In [1]:

```
import pandas as pd
import numpy as np
import pickle
import itertools
import functools
import collections
import random

from sklearn.linear_model import LogisticRegression
from sklearn import svm
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.metrics import confusion_matrix
from sklearn.metrics import average_precision_score
from sklearn.metrics import precision_recall_curve
from sklearn.utils.fixes import signature
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_validate
from sklearn.model_selection import train_test_split
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from sklearn.metrics import roc_auc_score
from sklearn.neighbors import NearestNeighbors

from gensim.test.utils import get_tmpfile
from gensim.utils import simple_preprocess

import matplotlib.pyplot as plt
% matplotlib inline
```

```
/anaconda3/lib/python3.7/site-packages/sklearn/ensemble/weight_boost
ing.py:29: DeprecationWarning: numpy.core.umath_tests is an internal
NumPy module and should not be imported. It will be removed in a fut
ure NumPy release.
  from numpy.core.umath_tests import inner1d
```

In [2]:

```
merged = pd.read_pickle('merged.pkl')
```

Parameter grid

In [3]:

```
params = {'m_size': [5000],
              'test_size': [0.2],
              'vector_size': [400],
              'min_count': [2],
              'epochs': [20],
              'window': [5],
              'steps': [30],
              'model': ['svm'],
              'SMOTE_en': [1],
              'k_SMOTE': [5],
              'N_SMOTE': [200]
             }
```

