data"
# vectorizer_new = TfidfVectorizer(max_features=500)
vectorizer_new = TfidfVectorizer(min_df=0.002)
vectorizer_new.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values
# vectorizer_new.fit_transform(np.concatenate((df_train[feature_name], df_test[feat
X_train_new = vectorizer_new.transform(df_train_with_validation[feature_name])
X_validation_new = vectorizer_new.transform(df_validation[feature_name])
X_test_new = vectorizer_new.transform(df_test[feature_name])
X_train_2_new = X_train_new.todense()
X_test_2_new = X_test_new.todense()
X_validation_2_new = X_validation_new.todense()
X_train_main_new = vectorizer_new.transform(df_train[feature_name])
X_train_main_2_new = X_train_main_new.todense()

Tfidf_Vector_length_2 = len(X_validation_new.toarray()[0])
print(type(X_test_new))
print(Tfidf_Vector_length_2)

from sklearn.model_selection import StratifiedKFold
seed = 7
np.random.seed(seed)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
kfold_split = kfold.split(X_train_main_2_new, df_train['label'])
# print(len(kfold_split))
cvscores = []
```

```
# for train, test in kfold_split:
#     # model = Sequential()
#     # model.add(Dense(4, input_dim=Tfidf_Vector_length, activation='relu'))
#     # model.add(Dropout(0.3))
#     # model.add(Dense(10, activation='relu'))
#     # model.add(Dropout(0.3))
#     # model.add(Dense(1, activation='sigmoid'))
#     # model.compile(optimizer='adam',
#     #               loss='binary_crossentropy',
#     #               metrics=['accuracy'])
#     # # Fit the model
#     # model.fit(X_train_main_2_new[train], df_train['label'][train], epochs=150,
#     # # evaluate the model
#     # scores = model.evaluate(X[test], Y[test], verbose=0)
#     # print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
#     # cvscores.append(scores[1] * 100)

#     estimator = KerasClassifier(build_fn=build_model3, epochs=10, batch_size=20)
#     estimator.fit(X_train_main_2_new[train], df_train['label'][train])
#     scores = estimator.evaluate(X_train_main_2_new[train], df_train['label'][tra
#     print("%s: %.2f%%" % (estimator.metrics_names[1], scores[1]*100))
#     cvscores.append(scores[1] * 100)

# print("%.2f%% (+/- %.2f%%)" % (np.mean(cvscores), np.std(cvscores)))

from sklearn.model_selection import KFold
from keras.wrappers.scikit_learn import KerasClassifier
estimator = KerasClassifier(build_fn=build_model3, epochs=3, batch_size=8)
kfold = KFold(n_splits=10)
scoring = ['accuracy', 'precision', 'recall', 'f1']
results = cross_validate(estimator=estimator,
                               X=X_train_main_2_new,
                               y=df_train['label'],
                               cv=kfold,
                               scoring=scoring,
                               return_train_score=True,
                              return_estimator=True)
print(results['test_accuracy'], results['test_precision'], results['test_recall'],
print((results['test_accuracy'] + results['test_precision'] + results['test_recall
print(np.mean((results['test_accuracy'] + results['test_precision'] + results['tes

feature_name = "Complete_Textual_Data"
# vectorizer_new = TfidfVectorizer(max_features=500)
vectorizer_new = TfidfVectorizer(min_df=0.002)
vectorizer_new.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values
# vectorizer_new.fit_transform(np.concatenate((df_train[feature_name], df_test[fea
X_train_new = vectorizer_new.transform(df_train_with_validation[feature_name])
X_validation_new = vectorizer_new.transform(df_validation[feature_name])
X_test_new = vectorizer_new.transform(df_test[feature_name])
X_train_2_new = X_train_new.todense()
X_test_2_new = X_test_new.todense()
X_validation_2_new = X_validation_new.todense()
X_train_main_new = vectorizer_new.transform(df_train[feature_name])
X_train_main_2_new = X_train_main_new.todense()

Tfidf_Vector_length_2 = len(X_validation_new.toarray()[0])
print(type(X_test_new))
print(Tfidf_Vector_length_2)

estimator = results['estimator'][6]
predictions = estimator.predict_proba(X_test_2_new)[:, 1]
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
```

