ArdernClaire_HW6

November 7, 2023

1 Classification Task

In this assignment, you will:

1. Propose a custom classification task and create a corpus of documents that are annotated for this task.
2. Propose discriminative features to be included in the feature vector representation for the examples in this task and evaluate their utility by training and testing Logistic Regression models.

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 *.pdf* showing the code and the output of all cells, and save it in the same folder that contains the notebook file *.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 them after posting them on Canvas.

2.1 Corpus acquisition and formatting

1. Create a corpus of documents that you can use to create at least 300 examples for a task that is of interest you, a task that can be modeled as classification. In class we discussed the following NLP tasks:
    - Document classification.
    - Named entity recognition.
    - Relation extraction.

However you can choose any text processing task as long as it can be solved using classification.

- Manually annotate your data, such that you have at least 300 classification examples.
- For manual annotation, you can use one of the schemes or annotation tools discussed in class.
- The more data in your collection, the better your classification models will tend to perform on it.

2. Partition your data into three datasets: *train*, *dev*, and *test*, with the training set containing 80% of the documents, development 10%, and test 10%.

3. Your choice of task, documents, and labels is completely up to you. For example, if you choose to work on document classification, some possible sources of data are:

    - **Project Gutenberg**: Metadata is available at this Github repo along with URLs for the texts. Labels here can be author, subject, genre, etc.
    - **News articles**: Crawl news articles from different domains (e.g,. CNN, FoxNews); the label for each article is the domain.
    - **Movie summaries**: Labels here can be any categorical metadata aspect (genre, release date); note real-valued metadata (like box office, runtime) can be discretized by selecting some reasonable thresholds.
    - **Tweets**: Download your own tweets. Labels here can be any categorical metadata included in the tweet, or labels you add by hand (e.g., sarcasm).
    - **Book reviews**: Data and metadata is available from the GoodReads dataset and the Amazon reviews dataset.
    - **Restaurant reviews**: Data and metada is available from the Yelp dataset.

4. Additional requirements:

    - No sentiment classification.
    - **Undergraduate students**: It is acceptable to use an existing dataset, e.g. from kaggle.com or other repositories. However, it is preferable that you create your own dataset.
    - **Graduate students**: It is highly recommended that you create your own dataset. This can also serve as the basis for your project, if you choose to submit a project instead of taking the final exam.

2.1.1 Task and Dataset description

Describe your data. What is the source of the documents, and what do the labels mean? How representative is the data, e.g. in terms of demographics, languages, social categories, … What are its limitations / biases?

Data Description The dataset used for this assignemnt was created manually by compiling move plot descriptions from the IMBD website. More specifically, as the IMDB website allows users to sort movies based on genre, three of these genres were chosen to create the three classes used for multiclass classification in this assignment. These three classes (genres) include action, comedy, and romance. These classes were chosen somewhat arbitrarily, though some thought was given to the fact that some genres may be quite similar and hard to differentiate between, such as horror and thriller.

Once the movies were sorted by genre, each individual listed movie's "storyline" (IMDB's section title for plot description), was copied into the corpus. The labels for each document (action, comedy, romance) indicate which genre the plot belongs to.

For example, here is the link to IMDB's comedy movie list: https://www.imdb.com/search/title/?title_type=movie&genres=comedy&genres=Comedy&explore=genres&ref_

How Representative Is the Data? Since the data includes movie plot descriptions, there are not many issues regarding representativeness that should have a significant impact on model performance or fairness. However, it is important to note the lack of variety as far as languages, authors, and social categories. The vast majority of the movies included in the dataset are American movies, with only a few exceptions - not enough foreign movies to ammount to a fair representation. This may have some influence in the way that movies are described, espeically if any plot descriptions are translated from another language to English. In addition, all of the plot descriptions were taken from the IMDB website. This greatly limits the perspectives and writing patterns that are recorded in the writing of the plot descriptions. In addition, this may limit the plot desctiptions to more attention-grabbing, show-business language rather than the average person's day-to-day speech. Lastly, the majority of the movies included in the data are for adult audiences. This is not specifically due to movie content but rather that the majority of the included movies are not "children's movies," though some are.

It may have been helpful to compile a dataset with greater variety in each of these areas. This could be helpful when testing on unseen data that may not have come from the same source, plot descriptions of foreign movies, or movies made for a younger audience.

Data Limitations / Biases The limitations that come with this dataset have been discussed in the previous section. There are no obvious harmful biases that may come with this dataset as it does not include any protected attributes or information that may result in unfair treatment. This is simply a text classification task that involves discriptions of mostly fictional stories.

In the cases where some movies may be describing real-world, international, historical events, it is important to recognize that there may be a bias due to the fact that these are, in the majority of cases, American-made movies. Plot descriptios made include a bias from the American perspective of the historical events. However, since we are simply classifying based on genre, this should not have an impact on our model or its results. For example, an action movie about a historical war is still an action movie regardless of the country's perspective that it is created from.

2.1.2 Dataset reading and statistics

Read the documents in each of the 3 datasets (*training*, *development*, and *test*) and for each dataset display the following statistics:

1. The total number of examples in the dataset.
2. The distribution of labels, which is task dependent. For example:
    - For document classification, this would be the number of documents for each label.
    - For NE recognition, this would be the number of names found for each NE type.
    - For RE, this would be the total number of NE pairs in a sentence that are in the relationship (positive) or not (negative).

For document classification, you can reuse the dataset reading code from the sentiment analysis assignments. For a different classification task, you would have to write different functions to read the data and the annotations. For example, if you use the Brat annotation tool, the annotations are stored in the `.ann` text files, which are straightforward to read.