Functions

In [15]:

```
def sampleData(df, m_size):
    """
    Sample the data with hyperparameter m_size specifying how many training exam
ples we want
    Hyperparameters: m_size

    :param df: Input Pandas DataFrame
    :type df: pd.DataFrame
    :param m_size: Number of training examples desired
    :type m_size: int

    :return: X, y of combined sampled training data and labels for the model
    :rtype: List*2
    """

    pos = df[df['label'] == 1]
    neg = (df[df['label'] == 0])
    rat = m_size/len(neg)

    sections = neg.groupby('section_x').count()['_id_x']
    sections = sections[sections > 10]
    sections_to_keep = set(sections.index)
    mask = neg['section_x'].apply(lambda x: x in sections_to_keep)
    neg = neg[mask]

    _, neg = train_test_split(neg, test_size=rat, random_state=42, stratify=neg[
'section_x'])

    data = neg.append(pos)
    combined = [(h + ' ' + s + ' ' + b, l) for h, s, b, l in
                    zip(list(data['headline_x']), list(data['summary_x']), list(
data['body_x']), list(data['label']))]
    print('Sampling Done')
    return zip(*combined)



def stratSpl(X, y, test_size):
    """
    Make a stratified test/train split to use for training and testing.
    Hyperparameters: None

    :param X: Input features of the combined (train and test) sampled set.
    :type X: List
    :param y: Input labels of the combined (train and test) sampled set
    :type y: List
    :param test_size: Test ratio to split up. Number between 0 and 1
    :type test_size: Float

    :return: 4 Lists corresponding to X_tr, X_te, y_tr, y_te
    :rtype: List*4
    """

    sss = StratifiedShuffleSplit(n_splits=1, test_size=test_size, random_state=4
2)
    for train_index, test_index in sss.split(X, y):
        X_tr, X_te = [X[i] for i in train_index], [X[i] for i in test_index]
        y_tr, y_te = [y[i] for i in train_index], [y[i] for i in test_index]
```

```
    print('Stratified test/train split done')
    return X_tr, y_tr, X_te, y_te



def read_corpus(data):
    """
    Prepare the data (using gensims simple_preprocess)
    Hyperparameters: None

    :param data: Document
    :type X_tr: List

    :return: Processed version data.
    :rtype: Iterator

    """

    print('Tokenizing data')

    for i, line in enumerate(data):
        yield TaggedDocument(simple_preprocess(line), tags=[i])



def doc2vec_model_train(X_tr, vector_size, min_count, epochs, window):
    """
    Doc2Vec model defined and trained via this function.
    Hyperparameters: size, min_count, epochs, window

    :param X_tr: Training data
    :type X_tr: List
    :param vector_size: Dimensionality of the feature vectors
    :type vector_size: Int
    :param min_count: Minimum occurences for which to still keep a word in the v
ocab.
    :type min_count: Int
    :param epochs: Number of epochs for which the model trains
    :type epochs: Int
    :param window: Window size of context to consider in a given instance.
    :type window: Int

    :return: Fully trained model
    :rtype: gensim model
    """

    model = Doc2Vec(vector_size=vector_size, min_count=min_count, window=window,
 epochs=epochs)
    model.build_vocab(X_tr)
    model.train(X_tr, total_examples=model.corpus_count, epochs=model.epochs)

    print('Doc2Vec Model Trained')

    return model


def embeddings(model, X, steps):

    """
    Embed documents into vector space for classification in the next stage.
    Hyperparameters: steps
```

```
    :param model: Trained Doc2Vec model
    :type model: gensim Doc2Vec model
    :param X: Input corpus
    :type X: List of TaggedDocuments
    :param steps: Hyperparameter to tune
    :type steps: Int

    :return: Embedded feature vector
    :rtype: List
    """

    z = [model.infer_vector(X[doc_id].words, steps=steps) for doc_id in range(le
n(X))]

    print('Documents embedded into vector space')

    return z


def FinalClassifier(X_tr, y_tr, model='logistic_regression'):

    """
    Models for final classifcation, will be hyperparameters
    Hyperparameters: The models themselves and their hyperparameters *Come back
 here for alteration

    :param X_tr: Input document vectors
    :type X_tr: List
    :param y_tr: Labels
    :type: List

    :return: Trained logreg model
    :rtype:
    """
    if model == 'logistic_regression':
        clf = LogisticRegression(random_state=42).fit(X_tr, y_tr)
        print('Logistic Regression Classifier Trained.')

    if model == 'svm':
        clf = svm.SVC().fit(X_tr, y_tr)
        print('SVM Classifier trained')

    if model == 'decision_tree':
        clf = tree.DecisionTreeClassifier(random_state=42).fit(X_tr, y_tr)
        print('Decision Tree Classifier')

    if model == 'random_forest':
        clf = RandomForestClassifier(random_state=42).fit(X_tr, y_tr)

    return clf




def SMOTE(T, N, k, pos):
    """
    Synthetic Minority Over-sampling Technique: https://jair.org/index.php/jair/
article/view/10302/24590
    Hyperparameters: N, k
```

```
    :param T: Number of minority class samples
    :type T: Int
    :param N: Amount of SMOTE N%
    :type N: Int (Integral multiple of 100)
    :param k: K nearest neighbors
    :type k: Int
    :param pos: Positive examples from training set
    :type pos: np array

    :return: (N/100) * T synthetic minority class samples T
    :rtype:

    """
    def Populate(N, i, nnarray):
        while N != 0:

            nn = random.randint(0,k-1)
            dif = Sample[nnarray[nn]] - Sample[i]
            gap = random.uniform(0, 1)
            syn = Sample[i] + gap*dif
            Synthetic.append(syn)

            N -= 1


    N = N//100 # Integer multiple of 100, specifies ratio of

    Sample = pos # Negative (minority class) document vectors (np array)

    Synthetic = [] # Array to keep track of newly generated synthetic examples

    # nearest neighbors for each vector in Sample. nnarray[i] corresponds to nea
rest neighbor indices for vector Sample[i]
    neigh = NearestNeighbors(n_neighbors=k+1)
    neigh.fit(Sample)
    nnarrays = neigh.kneighbors(Sample, return_distance=False)
    nnarrays = [ind[1:] for ind in nnarrays]


    for i in range(T):
        Populate(N, i, nnarrays[i])

    return np.array(Synthetic)


# def cross_val(clf, X_tr, y_tr):
#     """
#     Cross validation on training set. This will be used to score the grid sear
ch models and tune hyperparameters.
#     Hyperparameters: None

#     :param clf: Second stage classifier
#     :type clf: Sklearn (or other) classifier
#     :param X_tr: Training data emedded doc vectors
#     :type X_tr: List
#     :param y_tr: Training labels
#     :type y_tr:

#     """
#     scoring = ['f1', 'precision', 'recall', 'average_precision']
```

```
#     scores = cross_validate(clf, X_tr, y_tr, cv=3, scoring=scoring)

#     print('Cross val scores computed for this set of params.')

#     return scores

def extract_pos(X_tr, y_tr):
    return np.array([v for v, l in zip(X_tr, y_tr) if l==1])

def unison_shuffled_copies(a, b):
    assert len(a) == len(b)
    p = np.random.permutation(len(a))
    return a[p], b[p]

def precision(conf):
    num = conf[1][1]
    den = num + conf[0][1]

    return num/den

def recall(conf):
    num = conf[1][1]
    den = num + conf[1][0]

    return num/den

def F1(P, R):
    return 2 * P*R/(P+R)

def average(l):
    return functools.reduce(lambda x, y: x + y, l) / len(l)

def flatten(x):
    if isinstance(x, collections.Iterable) and not isinstance(x, tuple) and not
isinstance(x, str) and not isinstance(x, dict):
            return [a for i in x for a in flatten(i)]
    else:
        return [x]
```