```
pred_df.to_csv('NN_Submission_HP_Training.csv')
pred_df.head()
```

```
<class 'scipy.sparse.csr.csr_matrix'>
8360
Epoch 1/3
500/500 [==============================] - 4s 3ms/step - loss: 0.5907 - accuracy:
0.7388
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4601 - accuracy:
0.8177
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.3979 - accuracy:
0.8588
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.6140 - accuracy:
0.6609
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4979 - accuracy:
0.7826
Epoch 3/3
500/500 [==============================] - 2s 3ms/step - loss: 0.4538 - accuracy:
0.8157
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.6086 - accuracy:
0.7265
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4962 - accuracy:
0.8109
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4268 - accuracy:
0.8435
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.6410 - accuracy:
0.7037
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.5162 - accuracy:
0.7951
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4279 - accuracy:
0.8350
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.6219 - accuracy:
0.7173
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.5001 - accuracy:
0.8129
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4325 - accuracy:
0.8487
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.6338 - accuracy:
0.7135
Epoch 2/3
500/500 [==============================] - 2s 3ms/step - loss: 0.5196 - accuracy:
0.8152
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4288 - accuracy:
0.8442
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.5844 - accuracy:
0.7223
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4718 - accuracy:
0.8109
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4067 - accuracy:
```

```
0.8497
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.6181 - accuracy:
0.6642
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.5185 - accuracy:
0.8187
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4599 - accuracy:
0.8465
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.6058 - accuracy:
0.6925
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4806 - accuracy:
0.7979
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4194 - accuracy:
0.8493
Epoch 1/3
500/500 [==============================] - 2s 3ms/step - loss: 0.5905 - accuracy:
0.6943
Epoch 2/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4725 - accuracy:
0.8142
Epoch 3/3
500/500 [==============================] - 1s 3ms/step - loss: 0.4164 - accuracy:
0.8458
[0.81306306 0.81306306 0.79954955 0.83558559 0.80630631 0.81081081
 0.82432432 0.79458239 0.79232506 0.79006772] [0.85446009 0.84615385 0.77777778 0.
83084577 0.81018519 0.84978541
 0.84474886 0.88669951 0.82380952 0.84455959] [0.77777778 0.79237288 0.80382775 0.
81067961 0.79545455 0.80161943
 0.80786026 0.72580645 0.75877193 0.72123894] [0.81431767 0.81838074 0.79058824 0.
82063882 0.80275229 0.825
 0.82589286 0.79822616 0.78995434 0.77804296]
[0.81490465 0.81749263 0.79293583 0.82443745 0.80367458 0.82180391
 0.82570658 0.80132863 0.79121521 0.7834773 ]
0.8076976774144491
<class 'scipy.sparse.csr.csr_matrix'>
8360
[0.8952797  0.15240823 0.32218686 ... 0.1845768  0.23921704 0.201459  ]
```

Out[ ]:

	label
link_id	
4049	0.895280
3692	0.152408
9739	0.322187
1548	0.726858
5574	0.899557

Model 2: -

In [ ]:

```
feature_name = "Complete_Textual_Data"
# vectorizer_new = TfidfVectorizer(max_features=500)
vectorizer_new = TfidfVectorizer(min_df=0.01)
vectorizer_new.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values
# vectorizer_new.fit_transform(np.concatenate((df_train[feature_name], df_test[feat
```

```
X_train_new = vectorizer_new.transform(df_train_with_validation[feature_name])
X_validation_new = vectorizer_new.transform(df_validation[feature_name])
X_test_new = vectorizer_new.transform(df_test[feature_name])
X_train_2_new = X_train_new.todense()
X_test_2_new = X_test_new.todense()
X_validation_2_new = X_validation_new.todense()
X_train_main_new = vectorizer_new.transform(df_train[feature_name])
X_train_main_2_new = X_train_main_new.todense()

Tfidf_Vector_length_2 = len(X_validation_new.toarray()[0])
print(type(X_test_new))
print(Tfidf_Vector_length_2)

def build_model4():
    model = Sequential()
    model.add(Dense(16, input_dim=Tfidf_Vector_length_2, activation='relu'))
    model.add(Dropout(0.5))
    # model.add(Dense(10, activation='relu'))
    # model.add(Dropout(0.3))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

from keras.wrappers.scikit_learn import KerasClassifier
estimator = KerasClassifier(build_fn=build_model4, epochs=5, batch_size=20)
estimator.fit(X_train_main_2_new, df_train['label'])
predictions = estimator.predict_proba(X_test_2_new)[:, 1]
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('NN_Submission_HP_Training.csv')
pred_df.head()
```

