LR-SentimentAnalysis

October 18, 2023

1 Logistic Regression and Sentiment analysis

In this assignment you will implement and experiment with various feature engineering techniques in the context of Logistic Regression models for Sentiment classification of movie reviews.

1. Read lexicons of positive and negative sentiment words.
2. Implement overall positive and negative lexicon count features.
3. Implement per-lexicon-word count features.
4. Implement document length feature.
5. Implement deictic features.
6. [5111] Pre-processing for negation.
7. [5111] Plot learning curves.
8. Bonus points.
9. Analysis of results.

We will use the LR model implemented in sklearn:

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

1.1 Write Your Name Here: Claire Ardern

2 Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of ll cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a PDF version *LR-SentimentAnalysis.pdf* showing the code and the output of all cells, and save it in the same folder that contains the notebook file *LR-SentimentAnalysis.ipynb*.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing we will see when grading!
7. Submit **both** your PDF and notebook on Canvas.
8. Verify your Canvas submission contains the correct files by downloading it after posting it on Canvas.

2.1 [5111] Theory

Mandatory for graduate students, optional for undergraduate students

1. Prove that the derivative of the sigmoid function can be written as $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$.
2. Using the chain rule of differentiation, prove that $\frac{\delta \sigma(\mathbf{w}^T\mathbf{x} + b)}{\delta \mathbf{w}} = \sigma(\mathbf{w}^T\mathbf{x} + b) \cdot (1 - \sigma(\mathbf{w}^T\mathbf{x} + b)) \cdot \mathbf{x}$.

2.1.1 Theory 1 Proof:

$$\frac{d}{dz}\sigma(z) = \frac{d}{dz}\left[\frac{1}{1+e^{-z}}\right]$$

$$= \frac{d}{dz}(1+e^{-z})^{-1}$$

$$= -(1+e^{-z})^{-2} \cdot (-e^{-z})$$

$$= \frac{e^{-z}}{(1+e^{-z})^2}$$

$$= \left(\frac{1}{1+e^{-z}}\right) \cdot \left(\frac{e^{-z}}{1+e^{-z}}\right)$$

$$= \left(\frac{1}{1+e^{-z}}\right) \cdot \left(\frac{(1+e^{-z})-1}{1+e^{-z}}\right)$$

$$= \left(\frac{1}{1+e^{-z}}\right) \cdot \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right)$$

$$= \left(\frac{1}{1+e^{-z}}\right) \cdot \left(1 - \frac{1}{1+e^{-z}}\right)$$

$$= \sigma(z) \cdot (1 - \sigma(z))$$

2.1.2 Theory 2 Proof:

By the chain rule,

$$\frac{\delta \sigma(\mathbf{w}^T\mathbf{x} + b)}{\delta \mathbf{w}} = \frac{d}{dz}\sigma(\mathbf{w}^T\mathbf{x} + b) \cdot \frac{d}{dw}(\mathbf{w}^T\mathbf{x} + b)$$

From the previous proof (Theory 1 Proof), we know that $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$. So, we now have

$$\frac{\delta \sigma(\mathbf{w}^T\mathbf{x} + b)}{\delta \mathbf{w}} = (\sigma(\mathbf{w}^T\mathbf{x} + b) \cdot (1 - \sigma(\mathbf{w}^T\mathbf{x} + b)) \cdot \frac{d}{d\mathbf{w}}(\mathbf{w}^T\mathbf{x} + b)$$

Now, we must take the derivative $\frac{d}{d\mathbf{w}}(\mathbf{w}^T\mathbf{x} + b)$ which results in

$$\frac{\delta \sigma(\mathbf{w}^T\mathbf{x} + b)}{\delta \mathbf{w}} = \sigma(\mathbf{w}^T\mathbf{x} + b) \cdot (1 - \sigma(\mathbf{w}^T\mathbf{x} + b)) \cdot x$$

2.2 From documents to feature vectors

This section illustratess the prototypical components of machine learning pipeline for an NLP task, in this case document classification:

1. Read document examples (train, devel, test) from files with a predefined format:
   - assume one document per line, usign the format "<label> <text>".
2. Tokenize each document:
   - using a spaCy tokenizer.
3. Feature extractors:
   - so far, just words.
4. Process each document into a feature vector:
   - map document to a dictionary of feature names.
   - map feature names to unique feature IDs.
   - each document is a feature vector, where each feature ID is mapped to a feature value (e.g. word occurences).

[1]:
```
import spacy
from spacy.lang.en import English
from scipy import sparse
from sklearn.linear_model import LogisticRegression
```

[2]:
```
# Create spaCy tokenizer.
spacy_nlp = English()

def spacy_tokenizer(text):
    tokens = spacy_nlp.tokenizer(text)

    return [token.text for token in tokens]
```

[3]:
```
def read_examples(filename):
    X = []
    Y = []
    with open(filename, mode = 'r', encoding = 'utf-8') as file:
        for line in file:
            [label, text] = line.rstrip().split(' ', maxsplit = 1)
            X.append(text)
            Y.append(label)
    return X, Y
```

[4]:
```
def word_features(tokens):
    feats = {}
    for word in tokens:
        feat = 'WORD_%s' % word
        if feat in feats:
            feats[feat] +=1
        else:
            feats[feat] = 1
```

```
    return feats
```

[5]:
```
def add_features(feats, new_feats):
    for feat in new_feats:
        if feat in feats:
            feats[feat] += new_feats[feat]
        else:
            feats[feat] = new_feats[feat]
    return feats
```

This function tokenizes the document, runs all the feature extractors on it and assembles the extracted features into a dictionary mapping feature names to feature values. It is important that feature names do not conflict with each other, i.e. **different features should have different names**. Each document will have its own dictionary of features and their values.

