CS 475 Machine Learning: Project 2
Supervised Classifiers 2

Due: Friday March 9, 2018, 11:59 pm*
100 Points Total Version 1.0

Make sure to read from start to finish before beginning the assignment.

1 Programming (50 points)

In this assignment you will write a logistic regression classifier. Your implementation will be very similar to the algorithm we covered in class. Your code needs to handle data with binary and continuous features and only binary labels (no multi-class). We will be using the same data and code framework as in Project 1, so you may refer to that assignment for details on the datasets and how to run the framework.

1.1 Logistic Regression

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:

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

1.2 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 in directions where the gradient is negative, which decreases the objective function[1].

*Late submission deadline: Sunday March 11, 2018, 11:59 pm

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

1.3 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.

1.4 Learning Rate

In gradient descent, the update rule is $\mathbf{w}' = \mathbf{w} + \eta\nabla\ell(Y, X, \mathbf{w})$. Choosing η intelligently is a non-trivial task, and there are many ways to choose it in the literature. In this assignment, we will use a constant η. By default it should be 0.01, but your code should allow the user to change it by setting a command line argument.

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 logistic regression", default=0.01)
```

1.5 Feature selection using information gain

Often times there are irrelevant features in a given representation. In some cases, including these features in learning yields worse results than if they were excluded. Feature selection is a general technique that selects only a subset of the features to use for learning.

There are many ways to do feature selection. In this assignment, you will use information gain (IG) as a feature selection criteria.

You will use a command line parameter `num-features-to-select` that takes an integer value. When the option is provided you will first run feature selection to select the given number of features and then train the appropriate classifier using only those features. For example, if you get the argument `num-features-to-select` with a value of 10, then you will first select only 10 features and then only use those 10 features in learning. Note that this may result in having instances with no features since all of them were removed. In this case, you will not be able to use the example for training. At test time, you will end up with a value at the threshold (e.g. $y = 0.5$.) See the prediction rule below for details on how to translate a prediction into a binary output value.

To select features, first compute IG for all the features in the training data. After you have IG values for all features, select the `num-features-to-select` features to use for training that have the highest IG. Note that there is no reason to compute IG in test mode. Features that are unknown to the classifier at test time should be ignored by the classifier (receive 0 weight.)

To handle continuous features, select the mean of the feature in the training data as the threshold for IG calculation. Note that thresholding and IG are only used for feature selection. After you select `num-features-to-select` features from the data, use the original value of the features to perform your learning.

If `num-features-to-select` option is not provided, you will use all the features.

Recall that entropy is defined as:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i),$$

conditional entropy:

$$H(Y|X) = -\sum_{i=1}^{m} \sum_{j=1}^{n} p(y_i, x_j) \log \frac{p(y_i, x_j)}{p(x_j)}$$

and information gain:

$$IG(Y|X) = H(Y) - H(Y|X)$$

Remember that $H(Y)$ is a constant when comparing $IG(Y|X)$ for different values of X so it should be dropped for efficiency.

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

```
parser.add_argument("--num-features-to-select", type=int,
help="The number of features to use for logistic regression", default=-1)
```

1.6 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.

1.7 Convergence

In real gradient descent, you must decide when you have converged. Ideally, a maximized function has a gradient value of 0, but due to issues related to your step size, random noise, and machine precision, your gradient will likely never be exactly zero. Usually people check that the L_p norm of your gradient is less than some δ, for some p. For the sake of simplicity and consistent results, we will not do this in this assignment. Instead, your program should take a parameter **gd-iterations** which is *exactly* how many iterations you should run (not an upper bound). An iteration is a single pass over every training example. The default of **gd-iterations** should be 20.

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

```
parser.add_argument("--gd-iterations", type=int,
help="The number of iterations of gradient descent to perform", default=20)
```

1.8 Implementation Notes

1. Even though logistic regression predicts a probability that the label is 1, the output of your program should be binary. Round your solution based on whether the probability is greater than or equal to 0.5:

   $\hat{y}_{new} = 1$ if $p(y = 1|\mathbf{w}, \mathbf{x}) = g(\mathbf{wx}) = \frac{1}{1+e^{-\mathbf{wx}}} \geq 0.5$
   $\hat{y}_{new} = 0$ otherwise

2. Initialize the parameters $\mathbf{w}$ to 0.

1.9 Grading Programming

The programming section of your assignment will be graded using an automated grading program. 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.

1.10 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.

1.11 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.

1.12 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. Check results on `easy` and `hard`. For the sum of features classifier, you can make sure it is computing the sums correctly with a small subset of one of the datasets with a small number of features. In the case of the perceptron, you should get close to 100% on `easy` and close to 50% on `hard`.

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

3. 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 the bulletin board.

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

1.13 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.

1.14 How Your Code Will Be Called

To train a model we will call:

```
python3 classify.py --mode train --algorithm logisticregression \
        --model-file speech.logisticregression.model \
        --data speech.train
```

There are some additional parameters which your program must support during training:

```
--online-learning-rate k   // sets update rule for gradient step, default = 0.01
--gd-iterations t  // sets the number of GD iterations, default = 20
--num-features-to-select // sets the number of features to select, uses
all the features by default. You will not calculate IG in the default case.
```

All of these parameters are *optional*. If they are not present, they should be set to their default values.

To make predictions using a model we will call:

```
python3 classify.py --mode test --algorithm logisticregression \
        --model-file speech.logisticregression.model \
        --data speech.test \
        --predictions-file speech.test.predictions
```

Remember that your output should be 0/1 valued, not real valued.

2 Analytical (50 points)

1) Decision Tree and Logistic Regression (10 points) Consider a binary classification task (label y) with four features (x):

x_1	x_2	x_3	x_4	y
0	1	1	-1	1
0	1	1	1	0
0	-1	1	1	1
0	-1	1	-1	0

(a) Can this function be learned using a decision tree? If so, provide such a tree (describe each node in the tree). If not, prove it.

(b) Can this function be learned using a logistic regression classifier? If yes, give some example parameter weights. If not, why not.

(c) For the models above where you can learn this function, the learned model may over-fit the data. Propose a solution for each model on how to avoid over-fitting.

2) Stochastic Gradient Descent (10 points) In the programming part of this assignment you implemented Gradient Descent. A stochastic variation of that method (Stochastic Gradient Descent) takes an estimate of the gradient based on a single sampled example, and takes a step based on that gradient. This process is repeated many times until convergence. To summarize:

1. Gradient descent: compute the gradient over all the training examples, take a gradient step, repeat until convergence.
2. Stochastic gradient descent: sample a single training example, compute the gradient over that training example, take a gradient step, repeat until convergence.

In the limit, will both of these algorithms converge to the same optimum or different optimum? Answer this question for both convex and non-convex functions. Prove your answers.

3) Kernel Trick (10 points) The kernel trick extends SVMs to learn nonlinear functions. However, an improper use of a kernel function can cause serious over-fitting. Consider the following kernels.

(a) Inverse Polynomial kernel: given $\|x\|_2 \leq 1$ and $\|x'\|_2 \leq 1$, we define $K(x, x') = 1/(d - x^\top x')$, where $d \geq 2$. Does increasing d make over-fitting more or less likely?

(b) Chi squared kernel: Let x_j denote the j-th entry of x. Given $x_j > 0$ and $x'_j > 0$ for all j, we define $K(x, x') = \exp\left(-\sigma \sum_j \frac{(x_j - x'_j)^2}{x_j + x'_j}\right)$, where $\sigma > 0$. Does increasing σ make over-fitting more or less likely?

We say K is a kernel function, if there exists some transformation $\phi : \mathbb{R}^m \to \mathbb{R}^{m'}$ such that $K(x_i, x_{i'}) = \langle \phi(x_i), \phi(x_{i'}) \rangle$.

(c) Let K_1 and K_2 be two kernel functions. Prove that $K(x_i, x_{i'}) = K_1(x_i, x_{i'}) + K_2(x_i, x_{i'})$ is also a kernel function.

4) Dual Perceptron (8 points)

(c) You train a Perceptron classifier in the primal form on an infinite stream of data. This stream of data is not-linearly separable. Will the Perceptron have a bounded number of prediction errors?

(c) Switch the primal Perceptron in the previous step to a dual Perceptron with a linear kernel. After observing T examples in the stream, will the two Perceptrons have learned the same prediction function?

(c) What computational issue will you encounter if you continue to run the dual Perceptron and allow T to approach ∞? Will this problem happen with the primal Percepton? Why or why not?

5) Convex Optimization (12 points) Jenny at Acme Inc. is working hard on her new machine learning algorithm. She starts by writing an objective function that captures her thoughts about the problem. However, after writing the program that optimizes the objective and getting poor results, she returns to the objective function in frustration. Turning to her colleague Matilda, who took CS 475 at Johns Hopkins, she asks for advice. "Have you checked that your function is convex?" asks Matilda. "How?" asks Jenny.

(a) Jenny's function can be written as $f(g(x))$, where $f(x)$ and $g(x)$ are convex, and $f(x)$ is non-decreasing. Prove that $f(g(x))$ is a convex function. (Hint: You may find it helpful to use the definition of convexity. Do not use gradient or Hessian, since f and g may not have them.)

(b) Jenny realizes that she made an error and that her function is instead $f(x) - g(x)$, where $f(x)$ and $g(x)$ are convex functions. Her objective may or may not be convex. Give examples of functions $f(x)$ and $g(x)$ whose difference is convex, and functions $\bar{f}(x)$ and $\bar{g}(x)$ whose difference is non-convex.

"I now know that my function is non-convex," Jenny says, "but why does that matter?"

(c) Why was Jenny getting poor results with a non-convex function?

(d) One approach for convex optimization is to iteratively compute a descent direction and take a step along that direction to have a new value of the parameters. The choice of a proper stepsize is not so trivial. In gradient descent algorithm, the stepsize is chosen such that it is proportional to the magnitude of the gradient at the current point. What might be the problem if we fix the stepsize to a constant regardless of the current gradient? Discuss when stepsize is too small or too large.

3 What to Submit

You will need to create an account on gradescope.com and signup for this class. The course is `https://gradescope.com/courses/10244`. Use entry code `974Z3W`. See this video for instructions on how to upload a homework assignment: `https://www.youtube.com/watch?v=KMPoby5g_nE`. In each assignment you will submit two things to gradescope.

1. **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 `hw2/*.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., `hw2`):

```
zip code.zip *.py
```

We will run your code using the exact command lines described earlier, so make sure it works ahead of time, and make sure that it doesn't crash when you run it on the test data. Remember to submit all of the source code, including what we have provided to you. We will include `requirements.txt` but nothing else.

2. **Submit your writeup to gradescope.com**. **Your writeup must be compiled from latex and uploaded as a PDF**. It should contain all of the answers to the analytical questions asked in the assignment. Make sure to include your name in the writeup PDF and to use the provided latex template for your answers.

4 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: `https://piazza.com/class/j70ixzhajea1rt`.