Task 1

In [1]:
```
import pandas as pd
```

In [2]:
```
import numpy as np
```

Read Dataset

In [3]:
```
df_train = pd.read_csv("data/reddit_200k_train.csv",encoding="iso-8859-
1")[["body","REMOVED"]]
```

In [4]:
```
df_test = pd.read_csv("data/reddit_200k_test.csv",encoding="iso-8859-1")
[["body","REMOVED"]]
```

In [5]:
```
df_train.head()
```

Out[5]:

	body	REMOVED
0	I've always been taught it emerged from the ea...	False
1	As an ECE, my first feeling as "HEY THAT'S NOT...	True
2	Monday: Drug companies stock dives on good new...	True
3	i learned that all hybrids are unfertile i won...	False
4	Well i was wanting to get wasted tonight. Not...	False

In [6]:
```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer, TfidfTransf
ormer
import nltk
import gensim
from nltk.tokenize import sent_tokenize, word_tokenize
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
from scipy.sparse import hstack
from sklearn.metrics import average_precision_score
from sklearn.metrics import roc_auc_score
from nltk import word_tokenize,sent_tokenize
from gensim import corpora
```

In [7]:
```
import warnings
warnings.filterwarnings("ignore")
```

In [8]:
```
X_train = df_train["body"]
y_train = df_train["REMOVED"]
```

In [9]:
```
X_test = df_test["body"]
y_test = df_test["REMOVED"]
```

1.1 Baseline model - baseline model using a bag-of-words approach and a linear model.

We use a bag of words approach using CountVectorizer with default parameters

In [10]:
```
pipe  =  make_pipeline(CountVectorizer(),LogisticRegression(solver="sag"
))
print("Cross val score on baseline model")
pipe.fit(X_train,y_train)
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score on baseline model
0.7188717147032742
```

In [11]:
```
y_preds = pipe.predict(X_test)
print("Roc-auc score on test set")
print(roc_auc_score(y_test, y_preds))
```

```
Roc-auc score on test set
0.5820846015686182
```

Using GridSearch

In [12]:
```
param_grid = {"logisticregression__C": [100,10,1,0.1,0.01],
             }
grid = GridSearchCV(make_pipeline(CountVectorizer(),LogisticRegression(s
olver="sag"),
                                  memory="cache_folder"),
                    param_grid=param_grid, cv=5, scoring="roc_auc"
                   )
```

In [13]:
```
grid.fit(X_train, y_train)
```

Out[13]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory='cache_folder',
     steps=[('countvectorizer', CountVectorizer(analyzer='word', binary
=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
  ... penalty='l2', random_state=None, solver='sag',
          tol=0.0001, verbose=0, warm_start=False))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'logisticregression__C': [100, 10, 1, 0.1, 0.01]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='roc_auc', verbose=0)
```

Grid best score

In [14]:
```
grid.best_score_
```

Out[14]:
```
0.7188801150880733
```

Grid best parameters

In [15]:
```
grid.best_params_
```

Out[15]:
```
{'logisticregression__C': 1}
```

In [16]:
```
print("Cross val score on baseline model after grid search")
print(np.mean(cross_val_score(grid,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score on baseline model after grid search
0.7188831216291589
```

1.2 Try using n-grams, characters, tf-idf rescaling and possibly other ways to tune the BoW model

Using infrequent word removal, stop words and token patterns to restrict the vocaublary

Using TF-IDF Transformer

We use token pattern parameter to restrict the kind of acceptable tokens - only letters, no digits, no underscores. We removed the stopwords. We use min_df parameter to only consider tokens that occur atleast 2 times in the data.

tf-idf rescaling is used to down-weight tokens that are very common.

In [17]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",mi
n_df=2,stop_words="english"),TfidfTransformer(),LogisticRegression(solve
r="sag"))
print("Cross val score using tf-idf transformer")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score using tf-idf transformer
0.767313738740652
```

Using TF-IDF transformer improves performance as compared to baseline

Downweighting very common words improves the performance for this problem

Using n-grams

N-grams is used to look at pairs of words that appear next to each other

Looking at only unigrams

ngram_range=(1,1)

In [18]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",mi
n_df=2,ngram_range=(1,1),stop_words="english"),LogisticRegression(solver
="sag"))
print("Cross val score using n-grams")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score using n-grams
0.6672887809116439
```

Using ngrams with range (1,1) does not improve performance as compared to baseline model

ngram_range=(1,2)

Looking at unigrams and bigrams. This gives more context as compared to unigrams

In [19]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",mi
n_df=2,ngram_range=(1,2),stop_words="english"),LogisticRegression(solver
="sag"))
print("Cross val score using n-grams")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score using n-grams
0.660401562493334
```

Using ngrams with range (1,2) does not improve performance as compared to baseline model

Adding more context(bigrams) as compared to unigrams worsens performance a little bit.

