CSCE-470 Programming Assignment #2
Due: Sunday, October 13th, 2024 by 11:59pm

In this programming assignment, you will implement three classification algorithms, namely, Naive Bayes, k-NN and Rocchio classifier. The classification algorithms are abstracted away in their respective classes: `KNN`, `NaiveBayes`, `Rocchio`. You need to implement train() and predict() functions for each classifier.

Naive Bayes Classifier uses the Bayes rule to classify documents from their bag of words representations.

Train: you need to calculate 1). prior probabilities of classes and 2). word probabilities conditioned on each class. You need to apply laplace smoothing when you calculate word probabilities, the probability of a word for a candidate class can be computed with the following equation:

$$P(w \mid c) = \frac{COUNT(w,c) + 1}{\left(\sum_{w' \epsilon V} COUNT(w',c)\right) + |V + 1|}$$

Predict: using maximum a posteriori (MAP) estimation, the classifier predicts the class that has the maximum sum of the logarithm of prior and likelihood probabilities:

$$c_{NB} = argmax_{c_j \epsilon C} \{ log(P(c_j)) + \sum_{i \epsilon doclen} log(P(w_i \mid c_j)) \}$$

K-Nearest Neighbor (k-NN) Classifier predicts the class by looking at the k-nearest neighbor for the given candidate in a vector space.

Train: essentially nothing to be done except processing and storing training instances, you need to 1). convert the training documents into raw term frequency vectors.

Predict: given a new document to predict a label for, 1). convert the document into its raw term frequency vector, 2). identify the K (a hyperparameter you set, try to set it as 1, 3 or 5) nearest neighbors of this document among the training documents based on the cosine similarity measure, and 3). assign the majority label among class labels of the K nearest neighbors. The cosine similarity between two documents in the vector space is defined as:

$$similarity(A,B) = \frac{A.B}{||A|| * ||B||} = \frac{\sum_{i=1}^{n} A_i * B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} * \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Rocchio Classifier It predicts a class for a given candidate by finding out the closest class centroid in the document vector space.

Train: first, 1). convert the training documents into raw term frequency vectors and 2). calculates a centroid for each class using the normalized training document vectors. The centroid for a class c is calculated by the following equation:

$$\mu(c) = \frac{1}{|D_c|} \sum_{d \in D_c} v(d)$$

Predict: given a new document to predict a label for, 1). convert the document into its raw term frequency vector and 2). calculate cosine similarities between the normalized new document vector and class centroid vectors, and 3). assign the class label of the closest centroid to the given document.

For more details about the three classifiers, refer to the lecture slides in the learning module "Text-classification".

Data

The data consists of email documents where each email is associated with one of the following 10 different classes:
talk_politics_mideast (0), comp_sys_mac_hardware (1), rec_sport_baseball (2), rec_sport_hockey (3), talk_politics_misc (4), comp_windows_x (5), comp_graphics (6), comp_sys_ibm_pc_hardware (7), talk_politics_guns (8), talk_religion_misc (9).

The classes can be thought of like the topic of an email. For instance, the email classified as "*rec_sport_baseball*" will be related to the baseball sport.

The data is divided into three splits: train, validation, and test. Train split contains 357 emails and validation split consists of 94 emails. The test split is not made public as it will be used for grading.

Evaluation Metric

The three classifiers can be evaluated with the following metrics:

1. Accuracy: $correctly\ predicted\ documents\ /\ total\ documents$
2. Precision: $true\ positives\ /\ (true\ positives\ +\ false\ positives)$
3. Recall: $true\ positives\ /\ (true\ positives\ +\ false\ negatives)$
4. F1: $2 * precision * recall\ /\ (precision\ +\ recall)$

The starter code contains a script for evaluating the classification algorithms. The evaluation code prints the above-mentioned metrics along with the confusion matrix. Moreover, the script calculates precision, recall, and F1 for each class, as well as the macro-average of all the classes.

Code

The starter code contains two directories: `src` and `data`.

