COMP 6630-001: Final Project Report

Matthew Freestone*
Auburn University
United States of America
mf@auburn.edu

Will Humphlett
Auburn University
United States of America
wh@auburn.edu

Matthew Shiplett
Auburn University
United States of America
mss0033@auburn.edu

ABSTRACT

This project proposal will focus on an outline for using a multi-layer perceptron to perform sentiment analysis on reviews. The purpose of this model is to create an automated function capable of English text sentiment analysis. After training on a dataset of labeled amazon reviews, the model will be able to determine if review's text sentiment is positive or negative, and how much. The model's accuracy will be confirmed by testing on text that is not shown during training. Once trained, this model could be used by various entities to expedite sentiment analysis on large bodies of English text based feedback. Examples include corporate entities running analysis on feedback of their products is mostly positive or negative, or instructors running analysis on feedback of their courses.

CCS CONCEPTS

• **Theory of computation → Machine learning theory**; • **Computing methodologies → Machine learning**.

KEYWORDS

multi-layer perceptrons, machine learning, sentiment analysis, natural-language processing

ACM Reference Format:
Matthew Freestone, Will Humphlett, and Matthew Shiplett. 2022. COMP 6630-001: Final Project Report. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEM DESCRIPTION

1.1 Primary Application Domain

This project used a multi-layer perceptron to classify English text based on sentiment. The primary domain is on English text, with inputs being a dictionary of words, represented by a 1D vector, containing integer counts of the number of times a word in the dictionary appears within a selected body of text. The output is a number {1, 2, 3, 4, 5} representing a scale of how positive the sentiment is. A 1 represents very negative sentiment, and a 5 represents very positive sentiment.

1.2 Application of an MLP

Multi-layer Perceptrons (MLP) can be used to accomplish this classification task. With the use of multiple layers, we are able to achieve highly non-linear decision planes, allowing the model to separate text into categories of negative, neutral, and positive, or further into a sentiment scale from 1 to 5. The perceptrons will be fully connected at each layer, and each will learn weights based on the training data. Back-propagation will be used to update the weights at each level, and Stochastic Gradient descent will be used to converge upon a model with high accuracy.

In the case of the three-class problem, we have two possible implementations to try. We could create a single MLP with 3 classes { Positive, Negative, Neutral }. We could also create a two-stage system, with a first MLP making a binary decision between { Neutral, Some Sentiment }, and if that MLP determines that there was a sentiment, a second MLP will make a choice between {Positive, Negative}.

1.3 Potential Applications

Sentiment analysis is a valuable tool for data scientists attempting to analyze text. Ideally, the model will be able to quickly classify large amounts of data reasonably accurately, providing the users with a way to determine the sentiment of text. Additionally, corporate entities looking to perform sentiment analysis on a large body of feedback can use such a model to quickly gain insight into the sentiment of the feedback.

1.4 Challenges

Due to the large dictionary of words found in English, even a limited subset, the MLP will need to have inputs equal to the dictionary size. Even for a small network, the number of connections will be large in order to accommodate the large input space. The large size of the network can mean training and backpropagation will be computationally intensive. Compounding these issues, given our input mode, the order of words will be disregarded. The proposed dictionary vector will only model the frequency of appearance of words, but not the order in which they appear.

Determining how to pre-process text for the model will be challenging, as there are a large variety of methods to do this. We plan to try a number of them, including TF-IDF and N-Grams.

1.5 Dataset

1.5.1 The original Plan. Before starting our implementation and actual runs, we planned to use a labeled dataset from Kaggle containing 50,000 movie reviews from IMDB. The data points were labeled with "positive" and "negative" corresponding to the sentiment of the text. The description of the dataset describes the points as "highly polar," meaning that we are likely to find a model that separates the data well. However, to fulfill the project requirements, we would need to web scrape IMDB for neutral reviews, which was a difficult task. We decided to look elsewhere for data. The IMDB dataset we initially planned to use can be found here: https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews

*Project Coordinator

To solve the problem of having no neutral reviews, we planed to make use of a dataset that consists of 1.5 million tweets labeled with POSITIVE, NEGATIVE, NEUTRAL sentiment. However, the actual dataset labels did not match the claimed labels. This meant that the dataset did not in fact have neutrally labeled data. Following these discoveries, it was decided to pivot to an Amazon reviews dataset.

This dataset is comprised of Amazon product reviews, broken down into different categories, from reviewers who have left more than 5 reviews. Each data entry contains the user's name, product id, review text, and the number of stars on the review from 1-5. The datasets can be found here: https://jmcauley.ucsd.edu/data/amazon_v2/categoryFilesSmall/. For our purposes, we used the following categories:

- CDs_and_Vinyl_5
- Cell_Phones_and_Accessories_5
- Clothing_Shoes_and_Jewelry_5
- Electronics_5
- Home_and_Kitchen_5
- Kindle_Store_5
- Movies_and_TV_5
- Sports_and_Outdoors_5

Not all categories were used simultaneously for training. In order to train and validate the model, a subset of the data was loaded and trained on. The results presented later in this paper resulted from training on a combination of the following subsets:

- CDs_and_Vinyl_5
- Cell_Phones_and_Accessories_5
- Kindle_Store_5
- Sports_and_Outdoors_5

Additionally, unaltered, these datasets are highly skewed towards 5-star reviews, insofar as that the 5-star reviews often make up more than half of the data-set. This meant that work had to be done to combine and balance the datasets in order to ensure a more equal number of reviews per rating category. Finally, we tested on both a balanced dataset with all 5 labels, as well as modifying the dataset to only contain 3 labels, mapping 2 stars to a label of 1 and mapping 4 stars to a label of 5.

1.6 Evaluation Metrics and Benchmarking

The classifier's purpose is to perform sentiment analysis, and given that it is a machine classifier it should be able to perform sentiment analysis significantly faster than a human. Therefore, to demonstrate its usefulness, the model will be applied to a large body of previously uncategorized text, producing sentiment labels significantly faster than a human could categorize the same body of text.

