SI 630: Homework 1 – Classification

Due: Wednesday, January 26, 11:59pm EST

1 Introduction

Homework 1 will have you build your first NLP classifier, Logistic Regression. Logistic Regression is a simple, yet often surprisingly effective model for text classification—you may have even used Logistic Regression in one of the common python machine learning libraries like sklearn![1] In Homework 1, you'll build Logistic Regression *from scratch* to get a sense of how these models work. Your implementation will be very simple (and slow) but will give you much more intuition for how machine learning models work and, critically, how deep learning models work.

Homework 1 will have you implement Logistic Regression in *two ways*. In the first, you'll use `numpy` to implement all parts of Logistic Regression, including the stochastic gradient descent and update steps. In the second part, you'll implement the same approach using `pytorch`, a model deep learning library. For working with numeric data (e.g., vectors of numbers) pytorch and `numpy` are very similar.[2] However, `pytorch` is designed for training deep learning models and can automate much/all of the updating steps. We are having you implement Logistic Regression twice using different libraries to give you more intuition on what `pytorch` is doing (and how stochastic gradient descent works!) while also getting you up to speed on how to use `pytorch`, which you'll need for all later assignments.

The programming part of this assignment will be moderately challenging due to (1) needing to map concepts and equations to python code and (2) needing to learn how to use two new libraries. In practice, the code for logistic regression is about 20-30 lines (or less), with both the `numpy` and `pytorch` models sharing some additional code for simple I/O and setting up the data. 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 them—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 the Speech & Language Processing textbook that can help explain these more if you need. Logistic Regression is a *very* common technique, so you will also find good explanations online if you need (e.g., the visualizations in this video, which can give you a sense of what the bias term is doing). Getting used to equations will help you express yourself in the

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`

[2] `https://rickwierenga.com/blog/machine%20learning/numpy-vs-pytorch-linalg.html`

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 Piazza 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 Logistic Regression online. While you're allowed to look at them (sometimes they can be informative!),[3] all work you submit should be your own (see Section 9).

Finally, remember this is a no-busywork class. If you think some part of this assignment is unnecessary or too 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.

The learning goals for this assignment are as follows:

- Understand how Logistic Regression works, what its parameters do, and how to make predictions
- How to work with sparse data structures
- How gradient descent and stochastic gradient descent work in practice
- Understand how to implement classifiers using `numpy`, and more generally, how to work with new python libraries
- Learn how to implement and train a (shallow) neural network with `pytorch`
- Improve your software development and debugging skills[4]

2 Classification Task: Political Framing

The 2020 Election season in the United States brought in record amounts of funding and voter turnout. A key part of that was the use of electronic messaging to encourage donations and voting. Through some very neat data collection, our colleagues at Princeton have collected ∼435K emails sent from candidates running for state and federal office, political parties, and other political organizations like Political Action Committees (PACs) Mathur et al. (2020).[5] These emails contain a surprising and fascinating amount of diversity and tactics for getting people to contribute—including "dark patterns" like having the email appear to be sent from a different sender. You can read more if interested in their paper describing the data and some analyses on it (it's a great read).

[3]The instructors would also like to caution that not all implementations are correct online, nor are all the implementations of a particular algorithm actually what is being asked for in this homework. Some students in prior semesters have found themselves more confused after reading many different descriptions/implementations of the algorithms and only gained clarity after trying to walk through the material themselves. As always, your instructors are here to help if you have questions!

[4]Seriously! Advanced software development will have you finding all sorts of weird bugs that will take some leaps to figure out. Be prepared but know that the skills you learn here will *greatly* help you in your career. Learning how to debug code using unfamiliar libraries and do this development is a key goal of this assignment!

[5]`https://electionemails2020.org/`

Political actors frequently use framing devices and rhetorical strategies to motivate their bases. These frames might look like describing the need to donate as a moral imperative or that the person's health or safety is at risk if the emailing candidate loses. Different political parties will employ a common set of strategies that they know resonate with their base, which suggests we should see some regularity in the messaging, despite the emails being sent by thousands of candidates or organizations. Your task is to build a classifier to assess **to what degree can we predict the political party of the sender from this type of messaging?**

Of course, these emails contain the candidate's name, which would make the classification task very easy if done from the raw text—the model would just learn which candidate was from which party! To focus particularly on framing, we've removed references to all Named Entities (people, places, organizations) from the emails for you, as well as any mailing addresses. While some noise does exist (NLP isn't perfect, after all), the data should better capture the message we want to analyze using a classifier.

Important Side Note on the Data: This data is provided by the folks at Princeton with some restrictions. The most important is that you will not republish the underlying data, so this data should never be put on your website, left on a public URL, or pushed to a public git repo (for example). You can view the full set of conditions and terms here.

3 Homework Design Notes

This homework is divided into three parts. **Part 1** has you implementing the common pre-processing steps that turn the text into a term-document matrix. You'll need to write code to read in the data, tokenize it, and convert it to some numeric representation. **Part 2** has you implement Logistic Regression using `numpy`, including all the details of the stochastic gradient descent. **Part 3** has you implement Logistic Regression using `pytorch`. In Part 3, you'll also do a few experiments using development data to see what is the effect of tuning certain hyperparameters.

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 starting 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 two consecutive words, e.g., "not good" is a single feature. We refer 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 (i.e., no using sklearn). You'll be using plain old Python along with numeric libraries like `numpy` and `pytorch` to accomplish everything.

4 Data

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

- `train.csv` This file contains the comments, their IDs, and their labels you will use to train your classifiers. The `party_affiliation` column contains the political party of who sent the email text in the `email_text`.
- `dev.csv` This file is the emails and their IDs that you will make political party predictions using your trained classifiers and upload results to Kaggle.
- `test.csv` This file is the emails and their IDs that you will make political party predictions using your trained classifiers and upload results to Kaggle.

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 "leaderboard competition," 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 a Kaggle competition for both bare-bones Logistic Regression and Logistic Regression with PyTorch. You'll first need to sign up using the invitation link then submit your scores to the competition.

- LR competition link: `https://www.kaggle.com/c/umsi630w22hw1-1`
- LR invitation link: `https://www.kaggle.com/t/5cac649ea0614941bc338ef7ec01dadf`
- PyTorch LR competition link: `https://www.kaggle.com/c/umsi630w22hw1-2`
- PyTorch LR invitation link: `https://www.kaggle.com/t/ac95c0cf9f5b4443a30a06fea071`

5 Part 1: Representing Text Data

Before we can start classifying, we need to turn our text data into some numeric representation that classifiers can work with. In class, we talked about using a *bag of words* representation—i.e., each document is represented as a vector x of length $|\mathcal{V}|$ (the number of unique words across all our documents) and we count how many times each word occurs in that document so that x_i is the number of times the word represented by column i occurs.

Part 1 is divided into two tasks: (1) tokenizing documents into words and (2) turning those lists of words into vectors.

5.1 Task 1.1: Tokenization

There are many, *many* ways to tokenize text. We'll try to use two in this homework to give you a sense of how you might do it. There are few wrong ways to tokenize, though some approaches will give you better results both in terms of token quality and downstream performance when you train your models.

- 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).

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/SpaCy 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.

We'll use these functions later when we evaluate the models.

5.2 Task 1.2: Building the Term-Document Matrix

As a thought experiment, consider what might happen if we make a term-document matrix for our training data. This matrix would be $\mathcal{D} \times \mathcal{V}$ in size. Since our vocabulary likely gets larger with each new document, this matrix will be huge—potentially tens or hundreds of gigabytes—and won't fit on your personal computer. Yet, libraries like `sklearn` work just fine, so what's going on? In practice, the term-document matrix is *very sparse*; most documents don't have most of the words, so we don't actually need to represent all these zeros!

