COMP 642: Machine Learning

Lecture 3: Linear Classifiers I - *January 18, 2022*

Lecturer: Anshumali Shrivastava

Scribe by: Mieke Groothuizen, Maria Briceno-Rojas, Hannah Lei, Eric Smith

Previously in Machine Learning

The previous lectures introduced the foundational information about data and how we will represent it in this course.

The motivation moving from there is to understand using the data to predeict target values for new data. This is done by creating models and optimizing them - fundamental to Machine Learning.

The task is to

- Select a Function Class and Loss function
- Minimize the Loss function by solving for the value of w that minimizes it - the model parameters that predicts actual values best

Gradient Descent

Gradient descent is a local descent algorithm. It can give us the direction to move when solving the optimization problem iteratively. The optimization problem being: how do we minimize the loss function for our model?

We start with a random w_0 and iteratively calculate w_t. Our simple iterative gradient descent algorithm is:

$$w_t = w_{t-1} - \eta \nabla f(w_{t-1})$$

where $\nabla f(w_{t-1})$ is the gradient at $_{t-1}$ and η is the step size.

But how do we calculate the gradient? And what step sizes do we take?

We can take the gradient by calculating the partial derivatives with respect to every dimension in the weights vector:

$$\nabla f(w) = \begin{bmatrix} \frac{\delta f}{\delta w_1}(w) \\ \vdots \\ \frac{\delta f}{\delta w_n}(w) \end{bmatrix}$$

Some considerations:

1. The Step size can't be too large or too small
2. The gradient can't be too costly to compute

Newton's Method

As mentioned in the previous lecture, we could also use Newton's method to calculate a more precise w, but it would require calculating a second order derivative of the loss function, and is computationally expensive. While calculating it might make it take less steps to reach our minima, each step would be more costly to compute, resulting in the optimiztion taking longer overall. In practice, it is not used.

$$w_t = w_{t-1} - \frac{\nabla f(w_{t-1})}{\nabla^2 f(w_{t-1})}$$

Step Size

Step Size (η) can't be too big or two small. Sometimes η is called the "learning rate" in machine learning spaces

- If the step size is too big you can overshoot the minima and never find the best model
- If the step size to too small you will not make progress quickly enough toward the goal.

The ideal step size might be too costly to calculate and could vary depending on your data and chosen gradient descent optimization algorithm.

Computational Intensity

Calculating the gradient of the loss function requires taking a pass over the entire dataset, which, over billions of data can be too computationally expensive. This is because the loss function is defined as the average "distance" for all datapoints:

$$L(F) = \frac{1}{n}\sum_{t=1}^{n} \text{dist}(F(x_i) - y_i)$$

There are numerous methods of calculating the gradient in a way that does not require passing over the entire dataset. A select few of these are covered below. There are many, many more gradient descent optimization algorithms, and different ones will perform better in different situations. Overall, the goal of each is to find a good direction to move while minimizing the amount of computation required with each step.

Stochastic Gradient Descent (SGD)

By randomly sampling the dataset and calculating the gradient with those, we can approximate the overall gradient. This is covered in more detail in the previous lecture, but the summary is this: we want to find a batch size k that gives us a good gradient direction but doesn't take a long time to compute.

Average Gradient Descent

Since SGD doesn't always move us in the right direction, we can use Average SGD, which has us average the gradients with past-calculated gradients (called momentum). The update is the same as for ordinary stochastic gradient descent, but the algorighm also keeps track of $\overline{w}$. When the optimization is done, this averaged parameter vector takes the place of w.

$$\overline{w} = \frac{1}{t}\sum_{i=1}^{t-1} w_i$$

Additionally, we could start decaying the gradients that came before so the first few calculated gradients aren't steering the newly calculated gradients in the wrong direction.

$$g_t = \nabla f(w_t)$$

$$g_{t+1} = \beta g_{t-1} + (1-\beta) g_t$$

where g_t is the gradient and β is the decay

Both of these are covered in more detail in a previous lecture.

ADAM (Adaptive Moment Estimation)

$$g_{t+1} = \beta g_{t-1} + (1-\beta) g_t$$

$$v_t = \beta v_t + (1-\beta) g_t^2$$

$$\eta = \frac{1}{\sqrt{v_t + \epsilon}}$$

v_t is velocity which represents the sum of the squared gradient of the past

For further explanation of the equations noted above, refer to ADAM

Today in Machine Learning

Today's lecture introduces the methodology to find these minimum values as well as information about different Function Classes and Loss Functions that can be chosen to model

Linear Classifiers

A linear classifier uses an object's characteristics, or features, to identify which 'class' it belongs to. Classes are things like 'dog' or 'cat' or 'apple'.