```
<class 'scipy.sparse.csr.csr_matrix'>
2662
Epoch 1/5
222/222 [==============================] - 1s 3ms/step - loss: 0.5817 - accuracy:
0.7262
Epoch 2/5
222/222 [==============================] - 1s 3ms/step - loss: 0.4717 - accuracy:
0.8138
Epoch 3/5
222/222 [==============================] - 1s 3ms/step - loss: 0.4409 - accuracy:
0.8177
Epoch 4/5
222/222 [==============================] - 1s 3ms/step - loss: 0.4192 - accuracy:
0.8359
Epoch 5/5
222/222 [==============================] - 1s 3ms/step - loss: 0.4029 - accuracy:
0.8447
[0.92730993 0.17934395 0.38999224 ... 0.24027644 0.25496772 0.28894147]
```

Out[ ]:

link_id	label
4049	0.927310
3692	0.179344
9739	0.389992
1548	0.730503
5574	0.951353

Transfer Learning

Note: - All approaches but the best have been commented.

In [ ]:

```
# import torch
# import torch.nn as nn
# import transformers
# from transformers import AutoModel, BertTokenizerFast
# device = torch.device("cuda")

# bert = AutoModel.from_pretrained('bert-base-uncased', return_dict=False)
# tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')

# df_train.head()

# Text_list = df_train["Complete_Textual_Data"].tolist() + df_test["Complete_Textua
# seq_len = [len(i.split()) for i in Text_list]
# pd.Series(seq_len).hist(bins = 30)

# max_useful_length = 100

# train_text, temp_text, train_labels, temp_labels = train_test_split(df_train['Co
#                                                                     random_state=
#                                                                     test_size=0.
#                                                                     stratify=df_

# # we will use temp_text and temp_labels to create validation and test set
# val_text, test_text, val_labels, test_labels = train_test_split(temp_text, temp_
#                                                                 random_state=201
#                                                                 test_size=0.5,
#                                                                 stratify=temp_la

# # tokenize and encode sequences in the training set
# tokens_train = tokenizer.batch_encode_plus(
#     train_text.tolist(),
#     max_length = max_useful_length,
#     pad_to_max_length=True,
#     truncation=True,
#     return_token_type_ids=False
# )

# # tokenize and encode sequences in the validation set
# tokens_val = tokenizer.batch_encode_plus(
#     val_text.tolist(),
#     max_length = max_useful_length,
#     pad_to_max_length=True,
#     truncation=True,
```

```
#     return_token_type_ids=False
# )

# # tokenize and encode sequences in the test set
# tokens_test = tokenizer.batch_encode_plus(
#     test_text.tolist(),
#     max_length = max_useful_length,
#     pad_to_max_length=True,
#     truncation=True,
#     return_token_type_ids=False
# )

# # for train set
# train_seq = torch.tensor(tokens_train['input_ids'])
# train_mask = torch.tensor(tokens_train['attention_mask'])
# train_y = torch.tensor(train_labels.tolist())

# # for validation set
# val_seq = torch.tensor(tokens_val['input_ids'])
# val_mask = torch.tensor(tokens_val['attention_mask'])
# val_y = torch.tensor(val_labels.tolist())

# # for test set
# test_seq = torch.tensor(tokens_test['input_ids'])
# test_mask = torch.tensor(tokens_test['attention_mask'])
# test_y = torch.tensor(test_labels.tolist())

# from torch.utils.data import TensorDataset, DataLoader, RandomSampler, Sequentia

# #define a batch size
# batch_size = 32

# # wrap tensors
# train_data = TensorDataset(train_seq, train_mask, train_y)

# # sampler for sampling the data during training
# train_sampler = RandomSampler(train_data)

# # dataLoader for train set
# train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=batcl

# # wrap tensors
# val_data = TensorDataset(val_seq, val_mask, val_y)

# # sampler for sampling the data during training
# val_sampler = SequentialSampler(val_data)

# # dataLoader for validation set
# val_dataloader = DataLoader(val_data, sampler = val_sampler, batch_size=batch_si

# for param in bert.parameters():
#     param.requires_grad = False

# class BERT_Arch(nn.Module):

#     def __init__(self, bert):

#       super(BERT_Arch, self).__init__()

#       self.bert = bert

#       # dropout layer
#       self.dropout = nn.Dropout(0.1)
```

```
#       # relu activation function
#       self.relu =  nn.ReLU()