[1]:
```
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)
            #print([label, text])
            X.append(text)
            Y.append(label)
    return X, Y
```

[2]:
```
import os

datapath = "../data"

# YOUR CODE HERE
train_file = os.path.join(datapath + '/train', 'imdb_plot_train.txt')
trainX, trainY = read_examples(train_file)

dev_file = os.path.join(datapath + '/dev', 'imdb_plot_dev.txt')
devX, devY = read_examples(dev_file)

test_file = os.path.join(datapath  + '/test', 'imdb_plot_test.txt')
testX, testY = read_examples(test_file)
```

[3]:
```
# Statistics for training set:
#     1. total number of samples
#     2. distribution of labels (number of samples in each class)

print('There are %d samples in the training set' % len(trainX))

action_count, comedy_count, romance_count = 0, 0, 0
for x in trainY:
    if x == 'action':
        action_count += 1
    elif x == 'comedy':
        comedy_count += 1
    elif x == 'romance':
        romance_count += 1

print('There are %d documents in the training set with class "action"' %␣
 ↪action_count)
print('There are %d documents in the training set with class "comedy"' %␣
 ↪comedy_count)
print('There are %d documents in the training set with class "romance"' %␣
 ↪romance_count)
```

```
There are 480 samples in the training set
There are 160 documents in the training set with class "action"
There are 160 documents in the training set with class "comedy"
There are 160 documents in the training set with class "romance"
```

[4]:
```
# Statistics for development set:
#     1. total number of samples
#     2. distribution of labels (number of samples in each class)

print('There are %d samples in the development set' % len(devX))

action_count, comedy_count, romance_count = 0, 0, 0
for x in devY:
    if x == 'action':
        action_count += 1
    elif x == 'comedy':
        comedy_count += 1
    elif x == 'romance':
        romance_count += 1

print('There are %d documents in the development set with class "action"' %␣
 ↪action_count)
print('There are %d documents in the development set with class "comedy"' %␣
 ↪comedy_count)
print('There are %d documents in the development set with class "romance"' %␣
 ↪romance_count)
```

```
There are 60 samples in the development set
There are 20 documents in the development set with class "action"
There are 20 documents in the development set with class "comedy"
There are 20 documents in the development set with class "romance"
```

[5]:
```
# Statistics for test set:
#     1. total number of samples
#     2. distribution of labels (number of samples in each class)

print('There are %d samples in the test set' % len(testX))

action_count, comedy_count, romance_count = 0, 0, 0
for x in testY:
    if x == 'action':
        action_count += 1
    elif x == 'comedy':
        comedy_count += 1
    elif x == 'romance':
        romance_count += 1
```

```
print('There are %d documents in the test set with class "action"' %␣
 ↪action_count)
print('There are %d documents in the test set with class "comedy"' %␣
 ↪comedy_count)
print('There are %d documents in the test set with class "romance"' %␣
 ↪romance_count)
```

```
There are 60 samples in the test set
There are 20 documents in the test set with class "action"
There are 20 documents in the test set with class "comedy"
There are 20 documents in the test set with class "romance"
```

2.1.3 From textual examples to feature vectors

Read the documents in each dataset (train, dev, test) and generate the corresponding examples as feature vectors. Some skeleton code is provided below, but feel free to customize as you see fit for your task.

1. Tokenize each document using the spaCy tokenizer or the BPE tiktoken tonekinzer.
2. Create at least two feature functions, and include them in the *features* list.
   - A passing grade will be given to generic features that apply across arbitrary text classification problems (e.g., a feature for bigrams);
   - A better grade will be given for features that reveal your own understanding of your data. What features do you think will help for your particular problem? Would features based on higher level NLP processing tasks (syntactic parsing, coreference resolution) be useful? Your grade is not tied to whether accuracy goes up or down, so be creative!
   - You are free to read in any other external resources you like (dictionaries, document metadata, etc.), but make sure you include them in the `../data` folder.
3. Process each dataset into a set of examples, by mapping each document in the dataset to its corresponding set of examples. Process each example into a feature vector, using the feature functions from *features*. You can reuse feature vector representation code from previous assignments. Features need to be relevant for the task.
   - map example to a dictionary of feature names.
   - map feature names to unique feature IDs.
   - each example is a feature vector, where each feature ID is mapped to a feature value (e.g. word occurences).

[6]:
```
# THE FUNCTIONS IN THIS CELL WERE TAKEN FROM PREVIOUS HOMEWORK ASSIGNMENT SINCE␣
 ↪THEY ARE APPLICABLE FOR THIS TASK
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

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

    return examples

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

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
```

[7]:
```
# CREATING LEXICON FOR ACTION, COMEDY, AND ROMANCE KEYWORDS

# read_lexicon function taken from previous homework
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'

action_file = os.path.join(lexicon_path, 'action_words.txt')
comedy_file = os.path.join(lexicon_path, 'comedy_words.txt')
romance_file = os.path.join(lexicon_path, 'romance_words.txt')

action_lex = read_lexicon(action_file)
comedy_lex = read_lexicon(comedy_file)
romance_lex = read_lexicon(romance_file)

print(len(action_lex), 'entries in the action lexicon.')
print(len(comedy_lex), 'entries in the comedy lexicon.')
print(len(romance_lex), 'entries in the romance lexicon.')
```

```
154 entries in the action lexicon.
74 entries in the comedy lexicon.
67 entries in the romance lexicon.
```

[8]:
```
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]

# YOUR CODE HERE
features = []
```

```
# The following feature function three_lexicon_features creates:
#       A feature 'ACTION_LEX' whose value indicates how many tokens belong to␣
 ↪the action lexicon.
#       A feature 'COMEDY_LEX' whose value indicates how many tokens belong to␣
 ↪the comedy lexicon.
#       A feature 'ROMANCE_LEX' whose value indicates how many tokens belong to␣
 ↪the romance lexicon.
def three_lexicon_features(tokens):
    feats = {'ACTION_LEX': 0, 'COMEDY_LEX': 0, 'ROMANCE_LEX': 0}

    for tok in tokens:
        #print(tok)
        if tok in action_lex:
            feats['ACTION_LEX'] += 1
        if tok in comedy_lex:
            feats['COMEDY_LEX'] += 1
        if tok in romance_lex:
            feats['ROMANCE_LEX'] += 1
    #print(feats)
    return feats

# The following feature function lexicon_features creates:
#       A feature 'ACTION_LEX_word' whose value indicates how many times each␣
 ↪token which belongs to action lex appears.
#       A feature 'COMEDY_LEX_word' whose value indicates how many times each␣
 ↪token which belongs to comedy lex appears.
#       A feature 'ROMANCE_LEX_word' whose value indicates how many times each␣
 ↪token which belongs to romance lex appears.
def lexicon_features(tokens):
    feats = {}

    for tok in tokens:
        if tok in action_lex:
            #print(str('POSLEX_'+tok))
            if str('ACTION_LEX_'+tok) in feats:
                feats[str('ACTION_LEX_'+tok)] += 1
            if str('ACTION_LEX_'+tok) not in feats:
                feats[str('ACTION_LEX_'+tok)] = 1
        if tok in comedy_lex:
            if str('COMEDY_LEX_'+tok) in feats:
                feats[str('COMEDY_LEX_'+tok)] += 1
            if str('COMEDY_LEX_'+tok) not in feats:
                feats[str('COMEDY_LEX_'+tok)] = 1
        if tok in romance_lex:
            if str('ROMANCE_LEX_'+tok) in feats:
```

```
                feats[str('ROMANCE_LEX_'+tok)] += 1
            if str('ROMANCE_LEX_'+tok) not in feats:
                feats[str('ROMANCE_LEX_'+tok)] = 1

    #print(feats)
    return feats

# The following feature function ngram_features creates:
#     A feature for 'NGRAM_word' whose value indicates how many times each␣
 ↪ngram appears.
#     (certain groups of words e.g. phrases may appear often in plot␣
 ↪descriptions of the same genre)
from sklearn.feature_extraction.text import CountVectorizer
def ngram_features(tokens):
    feats = {}

    vectorizer = CountVectorizer(analyzer='word', ngram_range=(2, 6))

    tokens = [' '.join(tokens)]

    X = vectorizer.fit_transform(tokens)
    output_ngrams = vectorizer.get_feature_names_out()

    for ngram in output_ngrams:
        ngram = ngram.replace(" ", "_")

        if str('NGRAM_'+ngram) in feats:
            feats[str('NGRAM_'+ngram)] += 1
        if str('NGRAM_'+ngram) not in feats:
            feats[str('NGRAM_'+ngram)] = 1

    return feats
```