Linear classifiers have the form:

$$F_w(x) = w_1x_{i1} + w_2x_{i2} + \ldots + w_dx_{id} + b$$

which is equivalent to

$$w^Tx_i + b$$

Or, if we include a constant 1 at the end of the features array w:

$$F_w(x) = w^Tx$$

All three formulas are functionally the same

Below we dive into more detail on 3 types of linear classifiers

Linear Regression

Function:

$$F_w(x) = w^Tx + b$$

Loss Function:

$$L(F_w) = \frac{1}{n}\sum_{i=1}^{n}(F_w(x_i) - y_i)^2$$

Note:

- $F_w(x_i)$ is the predictor
- y_i is the actual value (which can be real number or vector)
- $(F_w(x_i) - y_i)^2$ is squared of the euclidean distance

Linear Regression also has a known closed form solution, which can be useful if, with your dataset, its possible to calculate the inverse of (X^TX):

$$\hat{w} = (X^TX)^{-1}X^TY$$

- X is the matrix of our input data with n rows and d columns
- Y is the vector of our output data

Derevation:

Group all of the output data into a single column vector:

$$Y = \begin{bmatrix} y1 \\ y2 \\ \vdots \\ y_n \end{bmatrix}$$

Group our parameters into a column vector:

$$w = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}$$

Group our input data into a matrix

$$X = \begin{bmatrix} 1 & x_{11} & \dots & x_{1d} \\ 1 & x_{21} & \dots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \dots & x_{nd} \end{bmatrix}$$

At each data point, using the parameters results in some error, which is a vector we can use in a mean squared error calculation, which we can then minimize:

$$\text{MSE}(w) = \frac{1}{n}\sum_{i=1}^{n}(Y - Xw)^2$$

To find the minimum, we take the gradient of the MSE with respect to w:

$$\begin{aligned} \text{MSE}(w) &= \frac{1}{n}(Y - Xw)^T(Y - Xw) \\ &= \frac{1}{n}(Y^T - w^T X^T)(Y - Xw) \\ &= \frac{1}{n}(Y^T Y - Y^T Xw - w^T X^T Y + w^T X^T Xw) \\ &= \frac{1}{n}(Y^T Y - 2w^T X^T Y + wX^T Xw) \end{aligned}$$

$$\begin{aligned} \nabla\text{MSE}(w) &= \frac{1}{n}(\nabla Y^T Y - 2\nabla w^T X^T Y + \nabla w^T X^T Xw) \\ &= \frac{1}{n}(0 - 2X^T Y + 2X^T Xw) \\ &= \frac{2}{n}(X^T Xw - X^T Y) \end{aligned}$$

We can now set this to zero and solve for w to find $\hat{w}$:

$$\begin{aligned} X^T Xw - X^T Y = 0 \\ \hat{w} = (X^T X)^{-1} X^T Y \end{aligned}$$

Perceptron

A binary classifier used in supervised learning for classification - usually used to classify data into two parts. The perceptron was inspired by the biological neuron and its ability to learn. \

One thing to note is that perceptrons are linearly separable, so the solution to the problem of linear separation is activation functions.

A perceptron consists of four parts: input values, weights and a bias, a weighted sum, and activation function.

$$(x_i, y_i)_{i=1}^{n}$$

where

$$y_i \in \{0, 1\} \text{ or } y_i \in \{-1, 1\} \text{ i.e. binary classification}$$

Function:

$$F_w(x_i) = \text{sign}\{w^T x_i + b\}$$

Loss Function:

$$L(F_w) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases} 1 & \text{if } F_w(x_i) \neq y_i \\ 0 & \text{otherwise} \end{cases}$$

Or, if $y_i \in \{0, 1\}$, we could also use this alternative loss function:

$$L(F_w) = \frac{1}{n}\sum_{i=1}^{n}(F_w(x_i) - y_i)^2$$

Since taking the derivative of the sign function is difficult (which is used to calculate the gradient), we can instead use the following update step:

$\text{If } F_w(x_i)) = y_i; \text{do nothing}$
$\text{Otherwise } w_{t+1} = w_t + \eta * y_i * x_i$

Where y is the desired output, x is the input vector, and η is the learning rate.

The learning rate constant η controls how quickly the update process changes in response to new data.

Note, there are several variants of this equation, e.g.

$$w_{t+1} = w_t + \eta(y_i - F_w(x_i))x_i$$

When trying to understand why they are equivalent, notice that the only possible values for y_i and $F_w(x_i)$ are 1 and -1. So if $y_i = F_w(x_i)$, then $(y_i - F_w(x_i))$ is zero and our update rule does nothing. Otherwise, $(y_i - F_w(x_i))$ is equal to either 2 or -2, with the sign matching that of the correct answer y_i. The factor of 2 is irrelevant, because we can tune η to whatever we wish.

This training procedure will converge if

- data are linearly separable or
- we throttle the size of the updates as training proceeds by decreasing η.

Apparently convergence is guaranteed if the learning rate is proportional to $\frac{1}{t}$ where t is the current iteration number.

For complementary reading, please refer to Perceptron Learning Rule in which it takes a deeper dive into Perceptrons (with diagrams!)

Logistic Regression

Used for classification to predict a class when there are more than two clases

Every Class has a probability associated with it

Function:

$$\mathbb{P}(x_i \in \text{``Class A''}) = \frac{e^{w_i^T x}}{\sum_{j=0}^{k-1} e^{w_i^T x}}$$

We go over the rest of logistic regression (and its associated loss function) in a later lecture