Neural Networks for Binary Classification

Ava Nafisi

November 27th, 2025

Abstract

We will examine the performance of a neural network multi-layer perceptron trained for classification on an Online Shopper dataset. We will discuss the architecture of the Neural Network before implementing the model in Python. Finally, we will assess the model's performance using visualizations and analyze accuracy metrics.

1 Introduction

Classification is a common problem tackled in Machine Learning when building predictive models. Classification models allow us to predict whether data fits into two or more classes. In this paper, we are interested in tackling a classification problem on predicting online visitor purchasing intentions based on their behavior in visiting a website. This will be a binary classification, where a single session by a unique user results in, or does not result in, revenue for the site. We will tackle this problem by building a neural network model. Neural networks have many applications due to the flexibility and adaptability permitted in building their structure. Classification prediction is one of the many applications that neural networks can perform.

Our purpose will be to examine how to build a type of neural network, the multilayer perceptron, for classification, implement it into Python, and test the performance of the model on the dataset. The performance of classification models can be measured by different metrics, but we will be focusing on the misclassification rate as our error, with further analysis into the recall, precision, and F1-scores of each class. We will visualize the accuracy of the training and validation sets of the neural network over each epoch, to get a look at the performance of the neural network and a comparison of the accuracy between the training and validation sets. Furthermore, we will use cross-validation to get an average accuracy over five folds of data, to ensure the performance of our model is consistent.

2 Neural Networks

2.1 Overview

Neural networks are models that consist of interconnected neurons, organized by layer, that parse data for patterns and relationships. In neural networks, each neuron receives an input of data that it computes using an activation function. Activation functions are various functions that aid in parsing the nonlinear relationships within the data. After a neuron computes the data based on the activation function, the neuron outputs the data, which can be used by other neurons in later layers until the network computes the output layer.

The type of neural network we will be implementing is a Multilayer Perceptron (MLP). MLPs are a type of feed-forward neural network (FNN). An FNN is a neural network where information flows in a single direction, from input to output. As there are no loops or feedback, FNNs are one of the simplest types of neural networks, but are well-suited for predictive applications. Additionally, MLPs being multilayer means the neural network will consist of an input layer, one or more hidden layers, and an output layer. The hidden layer(s) are what allow deeper complexity in learning patterns of the data [2][3].

2.2 Multilayer Perceptron

A single layer of a neural network is defined as:

$$F(x) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{1}$$

where W is the weight matrix, b is the bias vector, and σ is the activation function. The weights linearly transform x, then, in addition to the bias vector, the data is non-linearly activated by the function σ. In the case of a multilayer neural network, this layer can then become the inner composite function of the next layer, making the overall composition of a multi-layer NN:

$$F(x) = (F_L \circ \cdots \circ F_1)(x), \tag{2}$$

with L representing the total number of layers and each layer k being designated as:

$$F_k(x) = \sigma_k(\mathbf{W}_k\mathbf{x} + \mathbf{b}_k). \tag{3}$$

The goal is to find the best $\mathbf{W}_k$ and $\mathbf{b}_k$ of each layer to minimize the loss between $F(x)$ and y. In the case of classification, our loss will be the binary cross-entropy (BCE) function:

$$BCE = -\frac{1}{N}\sum_{i=1}^{N} [\, y_i \log(p_i) + (1 - y_i)\log(1 - p_i)], \tag{4}$$

Where N is the number of observations, y is the binary label (0 or 1) of the i^{th} observation, and p_i is the predicted probability of the i^{th} observation being class 1 [4]. BCE allows us the measure the distance between the true labels and the predicted probabilities of each observation. Therefore, we try to minimize BCE to make the probabilities as close to the true labels as possible.

As for training the MLP, this will be done through a process of forward propagation and back-propagation. Forward propagation is the initial phase that processes the input through the network to produce a prediction. This process follows solving equation (2) layer by layer in (3) as previously explained. Backpropagation, in contrast, is about propagating the errors of the prediction back through the network, so that each layer can update its weights and bias as needed to improve the next iterations of predictions. In short, backpropagation computes a gradient for its loss function with respect to each layer of weights and biases under the chain rule, the gradient indicating how much the weights and biases should be adjusted. This process can be done with many optimization algorithms, but for our MLP, we will be using the optimization algorithm Adam (Adaptive Moment Estimation).

