COMP0088 Introduction to Machine Learning
Lab Assignment 1

Matthew Caldwell

September 14, 2021

Task Summary

Week 1 Notes

The script for this assignment is `week_1.py` and the output file it generates is `week_1.pdf`. An example output from a completed assignment is shown in Fig. 1. Your output does not need to look exactly like this, but it will probably have the same general form. (In future lab assignments the plotting functions will often be provided for you, but in this introductory exercise you are asked to do the plotting yourself and may style it however you like.)

This first assignment may be considered something of a warm-up exercise. It is an opportunity to check that you have a functioning development environment and get acquainted with some basic data generation and plotting. If you already have experience using NumPy and Matplotlib, you may find these exercises trivial. There are suggestions of other things to try in Task 4, but do finish the data generation functions first: `week_2.py` will use some code you write here.

If you are **not** familiar with NumPy and Matplotlib, we include a few pointers in the exercises below. Some example plotting code is also provided in the script `plotting_examples.py`, and

there are a few potentially helpful utility functions in the module `utils.py`. You don't have to make use of either of these, although you should probably at least look at the utilities—they may help you avoid some tedious mucking about with things like array axis ordering. (A couple of this week's exercises can be reduced to one-liners with judicious use of `utils`.)

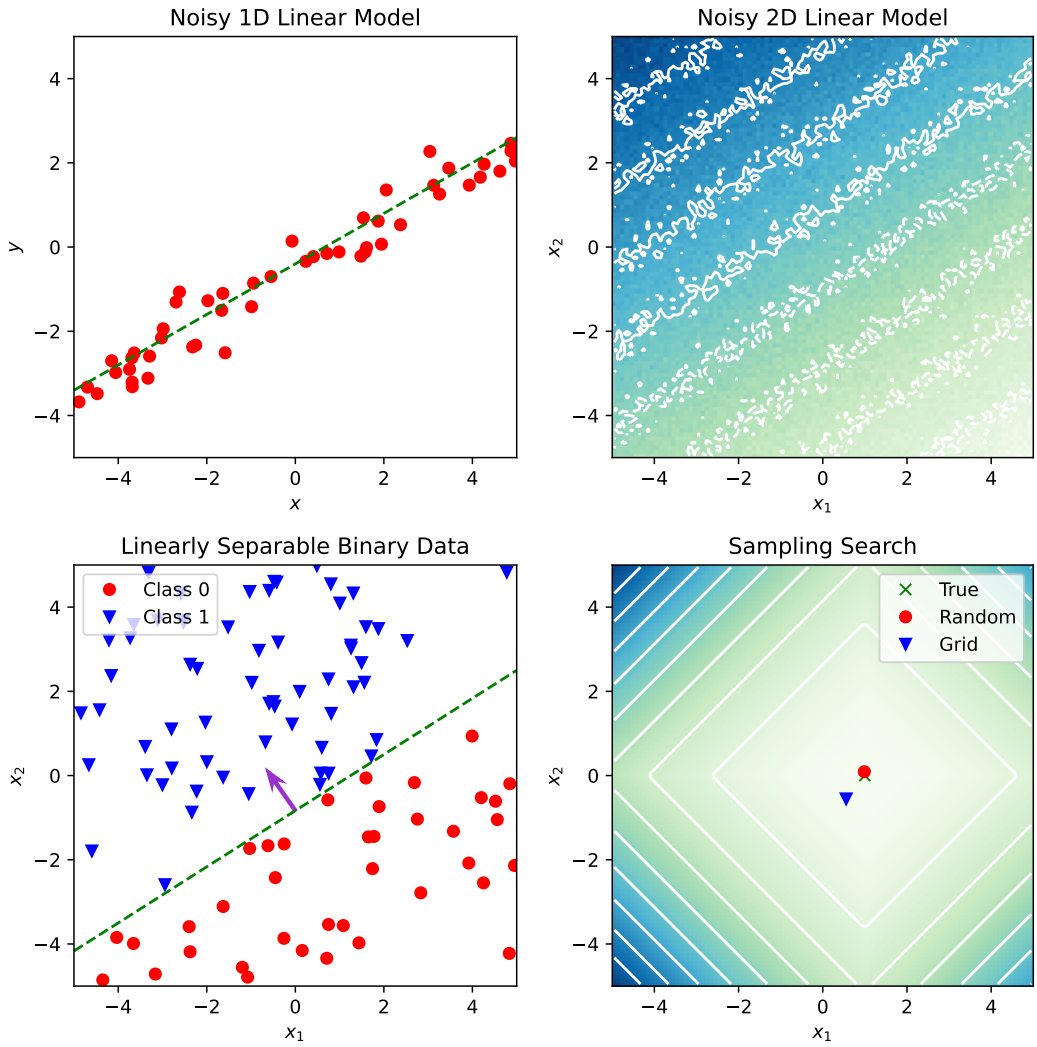

Figure 1: Example output from the `week_1.py` script

1 Generating linear continuous data

A continuous linear model is one whose output is just a weighted sum of the input features:

$$y = w_0 + w_1 x_1 + w_2 x_2 + ... + w_d x_d = \sum_{i=0}^{d} w_i x_i$$

where we've defined a constant dummy feature $x_0 = 1$ to capture the bias term w_0 in the sum. We can express this concisely in vector form:

$$y = \mathbf{w} \cdot \mathbf{x} \qquad \mathbf{w}, \mathbf{x} \in \mathbb{R}^{d+1} \tag{1}$$

This is a *deterministic* model, fully parameterised by the weight vector $\mathbf{w}$. In practice we might expect there to be some amount of measurement error or other uncertainty in the values we obtain for y. One way to represent this uncertainty is with an additive error term, ε:

$$y = \mathbf{w} \cdot \mathbf{x} + \varepsilon \tag{2}$$

This is not the only way to model the uncertainty, or necessarily the best, but it is nice and simple and very commonly used. In the absence of other information about ε, we will often further assume that it follows a Gaussian distribution with mean zero (since the location is already modelled by w_0) and standard deviation σ:

$$\varepsilon \sim N(0, \sigma^2) \tag{3}$$

Given $\mathbf{w}$ and σ, we can then generate any number of samples $(\mathbf{x}, y)$ from this model by choosing values for $\mathbf{x}$.

1.1 Generate samples from a linear model with additive Gaussian noise

Implement the body of the following function in `week_1.py`:

```
def generate_noisy_linear(num_samples, weights, sigma, limits, rng):
    """
    Draw samples from a linear 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 defining the model
            (including a bias term at index 0)
        sigma: standard deviation of the additive noise
        limits: a tuple (low, high) specifying the value
            range of all the input features x_i
        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
            features are the columns
            ie, its size should be:
              num_samples x (len(weights) - 1)
```

```
        y: a vector of num_samples output values
    """

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

Note the following:

- The function arguments `weights` and `sigma` correspond to $\mathbf{w}$ and σ in Eqs. (2) and (3) above.
- `weights[0]` is the bias term w_0. Hence, the number of features, m, is one less than the length of `weights`.
- Use `rng` to obtain random numbers. The Generator class is documented here, but probably the most immediately relevant methods are `random` and normal.

1.2 Plot a 1D linear model

Implement the remainder of this function in `week_1.py`:

```
def plot_noisy_linear_1d(axes, num_samples, weights, sigma, limits, rng):
    """
    Generate and plot points from a noisy single-feature linear model,
    along with a line showing the true (noiseless) relationship.

    # Arguments
        axes: a Matplotlib Axes object into which to plot
        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 defining the model
            (including a bias term at index 0)
        sigma: standard deviation of the additive noise
        limits: a tuple (low, high) specifying the value
            range of all the input features x_i
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        None
    """
    assert(len(weights)==2)
    X, y = generate_noisy_linear(num_samples, weights, sigma, limits, rng)

    # TODO: do the plotting
    utils.plot_unimplemented ( axes, 'Noisy 1D Linear Model' )
```

If you are new to plotting with Matplotlib, you might find it helpful to refer to the `xy_scatter` function in `plotting_examples.py`.

1.3 Plot a 2D linear model

Implement the remainder of this function in `week_1.py`:

```
def plot_noisy_linear_2d(axes, resolution, weights, sigma, limits, rng):
    """
    Produce a plot illustrating a noisy two-feature linear model.