In order to determine how well our model performs against others, we will create use models, like Random Forest, and train them on the same dataset. We will also try Tensorflow's Neural Network and scikit-learn's neural network to determine how well our implementation compare. We will use several metrics to determine how our model holds up, including Accuracy, Precision, Recall, and F1 Score. However, we will consider accuracy to be the single most important metric for this project. The formulas to compute these metrics for each label are as follows:

Accuracy: $\frac{\text{number of correct predictions}}{\text{total number of predictions}}$

Precision: $\frac{\text{number of true positives}}{\text{number of true positives+number of false positives}}$

Recall: $\frac{\text{number of true positives}}{\text{number of true positives+number of false negatives}}$

F1 Score: $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision+recall}}$

2 MLP THEORY AND IMPLEMENTATION

2.1 Mathematical Theory

Let's say we have a feedforward neural network with multiple layers, where each layer l has n_l units. Let $w_{i,j}^{(l)}$ denote the weight from unit i in layer l to unit j in layer $l+1$, and let $b_j^{(l)}$ denote the bias of unit j in layer $l+1$. Let $a_i^{(l)}$ denote the output of unit i in layer l, and let $z_j^{(l+1)}$ denote the weighted sum of the inputs to unit j in layer $l+1$. Then we can express the output of unit j in layer $l+1$ as:

$$z_j^{(l+1)} = \sum_{i=1}^{n_l} w_{i,j}^{(l)} a_i^{(l)} + b_j^{(l)}$$

And the output of unit j in layer $l+1$ after applying the activation function f is:

$$a_j^{(l+1)} = f\left(z_j^{(l+1)}\right)$$

To perform backpropagation, we need to compute the gradient of the cost function with respect to the weights and biases in the network. Let's say the cost function is C, and let y be the true label for a given input. Then the gradient of the cost function with respect to the weights and biases in the network can be computed as follows:

$$\frac{\partial C}{\partial w_{i,j}^{(l)}} = \frac{\partial C}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial w_{i,j}^{(l)}} = \delta_j^{(l+1)} a_i^{(l)}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial b_j^{(l)}} = \delta_j^{(l+1)}$$

where $\delta_j^{(l+1)}$ is the error of unit j in layer $l+1$ and is given by:

$$\delta_j^{(l+1)} = \frac{\partial C}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}}$$

This method using δ for updating parameters works well for the middle hidden layers, but for the first and last layers, we must use slightly different values.

The value of δ for the last hidden layer depends on the error function used. A common choice is Mean Squared Error, given by

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where n is the number of samples, y_i is the true target value for the ith sample, and $\hat{y}_i$ is the predicted value for the ith sample.

To calculate the derivative of the MSE with respect to the prediction, we can use the chain rule. The derivative of the MSE with respect to the predicted value $\hat{y}_i$ is given by:

$$\frac{\partial MSE}{\partial \hat{y}_i} = \frac{\partial MSE}{\partial (y_i - \hat{y}_i)} \cdot \frac{\partial (y_i - \hat{y}_i)}{\partial \hat{y}_i} = -2(y_i - \hat{y}_i)$$

Note that this derivative is just the negative of the difference between the true target value and the predicted value.

Therefore, if the cost function is the mean squared error, we can calculate the following as the value of δ for the last hidden layer:

$$\delta_j^{(l)} = \left(a_j^{(l)} - y\right) \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$

Once we have calculated delta for the last hidden layer, we can use it to update the weights and biases in that layer using the equations for backpropagation given above.

For the first hidden layer, we simply use X in place of a_i^l when updating the weights.
This completes the derivation of backpropagation.

2.2 Implementation

2.2.1 Python for Machine Learning. To implement the Multi-Layer Perceptron, our team decided to use python for a variety of reasons.

(1) Python is a high-level, dynamically typed, and interpreted programming language, which makes it easy to write and debug code. This is particularly useful when developing and testing machine learning algorithms, which can be complex and require many iterations to get right.
(2) Python has a large and active community of users, which means there is a wealth of resources and support available for learning and using Python for machine learning. There are many tutorials, forums, and other online communities where Python users can share knowledge and ask for help with their machine-learning projects.
(3) Python has a rich ecosystem of libraries and frameworks for machine learning, which makes it easy to perform a wide range of tasks in machine learning. Some of the most popular Python libraries and frameworks for machine learning include NumPy, Pandas, scikit-learn, TensorFlow, and PyTorch. These libraries and frameworks provide a wide range of functionality, from data manipulation and visualization to machine learning algorithms and deep learning models.

2.2.2 Parameters. In our implementation of a Multilayer Perceptron, we first initialize a class representing a single model. We created the following parameters for use in the model.

- `epochs`: int
  - Number of epochs to train the model
- `lr`: float
  - Learning rate
- `hidden_layers`: Sequence[int]
  - A sequence of integers representing the number of neurons in each hidden layer
- `regularization`: str, default=None
  - The regularization function to use. Can be either None, "l1" or "l2"
- `reg_const`: float, default=0.0
  - The regularization constant
- `activation`: str, default="sigmoid"
  - The activation function to use. Can be either "sigmoid", "tanh" or "relu"

2.2.3 Fit Function. Our model is trained using the `fit` function, the accepts the parameters below.

- `X`: np.ndarray
  - Input data of shape (n_examples, n_features)
- `y`: np.ndarray
  - Output data of shape (n_examples,)
- `X_val`: np.ndarray, optional
  - Validation input data of shape (n_examples, n_features)
- `y_val`: np.ndarray, optional
  - Validation output data of shape (n_examples,)
- `batch_size`: int, default=1
  - Size of the batch to be used for training
- `continue_fit`: bool, default=False
  - If True, the model will continue training from the last epoch

The function will perform some operations to be sure that it passed valid parameters, then use a `LabelBinarizer` from scikit-learn to convert the label data into a usable form.
The model then initializes weights for all layers by randomly picking values from a standard normal distribution.
Then, the model performs backpropagation in batches over epochs. That code is shown here.

```
for epoch_num, lr in tqdm(self.epochs(), total=self.num_epochs):
    train_loss = 0
    for i in range(0, X.shape[0], batch_size):
        X_batch = X[i:i+batch_size]
        y_batch = y[i:i+batch_size]
        dJdB, dJdW, c_loss = self._backprop(X_batch, y_batch)
        train_loss += c_loss

        self._biases = [b - lr * db for b, db in zip(self._biases, dJdB)]
        self._weights = [w - lr * dw for w, dw in zip(self._weights, dJdW)]
    num_batches = X.shape[0] // batch_size
    self.train_loss_curve.append(train_loss / num_batches)
    if use_val:
        val_loss = self._calc_loss(y_val, self._fast_forward_pass(X_val))
        self.val_loss_curve.append(val_loss)
```

We perform this loop based on the number of epochs specified in the constructor. We also created our learning rate calculations in such a way that allows customization of the `self.epochs()` function to create a variable learning rate.
We also used `tqdm`, which is a package built into python that creates progress bars. This allows us to determine how long training might take without simply waiting, which must be done for `sklearn`'s implementation.
When the backpropagation function is called, we use a batch of X and y data, allowing the function to use numpy's vectorization for better performance.
At each epoch, we also record the loss across the training data and validation data. This allows up to plot that data later to determine if overfitting occurred.