Looking at only bigrams

ngram_range=(2,2)

In [20]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",mi
n_df=2,ngram_range=(2,2),stop_words="english"),LogisticRegression(solver
="sag"))
print("Cross val score using n-grams")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score using n-grams
0.6452343885887948
```

Using ngrams with range (2,2) does not improve performance as compared to baseline model

This model performs worse than looking at unigrams only and looking at unigrams and bigrams

We try to add more context in the form on trigrams. We look at unigrams, bigrams and trigrams

ngram_range=(1,3)

In [21]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",min_df=2,ngram_range=(1,3),stop_words="english"),LogisticRegression(solver="sag"))
print("Cross val score using n-grams")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_auc")))
```

```
Cross val score using n-grams
0.6571242886465027
```

Using ngrams with range (1,3) does not improve performance as compared to baseline model

The performance is worse as compared to unigrams only and unigrams and bigrams

n-gram range (1,1) gives the best performance

Adding more context doesnt help in this particular problem.

Using character n-grams - can be helpful to be more robust towards misspelling or obfuscation

Using word boundary - respects word boundaries

In [22]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",min_df=2,stop_words="english",analyzer="char_wb"),LogisticRegression(solver="sag"))
print("Cross val score using character analyser")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_auc")))
```

```
Cross val score using character analyser
0.6917584429147609
```

Using character analyzer does not improve performance as compared to baseline model

Naive - does not respect character boundaries

In [23]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",mi
n_df=2,stop_words="english",analyzer="char"),LogisticRegression(solver=
"sag"))
print("Cross val score using character analyser")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score using character analyser
0.6960119397846068
```

Using character analyzer does not improve performance as compared to baseline model

analyzer="char" gives better performance

combining tf-idf transformer, count vectorizer, character analyzer - we use best parameter values obtained above

In [24]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",mi
n_df=2,stop_words="english",analyzer="char",ngram_range=(2,2)),TfidfTran
sformer(),LogisticRegression(solver="sag"))
print("Cross val score using tf-idf transformer, count vectorizer, chara
cter analyzer")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score using tf-idf transformer, count vectorizer, character a
nalyzer
0.7683041454603811
```

Performance improves significantly as compared to baseline model

In [25]:
```
pipe.fit(X_train, y_train)
y_preds = pipe.predict(X_test)
print("Roc-auc score on test set")
print(roc_auc_score(y_test, y_preds))
```

```
Roc-auc score on test set
0.6628629184710754
```

Performance on test set also improves as compared to baseline

Using GridSearchCV

In [26]:
```
param_grid = {"logisticregression__C": [100,10,1,0.1,0.01],
              }
grid = GridSearchCV(make_pipeline(CountVectorizer(token_pattern=r"\b[^\d
\W_]+\b",min_df=2,stop_words="english",analyzer="char",ngram_range=(2,2
)),TfidfTransformer(),LogisticRegression(solver="sag"),
                                  memory="cache_folder"),
                    param_grid=param_grid, cv=5, scoring="roc_auc"
                   )
```

In [27]:
```
grid.fit(X_train, y_train)
```

Out[27]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory='cache_folder',
     steps=[('countvectorizer', CountVectorizer(analyzer='char', binary
=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=2,
        ngram_range=(2, 2), preprocessor=None, stop_words='english... p
enalty='l2', random_state=None, solver='sag',
          tol=0.0001, verbose=0, warm_start=False))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'logisticregression__C': [100, 10, 1, 0.1, 0.01]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='roc_auc', verbose=0)
```

