Classification of Diabetic and Non-Diabetic Individuals Based on Survey Data

Introduction

Diabetes is one of the most prevalent chronic diseases in the US and worldwide. Early identification of diabetes risk factors is crucial for the prevention and management of the disease. This would not only result in a public health benefit, but also an economic benefit by lessening the stress on the budgets of affected families and decreasing the pressure on the healthcare system as a whole.

The aim of this project is to perform a classification task using the CDC Behavioral Risk Factor Surveillance System (BRFSS) 2015 survey dataset[1] to identify individuals at risk of diabetes based on various features.

Problem and Dataset Description

Identification of diabetes -especially early on- can be very challenging due to a variety of reasons. Symptoms of diabetes can be mild or non-existent.[2] Additionally, because symptoms of diabetes are quite general, even if they are observed, it would be difficult to determine they are being caused by diabetes without further testing.[3] For these reasons, it is necessary to explore different avenues of assessing diabetes risk in individuals. In this project, to achieve this goal, three machine learning (ML) methods are explored. The methods will be detailed in the next section.

The dataset used is the CDC BRFSS 2015 survey dataset. The dataset includes 253,680 survey responses, 218,334 belonging to the no diabetes class (0) and 35,346 belonging to the pre-diabetes or diabetes class (1). As a percentage, 86% of the data points belong to the no diabetes class. Out of the 330 features in the original BRFSS dataset, 21 features relevant to diabetes are selected. The features in the dataset are as follows.

1. High blood pressure: Adults who have been told they have high blood pressure by a doctor, nurse, or other health professional. *Range: 0, 1*
2. High cholesterol: Have you EVER been told by a doctor, nurse or other health professional that your blood cholesterol is high? *Range: 0, 1*
3. High cholesterol: Cholesterol check within the past five years. *Range: 0, 1*
4. Body mass index. *Range: 12 - 98*
5. Smoking: Have you smoked at least 100 cigarettes in your entire life? *Range: 0, 1*
6. Chronic health conditions: (Ever told) you had a stroke. *Range: 0, 1*
7. Chronic health conditions: Respondents that have ever reported having coronary heart disease (CHD) or myocardial infarction (MI). *Range: 0, 1*
8. Physical activity: Adults who reported doing physical activity or exercise during the past 30 days other than their regular job. *Range: 0, 1*
9. Diet: Consume Fruit 1 or more times per day. *Range: 0, 1*
10. Diet: Consume Vegetables 1 or more times per day. *Range: 0, 1*
11. Alcohol: Heavy drinkers (adult men having more than 14 drinks per week and adult women having more than 7 drinks per week). *Range: 0, 1*

[1] https://www.kaggle.com/datasets/alexteboul/diabetes-health-indicators-dataset
[2] https://www.webmd.com/diabetes/understanding-diabetes-symptoms
[3] https://www.healthline.com/health/diabetes/types-of-diabetes#symptoms

12. Health care: Do you have any kind of health care coverage, including health insurance, prepaid plans such as HMOs, or government plans such as Medicare, or Indian Health Service? *Range: 0, 1*
13. Health care: Was there a time in the past 12 months when you needed to see a doctor but could not because of cost? *Range: 0, 1*
14. General health and mental health: How healthy are you in general?: *Range: 1 - 5*
15. General health and mental health: Now thinking about your mental health, which includes stress, depression, and problems with emotions, for how many days during the past 30 days was your mental health not good? *Range: 0 - 30*
16. General health and mental health: Now thinking about your physical health, which includes physical illness and injury, for how many days during the past 30 days was your physical health not good? *Range: 0 - 30*
17. General health and mental health: Do you have serious difficulty walking or climbing stairs? *Range: 0, 1*
18. Demographics: Indicate sex of respondent. *Range: 0, 1*
19. Demographics: Fourteen-level age category. *Range: 1 - 13*
20. Demographics: What is the highest grade or year of school you completed? *Range: 1 - 6*
21. Demographics: Annual household income from all sources: *Range: 1 - 8*

The dataset has no missing values. In order to accelerate gradient descent and ensure better model performance, all features are standardized before any other processing.

Prior to the employment of machine learning methods, correlational analysis is conducted. The Pearson product-moment correlation coefficients (PPMCC) are used as the measure of correlation. It is defined as the covariance of the variables divided by the product of their standard deviations. The PPMCC takes values between -1 and 1 and is a measure of the linear relationship between two variables. Regarding the value of the coefficient, 0 indicates no correlation, 1 indicates perfect positive linear correlation, and -1 indicates perfect negative linear correlation.

Firstly, the following figure demonstrates the correlation between the features and the labels.

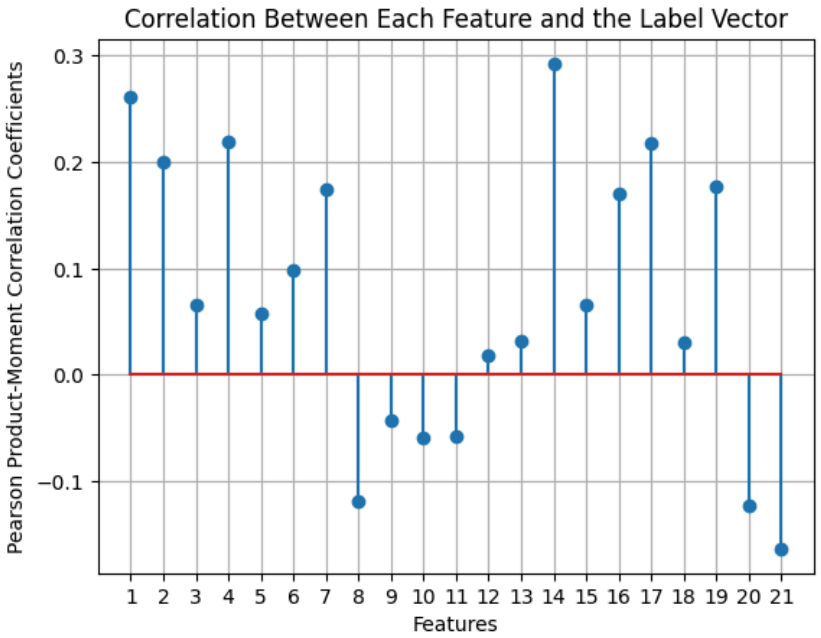

Figure 1: Pearson product-moment correlation coefficients between features and labels.

The feature that has the highest positive correlation with the label is feature 14 with coefficient 0.293. The feature that has the highest negative correlation with the label is feature 21 with coefficient -0.164. The feature that has the least correlation with the label is feature 12 with coefficient 0.018. In other words, one giving themselves a higher general health score is correlated with higher risk of having diabetes, increased income is correlated with lower risk of diabetes, and health care coverage is not significantly correlated with diabetes risk.

Secondly, the following figure demonstrates the correlation between the features.

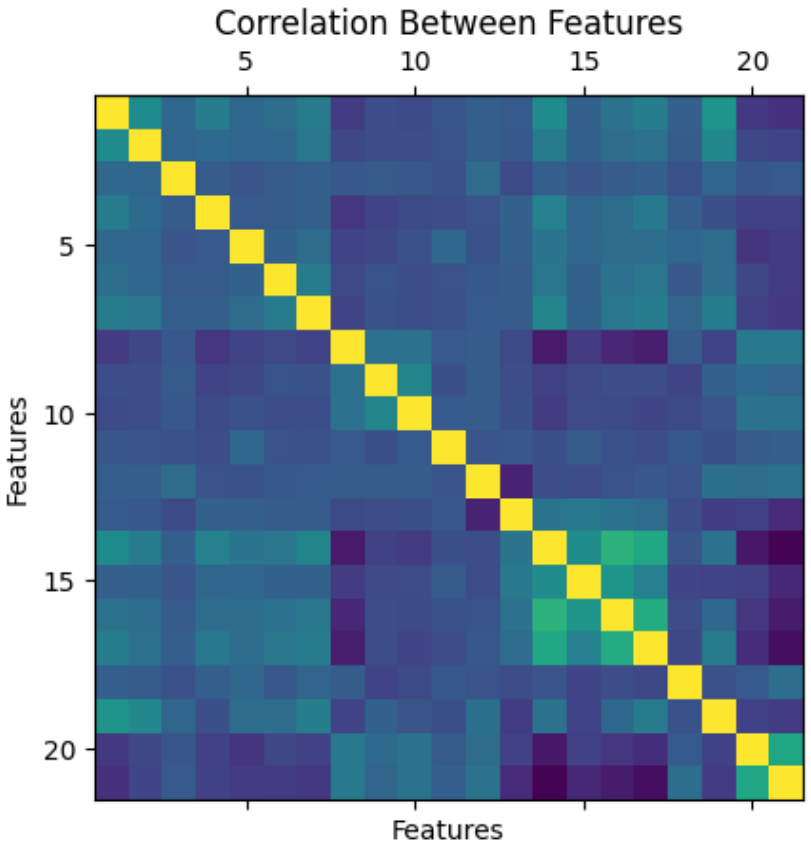

Figure 2: Pearson product-moment correlation coefficients between features.

In the figure above, lighter colors indicate more positive values and darker colors indicate more negative values. The main diagonal is all 1's because the correlation coefficient of a feature with itself is always 1. The features that have the highest positive correlation with each other are features 14 and 16 with coefficient 0.522. The features that have the highest negative correlation with each other are features 14 and 21 with coefficient -0.371. The features that have the least correlation with each other are features 3 and 20 with coefficient 0.000.

After a correlational overview of the dataset, the following section contains the review of the machine learning methods used.

Review of Machine Learning Methods

The primary machine learning task in this project is supervised binary classification. The two classes are "no diabetes" (0) and "pre-diabetes or diabetes" (1). Three algorithms are selected to train three models on the dataset described above. These algorithms are multiple logistic regression (MLR), K-nearest neighbors (KNN), and feedforward neural network (FFNN).

Multiple Logistic Regression

MLR is a statistical technique that can be used to model the relationship between multiple predictor variables and a binary outcome variable. It is widely used in various fields, such as medicine, social sciences, engineering, and business, to analyze data and make predictions based on the probability of an event occurring. The use case in this project fits into the medicine category. MLR can handle both continuous and categorical predictor variables, and both kinds of parameters are present in this project.

One of the main advantages of MLR is that it provides a probability rather than a result without any uncertainty. This feature of MLR makes the model more tunable by changing the threshold value. Another advantage of MLR is that it does not require any assumptions about the distribution of the predictor variables, unlike some other techniques such as linear regression.

However, MLR also has some limitations. One of the main limitations is that it requires the outcome variable to be binary, which means that it cannot handle multiple categories. For these cases, with some modifications to the MLR algorithm, multinomial logistic regression can be used. Another limitation of MLR is that it assumes a linear relationship between the logit of the outcome and the predictor variables, which may not always hold in reality.

MLR has many advantages over other techniques, such as flexibility, interpretability, and simplicity. In this project, it is chosen to observe how a linear model would perform on the dataset. The performances of the following models are expected to be better because of their higher complexity.

K-Nearest Neighbors

KNN is a supervised machine learning algorithm that can be used for both classification and regression tasks. It is based on the idea that the label of a new instance can be predicted by looking at the labels of its closest neighbors in the feature space. The number of neighbors and the distance metric must be chosen appropriately for the model to function optimally.

One of the main advantages of KNN is that it is simple and intuitive to implement and understand. It does not require any training or parameter estimation, and it can handle nonlinear and complex data. Just like MLR, it can also be easily adapted to different types of data, such as categorical, numerical, or mixed. In contrast toMLR, however, KNN uses the entire training data as its model. No training is required.

KNN also has some drawbacks. As is the case with many machine learning models, one of the main challenges is hyperparameter optimization (choosing the best value of k). A small value of k can lead to overfitting, while a large value of k can underfit the dataset. Another challenge is choosing the appropriate distance metric. The first that comes to mind is Euclidean distance, but it may not be suitable for all types of data. This distance metric especially struggles with high-dimensional or sparse data. Furthermore, KNN suffers from the curse of dimensionality. This can reduce the effectiveness and efficiency of the algorithm. To combat this issue, a dimension reduction algorithm such as principal component analysis (PCA) can be used. Finally, traditional KNN does not provide a probability as the output, just the predicted class.

The most significant advantage of KNN compared to MLR is that KNN can fit nonlinear datasets while MLR can not. KNN is chosen to observe how a nonlinear model performs on this dataset compared to a linear one.