2.2.4 Backpropogation Function. The fit function makes use of the `_backprop` function, which is shown below.

```
def _backprop(self, X: np.ndarray, y: np.ndarray) -> tuple[list[np.ndarray],
    # initialize empty lists to store the gradient of the biases and weights
    dBias = [np.zeros(b.shape) for b in self._biases]
    dWeights = [np.zeros(w.shape) for w in self._weights]
    n_samples = X.shape[0]

    # do a forward pass to get the activations and z values
    layer_raw = [] # stores the weighted sum of inputs for each layer
    layer_activations = [] # stores the output of each layer
    a = X # input layer
    for i, (b, W) in enumerate(zip(self._biases, self._weights)):
        # compute the weighted sum of inputs for this layer
        z = a @ W + b.T

        # apply the activation function to the output of this layer
        if i < self._num_layers - 2:
            a = self.activation(z)
        else:
            a = self._output_activation(z)
        # store the raw output and the activated output of this layer
        layer_raw.append(z)
        layer_activations.append(a)
```

The above is the first part of the actual implementation we created for backpropagation.

In this step, we initialize empty arrays to hold the changes in both weight and bias. Next, we do a 'forward pass' on the data. This means that we run the data through each layer, one at a time, recording the result of the weighted sum and the activated value. These will be used later to calculate δ.

Also, note that we apply a different activation to the last layer. In muti-class classification tasks, we used softmax. It maps a vector of real-valued inputs to a vector of values in the range [0, 1] that sum to 1. This allows the outputs to be interpreted as probabilities, with the output corresponding to each class representing the probability that the input belongs to that class.

The formula is given by:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$$

where z is a vector of real-valued inputs and n is the number of elements in the vector.

Performing this transformation on our last activation allows us to compare the results to a one-hot representation of our label and perform backpropagation easily.

```
# calculate the loss
loss = self._calc_loss(y, a)

# index of the last hidden layer
last_hidden = self._num_layers - 2

# compute the error at the last hidden layer
delta = (layer_activations[last_hidden] - y)

# compute the gradient of the biases and weights for the last hidden layer
dBias[last_hidden] = np.mean(delta, axis=0)
dWeights[last_hidden] = layer_activations[last_hidden-1].T @ delta
```

First, we calculate the loss on the current batch of data. The last layer of activations is the result of the 'prediction' from the forward pass, and we have the correct labels y, so the calculation is relatively inexpensive.

We then begin our backward calculation of δ and updates for weights and biases.

For the last hidden layer, we compare the result directly to the actual value, using the same method as the derivation of backpropagation showed for Mean Squared Error. When calculating change in bias, we average all of the examples to determine a change. Because we are doing operations in a vectorized way with numpy, operators may look slightly different than those in pure mathematical derivation.

```
# for all hidden layers except the first and last,
# compute the gradient of the biases and weights
for L in range(last_hidden-1, 0, -1):
    # compute the error at this layer
    delta = (delta @ self._weights[L+1].T) * self.dActivation(layer_raw[L])

    # compute the gradient of the biases and weights for this layer
    dBias[L] = np.mean(delta, axis=0)
    dWeights[L] = layer_activations[L-1].T @ delta

# for the input layer, compute the gradient of the biases and weights
# using the inputs, rather than the previous layer
delta = (delta @ self._weights[1].T) * self.dActivation(layer_raw[0])
dBias[0] = np.mean(delta, axis=0)
dWeights[0] = X.T @ delta
```

For the middle layers, the computations look much the same as the last layer, but we use the δ instead of the actual labels.

For the first layer, we use the actual inputs to update weights instead of the previous activations.

2.2.5 Predict Function. After the model has been trained, it must be able to make predictions on new data. The below code is our implementation of that function.

```
def predict(self, X:np.ndarray) -> np.ndarray:
    '''
    Predicts the output for the given input
    X: Input data of shape (n_examples, n_features)
    returns: Output data of shape (n_examples, )
    '''
    curr = self._fast_forward_pass(X)
    if self.output_layer == 1:
        curr = curr.ravel()
    return self._yencoder.inverse_transform(curr)
```

This code performs a forward pass on the data, then reverses the encoding done on the original data. In the multi-class case, this means converting a one-hot representation back into a numerical value representing the class the example belongs to.

The `fast_forward_pass` function used is shown below.

```
def _fast_forward_pass(self, X: np.ndarray) -> np.ndarray:
    curr = X
    for i in range(self._num_layers - 1):
        curr = curr @ self._weights[i]
        curr += self._biases[i].T
        if i < self._num_layers - 2:
            curr = self.activation(curr)
        else:
            curr = self._output_activation(curr)
    return curr
```

This uses the same forward-pass calculations as the backpropagation function, but it does not store the intermediate values. In the case of prediction, we do not need to store those values to learn, and only need to return the final activated result.

2.2.6 Other Utility Functions. Beyond the most basic and required functions, we also chose to implement a few other methods for convenience.

- `model.score(X, y)`
  - This function uses a trained model to make predictions on `X`, then compare those predictions to the actual values given in `y`. It returns the accuracy on those points. `scikit-learn` uses this function on their models, and our group found it very useful for benchmarking, so we decided to implement it in our own classifier.
- `model.plot_loss()`
  - This function returns a `matplotlib.pyplot` line graph of how the training loss (and validation loss, if those datasets were provided) changed over epochs.
  - This can give an indication of overfitting if the validation loss is increasing while training is decreasing.
  - This was also useful for determining if the model has converged without additional computation. After running, the user can see the validation loss is still decreasing, and decide to run more training, or stop there.

2.3 Implementation Challenges

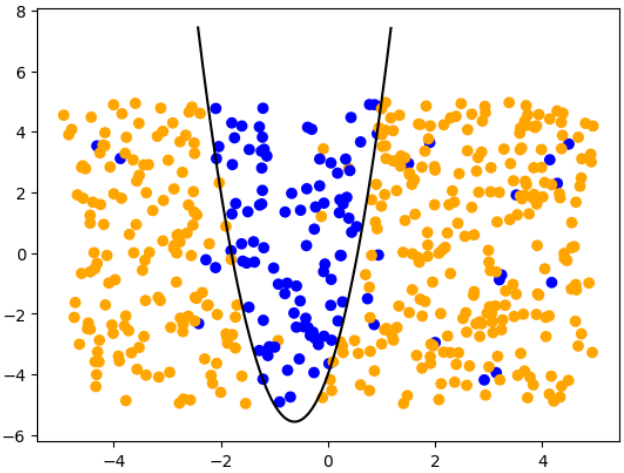
During the process of implementing the Multi-layer perceptron, our team encountered a number of challenges.