Adam is an optimizer that combines optimizers Momentum and Root Mean Square Propagation (RMSprop). Momentum accelerates gradient descent by adding an exponentially weighted moving average of past gradients [5]. The momentum term m_t is updated by:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\frac{\partial L}{\partial w_t}, \tag{5}$$

being the first moment estimate for Adam. RMSprop, in contrast to momentum, uses the exponentially weighted moving average of squared gradients. RMSprop's weighted average v_t is updated by:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\left(\frac{\partial L}{\partial w_t}\right)^2, \tag{6}$$

being the second moment estimate for Adam. These two estimates are then corrected for any bias towards zero (due to being initialized at zero),

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{7}$$

and then incorporated into Adam's final weight update equation as:

$$w_{t+1} = w_t - \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}\alpha, \tag{8}$$

where α is the learning rate (default = 1), β_1 and β_2 are the decay rates for the moving averages of the gradient and squared gradient (defaults $\beta_1 = 0.9$ and $\beta_2 = 0.999$), and ϵ is a small, positive constant to avoid division by zero [5].

Ultimately, Adam is a good optimizer to use as its adaptive learning rate based on past gradients helps avoid oscillations and local minima. In practice, Adam generally outperforms other optimizers in both time and accuracy with minimal tuning, making it our selection for our backpropagation algorithm.

2.3 Building the Neural Network

In tandem with the nature of neural networks being flexible, there is no clear-cut way to decide the number of hidden layers and neurons that will perform best on a dataset. Often, trial and error is needed to tune the ideal neural network for each situation. Building the architecture of a neural network can be based on the data, the complexity of the relationships we desire to parse, and any computational costs (in the case of large datasets).

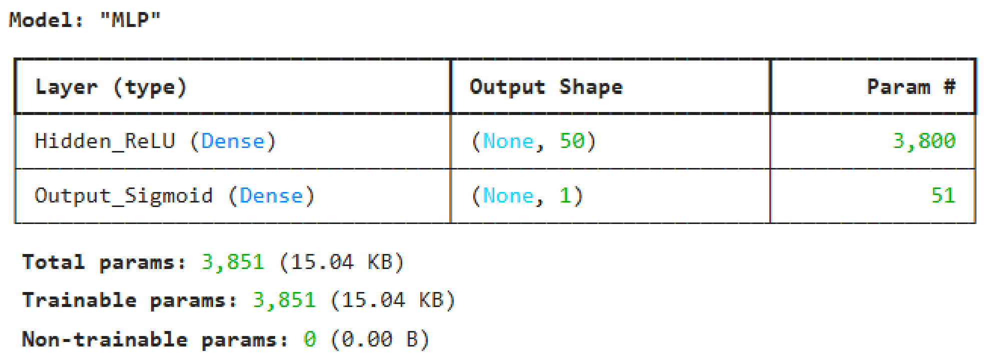
For deciding the number of hidden layers, even a single hidden layer is sufficient for most problems. Furthermore, with the moderate size of our dataset, we want to avoid overfitting with an overly complex neural network. Therefore, to be conservative, we started with one hidden layer. If the performance of the model showed underfitting when testing, more hidden layers could be added. However, the initial performance of the single-layer Neural Network was solid, and a quick additional test with a second hidden layer showed no improvement in the misclassification rate (shown in Section 5). So, the single hidden layer was kept.

For the activation functions, we decided to use ReLU for the hidden layer and sigmoid for the output layer,

$$\sigma_1 = \max\{t, 0\}, \quad \sigma_2 = \frac{1}{1+e^{-t}}. \tag{9}$$

ReLU is used in the hidden layer as a simple function to introduce nonlinearity into the dataset, and sigmoid is used for the output layer to transform the output of the data into a probability (0-1), as to allow for classifying the outputs.

As for neurons, there are a few rules of thumb in deciding the number of neurons in a hidden layer. We decided to follow the rule of taking two-thirds the sum of the number of inputs and outputs [7]. After preprocessing (detailed in Section 3), the data has 75 inputs, and classification means an output of 1, resulting in a sum of 76. Therefore, our hidden layer was decided to have 50 neurons. The architecture of the Neural Network is summarized in Table 1 below.



Model: "MLP"

Layer (type)	**Output Shape**	**Param #**
Hidden_ReLU (Dense)	(None, 50)	3,800
Output_Sigmoid (Dense)	(None, 1)	51

Total params: 3,851 (15.04 KB)
Trainable params: 3,851 (15.04 KB)
Non-trainable params: 0 (0.00 B)

Table 1: Table showing architecture of the NN built in Python.

With the architecture of the neural network itself set, the hyperparameters were decided next. The hyperparameters for fitting a neural network are the optimizer, batch size, and number of epochs. We have already explained that the optimizer will be Adam. Next, batch size is the number of training samples sent through one iteration of the forward/backward pass. Increasing in batch size increases memory computational costs, but often results in more accurate estimates of the gradient. Computational costs were not a major issue, so were are more concerned with accuracy. A grid search was used to test different values of batch size on the neural network (explained further in Section 4). The batch size of 15 ended up with the highest accuracy, making it what we used moving forward. The accuracies of all the batch sizes were within 1-2 percent of each other, however, so it seems the effect of batch size on training this dataset was negligible.

Epochs are a full iteration of training, as in, when enough batches are done for the network to have seen the full dataset one time. Basically, how long do we train the MLP for based on the number of epochs we want to run. Rather than have a static number of epoch, an early stopping callback with a patience of 10 was added to the algorithm. What this does is that if the validation accuracy of the model has not improved for 10 epochs, the training will automatically stop. This will make training more efficient, especially when we want to repeatedly train the model with cross-validation. We will then set a maximum of 80 epochs, in case the validation set keeps improving. With these hyperparameters chosen, the model was ready to be run.

3 Dataset

3.1 Overview

The dataset we will be running our MLP on is the Online Shoppers Purchasing Intention Dataset, sourced from the UC Irvine Machine Learning Repository [1]. The dataset recorded 12,330 sessions of online users visiting a website, with each session belonging to a different user. The sessions were recorded over a one-year period to avoid bias toward any specific campaign, holiday, or period of the year. Each session recorded the following 18 variables on user behavior:

Numerical:

- Administrative - Number of pages about account management visited.
- Administrative Duration - Total time spent on administrative pages (seconds).
- Informational - Number of pages about website information visited.
- Informational Duration - Total time spent on informational pages (seconds).
- Product Related - Number of product-related pages visited.
- Product Related Duration - Total time spent on product-related pages (seconds).
- Bounce Rates - Average bounce rate of the visited pages. Bounce rate is the percentage of visitors who enter the site from that page and then leave without triggering any other requests
- Exit Rates - Average exit rate of the visited pages. Exit rate is the percentage of visitors who exit the site after that page.
- Page Values - Average value for a page that a user visited before completing a transaction.
- Special Day - Closeness of the site visiting time to a special day (0-1).

Categorical:

- Month - Month of user visit date. (12)
- Operating Systems - The operating system of the user. (8)
- Browser - The browser used by the user. (13)
- Region - Geographic region where the session was started. (9)
- Traffic Type - Traffic source by which the visitor has arrived at the Web site (i.e., banner, SMS, direct) (20)
- Visitor Type - Labels the visitor as either ”New Visitor”, ”Returning Visitor”, or ”Other”. (3)
- Weekend - Boolean indicating if the date of visit was on the weekend. (2)

- Revenue - Boolean indicating whether the visit results in a transaction. (2)

Evidently, there is a mix of numerical and categorical columns. The categorical columns also vary in class amount, such as Traffic Type, which has 20 unique categories. We are interested in predicting whether a visitor will make a purchase based on the features of their session; therefore, the target variable will be Revenue. Of the 12,330 sessions, 10422 (84.5%) sessions resulted in false for revenue, and 1908 (15.5%) sessions resulted in true for revenue, meaning a larger majority of the visits ended without a purchase. Class imbalances in the target variable can cause performance issues for predictive models, as it is common for majority classes to be favored in predictions. This is something to consider when we evaluate the performance of the model later in the paper.

3.2 Preprocessing

Preprocessing was required before the data could be trained by the model. First, the categorical columns needed to be encoded for the model to work. The categorical columns were one-hot encoded, meaning a new column was created per unique class per feature, with each column's values being 1 or 0 if that specific class was present. This increased the number of columns in our input from 17 to 75 columns. The response, Revenue, was then label encoded. We used label encoding so that each class could be assigned a numerical value while keeping the response as one column.