Feedforward Neural Network

Artificial neural networks (ANN) are models inspired by the inner workings of the human brain. FFNN is the simplest type of artificial neural network which is a structure that consists of multiple layers of neurons connected by weighted edges. The first layer is called the input layer, the last layer is called the output layer, and all the layers in between are called hidden layers. The network takes an input vector and passes it through the layers, applying an activation function at each neuron, until it reaches the output layer. The output layer produces a prediction or a classification (classification in this case) based on the input. The nonlinear activation function inside each neuron is paramount to the operation of any ANN.

A feedforward neural network does not have any feedback loops or recurrent connections, meaning that the information flows only in one direction, from the input to the output. One similarity of FFNN with MLR is that provided the correct selection of activation function, they can both provide probabilities as output. Just like MLR and KNN, FFNN can handle categorical, numerical, and mixed datasets. In contrast to other models, ANNs are often considered “black boxes” which means they are very difficult to interpret.

A feedforward neural network can be trained using a technique called backpropagation which adjusts the weights of the edges based on the error between the network's output and the desired output. The network can learn to approximate any function that maps an input to an output, given enough data and hidden layers. However, a feedforward neural network also has some limitations, one of which is being prone to overfitting. Regularization techniques can be used to mitigate this issue.

FFNN model is chosen because of its wide variety of use cases and flexibility.

Challenges

- **Implementation of algorithms:** I faced many challenges and overcame them. In this process, I learned a lot about programming ML models and how they work. I expect to use my newly gained ML model programming experience in the future.

- **Computational power:** KNN was the heaviest algorithm to run followed by FFNN and LR. Dataset size had to be reduced significantly for KNN to be completed in a reasonable time frame. However, I do not think this caused the performance of KNN to suffer significantly.

- **Imbalanced dataset:** The dataset contains a significantly larger number of individuals without diabetes compared to those with pre-diabetes or diabetes. While working on KNN, this issue was overcome by class weights. In the other two algorithms this problem was handled by the threshold (0.2 in both cases).

- **Hyperparameter tuning:** Each machine learning algorithm has hyperparameters that need to be tuned for optimal performance. Finding the globally optimal combination of hyperparameters for each model, especially for KNN and FFNN, was impossible. In some cases grid search was conducted on single parameters separately rather than multiple parameters at the same time to save on time and computing resources.

- **Model evaluation:** Many different criteria exist for measuring how well a model describes a dataset. The simplest one, accuracy alone is not sufficient, especially in imbalanced datasets. Metrics such as confusion matrix and F1 score are used to more comprehensively determine the performance of the models.

Methodology

Before training any of the models, the dataset is divided into training and test sets. Then, the dataset is further separated into the X matrix (design matrix) and the y vector (labels). After that, each feature in the X matrix is standardized. This is done by first subtracting the mean, then dividing by the standard deviation. Each column of the resulting matrix has 0 mean and unit standard deviation. As an alternative to standardization, min-max range scaling was also tested but it performed worse across all machine learning methods.

Multiple Logistic Regression

The MLR algorithm is tested with imbalanced and balanced training sets. The test set is untouched in both cases.

The algorithm is initialized with random parameters, including a bias term, and the design matrix (X) is augmented with a column of ones to account for the bias. The bias term increases the flexibility of the algorithm .

The cost function used in this algorithm is logistic loss with L1 regularization. The bias term is excluded when computing the L1 regularization. The training process is completed using gradient descent. The gradient of the log loss and the gradient of the L1 regularization term are computed and combined to update the parameters. The training is performed for a specified number of iterations.

The model's predictions are generated using a threshold on the predicted probabilities. If the probability is equal to or greater than the threshold, the instance is classified as 1; otherwise, it is classified as 0.

Optimal hyperparameters are found using grid search and 10-fold cross-validation. The hyperparameters to be optimized are the λ of L1 regularization and the threshold value used in the sigmoid function. This process is repeated for both imbalanced and balanced training sets.

Finally, the best models trained from both imbalanced and balanced training sets are evaluated on the test set. Accuracy, F1 score, and confusion matrix are used as evaluation metrics.

K-Nearest Neighbors

The KNN algorithm is tested with a much smaller dataset as compared to other algorithms because of its computational complexity. The training and test datasets have 8118 data points each. These smaller datasets are constructed by randomly sampling from their respective parents.

In contrast to the other algorithms, the KNN algorithm does not have a training stage. In this case, the computational load is transferred from the training stage to the prediction stage. Therefore, each prediction is much more computationally expensive compared to the other algorithms.

In the prediction stage, every data point is sorted from low to high according to the distance computed by the distance metric. Then, the prediction is made according to the most frequent label in the closest k data points.

A non-exhaustive list of hyperparameters to tune are: distance metric selection, parameters of various distance metrics, number of nearest neighbors (k), various modifications of the original algorithm etc. In this case, generally through trial-and-error and rules of thumb found through research, the following hyperparameters are decided to be kept fixed: k = sqrt(number of data points) and the use of weighted KNN algorithm.

Optimal hyperparameters are found using grid search and 10-fold cross-validation. The hyperparameters to be optimized are the distance metric and the p value used in the Minkowski distance metric.

Finally, the best model is evaluated on the test set. Accuracy, F1 score, and confusion matrix are used as evaluation metrics.

Feedforward Neural Network

The FFNN algorithm is tested with training, validation, and test sets. These sets contain 162,356, 40,588, and 50,736 data points respectively.

The training procedure is implemented as follows. First, The weights are initialized using He initialization, while the biases are initialized as zero. Then, a pass of forward propagation is completed. In forward propagation, starting from the input nodes, in each layer, the values are first multiplied by their respective weights, then summed, and finally passed through a nonlinear function before getting passed to the next layer. In this case, the nonlinear function used for all layers except the output layer is the rectified linear unit, while the output layer uses the sigmoid function. After forward propagation, Cross-entropy is used to calculate the cost. Finally, a pass of backward propagation is completed. In backward propagation, the error between the actual output and the output of forward propagation is calculated, and the weights of the network are then adjusted to minimize this error. In this process, gradient descent is used. The process described above is repeated until the cost delta between subsequent passes is smaller than a threshold.

For prediction, the inputs are fed into the network along with the trained parameters and after one pass of forward propagation, the predicted class is computed.

Optimal hyperparameters are found using grid search and a hold out validation set. Many modifications are possible, however, in this case, the hyperparameters to be optimized are the threshold value used in the sigmoid function, neuron count in a single hidden layer FFNN, and layer count in a deep FFNN.

During the optimization of the threshold value, the neural network contains one hidden layer with 21 neurons. During the optimization of the neuron count, the threshold is set to 0.2 and the neural network contains one hidden layer. During the optimization of the layer count, the threshold is set to 0.2 and each hidden layer in the neural network contains 10 neurons.

Finally, the best model is evaluated on the test set. Accuracy, F1 score, and confusion matrix are used as evaluation metrics.

Results

Logistic Regression

Hyperparameter tuning results are as follows:

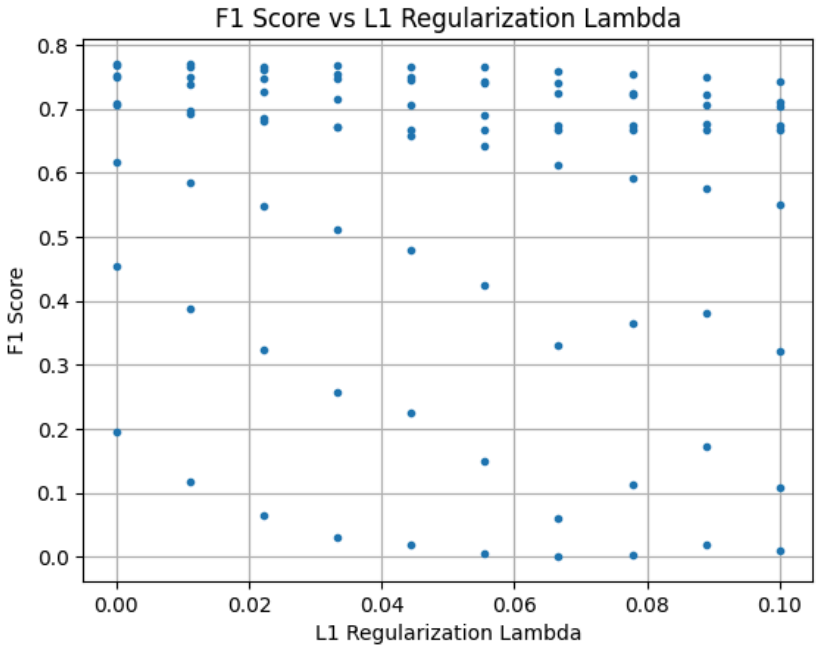

Figure 3: LR, balanced training set, F1 score vs regularization lambda.

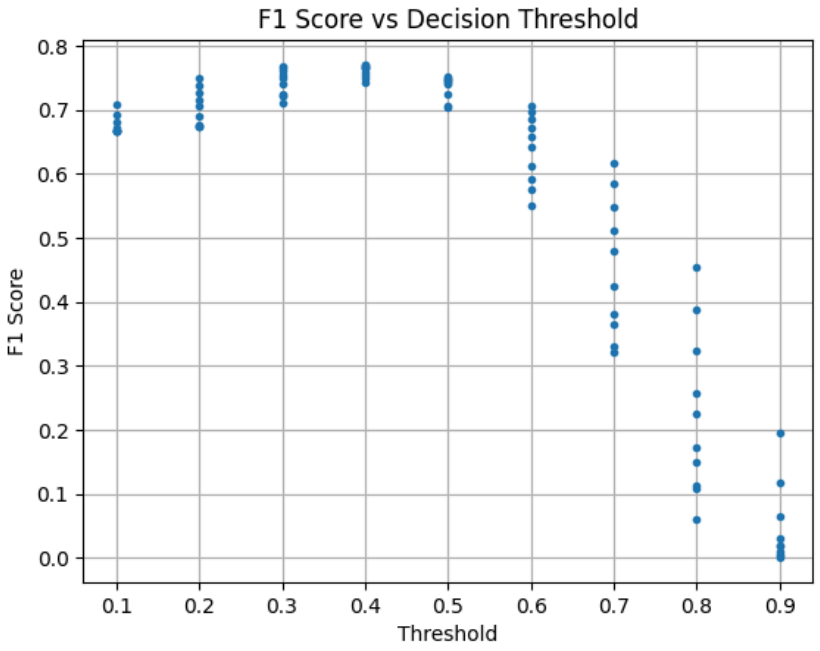

Figure 4: LR, balanced training set, F1 score vs decision threshold.

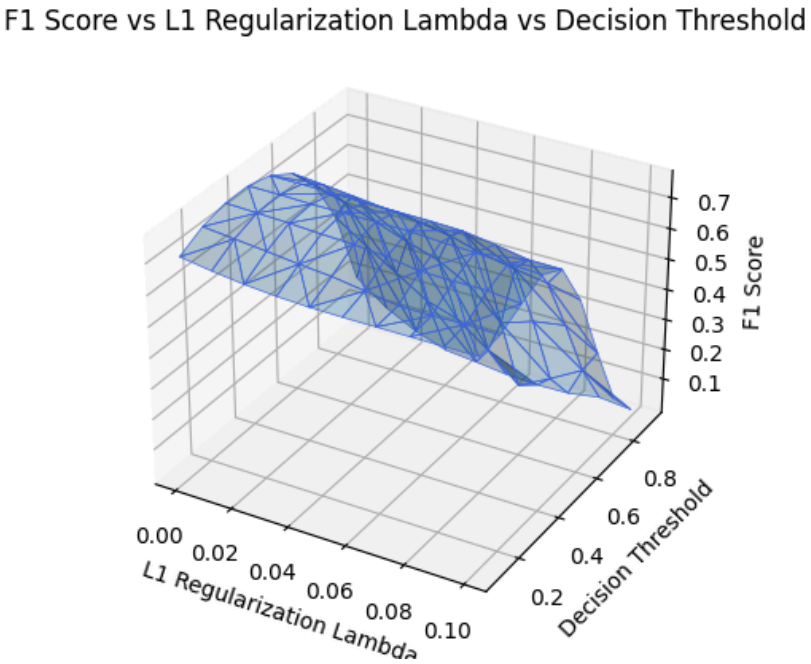

Figure 5: LR, balanced training set, F1 score vs regularization lambda vs threshold.