2.1.4 Train and evaluate

Write a `train_and_test` function that takes as input the training and test examples, trains a Logistic Regression model on the training examples and evaluates it on the test examples. You can use the default value for the `C` hyper-parameter, or tune it on the development examples. Report accuracy, or precision and recall, depending on the task.

[11]:
```
from sklearn.metrics import accuracy_score

def train_and_test(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪tokenizer):
    # YOUR CODE HERE

    trainX_feat = docs2features(trainX, feature_functions, tokenizer)
```

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

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    best_acc = [0, 0]
    c = [0.2, 0.5, 0.8, 1.0]

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

        # Pre-process development 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)
        print('Accuracy with c = %1.1f: %.3f' % (i, acc))
        if acc > best_acc[0]:
            best_acc = [acc, i]
            final_model = lr_model

    print("\nThe best accuracy achieved was %.3f with a c value of %1.1f. \n" %␣
↪(best_acc[0], best_acc[1]))

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

    pred_testY = final_model.predict(testX_ids)

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

    return
```

2.1.5 Ablation experiments

Evaluate the impact of your features by training and testing with vs. without each feature.

Using just three lexicon feature function (+ standard word_features func)

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

train_and_test(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9593

Accuracy with c = 0.2: 0.667
Accuracy with c = 0.5: 0.633
Accuracy with c = 0.8: 0.633
Accuracy with c = 1.0: 0.633

The best accuracy achieved was 0.667 with a c value of 0.2.

Accuracy of the model on test data:  0.7833333333333333
```

These are the results from using just the default word_features function with our new three_lexicon_features function, which creates 3 features that represent the individual counts of action, comedy, and romance key-words in each plot description. It is important to note that the best accuracy was achieved with a lower C value. This will be discussed later in the Analysis section. Although the development trials resulted in a best accuracy score of 66.7%, the testing accuracy of 78.3% is not bad. However, it could be better. Lets see if any combination of the other created feature functions can improve upon this accuracy score.

Using just lexicon feature function (+ standard word_features func)

[14]:
```
# Specify features to use.
feature_functions = [word_features, lexicon_features]

train_and_test(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9825

Accuracy with c = 0.2: 0.633
Accuracy with c = 0.5: 0.600
Accuracy with c = 0.8: 0.600
Accuracy with c = 1.0: 0.617

The best accuracy achieved was 0.633 with a c value of 0.2.

Accuracy of the model on test data:  0.75
```

These are the results from using just the default word_features function with our new lexicon_features function, which records each of the action, comedy, and romance key-words that are present in each plot description. It is important to note that the best accuracy was achieved with a lower C value. This will be discussed later in the Analysis section. The development trials resulted in a best accuracy score of 63.3% and the test accuracy for this model is 75%. Neither of

these scores are better than what was achieved in the first experiment. Lets see if any combination of the other created feature functions can improve upon the first accuracy score.

Using just ngram feature function (+ standard word_features func)

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

train_and_test(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
↪spacy_tokenizer)
```

```
Vocabulary size: 224842

Accuracy with c = 0.2: 0.583
Accuracy with c = 0.5: 0.583
Accuracy with c = 0.8: 0.583
Accuracy with c = 1.0: 0.583

The best accuracy achieved was 0.583 with a c value of 0.2.

Accuracy of the model on test data:  0.7
```

These are the results from using just the default word_features function with our new ngram_features function, which records each of the word-pairs or longer phrases that are present in each plot description. It is possible that similar phrases or word-pairs may be used in the same genre classes. The development trials resulted in a best accuracy score of 58.3% and the test accuracy for this model is 70%. Neither of these scores are better than what was achieved in the first experiment. Lets see if any combination of the other created feature functions can improve upon the first accuracy score.

Using both three lexicon and lexicon feature functions (+ standard word_features func)

[16]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, lexicon_features]

train_and_test(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
↪spacy_tokenizer)
```

```
Vocabulary size: 9828

Accuracy with c = 0.2: 0.667
Accuracy with c = 0.5: 0.633
Accuracy with c = 0.8: 0.633
Accuracy with c = 1.0: 0.633

The best accuracy achieved was 0.667 with a c value of 0.2.

Accuracy of the model on test data:  0.7666666666666667
```

These are the results from using just the default word_features function with our new three_lexicon_features and lexicon_features functions, which have both been described previously. It is important to note that the best accuracy was achieved with a lower C value. This will be discussed later in the Analysis section. The development trials resulted in a best accuracy score of 66.7% and the test accuracy for this model is 76.67%. Neither of these scores are better than what was achieved in the first experiment. Lets see if any combination of the other created feature functions can improve upon the first accuracy score.

Using both three lexicon and ngram feature functions (+ standard word_features func)

[17]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, ngram_features]

train_and_test(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 224845

Accuracy with c = 0.2: 0.650
Accuracy with c = 0.5: 0.650
Accuracy with c = 0.8: 0.650
Accuracy with c = 1.0: 0.650

The best accuracy achieved was 0.650 with a c value of 0.2.