Grid best score

In [28]:
```
grid.best_score_
```

Out[28]:
```
0.7683306035804637
```

Grid best parameters

In [29]:
```
grid.best_params_
```

Out[29]:
```
{'logisticregression__C': 10}
```

In [30]:
```
print("Cross val score after grid search")
print(np.mean(cross_val_score(grid,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score after grid search
0.7683044550126812
```

Hence an approach using n-grams, characters, tf-idf rescaling, stop words, token patterns and infrequent word removal is better than our baseline model

1.3 Explore other features you can derive from the text, such as html, length, punctuation, capitalization

Count of words that are all caps - could indicate spam comments if count is high

In [12]:
```
def getCapWordsCount(sentence):
    count=0
    for word in sentence.split():
        if word.isupper() and len(word)>2:
            count = count + 1
    return count
```

In [13]:
```
df_train["Cap_words_count"] = df_train.apply(lambda row: getCapWordsCoun
t(row["body"]),axis=1)
```

In [14]:
```
df_test["Cap_words_count"] = df_test.apply(lambda row: getCapWordsCount(
row["body"]),axis=1)
```

Count of punctuation - Use of too many ! indicate spam comments and ? indicate questions

In [15]:
```
def getPunctuationCount(sentence):
    count=0
    for word in sentence:
        if word in ["!","?"]:
            count = count + 1
    return count
```

In [16]:
```
df_train["Punc_words_count"] = df_train.apply(lambda row: getPunctuation
Count(row["body"]),axis=1)
```

In [17]:
```
df_test["Punc_words_count"] = df_test.apply(lambda row: getPunctuationCo
unt(row["body"]),axis=1)
```

Sentence Length - Very short or very long sentences might be spam

In [18]:
```
def getSentenceLength(sentence):
    return len(sentence)
```

In [19]:
```
df_train["Sentence_length"] = df_train.apply(lambda row: getSentenceLeng
th(row["body"]),axis=1)
```

In [20]:
```
df_test["Sentence_length"] = df_test.apply(lambda row: getSentenceLength
(row["body"]),axis=1)
```

Word count in sentence - very few words or too many words might be spam

In [21]:
```
def getWordsCount(sentence):
    count=0
    for word in sentence.split():
        count = count + 1
    return count
```

In [22]:
```
df_train["Word_Count"] = df_train.apply(lambda row: getWordsCount(row["b
ody"]),axis=1)
```

In [23]:
```
df_test["Word_Count"] = df_test.apply(lambda row: getWordsCount(row["bod
y"]),axis=1)
```

POS tagging

Count of nouns

In [41]:
```
def getNounsCount(sentence):
    sentence_nouns = []
    is_noun = lambda pos: pos == 'NOUN'
    sentence = nltk.sent_tokenize(sentence)
    sentence = [nltk.word_tokenize(sent) for sent in sentence]
    for sent in sentence:
        sentence_nouns.append([word for (word, pos) in nltk.pos_tag(sent
,tagset='universal') if is_noun(pos)])
    return len(sentence_nouns)
```

In [42]:
```
df_train["Noun_Count"] = df_train.apply(lambda row: getNounsCount(row["b
ody"]),axis=1)
```

In [44]:
```
df_test["Noun_Count"] = df_test.apply(lambda row: getNounsCount(row["bod
y"]),axis=1)
```

Count of adjectives

In [39]:
```
#nltk.download('averaged_perceptron_tagger')
#nltk.download('universal_tagset')

from nltk import word_tokenize,sent_tokenize
def getAdjCount(sentence):
    sentence_nouns = []
    is_noun = lambda pos: pos == 'ADJ'
    sentence = nltk.sent_tokenize(sentence)
    sentence = [nltk.word_tokenize(sent) for sent in sentence]
    for sent in sentence:
        sentence_nouns.append([word for (word, pos) in nltk.pos_tag(sent
,tagset='universal') if is_noun(pos)])
    return len(sentence_nouns)
```

```
[nltk_data] Downloading package universal_tagset to
[nltk_data]     /Users/ankitpeshin/nltk_data...
[nltk_data]   Unzipping taggers/universal_tagset.zip.
```

In [40]:
```
df_train["Adj_Count"] = df_train.apply(lambda row: getAdjCount(row["bod
y"]),axis=1)
```

