Logistic regression

1. Recall that the (logistic) sigmoid function is:

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

(for $x \in \mathbb{R}$). Also note that with some mathematical trickery, one can get:

$$\sigma(x) = \frac{e^x}{1 + e^x}, \tag{2}$$

these two formulae are equivalent. Mathematically, at least.

   a. For the following values of x, please calculate the values of the logistic sigmoid: 0, -1, 1, -10, 10, -1000, 1000.
   b. Create a sigmoid function implementation that works for all the values above.
   c. *(Optional)* Your implementation should avoid any overflows during calculation.

2. Using the `pla.npz` and `pocket.npz` datasets from the previous lab, classify the samples using logistic regression.

   Note that the class labels are -1 and 1, but the range of sigmoid is $[0, 1]$, take care of transformations where needed.

   a. Implement:

      - a prediction function for the logistic regression model, which calculates the probability that the given input belongs to the positive class,
      - an error function, which calculates the cross-entropy error for the given input and output
      - an error gradient function, which calculates the gradient with respect to the model parameters at the given input and output

   b. Using the functions implemented above, write a full batch gradient descent training algorithm. In each iteration, calculate the mean error on the training dataset, calculate its gradient and update your parameter vector using a properly chosen step size.

   c. Visualise the linear separator in each iteration.

   d. Add stochastic gradient descent functionality to your code with a batch size parameter, where `batch_size = None` means full batch descent. After each couple of iterations, visualise the linear separator (as before).

   e. In addition, in a separate subplot plot each of the training samples transformed into the score space (ie. just before the sigmoid) and a logistic sigmoid function as reference.

      This means a plot where the horizontal axis is the score $s = w^T x$ and the vertical axis is y. You will have two plots: one is the logistic sigmoid function, calculated analytically, and the other is the *ground truth* ys versus the score $s = w^T x$.

   f. Also print and plot the norm of the gradient and the in sample error (both cross-entropy and accuracy) as the training progresses.

3. (Problem 2.2 in the LfD book) Show that for the learning model of positive rectangles (aligned horizontally or vertically), $m_{\mathcal{H}}(4) = 2^4$ and $m_{\mathcal{H}}(5) < 2^5$. Hence, give a bound for $m_{\mathcal{H}}(N)$.

4. During logistic regression training, collect (training) accuracies and corresponding cross-entropy errors. Plot them on a scatterplot for datasets `pla.npz`, `pocket.npz`, `fade_in.npz`, `fade_out.npz` and `moons.npz` (denoting the different datasets by, for example, different colours).

   Can you see any patterns from the plot, any connections between the curves and features of the datasets?

5. *(Optional)* Use the dataset `yesno.npz` to build a mini speech recognition software (that can tell the words "yes" and "no" apart).

   Load the yes/no dataset, and transform it to a more easily classifiable form: apply Discrete Fourier transformation. Use `scipy.fftpack.rfft()` and `nb.abs` to get the magnitude of frequencies. Fit your classifier on the training data (`X_train` and `Y_train` for the inputs and labels, resp.), and measure your error on the test data.

   Watch out: the targets (outputs) in `Y_train` are either 0 or 1, so the gradient formula on the presentation slides won't work without modification.

   For optimisation, use the Stochastic Gradient Descent algorithm with variable minibatch size (implemented in Exercise 2). Experiment with different stopping criteria, learning rate and minibatch sizes. How do they affect training and generalisation error?

   The source of the dataset is this challenge. I put the indexes of the original WAV files in the npz file if you want to listen to the input.

   Optional: using your microphone, record your own samples, downsample them to 2205 Hz, then check how the algorithm generalises to your voice.

   Optional: using clever feature transformations, you can achieve better generalisation. Try to make your algorithm as general as possible by transforming the data or recording/finding new datasets! This usually requires domain knowledge, so it is not the scope of this course, but it has applications in data mining, for example.