CS 475 Machine Learning: Programming Homework
Supervised Classifiers 1

Due: Thursday October 1, 2020, 11:59pm

35 Points Total Version 1.1

Make sure to read from start to finish before beginning the assignment. This assignment must be completed individually. No partners allowed.

1 Introduction

In this assignment you will implement two classifiers: perceptron and logistic regression. We have provided Python code to serve as a testbed for your algorithms. We will reuse this testbed in future assignments as well. You will fill in the details. Search for comments that begin with `TODO`; these sections need to be written. You may change the internal code as you see fit but **do not change the names of any of the files or command-line arguments that have already been provided.** Other than the given filenames and command-line arguments, you are free to change what you wish: you can modify internal code, add internal code, add files, and/or add new command-line arguments (in which case you should include appropriate defaults as we won't add these arguments when we run your code). You must submit all the code, not just the files you modified.

2 Data

The first part of the semester will focus on supervised classification. We consider a real world binary classification dataset applied to *finance*. The test file contains unlabeled examples that we will use to test your algorithm. It is **a very good idea** to run on the test data just to make sure your code doesn't crash. You'd be surprised how often this happens.

2.1 Finance

Finance is a data rich field that employs numerous statistical methods for modeling and prediction, including the modeling of financial systems and portfolios.[1]

Our financial task is to predict which Australian credit card applications should be accepted (label 1) or rejected (label 0). Each example represents a credit card application, where all values and attributes have been anonymized for confidentiality. Features are a mix of continuous and discrete attributes and discrete attributes have been binarized.

[1] For an overview of such applications, see the proceedings of the 2005 NIPS workshop on machine learning in finance. `http://www.icsi.berkeley.edu/~moody/MLFinance2005.htm`

2.2 Python Libraries

We will be using Python 3.7.6. We are *not* using Python 2, and will not accept assignments written for this version. We recommend using a recent release of Python 3.7.x, but anything in this line (e.g. 3.x) should be fine.

For each assignment, we will tell you which Python libraries you may use. We will do this by providing a `requirements.txt` file. We *strongly* recommend using a *virtual environment* to ensure compliance with the permitted libraries. By strongly, we mean that unless you have a very good reason not to, and you really know what you are doing, you should use a virtual environment. You can also use anaconda environments, which achieve the same goal. The point is that you should ensure you are only using libraries available in the basic Python library or the requirements.txt file.

2.3 Virtual Environments

Virtual environments are easy to set up and let you work within an isolated Python environment. In short, you can create a directory that corresponds to a specific Python version with specific packages, and once you activate that environment, you are shielded from the various Python / package versions that may already reside elsewhere on your system. Here is an example:

```
# Create a new virtual environment.
python3 -m venv python3-hw2
# Activate the virtual environment.
source python3-hw2/bin/activate
# Install packages as specified in requirements.txt.
pip3 install -r requirements.txt
# Optional: Deactivate the virtual environment, returning to your system's setup.
deactivate
```

When we run your code, we will use a virtual environment with *only* the libraries in `requirements.txt`. If you use a library not included in this file, your code will fail when we run it, leading to a large loss of points. By setting up a virtual environment, you can ensure that you do not mistakenly include other libraries, which will in turn help ensure that your code runs on our systems.

Make sure you are using the correct `requirements.txt` file from the current assignment. We may add new libraries to the file in subsequent assignments, or even remove a library (less likely). If you are using the wrong assignments `requirements.txt` file, it may not run when we grade it. For this reason, we suggest creating a new virtual environment for each assignment.

It may happen that you find yourself in need of a library not included in the `requirements.txt` for the assignment. You may request that a library be added by posting to Piazza. This may be useful when there is some helpful functionality in another library that we omitted. However, we are unlikely to include a library if it either solves a major part of the assignment, or includes functionality that isn't really necessary.

In this and future assignments we will allow you to use `numpy` and `scipy`.

2.4 How to Run the Provided Framework

The framework operates in two modes: train and test. Both stages are in the main method of `classify.py`.

2.4.1 Train Mode

The usage for train mode is

```
python3 classify.py --mode train --algorithm algorithm_name --model-file model_file --data train_file
```

The `mode` option indicates which mode to run (train or test). The `algorithm` option indicates which training algorithm to use. Each assignment will specify the string argument for an algorithm. The `data` option indicates the data file to load. Finally, the `model-file` option specifies where to save the trained model.