In [5]:

```
# Add new hyperparameters here (twice)

def unpack_kwargs(**kwargs):
    m_size = kwargs.pop('m_size')
    test_size = kwargs.pop('test_size')
    vector_size = kwargs.pop('vector_size')
    min_count = kwargs.pop('min_count')
    epochs = kwargs.pop('epochs')
    window = kwargs.pop('window')
    steps = kwargs.pop('steps')
    model = kwargs.pop('model')
    SMOTE_en = kwargs.pop('SMOTE_en')
    k_SMOTE = kwargs.pop('k_SMOTE')
    N_SMOTE = kwargs.pop('N_SMOTE')

    return m_size, test_size, vector_size, min_count, epochs, window, steps, mod
el, SMOTE_en, k_SMOTE, N_SMOTE
```

Full Pipeline

In [6]:

```
def full_pipeline(scores, **kwargs):


    # Add new hyperparameters here too!
    m_size, test_size, vector_size, min_count, epochs, window, steps, model,
    SMOTE_en, k_SMOTE, N_SMOTE = unpack_kwargs(**kwargs)

    X, y = sampleData(merged, m_size)

    X_tr, y_tr, _, _ = stratSpl(X, y, test_size)

    X_tr = list(read_corpus(X_tr))

    skf = StratifiedKFold(n_splits=5, random_state=42)

    temp = []
    print('Cross validation commencing...')
    i = 0
    for train_index, test_index in skf.split(X_tr, y_tr):

        print('Split %r...' % i)

        X_tr_cv, X_te_cv = [X_tr[i] for i in train_index], [X_tr[i] for i in tes
t_index]
        y_tr_cv, y_te_cv = [y_tr[i] for i in train_index], [y_tr[i] for i in tes
t_index]

        d2v = doc2vec_model_train(X_tr_cv, vector_size, min_count, epochs, windo
w)

        X_tr_cv = embeddings(d2v, X_tr_cv, steps)

        # ----------- SMOTE code

        if SMOTE_en == 1:

            X_tr_cv = np.array(X_tr_cv)

            print(X_tr_cv.shape)

            X_tr_cv_pos = extract_pos(X_tr_cv, y_tr)

            print(X_tr_cv_pos.shape)

            X_tr_syn = SMOTE(len(X_tr_cv_pos), N_SMOTE, k_SMOTE, X_tr_cv_pos)

            print(X_tr_syn.shape)

            X_tr_cv = np.vstack((X_tr_cv, X_tr_syn))
            y_tr_cv += ([1 for _ in range(len(X_tr_syn))])

            print(X_tr_cv.shape, len(y_tr_cv))

            X_tr_cv, y_tr_cv = unison_shuffled_copies(X_tr_cv, np.array(y_tr_cv
))

        # ----------- Ended SMOTE code

        clf = FinalClassifier(X_tr_cv, y_tr_cv, model=model)
```

```
        X_te_cv = embeddings(d2v, X_te_cv, steps)

        y_pr = clf.predict(X_te_cv)

        y_sc = clf.decision_function(X_te_cv) if model=='svm' or model=='logisti
c_regression' else clf.predict_proba(X_te_cv)[:,1]

        conf = confusion_matrix(y_te_cv, y_pr)
        print(conf)

        p, r = precision(conf), recall(conf)

        auc, ap = roc_auc_score(y_te_cv, y_sc), average_precision_score(y_te_cv,
 y_sc)

        temp.append([p, r, auc, ap])

        i += 1

    scores.append(temp)

    print('------------------------------------------------')

    return scores
```

