COMP0088
Introduction to Machine Learning
Lab Assignment 2

Matthew Caldwell

September 17, 2021

Task Summary

Week 2 Notes

The assignment script is `week_2.py`, generating output file `week_2.pdf`. There are some potentially-useful command line options, which you can list using:

```
$ python week_2.py -h
```

Example output from a completed assignment is shown in Fig. 1. Plotting code is provided, so your output should look very similar (for completed parts of the assignment) apart from random variations in the data.

The script uses functions you implemented in the previous assignment to generate test data—make sure your `week_1.py` is present in the same directory. If you were not able to complete that assignment, the TAs can provide a substitute `week_1.py` with working code.

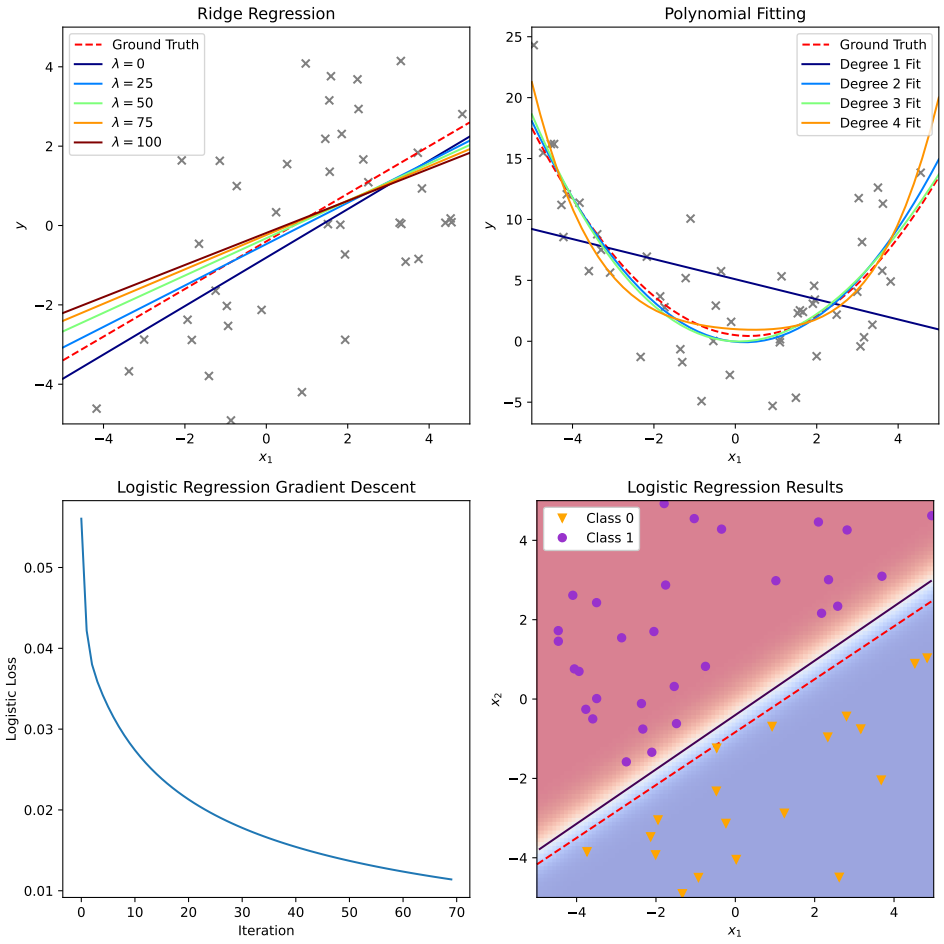

Figure 1: Example output from the `week_2.py` script

1 Ridge regression

Ridge regression is an extension of ordinary least squares (OLS) regression with a regularising penalty on the (squared) L_2 norm of the fitted weights. That is, given inputs $\mathbf{X}$ and corresponding outputs $\mathbf{y}$, we seek a vector $\mathbf{w}^\star$ such that:

$$\mathbf{w}^\star = \underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 + \lambda \|\mathbf{w}\|^2 \tag{1}$$

where λ is a hyperparameter that specifies the amount of regularisation. When $\lambda = 0$ the problem reduces to OLS.

Unlike many machine learning optimisations, ridge regression can be solved in closed form:

$$\mathbf{w}^\star = (\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda \mathbf{I})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} \tag{2}$$

Note that because this solution requires computing a matrix inverse—which can be computationally expensive—a numerical optimisation approach may still be preferable for larger problems.