In [43]:
```
df_test["Adj_Count"] = df_test.apply(lambda row: getAdjCount(row["body"
]),axis=1)
```

Count of pronouns

In [45]:
```
def getPronounCount(sentence):
    sentence_nouns = []
    is_noun = lambda pos: pos == 'PRON'
    sentence = nltk.sent_tokenize(sentence)
    sentence = [nltk.word_tokenize(sent) for sent in sentence]
    for sent in sentence:
        sentence_nouns.append([word for (word, pos) in nltk.pos_tag(sent
,tagset='universal') if is_noun(pos)])
    return len(sentence_nouns)
```

In [46]:
```
df_train["Pronoun_Count"] = df_train.apply(lambda row: getPronounCount(r
ow["body"]),axis=1)
```

In [47]:
```
df_test["Pronoun_Count"] = df_test.apply(lambda row: getPronounCount(row
["body"]),axis=1)
```

Count of verbs

In [48]:
```
def getVerbCount(sentence):
    sentence_nouns = []
    is_noun = lambda pos: pos == 'VERB'
    sentence = nltk.sent_tokenize(sentence)
    sentence = [nltk.word_tokenize(sent) for sent in sentence]
    for sent in sentence:
        sentence_nouns.append([word for (word, pos) in nltk.pos_tag(sent
,tagset='universal') if is_noun(pos)])
    return len(sentence_nouns)
```

In [49]:
```
df_train["Verb_Count"] = df_train.apply(lambda row: getVerbCount(row["bo
dy"]),axis=1)
```

In [50]:
```
df_test["Verb_Count"] = df_test.apply(lambda row: getVerbCount(row["bod
y"]),axis=1)
```

Link present or absent

In [51]:
```
def contains_link(data):
    if "http" in data:
        return 1
    else:
        return 0
```

In [52]:
```
df_train['Link'] = df_train.apply(lambda row: contains_link(row['body'
]),axis=1)
```

In [53]:
```
df_test['Link'] = df_test.apply(lambda row: contains_link(row['body']),a
xis=1)
```

Sentiment analysis

Negative sentiment - might indicate harsh language

In [54]:
```
analyser = SentimentIntensityAnalyzer()

def sentiment_analyzer_neg(sentence):
    score = analyser.polarity_scores(sentence)
    return score['neg']
```

In [55]:
```
df_train["Negative_sent"] = df_train.apply(lambda row: sentiment_analyze
r_neg(row["body"]),axis=1)
```

In [56]:
```
df_test["Negative_sent"] = df_test.apply(lambda row: sentiment_analyzer_
neg(row["body"]),axis=1)
```

Positive sentiment

In [57]:
```
analyser = SentimentIntensityAnalyzer()

def sentiment_analyzer_pos(sentence):
    score = analyser.polarity_scores(sentence)
    return score['pos']
```

In [58]:
```
df_train["Positive_sent"] = df_train.apply(lambda row: sentiment_analyze
r_pos(row["body"]),axis=1)
```

In [59]:
```
df_test["Positive_sent"] = df_test.apply(lambda row: sentiment_analyzer_
pos(row["body"]),axis=1)
```

Logistic Regression using only engineered features ie. no body

In [64]:
```
X_train = df_train.drop(["body","REMOVED"],axis=1)
y_train = df_train["REMOVED"]
```

In [65]:
```
pipe  =  make_pipeline(LogisticRegression(solver="sag"))
print("Cross val score using engineered features only")
print(np.mean(cross_val_score(pipe,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score using engineered features only
0.6644582068447398
```

Performance is not better than baseline model

Logistic Regression using engineered features and body feature

In [66]:
```
X_train = df_train.drop(["body","REMOVED"],axis=1)
X_train_body = df_train["body"]
y_train = df_train["REMOVED"]
```

In [67]:
```
X_test = df_test.drop(["body","REMOVED"],axis=1)
X_test_body = df_test["body"]
y_test = df_test["REMOVED"]
```

using n-grams, characters, tf-idf rescaling, stop words, token patterns and infrequent word removal approach that gave best performance in task 1.2

In [68]:
```
pipe  =  make_pipeline(CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",mi
n_df=2,ngram_range=(1,1),stop_words="english",analyzer="char"), TfidfTra
nsformer())
X_train_body_vectorized = pipe.fit_transform(X_train_body)
```