2.4.2 Test Mode

The test mode is run in a similar manner:

```
python3 classify.py --mode test --model-file model_file --data test_file --predictions-file predictions_file
```

The `model_file` is loaded and run on the `data`. Results are saved to the `predictions_file`.

2.4.3 Examples

As an example, the following trains a perceptron classifier on the speech training data:

```
python3 classify.py --mode train --algorithm perceptron --model-file finance.perceptron.model \
                    --data finance.train
```

To run the trained model on development data:

```
python3 classify.py --mode test --model-file finance.perceptron.model --data finance.dev \
                    --predictions-file finance.dev.predictions
```

As we add new algorithms we will also add command line flags using the `argparse` library to specify algorithmic parameters. These will be specified in each assignment.

2.5 Data Formats

The data are provided in what is known as SVM-light format. Each line contains a single example:

```
0 1:-0.2970 2:0.2092 5:0.3348 9:0.3892 25:0.7532 78:0.7280
```

The first entry on the line is the label. The label can be an integer (0/1 for binary classification) or a real valued number (for regression.) The classification label of -1 indicates unlabeled. Subsequent entries on the line are features. The entry `25:0.7532` means that feature 25 has value 0.7532. Features are 1-indexed.

Model predictions are saved as one predicted label per line in the same order as the input data. The code that generates these predictions is provided in the library. The script `compute_accuracy.py` can be used to evaluate the accuracy of your predictions for classification:

```
python3 compute_accuracy.py data_file predictions_file
```

We provide this script since it is exactly how we will evaluate your output. Make sure that your algorithm is outputting labels as they appear in the input files. If you use a different internal representation of your labels, make sure the output matches what's in the data files. The above script will do this for you, as you'll get low accuracy if you write the wrong labels.

2.6 Existing Components

The foundations of the learning framework have been provided for you. You will need to complete this library by filling in code where you see a `TODO` comment. You are free to make changes to the code as needed provided you do not change the behavior of the command lines described above. We emphasize this point: **do not change the existing command line flags, existing filenames, or algorithm names.** We use these command lines to test your code. If you change their behavior, we cannot test your code.

The code we have provided is fairly compact, and you should spend some time to familiarize yourself with it. Here is a short summary to get you started:

- `data.py` – This contains the `load_data` function, which parses a given data file and returns features and labels. The features are stored as a sparse matrix of floats (and in particular as a `scipy.sparse.csr_matrix` of floats), which has `num_examples` rows and `num_features` columns. The labels are stored as a dense 1-D array of integers with `num_examples` elements.

- `classify.py` – This is the main testbed to be run from the command line. It takes care of parsing command line arguments, entering train/test mode, saving models/predictions, etc. Once again, **do not change the names of existing command-line arguments**.

- `models.py` – This contains a `Model` class which you should extend. Models have (in the very least) a `fit` method, for fitting the model to data, and a `predict` method, which computes predictions from features. You are free to add other methods as necessary. **Note that all predictions from your model must be 0 or 1; if you use other intermediate values for internal computations, then they must be converted before they are returned.**

- `compute_accuracy.py` – This is a script which simply compares the true labels from a data file (e.g., `finance.dev`) to the predictions that were saved by running `classify.py` (e.g., `finance.dev.perceptron.predictions`).

We have also included a toy model, which we call `Useless`, for your reference. This is a binary classifier which forms predictions in a very simple and naive way. At training time, it simply memorizes the first example it sees (both its features and label). At prediction time, it forms the dot product between all of the examples and the memorized training example, and forms a prediction based on this dot product. (If the dot product is greater than 0, then it returns the same label as the memorized example; otherwise, the opposite label is returned.) Do not think too much about this toy model – it's simply included as a reference for how the `fit` and `predict` methods can be defined. In fact you can go through the whole training/testing process with this toy model on the provided datasets, and we recommend that you do so.