[6]:
```
def docs2features(trainX, feature_functions, tokenizer):
    examples = []
    count = 0
    for doc in trainX:
        feats = {}

        tokens = tokenizer(doc)

        for func in feature_functions:
            add_features(feats, func(tokens))

        examples.append(feats)
        count +=1

        if count % 100 == 0:
            print('Processed %d examples into features' % len(examples))

    return examples
```

[7]:
```
# This helper function converts feature names to unique numerical IDs.

def create_vocab(examples):
    feature_vocab = {}
    idx = 0
    for example in examples:
        for feat in example:
            if feat not in feature_vocab:
                feature_vocab[feat] = idx
                idx += 1

    return feature_vocab
```

[8]:
```
# This helper function converts a set of examples from a dictionary of feature␣
 ↪names to values representation
# to a sparse representation of feature ids to values. This is important␣
 ↪because almost all feature values will
# be 0 for most documents and it would be wasteful to save all in memory.

def features_to_ids(examples, feature_vocab):
    new_examples = sparse.lil_matrix((len(examples), len(feature_vocab)))
    for idx, example in enumerate(examples):
        for feat in example:
            if feat in feature_vocab:
                new_examples[idx, feature_vocab[feat]] = example[feat]

    return new_examples
```

[9]:
```
# Evaluation pipeline for the Logistic Regression classifier.

def train_and_test(trainX, trainY, devX, devY, feature_functions, tokenizer):
    # Pre-process training documents.
    trainX_feat = docs2features(trainX, feature_functions, tokenizer)

    # Create vocabulary from features in training examples.
    feature_vocab = create_vocab(trainX_feat)
    print('Vocabulary size: %d' % len(feature_vocab))

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    # Train LR model.
    lr_model = LogisticRegression(penalty = 'l2', C = 1.0, solver = 'lbfgs',␣
 ↪max_iter = 1000)
    lr_model.fit(trainX_ids, trainY)

    # Pre-process test documents.
    devX_feat = docs2features(devX, feature_functions, tokenizer)
    devX_ids = features_to_ids(devX_feat, feature_vocab)

    # Test LR model.
    print('Accuracy: %.3f' % lr_model.score(devX_ids, devY))
```

[10]:
```
import os

datapath = '../data'

train_file = os.path.join(datapath, 'imdb_sentiment_train.txt')
trainX, trainY = read_examples(train_file)

dev_file = os.path.join(datapath, 'imdb_sentiment_dev.txt')
```

```
devX, devY = read_examples(dev_file)

# Specify features to use.
features = [word_features]

# Evaluate LR model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 28692
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.837
```

2.3 Feature engineering

Evaluate LR model performance when adding positive and negative lexicon features. We will be using Bing Liu's sentiment lexicons from https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html

2.3.1 Read the positive and negative sentiment lexicons.

There should be 2006 entries in the positive lexicon and 4783 entries in the positive lexicon.

[11]:
```
def read_lexicon(filename):
    lexicon = set()
    with open(filename, mode = 'r', encoding = 'ISO-8859-1') as file:
        # YOUR CODE HERE
        for line in file:
            if line.strip():
                line = line.strip()
                if line[0] != ";":
                    lexicon.add(line)
    return lexicon

lexicon_path = '../data/bliu'

poslex_file = os.path.join(lexicon_path, 'positive-words.txt')
neglex_file = os.path.join(lexicon_path, 'negative-words.txt')

poslex = read_lexicon(poslex_file)
neglex = read_lexicon(neglex_file)


print(len(poslex), 'entries in the positive lexicon.')
print(len(neglex), 'entries in the negative lexicon.')
```

```
2006 entries in the positive lexicon.
4783 entries in the negative lexicon.
```

2.3.2 Use the lexicons to create two lexicon features

- A feature 'POSLEX' whose value indicates how many tokens belong to the positive lexicon.
- A feature 'NEGLEX' whose value indicates how many tokens belong to the negative lexicon.

[12]:
```
def two_lexicon_features(tokens):
    feats = {'POSLEX': 0, 'NEGLEX': 0}
    # YOUR CODE HERE
    for tok in tokens:
        #print(tok)
        if tok in poslex:
            feats['POSLEX'] += 1
        if tok in neglex:
            feats['NEGLEX'] += 1
    #print(feats)
    return feats
```

Evaluate the LR model using the two new lexicon features. Expected accuracy is around 83.8%.