1.1 Implement ridge regression in closed form

Implement the body of the following function in `week_2.py`:

```
def ridge_closed ( X, y, l2=0 ):
    """
    Implement L2-penalised least-squares (ridge) regression
    using its closed form expression.

    # Arguments
        X: an array of sample data, where rows are samples
           and columns are features (assume there are at least
           as many samples as features). caller is responsible
           for prepending x0=1 terms if required.
        y: vector of measured (or simulated) labels for the samples,
           must be same length as number of rows in X
        l2: optional L2 regularisation weight. if zero (the default)
           then this reduces to unregularised least squares

    # Returns
        w: the fitted vector of weights
    """
    assert(len(X.shape)==2)
    assert(X.shape[0]==len(y))

    # TODO: implement this
    return None
```

The NumPy function linalg.inv can be used to find the inverse of a matrix, though if you can rearrange the problem in suitable form for linalg.solve that is likely to be more efficient and numerically stable.[1]

[1] Both functions will fail if the problem is underdetermined. The pseudo-inverse (calculated by linalg.pinv) may be useful in that case, but you can assume it is not required here.

The `week_2.py` test driver will plot fits to the same data using multiple regularisation weights. The sample data itself is randomly generated using the noisy linear model function you wrote last week, so the data will differ each time you run. How do variations in the data affect the fits?

The script supports a command line option `--num_samples` (`-n`), which specifies how many data points to generate. How (if at all) does this interact with the L_2 regularisation?

If you are feeling completist, have a look at the code for `plot_ridge_regression_1d`. Do you notice anything problematic about it? (This is not a question about software engineering or code style!)

2 Linear models with basis expansion

Ridge regression and similar linear fitting approaches only require that the outputs are linear in the *parameters*. Fixed transformations of the inputs may be performed to fit non-linear functions of the *data*. The transformed data can be considered as a projection of the original samples into a new higher dimensional feature space.

For example, a **monomial basis** comprises the *products* of the source features (including with themselves). For a 1D feature space, the products are just the powers of that single feature, ie: $x \mapsto [x^0, x^1, x^2, ..., x^k]$ for some maximum degree k. A polynomial is just a weighted sum of monomial terms—that is, a vector in the monomial basis, which we can linearly fit.

2.1 Map 1D feature vectors to a monomial basis

Implement the body of the following function in `week_2.py`:

```
def monomial_projection_1d ( X, degree ):
    """
    Map 1d data to an expanded basis of monomials
    up to the given degree.

    # Arguments
        X: an array of sample data, where rows are samples
            and the single column is the input feature.
        degree: maximum degree of the monomial terms

    # Returns
        Xm: an array of the transformed data, with the
            same number of rows (samples) as X, and
            with degree+1 columns (features):
            1, x, x**2, x**3, ..., x**degree
    """
    assert(len(X.shape)==2)
    assert(X.shape[1]==1)

    # TODO: implement this
    return None
```

2.2 Generate noisy 1D polynomial data

Implement the body of the following function in `week_2.py`:

```
def generate_noisy_poly_1d ( num_samples, weights, sigma, limits, rng ):
    """
    Draw samples from a 1D polynomial model with additive
    Gaussian noise.

    # Arguments
        num_samples: number of samples to generate
            (ie, the number of rows in the returned X
            and the length of the returned y)
        weights: vector of the polynomial coefficients
```

```
            (including a bias term at index 0)
        sigma: standard deviation of the additive noise
        limits: a tuple (low, high) specifying the value
            range for the single input dimension x1
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        X: a matrix of sample inputs, where
            the samples are the rows and the
            single column is the 1D feature x1
            ie, its size should be:
              num_samples x 1
        y: a vector of num_samples output values
    """
    # TODO: implement this
    return None, None
```

This function is obviously rather similar to `generate_noisy_linear` from week 1. Note that `weights` includes a bias term, so the monomial degree will be 1 less than its length.

2.3 Implement linear fitting of a 1D polynomial

Implement the body of the following function in `week_2.py`:

```
def fit_poly_1d ( X, y, degree, l2=0 ):
    """
    Fit a polynomial of the given degree to 1D sample data.

    # Arguments
        X: an array of sample data, where rows are samples
            and the single column is the input feature.
        y: vector of output values corresponding to the inputs,
           must be same length as number of rows in X
        degree: degree of the polynomial
        l2: optional L2 regularisation weight

    # Returns
        w: the fitted polynomial coefficients
    """
    assert(len(X.shape)==2)
    assert(X.shape[1]==1)
    assert(X.shape[0]==len(y))

    # TODO: implement this
    return None
```

Plotting code is provided to show results from this function. As in Exercise 1, the data used is random. Run the script several times to see how random variations in the data affect the fits, and also try running with different values of `--num_samples`.

3 Gradient descent

Gradient descent is an iterative numerical method for solving optimisation problems where an analytic solution is unavailable or impractical. It is the basis for a family of closely-related methods that are very widely applied in ML and deep learning.

The basic procedure is shown in Algorithm 1; we might facetiously summarise it as "walk downhill".