2.3.1 Diagnosing Incorrect Initial Implementations. While creating our first versions of the multilayer perceptron, we found it difficult to determine if it was even functional. Training the model on the dataset we picked would take a prohibitively long time, and we wanted a problem that was solvable by learning a non-linear decision plane in a reasonably short amount of time. Existing methods were less graphical than we had hoped, as we'd like to be able to visually confirm the effectiveness of the model.
To realize this goal, we created the `TwoDimProblem` class. This class creates a binary classification problem with 2 features, and it assigns labels based on a separator of user-specified degree. Having only two features allows the data to be easily represented on a scatter plot.
This class can generate data using the following parameters:

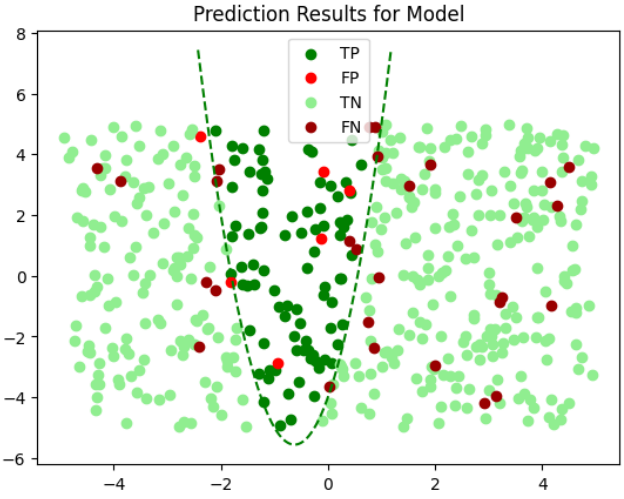
- `soln_rank: int`
  - The degree of the polynomial that separates the positive and negative points in the data.
- `noise_frac: float`
  - The fraction in the interval [0,1] of how many points are on the wrong side of the separator.
- `samples: int`
  - The number of points to generate

To use the generator, use the `create_data` method, then call `plot_data`. This will create points and labels, as well as show it graphically along with the best possible separator.

```
p = TwoDimProblem(5)
X, y = p.createData(2, 0.05, 500)
p.plotData(show_seperator=True)
✓ 0.1s
```



After training a model, call `p.plot_pred(y_pred, show_correct=True)` to get a scatter plot indicating what the model labeled as True Positive, False Positive, True Negative, and False Negative. This will also superimpose the correct separator.



2.3.2 Batching data to improve speed. After we confirmed that the initial model worked, we attempted to train it on a smaller subset of our dataset. Training took a long time, especially compared to scikit-learn's implementation of a Multi-layer perceptron – nearly 100 times faster. After digging through scikit-learn's source, it became clear that this difference was due to batching backpropagation. Instead of calculating a gradient on each point individually and updating weights, we instead calculate the gradient for multiple points, average them, then update weights.

This approach comes with a variety of benefits over training individually or on the entire dataset each iteration.

(1) Improved convergence speed: By training on smaller batches of data at a time, the network can make more frequent updates to its weights, which can lead to faster convergence.
(2) More efficient use of computational resources: When using mini-batching, the network can make better use of parallelization and vectorization, which can make the training process more efficient and faster.
(3) Better generalization: By training on smaller batches of data, the network can avoid overfitting to the training data and learn to generalize better to new examples.
(4) More stable gradients: When the batch size is too small, the gradients can be noisy and unstable, which can make training difficult. On the other hand, when the batch size is too large, the gradients can become too small and the network may not learn effectively. Using mini-batching allows the network to find a good balance between these two extremes.

To implement this functionality, we changed the backpropagation calculations to be highly vectorized using NumPy, and this greatly improved training speed without heavily affecting results.

2.3.3 Running on the GPU. While batching greatly improved training time, we noticed that packages like `Tensorflow` and `PyTorch` run much faster than ours or `scikit-learn`'s due to their ability to run on the GPU using CUDA. To attempt to accomplish this goal, we used Numba|.

Numba is a just-in-time (JIT) compiler for Python that can be used to speed up the execution of Python code on the CPU or GPU. It works by allowing us to annotate your Python code with type information, which Numba uses to generate optimized machine code at runtime. This allows us to write Python code that runs at near-native speeds.

Writing the code to be JIT compiled for the CPU was reasonably easy. Some variable types had to be more explicitly written and others had to be modified to not determine a type at runtime.

This change, however, did not yield significant performance improvements. The compilation introduced some additional overhead to our model that canceled out any performance increase that might have occurred. Additionally, many NumPy functions are written in C or C++, and they are already compiled to machine code, which can be executed directly by the CPU. This means that NumPy code is often already running at near-native speeds, and there may not be much room for additional performance improvements through JIT compilation.

To get better results, we would need to use the GPU, but that came with challenges. Using GPU acceleration can require a significant amount of programming effort and expertise. To use Numba effectively, we needed to have a good understanding of the underlying GPU architecture and how to write code that is well-suited for execution on the GPU. This was a steep learning curve, and we found that we would need to spend a significant amount of time reading documentation and experimenting with code to get good performance.

Due to these concerns, we opted to stick with the slower option that we had already created. We had to weigh the time and effort required to learn how to use it effectively against the potential benefits. In the end, we decided that the potential benefits were not worth the time and effort. Future work could involve implementing this functionality.

2.3.4 Memory Size. The datasets we used did fit in memory, but we encountered problems after trying to preprocess our string data into a numerical bag-of-words representation. After processing the data and attempting to store it in a NumPy array, we were greeted with the following error.

```
MemoryError: Unable to allocate 429. GiB for an array with shape (3410019,) and data type <U33759
```

As shown in the error, storing the bag-of-word vectors in a NumPy array was out of the question for our mere-mortal machines. However, Scipy provides a matrix alternative that works well for the problem we were facing: Sparse matrices.

Sparse matrices are more memory efficient than NumPy arrays because they only store non-zero values, rather than storing every single element in the matrix, including the zeros. This means that for large matrices with a significant number of zero values, a sparse matrix can take up much less memory than a NumPy array.

For example, if we have a matrix with 10^6 elements, and only 100 of those elements are non-zero, then a NumPy array would take up $10^6 * 8$ bytes (assuming double precision floating point values) of memory, while a sparse matrix would only take up $100 * 8$ bytes of memory to store the same information. This can be significant savings in memory, especially for matrices like our bag-of-words vectors that are mostly zeros.

The sparse matrix result of preprocessing was small enough to fit in memory, and `scikit-learn`'s implementation of `CountVectorizer` conveniently already returns this type. We had to modify our implementation of the MLP to handle the new data type, but this was a fairly trivial task.

3 SOFTWARE DOCUMENTATION

3.1 Repository

github.com/wumphlett/ML-Project

Code including implementation, testing, and benchmarking can be found in the repository above. To start, please clone the repository to your local machine.

3.2 Setup

The only file of interest is project.ipynb. Running it assumes you have conda installed and have activated the base environment. From there, execute the cells sequentially to set up your environment, witness implementation, analysis, and view comparisons to other similar models.

