In [49]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline

import pandas as pd
import numpy as np

from tools import load_data, save_prediction
import collections
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.stem import WordNetLemmatizer
from nltk import word_tokenize
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [60]:
```
import nltk
nltk.download('punkt')
nltk.download('wordnet')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\Chandler\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\Chandler\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping corpora\wordnet.zip.
```

Out[60]: True

Featurizer

In [40]:
```
"""
!! Do not modify !!
"""
def dumb_featurize(text):
    feats = {}
    words = text.split(" ")

    for word in words:
        if word == "love" or word == "like" or word == "best":
            feats["contains_positive_word"] = 1
        if word == "hate" or word == "dislike" or word == "worst" or word == "awf
            feats["contains_negative_word"] = 1

    return feats
```

In [55]:
```
class LemmaTokenizer(object):
    def __init__(self):
        self.wnl = WordNetLemmatizer()
    def __call__(self, articles):
        return [self.wnl.lemmatize(t) for t in word_tokenize(articles)]
```

In [66]:
```
def bagofNGram_featurize(text):
    """
    Bag of N-Grams Model
    """
    bv = CountVectorizer(
                        lowercase = True,
                        analyzer='word',
                        tokenizer=LemmaTokenizer(),
                        ngram_range=(1,2)
                        )
    listBvCount = bv.fit_transform([text])
    vocab = bv.get_feature_names()

    return dict(zip(vocab,list(listBvCount.toarray()[0])))
```

In [64]:
```
def bagOfOneGram_featurize(text):
    """
    Bag of words Model
    """
    bv = CountVectorizer(
                        lowercase = True,
                        analyzer='word',
                        tokenizer=LemmaTokenizer(),
                        min_df=0., max_df=1.
                        )
    listBvCount = bv.fit_transform([text])
    vocab = bv.get_feature_names()

    return dict(zip(vocab,list(listBvCount.toarray()[0])))
```

In [67]:
```
def tfIDF_featurize(text):
    """
    TF-IDF model
    """
    tv = TfidfVectorizer(
                        lowercase = True,
                        analyzer='word',
                        tokenizer=LemmaTokenizer(),
                        min_df=0., max_df=1.,
                        use_idf=True
                        )
    listTvCount = tv.fit_transform([text])
    vocab = tv.get_feature_names()
    return dict(zip(vocab,list(np.round(listTvCount.toarray()[0],2))))
```

Model Class

In [44]:
```
from collections import Counter
from scipy.sparse import dok_matrix
from sklearn.linear_model import LogisticRegression
from itertools import chain

class SentimentClassifier:
    def __init__(self, feature_method=dumb_featurize, min_feature_ct=1, L2_reg=1.(
        """
        :param feature_method: featurize function
        :param min_feature_count: int, ignore the features that appear less than t
        """
        self.feature_vocab = {}
        self.feature_method = feature_method
        self.min_feature_ct = min_feature_ct
        self.L2_reg = L2_reg

    def featurize(self, X):
        """
        # Featurize input text

        :param X: list of texts
        :return: list of featurized vectors
        """
        featurized_data = []
        for text in X:
            # Removing stopwords and special character
            for uselessWord in ['the','.',',','and','a','an',':','that','is','was
                try:
                    text.remove(uselessWord)
                except:
                    pass
            feats = self.feature_method(text)
            #print(feats)
            featurized_data.append(feats)
        #print(featurized_data)
        return featurized_data

    def pipeline(self, X, training=False):
        """
        Data processing pipeline to translate raw data input into sparse vectors
        :param X: featurized input
        :return: 2d sparse vectors

        Implement the pipeline method that translate the dictionary like feature v
        vectors, for example:
        [{"fea1": 1, "fea2": 2},
         {"fea2": 2, "fea3": 3}]
         -->
         [[1, 2, 0],
          [0, 2, 3]]

        Hints:
        1. How can you know the length of the feature vector?
        2. When should you use sparse matrix?
        3. Have you treated non-seen features properly?
        4. Should you treat training and testing data differently?
```

```
        """
        # Have to build feature_vocab during training
        if training:
            finalOutput = []
            # get the full feature vector
            #listFull = list(set(chain.from_iterable(X)))
            for ls in X:
                self.feature_vocab = dict(self.feature_vocab, **ls)
            # translate the dictionary like feature vectors into homogeneous numer
            for ls in X:
                output = []
                for vector in self.feature_vocab.keys():
                    if vector in list(ls.keys()):
                        output.append(ls[vector])
                    else:
                        output.append(0)
                finalOutput.append(output)

            return np.array(finalOutput)

            #raise NotImplementedError

        # Translate raw texts into vectors
        else:
            finalOutput = []
            # use same full feature vector from training data
            # translate the dictionary like feature vectors into homogeneous numer
            for ls in X:
                output = []
                for vector in self.feature_vocab.keys():
                    if vector in list(ls.keys()):
                        output.append(ls[vector])
                    else:
                        output.append(0)
                finalOutput.append(output)
            return np.array(finalOutput)

    def fit(self, X, y):
        X = self.pipeline(self.featurize(X), training=True)
        #print(X)
        D, F = X.shape
        self.model = LogisticRegression(C=self.L2_reg)
        self.model.fit(X, y)

        return self

    def predict(self, X):
        X = self.pipeline(self.featurize(X))
        return self.model.predict(X)

    def score(self, X, y):
        X = self.pipeline(self.featurize(X))
        return self.model.score(X, y)

    # Write learned parameters to file
    def save_weights(self, filename='weights.csv'):
        weights = [["__intercept__", self.model.intercept_[0]]]
```

```
        for feat, idx in self.feature_vocab.items():
            weights.append([feat, self.model.coef_[0][idx]])

        weights = pd.DataFrame(weights)
        weights.to_csv(filename, header=False, index=False)

        return weights
```