[13]:
```
# Specify features to use.
features = [word_features, two_lexicon_features]

# Evaluate LR model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 28694
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.838
```

2.3.3 Create a separate feature for each word that appears in each lexicon

- If a word from the positive lexicon (e.g. 'like') appears N times in the document (e.g. 5 times), add a positive lexicon feature 'POSLEX_word' for that word that is associated that value (e.g. {'POSLEX_like' : 5}.
- Similarly, if a word from the negative lexicon (e.g. 'dislike') appears N times in the document (e.g. 5 times), add a negative lexicon feature 'NEGLEX_word' for that word that is associated that value (e.g. {'NEGLEX_dislike' : 5}.

[14]:
```
def lexicon_features(tokens):
    feats = {}
    # YOUR CODE HERE
    # Assume the positive and negative lexicons are available in poslex and␣
 ↪neglex, respectively.
    for tok in tokens:
        if tok in poslex:
            #print(str('POSLEX_'+tok))
            if str('POSLEX_'+tok) in feats:
                feats[str('POSLEX_'+tok)] += 1
            if str('POSLEX_'+tok) not in feats:
                feats[str('POSLEX_'+tok)] = 1
        if tok in neglex:
            if str('NEGLEX_'+tok) in feats:
                feats[str('NEGLEX_'+tok)] += 1
            if str('NEGLEX_'+tok) not in feats:
                feats[str('NEGLEX_'+tok)] = 1

    #print(feats)
    return feats
```

Evaluate the LR model using the new per-lexicon word features. Expected accuracy is arpund 83.9%.

[15]:
```
# Specify features to use.
features = [word_features, lexicon_features]

# Evaluate LR model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 31722
Processed 100 examples into features
Processed 200 examples into features
```

```
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.839
```

2.3.4 Add document length feature

Add a feature 'DOC_LEN' whose value is the natural logarithm of the document length (use *math.log* to compute logarithms).

[16]:
```
import math
def len_feature(tokens):

    # Get 83.9% if using math.log
    feat = {'DOC_LEN': math.log(len(tokens))}

    # Get exactly 84.0% if not using math.log
    #feat = {'DOC_LEN': (len(tokens))}

    return feat
```

Evaluate the LR model using the new document length feature. Expected accuracy is around 84.0%.

[17]:
```
# Specify features to use.
features = [word_features, lexicon_features, len_feature]

# Evaluate LR model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
```

```
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 31723
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.839
```

2.3.5 Add deictic features

Add a feature 'DEICTIC_COUNT' that counts the number of 1st and 2nd person pronouns in the document.

[18]:
```
def deictic_feature(tokens):
    pronouns = set(('i', 'my', 'me', 'we', 'us', 'our', 'you', 'your'))
    count = 0

    # YOUR CODE HERE
    for tok in tokens:
        if tok in pronouns:
            count += 1

    return {'DEICTIC_COUNT': count}
```

Evaluate the LR model using the deictic features. Expected accuracy is around 84.2%.

[19]:
```
# Specify features to use.
features = [word_features, lexicon_features, len_feature, deictic_feature]

# Evaluate LR model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer)
```

```
Processed 100 examples into features
```

```
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 31724
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.841
```

Let's try without the word features. Expected accuracy is around 80.4%.

[20]:
```
# Specify features to use.
features = [lexicon_features, len_feature, deictic_feature]

# Evaluate LR model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
```

```
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 3032
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.806
```

2.4 5111: Pre-processing for negation

Mandatory for graduate students, optional for undergraduate students

Preprocess the tokens to account for negation, as explained on slide 36 in the LR lecture, and integrate in the model that uses `word_features`, `lexicon_features`, `len_feature`, and `deictic_feature`.

- Pre-process the text for all negation words contained in the lexicon `../data/negation_words.txt`.
- Need to rewrite the sentiment lexicon features such that whenever modified by a negation word, a positive sentiment word is counted as negative, i.e. 'NOT_like' will be a negative sentiment token. The prefix 'NOT_' should be added irrespective of the actual negative word used, e.g. 'not', 'never', etc.
  - For bonus points, you can also run evaluations where the actual negative word is used as a prefix, e.g. 'never' before 'like' would lead to a feature called 'NEVER_like'.
- Train and evaluate the performance of the new model.

[21]:
```python
import string

# Preprocess the text for all negation words contained in the lecixon
negation_words = read_lexicon('../data/negation_words.txt')
```

```
# Rewrite lexicon_features such that whenever modified by a negation word, a␣
 ↪positive sentiment word is counted negative
def lexicon_features_negation(tokens):
    feats = {}
    # YOUR CODE HERE
    # Assume the positive and negative lexicons are available in poslex and␣
 ↪neglex, respectively

    tokens = list(tokens)

    negation_flag = False

    for idx, tok in enumerate(tokens):

        if tok in negation_words:
            negation_flag = True
            negation_word = tok

        if tok in string.punctuation:
            negation_flag = False

        if tok in poslex:
            if negation_flag:
                if 'NEGLEX_'+negation_word.upper()+'_'+tok in feats:
                    feats['NEGLEX_'+negation_word.upper()+'_'+tok] += 1
                if 'NEGLEX_'+negation_word.upper()+'_'+tok not in feats:
                     feats['NEGLEX_'+negation_word.upper()+'_'+tok] = 1
            if not negation_flag:
                #print(str('POSLEX_'+tok))
                if str('POSLEX_'+tok) in feats:
                    feats[str('POSLEX_'+tok)] += 1
                if str('POSLEX_'+tok) not in feats:
                    feats[str('POSLEX_'+tok)] = 1
        if tok in neglex:
            if str('NEGLEX_'+tok) in feats:
                feats[str('NEGLEX_'+tok)] += 1
            if str('NEGLEX_'+tok) not in feats:
                feats[str('NEGLEX_'+tok)] = 1

    #print(feats)
    return feats
```