2.7 Perceptron (15 points)

You will implement the Perceptron algorithm for binary classification. The perceptron is a mistake-driven online learning algorithm. It takes as input a vector of real-valued inputs $\mathbf{x}$ and makes a prediction $\widehat{y} \in \{-1, +1\}$ (for this assignment we consider only binary labels). **This simplifies computations internally, but we again emphasize that your model's `predict` method must return labels that are 0 or 1.** Predictions

are made using a generalized linear classifier: $\widehat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$. The term $\mathbf{w} \cdot \mathbf{x}$ is the dot product of $\mathbf{w}$ and $\mathbf{x}$ computed as $\sum_i x_i w_i$. Updates to $\mathbf{w}$ are made only when a prediction is incorrect: $\widehat{y} \neq y$. The new weight vector $\mathbf{w}'$ is a function of the current weight vector $\mathbf{w}$ and example $\mathbf{x}$, y. The weight vector is updated so as to improve the prediction on the current example. Note that Perceptron naturally handles continuous and binary features, so no special processing is needed.

The basic structure of the algorithm is:

1. Initialize $\mathbf{w}$ to $\mathbf{0}$, set learning rate η and number of iterations I
2. For each training iteration $k = 1 \dots I$:
   (a) For each example $i = 1 \dots N$:
      i. Receive an example $\mathbf{x}_i$
      ii. Predict the label $\widehat{y_i} = \text{sign}(\mathbf{w} \cdot \mathbf{x}_i) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x}_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$
      iii. If $\widehat{y_i} \neq y_i$, make an update to $\mathbf{w}$: $\mathbf{w}' = \mathbf{w} + \eta y_i \mathbf{x}_i$

Note that there is no bias term in this version and you should *not* include one in your solution. Also observe the definition of "sign" to account for 0 values. Once again, while sign returns -1 and 1, the outputs from your `predict` method must be the actual labels, which are 0 or 1.

2.8 Logistic Regression (20 points)

The logistic regression model is used to model binary classification data. Logistic regression is a special case of generalized linear regression where the labels Y are modeled as a linear combination of the data X, but in a transformed space specified by g, sometimes called the "link function":

$$E[y \mid \mathbf{x}] = g(\mathbf{w}\mathbf{x} + \epsilon) \tag{1}$$

where ϵ is a noise term, usually taken to be Gaussian.

This "link function" allows you to model inherently non-linear data with a linear model. In the case of logistic regression, the link function is the logistic function[2]

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

2.8.1 Gradient Descent

In this assignment, we will solve for the parameters $\mathbf{w}$ in our logistic regression model using gradient descent to find the maximum likelihood estimate.

Gradient descent (GD) is an optimization technique that is both very simple and powerful. It works by taking the gradient of the objective function and taking steps (the size of the *learning rate*) in directions where the gradient is negative, which decreases the objective function[3].

[2] If you encounter an `exp` overflow error, this is because your input to the sigmoid function is negative. You may wish to use `scipy.special.expit`, which is a more stable implementation of `exp` in `scipy`.

[3] Gradient descent decreases the objective function if the gradient (first order approximation) is locally accurate, see Taylor expansion.

2.8.2 Maximum Conditional Likelihood

Since we seek to maximize the objective, we will use gradient *ascent*. We begin by writing the conditional likelihood:

$$P(Y \mid \mathbf{w}, X) = \prod_{i=1}^{n} p(y_i \mid \mathbf{w}, \mathbf{x}_i) \tag{3}$$

Since $y_i \in \{0, 1\}$, we can write the conditional probability inside the product as:

$$P(Y \mid \mathbf{w}, X) = \prod_{i=1}^{n} p(y_i = 1 \mid \mathbf{w}, \mathbf{x}_i)^{y_i} \times (p(y_i = 0 \mid \mathbf{w}, \mathbf{x}_i))^{1-y_i} \tag{4}$$

Note that one of these terms in the product will have an exponent of 0, and will evaluate to 1.

For ease of math and computation, we will take the log:

$$\ell(Y, X, \mathbf{w}) = \log P(Y \mid \mathbf{w}, X) = \sum_{i=1}^{n} y_i \log(p(y_i = 1 \mid \mathbf{w}, \mathbf{x}_i)) + (1 - y_i) \log(p(y_i = 0 \mid \mathbf{w}, \mathbf{x}_i)) \tag{5}$$

Plug in our logistic function for the probability that y is 1:

$$\ell(Y, X, \mathbf{w}) = \sum_{i=1}^{n} y_i \ \log(g(\mathbf{w}\mathbf{x}_i)) + (1 - y_i) \ \log(1 - g(\mathbf{w}\mathbf{x}_i))) \tag{6}$$