    # Arguments
        axes: a Matplotlib Axes object into which to plot
        resolution: how densely should the model be sampled?
        weights: vector defining the model
            (including a bias term at index 0)
        sigma: standard deviation of the additive noise
        limits: a tuple (low, high) specifying the value
            range of all the input features x_i
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        None
    """
    assert(len(weights)==3)

    # TODO: generate the data
    # TODO: do the plotting
    utils.plot_unimplemented ( axes, 'Noisy 2D Linear Model' )
```

While this problem is similar to the previous one, note that this model is significantly harder to visualise because there are three dimensions of data to consider—two inputs and one output. To make matters worse, the Axes object you are passed in the first argument is only a 2d cartesian one, not a 3d projection. So you'll need to consider how to represent it, and that may affect what data you need to generate.

(You might find it helpful to postpone this until after you have finished Exercise 3.)

2 Generating linearly separable binary data

We can use an equation akin to the model in Eq. (1) to define a linear boundary—or **separating hyperplane**—that divides a feature space into disjoint **half-spaces**:

$$\mathbf{w} \cdot \mathbf{x} = 0 \tag{4}$$

In a **binary classification** problem, if there exists at least one such hyperplane for which all samples of class 0 are on one side and all those of class 1 are on the other, then the data are said to be **linearly separable**, and we can define a classifier using the hyperplane as a decision boundary:

$$y = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Putting aside the question of how to *find* such a plane, or whether it makes a *good* classifier, we can see that (as with the continuous case in Task 1) it is easy to generate samples $(\mathbf{x}, y)$ given $\mathbf{w}$.

2.1 Generate sample data with binary labels that are linearly separable in a continuous feature space

Implement the body of the following function in `week_1.py`:

```
def generate_linearly_separable(num_samples, weights, limits, rng):
    """
    Draw samples from a binary model with a given linear
    decision boundary.

    # 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 defining the decision boundary
            (including a bias term at index 0)
        limits: a tuple (low, high) specifying the value
            range of all the input features x_i
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        X: a matrix of sample vectors, where
            the samples are the rows and the
            features are the columns
            ie, its size should be:
              num_samples x (len(weights) - 1)
        y: a vector of num_samples binary labels
    """
    # TODO: implement this
    return None, None
```

(The notes for Task 1.1 also apply here.)

2.2 Plot a set of labelled 2D samples and their boundary line

Implement the remainder of this function in `week_1.py`:

```
def plot_linearly_separable_2d(axes, num_samples, weights, limits, rng):
    """
    Plot a linearly separable binary data set in a 2d feature space.

    # Arguments
        axes: a Matplotlib Axes object into which to plot
        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 defining the decision boundary
            (including a bias term at index 0)
        limits: a tuple (low, high) specifying the value
            range of all the input features x_i
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        None
    """
    assert(len(weights)==3)
    X, y = generate_linearly_separable(num_samples, weights, limits, rng)

    # TODO: do the plotting
    utils.plot_unimplemented ( axes, 'Linearly Separable Binary Data' )
```

Aim to include the following in your plot:

- All the generated sample points, with their class clearly indicated (e.g. by colour and shape).
- A line marking the decision boundary. You will need to calculate the end points of this from the weight vector and the bounds of the plot. (Are there any edge cases you might need to take into account?)
- An arrow to show the weight vector itself.

Verify that the weight vector is normal to the boundary and points towards the positive class.

3 Searching for the minimiser of a function

In Lecture 1.3 we introduced the notation

$$\operatorname*{argmin}_x f(x)$$

to denote the problem of finding the value of the input x that minimises the output of f. In general this problem may be arbitrarily difficult, since f could be anything.

The NumPy library includes a function of the same name, `argmin`, which addresses a much more limited (and tractable) task: it finds the location of the smallest value in an array. Despite its simplicity, this can sometimes be enough to find the exact minimiser (if x can only take on a few discrete values) or at least to find an approximate one by **sampling**.

Let **x** be an vector of sampled inputs $[x_1, x_2, ..., x_n]$ and **y** the vector of corresponding outputs $[f(x_1), f(x_2), ..., f(x_n)]$. If $j =$ `argmin(y)` (in the NumPy sense), then x_j is (of the available choices) our best estimate of $\operatorname{argmin}_x f(x)$.

There are various possible ways to choose candidate values of x, but here we'll look at two common ones: either randomly or at regular intervals.

3.1 Perform a random search for the minimum value of a function

Implement the following function in `week_1.py`:

```
def random_search(function, count, num_samples, limits, rng):
    """
    Randomly sample from a function of `count` features and return
    the best feature vector found.

    # Arguments
        function: a function taking a single input array of
            shape (..., count), where the last dimension
            indexes the features
        count: the number of features expected by the function
        num_samples: the number of samples to generate & search
        limits: a tuple (low, high) specifying the value
            range of all the input features x_i
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        x: a vector of length count, containing the found features
    """

    # TODO: implement this
    return None
```

Some points to note:

- Here (and below, and in future assignments) we make use of the ability to pass one function to another as an argument. To execute the function passed in argument `function` on some array `X` you just call it like any other function:

```
    y = function(X)
```

- The return value should be a single feature vector: all other samples are discarded.

3.2 Perform a grid search for the minimum value of a function

Implement the following function in `week_1.py`:

```
def grid_search(function, count, num_divisions, limits):
    """
    Perform a grid search for a function of `count` features and
    return the best feature vector found.

    # Arguments
        function: a function taking a single input array of
            shape (..., count), where the last dimension
            indexes the features
        count: the number of features expected by the function
        num_divisions: the number of samples along each feature
            dimension (including endpoints)
        limits: a tuple (low, high) specifying the value
            range of all the input features x_i

    # Returns
        x: a vector of length count, containing the found features
    """

    # TODO: implement this
    return None
```

A common NumPy idiom for grid sampling (inherited from Matlab) employs the functions linspace and meshgrid to generate evenly spaced values and combine dimensions respectively. An example of this usage can be found in `utils.make_grid`. You may find it more convenient to use this function, since it also conforms the results to our feature indexing convention.

How does the computational complexity of `grid_search` scale with the number of features? What does that suggest about its potential applicability?

3.3 Plot a 2D function along with minimum values found by grid and random searching

Implement the following function in `week_1.py`:

```
def plot_searches_2d(axes, function, limits, resolution,
                     num_divisions, num_samples, rng, true_min=None):
    """
    Plot a 2D function aling with minimum values found by
    grid and random searching.

    # Arguments
        axes: a Matplotlib Axes object into which to plot
```

```
        function: a function taking a single input array of
            shape (..., 2), where the last dimension
            indexes the features
        limits: a tuple (low, high) specifying the value
            range of both input features x1 and x2
        resolution: number of samples along each side
            (including endpoints) for an image representation
            of the function
        num_divisions: the number of samples along each side
            (including endpoints) for a grid search for
            the function minimum
        num_samples: number of samples to draw for a random
            search for the function minimum
        rng: an instance of numpy.random.Generator
            from which to draw random numbers
        true_min: an optional (x1, x2) tuple specifying
            the location of the actual function minimum