[22]:
```
# Specify features to use.
features = [lexicon_features_negation, word_features, lexicon_features,␣
 ↪len_feature, deictic_feature]

# Evaluate LR model.
```

```
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 32353
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.848
```

2.5 5111: Compute learning curve

Mandatory for graduate students, optional for undergraduate students

- Select the best performing model and plot its accuracy vs. number of training examples. Vary the number of training examples by selecting for each class the first N examples in the file, where $N \in \{50, 100, 150, 250, 350, 450, 550, 650, 750\}$. For example, the first 50 positive examples would be inb `X[:50]`, whereas the first 50 negative examples would be in `X[750:800]`.

[23]:
```
# YOUR CODE HERE
accuracy = []
```

```
def train_and_test_new(trainX, trainY, devX, devY, feature_functions,␣
 ↪tokenizer):
    # Pre-process training documents.
    trainX_feat = docs2features(trainX, feature_functions, tokenizer)

    # Create vocabulary from features in training examples.
    feature_vocab = create_vocab(trainX_feat)
    print('Vocabulary size: %d' % len(feature_vocab))

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    # Train LR model.
    lr_model = LogisticRegression(penalty = 'l2', C = 1.0, solver = 'lbfgs',␣
 ↪max_iter = 1000)
    lr_model.fit(trainX_ids, trainY)

    # Pre-process test documents.
    devX_feat = docs2features(devX, feature_functions, tokenizer)
    devX_ids = features_to_ids(devX_feat, feature_vocab)

    # Test LR model.
    acc = lr_model.score(devX_ids, devY)
    accuracy.append(acc)
    print('Accuracy: %.3f' % acc)

N = [50, 100, 150, 250, 350, 450, 550, 650, 750]

for increment in N:

    new_trainX = trainX[:increment] + trainX[750:750+increment]
    new_trainY = trainY[:increment] + trainY[750:750+increment]

    new_devX = devX[:increment] + devX[750:750+increment]
    new_devY = devY[:increment] + devY[750:750+increment]

    # Specify features to use.
    features = [word_features, lexicon_features, len_feature, deictic_feature]

    # Evaluate LR model.
    train_and_test_new(new_trainX, new_trainY, new_devX, new_devY, features,␣
 ↪spacy_tokenizer)

print("List of model accuracies: ", accuracy)
```

```
Processed 100 examples into features
Vocabulary size: 6308
Processed 100 examples into features
```

```
Accuracy: 0.700
Processed 100 examples into features
Processed 200 examples into features
Vocabulary size: 9425
Processed 100 examples into features
Processed 200 examples into features
Accuracy: 0.745
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Vocabulary size: 12648
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Accuracy: 0.740
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Vocabulary size: 17283
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Accuracy: 0.778
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Vocabulary size: 20746
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Accuracy: 0.797
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
```

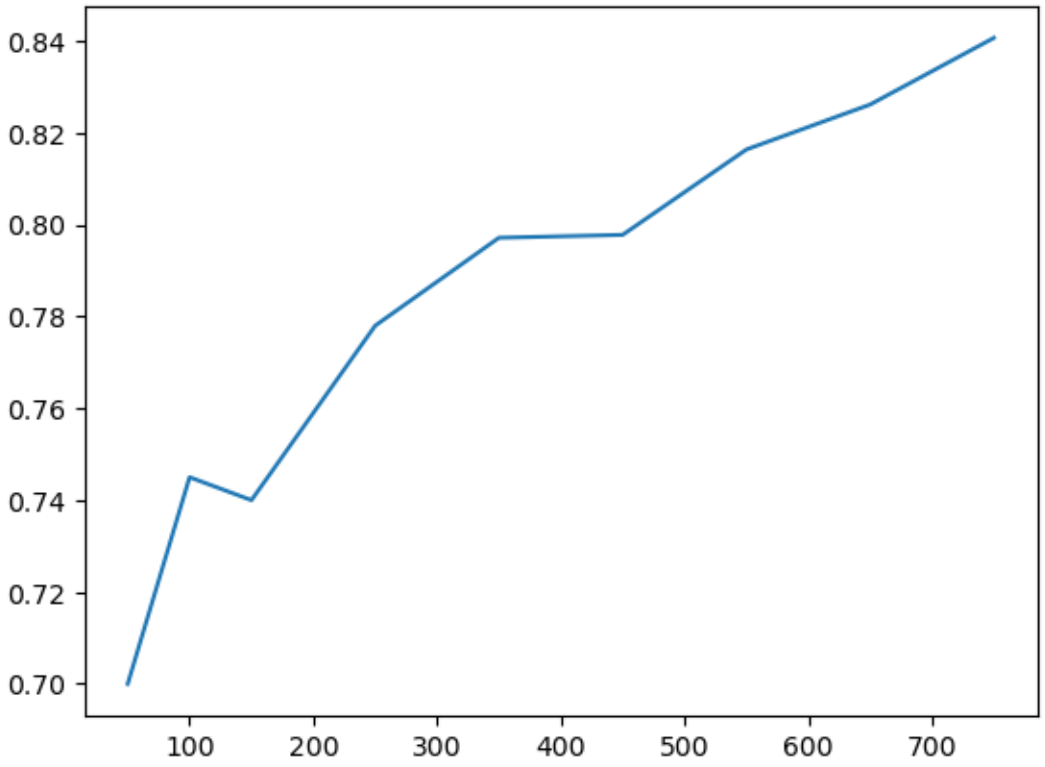
```
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Vocabulary size: 23881
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Accuracy: 0.798
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Vocabulary size: 26927
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Accuracy: 0.816
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
```

```
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Vocabulary size: 29452
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Accuracy: 0.826
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 31724
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
```

```
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.841
List of model accuracies:  [0.7, 0.745, 0.74, 0.778, 0.7971428571428572,
0.7977777777777778, 0.8163636363636364, 0.8261538461538461, 0.8406666666666667]
```