3.3 Tutorial

Each of these sections will correspond to a section of the notebook (project.ipynb) and will explain its purpose and how to use it. **It is recommended you execute the cells in order.**

3.3.1 Notebook Setup. This cell simply configures the notebook and bootstraps your local copy. Run it before any other cells and no changes should be required.

3.3.2 Environment Setup. This cell is responsible for downloading the configured review datasets. By default, this will be CDs and vinyls, cell phones, kindles, and sports product reviews. Other datasets are available and are fully defined in src/bootsrap.py. Change the entries in the DATASETS list in this cell and re-execute to download other datasets.

3.3.3 Preprocessing. This cell is responsible for creating the dataframes from the downloaded dataset files, cleaning up the data, and equalizing the label distribution to make for better training. No changes should be required, but re-execute if other datasets are downloaded or deleted.

3.3.4 Feature Extraction. This cell is responsible for taking the unschematized review text and transforming it into a feature array for use in training. We offer word vectorization natively, but other representations can be implemented here, including but not limited to n-grams and tf-idf. We also convert the five-class (star) review problem into a three-class problem, but this can be commented out should you desire. Splitting the dataset into training, testing, and validation sets also occurs here, and the weights controlling the length of each subset are configurable.

3.3.5 Implementation. This cell contains the full implementation of our Multi-Layer Perceptron. Multiple activation functions are offered as well as regularization. Initialize an MLP with num_epochs, lr (learning rate), regularization, reg_const (regularization constant), hidden_layers, and activation. Once this is done, your instance of the MLP class offers three public methods; fit, predict, and score. Fit takes as input an X training data set and a y training label set and will be used to train the model. Optionally, it takes as arguments additional validation data to plot validation loss and a batch size to allow for batched gradient descent. Predict takes as input an X testing data set and will return the predicted y testing labels. Score takes as input an X validation data set and a y validation label set and will return the accuracy score that results from predicting based on the X validation and comparing to the y validation.

3.3.6 Two Dimension Problem. This group of cells is responsible for defining and testing our MLP with a two-dimension problem. A random dataset is created that is separable given a non-linear decision plane, some noise is applied, and then this data is fed into our MLP to determine if it can learn the separator. This gives confidence that our MLP is capable of what it is meant to do.

3.3.7 Training. This group of cells is responsible for training our MLP given a variety of hyperparameters. An editable dictionary defines the axises of a hyperparameter matrix and every possible combination will be used to train the model. The hyperparameters that produce the highest accuracy on a separate testing dataset will be returned and recommended for use. Dependant on the size of the hyperparameter matrix, the execution of this cell takes a long time, so data/train.log is written during execution to give insight into the training progress. Once execution is complete, data/train.csv is written that contains the results of training and the testing accuracy discovered. Finally, the model given the best parameters found is trained to prepare for analysis.

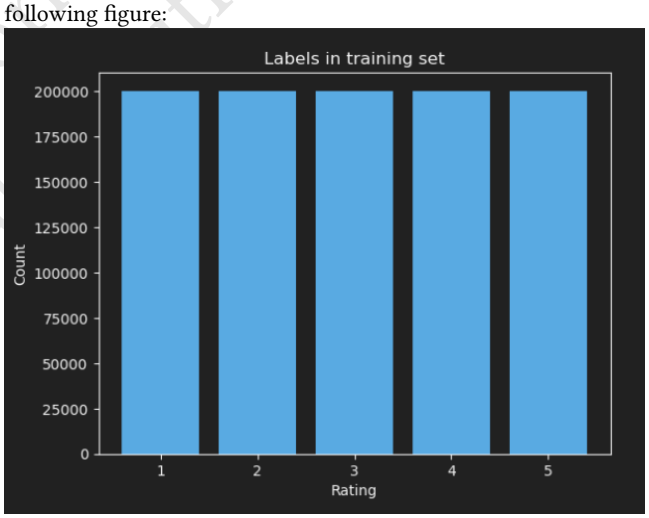
3.3.8 Performance. This cell is responsible for performing analysis on our MLP and determining the metrics necessary to evaluate it. By default, accuracy, precision, recall, and f1 score are provided, as well as a set of plots demonstrating the loss per epoch and the accuracy per training, testing, and validation datasets. Other performance measures can be added here should you wish.

3.3.9 Benchmarking. This group of cells is responsible for training other pre-existing models on our dataset for the purpose of comparing performance. By default, we offer `scikit-learn`'s `MLPClassifier`, `scikit-learn`'s `RandomForestClassifier`, and a `Tensorflow NN`.

4 BENCHMARKING AND OBSERVATIONS

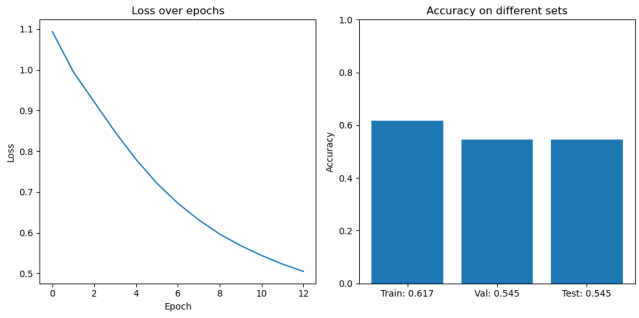
4.1 Benchmarking Overview

For benchmarking a number of models were trained using our MLP class and back-propagation implementation. Multiple models where trained by varying the hidden layers, both in terms of how many layers and the number of neurons per layer, as well as the number of training epochs. `scikit-learn`'s `MLPClassifier`, which can be found at: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html, will serve as a baseline of comparison for the models trained using our implementation. The balanced dataset used to train our models is visualized in the following figure:



4.2 5 Class Problem

4.2.1 Benchmark Model: Scikit-Learn MLP. For an overall benchmark comparison, a sk-learn MLP model was trained. This model will serve as a baseline of comparison for our models. For all models, two primary figures will be provided. The first figure contains two plots, the first plot is a Loss vs Epochs plot. The second blot is a bar graph showing the overall accuracy for each set between training, testing, and validation. The second figure is a table that will contain the precision, recall, and F1-score, by rating category, that the model achieved.
The benchmark model loss curve is presented alongside the training, validation, and testing accuracy:



From the above plots, it can be observed that the model achieved a testing accuracy of approximately 54.4%. This testing accuracy will serve as the target for our models to achieve. Additionally, it can be observed that the model began reaching loss convergence at approximately 12 epochs. This figure will be used as a base indicator for the number of epochs our models should train for.
The precision, recall, and F1 scores, by category, for this model are as follows:

Category	precision	recall	f1-score	support
1	0.62	0.66	0.64	25170
2	0.45	0.48	0.46	25012
3	0.49	0.39	0.44	25041
4	0.49	0.47	0.48	24973
5	0.64	0.73	0.68	24995

