SentimentAnalysis

October 1, 2024

1 Naive Bayes for Sentiment Analysis

This assignment is comprised of two parts:

1. **Theory**: Solve the Naive Bayes exercises 4.1 and 4.2 from Chapter 4 in the J&M textbook. Reformulate NB to emphasize title words.
2. **Implementation**: You will implement and experiment with various feature engineering techniques in the context of Naive Bayes models for Sentiment classification of movie reviews.

We will use the NB model implemented in sklearn:

https://scikit-learn.org/stable/modules/naive_bayes.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 all 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 *SentimentAnalysis.pdf* showing the code and the output of all cells, and save it in the same folder that contains the notebook file *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.

Make sure that you format your solutions to theory questions to show equations properly. We will not grade solutions that are not properly formatted. Jupyter-notebook understands Latex, alternatively you can edit in Word using its Equation editor and submit the PDF as a separate file in Canvas.

3 Theory (100p + 20p)

3.1 Theory: J&M Exercise 4.1 (40p)

Your solution goes here …

3.2 Theory: J&M Exercise 4.2 (60p)

Your solution goes here …

3.3 Bonus: Title is K times more important than Body (20p)

Mandatory for graduate students, optional for undergraduate students

The Naive Bayes algorithm for text categorization presented in class treats all sections of a document equally, ignoring the fact that words in the title are often more important than words in the text in determining the document category. Modify the Naive Bayes training algorithm to reflect that word occurrences in the title are K times more important than word occurrences in the rest of the document for deciding the class, where K is an input parameter. Describe the idea in English and include pseudocode, akin to the training pseudocode shown in class.

Your solution goes here …

4 Implementation (100p + 20p)

4.1 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).

[13]:
```
import spacy
from spacy.lang.en import English
from scipy import sparse
from sklearn.naive_bayes import MultinomialNB
```

[14]:
```
# Create spaCy tokenizer.
spacy_nlp = English()

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

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

[15]:
```
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
```

[16]:
```
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
```

[17]:
```
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.

[18]:
```
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
```

[19]:
```
# 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
```

[20]:
```
# 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
```

[21]:
```
# Evaluation pipeline for the Naive Bayes 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 NB model.
    nb_model = MultinomialNB(alpha = 1.0)
    nb_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 NB model.
    print('Accuracy: %.3f' % nb_model.score(devX_ids, devY))
```

[22]:
```
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 NB 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: 0.779
```

4.2 Feature engineering

[20p] Evaluate NB model performance when using only alpha tokens. This can be done by changing the tokenizer function.

[ ]:
```
def spacy_tokenizer1(text):
    tokens = spacy_nlp.tokenizer(text)

    # YOUR CODE HERE
    # Keep in the tokens list only those whose text is made up solely from␣
 ↪letters.
    tokens = []

    return tokens

# Evaluate NB model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer1)
```

[10p] Same as above, but lowercase all tokens before using as features.

[ ]:
```
def spacy_tokenizer2(text):
    tokens = spacy_nlp.tokenizer(text)

    # YOUR CODE HERE
    # Keep in the tokens list only those whose text is made up solely from␣
 ↪letters.
    # Return a list of lowercased token text.
    tokens == []

    return tokens

# Evaluate NB model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer2)
```

[15p] Same as above, but lowercase only tokens that appear at the beginning of sentences.

[ ]:
```
spacy_nlp = English()
spacy_nlp.add_pipe("sentencizer")

def spacy_tokenizer3(text):
```

```
    doc = spacy_nlp(text)

    tokens = []

    # YOUR CODE HERE


    return tokens

# Evaluate NB model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer3)
```

[20p] Use spacy_tokenizer2 (only alpha tokens, lowered all) and display the top 10 most frequent tokens in the vocabulary, as a list of tuples (token, frequency).

[ ]:
```
# First, count token occurrences across all examples, where features are still␣
 ↪strings.
def create_feature_counts(examples):
    feature_counts = {}

    # YOUR CODE HERE

    return feature_counts

# Create features for all training examples, compute feature counts
def fcounts_from_train(trainX, feature_functions, tokenizer):
    # Pre-process training documents.
    trainX_feat = docs2features(trainX, feature_functions, tokenizer)

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

    return feature_counts

# Return a list of the top K most frequent tokens in the vocabulary.
def topK_tokens(vocab, k):
    # YOUR CODE HERE


vocab = fcounts_from_train(trainX, features, spacy_tokenizer2)
stop_words = topK_tokens(vocab, 20)
for item in stop_words:
    print(item)
```

[20p] Evaluate NB model performance when ignoring the top 20 stop words. Use spacy_tokenizer2 (only alpha tokens, lowered all).

[ ]:
```
# Evaluation pipeline for the Naive Bayes 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_counts = create_feature_counts(trainX_feat)
    stop_words = topK_tokens(vocab, 20)

    # Remove from each example features that appear in the stop words list.
    # YOUR CODE HERE.


    # 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 NB model.
    nb_model = MultinomialNB(alpha = 1.0)
    nb_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 NB model.
    print('Accuracy: %.3f' % nb_model.score(devX_ids, devY))

# Evaluate NB model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer2)
```

[5p] Evaluate NB model performance when ignoring words that appear less than 5 times. Use spacy_tokenizer2 (only alpha tokens, lowered all).

[ ]:

4.3 [Bonus] Binary Multinomial Bayes (20p)

Write code for transforming documents to features such that features are Boolean and only represent whether a word occurred in a document, as in the Binary Multinomial Naive Bayes discussed in class. Evaluate the Naive Bayes model with this feature representation, using spacy_tokenizer2 (only alpha tokens, lowered all).

[ ]:

4.4 Analysis (10p)

Include an analysis of the results that you obtained in the experiments above. Take advantage of the Jupyter Notebook markdown language, which can also process Latex and HTML, to format your report so that it looks professional.

4.5 Bonus points (∞p)

Anything extra goes here. For example, implement NB from scratch in a separate module nbayes.py and use it for the exercises above.

[ ]: