CS 6362 Machine Learning: Homework 3

Non-Linear Supervised Learning

Due: 9/25/2017 11:59 PM Central Time

100 Points Total Version 1.0

1 Programming (50 points)

In this assignment you will write a Kernel logistic regression classifier. This classifier needs only support binary prediction. The `algorithm` option for this classifier should be `kernel_logistic_regression`.

1.1 Logistic Regression

1.2 Review: 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":

$$P(y = 1|\mathbf{x}, \mathbf{w}) = g(\mathbf{w} \cdot \mathbf{x}) \tag{1}$$

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:

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

1.3 Kernel Logistic Regression

The above model relies on a linear function of weight parameters and instances: $\mathbf{w} \cdot \mathbf{x}$. However, some data can be highly non-linear, which is impossible to learn using a linear classifier. In class, we learned about the kernel trick, which uses the dual of a linear classifier for kernel learning. Kernels project data into high dimensional spaces in which a linear separator may exist. The decision boundary learned by the linear classifier is linear in the high dimensional space but non-linear in the original feature space.

A kernel is a function which maps the input data into a new space using a basis function and computes the inner product between two basis functions. A kernel function

is defined as:

$$K(x, x') = (\phi(x) \cdot \phi(x')^T)$$

In this assignment you will implement a dual version of the logistic regression model so that you can learn with a kernel. In class, we derived the updates for the dual SVM, and we learned that the linear function of weights $\mathbf{w}$ and instances $\mathbf{x}$ can be rewritten:

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{N} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})$$

This dual formulation also holds in logistic regression. Under the kernel logistic regression model, we have:

$$P(y = 1|\mathbf{x}, \alpha) = g(\sum_{i=1}^{N} \alpha_i K(\mathbf{x}_i, \mathbf{x})) \tag{3}$$

$$P(y = 0|\mathbf{x}, \alpha) = 1 - g(\sum_{i=1}^{N} \alpha_i K(\mathbf{x}_i, \mathbf{x})) \tag{4}$$

Note that the y_i terms from the SVM formula are not needed here. Instead of learning $\mathbf{w}$, you will learn α_i for each example $\mathbf{x}_i$. The next subsection describes the learning procedure you will use.

1.3.1 Training with Gradient Ascent

In this assignment, we will solve for the parameters α in our logistic regression model using gradient ascent to find the maximum likelihood estimate.

Gradient ascent is an optimization technique that is both very simple and powerful. This optimization algorithm works by taking the gradient of the objective function and taking steps in directions where the gradient is positive, which increases the objective function.[1] Under certain conditions, gradient ascent is guaranteed to asymptotically converge to a local maximum of the objective function (or a global maximum if the objective is convex). This strategy is often referred to as "hill climbing."

Given a likelihood function $\ell(Y, X, \alpha)$, we need to compute the gradient $\nabla\ell(Y, X, \alpha)$. The partial derivative with respect to the parameter α_k is:

$$\frac{\partial \ell}{\partial \alpha_k} = \sum_{i=1}^{N} y_i g\left(-\sum_{j=1}^{N} \alpha_j K(x_j, x_i)\right)(K(x_i, x_k)) + (1-y_i) g\left(\sum_{j=1}^{N} \alpha_j K(x_j, x_i)\right)(-K(x_i, x_k)) \tag{5}$$

[1] In machine learning, we can either maximize or minimize an objective. In this assignment, we are *maximizing* an objective function (likelihood) as part of maximum likelihood estimation. One might also want to *minimize* an objective function that we think of as a *loss* or *error*. In this case, the algorithm is called gradient **descent**. You can trivially convert a maximization problem into a minimization problem by negating the objective function, so the terms gradient descent and gradient ascent are sometimes used interchangeably.

You can see how this is derived in the appendix of this handout.

In gradient descent, the update rule is $\alpha'_j = \alpha_j + \eta \frac{\partial}{\partial \alpha_j}$ for all values of j (one per training instance), where η is called the *learning rate*. 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 scalar η. See section 1.4.3 for more details.

This update is repeated for a given number of training iterations (see 1.4.4).

1.3.2 Making Predictions

Given an instance $\mathbf{x}_i$, you can predict a label y_i as the y value with higher probability under the model. You should choose a prediction $\hat{y}_i$ according to:

$$\hat{y}_i = \arg\max_y P(y_i = y | \mathbf{x}_i, \alpha) \tag{6}$$

In other words, if $g(\sum_{j=1}^{N} \alpha_j K(\mathbf{x}_j, \mathbf{x}_i)) \geq 0.5$, you should predict y_i to be 1, otherwise you predict y_i to be 0.