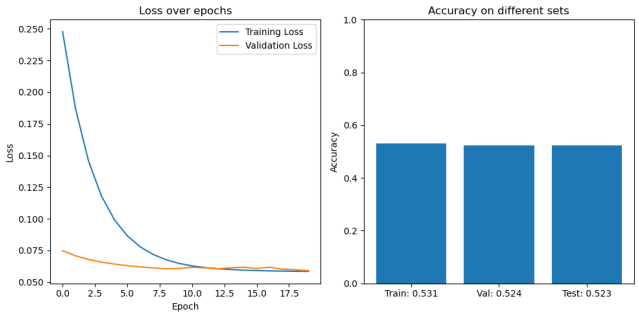
	precision	recall	f1-score	support
accuracy			0.54	125191
macro avg	0.54	0.54	0.54	125191
weighted avg	0.54	0.54	0.54	125191

From this table, it can be observed that The model performs best in the 1 and 2-star categories. Achieving over 60% precision and recall for these categories, in fact achieving 73% recall on the 5-star category. These scores will serve as the baseline for comparison for our model.

4.2.2 Our First model. The first model presented is an MLP using our implementation with the following hyper-parameters:

- Epochs : 20
- Learning rate: 0.2
- Hidden layer(s): [50, 20]
- Activation function: `Logistic` (`Sigmoid`)
- Regularization: `L2`
- Regularization constant: 0.0001
- Batch size: 200

As a note, the Hidden Layers should be interpreted as: each entry in the list represents a hidden layer, and the integer of that entry represents the number of neurons in that layer. For the models, the loss curve is presented alongside the training, validation, and testing accuracy:



It can be observed from the above that both the training and validation losses converge around 10 epochs. Additionally, it can be observed that the over-fitting of the training data appears to be minimal as the training, validation, and testing see similar accuracy, and a divergence between the training and validation loss is not observed after convergence. While just above 50% accuracy does not at first appear fantastic, and in the grand scheme it is nothing amazing, it is more than double the accuracy of randomly labeling reviews, which for 5 categories with a roughly equal number of reviews, would be approximately 20%.
The precision, recall, and F1 scores, by category, for this model are as follows:

Category	precision	recall	f1-score	support
1	0.59	0.69	0.64	25170
2	0.39	0.52	0.45	25012
3	0.48	0.32	0.39	25041
4	0.54	0.32	0.40	24973
5	0.62	0.75	0.68	24995

	precision	recall	f1-score	support
accuracy			0.52	125191
macro avg	0.52	0.52	0.51	125191
weighted avg	0.52	0.52	0.51	125191

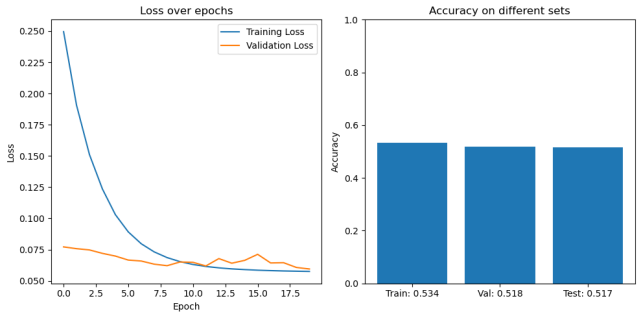
From this table, it can be observed that the model has the strongest precision, recall, and F1 scores in the 1 and 5-star categories, similar to the sk-learn model. Given that these are the most polar categories and therefore are likely to have features that strongly distinguish them, these results seem reasonable to the author(s). The next strongest categories are the 2 and 4-star categories, given that these categories will likely have an overlapping language with the 1 and 5-star categories respectively, the relatively lower performance is likely due to the difficulty with finding features that correctly predict these categories. The model performs the worst in the 3-star category. While still performing better than chance in precision, it scores especially low in recall for the 3-star category. This is likely due to there being fewer easily learnable features in our bag-of-words feature space which correlate highly with the 3-star category, but not other categories.
Overall, when compared to the baseline model our model performs within single-digit percentage points in overall categories. Most notably, the largest difference in performance comes in the performance in the 3 and 4-star categories, where our model does not

score as strongly as the benchmark model.

4.2.3 Our Second Model. The second model presented is an MLP using our implementation with the following hyper-parameters:

- Epochs : 20
- Learning rate: 0.2
- Hidden layer(s): [50, 20]
- Activation function: `ReLU`
- Regularization: `L2`
- Regularization constant: 0.0001
- Batch size: 200

This model makes use of the `ReLU` activation function but has otherwise identical hyper-parameters to the first model presented. The loss curve is presented alongside the training, validation, and testing accuracy:



It can be observed from the above that both the training and validation losses converge, once again, around 10 epochs. Additionally, it can once again be observed that over-fitting of the training data appears to be minimal as the training, validation, and testing see similar accuracy, and a divergence between the training and validation loss is not observed after convergence. In fact, this model performs remarkably similar to the model using `Logistic` activation. Though, at around 15 epochs it can be observed that the training and validation lost begin to diverge a bit, before re-converging at around 20 epochs.
The precision, recall, and F1 scores, by category, for this model are as follows:

Category	precision	recall	f1-score	support
1	0.65	0.59	0.62	25170
2	0.41	0.55	0.47	25012
3	0.50	0.29	0.37	25041
4	0.48	0.32	0.38	24973
5	0.55	0.84	0.67	24995

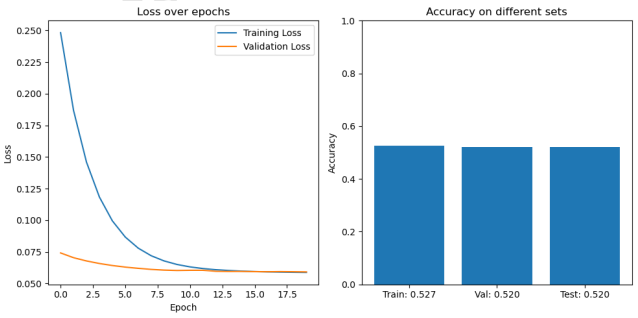
	precision	recall	f1-score	support
accuracy			0.52	125191
macro avg	0.52	0.52	0.50	125191
weighted avg	0.52	0.52	0.50	125191

Once again, this model appears to perform remarkably similarly to the model trained using `Logisitc` activation.
Overall, when compared to the baseline model our second model, similar to our first model, performs within single-digit percentage points in overall categories. Notably, however, our second model does score quite well in recall for the 5-star category, achieving a stellar 84% recall which bests the baseline model's 73% recall.