In [69]:
```
X_test_body_vectorized = pipe.transform(X_test_body)
```

In [70]:
```
X_train= hstack((X_train_body_vectorized,np.array(X_train)))
```

In [71]:
```
X_test= hstack((X_test_body_vectorized,np.array(X_test)))
```

In [72]:
```
lr = LogisticRegression(solver="sag")
lr.fit(X_train,y_train)
```

Out[72]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=
True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='sag',
          tol=0.0001, verbose=0, warm_start=False)
```

In [74]:
```
y_preds = lr.predict(X_test)
print("Roc-auc score on test set")
print(roc_auc_score(y_test, y_preds))
```

```
Roc-auc score on test set
0.5013500849866808
```

Adding engineered features gives similar results to baseline model on test set.

Using Grid Search

In [75]:
```
param_grid = {"logisticregression__C": [100,10,1,0.1,0.01],
             }
grid = GridSearchCV(make_pipeline(LogisticRegression(solver="sag"),
                                  memory="cache_folder"),
                    param_grid=param_grid, cv=5, scoring="roc_auc"
                   )
```

In [76]:
```
grid.fit(X_train, y_train)
```

Out[76]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory='cache_folder',
     steps=[('logisticregression', LogisticRegression(C=1.0, class_weig
ht=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='sag',
          tol=0.0001, verbose=0, warm_start=False))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'logisticregression__C': [100, 10, 1, 0.1, 0.01]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='roc_auc', verbose=0)
```

Grid best score

In [77]:
```
grid.best_score_
```

Out[77]: 0.6645328052052749

Grid best params

In [78]:
```
grid.best_params_
```

Out[78]: {'logisticregression__C': 100}

In [81]:
```
print("Cross val score after grid search")
print(np.mean(cross_val_score(grid,X_train,y_train,cv=5,scoring="roc_au
c")))
```

```
Cross val score after grid search
0.6645310851304156
```

Adding our engineered features to the best model we got in task 1.2 using n-grams, characters, tf-idf rescaling, stop words, token patterns and infrequent word removal did not improve performance. This indicates that there might not be a pattern related to capitalization, punctuation, links, pos tagging and sentiment analysis that differentiates comments that have been removed from ones that havent been removed.

At the end of task 1, the best model we have is using using n-grams, characters, tf-idf rescaling, stop words, token patterns and infrequent word removal with no feature engineering. This gives an roc-auc score of 0.76

Task 2 - Using pre trained embedding - (fasttext trained model)

In [1]:
```
import pandas as pd
```

In [2]:
```
import numpy as np
```

Read Dataset

In [5]:
```
df_train = pd.read_csv("data/reddit_200k_train.csv",encoding="iso-8859-
1")[["body","REMOVED"]]
```

In [6]:
```
df_test = pd.read_csv("data/reddit_200k_test.csv",encoding="iso-8859-1")
[["body","REMOVED"]]
```

In [7]:
```
df_train.head()
```

Out[7]:

	body	REMOVED
0	I've always been taught it emerged from the ea...	False
1	As an ECE, my first feeling as "HEY THAT'S NOT...	True
2	Monday: Drug companies stock dives on good new...	True
3	i learned that all hybrids are unfertile i won...	False
4	Well i was wanting to get wasted tonight. Not...	False

In [8]:
```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer, TfidfTransf
ormer
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
from scipy.sparse import hstack
from sklearn.metrics import average_precision_score
from sklearn.metrics import roc_auc_score
```

In [9]:
```
import warnings
warnings.filterwarnings("ignore")
```

In [10]:
```
X_train = df_train["body"]
y_train = df_train["REMOVED"]
```

In [11]:
```
X_test = df_test["body"]
y_test = df_test["REMOVED"]
```

Use pretrained word2vec

In [16]:
```
from gensim import models
w = models.KeyedVectors.load_word2vec_format('wiki-news-300d-1M.vec')
```

In [17]:
```
def preprocess(doc):
    output =[]
    for word in doc:
        if word in w.vocab:
            output.append(word)
    return output
```

In [18]:
```
vect_w2v = CountVectorizer(token_pattern=r"\b[^\d\W_]+\b",min_df=2,stop_
words="english")
```