In [6]:
```
"""
Run this to test your model implementation
"""

cls = SentimentClassifier()
X_train = [{"fea1": 1, "fea2": 2}, {"fea2": 2, "fea3": 3}]

X = cls.pipeline(X_train, True)
assert X.shape[0] == 2 and X.shape[1] >= 3, "Fail to vectorize training features"

X_test = [{"fea1": 1, "fea2": 2}, {"fea2": 2, "fea3": 3}]
X = cls.pipeline(X_test)
assert X.shape[0] == 2 and X.shape[1] >= 3, "Fail to vectorize testing features"

X_test = [{"fea1": 1, "fea2": 2}, {"fea2": 2, "fea4": 3}]
try:
    X = cls.pipeline(X_test)
    assert X.shape[0] == 2 and X.shape[1] >= 3
except:
    print("Fail to treat un-seen features")
    raise Exception

print("Success!!")
```

```
Success!!
```

Run your models

In [41]:
```
"""
Run this cell to test your model performance - dumb method
"""
from sklearn.model_selection import train_test_split

data = load_data("train_sample.txt")
X, y = data.text, data.target
X_train, X_dev, y_train, y_dev = train_test_split(X, y, test_size=0.3)
cls = SentimentClassifier(feature_method=dumb_featurize)
cls = cls.fit(X_train, y_train)
print("Training set accuracy: ", cls.score(X_train, y_train))
print("Dev set accuracy: ", cls.score(X_dev, y_dev))
```

```
Training set accuracy:  0.5409429280397022
Dev set accuracy:  0.5057803468208093

C:\Users\Chandler\AppData\Roaming\Python\Python36\site-packages\sklearn\linear_
model\logistic.py:433: FutureWarning: Default solver will be changed to 'lbfgs'
in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

In [ ]:
```
"""
Run this cell to test your model performance
"""
from sklearn.model_selection import train_test_split

data = load_data("train.txt")
X, y = data.text, data.target
X_train, X_dev, y_train, y_dev = train_test_split(X, y, test_size=0.3)

# Bag of N Grams
cls = SentimentClassifier(feature_method=bagofNGram_featurize)
cls = cls.fit(X_train, y_train)
print("Training set accuracy using bag of N Grams: ", cls.score(X_train, y_train))
print("Dev set accuracy: ", cls.score(X_dev, y_dev))
save_prediction(cls.predict(X_dev),filename="NGram_prediction_withLemma.csv")

# Bag of one word
cls = SentimentClassifier(feature_method=bagOfOneGram_featurize)
cls = cls.fit(X_train, y_train)
print("Training set accuracy using bag of words: ", cls.score(X_train, y_train))
print("Dev set accuracy: ", cls.score(X_dev, y_dev))
save_prediction(cls.predict(X_dev),filename="bagWords_prediction_withLemma.csv")
```

In [ ]:
```
"""
Run this cell to test your model performance
"""
from sklearn.model_selection import train_test_split

data = load_data("train.txt")
X, y = data.text, data.target
X_train, X_dev, y_train, y_dev = train_test_split(X, y, test_size=0.3)

# TF-IDF Model

cls = SentimentClassifier(feature_method=tfIDF_featurize)
cls = cls.fit(X_train, y_train)
print("Training set accuracy using IF-IDF: ", cls.score(X_train, y_train))
print("Dev set accuracy: ", cls.score(X_dev, y_dev))
save_prediction(cls.predict(X_dev),filename="TF_IDF_prediction_withLemma.csv")
```

In [ ]:
```
"""
Run this cell to save weights and the prediction
"""
weights = cls.save_weights()

X_test = load_data("test.txt").text
save_prediction(cls.predict(X_test))
```

(Optional) Use different learning methods

God job reaching this point! So far you have explored many different ways of doing feature engineering, but how about the learning method? In the previous implementation Logistic Regression was used. Now you can try to use different learning methods.

hint: inherit the previous model and overwrite the `fit` method

My Ref: https://towardsdatascience.com/understanding-feature-engineering-part-3-traditional-methods-for-text-data-f6f7d70acd41 (https://towardsdatascience.com/understanding-feature-engineering-part-3-traditional-methods-for-text-data-f6f7d70acd41) https://scikit-learn.org/stable/modules/feature_extraction.html (https://scikit-learn.org/stable/modules/feature_extraction.html)