Recall that the link function, g, is the logistic function. It has the nice property $1 - g(z) = g(-z)$.

$$\ell(Y, X, \mathbf{w}) = \sum_{i=1}^{n} y_i \ \log(g(\mathbf{w}\mathbf{x}_i)) + (1 - y_i) \ \log(g(-\mathbf{w}\mathbf{x}_i)) \tag{7}$$

We can now use the chain rule to take the gradient with respect to $\mathbf{w}$:

$$\nabla\ell(Y, X, \mathbf{w}) = \sum_{i=1}^{n} y_i \ \frac{1}{g(\mathbf{w}\mathbf{x}_i)} \nabla g(\mathbf{w}\mathbf{x}_i) + (1 - y_i) \ \frac{1}{g(-\mathbf{w}\mathbf{x}_i)} \nabla g(-\mathbf{w}\mathbf{x}_i) \tag{8}$$

Since $\frac{\partial}{\partial z} g(z) = g(z)(1 - g(z))$:

$$\begin{aligned} \nabla\ell(Y, X, \mathbf{w}) \ &= \ \sum_{i=1}^{n} y_i \ \frac{1}{g(\mathbf{w}\mathbf{x}_i)} g(\mathbf{w}\mathbf{x}_i)(1 - g(\mathbf{w}\mathbf{x}_i)) \nabla \mathbf{w}\mathbf{x}_i & (9) \\ &\quad + (1 - y_i) \ \frac{1}{g(-\mathbf{w}\mathbf{x}_i)} g(-\mathbf{w}\mathbf{x}_i)(1 - g(-\mathbf{w}\mathbf{x}_i)) \nabla(-\mathbf{w}\mathbf{x}_i) & (10) \end{aligned}$$

Simplify again using $1 - g(z) = g(-z)$ and cancel terms

$$\nabla\ell(Y, X, \mathbf{w}) = \sum_{i=1}^{n} y_i g(-\mathbf{w}\mathbf{x}_i) \nabla \mathbf{w}\mathbf{x}_i + (1 - y_i) g(\mathbf{w}\mathbf{x}_i) \nabla(-\mathbf{w}\mathbf{x}_i) \tag{11}$$

You can now get the partial derivatives (components of the gradient) out of this gradient function by:

$$\frac{\partial}{\partial \mathbf{w}_j}\ell(Y, X, \mathbf{w}) = \sum_{i=1}^{n} y_i g(-\mathbf{w}\mathbf{x}_i)\ \mathbf{x}_{ij} + (1 - y_i)g(\mathbf{w}\mathbf{x}_i)(-\mathbf{x}_{ij}) \tag{12}$$

This is the equation that you will use to calculate your updates. (If you'd like to, you can optimize this further, but only if you desire. Here the main computational gain comes from using this vectorized form; it is easily exploited using NumPy and will be much faster than manually looping through the elements of $\mathbf{w}$ to be updated. For example, see Python for Data and StackExchange.)

2.8.3 Offset Feature

None of the math above mentions an offset feature (bias feature), a $\mathbf{w}_0$, that corresponds to a $x_{i,0}$ that is always 1. It turns out that we don't need this if our data is centered. By centered we mean that $E[y] = 0$. While this may or may not be true, for this assignment you should assume that your data is centered. Do not include another feature that is always 1 (x_0) or weight ($\mathbf{w}_0$) for it.

2.9 Deliverables

You need to implement the Perceptron and Logistic Regression. To do this, see the `TODO` comments in the code, and feel free to modify / add code as necessary. However, once again, **do not modify the filenames or command-line arguments that have already been provided.** Your predictors will be selected by passing the string `perceptron` or `logistic` as the argument for the `algorithm` parameter.

2.10 Learning rate

In both algorithms, you will need to specify a learning rate η, where $0 < \eta \leq 1$. Your default value for η should be 1. You *must* add a command line argument to allow this value to be adjusted via the command line.

Add this command line option by adding the following code to the `get_args` function in `classify.py`.

```
parser.add_argument("--online-learning-rate", type=float, help="The learning rate for gradient based updates",
                    default=1.0)
```

Be sure to add the option name exactly as it appears above. You can then use the value read from the command line in your main method by referencing it as `args.online_learning_rate`. Note that the dashes have been replaced by underscores.