    # Returns
        None
    """
    # TODO: implement this
    utils.plot_unimplemented ( axes, 'Sampling Search' )
```

Use the functions defined in the previous sections to perform the grid and random searches.

Try to include most or all of the following features:

- An **image** (`resolution` × `resolution` pixels) showing the output values of the function over its full range in both feature dimensions, with the output value represented by colour. Generating this will require similar steps to those in your grid search, but you will need all the returned function values rather than the minimum feature vector. (You may find the `xyz_as_image` function in `plotting_examples.py` helpful here.)
- A marker indicating the minimum point found by a random search.
- A marker indicating the minimum point found by a grid search.
- If `true_min` is provided, a marker showing the real function minimum.

The driver code in `week_1.py` uses the following function as a test case:

$$y = (x_1 - 1)^2 + x_2^2 + \left|2(x_1 - 1)x_2\right| \tag{6}$$

It can be seen by inspection that this function has its minimum at (1, 0). How does this compare to the values discovered by the grid and random searches? How consistent is that if you re-run the test several times? Looking at the parameters with which the script calls `plot_searches_2d`, how might you estimate the probability that random search outperforms grid search here?

4 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 provide some additional perspective that could be useful in the weeks ahead.

These tasks are not included in the `week_1.py` script skeleton. You can add them in, but you may find it more convenient to start from scratch in a new script of your own.

(The `utils` module includes a function `plot_classification_map`—which we'll be using in future weeks—that you may find useful for visualising decision boundaries in 4.2–4.4.)

4.1 Plot the 2D models from Tasks 1.3 and 3.3 in 3D projection

For simplicity, in the exercises above we plotted the models above (with 2 input dimensions and 1 output) as flat images, using colour to represent the third dimension. Using a 3D projection can sometimes be more intuitive. Try plotting those models in such a projection.

(Matplotlib uses a different Axes class for 3D plots, so you will need to specify `projection='3d'` when adding the axes object to your figure. See `plotting_examples.py` for an example of this.)

4.2 Construct decision boundaries by preprocessing the inputs

In Eq. (5), the decision boundary is based on a weighted sum of the raw input vector $\mathbf{x}$. What would happen if you instead applied some preprocessing function $f : \mathbb{R}^d \mapsto \mathbb{R}^k$ to the $\mathbf{x}$ values before calculating the dot product (adjusting the dimension of $\mathbf{w}$ accordingly)?

In this case the decision function becomes:

$$y = \begin{cases} 1 & \text{if } \mathbf{w} \cdot f(\mathbf{x}) \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

What might be some interesting functions to use as f? Try implementing them for $d = 2$ and see what boundaries you can construct.

4.3 Construct decision boundaries by using a different functional form for y

In the previous question, the preprocessing function f does not involve the weights vector $\mathbf{w}$. Rather than invoking such an extra function, we might instead use a modified decision function that uses the weights in a different way.

For example, consider using a $(d+1) \times (d+1)$ *matrix* of weights, $\mathbf{W}$, rather than just a vector, and adapting the decision function like this:

$$y = \begin{cases} 1 & \text{if } \mathbf{x}^\mathsf{T}\mathbf{W}\mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

What kinds of decision boundaries might this enable? Can you suggest any conditions under which this decision function is or is not useful? Again, try implementing this for $d = 2$ and see what you get.

4.4 Construct decision boundaries iteratively or recursively

As another alternative, we might choose not restrict ourselves to a single set of weights but instead have multiple distinct weight vectors, $\mathbf{w}_1, \mathbf{w}_2, ...\mathbf{w}_k$. Each of these would define a linear boundary, just as in Eq. (5), but what might that mean collectively?

Suppose we define our decision function as

$$y = \sum_{i}^{k} \mathbf{1}(\mathbf{w}_i \cdot \mathbf{x} \geq 0) \tag{9}$$

where **1** is the **indicator function**:

$$\mathbf{1}(condition) = \begin{cases} 1 & \text{if } condition \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

What kind of value is y? What kinds of decision boundaries might result?

Suppose instead we have $k = 3$ and define our decision function as

$$y = \begin{cases} \mathbf{1}(\mathbf{w}_2 \cdot \mathbf{x} \geq 0) & \text{if } \mathbf{w}_1 \cdot \mathbf{x} \geq 0 \\ \mathbf{1}(\mathbf{w}_3 \cdot \mathbf{x} \geq 0) & \text{otherwise} \end{cases} \tag{11}$$

What sort of decision boundaries could we define like this? What if we allowed k to be much larger?

Again, try implementing some of these for $d = 2$ and see what you can come up with.