Algorithm 1 Iterative minimisation by gradient descent

function GRADIENTDESCENT($f, \mathbf{z}_0, \alpha$)
  f : *the function to be minimised*
  $\mathbf{z}_0$: *an initial value for the function parameters* $\mathbf{z}$
  α : *some small step size, known as the learning rate*

  $\mathbf{z} \leftarrow \mathbf{z}_0$
  repeat
    $\mathbf{z} \leftarrow \mathbf{z} - \alpha \nabla_{\mathbf{z}} f$
  until *some stopping criterion reached*
  return z
end function

Note the following:

- It requires the gradient $\nabla_{\mathbf{z}}$ to be calculable, so f must be (more or less) continuous and differentiable. (There may be scope for fudging over the odd known undefined point—see Task 5.1.)
- Only local information is used at each step, so the process can get stuck in local minima if f is not convex.
- There is a hyperparameter α that must be chosen somehow, which may have a significant effect on how long the process takes to converge (or whether it does at all).
- When the gradient is shallow, convergence might be very slow.

3.1 Implement a generic gradient descent optimiser

Implement the following function in `week_2.py`:

```
def gradient_descent ( z, loss_func, grad_func, lr=0.01,
                       loss_stop=1e-4, z_stop=1e-4, max_iter=100 ):
    """
    Generic batch gradient descent optimisation.
    Iteratively updates z by subtracting lr * grad
    until one or more stopping criteria are met.

    # Arguments
        z: initial value(s) of the optimisation var(s).
            can be a scalar if optimising a univariate
            function, otherwise a single numpy array
        loss_func: function of z that we seek to minimise,
            should return a scalar value
        grad_func: function calculating the gradient of
```

```
            loss_func at z. for vector z, this should return
            a vector of the same length containing the
            partial derivatives
        lr: learning rate, ie fraction of the gradient by
            which to update z each iteration
        loss_stop: stop iterating if the loss changes
            by less than this (absolute)
        z_stop: stop iterating if z changes by less than
            this (L2 norm)
        max_iter: stop iterating after iterating this
            many times

    # Returns
        zs: a list of the z values at each iteration
        losses: a list of the losses at each iteration
    """
    # TODO: implement this
    return None, None
```

Although our primary goal here is to fit the parameters of machine learning models from potentially noisy data, gradient descent can be used to minimise any reasonably well-behaved function for which the gradient can be calculated. The `week_2.py` test driver applies it to a simple quadratic function,

$$f(x) = x^2 - 2x + 1 \tag{3}$$

for which the gradient is trivially derived,

$$f'(x) = 2x - 2 \tag{4}$$

and we can see that the function reaches a minimum of 0 at $x = 1$. Check that your gradient descent implementation manages to adequately estimate this result.

4 Logistic regression

Logistic regression is an adaptation of linear regression for binary classification. It fits a linear decision boundary between two classes using a **sigmoid** transformation of the output value that maps it from $(-\infty, +\infty)$ to $(0, 1)$:

$$\hat{\mathbf{y}} = \sigma(\mathbf{X}\mathbf{w}) \tag{5}$$

where σ is the **logistic function**:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6}$$

The predictions $\hat{\mathbf{y}}$ are evaluated against the true labels $\mathbf{y}$ using a **binary cross-entropy** loss:[2]

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{n}\left[\mathbf{y} \cdot \log(\hat{\mathbf{y}}) + (1 - \mathbf{y}) \cdot \log(1 - \hat{\mathbf{y}})\right] \tag{7}$$

Unlike for ridge regression, there is no closed form expression for finding the $\mathbf{w}$ that minimises this loss, and we must instead optimise numerically. Luckily, the loss function is differentiable and convex, so gradient descent is a reasonable approach.[3] As shown in the lectures, the gradient of the loss with respect to $\mathbf{w}$ is:

$$\nabla_{\mathbf{w}} L = \mathbf{X}^{\mathsf{T}}(\hat{\mathbf{y}} - \mathbf{y}) \tag{8}$$

4.1 Implement logistic regression

Implement the body of the following function in `week_2.py`:

```
def logistic_regression ( X, y, w0=None, lr=0.05,
                          loss_stop=1e-4, weight_stop=1e-4, max_iter=100 ):
    """
    Fit a logistic regression classifier to data.

    # Arguments
        X: an array of sample data, where rows are samples
           and columns are features. caller is responsible
           for prepending x0=1 terms if required.
        y: vector of binary class labels for the samples,
           must be same length as number of rows in X
        w0: starting value of the weights, if omitted
           then all zeros are used
        lr: learning rate, ie fraction of gradients by
           which to update weights at each iteration
        loss_stop: stop iterating if the loss changes
            by less than this (absolute)
        weight_stop: stop iterating if weights change by less
            than this (L2 norm)
        max_iter: stop iterating after iterating this
            many times

    # Returns
        ws: a list of fitted weights at each iteration
```

[2] The loss is expressed here using dot product notation rather than the more common summation, but the two are equivalent and this representation lends itself to a concise implementation in NumPy.