For the balanced training set model, the most optimal λ, threshold pair is **0**, **0.4**.

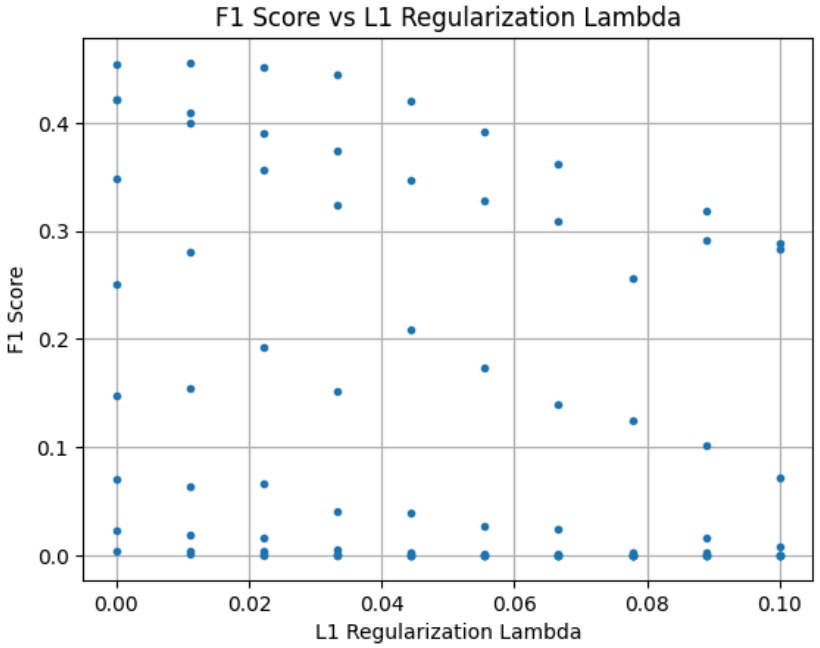

Figure 6: LR, imbalanced training set, F1 score vs regularization lambda.

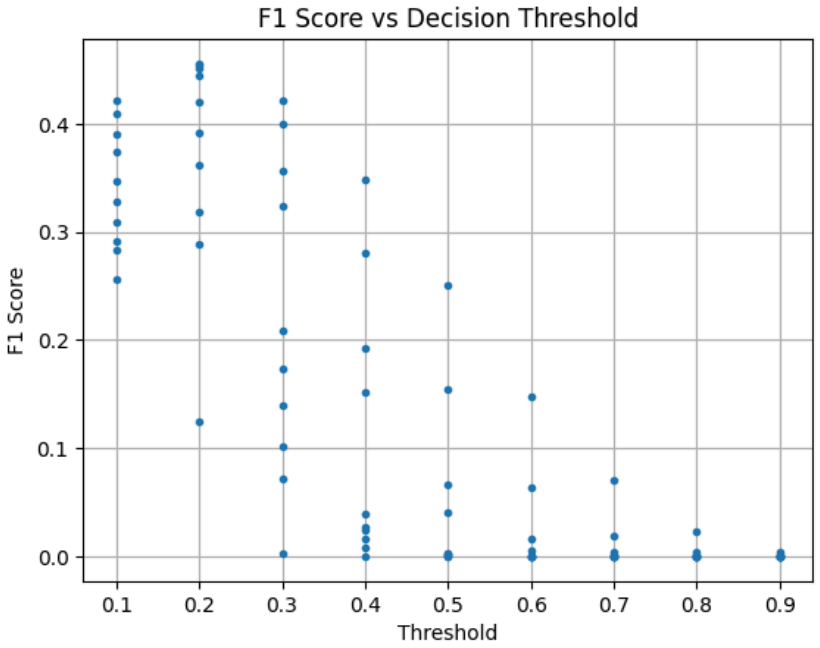

Figure 7: LR, imbalanced training set, F1 score vs decision threshold.

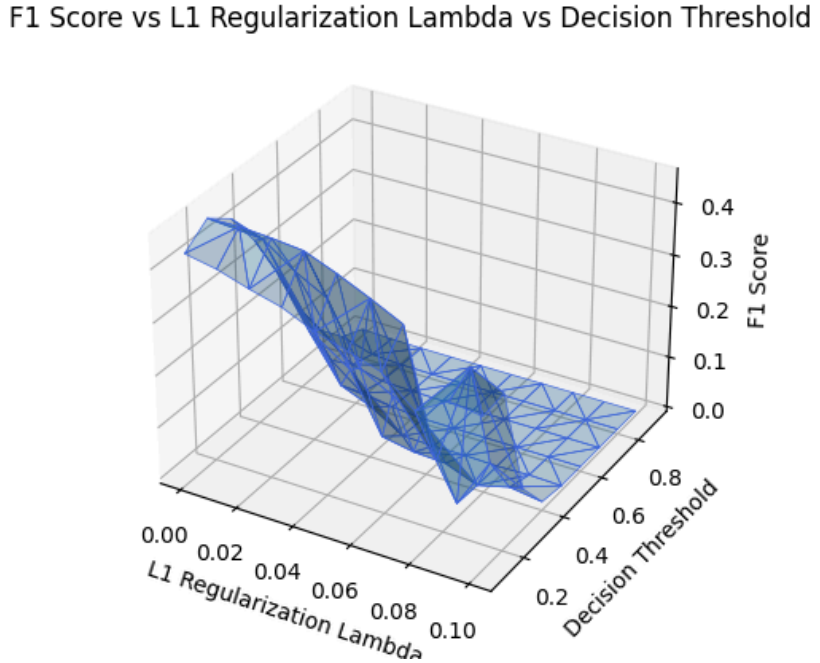

Figure 8: LR, imbalanced training set, F1 score vs regularization lambda vs threshold.

For the imbalanced training set model, the most optimal λ, threshold pair is **0.0111...**, **0.2**.

Final model results are as follows:

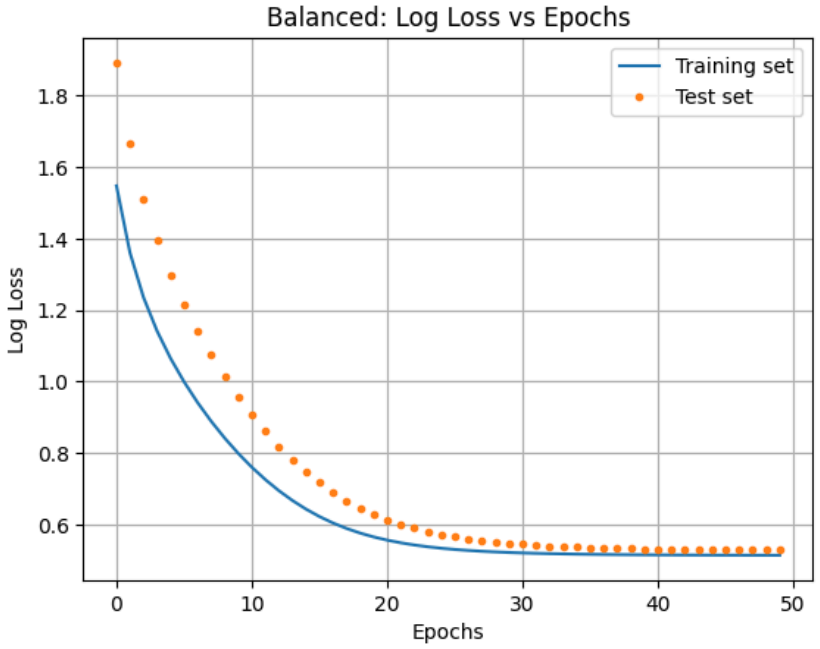

Figure 9: LR, balanced training set model, log loss vs epochs.

	Actual Positive	**Actual** Negative
Predicted Positive	6,051 (TP)	16,031 (FP)
Predicted Negative	1,036 (FN)	27,618 (TN)

Table 1: LR, balanced training set model, test set confusion matrix.

Balanced training set model test set accuracy: **0.66**

Balanced training set model test set F1 score: **0.41**

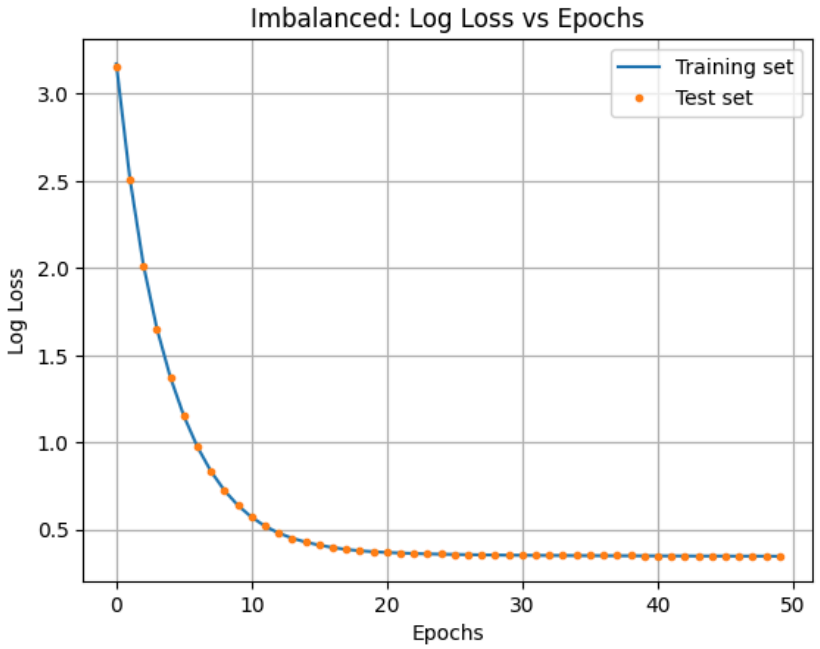

Figure 10: LR, imbalanced training set model, log loss vs epochs.

	Actual Positive	**Actual** Negative
Predicted Positive	4,386 (TP)	7,639 (FP)
Predicted Negative	2,701 (FN)	36,010 (TN)

Table 2: LR, imbalanced training set model, test set confusion matrix.

Imbalanced training set model test set accuracy: **0.80**

Imbalanced training set model test set F1 score: **0.46**

The *imbalanced training set model* has higher accuracy and higher F1 score on the test set.

K-Nearest Neighbors

Hyperparameter tuning results are as follows:

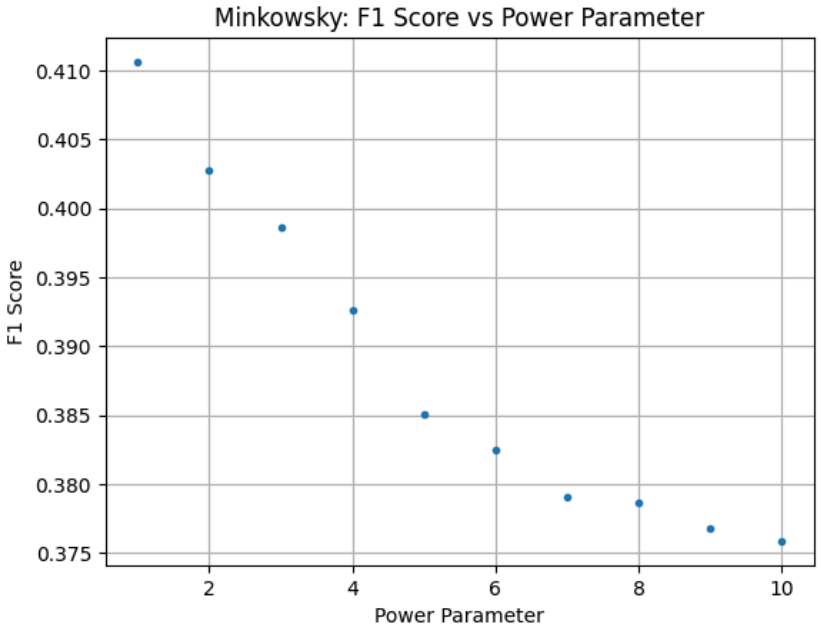

Figure 11: KNN, F1 score vs Minkowski power parameter.

F1 score of Minkowski power parameter seems to decrease with increasing p values. The optimal p value is **1**.

The 10-fold cross-validation F1 scores of Minkowski distance, cosine distance, and Chebyshev distance are respectively **0.41**, **0.39**, and **0.35**. The Minkowski distance metric is the optimal choice.

