HW1: Linear Classifiers

CSCI 662: Fall 2022

out: Aug 29, 2022
due: Sep 14, 2022

This assignment is about writing two (or more) linear classifiers and testing them out on multiple kinds of classification tasks. While the key models should not be difficult to implement, expect to spend some time setting up the general framework (which you will largely re-use in HW2) and considerable time investigating different classes of features that will be applicable for each of the tasks.

Code To Write

1. Write a trainer for at least a Naive Bayes model and a Perceptron model (you can also try a Support Vector machine and/or Logistic Regression model), in python. We have provided stub code for a trainer called `train.py`. It displays its invocation and brief help when invoked as `python3 train.py -h`. The program takes the following options:

   - `-m [naivebayes, perceptron, ...]` to specify which kind of model you are training
   - `-i <inputfile>` to specify a training file to be read. Training files will be in the form `<text>TAB<label>`, i.e. a line of text (that does not contain a tab), a tab, and a label.
   - `-o <modelfile>` to specify a model file to be written

   It can take other options too such as specifying hyperparameters, a dev set so that held-out loss can be displayed, etc.

2. Write a classifier that takes in a model and unlabeled data set and labels it. We have also provided stub code for a classifier called `classify.py`. It should display its invocation and a brief help when invoked as `python3 classify.py -h`. The program should take the following options:

   - `-m <modelfile>` to specify the trained model file to read (i.e. the output of `train.py`)
   - `-i <inputfile>` to specify a test file to be read. Test files will be in the form `<text>`, i.e. a line of text that does not contain a tab. For each line, a label should be predicted.
   - `-o <outfile>` to specify an output file to be written. The output file should contain one label for each line in the input file.

For pseudocode and more details about classifiers, please refer to the class notes on linear models.

Coding Requirements

- The classifier should be able to handle words that haven't been seen before (i.e. OOV). You could do this in a variety of ways, or even try multiple ways and compare their effects (e.g. using BPE vs. simple smoothing and OOV handling).
- The classifier should not need to know anything about the training method used by the model. However, the model file can (and should) contain information about how to generate features, as well as feature weights
- The trainer/classifier should be **multi-class**, not binary, and should generate actual class labels. Assume all classes are seen during training and save the label info in the model file.
- In your writeup, discuss different feature sets you tried and the different scores you got on your own internal test set and on the blind validation set (on Vocareum).
- Submit `train.py`, `classify.py`, any other code needed, and at least eight model files, as discussed below.
  The files should be named `[nb, perceptron].[questions,products,4dim,odiya].model`.
- Make sure the Vocareum auto-scoring script runs and gives reasonable results. You may also try running alternate models interactively on Vocareum. Some larger models may not run but baselines definitely should! We'd like to try these out so please provide instructions for trying the models you most want to demonstrate by creating a usable `README` file.

Coding Recommendations

- Structure your code with a `getFeatures` function inside a `Features` class (you could subclass this class in order to more flexibly try out features) that returns a vector of features for each input sentence.
- The model file should contain feature weights and a `Feature` instance so it knows how to process new sentences and how to calculate model cost.
- Write a score file that takes two output files and produces an accuracy so you can test your scores without having to submit to Vocareum.
- For each data set, save a small portion off to use as development and another small portion to use as test before submitting for blind test.
- Try different feature sets! First try to optimize on one of the tasks, then see how that generalizes to other tasks. Can you find one feature set that is reasonably good for all tasks? (Note: you don't need to try to be the super best on all tasks, because that might take you too long. However, you should probably spend more time on feature sets than on modeling; once you have the basic structure of the trainer/classifier, the model learning should be pretty easy.)
- The "4dim" data set can be broken down into two independent binary predictions. Optionally, try making those predictions independently, or even conditionally.

Data

We have provided several different classification tasks with the following datasets:

- `products`: Very variable length lines of various kinds of English product reviews that are either positive (pos) or negative (neg). There are 32,592 labelled reviews used for training and 1,000 reviews for validation.

- `4dim`: English reviews of variable length that are positive or negative and truthful or deceptive (pos.tru, pos.dec, neg.tru, neg.dec). There are 1,560 labelled reviews used for training and 40 reviews for validation.
- `odiya`: News headlines written in Odiya (or Oriya) which is a low-resourced language spoken in India. They are classified into business, sports and entertainment. There are 15,200 labelled headlines used for training and 3,801 for validation.
- `questions`: Questions to be classified into 6 categories: abbreviation, entity, description, human, location, and numeric value. The language of the questions is uncertain; before undertaking the arduous hand-transcription of this data, we noticed a faded label that said "ufvytar" on the library archive box. But there's something mysterious about this data...