[24]:
```
import matplotlib.pyplot as plt

plt.plot(N, accuracy)
plt.show()
```



2.6 Bonus points

Anything extra goes here. For example, can you do feature engineering or hyper-parameter tuning such that accuracy gets over 85%? The larger the gain in accuracy, the more bonus points awarded. - Evaluate the impact of other features, such as the presence of exclamation points, or replacing word features with lemma features. - Determine the importance of counts by using binary word features instead of count features, i.e. does the word appear or not in the document, instead of how many times. - Simple feature selection: use only features that appear at least K times in the training data (try K = 3, K = 5). * Also evaluate the impact of feature selection when usign smaller valeus for the `C` hyper-parameter for L2 regularization. - Look at the mistakes the model made (error analysis) and see if you can design new features to address the more common mistakes.

[25]:
```
# YOUR CODE HERE
```

2.7 Analysis

Include an analysis of the results that you obtained in the experiments above.

Error analysis: Do some basic error analysis where you try to explain the mistakes that the best model made and provide ideas for possible features that would alleviate these mistakes.

[5111] Interpretability: From each class of features take 2 features that you think should be strongly correlated with the positive or negative label, and determine if the model learned a corresponding parameter that correctly expresses this correlation. For example, the feature 'WORD_loved' is expected to be very correlated with the positive label, as such the model should learn a corresponding large positive weight. - *Hint: for this, you may consider using the* `coef_` *attribute of the LogisticRegression class.*

2.7.1 Error Analysis

[26]:
```
# Error analysis on the 5111 model. Accuracy is around 84.8%.
    # Rerun the model here to get the model back in current memory
    # Use model.predict
    # Compare predictions to ground truth --> if y_pred != y_true, print both,␣
 ↪talk about why !=
    # Discuss possible features to fix
```

[27]:
```
# Rerun the model here to get the model back in current memory for lr_model

features = [lexicon_features_negation, word_features, lexicon_features,␣
 ↪len_feature, deictic_feature]
trainX_feat = docs2features(trainX, features, spacy_tokenizer)

# Create vocabulary from features in training examples.
feature_vocab = create_vocab(trainX_feat)
print('Vocabulary size: %d' % len(feature_vocab))

trainX_ids = features_to_ids(trainX_feat, feature_vocab)

# Train LR model.
lr_model = LogisticRegression(penalty = 'l2', C = 1.0, solver = 'lbfgs',␣
 ↪max_iter = 1000)
lr_model.fit(trainX_ids, trainY)

# Pre-process test documents.
devX_feat = docs2features(devX, features, spacy_tokenizer)
devX_ids = features_to_ids(devX_feat, feature_vocab)

# Test LR model.
print('Accuracy: %.3f' % lr_model.score(devX_ids, devY))
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 32353
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.848
```

[28]:
```
# Get test data
test_file = os.path.join(datapath, 'imdb_sentiment_test.txt')
testX, testY = read_examples(test_file)

# Pre-process test documents.
testX_feat = docs2features(testX, features, spacy_tokenizer)
testX_ids = features_to_ids(testX_feat, feature_vocab)

# Use model to make predictions for testY using testX
pred_testY = lr_model.predict(testX_ids)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
```

```
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
```

[29]:
```
from sklearn.metrics import accuracy_score

#Compare predictions to ground truth
results = accuracy_score(testY, pred_testY)
print("Accuracy of the model on test data: ", results)

# Show some cases where model made mistakes
count = 0
for i in range(len(testY)):
    if testY[i] != pred_testY[i] and count < 10:
        print("Test case ", i, " with class ", testY[i], " was incorrectly␣
 ↪classified as ", pred_testY[i])
        count += 1
```

```
Accuracy of the model on test data:  0.822
Test case  23  with class  pos  was incorrectly classified as  neg
Test case  47  with class  pos  was incorrectly classified as  neg
Test case  48  with class  pos  was incorrectly classified as  neg
Test case  49  with class  pos  was incorrectly classified as  neg
Test case  55  with class  pos  was incorrectly classified as  neg
Test case  81  with class  pos  was incorrectly classified as  neg
Test case  84  with class  pos  was incorrectly classified as  neg
Test case  85  with class  pos  was incorrectly classified as  neg
Test case  96  with class  pos  was incorrectly classified as  neg
Test case  99  with class  pos  was incorrectly classified as  neg
```