#       # dense layer 1
#       self.fc1 = nn.Linear(768,512)

#       # dense layer 2 (Output layer)
#       self.fc2 = nn.Linear(512,2)

#       #softmax activation function
#       self.softmax = nn.LogSoftmax(dim=1)

#     #define the forward pass
#     def forward(self, sent_id, mask):

#       #pass the inputs to the model
#       _, cls_hs = self.bert(sent_id, attention_mask=mask)

#       x = self.fc1(cls_hs)

#       x = self.relu(x)

#       x = self.dropout(x)

#       # output layer
#       x = self.fc2(x)

#       # apply softmax activation
#       x = self.softmax(x)

#       return x

# # pass the pre-trained BERT to our define architecture
# model = BERT_Arch(bert)

# # push the model to GPU
# model = model.to(device)

# # optimizer from hugging face transformers
# from transformers import AdamW

# # define the optimizer
# optimizer = AdamW(model.parameters(), lr = 1e-3)

# from sklearn.utils.class_weight import compute_class_weight

# #compute the class weights
# class_wts = compute_class_weight('balanced', classes=np.unique(train_labels), y=t

# print(class_wts)

# # convert class weights to tensor
# weights= torch.tensor(class_wts,dtype=torch.float)
# weights = weights.to(device)

# # loss function
# cross_entropy  = nn.NLLLoss(weight=weights)

# # number of training epochs
epochs = 1

# # function to train the model
# def train():
```

```
#   model.train()

#   total_loss, total_accuracy = 0, 0

#   # empty list to save model predictions
#   total_preds=[]

#   # iterate over batches
#   for step,batch in enumerate(train_dataloader):

#     # progress update after every 50 batches.
#     if step % 50 == 0 and not step == 0:
#       print('  Batch {:>5,}  of  {:>5,}.'.format(step, len(train_dataloader)))

#     # push the batch to gpu
#     batch = [r.to(device) for r in batch]

#     sent_id, mask, labels = batch

#     # clear previously calculated gradients
#     model.zero_grad()

#     # get model predictions for the current batch
#     preds = model(sent_id, mask)

#     # compute the loss between actual and predicted values
#     loss = cross_entropy(preds, labels)

#     # add on to the total loss
#     total_loss = total_loss + loss.item()

#     # backward pass to calculate the gradients
#     loss.backward()

#     # clip the the gradients to 1.0. It helps in preventing the exploding gradie
#     torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

#     # update parameters
#     optimizer.step()

#     # model predictions are stored on GPU. So, push it to CPU
#     preds=preds.detach().cpu().numpy()

#     # append the model predictions
#     total_preds.append(preds)

#   # compute the training loss of the epoch
#   avg_loss = total_loss / len(train_dataloader)

#   # predictions are in the form of (no. of batches, size of batch, no. of classes
#   # reshape the predictions in form of (number of samples, no. of classes)
#   total_preds  = np.concatenate(total_preds, axis=0)

#   #returns the loss and predictions
#   return avg_loss, total_preds

# # function for evaluating the model
# def evaluate():

#   print("\nEvaluating...")

#   # deactivate dropout layers
#   model.eval()
```

```
#   total_loss, total_accuracy = 0, 0

#   # empty list to save the model predictions
#   total_preds = []

#   # iterate over batches
#   for step,batch in enumerate(val_dataloader):

#     # Progress update every 50 batches.
#     if step % 50 == 0 and not step == 0:

#       # Calculate elapsed time in minutes.
#       elapsed = format_time(time.time() - t0)

#       # Report progress.
#       print('  Batch {:>5,}  of  {:>5,}.'.format(step, len(val_dataloader)))

#     # push the batch to gpu
#     batch = [t.to(device) for t in batch]

#     sent_id, mask, labels = batch

#     # deactivate autograd
#     with torch.no_grad():

#       # model predictions
#       preds = model(sent_id, mask)

#       # compute the validation loss between actual and predicted values
#       loss = cross_entropy(preds,labels)

#       total_loss = total_loss + loss.item()

#       preds = preds.detach().cpu().numpy()

#       total_preds.append(preds)

#   # compute the validation loss of the epoch
#   avg_loss = total_loss / len(val_dataloader)

#   # reshape the predictions in form of (number of samples, no. of classes)
#   total_preds  = np.concatenate(total_preds, axis=0)

