SI 630: Homework 1 – Classification

Due: Wednesday, January 29, 5:30pm

1 Introduction

Homework 1 will introduce you to building your first NLP classifiers from scratch. Rather than use many of the wonderful, highly-customizable machine learning libraries out there (e.g., `sklearn`), you will build bare-bones (and much slower) implementations of two classic algorithms: Naive Bayes and Logistic Regression. Implementing these will provide you with deep understanding of how these algorithms work and how the libraries are actually calculating things. Further, you'll be implementing two knob to tune in the data: smoothing for Naive Bayes and the learning rate for Logistic Regression. In implementing these, you will get to adjust them and not only understand their impact but understand *why* they're impacting the result.

The programming assignment will be moderately challenging, mostly stemming from the conceptual level. The code for each algorithm is be 20-30 lines, with additional lines for simple I/O and boilerplate code. The big challenge will be from trying to ground the mathematical concepts we talk about in class in actual code, e.g., "what does it mean to calculate the likelihood???". Overcoming this hurdle will take a bit of head scratching and possibly some debugging but it will be worth it, as you will be well on your way to understanding how newer algorithms work and even how to design your own!

There are equations in the homework. You should not be afraid of then—even if you've not taken a math class since high school. Equations are there to provide precise notation and wherever possible, we've tried to explain them more. There are also many descriptions in the lecture materials and textbooks. Getting used to equations will help you express yourself in the future. If you're not sure what something means, please ask; there are six office hours between now and when this assignment is due, plus an active Canvas discussion board. You're not alone in wondering and someone will likely benefit from your courage in asking.

Given that these are fundamental algorithms for NLP and many other fields, you are likely to run into implementations of them online. While you're allowed to look at them (sometimes they can be informative!), all work you submit should be your own (see Section 8).

Finally, remember this is a no-busywork class. If you think some part of this assignment is unnecessary or to much effort, let us know. We are happy to provide more detailed explanations for *why* each part of this assignment is useful. Tasks that take inordinate amounts of time could even be a bug in the homework and will be fixed.

2 Classification Task

The School of Information's mission is literally "We create and share knowledge so that people will use information – with technology – to build a better world." One major impediment to people's

use of technology today is uncivil behavior online. Hate speech, bullying, toxic language, and other forms of harassment all affect people's ability to access information and engage in social activities. In this homework, you'll be tackling this problem head-on by building a cyber troll detector! Help mitigate people attacking one another by flagging aggressive tweets for administrators. Your task will be to build two classifiers that label text as either an aggressive tweet or a regular tweet. The fate of free discourse rests on your shoulders.

Given the toxic nature of people's tweets, please be aware that you may see messages that are deeply offensive and an affront to all that is right and good. Neither the instructor nor the community at SI agrees with or supports these viewpoints. Their inclusion in the homework is entirely for the purposes of development new NLP technologies that help administrators and moderators identify and remove these messages from otherwise civil discourse. As such, it would be difficult to develop an cyber troll classifier without actually seeing what they look like.

Notation Notes : For the whole assignment description we'll refer to classifier features as $x_1, \ldots, x_n \in X$ where x_i is a single feature and X is the set of all features. When start out each feature will correspond to the presence of a word; however, you are free in later parts of the homework to experiment with different kinds of features like bigrams that denote to consecutive words, e.g., "not good" is a single feature. We refer the to the class labels as $y_1, \ldots, y_n \in Y$ where y_i is a single class and Y is the set of all classes. In our case, we have a *binary* classification tasks so there's really only y_1 and y_2. When you see a phrase like $P(X = x_i|Y = y_j)$ you can read this as "the probability that we observe the feature (X) x_i is true, given that we have seen the class (Y) is y_j".

We'll also use the notation $exp(x_i)$ to refer to e^{x_i} at times. This notation lets us avoid superscript when the font might become too small or when it makes equations harder to follow.