[30]:
```
print("Data:", testX[47], '\n')
print("Correct Class: ", testY[47], '\n')
print("Predicted Class: ", pred_testY[47], '\n')
```

```
Data: I particularly enjoyed Delly's review of this film and agree that Howard
is not the only "damaged" character. Howard is rather ruthlessly "set-up" by the
script, but there is no evidence that his previous employer is actually dead or,
if she is, that he murdered her. Howard doesn't know and neither do we. In
terror and confusion at seeing the woman lying there, he bolts. However, he
never actually harms Helen Gordon, no matter how enraged he is. Indeed, he
```

reacts with horror at Helen's fainting spell and the fact that he is holding a pair of scissors…then he resumes his tidying up and greets the recovered Helen with the almost pathetic " I'm very tired now. I think I'll go home". Frankly, I don't think he's a psychopath. A sick puppy, certainly, but not a psychopath.

The problem with Howard is that he has no real male identity. He wanted to serve his country, but his mental condition denies him a place in the army. He is singularly rootless and isolated: no wife, no girl, no home (again, at least as far as we know). And, he does a woman's job - "Floor's are my speciality". Helen's niece ruthlessly strips away this pride in his thoroughness by exclaiming caustically that she would want a man with a real job. Also, although he finds himself strongly attracted to Helen, he is unable or unwilling to do more than scare her by making a strong sexual pass. He is remarkably powerless - can't fight, can't work, can't make love.

Helen is justifiably terrified, however. She tries to connect to him but, finding that he doesn't respond normally (i.e. way outside the comfort zone provided by her rose-tinted memories of husband Ned), unwittingly presses all Howard's buttons by lying to him in her attempt to escape.

Both characters, trapped in the house, trapped by fear, neuroses, rage and memory, deserve sympathy. I know the sudden ending has disappointed some reviewers, but I felt it fitted well, as it offered a kind of release to the characters. Helen is freed, I think, from the past. When Howard tries on her husband's army coat, Helen's disgusted reaction is highlighted. She no doubt feels that the "sacredness" of Ned's possessions has been violated but, hopefully, her need to keep everything "untouched" has been lost in the reality of her own struggle with danger. Perhaps she can move on.

Howard is also freed - from his endless cycle of anger, hurt and violence. Whether he moves on to treatment or to jail is debatable, but I hope it's the former.

Great performances from Ryan and Lupino. I prefer "On Dangerous Ground", but this is pretty good too.

Correct Class: pos

Predicted Class: neg

[31]:
```
# Print the information for some of these incorrectly predicted cases
print("Data:", testX[48], '\n')
print("Correct Class: ", testY[48], '\n')
print("Predicted Class: ", pred_testY[48], '\n')
```

Data: "Hit and Run" is a shattering story starring the always wonderful Margaret Colin as a society lady who "has it all" until she hits a child with her car and leaves the scene. Hence the title. The tragedy is that she goes to call for help and returns, but is frightened away by angry passers-by who think the hitter abandoned the scene. This was made in the days when not everyone had a cell phone or there wouldn't be a story.

Colin's guilt and anguish are palpable and cause her to act so strangely that a detective gets onto her right away. Her lies sink her deeper and deeper into a self-loathing hole, causing her to make a bad situation worse.

This is a very thought-provoking

```
story, and one can't help but to feel this lady's pain, wishing throughout that
she would simply come clean.<br /><br />As a TV movie, thanks to Colin and a
strong script, this is a well above average TV movie.

Correct Class:  pos

Predicted Class:  neg
```

[32]:
```
print("Data:", testX[99], '\n')
print("Correct Class: ", testY[99], '\n')
print("Predicted Class: ", pred_testY[99], '\n')
```

```
Data: Good, boring or bad? It's good. Worth your money? If you can spare it for
a ticket, sure. Better than the trailer makes it seem? Yes, oddly.<br /><br
/>There isn't much to the script - Guards working at armored truck company move
vast amounts of cash. Guards see opportunity to retire as millionaires, one of
them is too honest to go along with it all, and a well-laid plan goes to hell.
<br /><br />This could have been a poorly-executed Reservoir Dogs ripoff, but
the skill of the cast and the director's ability to make just about anything
tense pull it out of that realm and put it onto a solid footing.

Correct Class:  pos

Predicted Class:  neg
```

2.7.2 Explain the Mistakes and provide ideas for possible features that would alleviate these mistakes.

It seems that in some cases the model is incorrectly classifying some of the movie reviews as negative due to the presence of negative words in these reviews. This is a mistake because although there are negative words in the review, they are used to describe the plot or story contents of the movie, not the reviewer's opinion of the movie. For example, in the second example of misclassification that is displayed, the sentence segment "make a bad situation worse," contains negative words "bad" and "worse." However, words are used to describe a portion of the story, not an opinon on the movie.