2.11 Number of training iterations

Usually we iterate multiple times over the data. This can improve performance by increasing the number of updates made. We will define the number of times each algorithm iterates over all of the data using the parameter `online_training_iterations`. You *must* define a new command line option for this parameter. Use a default value of 5 for this parameter.

You can add this option by adding the following code to the `get_args` function of `classify.py`.

```
parser.add_argument("--online-training-iterations", type=int,
                    help="The number of training iterations for online methods.", default=5)
```

You can then use the value read from the command line in your main method by referencing it as `args.online_training_iterations`.

During training, you should not change the order of examples. You must iterate over examples exactly as they appear in the data file, i.e. as provided by the data loader.

2.12 Grading Programming

The programming section of your assignment will be graded using an autograder in Gradescope. Your code will be run using the provided command line options, as well as other variations on these options (different parameters, data sets, etc.) The grader will consider the following aspects of your code.

1. **Exceptions:** Does your code run without crashing?
2. **Output:** Some assignments will ask you to write some data to the console. Make sure you follow the provided output instructions exactly.
3. **Accuracy:** If your code works correctly, then it should achieve a certain accuracy on each data set. While there are small difference that can arise, a correctly working implementation will get the right answer.
4. **Speed/Memory:** As far as grading is concerned, efficiency largely doesn't matter, except where lack of efficiency severely slows your code (so slow that we assume it is broken) or the lack of efficiency demonstrates a lack of understanding of the algorithm. For example, if your code runs in two minutes and everyone else runs in 2 seconds, you'll lose points. Alternatively, if you require 2 gigs of memory, and everyone else needs 10 MB, you'll lose points. In general, this happens not because you did not optimize your code, but when you've implemented something incorrectly.

2.13 Code Readability and Style

In general, you will not be graded for code style. However, your code should be readable, which means minimal comments and clear naming / organization. If your code works perfectly then you will get full credit. However, if it does not we will look at your code to determine how to allocate partial credit. If we cannot read your code or understand it, then it is very difficult to assign partial credit. Therefore, it is in your own interests to make sure that your code is reasonably readable and clear.

2.14 Code Structure

Your code must support the command line options and the example commands listed in the assignment. Aside from this, you are free to change the internal structure of the code, write new classes, change methods, add exception handling, etc. However, once again, do not change the names of the files or command-line arguments that have been provided. We suggest you remember the need for clarity in your code organization.

2.15 Knowing Your Code Works

How do you know your code really works? That is a very difficult problem to solve. Here are a few tips:

1. Use Piazza. While **you cannot share code**, you can share results. We encourage you to post your results on dev data for your different algorithms. A common result will quickly emerge that you can measure against.

2. Output intermediate steps. Looking at final predictions that are wrong tells you little. Instead, print output as you go and check it to make sure it looks right. This can also be helpful when sharing information on Piazza.

3. Debug. Find a Python debugger that you like and use it. This can be very helpful.

2.16 Debugging

The most common question we receive is "how do I debug my code?" The truth is that machine learning algorithms are very hard to debug because the behavior of the algorithm is unknown. In these assignments, you won't know ahead of time what accuracy is expected for your algorithm on a dataset. This is the reality of machine learning development, though in this class you have the advantage of your classmates, who may post the output of their code to the bulletin board. While debugging machine learning code is therefore harder, the same principles of debugging apply. Write tests for different parts of your code to make sure it works as expected. Test it out on the easy datasets to verify it works and, when it doesn't, debug those datasets carefully. Work out on paper the correct answer and make sure your code matches. Don't be afraid of writing your own data for specific algorithms as needed to test out different methods. This process is part of learning machine learning algorithms and a reality of developing machine learning software.

3 What to Submit

You will need to create an account on gradescope.com and signup for this class. The course is `https://www.gradescope.com/courses/153788`.

Submit your code (.py files) to gradescope.com as a zip file. **Your code must be uploaded as code.zip with your code in the root directory**. By 'in the root directory,' we mean that the zip should contain `*.py` at the root (`./*.py`) and not in any sort of substructure (for example `hw1/*.py`). One simple way to achieve this is to zip using the command line, where you include files directly (e.g., `*.py`) rather than specifying a folder (e.g., `hw1`):

4 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: `https://piazza.com/class/kdfbbwz3hwb3an`.