After encoding, the data was split into an 80-20 train and test set. Using a test set to calculate performance allows us to check that the model can generalize when predicting unseen data. After splitting the data, the numerical columns were min-max scaled:

$$\mathbf{x}' = \frac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})}. \tag{10}$$

The scaling was done after splitting the data to make sure there was no data leakage from the test set for any min or max calculations. Min-max scaling scales all the features to a fixed range, improving optimization and preventing dominance from numerically large features [8]. Once both the train and test sets were transformed based on the training scaling, the dataset was ready to be used.

4 Implementation

4.1 Initial Set Up

The packages required for implementing our MLP were imported, and our data was loaded in using Pandas.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.api as sm
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler

from sklearn.preprocessing import LabelEncoder
import tensorflow.keras as keras
from scikeras.wrappers import KerasClassifier
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense, Dropout, Input
from keras.callbacks import EarlyStopping
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score


filename = 'online_shoppers_intention.csv'
df = pd.read_csv('online_shoppers_intention.csv')
```

4.2 Preprocessing

Preprocessing is next, following the steps explained in Section 3. The function takes in the dataset. First, it lists of the numerical columns and categorical columns were made, and the X and y are created. For each categorical column, the one-hot encoded columns are made, added to X, and the original column is dropped. Y is additionally label encoded. The train and test sets are then made, with the X train being used to fit the MinMaxScaler() to transform both sets.

```
def preprocessing(df):
    num_cols = list(df.drop(columns = ['Month', 'OperatingSystems', 'Browser',
        'Region', 'TrafficType', 'VisitorType', 'Weekend', 'Revenue']).columns)
    cat_cols = ['Month', 'OperatingSystems', 'Browser', 'Region', 'TrafficType',
        'VisitorType', 'Weekend']

    X = df.drop(columns = ['Revenue'])
    y = df['Revenue']

    # Encode Categorical Columns
    encoder = OneHotEncoder(sparse_output=False)
    for col in cat_cols:
        # encode categorical feature and create a new df
        encoded_col = encoder.fit_transform(X[[col]])
        col_df = pd.DataFrame(encoded_col, columns=encoder.get_feature_names_out([col]),
                            index=X.index)

        X = pd.concat([X, col_df], axis = 1) # add encoded df to df
        X = X.drop(columns = [col]) # drop original column
    # Label encode response variable
    le = LabelEncoder()
    y = le.fit_transform(y)

    # Split into training and test set
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Scale Numerical Columns
    scaler = MinMaxScaler()
    scaler.fit(X_train[num_cols])
    X_train[num_cols] = scaler.transform(X_train[num_cols])
    X_test[num_cols] = scaler.transform(X_test[num_cols])

    return X_train, X_test, y_train, y_test
```

4.3 Model Building

Next is a function to build the MLP. Keras is used to create the neural network. The model is set as sequential, which signifies the neural network as being a simple feed forward model. The two layers are added and the model is then compiled, with Adam set as the optimizer, the loss being set to binary cross entropy, and the metric for scoring being accuracy. The early stopping callback is then created based on validation accuracy, with a patience of 10. Finally, the function returns both the model and the call back to be used for training.

```
def build_nn():
    # build a model
    model = Sequential()
    model.add(layers.Input(shape=(X_train.shape[1],)))  # Explicitly define input layer
    model.add(Dense(50, activation='relu')) # number of neurons chosen using 2/3 rule of thumb
    model.add(Dense(1, activation='sigmoid')) # output layer
    model.summary()

    # compile the model
    model.compile(optimizer='Adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

    # early stopping callback if validation accuracy does not improve for 10 epochs
    es = EarlyStopping(monitor='val_accuracy',
                                   mode='max', # don't minimize the accuracy!
                                   patience=10,
                                   restore_best_weights=True)

    return model, es
```

4.4 Grid Search

To tune our hyperparameter batch size, a grid search was created. The grid uses a range of 7 numbers between 5 and 100. For each possible batch size, a model is created and fit to the data, recording the highest validation accuracy during training. After the entire grid is tested, all list of accuracies is printed.