Implementation Note : Unless otherwise specified, your implementations should *not* use any existing off-the-shelf machine learning libraries or methods. You'll be using plain old Python (or R, if brave) along with numeric libraries like numpy to accomplish everything.

3 Data

Homework 1 has three associated files (you can find them on our Kaggle competition associated with this homework):

- `X_train.txt` This file contains the tweets you will use to train your classifiers.
- `y_train.txt` This file contains the human-annotated labels of the tweets in the previous file.
- `X_dev.txt` This file contains the tweets you will use as the development data.
- `y_dev.txt` This file contains the human-annotated labels of the tweets in the previous file.
- `X_test.txt` This file is the tweets you will make predictions using your trained classifiers and upload results.

As a part of this assignment, we'll be using Kaggle in the classroom to report predictions on the test data. This homework is not a challenge, per se. Instead, we're using Kaggle so you can get a sense of how good your implementation's predictions are relative to other students. Since we're all using the same data and implementing the same algorithms, your scores should be relatively close to other students. If you decide to take up the optional part and do some feature engineering, you might have slightly higher scores, but no one will be penalized for this.

We've set up two Kaggle competitions for Logistic Regression (LR) and Naive Bayes (NB) classifiers respectively:

- LR competition link: `https://www.kaggle.com/c/si630w20hw1lr`
- LR invitation link: `https://www.kaggle.com/t/5f1651bc7bab410389b14c8697b07df6`
- NB competition link: `https://www.kaggle.com/c/si630w20hw1nb`
- NB invitation link: `https://www.kaggle.com/t/2573fa891af14b53b4009f63068ead45`

4 Task 1: Naive Bayes Classifier

4.1 Part 1

Implement a Naive Bayes classifier. Recall that the classifier was defined as

$$\hat{y} = \arg\max_{y_i \in \mathcal{Y}} P(Y = y_i|X)$$

where $\mathcal{Y}$ is the set of classes (i.e., personal insult or not, in our case). This equation can seem pretty opaque at first, but remember that $P(Y = y_i|X)$ is really what's the probability of the class y_i given the data we see in an instance. As such, we can use Bayes Rule to learn this from the training data.

In Task 1, you'll implement each step of the Naive Bayes classifier in separate functions. As a part of this, you'll also write code to read in and tokenize the text data. Here, tokenization refers to separating words in a string. In the second half, you'll revisit tokenization to ask if you can do a better job at deciding what are words. Task 1 will provide much of the boiler plate code you'll need for Task 2.

- Write a function called `tokenize` that takes in a string and tokenizes it by whitespace, returning a list of tokens. You should use this function as you read in the training data so that each whitespace separated word will be considered a different feature (i.e., a different x_i).
- Write a function called `train` to compute $P(X = x_i)$, $P(Y = y_j)$, and $P(X = x_i|Y = y_i)$ from the training data. The function should also include an argument called `smoothing_alpha` that by default is 0 but can take on any non-negative value to do additive smoothing (see Slide 143 from Lecture 1), which you might also see as Laplacian smoothing.
- Write a function called `classify` that takes in a tokenized document (i.e., a list of words) and computes the Naive Bayes classification, returning the class with the highest posterior

probability. Note that not you might not have all the new document's words in the training data. Be sure to take this into account in your implementation depending on whether you used smoothing or not!

- Train the classifier on the training data the run the classifier on the development data with no smoothing and report your performance in your submission in terms of F1.[1]
- What happens as you change the value of `smoothing_alpha`? Include a plot of your classifier's performance on the development data where (i) your model's performance is on the y-axis and (ii) the choice in `smoothing_alpha` is on the x-axis. Note that most people use $\alpha = 1$; does this value give good performance for you?
- Submit your best model's predictions on the test data to KaggleInClass competition for Naive Bayes. **Note that this is a separate competition from the Logistic Regression one so that you can compare your scores.**

4.2 Part 2

Notice that we are probably doing a bad job of tokenizing due to punctuation. For example, "good" and "good," are treated as different features because the latter has a comma at the end and "Good" and "good" are also different features because of capitalization. Do we want these to be different? Furthermore, do we want to include every token as a feature? (Hint: could a regular expression help you filter possible features?) Note that you have to implement the tokenization yourself; no importing NLTK and wrapping their functions (though you might take some inspiration from them).