The `src` directory contains the following files:

1. `knn.py`: Implementation of the k-nearest neighbor classifier. **(3 points)**
2. `naive-bayes.py`: Implementation of the naive bayes classifier. **(3 points)**
3. `rocchio.py`: Implementation of the rocchio classifier. **(3 points)**
4. `utils.py`: Helper functions used in the classifiers including the evaluation function.
5. `data.py`: Responsible for reading the train, validation and test split.

The `data` directory includes the email documents in the `train` directory. The labels for each email is provided in `train-split.txt (contains labels for 357 training documents)`, `train-half-split.txt (contains labels for half of the training documents)` and `val-split.txt (contains labels for validation documents)`.

You are expected to modify the classifier files only and not change `utils.py` and `data.py`, but you should read all the python files to best understand how to optimally implement the classification algorithms. Each classifier contains `train()` and `predict()` functions that contain helpful TODOs to be completed by you. Read the code at the end of the classifier files to see how these functions are used to train and evaluate the models. Feel free to add `print()` statements for debugging or visualizing the arguments for the functions but remove all `print()` statements before your final submission.

The classifiers can be trained and evaluated by running the corresponding files with `python3` and **will not run** with any version of `python(<3)`.

```
# By default all classifiers will train on the entire training
# set, train-split.txt
$ python3 naive-bayes.py
$ python3 knn.py
$ python3 rocchio.py
# For using train-half-split.txt for training, pass
# "train_half" as an argument (works for all classifiers):
$ python3 naive-bayes.py train_half
```

The above classifier scripts will print the confusion matrix, accuracy, precision, recall and F1 score for train and validation data split.

Report

The report should consist of three parts:

1. The first part should report the results for each classification algorithm. Run your classifiers using both the full training dataset (use no argument) and the half training data (pass the argument `train_half`), and report the performance of the classifiers in both settings. **(2 points)**
2. Secondly, you should talk about the variance-bias trade-off for each classifier and how it is reflected in your results. **(2 points)**
3. Finally, you should comment on the time complexity of `train()` and `predict()` implementations for each classifier. You should also reason about why one classifier is faster/slower than the other two. **(2 points)**

Extra Credit (4 points!)

Features are extremely important for machine learning algorithms. The current features used by the three algorithms are *frequency* based *unigram* (bag-of-words) features. To earn extra credit for this assignment, you are required to modify the features or implement new features, and report experimental results and analysis in your report. The following are the two items each deserving 2 extra credits, and feel free to work on any one of them or both.

(**2 Points!)** Ignore word *frequency* information, convert the current raw term frequency vectors to **binary** vectors. Specifically, represent each document as a vector with only binary (1 or 0) values, set 1 as the entry value for words that have appeared *one or more* times in a document. In your report, include results of the three classifiers with the new binary features and answer the following questions: for each classifier, do binary features improve or hurt the performance compared with the original frequency based features? Please explain why the new features have caused the performance change.

(**2 Points!**) Instead of *unigram* features, implement frequency based **bigram** features. Every two consecutive words forms a bigram. Specifically, represent each document as a vector where each dimension corresponds to a unique bigram (instead of a unique word as in the original bag-of-word document representations), and the value for each dimension is equal to the frequency of the corresponding bigram in the document. In your report, include results of the three classifiers with the new frequency based bigram features and answer the following questions: for each classifier, do bigram features improve or hurt the performance compared with the original unigram features? Please explain why the new features have caused the performance change.

Grading Criteria

Your code will be graded mainly based on the correctness. You should analyze the correctness of your implementations with the help of the evaluation metrics provided in the starter code. We will run your classification algorithms on a separate held-out test set and make sure it runs properly. Your written report will be graded based on the grade allocations on individual items:

1. Code: 9 points. 3 points for each classifier.
2. Reports: 6 points.
3. (Optional) Extra credit: 4 points.

Cheating on either code or report will be harshly penalized.

Electronic Submission Instructions

You only need to submit 2 things:

1. The source code files. Please compress the whole "src" folder into a .zip file and submit the .zip file.
2. The written report in pdf format.

Please do NOT include the data files or any other file in your submission. Submit the two files, the source code .zip file and the written report pdf file, via canvas.