Final model results are as follows:

	Actual Positive	**Actual** Negative
Predicted Positive	865 (TP)	2,075 (FP)
Predicted Negative	259 (FN)	4,919 (TN)

Table 3: KNN, test set confusion matrix.

Test set accuracy: **0.66**

Test set F1 score: **0.41**

Considering accuracy and F1 score, KNN performs worse than LR. A much smaller data set may be causing this result.

Feedforward Neural Network

Hyperparameter tuning results are as follows:

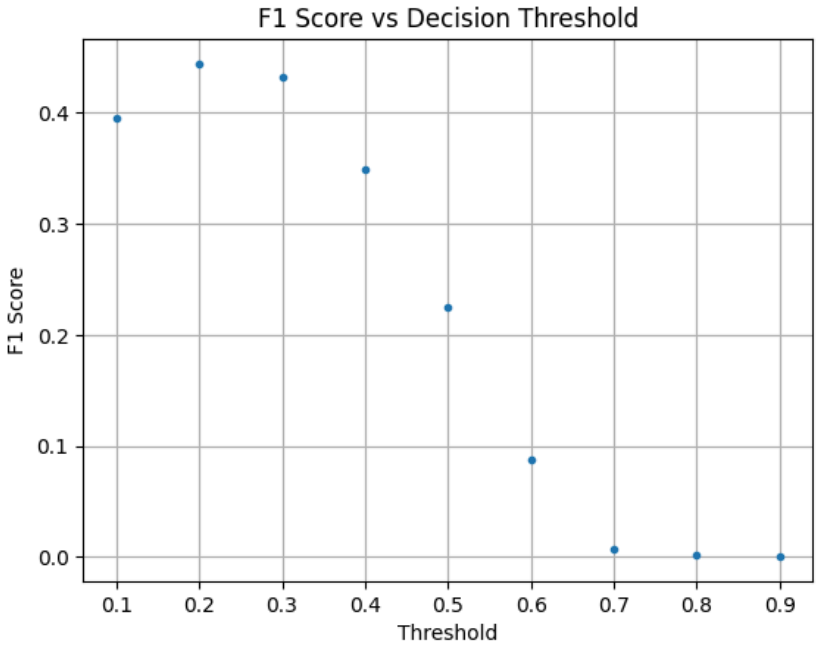

Figure 12: FFNN, F1 score vs decision threshold.

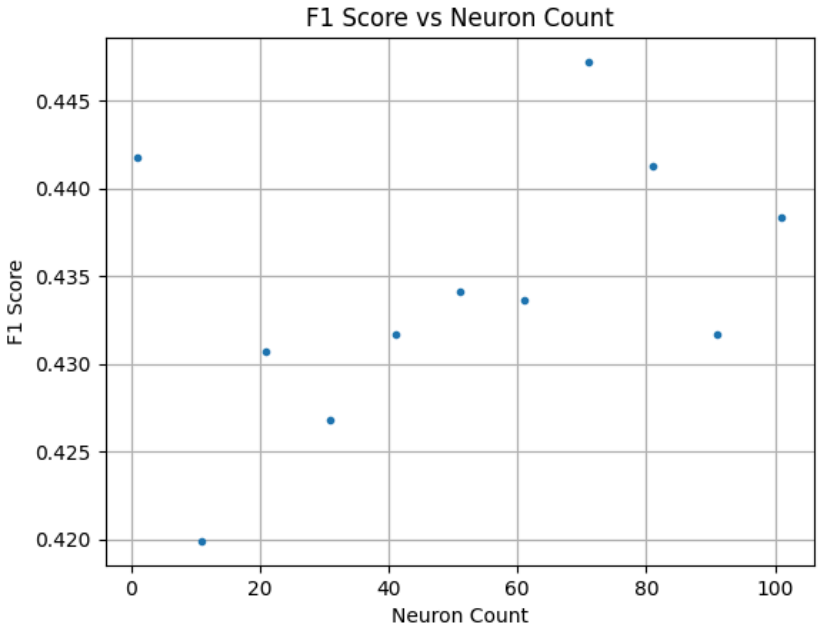

Figure 13: FFNN, F1 score vs neuron count.

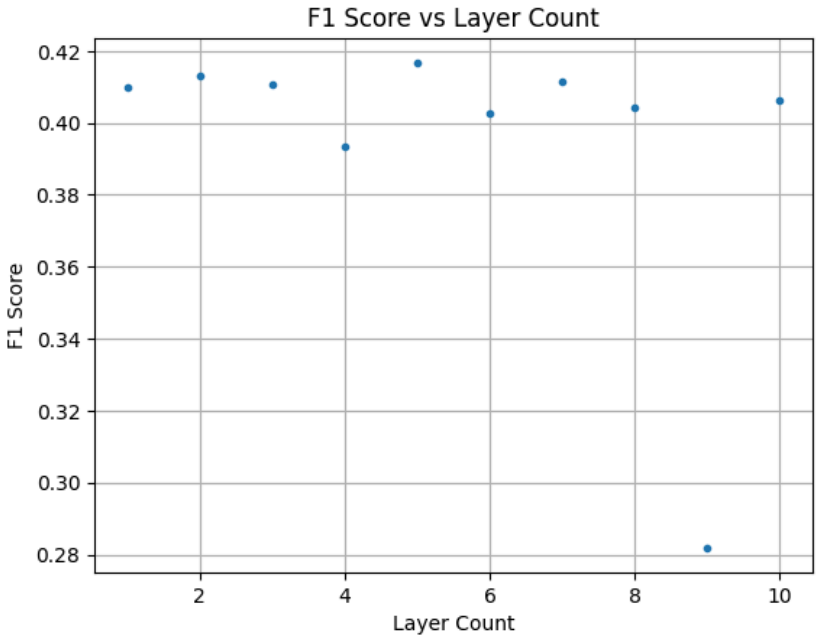

Figure 14: FFNN, F1 score vs layer count.

The best decision threshold is identical to LR, **0.2**. The F1 score seems to increase slightly with neuron count in a single hidden layer, and the layer count seems to have no effect.

Final model results are as follows:

Neural network of shape (**21, 21, 1**) (input, hidden, output):

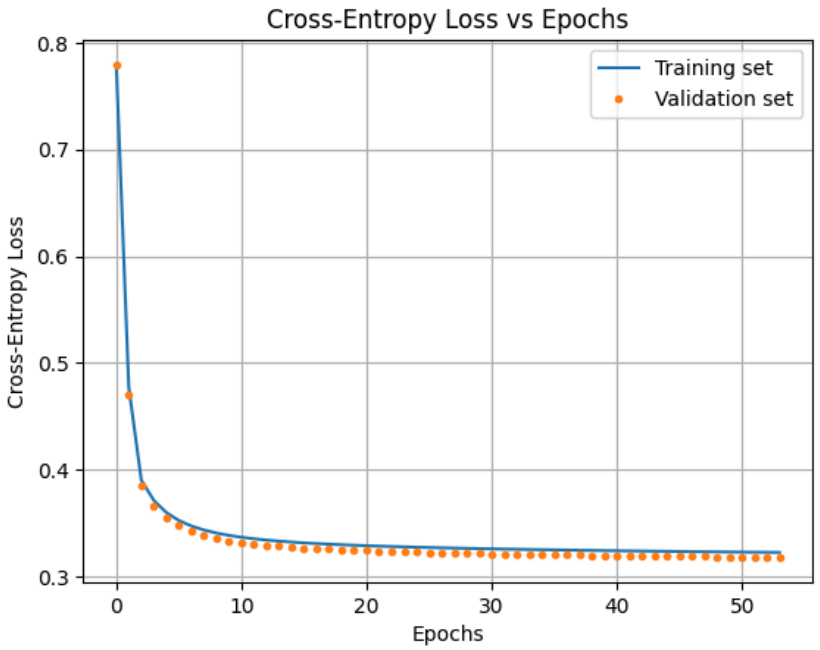

Figure 15: FFNN, cross-entropy loss vs epochs.

	Actual Positive	**Actual** Negative
Predicted Positive	4,737 (TP)	9,446 (FP)
Predicted Negative	2,214 (FN)	34,339 (TN)

Table 4: FFNN, test set confusion matrix.

Test set accuracy: **0.77**

Test set F1 score: **0.45**

Neural network of shape (**21, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 1**):

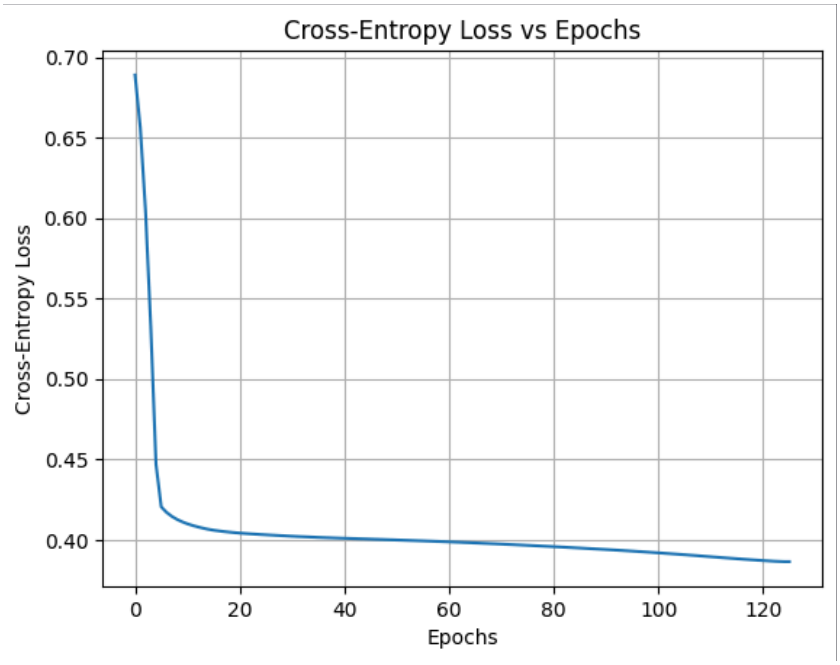

Figure 16: FFNN, cross-entropy loss vs epochs.

	Actual Positive	**Actual** Negative
Predicted Positive	2,037 (TP)	5,769 (FP)
Predicted Negative	4,971 (FN)	37,959 (TN)

Table 5: FFNN, test set confusion matrix.

Test set accuracy: **0.79**

Test set F1 score: **0.27**

Neural network of shape (**21, 1000, 1**):

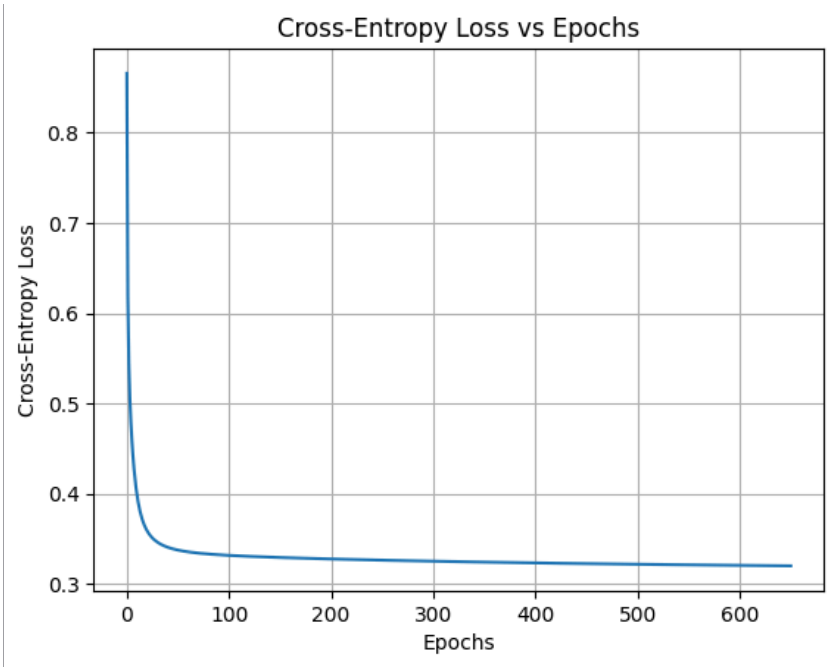

Figure 17: FFNN, cross-entropy loss vs epochs.

	Actual Positive	**Actual** Negative
Predicted Positive	4,661 (TP)	8,338 (FP)
Predicted Negative	2,497 (FN)	35,240 (TN)

Table 6: FFNN, test set confusion matrix.