[3] Other algorithms are also available that may converge faster, but we won't worry about that here.

```
        losses: a list of the loss values at each iteration
    """
    assert(len(X.shape)==2)
    assert(X.shape[0]==len(y))

    # TODO: implement this
    return None, None
```

Use the `gradient_descent` function from Exercise 3.1 to do the optimisation. You will need to provide functions for the loss and gradient that take just the weights vector as their argument.

One thing to note when implementing this is that errors are likely to arise from trying to take the logarithm of zero. Strictly, the logistic function is never quite 0 or 1, but it gets *really* close. You will probably find it helpful to add a small offset value to $\hat{\mathbf{y}}$ and $1 - \hat{\mathbf{y}}$ when calculating the loss.

As with earlier exercises, plots will be produced using randomly generated data, with the quantity controlled by `--num_samples`. Observe how the data size and random fluctuations affect the fit over multiple runs.

5 Further exploration

If you have exhausted the previous exercises, you might find it interesting to try out one or more of the following tasks. Doing so is entirely optional, but may help to cement or extend some of the ideas learned this week.

These tasks are not included in the `week_2.py` script skeleton. You can add them in or start from scratch in a new script of your own. If you do the latter you will probably want to copy over your gradient descent implementation from Task 3.1.

5.1 Implement lasso regression using gradient descent

The loss function for lasso (L_1 regularised least squares) looks like this:

$$L(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 + \lambda\|\mathbf{w}\|_1 \tag{9}$$

The L_1 norm has a corner at zero, so this function is not strictly differentiable. However, it is easy to define a **subgradient** for the norm at this point—this could be any value in the interval $[-1, 1]$, but since this really is its minimum point the obvious choice is 0.

Armed with this subgradient, you should be able to implement the lasso loss and gradient functions and fit a lasso model by gradient descent. How do the results of such a fit compare with those from ridge regression?

5.2 Implement multinomial logistic regression using gradient descent

As discussed in the lectures, the binary classification model of logistic regression can be adapted for multiclass problems using **softmax** squashing and a **categorical cross-entropy** loss. The model becomes:

$$\hat{\mathbf{Y}} = \varsigma(\mathbf{X}\mathbf{W}) \tag{10}$$

where ς is the softmax function:

$$\varsigma(\mathbf{z}) = \frac{e^{\mathbf{z}}}{\|e^{\mathbf{z}}\|_1} \tag{11}$$

with exponentiation performed elementwise. The loss is:

$$L(\mathbf{Y}, \hat{\mathbf{Y}}) = -\frac{1}{n}\sum_i^n \mathbf{y}_i \cdot \log(\hat{\mathbf{y}}_i) = \frac{1}{n}\left\|\mathbf{Y}^\mathsf{T} \log(\hat{\mathbf{Y}})\right\|_1 \tag{12}$$

where the logs are similarly taken elementwise. The gradient is:

$$\nabla_{\mathbf{W}} L = \mathbf{X}^\mathsf{T}(\hat{\mathbf{Y}} - \mathbf{Y}) \tag{13}$$

Fitting this via gradient descent is similar to binary logistic regression, but you will need to manage mapping ground truth values into one-hot vectors and prediction vectors back to class labels. And remember that $\mathbf{Y}$, $\hat{\mathbf{Y}}$ and $\mathbf{W}$ are all matrices.

Multinomial logistic regression can be applied to data with two classes, so you could test your implementation with synthetic data from last week's `generate_linearly_separable` function. But you'll probably want to try it with more than two classes. One good source of test data is the `sklearn.datasets` package from `scikit-learn`, which should already be installed in your Python environment. For example, have a look at the `load_iris` function for Fisher's classic Iris dataset.

5.3 Implement stochastic/mini-batch gradient descent

In Task 3.1 you implemented *batch* gradient descent, where the loss gradient is calculated with respect to the whole training set before updating the parameters. In *mini-batch* gradient descent, updates are applied more frequently using gradients estimated using only a fraction of the training set. If that fraction is just a single sample at a time, then it is known as *stochastic* gradient descent.[4]

Try implementing mini-batch gradient descent with a configurable batch size and see how it behaves in comparison to the full batch version. (You will need to adapt the interfaces both for the optimiser and the loss functions to accept the training data.) Plot the loss history for all the mini-batch updates—is it noticeably different to that for batch gradient descent? Does the batch size affect this?

[4] In practice this distinction is quite often glossed over and the term 'stochastic gradient descent' applied for any mini-batch size.