4.2.4 Our Third Model. The third model presented is an MLP using our implementation with the following hyper-parameters:

- Epochs : 20
- Learning rate: 0.2
- Hidden layer(s): [50, 50, 20, 20]
- Activation function: `Logistic`
- Regularization: `L2`
- Regularization constant: 0.0001
- Batch size: 200

This model makes use of the `Logistic` activation function once again but has more hidden layers. This model was trained to see if having an increased number of hidden layers significantly impacted the results on the 5-class problem. The loss curve is presented alongside the training, validation, and testing accuracy:



It can be observed that this model performs similarly to the first and second models. The third model appears to perform somewhere between the first and second models in terms of accuracy. Once again achieving just below 60% accuracy on the training set and just above 50% accuracy on the validation and testing sets. Notably, the third model does does slightly worse than the first model with fewer hidden layers in all 3 sets. Because the loss for this model had largely converged for both training and validation, it may be that for this problem simply adding more hidden layers of similar sizes does not offer an improvement to performance.
The precision, recall, and F1 scores, by category, for this model are as follows:

Category	precision	recall	f1-score	support
1	0.57	0.72	0.64	25170
2	0.39	0.50	0.44	25012
3	0.48	0.31	0.38	25041
4	0.54	0.31	0.39	24973
5	0.62	0.75	0.68	24995

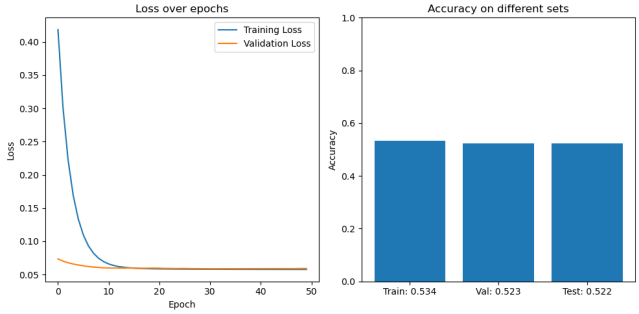
	precision	recall	f1-score	support
accuracy			0.52	125191
macro avg	0.52	0.52	0.51	125191
weighted avg	0.52	0.52	0.51	125191

Once again, our third model performed remarkably similar to our first model. In most categories being within one percent, if not achieving the same score. However, as a note, it does not ever perform worse in any category, if anything offering extremely small improvements to individual precision or accuracy, but achieving the same average and weighted average in all categories.
Overall, when compared to the baseline model our third model, similar to our first and second models, performs within single-digit percentage points in overall categories. Notably, however, our third model also does score quite well in recall for the 5-star category, achieving a respectable 75% recall. This may not be as strong as the second model's 84%, but it is still enough to beat the 73% achieved by the baseline model.

4.2.5 Our Fourth Model. The fourth model presented is an MLP using our implementation with the following hyper-parameters:

- Epochs : 50
- Learning rate: 0.2
- Hidden layer(s): [100, 40]
- Activation function: `Logistic`
- Regularization: `L2`
- Regularization constant: 0.0001
- Batch size: 200

Similar to the third model, the fourth model differentiates itself by varying the hidden layers, as well as being trained for more epochs. For this model, however, what changes is not the number of hidden layers but rather how many neurons per layer. The fourth model returns to having two hidden layers, but double the number of neurons per layer from 50 to 100 in the first layer and from 20 to 40 in the second layer. This model was trained to see if having hidden layers with more neurons significantly impacted the results on the 5-class problem. The loss curve is presented alongside the training, validation, and testing accuracy:



The first observation is that increasing the number of epochs was unnecessary and yielded little to no change. As with previous models, loss convergence occurred at around 12 epochs and certainly by the 20th epoch. This model performed the best so far in terms of training accuracy, but as the model with the widest hidden layers, this result is not entirely surprising. Our fourth model also performed the most closely so far to the first model.
The precision, recall, and F1 scores, by category, for this model are as follows:

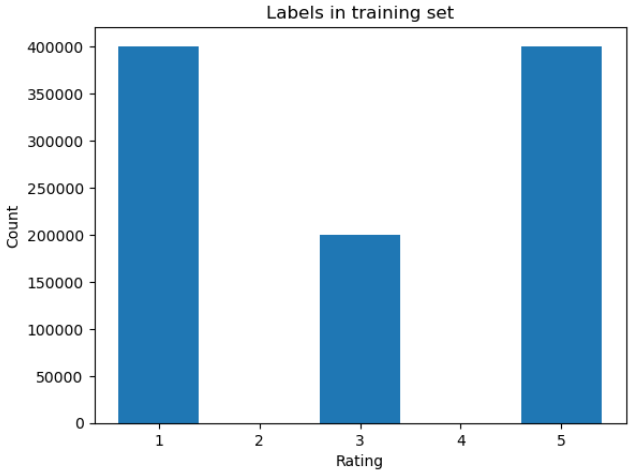
Category	precision	recall	f1-score	support
1	0.65	0.59	0.62	25170
2	0.41	0.55	0.47	25012
3	0.50	0.29	0.37	25041
4	0.48	0.32	0.38	24973
5	0.55	0.84	0.67	24995

	precision	recall	f1-score	support
accuracy			0.52	125191
macro avg	0.52	0.52	0.50	125191
weighted avg	0.52	0.52	0.50	125191

As perhaps be expected by this point, our fourth model has precision, accuracy, and F1 scores which closely match that of our first model. Once again, indicating that for our problem, at least for the size tested, adding more neurons to the hidden layers does not offer a significant leap in performance. Notably, our fourth model does show small degradation in some areas resulting in slightly lower F1 scores, in the 3 and 4-star categories especially, which results in a one percent drop in macro average and weighted average F1 scores.
Overall, when compared to the baseline model our fourth model, similar to our first, second, and third models, perform within single-digit percentage points in overall categories. Notably, however, our fourth model does score quite well in recall for the 5-star category, achieving a respectable 84% recall. This matches the second model's 84%, and once again is enough best the 73% achieved by the baseline model.

4.3 3 Class Problem

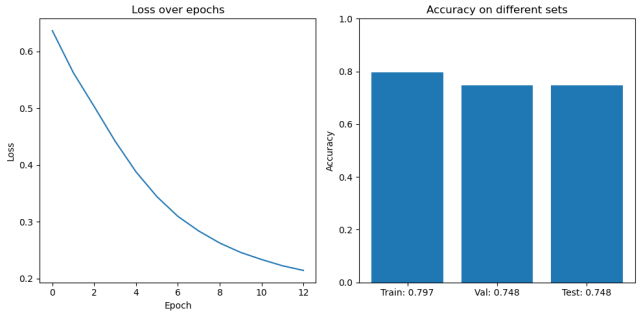
Following the results of the 5 Class problem, it was observed that the weakest performance was in the 2 and 4-star categories. It was hypothesized that perhaps these categories were too similar to be reliably learned well by the size of models that could be trained in reasonable time on the mere mortal machines available to our team. Therefore, it was decided to collapse the problem from 5 classes to 3 classes by modifying the data such that all 2-star review labels were transformed into 1-star labels, and all 4-star labels were transformed into 5-star labels. Although this once again unbalances the dataset, it does not have the horrific skew where one category makes up more than 50% of the examples. In order to mitigate compounding imbalance, the modifications to the dataset were made post-balancing. This results in a dataset which can be visualized as follows:



This image shows that the 1 and 5-star categories approximately double in terms of the number of examples, while the 3-star remain the same. Perhaps in future work, additional testing of different modifications could be done. Such as splitting the 2 and 4-star reviews and putting half of each into the 3 stars and half of the 2 stars into the 1-star and half of the 4-star into the 5-star. However, for the purposes of this evaluation, the desire is simply to observe if this projection into a 3 Class problem results in higher accuracy.