#   return avg_loss, total_preds

# # set initial loss to infinite
# best_valid_loss = float('inf')

# # empty lists to store training and validation loss of each epoch
# train_losses=[]
# valid_losses=[]

# #for each epoch
# for epoch in range(epochs):

#     print('\n Epoch {:} / {:}'.format(epoch + 1, epochs))

#     #train model
#     train_loss, _ = train()

#     #evaluate model
#     valid_loss, _ = evaluate()

#     #save the best model
```

```
#     if valid_loss < best_valid_loss:
#         best_valid_loss = valid_loss
#         torch.save(model.state_dict(), 'saved_weights.pt')

#     # append training and validation loss
#     train_losses.append(train_loss)
#     valid_losses.append(valid_loss)

#     print(f'\nTraining Loss: {train_loss:.3f}')
#     print(f'Validation Loss: {valid_loss:.3f}')

# path = 'saved_weights.pt'
# model.load_state_dict(torch.load(path))

# import tensorflow as tf
# import tensorflow_hub as hub
# from datetime import datetime
# import bert
# from bert import run_classifier
# from bert import optimization
# from bert import tokenization

# !python -m pip install tensorflow-text
# import tensorflow as tf
# import tensorflow_text as text
# import tensorflow_hub as hub

# import tensorflow as tf
# import tensorflow_hub as hub

# !python -m pip install -q -U "tensorflow-text==2.8.*" --user

# import tensorflow as tf
# import tensorflow_hub as hub
# import tensorflow_text as text

# bert_preprocess = hub.KerasLayer("https://tfhub.dev/tensorflow/bert_en_uncased_pr
# bert_encoder = hub.KerasLayer("https://tfhub.dev/tensorflow/bert_en_uncased_L-12_

# def get_sentence_embeding(sentences):
#     preprocessed_text = bert_preprocess(sentences)
#     return bert_encoder(preprocessed_text)['pooled_output']

# from sklearn.metrics.pairwise import cosine_similarity

# text_input = tf.keras.layers.Input(shape=(), dtype=tf.string, name='text')
# preprocessed_text = bert_preprocess(text_input)
# outputs = bert_encoder(preprocessed_text)

# l = tf.keras.layers.Dropout(0.1, name="dropout")(outputs['pooled_output'])
# l = tf.keras.layers.Dense(1, activation='sigmoid', name="output")(l)

# model = tf.keras.Model(inputs=[text_input], outputs = [l])

# model.summary()

# METRICS=[tf.keras.metrics.BinaryAccuracy(name='accuracy'),
#       tf.keras.metrics.Precision(name='precision'),
#       tf.keras.metrics.Recall(name='recall')]

# model.compile(optimizer='adam',
#               loss='binary_crossentropy',
#               metrics=METRICS)
```

```
# X_train, X_test, y_train, y_test = train_test_split(df_train['Complete_Textual_Da
# model.fit(X_train,y_train,epochs=3)

# model.evaluate(X_test, y_test)

# model.fit(df_train["Complete_Textual_Data"], df_train["label"], epochs=5)

# predictions = model.predict(df_test["Complete_Textual_Data"])

# predictions_list = predictions.flatten()
# print(predictions_list)
# pred_df = pd.DataFrame(predictions_list, index=df_test.link_id, columns=['label'_
# pred_df.to_csv('submission_Tranfer_Learning_BERT.csv')
# pred_df.head()

# print(bert_preprocess(df_train["Complete_Textual_Data"][0]))

# from transformers import BertTokenizer, BertModel

# from torch import nn
# from torch.optim import Adam

# tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
# labels = {0:0,
#           1:1,
#           }

# class Dataset(torch.utils.data.Dataset):

#     def __init__(self, df):

#         self.labels = [labels[label] for label in df['label']]
#         self.texts = [tokenizer(text,
#                                padding='max_length', max_length = 512, truncatio
#                                 return_tensors="pt") for text in df['text']]

#     def classes(self):
#         return self.labels

#     def __len__(self):
#         return len(self.labels)

#     def get_batch_labels(self, idx):
#         # Fetch a batch of labels
#         return np.array(self.labels[idx])

#     def get_batch_texts(self, idx):
#         # Fetch a batch of inputs
#         return self.texts[idx]

#     def __getitem__(self, idx):

#         batch_texts = self.get_batch_texts(idx)
#         batch_y = self.get_batch_labels(idx)

#         return batch_texts, batch_y

# class BertClassifier(nn.Module):

#     def __init__(self, dropout=0.5):

#         super(BertClassifier, self).__init__()

#         self.bert = BertModel.from_pretrained('bert-base-cased')
```

```
#         self.dropout = nn.Dropout(dropout)
#         self.linear = nn.Linear(768, 5)
#         self.relu = nn.ReLU()

