**Machine Learning Implementation of Email Spam Detection and Facial Landmark Recognition**

**Task 1: Spam Detection** [**task1 main.py**](task1%20main.py)

Preprocessing

The only preprocessing I deemed necessary for spam detection was to turn numbers into the term “NUM”. This is important, because the presence of numbers at all is more indicative of the text classification than what the individual digits are. This check is done to each word individually, in order.

I had considered removing definite articles and pronouns, as they might dilute the more information-rich words decreasing accuracy, but the presence of good grammar is indicative of non-spam emails, so they were kept in. I also considered setting each word to lowercase, however this made a 0% difference in the accuracy when tested.

Feature representation

I used a “bag of words” approach, that predicts accuracy by comparing the proportion of spam/not spam emails that word is featured in. Each word as it is discovered, is put into both a spam and notspam dictionary, and then the frequency is adjusted in the appropriate one. A counter increases for the total number of words in spam/not spam emails, so we can find the proportion of emails that contain this word.

Prediction method

Spam detection is a binary classification task. My approach uses supervised learning, meaning it learns from a given set of labelled test data and applies features from it to predict unlabelled data. It doesn’t use a loss function, because it doesn’t learn by minimising error, instead by classification and key feature detection.

Results

On a training set of 2700, and a validation set of 919 emails, my algorithm achieves an accuracy of 96.95%. This was calculated by taking 919 of the labelled training emails, comparing the label to the predicted value given by is\_spam(), and dividing the number of correct predictions by 919.

This value was affected 0% by altering preprocessing to set all words to lowercase, and since my approach is deterministic, it returns the same value every time.

Failure cases and potential biases

The training data given does not include ‘leetspeak’, and is only written in English. If a word has never been read before, then it applies to neither spam or not spam. The program defaults to ‘not spam’, since a false positive is worse logically than a false negative, and so emails that have these features will often be flagged incorrectly as not spam. However, given enough training data in different languages or with these workarounds, the false negative behaviour would be learned and would lead to a higher accuracy.

Individual emails can be tested at any time by running is\_spam() on any given string. Below are the inputs and outputs of two spam emails, one written in ‘leetspeak’ and one written in Spanish, to show these points of failure.

A white background with green text

AI-generated content may be incorrect.

**Task 2: Face Alignment** [**task2 main.py**](task2%20main.py)

Preprocessing

The ideal image for feature detection would:

1. Have no background,
2. Be grayscale,
3. Be unobstructed,
4. Be filtered to only show contrast lines.

Because using image detection to remove background noise is both paradoxical and would only improve performance marginally, we will not be removing them and the CNN will have to adapt. Therefore, the full preprocessing is:

1. The kernel in Figure 1 is applied, to show only contrast lines
2. The image is resized to 128x128, to reduce the number of operations 4x
3. The image is made monochrome to show only pixel intensity and reduce overfitting
4. The intensity value for each pixel is divided by 255, so the intensities range from 0-1.

Image representation

The image is passed as a numpy array of shape (1, 128, 128, 1), which was calculated to fit the format (1 image, 128px wide, 128px tall, 1 brightness value). This layout was chosen because it is the arrangement used by TensorFlow. Each “pixel” is a 32-bit float between 0-1 representing its intensity, calculated grayscaling and dividing each pixel by 255.

Prediction method

The two algorithms I considered were SIFT and a CNN. SIFT is excellent for finding patterns, and is resilient to rotations and variable contrast, but cannot be “trained” in the way that CNNs can. While using every bit of training data as a template for every image would probably work, the efficiency of CNNs meant they were a more appropriate choice. This also means the model only needs training once before it will work on any image presented.

The loss function I chose was Mean Squared Error (MSE), as is standard for computer vision tasks. The only regularizations were the preprocessing steps, which should allow every image to be compared regardless of relative intensity, scale or colours.

Designs / Parameters

Systematically I tried batch sizes of 16, 32, 48 and 64, and recorded the loss at each epoch up to 50. This data can be found in ‘<batch\_size> lossdata.xlsx’. Generally, smaller batch sizes will lead to a higher accuracy, but can lead to overfitting if done excessively. I did not factor the time taken to train or predict for each batch size, because this has a negligible effect on performance between tested values when considering individual pictures. As per the results shown in Figure 2, the final parameters were determined as 50 epochs and a batch size of 64.

The model itself has an input layer of shape (1, 128, 128, 1), ie one brightness score for every pixel. There are 4 hidden layers, each analysing the image at one half the resolution of the layer before it. Each layer identifies features at a broader scale, allowing for a higher level of spatial awareness. Finally, it is flattened, one final dense layer of size 256 is applied, and then an output layer of 10 values (x0, y0, …, x4, y4). This later will get manipulated with the cardinalise() function, which just turns these values into coordinate pairs of the points they represent.

Between each layer, Rectified Linear Units (ReLU) is applied, meaning negative values are zero’d. I chose this because I had to run it several times, and ReLU trains in a shorter timeframe than sigmoid functions.

Result Analysis

The amount of time taken for the model to train and predict is correlated to the number of epochs. Although training and prediction time is of low concern as mentioned above, I also measured these to find the point at which increased training becomes detrimental overall, as another efficiency metric.

The relative ‘score’ of each prediction has been calculated as -log(loss), resulting in visibly comparable positive numbers. When plotted, we can see that smaller batch sizes begin overfitting very quickly, but tend to higher accuracy. The peak score was at a batch size of 32, but that begins a pattern of overfitting at 15 epochs. Higher batch sizes showed much more stable high scores, seemingly with the best balance between accuracy and reliability at a batch size of 64 and all 50 epochs.

The figures for the results of these tests are in files named ‘<batch size> lossdata.xlsx’, and the graphs are shown below.

Figure 2: Y-axis: Relative score. X-axis: Epoch count.

Y-axis: Relative score, calculated -log(loss). X-axis: Epoch count.

A graph with a line

AI-generated content may be incorrect.A graph with a line

AI-generated content may be incorrect.

Figure 2.1: Overfitting @12, Peak @47 Figure 2.2: Overfitting @16, Peak @43

A graph with a line

AI-generated content may be incorrect.A graph with a line

AI-generated content may be incorrect.

Figure 2.3: Overfitting n/a, Peak @50 Figure 2.4: Overfitting n/a, Peak @50