Accuracy of the model on test data:  0.7666666666666667
```

These are the results from using just the default word_features function with our new three_lexicon_features and ngram_features functions, which have both been described previously. The development trials resulted in a best accuracy score of 65% and the test accuracy for this model is 76.67%. Neither of these scores are better than what was achieved in the first experiment. Lets see if any combination of the other created feature functions can improve upon the first accuracy score.

Using both lexicon and ngram feature function (+ standard word_features func)

[18]:
```
# Specify features to use.
feature_functions = [word_features, lexicon_features, ngram_features]

train_and_test(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 225077

Accuracy with c = 0.2: 0.600
Accuracy with c = 0.5: 0.600
Accuracy with c = 0.8: 0.600
Accuracy with c = 1.0: 0.600
```

```
The best accuracy achieved was 0.600 with a c value of 0.2.

Accuracy of the model on test data:  0.7166666666666667
```

These are the results from using just the default word_features function with our new lexicon_features and ngram_features functions, which have both been described previously. The development trials resulted in a best accuracy score of 60% and the test accuracy for this model is 71.67%. Neither of these scores are better than what was achieved in the first experiment. Lets see the combination of all of the created feature functions can improve upon the first accuracy score.

Using all 3 feature functions (+ standard word_features func)

[19]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, lexicon_features,␣
 ↪ngram_features]

train_and_test(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 225080

Accuracy with c = 0.2: 0.650
Accuracy with c = 0.5: 0.650
Accuracy with c = 0.8: 0.650
Accuracy with c = 1.0: 0.650

The best accuracy achieved was 0.650 with a c value of 0.2.

Accuracy of the model on test data:  0.7666666666666667
```

These are the results from using just the default word_features function with our new three_lexicon_features, lexicon_features, and ngram_features functions, which have all been described previously. The development trials resulted in a best accuracy score of 65% and the test accuracy for this model is 76.67%. Neither of these scores are better than what was achieved in the first experiment. This means that the first experiment achieved the best possible results among all the different combinations of the created feature functions.

2.2 Plot ROC or PR Curve

Mandatory for graduate students, optional for undergraduate students.

Take your best classifier and plot a Receiver Operating Characteristic (ROC) curve if you use accuracy for evaluation, by varying a threshold on the probabilistic output. You can use the sklearn implementation, or implement your own. If you use precision and recall, plot a precision vs. recall (PR) curve instead.

[24]:
```
# YOUR CODE HERE
# RERUN THE BEST MODEL TO GET IT BACK IN CURRENT MEMORY. REMOVING THE CONTENTS␣
 ↪FROM THE TRAIN_AND_TEST FUNCTION TO
```

```
# MAKE IT EASIER TO ACCESS THE CONTENTS AND COMPUTE ROC CURVE

feature_functions = [word_features, three_lexicon_features]

trainX_feat = docs2features(trainX, feature_functions, spacy_tokenizer)

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

trainX_ids = features_to_ids(trainX_feat, feature_vocab)

best_acc = [0, 0]
c = [0.2, 0.5, 0.8, 1.0]

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

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

    # Test LR model.
    acc = lr_model.score(devX_ids, devY)
    print('Accuracy with c = %1.1f: %.3f' % (i, acc))
    if acc > best_acc[0]:
        best_acc = [acc, i]
        final_model = lr_model

print("\nThe best accuracy achieved was %.3f with a c value of %1.1f. \n" %␣
 ↪(best_acc[0], best_acc[1]))

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

pred_testY = final_model.predict(testX_ids)

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

```
Vocabulary size: 9593
```

```
Accuracy with c = 0.2: 0.667
Accuracy with c = 0.5: 0.633
Accuracy with c = 0.8: 0.633
Accuracy with c = 1.0: 0.633

The best accuracy achieved was 0.667 with a c value of 0.2.

Accuracy of the model on test data:  0.7833333333333333
```

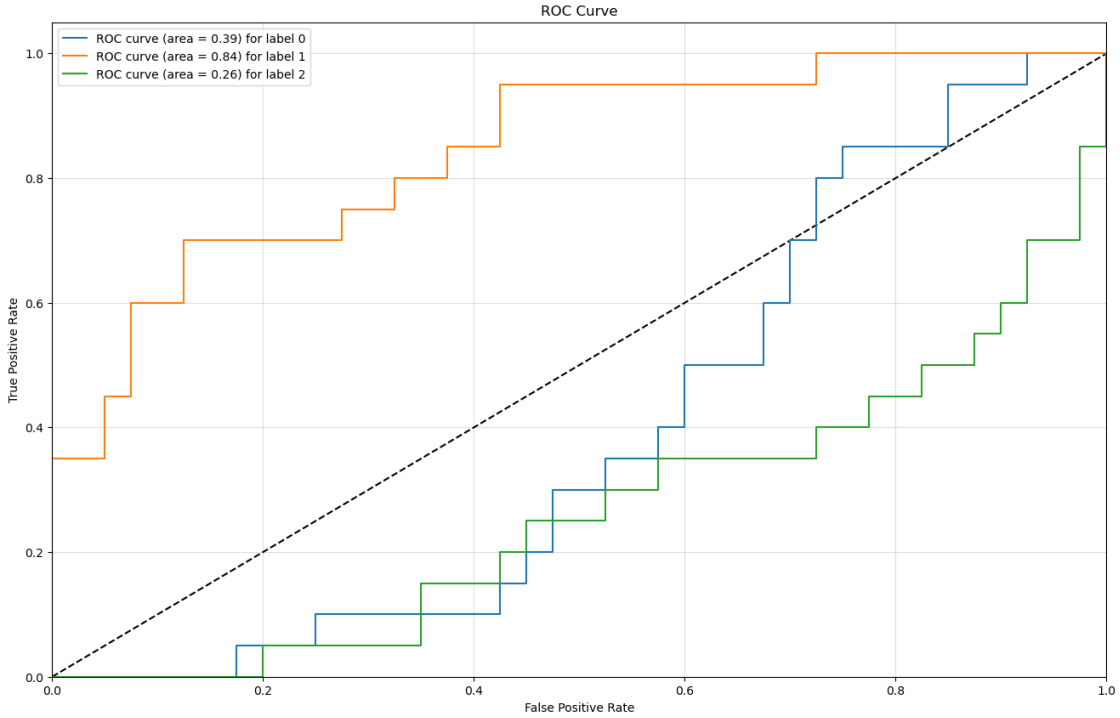
[25]:
```
testX_feat = docs2features(testX, feature_functions, spacy_tokenizer)
testX_ids = features_to_ids(testX_feat, feature_vocab)

y_prob = final_model.predict_proba(testX_ids)
```

[26]:
```
classes = final_model.classes_
print(classes)
```