#     def forward(self, input_id, mask):

#         _, pooled_output = self.bert(input_ids= input_id, attention_mask=mask,re
#         dropout_output = self.dropout(pooled_output)
#         linear_output = self.linear(dropout_output)
#         final_layer = self.relu(linear_output)

#         return final_layer

# def train(model, train_data, val_data, learning_rate, epochs):

#     train, val = Dataset(train_data), Dataset(val_data)

#     train_dataloader = torch.utils.data.DataLoader(train, batch_size=2, shuffle=
#     val_dataloader = torch.utils.data.DataLoader(val, batch_size=2)

#     use_cuda = torch.cuda.is_available()
#     device = torch.device("cuda" if use_cuda else "cpu")

#     criterion = nn.CrossEntropyLoss()
#     optimizer = Adam(model.parameters(), lr= learning_rate)

#     if use_cuda:

#             model = model.cuda()
#             criterion = criterion.cuda()

#     for epoch_num in range(epochs):

#             total_acc_train = 0
#             total_loss_train = 0

#             for train_input, train_label in tqdm(train_dataloader):

#                 train_label = train_label.to(device)
#                 mask = train_input['attention_mask'].to(device)
#                 input_id = train_input['input_ids'].squeeze(1).to(device)

#                 output = model(input_id, mask)

#                 batch_loss = criterion(output, train_label.long())
#                 total_loss_train += batch_loss.item()

#                 acc = (output.argmax(dim=1) == train_label).sum().item()
#                 total_acc_train += acc

#                 model.zero_grad()
#                 batch_loss.backward()
#                 optimizer.step()

#             total_acc_val = 0
#             total_loss_val = 0

#             with torch.no_grad():

#                 for val_input, val_label in val_dataloader:

#                     val_label = val_label.to(device)
#                     mask = val_input['attention_mask'].to(device)
#                     input_id = val_input['input_ids'].squeeze(1).to(device)
```

```
#                       output = model(input_id, mask)

#                       batch_loss = criterion(output, val_label.long())
#                       total_loss_val += batch_loss.item()

#                       acc = (output.argmax(dim=1) == val_label).sum().item()
#                       total_acc_val += acc

#               print(
#                   f'Epochs: {epoch_num + 1} | Train Loss: {total_loss_train / len(i

# def evaluate(model, test_data):

#     test = Dataset(test_data)

#     test_dataloader = torch.utils.data.DataLoader(test, batch_size=2)

#     use_cuda = torch.cuda.is_available()
#     device = torch.device("cuda" if use_cuda else "cpu")

#     if use_cuda:

#         model = model.cuda()

#     total_acc_test = 0
#     with torch.no_grad():

#         for test_input, test_label in test_dataloader:

#               test_label = test_label.to(device)
#               mask = test_input['attention_mask'].to(device)
#               input_id = test_input['input_ids'].squeeze(1).to(device)

#               output = model(input_id, mask)

#               acc = (output.argmax(dim=1) == test_label).sum().item()
#               total_acc_test += acc

#     print(f'Test Accuracy: {total_acc_test / len(test_data): .3f}')

# df_train['text'] = df_train['Complete_Textual_Data']
# df_test['text'] = df_test['Complete_Textual_Data']

# EPOCHS = 5
# model = BertClassifier()
# LR = 1e-6

# df_train_for_validation, df_validation = train_test_split(df_train, test_size=0.1

# # train(model, df_train_for_validation, df_validation, LR, EPOCHS)
```

Model 1:- BERT Encoder + LSTM

In [ ]:
```
from transformers import AutoTokenizer
import tensorflow as tf
import torch
device = torch.device('cuda')

#Downloading the tokenizer and the Albert model for fine tuning

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
```

In [ ]:
```
SEQ_length=512

#Lets create the X and Y matrix from the Df train set

Xids=np.zeros((df_train.shape[0],SEQ_length))
Xmask=np.zeros((df_train.shape[0],SEQ_length))
y=np.zeros((df_train.shape[0],1))

#Preparing the test dataframe

Xids_test=np.zeros((df_test.shape[0],SEQ_length))
Xmask_test=np.zeros((df_test.shape[0],SEQ_length))
Xids
```

Out[ ]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

In [ ]:
```
for i,sequence in enumerate(df_train['Complete_Textual_Data']):
    tokens=tokenizer.encode_plus(sequence,max_length=SEQ_length,padding='max_lengtl
                                 truncation=True,return_token_type_ids=False,return_atte
                                 return_tensors='tf')

    Xids[i,:],Xmask[i,:],y[i,0]=tokens['input_ids'],tokens['attention_mask'],df_tra


for i,sequence in enumerate(df_test['Complete_Textual_Data']):
    tokens=tokenizer.encode_plus(sequence,max_length=SEQ_length,padding='max_lengtl
                                 truncation=True,return_token_type_ids=False,return_atte
                                 return_tensors='tf')

    Xids_test[i,:],Xmask_test[i,:]=tokens['input_ids'],tokens['attention_mask']
```

In [ ]:
```
tf.config.get_visible_devices()
```

Out[ ]:
```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU'),
 PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

In [ ]:
```
dataset=tf.data.Dataset.from_tensor_slices((Xids,Xmask,y))

def map_func(input_ids,mask,labels):
    return {'input_ids':input_ids,'attention_mask':mask},labels

dataset=dataset.map(map_func)
dataset=dataset.shuffle(100000).batch(32).prefetch(1000)

DS_size=len(list(dataset))

train=dataset.take(round(DS_size*0.85))
val=dataset.skip(round(DS_size*0.85))
```

In [ ]:
```
dataset_test=tf.data.Dataset.from_tensor_slices((Xids_test,Xmask_test))

def map_func(input_ids,mask):
    return {'input_ids':input_ids,'attention_mask':mask}

dataset_test=dataset_test.map(map_func)
dataset_test=dataset_test.batch(32).prefetch(1000)
```

In [ ]:
```
from transformers import TFDistilBertModel, DistilBertConfig
distil_bert = 'distilbert-base-uncased'

config = DistilBertConfig(dropout=0.2, attention_dropout=0.2)
config.output_hidden_states = False
transformer_model = TFDistilBertModel.from_pretrained(distil_bert, config = config

input_ids_in = tf.keras.layers.Input(shape=(SEQ_length,), name='input_ids', dtype=
input_masks_in = tf.keras.layers.Input(shape=(SEQ_length,), name='attention_mask',

embedding_layer = transformer_model(input_ids_in, attention_mask=input_masks_in)[0
# X = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(50, return_sequences=True,
X = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(50, return_sequences=True, 
X = tf.keras.layers.GlobalMaxPool1D()(X)
X = tf.keras.layers.Dense(50, activation='relu')(X)
X = tf.keras.layers.Dropout(0.5)(X)
X = tf.keras.layers.Dense(1, activation='sigmoid')(X)
# X = tf.keras.layers.Dense(1, activation='sigmoid')(X)
model = tf.keras.Model(inputs=[input_ids_in, input_masks_in], outputs = X)

for layer in model.layers[:3]:
  layer.trainable = False
```

```
Some layers from the model checkpoint at distilbert-base-uncased were not used whe
n initializing TFDistilBertModel: ['activation_13', 'vocab_transform', 'vocab_laye
r_norm', 'vocab_projector']
- This IS expected if you are initializing TFDistilBertModel from the checkpoint o
f a model trained on another task or with another architecture (e.g. initializing
a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing TFDistilBertModel from the checkpoi
nt of a model that you expect to be exactly identical (initializing a BertForSeque
nceClassification model from a BertForSequenceClassification model).
All the layers of TFDistilBertModel were initialized from the model checkpoint at
distilbert-base-uncased.
If your task is similar to the task the model of the checkpoint was trained on, yo
u can already use TFDistilBertModel for predictions without further training.
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the
criteria. It will use a generic GPU kernel as fallback when running on GPU.
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the
criteria. It will use a generic GPU kernel as fallback when running on GPU.
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the
criteria. It will use a generic GPU kernel as fallback when running on GPU.
```