In [19]:
```
vect_w2v.fit(X_train)
docs_train = vect_w2v.inverse_transform(vect_w2v.transform(X_train))
docs_train = [preprocess(doc) for doc in docs_train]
```

In [20]:
```
docs_test = vect_w2v.inverse_transform(vect_w2v.transform(X_test))
docs_test = [preprocess(doc) for doc in docs_test]
```

In [21]:
```
def sentence_embedding(docs_val):
    X_train=[]
    i=-1
    index_array =[]
    for doc in docs_val:
        i=i+1
        sum0 = 0
        n=0
        if len(doc)>=1:
            for d in doc:
                sum0 = sum0 + w[d]
                n=n+1
            avg = sum0/n
            X_train.append(avg)
        else:
            index_array.append(i)
    return np.array(X_train),index_array
```

In [22]:
```
X_train_emb, train_ind_drop = sentence_embedding(docs_train)
```

In [23]:
```
X_test_emb, test_ind_drop = sentence_embedding(docs_test)
```

In [24]:
```
y_train_emb = y_train.drop(train_ind_drop,axis=0)
```

In [25]:
```
y_test_emb = y_test.drop(test_ind_drop,axis=0)
```

2: Using Word2Vec to get sentence embeddings

For this task, we use a pre-trained word-vectors generated using fasttext.

We first import the word vectors trained over Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset.

This is a 300-dimensional word vector with 1 million word vectors.

In [26]:
```
pipe  =  make_pipeline(LogisticRegression(solver="sag"))
print("Cross val score on word2vec model")
pipe.fit(X_train_emb,y_train_emb)
print(np.mean(cross_val_score(pipe,X_train_emb,y_train_emb,cv=5,scoring=
"roc_auc")))
```

```
Cross val score on word2vec model
0.7265647985815331
```

In [27]:
```
y_preds = pipe.predict(X_test_emb)
print("Roc-auc score on test set")
print(roc_auc_score(y_test_emb, y_preds))
```

```
Roc-auc score on test set
0.6281712173401774
```

Using GridSearch

In [29]:
```
param_grid = {"logisticregression__C": [100,10,1,0.1,0.01],
             }
grid = GridSearchCV(make_pipeline(LogisticRegression(solver="sag"),
                                  memory="cache_folder"),
                    param_grid=param_grid, cv=5, scoring="roc_auc"
                   )
```

In [30]:
```
grid.fit(X_train_emb, y_train_emb)
```

Out[30]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory='cache_folder',
     steps=[('logisticregression', LogisticRegression(C=1.0, class_weig
ht=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='sag',
          tol=0.0001, verbose=0, warm_start=False))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'logisticregression__C': [100, 10, 1, 0.1, 0.01]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='roc_auc', verbose=0)
```

Grid best score

In [31]:
```
grid.best_score_
```

Out[31]:
```
0.7265648822809037
```

Grid best parameters

In [32]:
```
grid.best_params_
```

Out[32]:
```
{'logisticregression__C': 1}
```

In [33]:
```
print("Cross val score on word2vec model after grid search")
print(np.mean(cross_val_score(grid,X_train_emb,y_train_emb,cv=5,scoring=
"roc_auc")))
```