```
['action' 'comedy' 'romance']
```

[27]:
```
import pandas as pd
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

n_classes = len(classes)

fpr, tpr, roc_auc = {}, {}, {}

# Compute fpr and tpr for each class, compute roc auc for each class
y_test = pd.get_dummies(testY, drop_first=False).values
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_prob[:,1])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot ROC for each class
fig, ax = plt.subplots(figsize=(16, 10))
ax.plot([0, 1], [0, 1], 'k--')
ax.set_xlim([0.0, 1.0])
ax.set_ylim([0.0, 1.05])
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.set_title('ROC Curve')

for i in range(n_classes):
    ax.plot(fpr[i], tpr[i], label='ROC curve (area = %0.2f) for label %i' %␣
 ↪(roc_auc[i], i))

ax.legend(loc="best")
```

```
ax.grid(alpha=.4)
plt.show()
```



This ROC curve shows us how well the best performing model is classifying each class. With the 'action' class represented as 0, the 'comedy' class represented as 1, and the 'romance' class represented as 2, we can see that the model is best at classifying the comedy class. We will discuss the meaning of these results in the analysis section.

2.3 Bonus points

Anything extra goes here, e.g. evaluate other ML models, evaluate an LLM such as GPT using the Chat completion API, …

2.3.1 Decision Tree Classifier

To compare against the performance of the Logistic Regression model, we can feed the same data to a Decision Tree model. It is hard to tell whether any specific model will work best for our data before testing it. There is one best way to find out - try it! So, lets see if we can achieve better results with this model.

We can easily modify our train_and_test function to use the Decision Tree model instead of the Logistic Regression model. Then, we can perform all of the same experiments to see if any combination of our feature functions achieves a better accuracy score than what was achieved by the best Logistic Regression model.

[28]:
```
# Evaluate performance using Decision Tree Classifier from sklearn
from sklearn.tree import DecisionTreeClassifier

def train_and_test_DT(trainX, trainY, devX, devY, testX, testY,␣
 ↪feature_functions, tokenizer):

    trainX_feat = docs2features(trainX, feature_functions, spacy_tokenizer)

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

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    best_acc = [0, 0]
    d = [2, 4, 6, 8, 10]

    for i in d:
        # Train DT model.
        dtree_model = DecisionTreeClassifier(max_depth = i)
        dtree_model.fit(trainX_ids, trainY)

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

        # Test DT model.
        acc = dtree_model.score(devX_ids, devY)
        print('Accuracy with max depth = %2f: %.3f' % (i, acc))
        if acc > best_acc[0]:
            best_acc = [acc, i]
            final_model = dtree_model

    print("\nThe best accuracy achieved was %.3f with a max depth of %2f. \n" %␣
 ↪(best_acc[0], best_acc[1]))

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

    pred_testY = final_model.predict(testX_ids)

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

    return
```

[29]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features]

train_and_test_DT(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9593

Accuracy with max depth = 2.000000: 0.550
Accuracy with max depth = 4.000000: 0.533
Accuracy with max depth = 6.000000: 0.533
Accuracy with max depth = 8.000000: 0.500
Accuracy with max depth = 10.000000: 0.500

The best accuracy achieved was 0.550 with a max depth of 2.000000.

Accuracy of the model on test data:  0.6333333333333333
```

[30]:
```
# Specify features to use.
feature_functions = [word_features, lexicon_features]

train_and_test_DT(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9825

Accuracy with max depth = 2.000000: 0.467
Accuracy with max depth = 4.000000: 0.517
Accuracy with max depth = 6.000000: 0.450
Accuracy with max depth = 8.000000: 0.450
Accuracy with max depth = 10.000000: 0.450

The best accuracy achieved was 0.517 with a max depth of 4.000000.

Accuracy of the model on test data:  0.5333333333333333
```

[31]:
```
# Specify features to use.
feature_functions = [word_features, ngram_features]

train_and_test_DT(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 224842

Accuracy with max depth = 2.000000: 0.467
Accuracy with max depth = 4.000000: 0.517
Accuracy with max depth = 6.000000: 0.450
Accuracy with max depth = 8.000000: 0.450
```

```
Accuracy with max depth = 10.000000: 0.433

The best accuracy achieved was 0.517 with a max depth of 4.000000.

Accuracy of the model on test data:  0.5333333333333333
```

[32]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, lexicon_features]

train_and_test_DT(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9828

Accuracy with max depth = 2.000000: 0.550
Accuracy with max depth = 4.000000: 0.550
Accuracy with max depth = 6.000000: 0.517
Accuracy with max depth = 8.000000: 0.500
Accuracy with max depth = 10.000000: 0.467

The best accuracy achieved was 0.550 with a max depth of 2.000000.

Accuracy of the model on test data:  0.6333333333333333
```

[33]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, ngram_features]

train_and_test_DT(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 224845

Accuracy with max depth = 2.000000: 0.550
Accuracy with max depth = 4.000000: 0.550
Accuracy with max depth = 6.000000: 0.550
Accuracy with max depth = 8.000000: 0.567
Accuracy with max depth = 10.000000: 0.550

The best accuracy achieved was 0.567 with a max depth of 8.000000.

Accuracy of the model on test data:  0.6333333333333333
```

[34]:
```
# Specify features to use.
feature_functions = [word_features, lexicon_features, ngram_features]

train_and_test_DT(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 225077

Accuracy with max depth = 2.000000: 0.467
Accuracy with max depth = 4.000000: 0.517
Accuracy with max depth = 6.000000: 0.450
Accuracy with max depth = 8.000000: 0.450
Accuracy with max depth = 10.000000: 0.433

The best accuracy achieved was 0.517 with a max depth of 4.000000.

Accuracy of the model on test data:  0.5333333333333333
```

[35]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, lexicon_features,␣
 ↪ngram_features]

train_and_test_DT(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 225080

Accuracy with max depth = 2.000000: 0.550
Accuracy with max depth = 4.000000: 0.533
Accuracy with max depth = 6.000000: 0.567
Accuracy with max depth = 8.000000: 0.567
Accuracy with max depth = 10.000000: 0.533

The best accuracy achieved was 0.567 with a max depth of 6.000000.

Accuracy of the model on test data:  0.6333333333333333
```

As seen in all of the above Decision Tree experiments, this model was not able to achieve a better accuracy score than our best Logistic Regression model, regardless of the combination of feature functions. However, we can still try other models…

2.3.2 Support Vector Machine (SVM) Classifier

To compare against the performance of the Logistic Regression and Decision Tree models, we can feed the same data to a SVM model.

We can easily modify our train_and_test function to use the SVM model instead of the Logistic Regression model. Then, we can perform all of the same experiments to see if any combination of our feature functions achieves a better accuracy score than what was achieved by the best Logistic Regression model.

[36]:
```
# Evaluate performance using SVM Classifier from sklearn
from sklearn.svm import SVC
```

```
def train_and_test_SVM(trainX, trainY, devX, devY, testX, testY,␣
↪feature_functions, tokenizer):

    trainX_feat = docs2features(trainX, feature_functions, tokenizer)

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

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    best_acc = [0, 0]
    c = [0.2, 0.5, 0.8, 1.0]

    for i in c:
        # Train SVM model.
        svm_model_linear = SVC(kernel = 'linear', C = 1)
        svm_model_linear.fit(trainX_ids, trainY)

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

        # Test SVM model.
        acc = svm_model_linear.score(devX_ids, devY)
        print('Accuracy with c = %1.1f: %.3f' % (i, acc))
        if acc > best_acc[0]:
            best_acc = [acc, i]
            final_model = svm_model_linear

    print("\nThe best accuracy achieved was %.3f with a c value of %1.1f. \n" %␣
↪(best_acc[0], best_acc[1]))

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

    pred_testY = final_model.predict(testX_ids)

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

    return
```

[37]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features]
```

```
train_and_test_SVM(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9593

Accuracy with c = 0.2: 0.633
Accuracy with c = 0.5: 0.633
Accuracy with c = 0.8: 0.633
Accuracy with c = 1.0: 0.633

The best accuracy achieved was 0.633 with a c value of 0.2.

Accuracy of the model on test data:  0.8166666666666667
```

[38]:
```
# Specify features to use.
feature_functions = [word_features, lexicon_features]

train_and_test_SVM(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9825

Accuracy with c = 0.2: 0.600
Accuracy with c = 0.5: 0.600
Accuracy with c = 0.8: 0.600
Accuracy with c = 1.0: 0.600

The best accuracy achieved was 0.600 with a c value of 0.2.

Accuracy of the model on test data:  0.7
```

[39]:
```
# Specify features to use.
feature_functions = [word_features, ngram_features]

train_and_test_SVM(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 224842

Accuracy with c = 0.2: 0.567
Accuracy with c = 0.5: 0.567
Accuracy with c = 0.8: 0.567
Accuracy with c = 1.0: 0.567

The best accuracy achieved was 0.567 with a c value of 0.2.

Accuracy of the model on test data:  0.7166666666666667
```

[40]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, lexicon_features]

train_and_test_SVM(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9828

Accuracy with c = 0.2: 0.617
Accuracy with c = 0.5: 0.617
Accuracy with c = 0.8: 0.617
Accuracy with c = 1.0: 0.617

The best accuracy achieved was 0.617 with a c value of 0.2.

Accuracy of the model on test data:  0.7666666666666667
```

[41]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, ngram_features]

train_and_test_SVM(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 224845

Accuracy with c = 0.2: 0.650
Accuracy with c = 0.5: 0.650
Accuracy with c = 0.8: 0.650
Accuracy with c = 1.0: 0.650

The best accuracy achieved was 0.650 with a c value of 0.2.

Accuracy of the model on test data:  0.8
```

[42]:
```
# Specify features to use.
feature_functions = [word_features, lexicon_features, ngram_features]

train_and_test_SVM(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 225077

Accuracy with c = 0.2: 0.583
Accuracy with c = 0.5: 0.583
Accuracy with c = 0.8: 0.583
Accuracy with c = 1.0: 0.583

The best accuracy achieved was 0.583 with a c value of 0.2.
```

```
Accuracy of the model on test data:  0.7166666666666667
```

[43]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, lexicon_features,␣
 ↪ngram_features]

train_and_test_SVM(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 225080

Accuracy with c = 0.2: 0.617
Accuracy with c = 0.5: 0.617
Accuracy with c = 0.8: 0.617
Accuracy with c = 1.0: 0.617

The best accuracy achieved was 0.617 with a c value of 0.2.

Accuracy of the model on test data:  0.8
```

As seen in the above SVM experiments, this model was able to achieve a better accuracy score than our best Logistic Regression model, depending the combination of feature functions. The best test accuracy score achieved with a Logistic Regression model was 78.3%. When using the SVM model, the use of our three_lexicon_features function achieved an accuracy of 81.67%. In addition, when using the SVM model, the use of our three_lexicon_features and ngram_features functions achieved an accuracy of 80%. The use of all three of our feature functions (three_lexicon_features, lexicon_features, and ngram_features) also resulted in an accuracy of 80%. All three of these SVM model scores are better than what was achieved with the Logistic Regression model! This is an interesting observation. So far, it seems that the SVM model is the best model for this classificaiton task. However, we can still try another model...

2.3.3 K-Nearest Neighbors (KNN) Classifier

To compare against the performance of the Logistic Regression, Decision Tree, and SVM models, we can feed the same data to a KNN model.

We can easily modify our train_and_test function to use the KNN model instead of the Logistic Regression model. Then, we can perform all of the same experiments to see if any combination of our feature functions achieves a better accuracy score than what was achieved by the best Logistic Regression or SVM models.

[44]:
```
# Evaluate performance using KNN Classifier from sklearn
from sklearn.neighbors import KNeighborsClassifier

def train_and_test_KNN(trainX, trainY, devX, devY, testX, testY,␣
 ↪feature_functions, tokenizer):

    trainX_feat = docs2features(trainX, feature_functions, spacy_tokenizer)
```

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

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    best_acc = [0, 0]
    k = [2, 4, 6, 8, 10]

    for i in k:
        # Train KNN model.
        knn_model = KNeighborsClassifier(n_neighbors = i)
        knn_model.fit(trainX_ids, trainY)

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

        # Test KNN model.
        acc = knn_model.score(devX_ids, devY)
        print('Accuracy with K neighbors = %2f: %.3f' % (i, acc))
        if acc > best_acc[0]:
            best_acc = [acc, i]
            final_model = knn_model

    print("\nThe best accuracy achieved was %.3f with K neighbors = %2f. \n" %␣
↪(best_acc[0], best_acc[1]))

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

    pred_testY = final_model.predict(testX_ids)

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

    return
```

[45]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features]

train_and_test_KNN(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
↪spacy_tokenizer)
```

```
Vocabulary size: 9593

Accuracy with K neighbors = 2.000000: 0.433
Accuracy with K neighbors = 4.000000: 0.533
Accuracy with K neighbors = 6.000000: 0.517
Accuracy with K neighbors = 8.000000: 0.567
Accuracy with K neighbors = 10.000000: 0.583

The best accuracy achieved was 0.583 with K neighbors = 10.000000.

Accuracy of the model on test data:  0.6166666666666667
```

[46]:
```
# Specify features to use.
feature_functions = [word_features, lexicon_features]

train_and_test_KNN(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9825

Accuracy with K neighbors = 2.000000: 0.367
Accuracy with K neighbors = 4.000000: 0.400
Accuracy with K neighbors = 6.000000: 0.533
Accuracy with K neighbors = 8.000000: 0.433
Accuracy with K neighbors = 10.000000: 0.500

The best accuracy achieved was 0.533 with K neighbors = 6.000000.

Accuracy of the model on test data:  0.38333333333333336
```

[47]:
```
# Specify features to use.
feature_functions = [word_features, ngram_features]

train_and_test_KNN(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 224842

Accuracy with K neighbors = 2.000000: 0.317
Accuracy with K neighbors = 4.000000: 0.317
Accuracy with K neighbors = 6.000000: 0.333
Accuracy with K neighbors = 8.000000: 0.333
Accuracy with K neighbors = 10.000000: 0.317

The best accuracy achieved was 0.333 with K neighbors = 6.000000.

Accuracy of the model on test data:  0.3333333333333333
```

[48]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, lexicon_features]

train_and_test_KNN(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 9828

Accuracy with K neighbors = 2.000000: 0.450
Accuracy with K neighbors = 4.000000: 0.533
Accuracy with K neighbors = 6.000000: 0.550
Accuracy with K neighbors = 8.000000: 0.567
Accuracy with K neighbors = 10.000000: 0.550

The best accuracy achieved was 0.567 with K neighbors = 8.000000.

Accuracy of the model on test data:  0.55
```

[49]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, ngram_features]

train_and_test_KNN(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 224845

Accuracy with K neighbors = 2.000000: 0.350
Accuracy with K neighbors = 4.000000: 0.367
Accuracy with K neighbors = 6.000000: 0.383
Accuracy with K neighbors = 8.000000: 0.367
Accuracy with K neighbors = 10.000000: 0.367

The best accuracy achieved was 0.383 with K neighbors = 6.000000.

Accuracy of the model on test data:  0.36666666666666664
```

[50]:
```
# Specify features to use.
feature_functions = [word_features, lexicon_features, ngram_features]

train_and_test_KNN(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 225077

Accuracy with K neighbors = 2.000000: 0.300
Accuracy with K neighbors = 4.000000: 0.317
Accuracy with K neighbors = 6.000000: 0.333
Accuracy with K neighbors = 8.000000: 0.333
```

```
Accuracy with K neighbors = 10.000000: 0.300

The best accuracy achieved was 0.333 with K neighbors = 6.000000.

Accuracy of the model on test data:  0.3333333333333333
```

[51]:
```
# Specify features to use.
feature_functions = [word_features, three_lexicon_features, lexicon_features,␣
 ↪ngram_features]

train_and_test_KNN(trainX, trainY, devX, devY, testX, testY, feature_functions,␣
 ↪spacy_tokenizer)
```

```
Vocabulary size: 225080

Accuracy with K neighbors = 2.000000: 0.350
Accuracy with K neighbors = 4.000000: 0.350
Accuracy with K neighbors = 6.000000: 0.383
Accuracy with K neighbors = 8.000000: 0.383
Accuracy with K neighbors = 10.000000: 0.350

The best accuracy achieved was 0.383 with K neighbors = 6.000000.

Accuracy of the model on test data:  0.36666666666666664
```

As seen in all of the above KNN experiments, this model was not able to achieve a better accuracy score than our best Logistic Regression or SVM models, regardless of the combination of feature functions. In fact, in all of the experiments, the KNN model performed significantly worse than any of the previous models. It is clear that the KNN model is not a good model to use for this classification task.

2.4 Analysis

Very important: Include an analysis of the data and the results that you obtained in the experiments above. It is important that results are formatted well, e.g. using tables, lists, etc.

2.4.1 Results of Logistic Regression Model

The best performing Logistic Regression model was identified by performing several experiments that included different combinations of the created feature functions. This best performing model used the baseline word_features function that was used in every experiment in combination with the new three_lexicon_features function. This three_lexicon_features function creates 3 features that record the individual counts of action, comedy, and romance keywords from the manually created action, comedy, and romance lexicons.

The results of this model included a best development accuracy score of 66.7% which was achieved when using a C value of 0.2. This C value determines how much the model relies on (or "trusts") the training data when making predictions on new data. A lower value, like 0.2, indicates that the model should not rely heavily on the training data. It makes sense that a lower C value would

result in better performance for this case as each individual movie plot description will likely be very different from the next. It is hard to generalize across these documents as each movie plot will be different and there are infinite ways a plot can be described, even within the same genre. So, as the new development and testing documents that the model sees will likely be very different from the documents seen in training, the model should not rely too heavily on what it has seen in the training dataset.

This model achieved a testing accuracy of 78.33% which is not a great score, but is not bad considering the small amount of training data. The ROC curve shown above shows that the comedy class was by far the 'easiest' class for the model to identify. The other two curves shown in the graph, representing the action and romance classes, do not show a good performance. This means that in most cases, the model correctly identified the documents in the comedy class, but often made mistakes with documents in the action and romance classes. There are many possible reasons for these mistakes but one possible explanation could be the fact that the romance lexicon had the fewest amount of keywords, giving the model less of a chance to pick out identifying words in a document that would indicate the romance class. On the other hand, it could be that the comedy keywords simply appeared more often among the comedy documents than action and romance keywords did in their respective documents.

It is important to note that there is an even distribution of examples for each class among each of the datasets. The train, development, and test datasets each have exactly the same amount of documents in each of the three classes (160 examples per class in train, 20 examples per class in dev and test). This means that there should be no bias due to an unequal representation of classes.

Error Analysis There are several reasons why a model may be inaccurately classifying documents in a text classification problem. In order to identify where and why a model is making mistakes, we must compare the model's predictions to the ground truth. By doing so, we can gain a better understanidng of the flaws in our model.

[52]:
```
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
```

```
Test case  8  with class  action  was incorrectly classified as  comedy
Test case  10  with class  action  was incorrectly classified as  comedy
Test case  17  with class  action  was incorrectly classified as  romance
Test case  20  with class  comedy  was incorrectly classified as  action
Test case  21  with class  comedy  was incorrectly classified as  action
Test case  23  with class  comedy  was incorrectly classified as  romance
Test case  25  with class  comedy  was incorrectly classified as  action
Test case  28  with class  comedy  was incorrectly classified as  romance
Test case  29  with class  comedy  was incorrectly classified as  romance
Test case  33  with class  comedy  was incorrectly classified as  romance
```

Lets take a closer look into some of these example cases where the model has made incorrect predictions. By doing so, we may be able to determine why the model has made an incorrect

prediction and, in the future, improve our model to avoid these mistakes.

[55]:
```
print("Data:", testX[17], '\n')
print("Correct Class: ", testY[17], '\n')
print("Predicted Class: ", pred_testY[17], '\n')
```

```
Data: Dracula is alive. In fact, he plans to rule the world and that is why he
seeks the help of other legendary monsters. However, a bunch of kids regarded by
their peers as losers uncover the devious plan and prepare for a counter strike.

Correct Class:  action

Predicted Class:  romance
```

In this case, it is already clear that there is not much material for the model to work with. This plot description is very short, which means there is not much content that could help indicate which class the document belongs too. We can also see that there are no action keywords in this document that would help indicate the correct class. It is likely that, with such little information to work with and such a low C value forcing the model to rely less on the training data, this prediction is more of a "guess" than an informed prediction. In the future, it would likely be helpful to design more feature functions (that are actually helpful, unlike the other created functions) that would give the model more information to support an informed decision. Even something as simple as document length could turn out to be helpful. Maybe action movies tend to have shorter plot descriptions?

[56]:
```
print("Data:", testX[20], '\n')
print("Correct Class: ", testY[20], '\n')
print("Predicted Class: ", pred_testY[20], '\n')
```

```
Data: As a boy, Carl Fredricksen wanted to explore South America and find the
forbidden Paradise Falls. About 64 years later he gets to begin his journey
along with Boy Scout Russell by lifting his house with thousands of balloons. On
their journey, they make many new friends including a talking dog, and figure
out that someone has evil plans. Carl soon realizes that this evildoer is his
childhood idol.

Correct Class:  comedy

Predicted Class:  action
```

In this case, it is more obvious why the model has made its incorrect prediction. The words "evil" and "journey" are both action keywords and both appear more than once in this document. Given these features, it makes sense that the model would have predicted that this document is of the action class. This brings up the complicating factor that comedy movies often have the same general plots of movies from other specific genres, such as action or horror, but are created using sarcasm and humerous dialogues or extremes. In any case, that fact would make it difficult to identify a comedy movie simply from a description of the plot. Perhaps, in the future, we could create some sentiment analysis feature that could determine whether there are some comedic indications hidden

within the plot description.

Impact of Created Feature Functions The three_lexicon_features was the only feature function created that seemed to have a positive impact on model performance. This feature function created 3 features that kept a count of the number of action, comedy, and romance keywords, respectively, in each of the documents. It makes sense for this to have a positive impact on the model performance as the number of keywords from any one genre (class) that appear in the document could be a great indication of which class the document belongs to. For example, it would make sense to see many action keywords in an action movie plot description. The results achieved by the above experiments supports this idea, as the best performing model was the model that used only this feature function.

The lexicon_features function did not seem to have a positive impact on model performance. This feature function created a feature for each of the action, comedy, and romance keywords that appeared in the document along with a count for the number of times each keyword appeared. It was theorized that these features could improve the performance of the model as heavier weights could be learned for keywords that have a stronger indication for any of the classes (such as 'journey' vs 'kill' for action class keywords - it could be imagined that 'kill' would have more weight). However, the resulting performance in experiments which used this feature function was not improved, which in turn disproved this theory.

The ngram_features function did not seem to have a positive impact on model performance. This feature function created a feature for each identified n-gram (with n ranging from 2 to 6, meaning phrases of 2-6 words) within a document as well as the number of times each n-gram appeared. The theory behind this feature function was that certain groups of words (e.g. phrases) may appear often in plot descriptions of the same genre. For example, the phrase "seeking revenge" may appear often in action movie plot descriptions. However, the resulting performance in experiments which used this feature function was not improved, which in turn disproved this theory.

2.4.2 Results of Bonus Models

The table below shows a summary of the best results achieved by each of the models that were tested in the Bonus section. This table includes the model name, the best development accuracy achieved by that model, the parameters that were used to achieve the best development accuracy, the test accuracy, and the features used to achieve these results.

Model	Best Dev Accuracy	Best Parameter Selection	Test Accuracy	Feature Functions
Decision Tree	56.7%	Max Depth = 8	63.33%	word_features, three_lexicon_features, ngram_features
SVM	63.3%	C = 0.2	81.67%	word_features, three_lexicon_features
KNN	58.3%	K = 10	61.67%	word_features, three_lexicon_features

As seen in the table, the SVM model achieved the best development and test accuracy of all of the Bonus models that were used. In addition, the score for the SVM model was an improvement on

the best accuracy score achieved with the required Logistic Regression model.

Although the SVM model is quite simple, it clearly achieved the best results of all of the models. This model works by finding a hyperplane in an N-dimentional space (with N representing the number of features) that can distinclty classify datapoints. In other words, its a margin (or a type of threshold) that defines which feature values in a document will indicate any of the classes. Since we are primarily using a count of class keywords to determine the class, it makes sense that this model would perform well for this task. To offer a simplified explanation, the model is essentially thinking "if I see a keywords of class 0 in a document and b keywords of class 1 in the same document, as long as a and b are within these ranges (with 'these ranges' specified by the hyperplane), I can assume the document is of class 0."

The Decision Tree model was the next best performing model of all the Bonus models. This model did not perform as well as the SVM model nor the Logistic Regression model. The decision tree works by starting with one conditon and branching from there based on the results of this condition. From there, each branch will lead to another conidition which will in turn create more branches. To provide a simple example for this task, the decision tree may start with the condition "does this document contain action keywords?" The answer to that condition may be yes or no, and from those two branches, another condition may appear such as "does this document contain comedy keywords?" These branches will continue up until the specified max-depth of the model and will use the provided features. Since there may be keywords of multiple genres in any one document, and the other included features are not necessarily able to be used in a simple conditional case, it is not surprising that this model would not lead to the best conclusions (predictions).

The KNN model was the worst performing model of all the Bonus models. This model did not perform as well as the SVM model, Decision Tree model, nor the Logistic Regression model. This model works on the base assumption that similar things exist in close proximity. In other words, in an N-dimensional space, documents of the same class would appear in the same area. It makes sense for this model to not perform as well for this task as there is such a wide variety among the documents in the corpus. As stated earlier, it is hard to generalize across plot descriptions as there are likely infinite possible plots and infinite ways to describe said plots, even within the same genre.