1.3.3 Kernel Functions

You will implement three kernel functions for use with kernel logistic regression. Each version of the kernel logistic regression algorithm will be its own class, but it is highly recommended that most of the code be in a parent class: `KernelLogisticRegression` to avoid code duplication. Each subclass will overload and implement the necessary functions for their specific kernels.

1. Linear Kernel: $K(x, x') = (x \cdot x')$

   Implement this algorithm as the `LinearKernelLogisticRegression` class.

2. Polynomial Kernel: $K(x, x') = (1 + (x \cdot x'))^d$

   Implement this algorithm as the `PolynomialKernelLogisticRegression` class. The parameter d is explained in section 1.4.2.

3. Gaussian Kernel: $K(x, x') = \exp(-||x - x'||^2 / 2\sigma^2)$

   Implement this algorithm as the `GaussianKernelLogisticRegression` class. The parameter σ is explained in section 1.4.2.

Note that the naive implementation may compute $K(x_i, x_j)$ many times during training. To improve training efficiency you should store (cache) computed values of $K(x_i, x_j)$ in a matrix, $G_{ij} = K(x_i, x_j)$. G is called a Gram Matrix. If you do not do this, your code will run much slower than it should, and we may simply assume that it is broken (which may cause you to fail the programming part of the assignment).

You can improve efficiency further by caching

$$\sum_{j=1}^{N} \alpha_j K(x_j, x_i)$$

for each feature vector i in the data. We recommend this as well.

1.4 Implementation Details

1.4.1 Kernel

The choice of kernel will be controlled by a command line option. You need to add this option by adding the following code block to the `createCommandLineOptions` method of `Learn`.

```
registerOption("kernel", "String", true, "The kernel for
Kernel Logistic regression [linear_kernel, polynomial_kernel, gaussian_kernel].");
```

There are exactly three allowed values for this option: "linear_kernel", "polynomial_kernel", and "gaussian_kernel", with "linear_kernel" being default when no parameter is supplied.

You can then read the value from the command line by adding the following to the main method of `Learn`:

```
String kernel = "linear_kernel";
if (CommandLineUtilities.hasArg("kernel"))
   kernel = CommandLineUtilities.getOptionValue("kernel");
```

1.4.2 Kernel Parameters

The parameters d for the polynomial kernel and σ for the Gaussian kernel are set using command line options.

For the polynomial kernel parameter d, add this command line option by adding the following code block to the `createCommandLineOptions` method of `Learn`.

```
registerOption("polynomial_kernel_exponent", "double", true, "The exponent of the polynomial kernel.");
```

You can then read the value from the command line by adding the following to the main method of `Learn`:

```
double polynomial_kernel_exponent = 2;
if (CommandLineUtilities.hasArg("polynomial_kernel_exponent"))
   polynomial_kernel_exponent = CommandLineUtilities.getOptionValueAsFloat("polynomial_kernel_exponent");
```

For the Gaussian kernel parameter σ, add this command line option by adding the following code block to the `createCommandLineOptions` method of `Learn`.

```
registerOption("gaussian_kernel_sigma", "double", true, "The sigma of the Gaussian kernel.");
```

You can then read the value from the command line by adding the following to the main method of `Learn`:

```
double gaussian_kernel_sigma = 1;
if (CommandLineUtilities.hasArg("gaussian_kernel_sigma"))
   gaussian_kernel_sigma = CommandLineUtilities.getOptionValueAsFloat("gaussian_kernel_sigma");
```

The default value for d should be 2 and the default value for σ should be 1.

1.4.3 Learning Rate

Your default value for η should be 0.01. 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 `createCommandLineOptions` method of `Learn`.

```
registerOption("gradient_ascent_learning_rate", "double", true, "The learning rate for logistic regression.");
```

Be sure to add the option name exactly as it appears above. You can then read the value from the command line by adding the following to the main method of `Learn`:

```
double gradient_ascent_learning_rate = 0.01;
if (CommandLineUtilities.hasArg("gradient_ascent_learning_rate"))
    gradient_ascent_learning_rate =
        CommandLineUtilities.getOptionValueAsFloat("gradient_ascent_learning_rate");
```

1.4.4 Number of Training Iterations

We will define the number of times each algorithm iterates over all of the data by the parameter `gradient_ascent_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 `createCommandLineOptions` method of `Learn`.

```
registerOption("gradient_ascent_training_iterations", "int", true, "The number of training iterations.");
```

You can then read the value from the command line by adding the following to the main method of `Learn`:

```
int gradient_ascent_training_iterations = 5;
if (CommandLineUtilities.hasArg("gradient_ascent_training_iterations"))
    gradient_ascent_training_iterations =
         CommandLineUtilities.getOptionValueAsInt("gradient_ascent_training_iterations");
```

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.

1.4.5 Initialization

You should initialize all α values to 0.

1.4.6 Numeric Stability

For all numerical calculations involving floating point numbers, use the `double` type and NOT the `float` type to store values. This will help in achieving numerical precision.

1.5 Data Sets

We are providing a new synthetic data set for this assignment called `Circle`.

2 Analytical (50 points)

1) Curse-of-dimensionality (10 points) In this problem, we study why K-NN could fail in high dimensions by a very simple example. Consider a sphere of radius r in d-dimensions together with a concentric hypercube of side $2r$. The sphere touches the hypercube at the center of each of its sides.

(a) V_c is the volume of the cube and V_s is the volume of the sphere, where the volume of a d-dimensional sphere with radius r is given as

$$V_s = \frac{r^d \pi^{d/2}}{\Gamma(d/2+1)},$$

where $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} \mathrm{d}t$. Please show that:

$$\lim_{d\to\infty} \frac{V_s}{V_c} = 0 \tag{7}$$

Note that this relies on algebra and will not require any complex calculus. You may find the following limit useful:

$$\lim_{z\to\infty} \frac{\Gamma(z+1)}{\sqrt{2\pi z} e^{-z} z^z} = 1$$

(b) What is the connection between (7) and the curse of dimensionality?

2) Overfitting (10 points) SVMs using nonlinear kernels usually have two tuning parameters (regularization parameter C and kernel parameter γ), which are usually determined by cross validation. Suppose we use cross validation to determine the non-linear kernel parameter and slack variable for an SVM. We find that classifiers using parameters (c_1, γ_1) and (c_2, γ_2) achieve the same cross validation error, but (c_1, γ_1) leads to fewer support vectors than (c_2, γ_2). Explain which set of parameters should we choose for the final model?

3) Hinge Loss (10 points) Linear SVMs can be formulated in an unconstrained optimization problem

$$\min_{w,b} \sum_{i=1}^{n} H(y_i(w^T x_i)) + \lambda \|w\|_2^2, \tag{8}$$

where λ is the regularization parameter and $H(a) = \max(1-a, 0)$ is the well known hinge loss function. The hinge loss function can be viewed as a convex surrogate of the 0/1 loss function $I(a \leq 0)$.

(a) Prove that $H(a)$ is a convex function of a.

(b) If $H'(a) = \max(0.5 - a, 0)$, please show that there exists λ' such that (9) is equivalent to (8)

$$\min_{w,b} \sum_{i=1}^{n} H'(y_i(w^T x_i)) + \lambda' \|w\|_2^2. \tag{9}$$

4) Kernel Trick (10 points) The kernel trick extends SVMs to handle nonlinearly separable data sets. However, an improper use of a kernel function can cause serious over-fitting. Consider the following kernels.

(a) Polynomial kernel: $K(x, x') = (1 + (xx'^T))^d$, where $d \in \mathbb{N}$. Does increasing d make over-fitting more or less likely?

(b) Gaussian kernel: $K(x, x') = \exp(-||x - x'||^2/2\sigma^2)$, 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.

5) Prediction using Kernel (10 points) One of the differences between linear SVMs and kernel SVMs concerns computational complexity at prediction time.

(a) What is the computational complexity of prediction of a linear SVM in terms of the examples n and features m?

(b) What is the computational complexity of prediction of a non-linear SVM in terms of the examples n, features m and support vectors s?

3 What to Submit

In each assignment you will submit two things.

1. **Code:** Your code as a zip file named `hw3code.zip`. **You must submit source code (.java files)**. We will run your code using the exact command lines described above, so make sure it works ahead of time. Remember to submit all of the source code, including what we have provided to you.

2. **Writeup:** Your writeup as a **PDF file** (compiled from latex) containing answers to the analytical questions asked in the assignment. Make sure to include your name in the writeup PDF and use the provided latex template for your answers.