```
# Get processed data
X_train, X_test, y_train, y_test = preprocessing(df)

batch_size_list = [5, 10, 15, 20, 30, 50, 100]
grid_search_results = []
for batch in batch_size_list:
    model, es = build_nn()

    # now we just update our model fit call
    history = model.fit(X_train,
                        y_train,
                        callbacks=[es],
                        epochs=80,
                        batch_size=batch,
                        validation_data=(X_test, y_test),
                        shuffle=True,
                        verbose=0)

    # get accuracy
    accuracy = np.max(history.history['val_accuracy'])
    grid_search_results.append([batch, accuracy])

for i in range(len(grid_search_results)):
    print(f"Batch = {grid_search_results[i][0]}, Accuracy = {grid_search_results[i][1]}")
```

4.5 Training NN

After a batch size is selected, we do a full run through, putting the train set into the model. In the fit function, verbose is set to 1 to output the history of epochs and their train/validation accuracies. Once the model is trained, predictions of the test set are computed, then rounded into either class 0 or class 1. A confusion matrix and classification report from sklearn are then outputted to show the test performance of the model.

```
# Get processed data
X_train, X_test, y_train, y_test = preprocessing(df)

# Build NN from function
model, es = build_nn()

# now we just update our model fit call
```

```
history = model.fit(X_train,
                    y_train,
                    callbacks=[es],
                    epochs=80,
                    batch_size=15,
                    validation_data=(X_test, y_test),
                    shuffle=True,
                    verbose=1)

# Predict and round values to 1 or 0
model.predict(X_test)
preds = np.round(model.predict(X_test),0)

# confusion matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, preds))

print("\nClassification Report:")
print(classification_report(y_test, preds))
```

4.6 Visualizing Performance

To get a better look at the training history, we used matplotlib to create a graph that plots the training and validation accuracy of the epochs over time. Additionally, the best validation accuracy of the model training is then printed out underneath the graph.

```
# accuracy for learning curve
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

# range of X (no. of epochs)
epochs = range(1, len(acc) + 1)

# plot
plt.plot(epochs, acc, 'bo', label='Training accuracy') # training accuracy as dots
plt.plot(epochs, val_acc, 'orange', label='Validation accuracy') # validation accuracy as line
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Get best accuracy from epochs
best_acc = np.max(val_acc)
print("The Neural Network Model epoch with the best validation accuracy was " + str(best_acc))
```

4.7 Cross-Validation

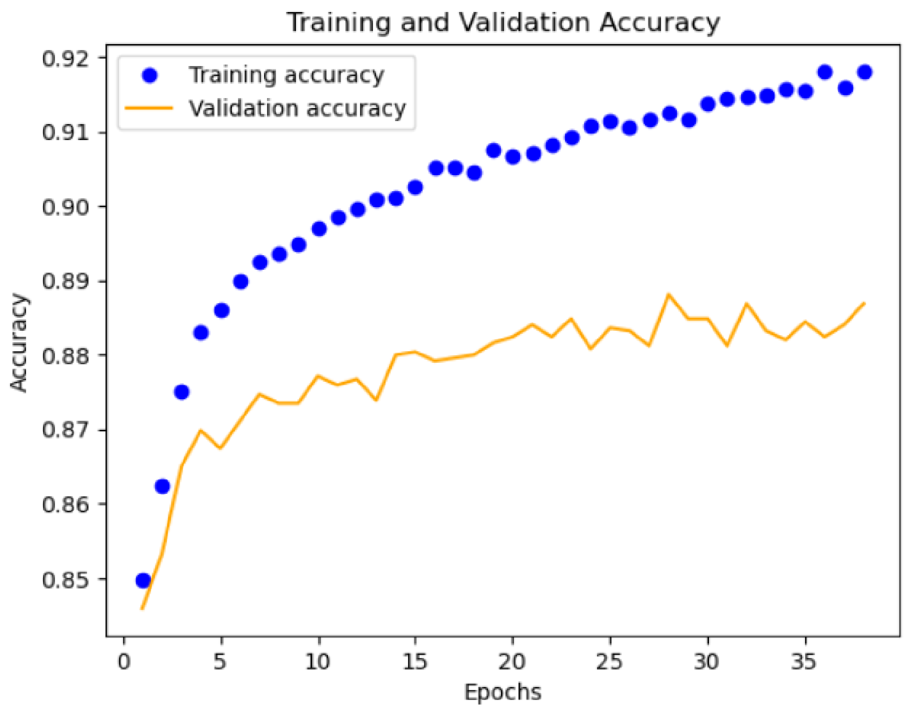
Finally, to get a better assessment of the performance of our model, a 5-fold cross-validation was applied to our model. First, our dataset is preprocessed without the scaling, as the data will be split later. The inputs are flattened, which will allow KFold() to split the dataset properly [9]. Then, our k = 5 cross-validation is set up using sklearn. For 5 folds, the data is split, scaled, and fit to the model, with one of the folds being the validation set each time. The accuracy of each iteration is recorded. Once all the folds have been run, a list and average of the accuracies is printed out.

