Nonlinear transformations

1. The dataset in `rings.npz` is a binary classification problem.

   If you do not yet have a working logistic regression implementation, you can use that of scikit learn; see the documentation and the user guide for more information.

   a. Plot the data points with their classes (you can use the `visualise.py` provided (this is a new one)).

   b. Try the classification with a linear classifier (eg. PLA or logistic regression), and print the training/test errors, and plot the resulting classifier.

   c. Try the classification with the norm of the original input samples ($\|x\|_2$) added as a new feature. (Again, print the errors, plot the classifier.)

   d. Try the classification with polynomials of degree 2 made from the inputs as new features. (Again, print the errors and plot the classifier.)

      If you have two variables x_1 and x_2, then the degree 2 polinomials are: x_1^2, $x_1 x_2$ and x_2^2. Use these as new features.

2. You have been given a training dataset of size N that was generated by a target distribution with two classes. The distribution of the negative class is uniform in the $\{x : x_1^2 + x_2^2 \leq 1\}$ set, the distribution of the positive class is uniform on the $\{x : 1 < x_1^2 + x_2^2 \leq 2\}$ set.

   You have run a very clever classification algorithm, the result of which is the hypothesis: $f(x) = \text{sign}\{x_1^2 + x_2^2 - 1.1\}$.

   What is the E_{out} for this hypothesis (using accuracy as an error measure)?

3. *This problem is optional and is not worth points.* Three.

4. When implementing the gradient calculation by hand,[1] one sometimes makes mistakes. One easy way to verify your calculations is to compare the derivatives to derivatives computed by the method of *finite differences*.

   Derived from the definition of the derivative, the estimation is:

   $$f'(x) = \frac{f(x + \epsilon/2) - f(x - \epsilon/2)}{\epsilon} \tag{1}$$

   for some suitable ϵ: it shouldn't be too small to avoid numerical errors (eg. from rounding), but it should be small enough to approximate the derivative.

   For functions with vector input, this only estimates the partial derivative, and needs to be repeated for each dimension of the input.

   a. In your logistic regression implementation, put an estimation of the finite difference of the error (estimating the gradient). (Here, the f in (1) is the $E_i n$ with w as the input.) Print out both values: the exact gradient ∇E_{in} and the finite difference.

      Modify the formula calculating the gradient so it has an error in it. Compare the two values again.

[1] There are libraries nowadays that can (most of the time) do this for you, see, for example, PyTorch, TensorFlow or JAX. (More on these later.)

b. *(Optional)* A common practice to speed up this calculation for vector valued functions (which, for m inputs and n outputs need mn difference calculations) is to randomly project the inputs and the outputs: for $g : \mathbb{R}^m \to \mathbb{R}^n$, test the function $f(x) = u^T g(vx)$ instead, with $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^m$ random vectors.

(If you haven't done so already) extract the calculations of the error and its gradient into their own functions, then test them on a number random inputs x and vectors u and v.

c. *(Optional)* If one has access to complex numbers (which we do in Python), there is another method:

$$f(x + i\epsilon) = f(x) + i\epsilon f'(x) + O(\epsilon^2) \tag{2}$$

(where i is the imaginary unit), which means the derivative is

$$f'(x) = \Im\left\{\frac{1}{\epsilon} f(x + i\epsilon)\right\} + O(\epsilon^2). \tag{3}$$

Repeat Exercise 2a or 2b with this estimation as well.