In your solution, we'll want to use a sparse matrix representation. Conveniently the SciPy library has several sparse matrix implementations that we can use.[6] When building your term-document matrix you'll want to create one of these matrices.

When you create the matrix, you'll need to keep track of which dimensions correspond to which words, so that when you see a new document, you'll know how to create a new vector with the correct dimensions filled in for its words.

How many words should you include in your vocabulary $\mathcal{V}$? Including everything is tempting (it's easy after all). However, let's think about what happens if we include all words. First, word frequency follows Zipf's Law, which, in practice, means that most words occur very rarely (roughly 80% of the unique words make up less than 20% of the tokens we seen). As a result, most of the unique words will have very low frequency—with many occurring only once in our data. If a

[6] https://cmdlinetips.com/2018/03/sparse-matrices-in-python-with-scipy/

word occurs only a handful of times, it's not particularly useful to us for classification since (i) the presence of the word doesn't generalize to many documents and (ii) including the word increases the number of parameters we have to learn (without likely helping performance). As a result, most text classification systems will impose some *minimum word frequency* for including the word in the vocabulary.

In your solution, you should use a minimum word frequency of 10; every word occurring fewer than 10 times will *not* be included in the vocabulary (and therefore doesn't get a dimension in the β vector.

Import Caveat: In building the matrix, we'll *eventually* need to account for the bias term in our `numpy` implementation. Frequently, practitioners will simply add a single column to the term-document matrix that always has the value 1, which will get multiplied by the bias coefficient. However, the `pytorch` library will handle the bias term for us (yay!) so we don't always need to add t this column. You'll want to design your matrix-building code so that you can correctly account for the bias term.

One hint is that in building the matrix, you might find the `set`, `defaultdict` and `Counter` classes in python to come in handy.

6 Part 2: Logistic Regression in `numpy`

In Part 2, you'll implement logistic regression, which you might recall is

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

Note that our implementation of Logistic Regression is restricted to two classes, which we represent as *binary* so that y is either 0 or 1.[7] Conveniently, your training data is already set up for binary classification.

Your implementation will be one of the simplest possible formulations of logistic regression where you use stochastic 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 the homework setup is to give you a sense of how to compute a gradient.

In Task 1, you'll implement each step of the Logistic Regression 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.

- 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

[7]Important note: There *is* multiclass Logistic Regression, which the SLP3 textbook goes into detail about. This multiclass setup is more complicated (though also more useful) so we are not implementing it in the homework.

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

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

  where β is the vector of all of our coefficients. It takes some steps to reach this formula, the detail of how this formula is derived will be discussed in the discussion section. Feel free to skip the tedious math derivation and just use the formula above to implement! 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 if we were doing Gradient Descent, we could compute the whole gradient across the whole dataset 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. However, we're going to compute the gradient of the likelihood one instance at a time, so you can calculate the gradient for how to update β as

$$\frac{\delta}{\delta\beta} ll(\beta) = (\sigma(\beta x_i) - y_i)x_i$$

  This equation tells us how close our binary prediction $\sigma(\beta x_i)$ is to the ground truth $(y_i - \sigma(\beta x_i))$ and then uses the loss to scale the feature values of the current instance x_i to see how to update β. In essence, this update tell us how to update β so that it learns which features to weight more heavily and how to weight them (or conversely, which features to ignore!). When doing learning we'll scale this gradient by the learning rate α to determine how much to update β at each step as

$$\beta_{new} = \beta_{old} - \alpha \frac{\delta}{\delta\beta} ll(\beta)$$

  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?

Important Tip: The SLP3 book also has a very nice walk-through of logistic regression and stochastic gradient descent in Chapter 5, Section 5.4, which includes a detailed example of the update steps in 5.4.3. If anything in the homework description is confusing, please feel free to reach out to us on Piazza and/or check out the SLP3 material![8]

- Putting it all together, write a function `logistic_regression` that takes in a
  - a matrix X where each row is a vector that 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, $\hat{y}_i$, for each row i of X and then using those predictions to calculate the gradient. You should also include an *intercept* coefficient.[9]

  Note that you can make your life easier by using matrix operations. For example, to compute $\hat{y}_i$, multiply the row of 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!