In [ ]:
```
model.summary()
```

```
Model: "model"
__________________________________________________________________________________
________________
 Layer (type)                   Output Shape          Param #     Connected to
==================================================================================
================
 input_ids (InputLayer)         [(None, 512)]         0           []

 attention_mask (InputLayer)    [(None, 512)]         0           []

 tf_distil_bert_model (TFDistil  TFBaseModelOutput(l  66362880    ['input_ids[0]
[0]',
 BertModel)                     ast_hidden_state=(N                'attention_mask
[0][0]']
                                one, 512, 768),
                                 hidden_states=None
                                , attentions=None)

 bidirectional (Bidirectional)  (None, 512, 100)      327600      ['tf_distil_bert_
model[0][0]']

 global_max_pooling1d (GlobalMa  (None, 100)          0           ['bidirectional
[0][0]']
 xPooling1D)

 dense_32 (Dense)               (None, 50)            5050        ['global_max_pool
ing1d[0][0]']

 dropout_40 (Dropout)           (None, 50)            0           ['dense_32[0]
[0]']

 dense_33 (Dense)               (None, 1)             51          ['dropout_40[0]
[0]']

==================================================================================
================
Total params: 66,695,581
Trainable params: 332,701
Non-trainable params: 66,362,880
__________________________________________________________________________________
________________
```

In [ ]:
```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer='adam',metrics=[tf.keras.metrics.AUC(),tf.keras.metrics.Pr
])
```

In [ ]:
```
model.fit(train,validation_data=val,epochs=1)
```

```
118/118 [==============================] - 702s 6s/step - loss: 0.5335 - auc: 0.80
98 - precision: 0.7879 - recall: 0.7059 - val_loss: 0.4580 - val_auc: 0.8737 - val
_precision: 0.8194 - val_recall: 0.7840
```

Out[ ]:
```
<keras.callbacks.History at 0x2381f6ba7c0>
```

In [ ]:
```
predicitions = model.predict(dataset_test)
print(predictions)
predicitions1 = predicitions

pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('BERT_Transfer_Learning_With_LSTM.csv')
pred_df.head()
```

```
[0.92730993 0.17934395 0.38999224 ... 0.24027644 0.25496772 0.28894147]
```

Out[ ]:

link_id	label
4049	0.927310
3692	0.179344
9739	0.389992
1548	0.730503
5574	0.951353

Model 2: - Hybrid model. Combination of previous Model with Best Model (Logistic Regression with Tuned Hyperparameters)

In [ ]:

```
from sklearn.model_selection import RandomizedSearchCV
vectorizer = TfidfVectorizer()
vectorizer.fit_transform(np.concatenate((df_train.Complete_Textual_Data.values, df_
# vectorizer.fit_transform(np.concatenate((df_local_train.Complete_Textual_Data.val
# vectorizer.fit_transform(df_local_train['Complete_Textual_Data'])
X_train = vectorizer.transform(df_train['Complete_Textual_Data'])
Y_train = df_train['label']
X_test = vectorizer.transform(df_test['Complete_Textual_Data'])
# y_test = df_test['label']

max_iter = range(100, 500)
solver = ['lbfgs', 'newton-cg', 'liblinear']
warm_start = [True, False]
C = np.arange(0, 1, 0.01)
random_grid ={
        'warm_start' : warm_start,
        'solver' : solver,
        'C' : C,
}
estimator = LogisticRegression()
random_estimator = RandomizedSearchCV(estimator = estimator,
                                   param_distributions = random_grid,
                                   n_iter = 10,
                                   scoring = 'accuracy',
                                   n_jobs = -1,
                                   verbose = 1,
                                   random_state = 1,
                                  )
random_estimator.fit(X_train, Y_train)
best_estimator = random_estimator.best_estimator_
# print(best_estimator.score(X_test, y_test))
print(random_estimator.best_params_)

predictions = best_estimator.predict_proba(X_test)[:, 1]
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
pred_df.to_csv('LR_HP_Tuned.csv')
pred_df.head()
predicitions2 = predicitions

predictions = (predicitions1 + predicitions2)/2
print(predictions)
pred_df = pd.DataFrame(predictions, index=df_test.link_id, columns=['label'])
```

```
pred_df.to_csv('Hybrid.csv')
pred_df.head()
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
{'warm_start': True, 'solver': 'newton-cg', 'C': 0.91}
[0.87727299 0.23241382 0.41605581 ... 0.29071209 0.28871812 0.27832926]
[[0.8983876 ]
 [0.3119287 ]
 [0.594178  ]
 ...
 [0.3560375 ]
 [0.3546402 ]
 [0.29490325]]
```

Out[ ]:

	label
link_id	
4049	0.898388
3692	0.311929
9739	0.594178
1548	0.777717
5574	0.915667