```
Cross val score on word2vec model after grid search
0.7265648279548907
```

In [34]:
```
df_train = df_train.drop(train_ind_drop,axis=0)
df_test = df_test.drop(test_ind_drop,axis=0)
```

This approach using word2vec embeddings is better than our baseline model but it doesnt outperform our approach using n-grams, characters, tf-idf rescaling, stop words, token patterns and infrequent word removal in task 1.2.

Explore other features you can derive from the text, such as html, length, punctuation, capitalization in addition to word2vec

Count of words that are all caps - could indicate spam comments if count is high

In [35]:
```
def getCapWordsCount(sentence):
    count=0
    for word in sentence.split():
        if word.isupper() and len(word)>2:
            count = count + 1
    return count
```

In [36]:
```
df_train["Cap_words_count"] = df_train.apply(lambda row: getCapWordsCoun
t(row["body"]),axis=1)
```

In [37]:
```
df_test["Cap_words_count"] = df_test.apply(lambda row: getCapWordsCount(
row["body"]),axis=1)
```

Count of punctuation - Use of too many ! indicate spam comments and ? indicate questions

In [38]:
```
def getPunctuationCount(sentence):
    count=0
    for word in sentence:
        if word in ["!","?"]:
            count = count + 1
    return count
```

In [39]:
```
df_train["Punc_words_count"] = df_train.apply(lambda row: getPunctuation
Count(row["body"]),axis=1)
```

In [40]:
```
df_test["Punc_words_count"] = df_test.apply(lambda row: getPunctuationCo
unt(row["body"]),axis=1)
```

Sentence Length - Very short or very long sentences might be spam

In [41]:
```
def getSentenceLength(sentence):
    return len(sentence)
```

In [42]:
```
df_train["Sentence_length"] = df_train.apply(lambda row: getSentenceLeng
th(row["body"]),axis=1)
```

In [43]:
```
df_test["Sentence_length"] = df_test.apply(lambda row: getSentenceLength
(row["body"]),axis=1)
```

Word count in sentence - very few words or too many words might be spam

In [44]:
```
def getWordsCount(sentence):
    count=0
    for word in sentence.split():
        count = count + 1
    return count
```

In [45]:
```
df_train["Word_Count"] = df_train.apply(lambda row: getWordsCount(row["b
ody"]),axis=1)
```

In [46]:
```
df_test["Word_Count"] = df_test.apply(lambda row: getWordsCount(row["bod
y"]),axis=1)
```

POS tagging

Count of nouns

In [47]:
```
def getNounsCount(sentence):
    sentence_nouns = []
    is_noun = lambda pos: pos == 'NOUN'
    sentence = nltk.sent_tokenize(sentence)
    sentence = [nltk.word_tokenize(sent) for sent in sentence]
    for sent in sentence:
        sentence_nouns.append([word for (word, pos) in nltk.pos_tag(sent
,tagset='universal') if is_noun(pos)])
    return len(sentence_nouns)
```

In [48]:
```
df_train["Noun_Count"] = df_train.apply(lambda row: getNounsCount(row["b
ody"]),axis=1)
```

In [49]:
```
df_test["Noun_Count"] = df_test.apply(lambda row: getNounsCount(row["bod
y"]),axis=1)
```

Count of adjectives

In [50]:
```
def getAdjCount(sentence):
    sentence_nouns = []
    is_noun = lambda pos: pos == 'ADJ'
    sentence = nltk.sent_tokenize(sentence)
    sentence = [nltk.word_tokenize(sent) for sent in sentence]
    for sent in sentence:
        sentence_nouns.append([word for (word, pos) in nltk.pos_tag(sent
,tagset='universal') if is_noun(pos)])
    return len(sentence_nouns)
```

In [51]:
```
df_train["Adj_Count"] = df_train.apply(lambda row: getAdjCount(row["bod
y"]),axis=1)
```

In [52]:
```
df_test["Adj_Count"] = df_test.apply(lambda row: getAdjCount(row["body"
]),axis=1)
```

Count of pronouns

In [53]:
```
def getPronounCount(sentence):
    sentence_nouns = []
    is_noun = lambda pos: pos == 'PRON'
    sentence = nltk.sent_tokenize(sentence)
    sentence = [nltk.word_tokenize(sent) for sent in sentence]
    for sent in sentence:
        sentence_nouns.append([word for (word, pos) in nltk.pos_tag(sent
,tagset='universal') if is_noun(pos)])
    return len(sentence_nouns)
```

In [54]:
```
df_train["Pronoun_Count"] = df_train.apply(lambda row: getPronounCount(r
ow["body"]),axis=1)
```

In [55]:
```
df_test["Pronoun_Count"] = df_test.apply(lambda row: getPronounCount(row
["body"]),axis=1)
```

Count of verbs

In [56]:
```
def getVerbCount(sentence):
    sentence_nouns = []
    is_noun = lambda pos: pos == 'VERB'
    sentence = nltk.sent_tokenize(sentence)
    sentence = [nltk.word_tokenize(sent) for sent in sentence]
    for sent in sentence:
        sentence_nouns.append([word for (word, pos) in nltk.pos_tag(sent
,tagset='universal') if is_noun(pos)])
    return len(sentence_nouns)
```

In [57]:
```
df_train["Verb_Count"] = df_train.apply(lambda row: getVerbCount(row["bo
dy"]),axis=1)
```

In [58]:
```
df_test["Verb_Count"] = df_test.apply(lambda row: getVerbCount(row["bod
y"]),axis=1)
```

Link present or absent

In [59]:
```
def contains_link(data):
    if "http" in data:
        return 1
    else:
        return 0
```

In [60]:
```
df_train['Link'] = df_train.apply(lambda row: contains_link(row['body'
]),axis=1)
```

In [61]:
```
df_test['Link'] = df_test.apply(lambda row: contains_link(row['body']),a
xis=1)
```

Sentiment analysis

Negative sentiment - might indicate harsh language

In [62]:
```
analyser = SentimentIntensityAnalyzer()