- Write a function `predict` that given some new text, converts it to a vector (i.e., something like a row from X), and then uses the β vector to predict the class and returns the class label.

- To see how your model is learning, first, train your model on the training data `learning_rate=5e-5` (i.e., a very small number) and just use `num_steps = 1000`. **Note:** This means training only on 1000 emails; if your function takes a while, you might have implemented full Gradient Descent, which isn't asked for. Make a plot of the log-likelihood for the full data every 100 steps for this initial pass. Did the model converge at some point (i.e., does the log likelihood remain stable)?

- Now that you have some sense of how your hyperparameters work, train the model until it converges. This will likely take several epochs, depending on your learning rate. You can stop when the log-likelihood does not change too much between steps. Typically a very small number is used (e.g., 1e-5), though you are welcome to define your own.

- 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 score.

- Submit your best model's predictions on the test data to the KaggleInClass competition for `numpy` Logistic Regression.

A few more hints:

[8] We also want to add that despite the fancy-math, your actual implementation will mostly consist of very common operations!

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

- 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 Part 3: Logistic Regression with PyTorch

You've come through a long journey when you are finally at this exciting PyTorch part! `PyTorch` is a very popular deep learning framework in both academic and industrial fields *and* is the library we'll use for this and all future NLP assignments. With this framework, one can build deep learning models with flexible architectures in a clean and concise manner. It may be a little bit challenging at the beginning, but once you've got the idea, you will be amazed to find PyTorch is so powerful that can enable you to build complex models within a few lines of code. The goal of Parth 3 is to expose you to the "workflow" of how PyTorch trains models (since we'll be using this framework throughout the whole course) while also giving you some reference comparison point in your `numpy` implementation

In Part 3, you will implement the logistic regression model with `PyTorch`. You'll compare your PyTorch version with your `numpy` version and see if one performs better than the other.

7.1 Setting up the Data

PyTorch works with tensors.[10] A tensor is a fancy math term for as a matrix with an arbitrary number of dimensions (e.g., a vector is a tensor with 1 dimension!), which is very similar to arrays in `numpy`. *Conveniently*, `pytorch` also provides sparse tensor abilities,[11] which we'll need to use to represent our term document ~~matrix~~ tensor (it's still a matrix though!).

- Write a `to_sparse_tensor` function that takes the sparse `numpy` (SciPy) representation of the term-document matrix you used earlier and converts it into sparse Tensor objects. You will use the transformed data in tensor format for training the model in the later part.

7.2 Building the Logistic Regression Neural Network

Logistic Regression can be considered a shallow neural network ($|\mathcal{V}|$ input neurons and 1 output neuron), so we'll use PyTorch's functionality for building neural networks to help you get started for the semester. PyTorch has a special way to define neural networks so that we can use its very convenient functionality for training them. To create a new newtork, you'll extend the `nn.Module` class,[12] which is the base class for all networks. Extending this class lets us hook into PyTorch's deep learning training. In particular, the class will keep track of which parameters will need updating during gradient descent if they are assigned as a field to the class—this means

[10] `https://en.wikipedia.org/wiki/Tensor`

[11] `https://pytorch.org/docs/stable/sparse.html`

[12] If you haven't seen class extension before, there are many good examples (like this one online; the code for your subclass is not complex, so you don't need to understand it too well, provided you get the syntax right.

we won't have to calculate the loss like we did before *or* do the updating for stochastic gradient descent.

The `nn.Module` class defines the `forward()` function which is the most important function for us that we need to override. The forward function goes from the inputs to the outputs (i.e., from our bag of words vector to the predicted party). We'll specify the inputs and then use the network layers in the method to produce our outputs. Important note: Since neural networks are non-linear functions in the mathematical sense, **PyTorch has overloaded the function operator `()` so that calling a network will call forward().** For example with the code `model.forward(inputs)` and `model(inputs)` are equivalent! Note that this convention will apply to all network layers too; these layers take inputs and produce outputs, so we can call them like functions.

You might also guess that there's a backward() function too—and you're right! That's the step that goes from the outputs back to the inputs and updates the weights along the way. However, we don't have to do anything here. That's the magic: PyTorch handles the backwards updates for us and will update the β weights in our Logistic Regression model automatically as long as we call the training loop code correctly.

- In this part, you'll write a `LogisticRegression` class which is an extension of `nn.Module`. Think about: What's the input and the output? What's the shape of weights need to be trained? What kind of layer(s) do we have in logistic regression?

7.3 Setting up the Training

After your model implementation, the next step is to train the network. In particular, you need to define the loss function and optimizer and then use your prepared data to train the model. These are specific kinds of objects for pytorch but we've already seen examples of these concepts. The *loss function* tells pytorch how to measure how close our predictions are to the correct outputs. In our setting, we have a *binary* prediction task, so we'll use Binary Cross Entropy Loss or `BCELoss`[13]. You actually already implemented this in `numpy`! Pytorch has many other loss functions that make it easy to switch to multiclass classification (1 of n outputs is true) or even multilabel classification (k of n outputs is true).

An *optimizer* object tells pytorch how to update the model's parameters based on how wrong the model's predictions were. We've already seen one such optimizer: Stochastic Gradient Descent! In our case, the SGD update rules are quite simple for updating our β parameters—so simple that we could easily write them in `numpy`; however, for larger networks with multiple layers, the update process becomes very complicated—just think of the two layer network example from Week 2's lecture! Part of the power of PyTorch comes from the optimizer being able to *automatically* figure out how to update the parameters without us having to specify the code to do so. Here, we'll use the `SGD`[14] optimizer to start with. However, there are many other optimizers to start with and you'll try a few later in the assignment. All of the optimizers need to know which parameters they're updating so as a part of instantiating these objects, you'll need to pass in your model's parameters as an argument, which is why we need to extend the `nn.Module` so that all of this process works.

[13]`https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html#torch.nn.BCELoss`

[14]https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

The training process works in two steps. First, you'll run the *forward* pass of your code where you provide some input and the model makes a prediction. Then you'll start the *backwards* part of the process to update the parameters to make better predictions. This is the *backpropagation* algorithm[15] that calculates the updates based on how each weight in the network contributed to the ultimate prediction. The loss function object will calculate the loss and subsequent gradient of change for us. Then, the optimizer will update the parameters with respect to that gradient. There are many, *many* tutorials on how to do this training process, since every pytorch network does the same series of operations regardless of what the network is being trained to do. The pytorch documentation has one good example training process for a vision classifier and you can find others like this one that describe similar steps.

PyTorch will automatically keep track of the gradients over multiple steps as long as we tell it to. When it's time for evaluating, rather than training, we need to tell the model to stop remembering those gradients. The `pytorch Module` class has `train()` and `eval()` functions that

- Instantiate a `BCELoss` object that you'll use as your loss function
- Instantiate a `SGD` object as your optimizer.
- Write your training loop that iterates over the data set for the specified number of epochs and then for each step in the epoch, randomly samples one instance (row from the term-document tensor) and gets a prediction from `pytorch LogisticRegression` network. This process will look very similar to how you did your core training loop in `numpy`, only you'll add in the `pytorch` specific bits. Use the loss function to measure the loss and then the optimizer to perform the backpropagation step (using the `backwards()` and `step()` functions.
- Write some code that during the training process, will evaluate the current model on the development data. You'll want to only do this evaluation periodically (not every step) and the prediction code will be the same for the test data (though without the score). Remember the `eval()` call!

7.4 Training and Experiments

Now we've gotten the whole network and training procedure defined so it's time to start training. Since training in `pytorch` should be a bit faster and we want you to explore the code a bit, you'll do a few experiments in `pytorch` to see their effect.