```
# redo preprocessing without scaling
num_cols = list(df.drop(columns = ['Month', 'OperatingSystems', 'Browser', 'Region', 'TrafficType',
cat_cols = ['Month', 'OperatingSystems', 'Browser', 'Region', 'TrafficType', 'VisitorType', 'Weekenc

X = df.drop(columns = ['Revenue'])
y = df['Revenue']

# Encode Categorical Columns
encoder = OneHotEncoder(sparse_output=False)
for col in cat_cols:
    # encode categorical feature and create a new df
    encoded_col = encoder.fit_transform(X[[col]])
    col_df = pd.DataFrame(encoded_col, columns=encoder.get_feature_names_out([col]), index=X.index)

    X = pd.concat([X, col_df], axis = 1) # add encoded df to df
    X = X.drop(columns = [col]) # drop original column

# Label encode response variable
le = LabelEncoder()
y = le.fit_transform(y)

# Flatten
X = X.to_numpy().reshape((X.shape[0], -1))

# Get 5-fold cross-validation
cv = KFold(n_splits=5, shuffle=True, random_state=42)

accuracies = []
for fold, (train_i, test_i) in enumerate(cv.split(X)):
    # split train set into further train-test folds
    X_fold_train = X[train_i]
    X_fold_test = X[test_i]
    y_fold_train = y[train_i]
    y_fold_test = y[test_i]

    # Scale after splitting into train/test
    scaler = MinMaxScaler()
    scaler.fit(X_fold_train)
    X_fold_train = scaler.transform(X_fold_train)
    X_fold_test = scaler.transform(X_fold_test)

    # create model and fit
    model, es = build_nn()
    model.fit(X_fold_train, y_fold_train,
              callbacks=[es],
              epochs=80,
              batch_size=15,
              validation_data=(X_fold_test, y_fold_test),
              shuffle=True,
              verbose=0)

    # Get predictions and record accuracy
    model.predict(X_fold_test)
    preds = np.round(model.predict(X_fold_test),0)
    accuracy = accuracy_score(y_fold_test, preds)
    accuracies.append(accuracy)

# After all runs, calculate average accuracy
for i in range(len(accuracies)):
    print(f"Fold {i} Accuracy = {accuracies[i]}")
average_acc = np.mean(accuracies)
print(f"\nThe average accuracy of the Cross-Validation is: {average_acc}")
```

5 Results

5.1 Validation Set Results

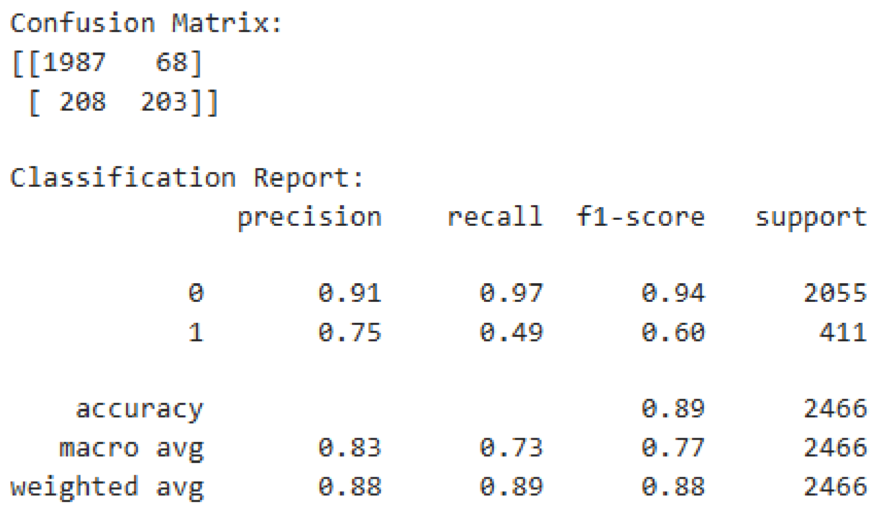
Figure 1 below shows the history of training the MLP. After an initial jump in accuracy from the first few epochs, a slow, steady growth is seen in both accuracies until about epoch 20, where the validation accuracy begins to oscillate around 88% accuracy. Furthermore, the validation accuracy seems to consistently be around 1-3% less than the training accuracy. While a bit less than the training accuracy, it is close enough that overfitting does not seem to be an issue.