All datasets can be found on Vocareum in `work/datasets` where train labelled files are named 4dim, questions, odiya, products.train.txt whereas validation files are hidden from your view as auto-grading scripts will automatically run your classify.py code with your trained models on the validation on our end; when you submit, if you provide trained model files called `[nb,perceptron].[questions,products,4dim,odiya].model`, we will test your models on hidden files and return a score for each model. If this score differs greatly from your expectation you may a) be overfitting, or b) have some design flaw in your code structure (e.g. hard-coded assumptions). You can submit any models you want and name them how you choose but you should have at least one `nb.` and `perceptron.` model file for each data set.

Your Report

Your report should at a *minimum*:

- Explain the key differences in the methods you used
- Justify your choice of features (include citations where relevant), when optimizing for one task and/or when trying to generalize over many tasks
- Where relevant, show learning rates, discuss overfitting and loss convergence. Use graphs and tables *appropriately*, not superfluously. This means the graphs/tables should emphasize the message you are delivering, not simply be in place without thinking about why you are using that particular medium to convey an idea.
- Discuss: How do different properties of the different data sets affect performance? Apart from different categorizations, do the size, genre, domain of the data matter? How so?

Use the ACL style files : https://github.com/acl-org/acl-style-files

There are many ways to write and complie LaTeX; I generally use Overleaf (www.overleaf.com) for minimal headaches but I have colleagues who abhor Overleaf and greatly prefer to compile on their own machines. Do what works for you.

Your report should be at least two pages long, including references, and not more than four pages long, not including references (i.e. you can have up to four pages of text if you need to). Just like a conference paper or journal article it should contain an abstract, introduction, experimental results, and conclusion sections (as well as other sections as deemed necessary). Unlike a conference paper/journal article, a complete related works section is not obligatory (but you may include it if it is relevant to what you do).

Grading

As discussed in class, grading will be roughly broken down as follows:

- about 50% – did you clearly communicate your description of what you implemented, how you implemented it, what your experiments were, and what conclusions you drew from them? This includes appropriate use of graphics and tables where warranted that clearly explain your point. This also includes well written explanations that tell a compelling story. Grammar and syntax are a small part of this (maybe 5% of the grade, so 10% of this section) but much more important is the narrative you tell. Also a part of this is that you clearly acknowledged your sources and influences with appropriate bibliography and, where relevant, cited influencing prior work.
- about 20% – is your code correct? Did you implement what was asked for, and did you do it correctly?
- about 20% – is your code well-written, documented, and robust? Will it run from a different directory than the one you ran it in? Does it rely on hard-codes? Is it commented and structured such that we can read it and understand what you are doing?
- about 10% – did you go the extra mile? Did you push beyond what was asked for in the assignment, trying new models, features, or approaches? Did you use motivation (and document appropriately) from another researcher trying the same problem or from an unrelated but transferrable other paper?

'Extra Mile' ideas

This is not meant to be comprehensive and you do not have to do any of the things here (nor should you do all of them). But an 'extra mile' component is 10% of your grade. Doing something more innovative (though well-reasoned) than what is listed here will be better than doing exactly what is written here, and doing a correct implementation with a thorough analysis will be better than an incorrect implementation or a trivial analysis.

- Add in the logistic regression model and/or SVM and compare to the others.
- Use some out-of-the-box pretrained models, e.g. BERT or out-of-the-box frameworks, e.g. LSTMs or Transformers. See below on restrictions of the use of libraries and pretrained models.
- Add additional data to train your models.
- Once you are done with your primary models, compare your implementations to 'off the shelf' implementations. Are there any differences in speed and/or performance? Can you determine what accounts for these differences?
- Dig into the ACL archives and find ideas for feature sets; try them out, analyze performance.

Rules

- This is an individual assignment. You may not work in teams or collaborate with other students. You must be the sole author of 100% of the code and writing you turn in.
- You may not look for solutions on the web, or use code you find online or anywhere else.
- You may not download the data from any source other than the files provided on Vocareum, and you may not attempt to locate the test data on the web or anywhere else.
- You may use other *external* data that is not the training/test data provided, and you can make modifications to that data as needed, but you must document the data and modifications in your writeup. For the core assignment (i.e. not the 'extra mile' component) you may *not* use externally trained models such as BERT.
- For the core assignment (i.e. not the 'extra mile' component) you may use packages in the Python Standard Library (including numpy). You may not use any other packages (e.g. scikit-learn, keras, tensorflow, or other machine learning libraries).

- You may use external resources to learn basic functions of Python (such as reading and writing files, handling text strings, and basic math), but the extraction and computation of model parameters, as well as the use of these parameters for classification, must be your own work.
- Failure to follow the above rules is considered a violation of academic integrity, and is grounds for failure of the assignment, or in serious cases failure of the course.
- We use plagiarism detection software to identify similarities between student assignments, and between student assignments and known solutions on the web. Any attempt to fool plagiarism detection, for example the modification of code to reduce its similarity to the source, will result in an automatic failing grade for the course.
- If you have questions about what is and isn't allowed, post them to Piazza!