def sentiment_analyzer_neg(sentence):
    score = analyser.polarity_scores(sentence)
    return score['neg']
```

In [63]:
```
df_train["Negative_sent"] = df_train.apply(lambda row: sentiment_analyze
r_neg(row["body"]),axis=1)
```

In [64]:
```
df_test["Negative_sent"] = df_test.apply(lambda row: sentiment_analyzer_
neg(row["body"]),axis=1)
```

Positive sentiment

In [65]:
```
analyser = SentimentIntensityAnalyzer()

def sentiment_analyzer_pos(sentence):
    score = analyser.polarity_scores(sentence)
    return score['pos']
```

In [66]:
```
df_train["Positive_sent"] = df_train.apply(lambda row: sentiment_analyze
r_pos(row["body"]),axis=1)
```

In [67]:
```
df_test["Positive_sent"] = df_test.apply(lambda row: sentiment_analyzer_
pos(row["body"]),axis=1)
```

Logistic Regression using engineered features and body feature

In [74]:
```
X_train = df_train.drop(["body","REMOVED"],axis=1)
```

In [75]:
```
X_test = df_test.drop(["body","REMOVED"],axis=1)
```

In [77]:
```
X_train= np.hstack((X_train_emb,np.array(X_train)))
```

In [78]:
```
X_test= np.hstack((X_test_emb,np.array(X_test)))
```

In [80]:
```
lr = LogisticRegression(solver="sag")
lr.fit(X_train,y_train_emb)
```

Out[80]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=
True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='sag',
          tol=0.0001, verbose=0, warm_start=False)
```

In [81]:
```
y_preds = lr.predict(X_test)
print("Roc-auc score on test set")
print(roc_auc_score(y_test_emb, y_preds))
```

```
Roc-auc score on test set
0.5011016674916332
```

Using Grid Search

In [82]:
```
param_grid = {"logisticregression__C": [100,10,1,0.1,0.01],
             }
grid = GridSearchCV(make_pipeline(LogisticRegression(solver="sag"),
                                  memory="cache_folder"),
                    param_grid=param_grid, cv=5, scoring="roc_auc"
                   )
```

In [84]:
```
grid.fit(X_train, y_train_emb)
```

Out[84]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory='cache_folder',
     steps=[('logisticregression', LogisticRegression(C=1.0, class_weig
ht=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='sag',
          tol=0.0001, verbose=0, warm_start=False))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'logisticregression__C': [100, 10, 1, 0.1, 0.01]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='roc_auc', verbose=0)
```

Grid best score

In [85]:
```
grid.best_score_
```

Out[85]:
```
0.6647347712696008
```

Grid best params

In [86]:
```
grid.best_params_
```

Out[86]:
```
{'logisticregression__C': 100}
```

In [87]:
```
print("Cross val score after grid search")
print(np.mean(cross_val_score(grid,X_train,y_train_emb,cv=5,scoring="roc
_auc")))
```

```
Cross val score after grid search
0.6647319497134462
```

Adding our engineered features to the best model we got in task 1.2 in addition to word2vec did not improve performance. This indicates that there might not be a pattern related to capitalization, punctuation, links, pos tagging and sentiment analysis that differentiates comments that have been removed from ones that havent been removed.

At the end of task 2, the best model we have is using using n-grams, characters, tf-idf rescaling, stop words, token patterns and infrequent word removal with no feature engineering. This gives an roc-auc score of 0.76 (Task 1.2)