LR-SentimentAnalysis

March 19, 2024

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:

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 (10 + 10p)

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

2.2 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).

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

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

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

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

[ ]:
```
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
```

[ ]:
```
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
```

[ ]:
```
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.

[ ]:
```
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
```

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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))
```

[ ]:
```
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)
```

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 (0p)

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

[ ]:
```
def read_lexicon(filename):
    lexicon = set()
    with open(filename, mode = 'r', encoding = 'ISO-8859-1') as file:
        # YOUR CODE HERE


    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.')
```

2.3.2 Use the lexicons to create two lexicon features (15p)

- 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.

[ ]:
```
def two_lexicon_features(tokens):
    feats = {'POSLEX': 0, 'NEGLEX': 0}
    # YOUR CODE HERE
```

```
    return feats
```

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

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

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

2.3.3 Create a separate feature for each word that appears in each lexicon (20p)

- 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}.

[ ]:
```
def lexicon_features(tokens):
    feats = {}
    # YOUR CODE HERE
    # Assume the positive and negative lexicons are available in poslex and␣
 ↪neglex, respectively.



    return feats
```

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

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

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

2.3.4 Add document length feature (10p)

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

[ ]:
```
import math
def len_feature(tokens):
    feat = {'DOC_LEN': 'YOUR CODE HERE'}
```

```
    return feat
```

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

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

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

2.3.5 Add deictic features (15p)

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

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

    # YOUR CODE HERE



    return {'DEICTIC_COUNT': count}
```

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

[ ]:
```
# 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)
```

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

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

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

2.4 5111: Pre-processing for negation (5p)

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.

[1]:
```
# YOUR CODE HERE
```

2.5 5111: Compute learning curve (15p)

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]`.

[2]:
```
# YOUR CODE HERE
```

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. - Replace the spaCy tokenizer with the tiktoken BPE tokenizer and see impact on performance (accuracy).

[3]:
```
# YOUR CODE HERE
```

2.7 Analysis

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

Error analysis (10p): 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 (10p): 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.*