Test set accuracy: **0.79**

Test set F1 score: **0.46**

The neural network of shape (21, 1000, 1) has the highest accuracy and the highest F1 score. However, it also took the longest to train out of all the neural network models, more than 70 minutes!

Conclusion

Out of the three machine learning methods, logistic regression performed the best. Examining the overall results, classification on an imbalanced dataset seems to pose a challenge when compared to a balanced dataset. When the dataset was balanced, the achievable F1 score reached 0.75 (figure 3 & 4). However, the balanced dataset condition does not reflect the reality of the situation, thus the model could not perform as well as the imbalanced one on the test set. The FFNN model not being able to beat the LR model was surprising. Wider and deeper networks may be necessary to beat the performance of the LR model. Finally, after implementing these algorithms from scratch, I learnt a lot about how they function and how to program them in Python.

Appendix

Logistic Regression

```
# %% [markdown]
# Alkım Ege Akarsu | 21901461 | EEE 485 | Term Project
# # Classification of Diabetic and Non-Diabetic Individuals Based on Survey
Data
# # Logistic Regression
# ## Package Imports

# %%
import numpy as np  # For math operations
import pandas as pd  # For importing and handling datasets
import matplotlib.pyplot as plt  # For plotting
from mpl_toolkits.mplot3d import axes3d  # For 3D component of plotting

# Pandas options
pd.set_option("display.max_columns", None)

# %% [markdown]
# ## Functions

# %%
# DATASET PREPARATION
def train_test_split(dataset, train_fraction):
      """Split the dataset into train and test sets

      Args:
      dataset (DataFrame): All available data points.
      Label and features as columns, Datapoints as rows.
      train_fraction (float): Fraction of data
      points to be added to the training set.

      Returns:
      DataFrame: Training dataset
      DataFrame: Test dataset
      """
      # Randomly sample for train
      train = dataset.sample(frac=train_fraction, axis="index")
      # Subtract train from original
      test = dataset.drop(index=train.index)
      # Reset indexes
      train = train.reset_index(drop=True)
      test = test.reset_index(drop=True)

      return train, test
```

```
def balance_dataset(dataset):
      """Returns a dataset with equal members of each class. Takes all
      data points of the less represented class.

      Args:
      dataset (DataFrame): Original dataset. Has shape (n, p).

      Returns:
      DataFrame: Balanced dataset. Has shape (_, p).
      """
      # Determine the less and more represented classes
      label_counts = dataset["Diabetes"].value_counts()
      least_frequent_class = label_counts.idxmin()
      most_frequent_class = label_counts.idxmax()
      # Get all data points of the less represented class
      least_frequent_class_data = dataset[dataset["Diabetes"] ==
least_frequent_class]
      # Get equal number of data points of the most represented class
      most_frequent_class_data = dataset[
      dataset["Diabetes"] == most_frequent_class
      ].sample(len(least_frequent_class_data.index))
      # Combine and shuffle the DataFrames to get the result
      result = pd.concat([least_frequent_class_data,
most_frequent_class_data])
      result = result.sample(frac=1).reset_index(drop=True)

      return result


def standardize(train, test):
      """Standardize X matrices of train and test splits. Uses the mean and
      standard deviation of the training set to standardize both training
      and test sets. This prevents data leakage between training and test
sets.

      Args:
      train (DataFrame): Training set.
      test (DataFrame): Test set.

      Returns:
      DataFrame: Standardized training set
      DataFrame: Standardized test set
      """
      # Get mean and standard deviation
      mean = train.mean()
      std = train.std()
      # Get results
```

```
    train_result = (train - mean) / std
    test_result = (test - mean) / std

    return train_result, test_result


# LOGISTIC REGRESSION
def initialize(X):
    """Initialize the parameters and the design matrix.

    Args:
    X (DataFrame): Design matrix

    Returns:
    ndarray: Design matrix with bias column
    ndarray: Randomly initialized parameter vector.
    Has shape (Parameter count + 1 (bias term), 1 ).
    """
    # Initialize parameters
    # Parameter count + 1 (bias term) rows, 1 column
    parameters = np.random.randn(X.shape[1] + 1, 1)
    # Add a column of ones to the beginning of the design matrix
    X_logistic = np.concatenate((np.ones((X.shape[0], 1)), X), axis=1)

    return X_logistic, parameters


def sigmoid(X, parameters):
    """Prediction values between 0 and 1 using the sigmoid function.

    Args:
    X (ndarray): Design matrix. Has shape (n, p + 1)
    parameters (ndarray): Parameter vector (weights). Has shape (p + 1, 1)

    Returns:
    ndarray: A prediction for every data point in the design matrix.
    Has shape (n, 1)
    """
    sigmoid = 1 / (1 + np.exp(-np.dot(X, parameters)))

    return sigmoid


def get_cost(X, y, parameters, lambda_reg):
    """L1 regularized log loss

    Args:
    X (ndarray): Design matrix. Has shape (n, p + 1)
```

```
        y (ndarray): Label vector. Has shape (n, 1)
        parameters (ndarray): Parameter vector (weights). Has shape (p + 1, 1)
        lambda_reg (double): L1 Regularization parameter

        Returns:
        double: L1 regularized log loss of the dataset
        """
        # Log loss + L1 regularization
        # h is the prediction
        h = sigmoid(X, parameters)
        # Log loss
        log_loss = -(y * np.log(h) + (1 - y) * np.log(1 - h)).mean()
        # L1 regularization (exclude bias term)
        regularization = lambda_reg * np.sum(np.abs(parameters[1:]))

        return log_loss + regularization


def train(
        X_train,
        y_train,
        X_test,
        y_test,
        parameters,
        learning_rate,
        iterations,
        lambda_reg,
        test=False,
):
        """Train the logistic regression model using gradient descent

        Args:
        X_train (ndarray): Design matrix. Has shape (n, p + 1)
        y_train (ndarray): Label vector. Has shape (n, 1)
        X_test (ndarray): Design matrix. Has shape (t, p + 1)
        y_test (ndarray): Label vector. Has shape (t, 1)
        parameters (ndarray): Parameter vector (weights). Has shape (p + 1, 1)
        learning_rate (double): Gradient descent learning rate
        iterations (int): How many steps to take in gradient descent
        lambda_reg (double): L1 Regularization parameter
        test (bool): Get costs on the test set after each iteration

        Returns:
        ndarray: Trained parameter vector (weights). Has shape (p + 1, 1)
        ndarray: Training costs for each epoch. Has shape (iterations,)
        ndarray: Test costs for each epoch. Has shape (iterations,)
        """
        # Initialize array of costs over time to return
```

```
	costs_train = np.zeros(iterations)
	costs_test = np.zeros(iterations)
	# Initialize copy of original parameters
	new_parameters = parameters.copy()
	# Start gradient descent
	for i in range(iterations):
	# h is the prediction
	h = sigmoid(X_train, new_parameters)
	# Gradient of log loss
	gradient_log = np.dot(X_train.T, (h - y_train)) / y_train.shape[0]
	# Gradient of L1 regularization (exclude bias term)
	gradient_L1_reg = np.concatenate(
		([0], lambda_reg * np.sign(new_parameters[1:].squeeze()))
	)
	gradient_L1_reg = gradient_L1_reg.reshape(-1, 1)
	# Total gradient
	gradient = gradient_log + gradient_L1_reg
	# New parameters
	new_parameters -= learning_rate * gradient
	# Add new cost to cost list
	costs_train[i] = get_cost(X_train, y_train, new_parameters, lambda_reg)
	if test == True:
		costs_test[i] = get_cost(X_test, y_test, new_parameters,
lambda_reg)
	# Print latest cost occasionally
	if i % 10 == 0:
		print(f"Iteration: {i}\n" f"Cost: {costs_train[i]}\n")
	# Print final cost
	print(f"\nFinal Cost: {costs_train[-1]}\n")

	return new_parameters, costs_train, costs_test


# MODEL EVALUATION
def predict(X, parameters, threshold):
	"""Predict labels using trained parameters.

	Args:
	X (ndarray): Feature values for each data point. Has shape (t, p + 1)
	parameters (ndarray): Trained parameter vector (weights).
	Has shape (p + 1, 1)
	threshold (double): Threshold determining class of data point.
	0 if under and 1 if over.

	Returns:
	ndarray: Predicted labels of all data points. Has shape (t, 1)
	"""
	probabilities = sigmoid(X, parameters)
```

```
    predictions = probabilities >= threshold
    predictions = predictions.astype(np.uint8)

    return predictions


def get_confusion_matrix(true, pred):
    """Calculate confusion matrix for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    ndarray: Confusion matrix
    TN FP
    FN TP
    """
    result = np.zeros((2, 2))

    for i in range(true.shape[0]):
    result[true[i][0]][pred[i][0]] += 1

    return result


def get_accuracy(true, pred):
    """Calculate accuracy for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    double: Accuracy value.
    """
    TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
    return (TP + TN) / (TP + TN + FP + FN)


def get_precision(true, pred):
    """Calculate precision for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
```

```
        double: Precision value.
        """
        TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
        return TP / (TP + FP)


def get_recall(true, pred):
        """Calculate recall for performance evaluation.

        Args:
        true (ndarray): Array of true labels. Has shape (t, 1).
        pred (ndarray): Array of predicted labels. Has shape (t, 1).

        Returns:
        double: Recall value.
        """
        TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
        return TP / (TP + FN)


def get_f1_score(true, pred):
        """Calculate F1 score for performance evaluation.

        Args:
        true (ndarray): Array of true labels. Has shape (t, 1).
        pred (ndarray): Array of predicted labels. Has shape (t, 1).

        Returns:
        double: F1 score.
        """
        TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
        return (2 * TP) / ((2 * TP) + FP + FN)


# HYPERPARAMETER TUNING
def cross_validation(
        X_train,
        y_train,
        X_test,  # Don't use
        y_test,  # Don't use
        parameters,  # Initialized parameter vector
        learning_rate,
        iterations,
        lambda_reg,
        K,
        threshold,
):
        """Evaluate model performance using K-fold cross-validation.
```

```
    Args:
    X_train (ndarray): Design matrix. Has shape (n, p + 1)
    y_train (ndarray): Label vector. Has shape (n, 1)
    X_test (ndarray): Design matrix. Has shape (t, p + 1) (Not used)
    y_test (ndarray): Label vector. Has shape (t, 1) (Not used)
    parameters (ndarray): Parameter vector (weights). Has shape (p + 1, 1)
    learning_rate (double): Gradient descent learning rate
    iterations (int): How many steps to take in gradient descent
    lambda_reg (double): L1 Regularization parameter
    K (int): Number of folds
    threshold (double): Threshold determining class of data point.

    Returns:
    double: Mean F1 score of all K folds
    """
    # Initialize f1 score storage
    f1_scores = np.zeros(K)
    # Split dataset into k folds
    X_folds = np.array_split(X_train, K)
    y_folds = np.array_split(y_train, K)
    # Loop over folds
    for i in range(K):
    # Get validation set
    X_validation_set = X_folds[i]
    y_validation_set = y_folds[i]
    # Get training set (list comprehension ftw)
    X_training_set = np.concatenate([X_folds[j] for j in range(K) if j !=
i])
    y_training_set = np.concatenate([y_folds[j] for j in range(K) if j !=
i])
    # Train model on training set
    trained_parameters, _, _ = train(
        X_training_set,
        y_training_set,
        X_test,  # Don't use
        y_test,  # Don't use
        parameters,  # Initialized parameter vector
        learning_rate,
        iterations,
        lambda_reg,
    )
    # Validate model on validation set
    predictions = predict(X_validation_set, trained_parameters, threshold)
    f1_score = get_f1_score(y_validation_set, predictions)
    # Save F1 score
    f1_scores[i] = f1_score
    # Return average F1 score
```

```
	return f1_scores.mean()


def tune_hyperparameters(
	X_train,
	y_train,
	X_test,
	y_test,
	parameters,
	learning_rate,
	iterations,
	lambda_reg_values,
	threshold_values,
	K,
):
	"""Try to find the best lambda_reg and threshold values using
	grid search and K-fold cross validation.

	Args:
	X_train (ndarray): Design matrix. Has shape (n, p + 1)
	y_train (ndarray): Label vector. Has shape (n, 1)
	X_test (ndarray): Design matrix. Has shape (t, p + 1) (Not used)
	y_test (ndarray): Label vector. Has shape (t, 1) (Not used)
	parameters (ndarray): Parameter vector (weights). Has shape (p + 1, 1)
	learning_rate (double): Gradient descent learning rate
	iterations (int): How many steps to take in gradient descent
	lambda_reg_values (ndarray): L1 Regularization parameters to try
	threshold_values (ndarray): Threshold values to try
	K (int): Number of folds

	Returns:
	double: Tuned lambda_reg value
	double: Tuned threshold value
	"""
	# Initialize results matrix
	# 3 columns: lambda_reg, threshold, score
	results = np.empty((0, 3))

	# Perform grid search
	# Loop over lambda_reg_values
	for lambda_reg in lambda_reg_values:
	# Loop over threshold_values
	for threshold in threshold_values:
		# Perform cross-validation
		f1_score = cross_validation(
			X_train,
			y_train,
			X_test,
```

```
                y_test,
                parameters,
                learning_rate,
                iterations,
                lambda_reg,
                K,
                threshold,
            )
            # Append lambda_reg, threshold, f1_score row to results matrix
            row = np.array([lambda_reg, threshold, f1_score])
            results = np.vstack((results, row))

    lambda_reg_results = results[:, 0].flatten()
    threshold_results = results[:, 1].flatten()
    f1_score_results = results[:, 2].flatten()
    # Plot scores against lambda_reg values
    plt.plot(lambda_reg_results, f1_score_results, ".")
    plt.title("F1 Score vs L1 Regularization Lambda")
    plt.xlabel("L1 Regularization Lambda")
    plt.ylabel("F1 Score")
    plt.grid(True)
    plt.show()
    # Plot scores against threshold values
    plt.plot(threshold_results, f1_score_results, ".")
    plt.title("F1 Score vs Decision Threshold")
    plt.xlabel("Threshold")
    plt.ylabel("F1 Score")
    plt.grid(True)
    plt.show()
    # Plot scores against lambda_reg values against threshold values
    plt.clf()
    ax = plt.figure().add_subplot(projection="3d")
    # Plot the 3D surface
    ax.plot_trisurf(
    lambda_reg_results,
    threshold_results,
    f1_score_results,
    edgecolor="royalblue",
    lw=0.5,
    alpha=0.3,
    )
    ax.set(
    xlabel="L1 Regularization Lambda",
    ylabel="Decision Threshold",
    zlabel="F1 Score",
    )
    plt.title("F1 Score vs L1 Regularization Lambda vs Decision Threshold")
    plt.show()
```

```
        # Get best_lambda_reg, best_threshold pair
        # Row index of max score
        row_index = np.argmax(results[:, 2])
        best_row = results[row_index]
        best_lambda_reg = best_row[0]
        best_threshold = best_row[1]

        return best_lambda_reg, best_threshold

# %% [markdown]
# ## Dataset Preparation

# %%
# Import dataset into a pandas dataframe
# Data points classified as "diabetes or pre-diabetes"(1) or "no diabetes"(0)
dataset = pd.read_csv("datasets/binary.csv", dtype="uint8")
# Dataset overview
dataset.info(verbose=True, show_counts=True)

# Divide dataset into training and test sets
train_fraction = 0.8
training, test = train_test_split(dataset, train_fraction)
# Create a balanced training set
training_balanced = balance_dataset(training)
# Number of data points
print(f"\n\nNumber of data points in train set: {len(training.index)}")
print(f"Number of data points in balanced train set:
{len(training_balanced.index)}")
print(f"Number of data points in test set: {len(test.index)}")

# Get X and Y from train, balanced_train, and test datasets
# Train
y_train = training["Diabetes"]
X_train = training.drop("Diabetes", axis=1)
# Balanced train
y_balanced_train = training_balanced["Diabetes"]
X_balanced_train = training_balanced.drop("Diabetes", axis=1)
# Test
y_test = test["Diabetes"]
X_test = test.drop("Diabetes", axis=1)

# Standardize Xs
X_train_standardized, X_test_standardized = standardize(X_train, X_test)
X_balanced_train_standardized, X_balanced_test_standardized = standardize(
        X_balanced_train, X_test
)
```

```
# Convert everything from dataframe to ndarray and reshape
# Train
X_train_standardized = X_train_standardized.to_numpy()
y_train = y_train.to_numpy().reshape(-1, 1)
# Balanced train
X_balanced_train_standardized = X_balanced_train_standardized.to_numpy()
y_balanced_train = y_balanced_train.to_numpy().reshape(-1, 1)
# Test
X_test_standardized = X_test_standardized.to_numpy()
y_test = y_test.to_numpy().reshape(-1, 1)
# Balanced test
X_balanced_test_standardized = X_balanced_test_standardized.to_numpy()

# %% [markdown]
# ## Hyperparameter Tuning
# ### Balanced Training Set

# %%
# Initialize necessary variables
X_balanced_train_logistic, parameters_balanced_logistic = initialize(
      X_balanced_train_standardized
)
X_balanced_test_logistic, _ = initialize(X_balanced_test_standardized)

# Get best hyperparameters (lambda and threshold)
learning_rate = 1
iterations = 50
K = 10
lamda_reg_values = np.linspace(0, 0.1, 10)
threshold_values = np.arange(0.1, 1.0, 0.1)  # 0.1, 0.2, ..., 0.8, 0.9
lambda_reg_balanced, threshold_balanced = tune_hyperparameters(
      X_balanced_train_logistic,
      y_balanced_train,
      X_balanced_test_logistic,
      y_test,
      parameters_balanced_logistic,
      learning_rate,
      iterations,
      lamda_reg_values,
      threshold_values,
      K,
)

# Print the results
print(
      f"Best lambda_reg, threshold combination: {lambda_reg_balanced},
{threshold_balanced}"
)
```

```
# %% [markdown]
# ### Imbalanced Training Set

# %%
# Initialize necessary variables
X_train_logistic, parameters_logistic = initialize(X_train_standardized)
X_test_logistic, _ = initialize(X_test_standardized)

# Get best hyperparameters (lambda and threshold)
learning_rate = 1
iterations = 50
K = 10
lamda_reg_values = np.linspace(0, 0.1, 10)
threshold_values = np.arange(0.1, 1.0, 0.1)  # 0.1, 0.2, ..., 0.8, 0.9
lambda_reg, threshold = tune_hyperparameters(
      X_train_logistic,
      y_train,
      X_test_logistic,
      y_test,
      parameters_logistic,
      learning_rate,
      iterations,
      lamda_reg_values,
      threshold_values,
      K
)

# Print the results
print(f"Best lambda_reg, threshold combination: {lambda_reg}, {threshold}")

# %% [markdown]
# ## Final Models and Results
# ### Balanced Training Set

# %%
# Initialize necessary variables
X_balanced_train_logistic, parameters_balanced_logistic = initialize(
      X_balanced_train_standardized
)
X_balanced_test_logistic, _ = initialize(X_balanced_test_standardized)

# Train logistic model
learning_rate = 1
iterations = 50
(parameters_balanced_logistic,
costs_balanced_train_logistic,
costs_balanced_test_logistic) = train(
```

```
      X_balanced_train_logistic,
      y_balanced_train,
      X_balanced_test_logistic,
      y_test,
      parameters_balanced_logistic,
      learning_rate,
      iterations,
      lambda_reg=lambda_reg_balanced,
      test=True,
)

# Plot loss functions over time
plt.plot(costs_balanced_train_logistic, label="Training set")
plt.plot(costs_balanced_test_logistic, ".", label="Test set")
plt.title("Balanced: Log Loss vs Epochs")
plt.xlabel("Epochs")
plt.ylabel("Log Loss")
plt.legend()
plt.grid(True)
plt.show()

# Get evaluation metrics on test set
predictions = predict(
      X_balanced_test_logistic, parameters_balanced_logistic,
threshold=threshold_balanced
)
trained_accuracy = get_accuracy(y_test, predictions)
trained_confusion_matrix = get_confusion_matrix(y_test, predictions)
trained_f1_score = get_f1_score(y_test, predictions)
TN, FP, FN, TP = get_confusion_matrix(y_test, predictions).ravel()
print(
      f"Accuracy: {trained_accuracy}\n"
      f"F1-score: {trained_f1_score}\n"
      f"TN, FP, TP, FN: {TN}, {FP}, {TP}, {FN}\n"
)

# %% [markdown]
# ### Imbalanced Training Set

# %%
# Initialize necessary variables
X_train_logistic, parameters_logistic = initialize(X_train_standardized)
X_test_logistic, _ = initialize(X_test_standardized)

# Train logistic model
learning_rate = 1
iterations = 50
parameters_logistic, costs_train_logistic, costs_test_logistic = train(
```

```
      X_train_logistic,
      y_train,
      X_test_logistic,
      y_test,
      parameters_logistic,
      learning_rate,
      iterations,
      lambda_reg=lambda_reg,
      test=True,
)

# Plot loss functions over time
plt.plot(costs_train_logistic, label="Training set")
plt.plot(costs_test_logistic, ".", label="Test set")
plt.title("Imbalanced: Log Loss vs Epochs")
plt.xlabel("Epochs")
plt.ylabel("Log Loss")
plt.legend()
plt.grid(True)
plt.show()

# Get evaluation metrics on test set
predictions = predict(X_test_logistic, parameters_logistic,
threshold=threshold)
trained_accuracy = get_accuracy(y_test, predictions)
trained_confusion_matrix = get_confusion_matrix(y_test, predictions)
trained_f1_score = get_f1_score(y_test, predictions)
TN, FP, FN, TP = get_confusion_matrix(y_test, predictions).ravel()
print(
      f"Accuracy: {trained_accuracy}\n"
      f"F1-score: {trained_f1_score}\n"
      f"TN, FP, TP, FN: {TN}, {FP}, {TP}, {FN}\n"
)
```