One possible way to alleviate these mistakes would be to create a feature that describes whether the words in the review discuss the plot of the movie or the reviewer's opinion of the movie. This way, the model could better differentiate between opinions and descriptions. Perhaps the words that are used to describe the movie plot rather than the reviewer's opinion could be given a different weight, or even no weight.

2.7.3 Interpretability

[33]:
```
# Take 2 features from each class:
# (word_features, lexicon_features, len_feature, deictic_feature,␣
 ↪lexicon_features_negation)
# Explain why they should be positive or negative, did model agree?
# Look into coef_ attribute of Logistic Regression class
```

Feature Class: word_features

[34]:
```
#print(len(feature_vocab))
#print(feature_vocab.keys())

keys = list(feature_vocab.keys())
words = [(idx, key) for idx, key in enumerate(keys) if "WORD" in key]
print(words[10:30])
```

```
[(60, 'WORD_then'), (61, 'WORD_there'), (62, 'WORD_has'), (63, 'WORD_been'),
(64, 'WORD_nothing'), (65, 'WORD_but'), (66, 'WORD_an'), (67,
'WORD_unbearable'), (68, 'WORD_streak'), (69, 'WORD_of'), (70,
'WORD_Hollywood'), (71, 'WORD_trash'), (72, 'WORD_barely'), (73, 'WORD_good'),
(74, 'WORD_enough'), (75, 'WORD_for'), (76, 'WORD_a'), (77, 'WORD_blockbuster'),
(78, 'WORD_night'), (79, 'WORD_,')]
```

[35]:
```
print('Coef for word_features word "good":', lr_model.coef_[0][73])
print('Coef for word_features word "unbearable":',lr_model.coef_[0][67])
```

```
Coef for word_features word "good": 0.11599261751211078
Coef for word_features word "unbearable": 0.056787670600894895
```

Note: positive coefficient values indicate a correlation to the positive class while negative coefficient values indicate a correlation to the negative class.

> The word_features word “good” shuold have a fairly positive corresponding parameter. We can see that the model has assigned a weight of around 0.116 for this word feature. Since “good” is not a particularly strong positive word, this could be a reasonable weight for the model to assign.
>
> The word_features word “unbearable” should have a strong negative corresponding parameter. However, we can see that the model has assigned a positive weight of around 0.0568 for this word feature. This is not a good weight to assign to such a strong negative word. It is likely that this weight was assigned due to the model being trained on many reviews that contain descriptions of the movie rather than viewer’s opinions, as discussed in the previous section of Analysis.

Feature Class: lexicon_features

[36]:
```
lex = [(idx, key) for idx, key in enumerate(keys) if "NEGLEX" or "POSLEX" in␣
 ↪key]
print(lex[:30])
```

```
[(0, 'POSLEX_decent'), (1, 'NEGLEX_unbearable'), (2, 'NEGLEX_trash'), (3,
'NEGLEX_BARELY_good'), (4, 'NEGLEX_BARELY_enough'), (5,
'NEGLEX_BARELY_blockbuster'), (6, 'NEGLEX_disappointment'), (7,
'POSLEX_genius'), (8, 'NEGLEX_drained'), (9, 'NEGLEX_bad'), (10,
'POSLEX_delicious'), (11, 'NEGLEX_ruthless'), (12, 'POSLEX_enjoyable'), (13,
'NEGLEX_unfortunate'), (14, 'POSLEX_important'), (15, 'POSLEX_better'), (16,
'NEGLEX_delayed'), (17, 'POSLEX_breeze'), (18, 'NEGLEX_fails'), (19,
'POSLEX_good'), (20, 'POSLEX_beauty'), (21, 'NEGLEX_cheap'), (22,
'POSLEX_classic'), (23, 'NEGLEX_desperation'), (24, 'NEGLEX_regret'), (25,
'POSLEX_cheer'), (26, 'POSLEX_entertaining'), (27, 'POSLEX_convincing'), (28,
'POSLEX_sweet'), (29, 'POSLEX_sincerely')]
```

[37]:
```
print('Coef for lexicon_features word "entertaining":', lr_model.coef_[0][26])
print('Coef for lexicon_features word "disappointment":',lr_model.coef_[0][6])
```

```
Coef for lexicon_features word "entertaining": 0.47521974922720783
Coef for lexicon_features word "disappointment": -0.37964694326948445
```

The lexicon_features word “entertaining” should have a fairly strong positive corresponding parameter. We can see that the model has assigned a weight of around 0.475 for this lexicon feature. Since “entertaining” is a fairly strong positive word, this could be a reasonable weight for the model to assign.

The lexicon_features word “disappointment” should have a fairly strong negative corresponding parameter. We can see that the model has assigned a weight of around -0.3796 for this lexicon feature. Since “disappoinment” is a fairly strong negative word, this could be a reasonable weight for the model to assign.