- Write a better tokenizing function called `better_tokenize` that fixes these issues. In your report, describe which kinds of errors you fixed, what kind of features you included, and why you made the choices you did.
- Recompute your performance on the development data using the `better_tokenize` method. Describe in your report what impact this had on the performance.

4.3 Part 3 (optional)

Parts 1 and 2 only used *unigrams*, which are single words. However, longer sequences of words, known as n-grams, can be very informative as features. For example "not offensive" and "very offensive" are very informative features whose unigrams might not be informative for classification on their own. However, the downside to using longer n-grams is that we now have many more features. For example, if we have n words in our training data, we could have n^2 bigrams in the worst case; just 1000 words could quickly turn into 1,000,000 features, which will slow things down quite a bit. As a result, many people threshold bigrams based on frequency or other metrics to reduce the number of features.

[1] You can use `sklearn`'s implementation of F1 (`sklearn.metrics.f1_score`) to make your life easier: `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html`

In Part 3, you'll experiment with adding bigrams (two words) and trigrams (three words) and measuring their impact on performance. Part 3 is entirely optional and included for people who want to go a bit further into the feature engineering side of things.

- Count how many unique, unigram, bigrams, and trigrams there are in the training data and report each number.
- Are the bigram and trigram counts you observe close to the worst case in terms of how many we could observe? If not, why do you think this is the case? (Hint: are all words equally common? You might also check out Zipf's Law).
- What percent of the unique bigrams and trigrams in the development data were also seen in the training data?
- Choose a minimum frequency threshold and try updating your solution to use these as features. We recommend creating a new method that wraps your tokenize method and returns a list of features.

5 Task 2: Logistic Regression

In the second task, you'll implement logistic regression, which you might recall is

$$P(y = 1|x, \beta) = \frac{1}{1 + exp\left(-\sum_{i=1,\ldots,N} x_i\beta_i\right)}$$

Note that when you implemented Naive Bayes, it didn't care how many classes were present. In contrast, Logistic Regression is restricted to two classes, which we represent as *binary* so that y is either 0 or 1. Conveniently, your training data is already set up for this (though you'll need to use `int()` to convert the string).

Your implementation will be one of the simplest possible formulations of logistic regression where you use gradient descent to iteratively find better parameters β. Smarter solutions like those in `sklearn` will use numerical solvers, which are much (much) faster. The purpose of this problem is to give you a sense of how to compute a gradient.

For Task 2, you can (and should) re-use all the boilerplate and tokenization code from the Naive Bayes classifier. It is assumed that your solution will already include something to read in the training data and construct a matrix where rows are instances and columns denote features (e.g., if $X[0, 7] = 3$, it means that the first instance (index 0) saw the word whose feature index is 7 occur 3 times).

- Implement a function called `sigmoid` that implements the sigmoid function, S

$$S(X) == \frac{1}{1 + exp(-X)}$$

. Your function should be *vectorized* so that it computes the sigmoid of a whole vector of numbers at once. Conveniently, numpy will often do this for you, so that if you multiply

a number to a numpy array, it will multiply each item in the array (the same applies to functions used on an array (hint)). If you're not sure, please ask us! You'll need to use the sigmoid function to make predictions later.

- Implement a function called `log_likelihood` that calculates the log likelihood of the training data given our parameters β. Note that we could calculate the likelihoood but since we'll be using log-arithmetic version (hence the *log*-likelihood) to avoid numeric underflow and because it's faster to work with. Note that we **could** calculate the log likelihood ll over the whole training data as

$$ll = \sum_{i=1,...,n} y_i B^T x_i - log(1 + exp(B^T x_i))$$

  where β is the vector of all of our coefficients. However, you'll be implementing *stochastic gradient descent*, where you update the weights after computing the loss for a *single* randomly-selected (stochasticly!) item from the training set.