K-Nearest Neighbors

```
# %% [markdown]
# Alkım Ege Akarsu | 21901461 | EEE 485 | Term Project
# # Classification of Diabetic and Non-Diabetic Individuals Based on Survey
Data
# # K-Nearest Neighbors
# ## Package Imports

# %%
import numpy as np  # For math operations
import pandas as pd  # For importing and handling datasets
import matplotlib.pyplot as plt  # For plotting
from mpl_toolkits.mplot3d import axes3d  # For 3D component of plotting
```

```
# Pandas options
pd.set_option('display.max_columns', None)

# %% [markdown]
# ## Functions

# %%
# DATASET PREPARATION
def train_test_split(dataset, train_fraction):
      """Split the dataset into train and test sets

      Args:
      dataset (DataFrame): All available data points.
      Label and features as columns, Datapoints as rows.
      train_fraction (float): Fraction of data
      points to be added to the training set.

      Returns:
      DataFrame: Training dataset
      DataFrame: Test dataset
      """
      # Randomly sample for train
      train = dataset.sample(frac=train_fraction, axis="index")
      # Subtract train from original
      test = dataset.drop(index=train.index)
      # Reset indexes
      train = train.reset_index(drop=True)
      test = test.reset_index(drop=True)

      return train, test


def mini_dataset(dataset, fraction):
      """Return a fraction of the dataset.

      Args:
      dataset (DataFrame): Original dataset. Has shape (n, p).
      fraction (double): Fraction of dataset to return.

      Returns:
      DataFrame: Smaller dataset. Has shape (n / fraction, p).
      """
      return dataset.sample(frac=fraction,
                        axis="index",
                        ignore_index=True)


def standardize(train, test):
```

```
    """Standardize X matrices of train and test splits. Uses the mean and
    standard deviation of the training set to standardize both training
    and test sets. This prevents data leakage between training and test
sets.

    Args:
    train (DataFrame): Training set.
    test (DataFrame): Test set.

    Returns:
    DataFrame: Standardized training set
    DataFrame: Standardized test set
    """
    # Get mean and standard deviation
    mean = train.mean()
    std = train.std()
    # Get results
    train_result = (train - mean) / std
    test_result = (test - mean) / std

    return train_result, test_result


def range_scaling(train, test):
    """Scale X matrices of train and test splits. Uses the min and
    max values of the training set to scale both training
    and test sets. This prevents data leakage between training and test
sets.

    Args:
    train (DataFrame): Training set.
    test (DataFrame): Test set.

    Returns:
    DataFrame: Standardized training set
    DataFrame: Standardized test set
    """
    # Get min and max values
    min_val = train.min()
    max_val = train.max()
    # Get results
    train_result = (train - min_val) / (max_val - min_val)
    test_result = (test - min_val) / (max_val - min_val)

    return train_result, test_result


# K-NEAREST NEIGHBORS
```

```
def get_minkowski_distance(X, row_index, p):
      """Calculate the minkowski distance between
      a point and every point in X. INCLUDES THE POINT ITSELF.

      Args:
      X (ndarray): Feature matrix. Has shape (n, p).
      row_index (int): Index of point in X.
      p (int): The power parameter

      Returns:
      ndarray: Distances between the given point and every point in X.
      Index [n] has the distance between the given point and point
      at index n in X. Has shape (n,).
      """
      # Get the point of interest
      point = X[row_index]

      # Calculate the Minkowski distance
      distances = np.sum(np.abs(X - point)**p, axis=1)**(1/p)

      return distances


def get_cosine_distance(X, row_index):
      """Calculate the cosine distance between
      a point and every point in X. INCLUDES THE POINT ITSELF.

      Args:
      X (ndarray): Feature matrix. Has shape (n, p).
      row_index (int): Index of point in X.

      Returns:
      ndarray: Distances between the given point and every point in X.
      Index [n] has the distance between the given point and point
      at index n in X. Has shape (n,).
      """
      # Get the point of interest
      point = X[row_index]

      # Normalize the vectors
      X_norm = np.linalg.norm(X, axis=1)
      point_norm = np.linalg.norm(point)

      # Calculate the cosine similarity
      cosine_similarities = np.dot(X, point) / (X_norm * point_norm)

      # Convert cosine similarities to cosine distances
      cosine_distances = 1 - cosine_similarities
```

```
        return cosine_distances


def get_chebyshev_distance(X, row_index):
        """Calculate the chebyshev distance between
        a point and every point in X. INCLUDES THE POINT ITSELF.

        Args:
        X (ndarray): Feature matrix. Has shape (n, p).
        row_index (int): Index of point in X.

        Returns:
        ndarray: Distances between the given point and every point in X.
        Index [n] has the distance between the given point and point
        at index n in X. Has shape (n,).
        """
        # Get the point of interest
        point = X[row_index]

        # Calculate the Chebyshev distance
        distances = np.max(np.abs(X - point), axis=1)

        return distances


def get_k_nearest_labels(X, y, k, get_distance, p_minkowski=None):
        """Get the labels of the k nearest neighbors of every point in X.

        Args:
        X (ndarray): Feature matrix. Has shape (n, p).
        y (ndarray): Labels. Has shape (n, 1).
        k (int): Number of nearest neighbors.
        get_distance (function): Distance function to be used.
        p_minkowski (optional(int)): Minkowski distance parameter.

        Returns:
        ndarray: Labels of the nearest neighbors. Has shape (n, k).
        """
        # Initialize an empty ndarray to store the labels of the k nearest
neighbors
        nearest_labels = np.empty((X.shape[0], k))

        # Iterate over each point in X
        for i in range(X.shape[0]):
        # Print progress
        if i % 100 == 0:
                print(f"Point: {i}\n")
```

```
        # Calculate the distances from the point to all other points
        if p_minkowski == None:
                distances = get_distance(X, i)
        else:
                distances = get_distance(X, i, p_minkowski)

        # Get the indices of the k nearest NEIGHBORS
        nearest_indices = np.argsort(distances)[1 : k + 1]

        # Get the labels of the k nearest neighbors
        nearest_y = y[nearest_indices]

        # Store the labels in the ndarray
        nearest_labels[i, :] = nearest_y.ravel()

        return nearest_labels


def get_k_nearest_labels_point(X, y, k, row_index, get_distance,
p_minkowski=None):
        """Get the labels of the k nearest neighbors of given row index.

        Args:
        X (ndarray): Feature matrix. Has shape (n, p).
        y (ndarray): Labels. Has shape (n, 1).
        k (int): Number of nearest neighbors.
        row_index (int): Index of point in X.
        get_distance (function): Distance function to be used.
        p_minkowski (optional(int)): Minkowski distance parameter.

        Returns:
        ndarray: Labels of the nearest neighbors. Has shape (k, 1).
        """
        # Calculate the distances from the point to all other points
        if p_minkowski == None:
        distances = get_distance(X, row_index)
        else:
        distances = get_distance(X, row_index, p_minkowski)

        # Get the indices of the k nearest neighbors
        nearest_indices = np.argsort(distances)[1 : k + 1]

        # Get the labels of the k nearest neighbors
        nearest_y = y[nearest_indices]

        # Return the labels
        return nearest_y.ravel()
```

```
def get_weights(y):
      """Determine the weights of each class in relation to their frequency.
      Class weights are inversely proportional to the frequency of each
class.

      Args:
      y (ndarray): Labels. Has shape (n, 1).

      Returns:
      tuple(double, double): Weights of respective classes.
      """
      # Count the occurrence of each label
      counts = np.bincount(y.astype("int").flatten())

      # Get the total number of labels
      total = y.shape[0]

      # Determine the weights of each class in relation to their frequency
      weight_of_0 = total / counts[0]
      weight_of_1 = total / counts[1]

      return (weight_of_0, weight_of_1)


def get_k_value(X):
      """Returns the k value to be used for k-nearest neighbors.

      Args:
      X (ndarray): Feature matrix. Has shape (n, p).

      Returns:
      int: k value
      """
      # Optimal value of k is the square root of the number of points
      k_opt = np.sqrt(X.shape[0])
      # Return closest odd number
      lower = int(k_opt) - int(k_opt) % 2 + 1
      upper = lower + 2
      return lower if k_opt - lower < upper - k_opt else upper


def predict_from_same_set(X, y, k, weights, get_distance, weighted=True,
p_minkowski=None):
      """Predict the class of every point (row) in X.

      Args:
```

```
      X (ndarray): Feature matrix. Has shape (n, p).
      y (ndarray): Labels. Has shape (n, 1).
      k (int): Number of nearest neighbors.
      weights (tuple(double, double)): Weights of respective classes.
      get_distance (function): Distance function to be used.
      weighted (bool): If True, use weighted voting.
      p_minkowski (optional(int)): Minkowski distance parameter.

      Returns:
      ndarray: Predicted labels. Has shape (n, 1).
      """
      # Get the labels of the k nearest neighbors for each point in X
      if p_minkowski == None:
      nearest_labels = get_k_nearest_labels(X, y, k, get_distance)
      else:
      nearest_labels = get_k_nearest_labels(X, y, k, get_distance,
p_minkowski)

      # Initialize an empty ndarray to store the predicted labels
      predicted_labels = np.empty(X.shape[0], dtype=np.uint8)

      # Iterate over each point in X
      for i in range(X.shape[0]):
      # Get the labels of the k nearest neighbors for the point
      labels = nearest_labels[i, :]

      # Count the occurrence of each label
      counts = np.bincount(labels.astype("int"))

      # If weighted voting is enabled, multiply the counts by the weights
      if weighted:
            for j in range(counts.shape[0]):
                  counts[j] *= weights[j]

      # Get the label with the highest occurrence
      predicted_label = np.argmax(counts)

      # Store the predicted label in the ndarray
      predicted_labels[i] = predicted_label

      return predicted_labels.reshape(-1, 1)


def predict_from_different_set(X_train, y_train, X_valid, y_valid, k,
weights, get_distance, weighted=True, p_minkowski=None):
      """Predict the class of every point (row) in X_valid. Picks point from
      X_valid and puts it in X_train for prediction.
```

```
      Args:
      X_train (ndarray): Training feature matrix. Has shape (n, p).
      y_train (ndarray): Training labels. Has shape (n, 1).
      X_valid (ndarray): Validation feature matrix. Has shape (m, p).
      y_valid (ndarray): Validation labels. Has shape (m, 1).
      k (int): Number of nearest neighbors.
      weights (tuple(double, double)): Weights of respective classes.
      get_distance (function): Distance function to be used.
      weighted (bool, optional): If True, use weighted voting.
      p_minkowski (optional(int)): Minkowski distance parameter.

      Returns:
      ndarray: Predicted labels. Has shape (m, 1).
      """
      # Initialize an empty ndarray to store the predicted labels
      predicted_labels = np.empty(X_valid.shape[0], dtype=np.uint8)

      # Iterate over each point in X_valid
      for i in range(X_valid.shape[0]):

      # Append the point to the end of X_train
      X = np.vstack((X_train, X_valid[i, :]))
      y = np.vstack((y_train, y_valid[i, :]))

      # Get the labels of the k nearest neighbors for the point
      row_index = -1
      if p_minkowski == None:
            nearest_labels = get_k_nearest_labels_point(X, y, k, row_index,
get_distance)
      else:
            nearest_labels = get_k_nearest_labels_point(X, y, k, row_index,
get_distance, p_minkowski)

      # Count the occurrence of each label
      counts = np.bincount(nearest_labels.astype("int"))

      # If weighted voting is enabled, multiply the counts by the weights
      if weighted:
            for j in range(counts.shape[0]):
                  counts[j] *= weights[j]

      # Get the label with the highest occurrence
      predicted_label = np.argmax(counts)

      # Store the predicted label in the ndarray
      predicted_labels[i] = predicted_label

      return predicted_labels.reshape(-1, 1)
```

```
# MODEL EVALUATION
def get_confusion_matrix(true, pred):
      """Calculate confusion matrix for performance evaluation.

      Args:
      true (ndarray): Array of true labels. Has shape (t, 1).
      pred (ndarray): Array of predicted labels. Has shape (t, 1).

      Returns:
      ndarray: Confusion matrix
      TN FP
      FN TP
      """
      result = np.zeros((2, 2))

      for i in range(true.shape[0]):
      result[true[i][0]][pred[i][0]] += 1

      return result


def get_accuracy(true, pred):
      """Calculate accuracy for performance evaluation.

      Args:
      true (ndarray): Array of true labels. Has shape (t, 1).
      pred (ndarray): Array of predicted labels. Has shape (t, 1).

      Returns:
      double: Accuracy value.
      """
      TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
      return (TP + TN) / (TP + TN + FP + FN)


def get_precision(true, pred):
      """Calculate precision for performance evaluation.

      Args:
      true (ndarray): Array of true labels. Has shape (t, 1).
      pred (ndarray): Array of predicted labels. Has shape (t, 1).

      Returns:
      double: Precision value.
      """
      TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
```

```
    return TP / (TP + FP)


def get_recall(true, pred):
    """Calculate recall for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    double: Recall value.
    """
    TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
    return TP / (TP + FN)


def get_f1_score(true, pred):
    """Calculate F1 score for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    double: F1 score.
    """
    TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
    return (2 * TP) / ((2 * TP) + FP + FN)


# HYPERPARAMETER TUNING
def cross_validation(X_train, y_train, k, weights, get_distance, K,
weighted=True, p_minkowski=None):
    """Evaluate model performance using K-fold cross-validation.

    Args:
    X_train (ndarray): Training feature matrix. Has shape (n, p).
    y_train (ndarray): Training labels. Has shape (n, 1).
    k (int): Number of nearest neighbors.
    weights (tuple(double, double)): Weights of respective classes.
    get_distance (function): Distance function to be used.
    K (int): Number of folds
    weighted (bool, optional): If True, use weighted voting.
    p_minkowski (optional(int)): Minkowski distance parameter.

    Returns:
    double: Mean F1 score of all K folds
```

```
	"""
	# Initialize f1 score storage
	f1_scores = np.zeros(K)
	# Split dataset into k folds
	X_folds = np.array_split(X_train, K)
	y_folds = np.array_split(y_train, K)
	# Loop over folds
	for i in range(K):
	# Get validation set
	X_validation_set = X_folds[i]
	y_validation_set = y_folds[i]
	# Get training set (list comprehension ftw)
	X_training_set = np.concatenate([X_folds[j] for j in range(K) if j != 
i])
	y_training_set = np.concatenate([y_folds[j] for j in range(K) if j != 
i])
	# Validate model on validation set
	if p_minkowski is None:
		predictions = predict_from_different_set(X_training_set, 
y_training_set, X_validation_set, y_validation_set, k, weights, get_distance, 
weighted)
	else:
		predictions = predict_from_different_set(X_training_set, 
y_training_set, X_validation_set, y_validation_set, k, weights, get_distance, 
weighted, p_minkowski)
	f1_score = get_f1_score(y_validation_set, predictions)
	# Save F1 score
	f1_scores[i] = f1_score
	# Return average F1 score
	return f1_scores.mean()


def tune_minkowski(X_train, y_train, weights, K, p_values, k_values, 
weighted=True):
	"""Try to find the best p and k values using grid search
	and K-fold cross validation.

	Args:
	X_train (ndarray): Training feature matrix. Has shape (n, p).
	y_train (ndarray): Training labels. Has shape (n, 1).
	weights (tuple(double, double)): Weights of respective classes.
	K (int): Number of folds
	p_values (ndarray of ints): List of p values to try.
	k_values (ndarray): List of k values to try.
	weighted (bool, optional): If True, use weighted voting.

	Returns:
	tuple of ints: Best value of p, best value of k
```

```
    """
    # Initialize results matrix
    # 3 columns: p, k, score
    results = np.empty((0, 3))

    # Perform grid search
    # Loop over p
    for p in p_values:
    # Loop over k
    for k in k_values:
        # Print progress
        print(f"Grid search current values: p:{p}, k:{k}\n")
        # Perform cross-validation
        f1_score = cross_validation(X_train, y_train, k, weights,
get_minkowski_distance, K, weighted, p)
        # Append p, k, f1_score row to results matrix
        row = np.array([p, k, f1_score])
        results = np.vstack((results, row))

    p_results = results[:, 0].flatten()
    k_results = results[:, 1].flatten()
    f1_score_results = results[:, 2].flatten()
    # Plot scores against p values
    plt.plot(p_results, f1_score_results, ".")
    plt.title("Minkowsky: F1 Score vs Power Parameter")
    plt.xlabel("Power Parameter")
    plt.ylabel("F1 Score")
    plt.grid(True)
    plt.show()
    # # Plot scores against k values
    # plt.plot(k_results, f1_score_results, ".")
    # plt.title("Minkowsky: F1 Score vs Neighbor Count")
    # plt.xlabel("Neighbor Count")
    # plt.ylabel("F1 Score")
    # plt.grid(True)
    # plt.show()
    # # Plot scores against p values against k values
    # plt.clf()
    # ax = plt.figure().add_subplot(projection="3d")
    # # Plot the 3D surface
    # ax.plot_trisurf(
    #     p_results,
    #     k_results,
    #     f1_score_results,
    #     edgecolor="royalblue",
    #     lw=0.5,
    #     alpha=0.3,
    # )
```

```
    # ax.set(
    #     xlabel="Power Parameter",
    #     ylabel="Neighbor Count",
    #     zlabel="F1 Score",
    # )
    # plt.title("Minkowsky: F1 Score vs Power Parameter vs Neighbor Count")
    # plt.show()

    # Get p, k pair
    # Row index of max score
    row_index = np.argmax(results[:, 2])
    best_row = results[row_index]
    best_p = best_row[0]
    best_k = best_row[1]

    return best_p, best_k


def tune_other_distance(X_train, y_train, weights, K, k_values, get_distance,
weighted=True):
    """Try to find the best k value using grid search and K-fold cross
validation.

    Args:
    X_train (ndarray): Training feature matrix. Has shape (n, p).
    y_train (ndarray): Training labels. Has shape (n, 1).
    weights (tuple(double, double)): Weights of respective classes.
    K (int): Number of folds
    k_values (ndarray): List of k values to try.
    get_distance (function): Distance function to be used.
    weighted (bool, optional): If True, use weighted voting.

    Returns:
    int: Best value of k
    """
    # Initialize results matrix
    # 2 columns: k, score
    results = np.empty((0, 2))

    # Perform grid search
    # Loop over k
    for k in k_values:
    # Print progress
    print(f"Grid search current value: {k}\n")
    # Perform cross-validation
    f1_score = cross_validation(X_train, y_train, k, weights, get_distance,
K, weighted)
    # Append k, f1_score row to results matrix
```

```
        row = np.array([k, f1_score])
        results = np.vstack((results, row))

        k_results = results[:, 0].flatten()
        f1_score_results = results[:, 1].flatten()
        # Plot scores against k values
        plt.plot(k_results, f1_score_results)
        if get_distance == get_cosine_distance:
        plt.title("Cosine: F1 Score vs Neighbor Count")
        else:
        plt.title("Chebyshev: F1 Score vs Neighbor Count")
        plt.xlabel("Neighbor Count")
        plt.ylabel("F1 Score")
        plt.grid(True)
        plt.show()

        # Get best k value
        # Row index of max score
        row_index = np.argmax(results[:, 1])
        best_row = results[row_index]
        best_k = best_row[0]

        return best_k

# %% [markdown]
# ## Dataset Preparation

# %%
# Import dataset into a pandas dataframe
# Data points classified as "diabetes or pre-diabetes"(1) or "no diabetes"(0)
dataset = pd.read_csv("datasets/binary.csv", dtype="uint8")
# Dataset overview
dataset.info(verbose=True, show_counts=True)

# Divide dataset into training and test sets
train_fraction = 0.8
training, test = train_test_split(dataset, train_fraction)
# Get smaller datasets (computationally unfeaseble otherwise)
training = mini_dataset(training, 0.04)
test = mini_dataset(test, 0.16)
# Number of data points
print(f"\n\nNumber of data points in train set: {len(training.index)}")
print(f"Number of data points in test set: {len(test.index)}")

# Get X and Y from train and test datasets
# Train
y_train = training["Diabetes"]
X_train = training.drop("Diabetes", axis=1)
```

```
# Test
y_test = test["Diabetes"]
X_test = test.drop("Diabetes", axis=1)

# Standardize X
X_train_standardized, X_test_standardized = standardize(X_train, X_test)
# Range scale X
X_train_scaled, X_test_scaled = range_scaling(X_train, X_test)

# Convert everything from dataframe to ndarray and reshape
# Train
X_train_standardized = X_train_standardized.to_numpy()
X_train_scaled = X_train_scaled.to_numpy()
y_train = y_train.to_numpy().reshape(-1, 1)
# Test
X_test_standardized = X_test_standardized.to_numpy()
X_test_scaled = X_test_scaled.to_numpy()
y_test = y_test.to_numpy().reshape(-1, 1)

# %% [markdown]
# ## Hyperparameter Tuning and Model Selection

# %%
# Things to tune:
# distance metric
# standardize vs scale: STANDARDIZE
# p in minkowsky distance
# k in k-nearest neighbors: k = sqrt(n)
# weighted vs standard k-nearest neighbors algorithm: WEIGHTED

# Tune minkowski distance
weights = get_weights(y_train)
p_values = np.arange(1, 11, 1)
k = get_k_value(X_train_standardized)
k_values = [k]
best_p_minkowski, best_k_minkowski = tune_minkowski(X_train_standardized,
y_train, weights, 10, p_values, k_values, weighted=True)
# Tune cosine distance
#k_values = np.arange(1, 301, 24)
#best_k_cosine = tune_other_distance(X_train_standardized, y_train, weights,
10, k_values, get_cosine_distance, weighted=True)
# Tune chebyshev distance
#best_k_chebyshev = tune_other_distance(X_train_standardized, y_train,
weights, 10, k_values, get_chebyshev_distance, weighted=True)

# Evaluate minkowski distance model
minkowski_score = cross_validation(X_train_standardized, y_train, k, weights,
get_minkowski_distance, 10, weighted=True, p_minkowski=best_p_minkowski)
```

```
# Evaluate cosine distance model
cosine_score = cross_validation(X_train_standardized, y_train, k, weights,
get_cosine_distance, 10, weighted=True)
# Evaluate chebyshev distance model
chebyshev_score = cross_validation(X_train_standardized, y_train, k, weights,
get_chebyshev_distance, 10, weighted=True)
# Print the results
print(f"Minkowski distance F1 score: {minkowski_score}\n"
      f"Cosine distance F1 score: {cosine_score}\n"
      f"Chebyshev distance F1 score: {chebyshev_score}")

# %% [markdown]
# ## Final Model and Results

# %%
# Get weights and k value
weights = get_weights(y_test)
k = get_k_value(X_test_standardized)
# Make predictions
trained_predictions = predict_from_same_set(X_test_standardized, y_test, k,
weights, get_minkowski_distance, weighted=True, p_minkowski=best_p_minkowski)
# Evaluate results
trained_accuracy = get_accuracy(y_test, trained_predictions)
trained_confusion_matrix = get_confusion_matrix(y_test, trained_predictions)
trained_f1_score = get_f1_score(y_test, trained_predictions)
TN, FP, FN, TP = get_confusion_matrix(y_test, trained_predictions).ravel()
print(
      f"Accuracy: {trained_accuracy}\n"
      f"F1-score: {trained_f1_score}\n"
      f"TN, FP, TP, FN: {TN}, {FP}, {TP}, {FN}\n"
)
```