For most of these mini-experiments, you'll be doing the same type of operation: varying one hyperparameter and then plotting the performance of the model during certain training steps. We recommend trying to refactor your code in such a way that it is easy to periodically call the evaluation step to get the model's loss and its performance on the development data. For plotting, we highly recommend plotting with seaborn, though any plots will do. While we recognize that some of these tasks can look repetitive, they are intentionally designed to give you a deeper intuition for how design and hyperparameter choices can ultimately impact performance. The trends you see

[15] `https://en.wikipedia.org/wiki/Backpropagation`

here may vary on future models and data, but the intuition and ability to quickly test the impact of these choices will come in handy in future homeworks and beyond as a practitioner.

- Like you did for the `numpy` code, train your model for a total of 1000 steps (i.e., only showing it 1000 randomly sampled documents) and report the loss after each 100 steps. This should verify that the loss is going down.

- Once you're satisfied that the model is working, train your model for at least 5 epochs and compute (and save) both (1) the loss every 1000 steps and (2) the F1 score of the model on the development data.

- Let's see what are the effects of adding regularization. PyTorch builds in regularization through the `weight_decay` argument to most `Optimizer` classes (including the one you use, `SGD`. Let's see what the effect is for setting the L2 penalty to 0 (default), 0.001, and 0.1. For each L2 penalty value, train the model for 1 epoch total and plot the loss and F1 score on the development set every 1000 steps using one line for each L2 value (use separate plots for F1 vs. Loss). In a few sentences, describe what you see: What effect does L2 have on the convergence speed and overall model performance?

- PyTorch has more than just the SGD optimizer. Recall that SGD takes a step with respect to the gradient using the learning rate to scale how big of a step to take. But what if we used more than just the gradient? For example, could we keep a bit of momentum to keep our gradient heading in the same direction? Many new optimizers have been proposed and PyTorch keeps implementations of the more successful ones. The big benefit of a better optimizer is that it helps the model learn faster. Since some of our big models may take hours to converge, having the optimizer reduce the training time to under an hour can be a huge time and environmental benefit. For this step, replace your `SGD` optimizer with two other common alternatives `RMSprop` and `AdamW`. For each optimizer, train the model for 1 epoch total and plot the loss and F1 score on the development set every 1000 steps using one line per optimizer (use separate plots for F1 vs. Loss). In a few sentences, describe what you see: What effect does the choice in optimizer have on the convergence speed and overall model performance?

- We had two methods for tokenizing. Does one method perform better in practice here? Train your model in the basic setting (using SGD, no L2 loss) for 1 epoch and plot the loss and F1 score on the development set every 1000 steps using one line per optimizer (use separate plots for F1 vs. Loss). In a few sentences, describe what you see: what effect does tokenization have on the overall model performance?

- What effect does the learning rate have on our model's convergence? Using the basic setup with SGD, change the `lr` argument (i.e., learning rate) to a much larger and much smaller value and for 1 epoch and plot the loss and F1 score on the development set every 1000 steps using one line per `lr` value. Plot all three curves together and describe what you observe: what effect does the learning rate have on the model's convergence speed? You're welcome (encouraged, even!) to try additional learning rates. If your model converges quickly, you can also reduce the number of steps between evaluation for this question.

- Finally, use your best model to generate prediction results and report the final F1 score. Don't forget to upload your prediction results on Kaggle. Note that this is a separate competition so that you can compare your scores with the `numpy` version.

7.5 Optional Part 4

Parts 1 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.

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.

8 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 all parts. Could should be uploaded as separate files, *not* in a .zip or other archive.

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**.

9 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.

References

Arunesh Mathur, Angelina Wang, Carsten Schwemmer, Maia Hamin, Brandon M. Stewart, and Arvind Narayanan. 2020. Manipulative tactics are the norm in political emails: Evidence from 100k emails from the 2020 u.s. election cycle. https://electionemails2020.org.