4.3.1 Benchmark Model: Scikit-Learn MLP. Once again, for an overall benchmark comparison, a sk-learn MLP model was trained. This model will serve as a baseline of comparison for our models. For all models, two primary figures will be provided. The first figure contains two plots, the first plot is a Loss vs Epochs plot. The second blot is a bar graph showing the overall accuracy for each set between training, testing, and validation. The second figure is a table that will contain the precision, recall, and F1-score, by rating category, that the model achieved.

The benchmark model loss curve is presented alongside the training, validation, and testing accuracy:



From the above plots, it can be observed that the model achieved a testing accuracy of approximately 74.8%. This testing accuracy will serve as the target for our models to achieve. Additionally, it can be observed that the model began reaching loss convergence at approximately 12 epochs. This figure will be used as a base indicator for the number of epochs our models should train for.

The precision, recall, and F1 scores, by category, for this model are as follows:

Category	precision	recall	f1-score	support
1	0.76	0.86	0.81	50182
3	0.53	0.29	0.38	25041
5	0.79	0.86	0.82	49968

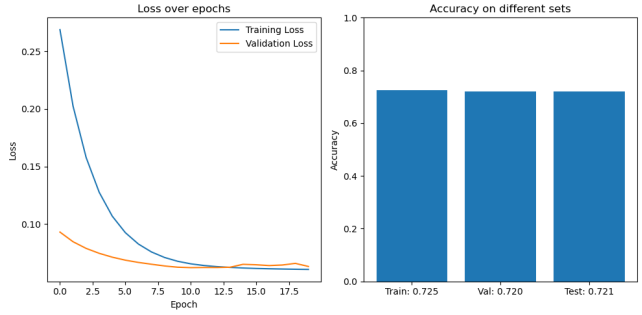
	precision	recall	f1-score	support
accuracy			0.75	125191
macro avg	0.69	0.67	0.67	125191
weighted avg	0.73	0.75	0.73	125191

From this table, it can be observed that The model performs best in the 1 and 2-star categories. Achieving over 75% precision and recall for these categories, in fact achieving 86% recall in both the 1 and 5-star categories. This performance results in F-1 scores for both the 1 and 5-star categories above 80% These scores will serve as the baseline for comparison for our model.

4.3.2 Our Fifth Model. The fifth model presented is an MLP using our implementation with identical hyper-parameters to our first model:

- Epochs : 20
- Learning rate: 0.2
- Hidden layer(s): [50, 20]
- Activation function: `Logistic` (`Sigmoid`)
- Regularization: `L2`
- Regularization constant: 0.0001
- Batch size: 200

The loss curve is presented alongside the training, validation, and testing accuracy:



It can be observed from the above that both the training and validation losses converge around 11 epochs. Additionally, it can be observed that the over-fitting of the training data appears to be minimal as the training, validation, and testing see similar accuracy, and a divergence between the training and validation loss is not observed after convergence. In the 3-class problem, our model achieves 72.1% accuracy. While this is lower than the baseline of 74.8%, it is still within less than 3% overall. Both models saw a significant improvement in accuracy when moving to this problem.

The precision, recall, and F1 scores, by category, for this model are as follows:

Category	precision	recall	f1-score	support
1	0.68	0.92	0.78	50182
3	0.52	0.22	0.31	25041
5	0.83	0.77	0.80	49968

	precision	recall	f1-score	support
accuracy			0.72	125191
macro avg	0.68	0.64	0.63	125191
weighted avg	0.71	0.72	0.69	125191

From this table, it can be observed that, once again, the model has the strongest precision, recall, and F1 scores in the 1 and 5-star categories, similar to the sk-learn model. The 3-star category actually sees a rather drastic decline in recall from the 5-class problem performance, dropping from 32% to 22%.
Overall, when compared to the baseline model our model performs within single-digit percentage points in overall categories. The largest differences are in our fifth model's comparatively high 1-star category recall, which is 92%, over 5% higher than the baselines 86%. As well as our fifth model's high, 83% precision in the 5-star category. This bests the baseline's 79%.

4.4 Observations and Future Work

Following our benchmarks and comparisons, the following observations have been made. Varying the hidden layers of our network to the degree available given the available time and computing resources did not offer meaningful impacts to performance. Additionally, varying the activation function to `ReLU` did not, by itself, offer a meaningful impact on performance.
If this work was extended, testing significantly larger networks, with varying learning rates and allowing different activation functions per layer may be areas from which meaningfully additional performance could be gained, bringing our model(s) closer or perhaps even beating the baseline.

5 GROUP MEMBER CONTRIBUTIONS

5.1 Matthew Freestone (maf0083: 904043882)

Contributions Include:

- Derivation and Implementation of Backpropagation and MLP class.
  - Creation and debugging of the original backpropagation algorithm.
  - Vectorizing Batch Backpropagation.
  - Making MLP public methods intuitive and easy to use.
- Creation of the TwoDimProblem for demonstrating the efficacy of MLP model/
- Implementation and some documentation portions of the final presentation and report.

5.2 Will Humphlett (wah0028 : 903997842)

Contributions include:

- Infrastructure supporting the implementation of our mlp:
  - Environment bootstrapping and consistent performance across platforms.
  - Downloading and preprocessing of other dataset alternatives.
  - Hyperparameter matrix training.
  - Final organization of code and documentation.
- Donation of computing time to training and evaluation of the model(s).
- Portions of the final presentation and report.

5.3 Matthew S. Shiplett (mss0033 : 903656232)

Contributions include:

- Implementation of portions of data preprocessing:
  - Implementation to ensure multiple files could be read in and combined into a single Pandas dataframe.
  - Implementation of data-set balancing; i.e. ensuring it was possible to truncate entries dataframe in such a way that there would be a roughly equal number of examples per label.
  - Implementation of code allowing for the 5 class problem to be modified into a 3 class problem
- Donation of computing time to training and evaluation of the model(s)
- Portions of the final presentation and report.