Make sure you name each of the files exactly as specified (`hw3code.zip` and `hw3solutions.pdf`).

4 Appendix: Deriving the Gradient for Kernel Logistic Regression

In this section, we show how the gradient $\nabla \ell(Y, X, \alpha)$ is derived. We begin by writing the conditional likelihood:

$$P(Y \mid \alpha, X) = \prod_{i=1}^{N} p(y_i \mid \alpha, \mathbf{x}_i) \tag{10}$$

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

$$P(Y \mid \alpha, X) = \prod_{i=1}^{N} p(y_i = 1 \mid \alpha, \mathbf{x}_i)^{y_i} \times (1 - p(y_i = 0 \mid \alpha, \mathbf{x}_i))^{1-y_i} \tag{11}$$

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:

$$\ell(Y, X, \alpha) = \log P(Y \mid \alpha, X) = \sum_{i=1}^{N} y_i \log(p_{y_i=1}(\alpha, \mathbf{x}_i)) + (1 - y_i) \log(p_{y_i=0}(\alpha, \mathbf{x}_i)) \tag{12}$$

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

$$\ell(Y, X, \alpha) = \sum_{i=1}^{N} y_i \log(g(\textstyle\sum_{j=1}^{N} \alpha_j K(\mathbf{x}_j, \mathbf{x}_i))) + (1 - y_i) \log(1 - g(\textstyle\sum_{j=1}^{N} \alpha_j K(\mathbf{x}_j, \mathbf{x}_i))) \tag{13}$$

To keep the notation clean, we will use the shorthand:

$$\kappa(\mathbf{x}_i) = \sum_{j=1}^{N} \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) \tag{14}$$

With this notation, we have:

$$\ell(Y, X, \alpha) = \sum_{i=1}^{N} y_i \log(g(\kappa(\mathbf{x}_i))) + (1 - y_i) \log(1 - g(\kappa(\mathbf{x}_i))) \tag{15}$$

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

$$\ell(Y, X, \alpha) = \sum_{i=1}^{N} y_i \ \log(g(\kappa(\mathbf{x}_i))) + (1 - y_i) \ \log(g(-\kappa(\mathbf{x}_i))) \tag{16}$$

We can now use the chain rule to take the gradient with respect to α:

$$\nabla \ell(Y, X, \alpha) = \sum_{i=1}^{N} y_i \ \frac{1}{g\kappa(\mathbf{x}_i))} \nabla g(\kappa(\mathbf{x}_i)) + (1 - y_i) \ \frac{1}{g(-\kappa(\mathbf{x}_i))} \nabla g(-\kappa(\mathbf{x}_i)) \tag{17}$$

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

$$\nabla\ell(Y,X,\alpha) = \sum_{i=1}^{N} y_i \frac{1}{g(\kappa(\mathbf{x}_i))} g(\kappa(\mathbf{x}_i))(1-g(\kappa(\mathbf{x}_i)))\nabla(\kappa(\mathbf{x}_i)) \quad (18)$$

$$+(1-y_i)\frac{1}{g(-\kappa(\mathbf{x}_i))} g(-\kappa(\mathbf{x}_i))(1-g(-\kappa(\mathbf{x}_i)))\nabla(-\kappa(\mathbf{x}_i)) \quad (19)$$

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

$$\nabla\ell(Y,X,\alpha) = \sum_{i=1}^{N} y_i g(-\kappa(\mathbf{x}_i))\nabla(\kappa(\mathbf{x}_i)) + (1-y_i)g(\kappa(\mathbf{x}_i))\nabla(-\kappa(\mathbf{x}_i)) \quad (20)$$

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

$$\frac{\partial\ell}{\partial\alpha_k} = \sum_{i=1}^{N} y_i g\left(-\sum_{j=1}^{N}\alpha_j K(x_j,x_i)\right)(K(x_i,x_k)) + (1-y_i)g\left(\sum_{j=1}^{N}\alpha_j K(x_j,x_i)\right)(-K(x_i,x_k)) \quad (21)$$

because

$$\frac{\partial}{\partial\alpha_k}\kappa(\mathbf{x}_i) = \frac{\partial}{\partial\alpha_k}\left[\sum_{j=1}^{N}\alpha_j K(\mathbf{x}_j,\mathbf{x}_i)\right] = K(\mathbf{x}_k,\mathbf{x}_i). \quad (22)$$