Grid Search

In [7]:

```
def product_dict(**kwargs):
    keys = kwargs.keys()
    vals = kwargs.values()

    for instance in itertools.product(*vals):
        yield dict(zip(keys, instance))

results = {}
scores=[]

for i, param in enumerate(list(product_dict(**params))):
    print('Checking set %r of parameters...' % i)

    scores = full_pipeline(scores, **param)

    results[i] = flatten([param, list(zip(*scores[i]))])
```

```
Checking set 0 of parameters...
Sampling Done
Stratified test/train split done
Tokenizing data
Cross validation commencing...
Split 0...
Doc2Vec Model Trained
Documents embedded into vector space
(3535, 400)
(331, 400)
(662, 400)
(4197, 400) 4197
SVM Classifier trained
Documents embedded into vector space
[[796   4]
 [ 18  67]]
Split 1...
Doc2Vec Model Trained
Documents embedded into vector space
(3535, 400)
(331, 400)
(662, 400)
(4197, 400) 4197
SVM Classifier trained
Documents embedded into vector space
[[797   3]
 [ 13  72]]
Split 2...
Doc2Vec Model Trained
Documents embedded into vector space
(3536, 400)
(331, 400)
(662, 400)
(4198, 400) 4198
SVM Classifier trained
Documents embedded into vector space
[[797   2]
 [  6  79]]
Split 3...
Doc2Vec Model Trained
Documents embedded into vector space
(3537, 400)
(331, 400)
(662, 400)
(4199, 400) 4199
SVM Classifier trained
Documents embedded into vector space
[[797   2]
 [  4  80]]
Split 4...
Doc2Vec Model Trained
Documents embedded into vector space
(3537, 400)
(331, 400)
(662, 400)
(4199, 400) 4199
SVM Classifier trained
Documents embedded into vector space
[[791   8]
 [  6  78]]
--------------------------------------------
```

In [8]:

```
results
```

Out[8]:

```
{0: [{'m_size': 5000,
   'test_size': 0.2,
   'vector_size': 400,
   'min_count': 2,
   'epochs': 20,
   'window': 5,
   'steps': 30,
   'model': 'svm',
   'SMOTE_en': 1,
   'k_SMOTE': 5,
   'N_SMOTE': 200},
  (0.5, 0.5, 0.5, 0.5, 0.5),
  (0.788235294117647,
   0.8470588235294118,
   0.9294117647058824,
   0.9523809523809523,
   0.9285714285714286),
  (0.9791911764705883,
   0.9927499999999999,
   0.9973790767871604,
   0.9996871088861077,
   0.9969902854758924),
  (0.9235494035059986,
   0.9567684460742579,
   0.9856313349545774,
   0.9971792088364253,
   0.9730267846862742)]}
```

Print Cross Val Results (all splits)

In [9]:

```
data = [[key] + [val for val in vals] for key, vals in results.items()]


pr = pd.DataFrame(data, columns=['Model #', 'Parameters', 'Precision', 'Recall',
                                 'AUC', 'AP'])

pd.set_option('display.max_colwidth', -1)

pr
```

Out[9]:

	Model #	Parameters	Precision	Recall	AUC	
0	0	{'m_size': 5000, 'test_size': 0.2, 'vector_size': 400, 'min_count': 2, 'epochs': 20, 'window': 5, 'steps': 30, 'model': 'svm', 'SMOTE_en': 1, 'k_SMOTE': 5, 'N_SMOTE': 200}	(0.5, 0.5, 0.5, 0.5, 0.5)	(0.788235294117647, 0.8470588235294118, 0.9294117647058824, 0.9523809523809523, 0.9285714285714286)	(0.9791911764705883, 0.9927499999999999, 0.9973790767871604, 0.9996871088861077, 0.9969902854758924)	(0.923549403505 0.956768446074 0.985631334954 0.997179208836 0.973026784686

Print Cross Val Results(average)

In [10]:

```
pr_av = pr.copy()
```

In [11]:

```
pr_av['Precision'] = pr_av['Precision'].apply(average)
```

In [12]:

```
pr_av['Recall'] = pr_av['Recall'].apply(average)
pr_av['AUC'] = pr_av['AUC'].apply(average)
pr_av['AP'] = pr_av['AP'].apply(average)
```

In [13]:

```
with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    display(pr_av)
```

	Model #	Parameters	Precision	Recall	AUC	AP
0	0	{'m_size': 5000, 'test_size': 0.2, 'vector_size': 400, 'min_count': 2, 'epochs': 20, 'window': 5, 'steps': 30, 'model': 'svm', 'SMOTE_en': 1, 'k_SMOTE': 5, 'N_SMOTE': 200}	0.5	0.889132	0.9932	0.967231

In [ ]:

```

```