```
The Neural Network Model epoch with the best validation accuracy was 0.8880778551101685
```

Figure 1: Graph displaying training and validation accuracy over epochs.

As for the performance of the model on the test set, Table 2 shows that the MLP had an accuracy of 89%, or a misclassification rate of 11%. Overall, this is quite a good misclassification rate, with most user sessions being correctly predicted to generate revenue or not. Further analysis of the classification report, however, does show where the accuracy is suffering the most. Notably, class 1 has a lower F1-score of 0.60, especially due to its 49% recall rate. A low recall rate means that the model is frequently giving false negatives to sessions that actually resulted in revenue (208 out of 411 revenue positive sessions).



```
Confusion Matrix:
[[1987   68]
 [ 208  203]]

Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.97      0.94      2055
           1       0.75      0.49      0.60       411

    accuracy                           0.89      2466
   macro avg       0.83      0.73      0.77      2466
weighted avg       0.88      0.89      0.88      2466
```

Table 2: Confusion matrix and classification report of MLP predicting test set.

A likely reason for the lower class 1 performance is the class imbalance that was previously mentioned in Section 3. Revenue positive sessions make up only 15.5% of the dataset. It is common, even expected, that a much smaller minority class will perform worse than the majority class. Even models that can handle complexities in the data such as deep learning neural networks can struggle to predict observations in the minority class.

Regardless, identifying session that resulted in revenue is the primary goal. A business owner would likely be interested in making sure they identify paying customers, therefore improving the performance of class 1 would be a priority in any further testing.

5.2 Cross-Validation Results

Fold	Accuracy
Fold 1	88.52%
Fold 2	90.00%
Fold 3	89.98%
Fold 4	89.33%
Fold 5	89.50%
Average	89.47%

The performance of the 5-fold cross-validation kept within an 88%-90% accuracy range, resulting in an average accuracy of 89.47%. Based on the consistency of the model performance across all folds, it seems that the MLP model generalizes well to new, unseen data.

6 Conclusion

Overall, the performance of the MLP on the online shopping dataset was good, but not perfect. The model performed consistently at an accuracy averaging around 90%. There was, however, notably worse performance in the minority of sessions that results in revenue. If this project were to be expanded further, the priority would be in tackling the class imbalance of the dataset. Sampling techniques such as undersampling or SMOTE (Synthetic Minority Over-sampling Technique) may help balance the dataset for the model to improve performance. Additionally, modifying class weights of the model can possibly aid performance by assigning heavier penalties to misclassifications of the minority class. Either way, the MLP was shown to be effective in predicting binary classification.

7 References

[1] Sakar, C., & Kastro, Y. (2018, August 30). Online shoppers purchasing intention dataset. UCI Machine Learning Repository.
https://archive.ics.uci.edu/dataset/468/online+shoppers+purchasing+intention+dataset
[2] Jaiswal, S. (2024, February 7). Multilayer Perceptrons in Machine Learning: A Comprehensive Guide. DataCamp; DataCamp. https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning
[3] GeeksforGeeks. (2024, June 20). Feedforward Neural Network. GeeksforGeeks.
https://www.geeksforgeeks.org/deep-learning/feedforward-neural-network/
[4] GeeksforGeeks. (2024a, May 27). Binary Cross Entropy/Log Loss for Binary Classification. GeeksforGeeks. https://www.geeksforgeeks.org/deep-learning/binary-cross-entropy-log-loss-for-binary-classification/
[5] GeeksforGeeks. (2020, October 22). What is Adam Optimizer? GeeksforGeeks.
https://www.geeksforgeeks.org/deep-learning/adam-optimizer/
[6] Chuang, L. (2020, October 1). Build a Neural Network in Python (Binary Classification). Medium. https://medium.com/luca-chuangs-bapm-notes/build-a-neural-network-in-python-binary-classification-49596d7dcabf
[7] Heaton, J. (2017, June 1). The Number of Hidden Layers. Heaton Research.
https://www.heatonresearch.com/2017/06/01/hidden-layers.html
[8] SciKit-Learn. (2019). sklearn.preprocessing.MinMaxScaler — scikit-learn 0.22.1 documentation. Scikit-Learn.org.
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
[9] GeeksforGeeks. (2024c, November 12). How to Use KFold CrossValidation in a Neural Network. GeeksforGeeks.