- Given some choice of β to make predictions, we want to use the difference in our prediction $\hat{Y}$ from the ground truth Y to update β. The gradient of the log likelihood tells us which direction (positive or negative) to make the update and how large the update should be. Write a function `compute_gradient` to compute the gradient. Note that we can compute the whole gradient using

$$\nabla ll = X^T(Y - \hat{Y})$$

  Note that Y is a binary vector with our ground truth (i.e., the training data labels) and $\hat{Y}$ is the binary vector with the predictions. To get a sense of why this works, think about what gradient will equal if our prediction for item i, $\hat{Y}_i$ is the same as the ground truth Y_i; if we use this gradient to update our weight for β_i, what effect will it have?

- Putting it all together, write a function `logistic_regression` that takes in a
  - a matrix X where each row has the features for that instance
  - a vector Y containing the class of the row
  - `learning_rate` which is a parameter to control how much you change the β values each step
  - `num_step` how many steps to update β before stopping

  Your function should iteratively update the weight vector β at each step by making predictions, $\hat{Y}$, for each row of X and then using those predictions to calculate the gradient. You should also include an *intercept* coefficient.[2]

  Note that you can make your life easier by using matrix operations. For example, to compute $\hat{Y}$, multiply the whole feature matrix X by the β vector If you're not sure how to do this, don't worry! Please come talk to us during office hours!

[2]The easiest way to do this is to add an extra feature (i.e., column) with value 1 to X; the functions `np.ones` and `np.stack` with `axis=1` will help.

- Write a function `predict` that given some new vector (i.e., something like a row from X), predict the class.
- Train your model on the training data `learning_rate=5e-5` (i.e., a very small number) and `num_steps = 1000`. Make a plot of the log-likelihood every step. Did the model converge at some point (i.e., does the log likelihood remain stable)?
- Change the `learning_rate` to a much larger and much smaller value and repeat the training procedure for each. Plot all three curves together and describe what you observe. You're welcome (encouraged, even!) to try additional learning rates. If your model is very slow or if it converges quickly, you can also reduce the number of steps for this question.
- After training on the training data, use your logistic regression classifier to make predictions on the validation dataset and report your performance using the F1.
- Submit your best model's predictions on the test data to the KaggleInClass competition for Logistic Regression. **Note that this is a separate competition so that you can compare your scores with the Naive Bayes model.**

6 Hints

A few hints to help get you started:

- The `set`, `defaultdict` and `Counter` classes in python will likely come in handy—especially for Naive Bayes.
- Be sure you're not implementing full gradient descent where you compute the gradient with respect to all the items. Stochastic gradient descent uses *one* instance at a time.
- If you're running into issues of memory and crashing, try ensuring you're multiplying *sparse* matrices. The dense term-document matrix is likely too big to fit into memory.

7 Submission

Please upload the following to Canvas by the deadline:

1. a PDF (preferred) or .docx with your responses and plots for the questions above. **Your submission needs to include your username on Kaggle.**
2. your code for the Naive Bayes part
3. your code for the Logistic Regression part

Code may be submitted as a stand-alone file (e.g., a `.py` file) or as a Jupyter notebook. We reserve the right to run any code you submit; **code that does not run or produces substantially different outputs will receive a zero**.

8 Academic Honesty

Unless otherwise specified in an assignment all submitted work must be your own, original work. Any excerpts, statements, or phrases from the work of others must be clearly identified as a quotation, and a proper citation provided. Any violation of the University's policies on Academic and Professional Integrity may result in serious penalties, which might range from failing an assignment, to failing a course, to being expelled from the program. Violations of academic and professional integrity will be reported to Student Affairs. Consequences impacting assignment or course grades are determined by the faculty instructor; additional sanctions may be imposed.