Feature Class: lexicon_features_negation

[38]:
```
negat = [(idx, key) for idx, key in enumerate(keys) if 'NEGLEX_NOT' in key]
print(negat[:50])
```

```
[(44, 'NEGLEX_NOT_humor'), (1028, 'NEGLEX_NOT_perfect'), (1212,
'NEGLEX_NOT_like'), (1460, 'NEGLEX_NOT_greatest'), (2090, 'NEGLEX_NOT_best'),
(2149, 'NEGLEX_NOT_good'), (2226, 'NEGLEX_NOTHING_leading'), (2670,
'NEGLEX_NOT_well'), (3047, 'NEGLEX_NOTHING_nice'), (3094,
'NEGLEX_NOTHING_flashy'), (3807, 'NEGLEX_NOT_spectacular'), (3943,
'NEGLEX_NOT_top'), (3944, 'NEGLEX_NOT_satisfying'), (4122,
'NEGLEX_NOT_available'), (4238, 'NEGLEX_NOT_classic'), (4306,
'NEGLEX_NOTHING_like'), (4312, 'NEGLEX_NOT_suffice'), (4400,
'NEGLEX_NOT_right'), (4583, 'NEGLEX_NOT_stunning'), (4586,
'NEGLEX_NOT_comfortable'), (5135, 'NEGLEX_NOT_realistic'), (5164,
'NEGLEX_NOT_golden'), (5282, 'NEGLEX_NOT_worth'), (5340,
'NEGLEX_NOT_authentic'), (5563, 'NEGLEX_NOT_steady'), (5657,
'NEGLEX_NOTHING_amazing'), (5679, 'NEGLEX_NOTHING_revolutionary'), (5685,
'NEGLEX_NOT_famous'), (5842, 'NEGLEX_NOT_great'), (5843, 'NEGLEX_NOT_better'),
(5990, 'NEGLEX_NOTHING_excited'), (5996, 'NEGLEX_NOT_brilliant'), (6144,
'NEGLEX_NOT_interesting'), (6246, 'NEGLEX_NOT_exceeded'), (6318,
```

```
'NEGLEX_NOT_talent'), (7153, 'NEGLEX_NOT_smooth'), (7259, 'NEGLEX_NOT_excite'),
(7260, 'NEGLEX_NOT_awe'), (7261, 'NEGLEX_NOT_precise'), (7262,
'NEGLEX_NOT_skill'), (7306, 'NEGLEX_NOT_beauty'), (7456, 'NEGLEX_NOT_wealthy'),
(7791, 'NEGLEX_NOT_advantage'), (7792, 'NEGLEX_NOT_survivor'), (7872,
'NEGLEX_NOT_lead'), (7873, 'NEGLEX_NOT_freedom'), (8016,
'NEGLEX_NOT_patriotic'), (8183, 'NEGLEX_NOT_work'), (8373, 'NEGLEX_NOT_lover'),
(8444, 'NEGLEX_NOT_significant')]
```

[39]:
```
print('Coef for lexicon_features_negation word "NEGLEX_NOT_good":', lr_model.
  ↪coef_[0][2149])
print('Coef for lexicon_features_negation word "NEGLEX_NOT_worth":',lr_model.
  ↪coef_[0][5282])
```

```
Coef for lexicon_features_negation word "NEGLEX_NOT_good": 0.003328795388746331
Coef for lexicon_features_negation word "NEGLEX_NOT_worth": -0.23318610196441694
```

The lexicon_features_negation word "NEGLEX_NOT_good" should have a negative corresponding parameter. We can see that the model has assigned a weight of around 0.003 for this lexicon negation feature. Since NOT_good is a fairly negative word, it should have a negative corresponding parameter. However, since "good" is a positive word on its own, this could be a reasonable weight for the model to assign.

The lexicon_features_negation word "NEGLEX_NOT_worth" should have a fairly strong negative corresponding parameter. We can see that the model has assigned a weight of around -0.233 for this lexicon negation feautre. Since "NOT_worth" is a fairly strong negative word, this could be a reasonable weight for the model to assign.

Feature Class: deictic_features

[40]:
```
d = [(idx, key) for idx, key in enumerate(keys) if 'DEICTIC_COUNT' in key]
print(d)
```

```
[(360, 'DEICTIC_COUNT')]
```

[41]:
```
print('Coef for "DEICTIC_COUNT":', lr_model.coef_[0][360])
```

```
Coef for "DEICTIC_COUNT": 0.23535709417175646
```

There is only one feature listed for this feature class. Regardless, I do not think that the deictic count would have any strong correlation to any one class, positive or negative. However, we can see that the model has assigned a positive weight to this feature. This could mean that reviews with a higher deictic count tend to be positive.

Feature Class: len_features

[42]:
```
len_f = [(idx, key) for idx, key in enumerate(keys) if 'DOC_LEN' in key]
print(len_f)
```

```
[(359, 'DOC_LEN')]
```

[43]:
```
print('Coef for "DOC_LEN":', lr_model.coef_[0][359])
```

```
Coef for "DOC_LEN": -0.15856565759520008
```

There is only one feature listed for this feature class. Regardless, I do not think that the document length would have any strong correlation to any one class, positive or negative. However, we can see that the model has assigned a negative weight to this feature. This could mean that reviews of longer length tend to be negative.

[ ]: