Project: Part 1

CS 421: Natural Language Processing

Due: March 5, 2025 (11:59 p.m. CST)

1. Introduction

Welcome to part one of your CS 421 course project! In this project, you will work on two popular tasks in the NLP domain namely named entity recognition (NER) and text classification. The project is divided into two deliverables, each of which will leverage the skills you have learned in class. These deliverables will sequentially build on each other, culminating into a comprehensive NLP system by the end of the semester.

In part one of this project, you will build upon knowledge that you have gained in class regarding **semantics, syntactic parsing** and **text classification**. You will develop a part-of-speech tagger, experiment with existing parsers, embeddings, and you will also compare a variety of statistical machine learning models that are commonly used for text classification. Feel free to brush up on all of these topics before diving in by reviewing the instructional materials on Blackboard. You will work with a small dataset from the CMU movie summary corpus, classifying movie genres (labels) (Thriller (0), Drama (1), Crime Fiction (2), Romantic comedy (3)) based on their summaries. You will also create a part-of-speech tagger using Hidden Markov models. Lastly, you will experiment with the Stanford NER tagger for named entity recognition.

2. Instructions

Each part of this deliverable is labeled as Code or Written. The guidelines for these two types of components are provided below.

Code

The Code questions need to be completed using Python (version 3.10+). This portion of the assignment requires use of the following Python packages:

- numpy
- pandas
- scikit-learn
- random
- nltk

You can install each of these using PyCharm or the pip `install` <package> tool if they do not already exist on your system. If you want to use any other external package(s) for any reason, you are required to get approval from the course staff

on Piazza prior to submission. Templates are provided for each Code question (py file) as part of the supplementary material. Do not rename/delete any functions or global variables provided in these templates and write your solution in the specified sections. Use the main function (provided in templates) to test your code when running it from a terminal. Avoid writing that test code in the global scope; however, you should write additional functions or classes as needed in the global scope.
These templates may also contain important information and/or examples in the comments so please read them carefully. Code parts of your deliverables will be graded automatically or manually using Gradescope; further specifications are provided in the question text.

To submit your solution for Code questions, you need to compress the following files (after completion) into a single zip file. These files should be in the root of your zip archive for autograding to work correctly.

- □ classification.py
- □ csv files with model predictions on the test dataset for genre classification
- □ parser.py
- □ ner.py

Submit this zip file on Gradescope under Project Part 1 - Code. All specified files need to be submitted to receive full credit.

Written

You are required to submit all Written questions in a single PDF file. You may create this PDF using Microsoft Word, scans of your handwritten solution, LATEX or any other method or design tool you prefer.
To submit your solution for Written questions, you need to provide answers to the following questions in a single PDF.

- □ Q9

Before submission, ensure that all pages of your solution are present and in order. Submit this PDF on Gradescope under Project Part 1 - Written. Please match all questions to their respective solutions (pages) on Gradescope. Questions not associated with any pages will be considered blank or missing and all questions need to be completed to receive full credit.

3. Code

Q1 (5): Text preprocessing

Text processing improves the consistency and accuracy of natural language processing by standardizing raw data. Clean and well-structured text is crucial for effective analysis.

1. Implement *get_sentences()* using NLTK's sent_tokenize to split the text into sentences, accommodating tasks that analyze text at the sentence level.
2. Implement *get_tokens()* using NLTK's word_tokenize to break sentences into words, essential for detailed linguistic analysis.
3. Implement *clean_text()* to remove stopwords and convert text to lowercase using

NLTK. This function prepares the text by focusing on meaningful content. Output for this function will be a string.

Q2 (10): Compute TF-IDF vectors for the dataset.

1. Implement function *train_vectorizer()* to compute TF-IDF weights from your training corpus.
2. Read the dataset using the *load_the_dataset()* function in the skeleton code. This function reads the file and returns three separate lists: summaries, labels (genres), and ids.

3. Utilize TfidfVectorizer from sklearn.feature_extraction to understand and compute TF-IDF weights for the movie summaries.

   Note: Not only does it compute TF-IDF vectors for your string, but it can also preprocess the text. If you're going to use it with preprocessing, ensure that you specify the correct parameters for TF-IDF during the initialization process such that:

   3.1. It tokenizes the string using the provided *get_tokens()* function implemented before.
   3.2. Convert the tokens to lowercase (e.g., I love NLP → i love nlp).

4. Instantiate the vectorizer as shown in the skeleton code and then call the vectorizer's *fit_transform()* function on the documents. This will build a sparse document-term matrix for all documents (rows) and terms (individual tokens within those rows) in documents, returning the matrix as output.

Q3 (10): Compute Word2Vec vectors for the dataset.

1. Use the *load_w2v()* function to import word vector representations into your Python environment. This function will populate a dictionary with words as keys and their corresponding 300-dimensional NumPy arrays as values, based on pretrained model .

2. Following it you will implement the *w2v ()* function to fetch embeddings of a word from the loaded dictionary. If the word is not found in the dictionary, return a zero vector of size (300,).

3. Now to create vector-based embeddings for your dataset you will need to implement string2vec function. The string2vec function will convert a string into a single, averaged word vector.
   3.1. First, tokenize the string using the *get_tokens ()* function.
   3.2. Then, use the w2v function to retrieve a vector for each token. Finally, compute the average of these vectors.

   Step-by-Step Process:

- Tokenization: Split the input string into tokens.
- Vector Extraction: Retrieve a vector for each token using the w2v function.
- Averaging: Calculate the mean across all token vectors to produce a single 300-dimensional embedding.

Example:

- Tokens: ["i", "love", "nlp"]
- Vectors: [[0.5, 0.2, 0.3, 0.4, 0.1], [0.4, 0.3, 0.1, 0.2, 0.1], [0.1, 0.3, 0.2, 0.0, 0.4]]
- Averaged Vector: [0.33, 0.27, 0.2, 0.2, 0.2]

Q4 (5): Instantiate your machine learning models.

Implement the instantiate models function by calling the constructors for the following classes. Set the random state parameter (if applicable) in each model to exactly 18 when instantiating to produce replicable results (this is also necessary for auto grading to work correctly). Return both the classification models:

- Naive Bayes (sklearn.naive_bayes.GaussianlNB)
- Logistic Regression (sklearn.linear_model.LogisticRegression)
    - Set it for multinomal regression and use 'lbfgs' solver.

Both the models have already been imported in the skeleton code. Review the documentation for these models to make sure you understand how to use the model constructors correctly.

Q5 (10): Train your machine learning models.

Implement the *train_model_tfidf()* and *train_model_w2v ()* functions to train your machine learning models. You will need to implement the beginning parts of these two functions slightly differently.

1. For *train_model_tfidf()*, you'll convert your input TF-IDF features to a dense NumPy array and then fit your model to this data.
2. For *train_model_w2v ()*, you'll call string2vec for each training document, create a list of training document vectors, and then transform it so that it is the correct shape before fitting your model to the data.
3. In each case, the input training data array to a model's **fit()** function should have one row per document, with each row containing a representation vector (either a TFIDF vector or a 300-dimensional averaged embedding vector). The input training data labels should be fine as they are, without any changes.

Q6 (10): Test your classification models on the provided test set.

Implement the *test_model_tfidf()* and *test_model_w2v()* functions. Again, you'll need to implement these functions slightly differently. A model and embedding specific csv file will be generated in the *main()*. Remember to upload them along with your code.

1. For *test_model_tfidf(),* extract a TFIDF vector using the vectorizer you created

previously. Make sure to convert this vector to a dense NumPy array.

2. For *test_model_w2v()*, convert each document into an averaged Word2Vec embedding using string2vec function and create a list of test document vectors in the correct shape for predicting model output.

3 . In each case, you should then predict a label for each test document to compare your predicted labels with the "gold standard" labels for those documents to see how well each model performs.

4. Implement *evaluate_performance()* function to compare the predicted labels with the actual "gold standard" labels provided for the test documents.

   You should compare performance using four different metrics:

   - Precision
   - Recall
   - F1-score
   - Accuracy

You are recommended to use the relevant functions from **sklearn.metrics** to obtain performance measures as our target is multiclass and you will need to calculate *weighted* scores.

Q7 (25): Part-of-Speech Tagger using Hidden Markov Models

Complete *get_tree_bank_data()* using nltk.corpus.treebank.tagged_sents() to load NLTK Treebank tagged sentences. Allocate the first 80% of these sentences for training and the remaining 20% for testing.

From each sentence, extract words and their corresponding tags to create a comprehensive vocabulary of all unique words and a set of all unique tags.

To implement the Viterbi algorithm, you need 2 components,

• Tag transition probability matrix A: It represents the probability of a tag occurring given the previous tag or $p(t_i|t_{i-1})$. We compute the maximum likelihood estimate (MLE) of the probability by counting the occurrences of the tag t_{i-1} followed by tag t_i.

$p(t_i|t_{i-1}) = count\ (t_{i-1}, t_i)\ /\ count(t_{i-1})$

• Emission probability matrix B: It represents the probability of a tag t_i being associated with a given word w_i or $p(w_i|t_i)$. MLE estimate is:

$p(w_i|t_i) = count\ (t_i, w_i)\ /\ count(t_i)$

Since the number of tags is smaller, creating matrix A is time efficient whereas generation of matrix B will be very expensive due to vocabulary size.

Implement the method compute_tag_trans_probs() to calculate matrix A by parsing the sentences in the training set and counting the occurrences of the tag $ti-1$ followed by ti.
Implement the method emission_probs() to calculate the emission probability of a given word for each tag.

Next step in HMM is decoding which entails determining the hidden variable sequence of observations. In POS tagging, decoding is to choose the sequence of tags most probable to the sequence of words. We compute this using the following equation:

$$\hat{t}_{1:n} = \underset{t_1 \ldots t_n}{\operatorname{argmax}} \prod_{i=1}^{n} p(w_i|t_i)p(t_i|t_{i-1})$$

The optimal solution for HMM decoding is given by the Viterbi algorithm, a dynamic approach to the computation of the decoded tags. Implement the algorithm using the two methods, compute_tag_trans_probs() and emission_probs() implemented above and return the sequence of tags corresponding to the given sequence of words. Refer to section 8.4.5, Fig. 8.10 of Speech and Language Processing book.

Evaluate the performance of the model in terms of accuracy on the test set.

Q8 (10): Named Entity recognition using Stanford NER

Implement Named Entity Recognition (*get_ner()*) using Stanford NER tagger to identify different entities in movie summaries.

1. **Setup Stanford NER:**
    - Java: The Stanford NER tagger is written in Java, so you will need Java installed on your machine in order to run it. Make sure that you have Java version 1.8+ on your system.
    - Download Stanford Named Entity Recognizer version 4.2.0: https://nlp.stanford.edu/software/CRF-NER.shtml#Download . This includes the jar file for the NER tagger, as well as pre-trained models that will be used to label the text with named entities.
    - Do remember the paths to your model and jar file
2. **Implement Named Entity Recognition:**
    - Set the paths for your jar file and the 7 classes ner model.
    - Add your previous functions to process the text from the movie summaries. It involves splitting the text into sentences and further into words.

Link to the book: https://web.stanford.edu/~jurafsky/slp3/8.pdf

- Setup the tagger model using nltk's StanfordNERTagger in get_ner function.
- Complete the get_ner function to generate a list of tuples containing token and their respective named entity tag.

4. Written (15 points)

Q9 (15): Select a "best" classification model for genre classification.

Review the precision, recall, F_1, and accuracy results reported for each model, with each type of representation (TFIDF and Word2Vec), and compare the models' performance with one another. You can do this using the classification_report.csv file that is automatically generated as part of the skeleton code in main().To find incorrect instances you, use the csv files with predicted labels generated and compare them against the labels in test data.

Answer the following questions in at least one sentence each:

- Which model do you think performs best on the test set?
- Which metric(s) did you base your decision on?
- Why did you decide that these metrics were most important for this task?
- Briefly describe two instances where your best model predicted incorrectly.
- What potentially might have led the model to predict them incorrectly?

You may find that your choice for the "best" model may be different from that of your peers. Different designers may have different rationale for their design choices, and that's why writing a strong justification for your methods is important.

5. Rubric

This assignment will be graded according to the rubric below. Partial points may be awarded for rubric items at the discretion of the course staff.

Q1 (5 points possible)	
(Autograded)	+5
Q2 (10 points possible)	
(Autograded)	+10
Q3 (10 points possible)	
(Autograded)	+10
Q4 (10 points possible)	
(Autograded)	+10
Q5 (5 points possible)	
(Autograded)	+5
Q6 (10 points possible)	
(Autograded)	+10
Q7 (25 points possible)	
(Autograded)	+25
Q8 (10 points possible)	
(Autograded)	+10
Q9 (15 points possible)	
Which model do you think performs best on the test set?	+2
Which metric(s) did you base your decision on?	+2
Why did you decide that these metrics were most important for this task?	+4
Briefly describe two instances where your best model predicted incorrectly.	+4
What potentially might have led the model to predict them incorrectly?	+3