Feedforward Neural Network

```
# %% [markdown]
# Alkım Ege Akarsu | 21901461 | EEE 485 | Term Project
# # Classification of Diabetic and Non-Diabetic Individuals Based on Survey
Data
# # Feedforward Neural Network
# ## Package Imports

# %%
import numpy as np  # For math operations
import pandas as pd  # For importing and handling datasets
import matplotlib.pyplot as plt  # For plotting
from mpl_toolkits.mplot3d import axes3d  # For 3D component of plotting

# Pandas options
pd.set_option("display.max_columns", None)
```

```
# NumPy options
np.set_printoptions(precision=3)
np.set_printoptions(suppress=True)

# %% [markdown]
# ## Functions

# %%
# DATASET PREPARATION
def train_test_split(dataset, train_fraction):
      """Split the dataset into train and test sets

      Args:
      dataset (DataFrame): All available data points.
      Label and features as columns, Datapoints as rows.
      train_fraction (float): Fraction of data
            points to be added to the training set.

      Returns:
      DataFrame: Training dataset
      DataFrame: Test dataset
      """
      # Randomly sample for train
      train = dataset.sample(frac=train_fraction, axis="index")
      # Subtract train from original
      test = dataset.drop(index=train.index)
      # Reset indexes
      train = train.reset_index(drop=True)
      test = test.reset_index(drop=True)

      return train, test


def mini_dataset(dataset, fraction):
      """Return a fraction of the dataset.

      Args:
      dataset (DataFrame): Original dataset. Has shape (n, p).
      fraction (double): Fraction of dataset to return.

      Returns:
      DataFrame: Smaller dataset. Has shape (n / fraction, p).
      """
      return dataset.sample(frac=fraction, axis="index", ignore_index=True)


def standardize(train, test):
      """Standardize X matrices of train and test splits. Uses the mean and
```

```
      standard deviation of the training set to standardize both training
      and test sets. This prevents data leakage between training and test
sets.

      Args:
      train (DataFrame): Training set.
      test (DataFrame): Test set.

      Returns:
      DataFrame: Standardized training set
      DataFrame: Standardized test set
      """
      # Get mean and standard deviation
      mean = train.mean()
      std = train.std()
      # Get results
      train_result = (train - mean) / std
      test_result = (test - mean) / std

      return train_result, test_result


def split_validation(X_train, y_train, valid_fraction):
      """Split a valdation set from a training set.

      Args:
      X_train (ndarray): Feature matrix. Has shape (n, p).
      y_train (ndarray): Label vector. Has shape (n, 1).
      valid_fraction (double): Fraction of data points to be added to the
            validation set.

      Returns:
      ndarray: Training features
      ndarray: Training labels
      ndarray: Validation features
      ndarray: Validation labels
      """
      # Calculate the number of validation samples
      num_validation_samples = int(X_train.shape[0] * valid_fraction)
      # Generate a random permutation of the indices
      indices = np.random.permutation(X_train.shape[0])
      # Split the indices into training and validation sets
      training_i, validation_i = (
      indices[num_validation_samples:],
      indices[:num_validation_samples],
      )
      # Use the indices to create the training and validation sets
      X_train, X_valid = X_train[training_i, :], X_train[validation_i, :]
```

```
      y_train, y_valid = y_train[training_i, :], y_train[validation_i, :]

      return X_train, y_train, X_valid, y_valid


# FEEDFORWARD NEURAL NETWORK
def initialize(layer_dimensions):
      """Initialize the weights and biases of the neural network.

      Args:
      layer_dimensions (list of ints): Contains the dimension of every layer.
            Ex: [4, 7, 4] Represents a network with 4 input, 7 hidden_2
            and 4 output layer neurons.

      Returns:
      dictionary of ndarrays: Contains the weights and biases of the neural
network.
            Formatted as W1, b1, W2, b2 etc.
            Wl has shape (layer_dimensions[l], layer_dimensions[l - 1])
            bl has shape (layer_dimensions[l], 1)
      """
      # Dict containing weights and biases
      parameters = {}
      # Number of layers
      L = len(layer_dimensions)
      # Loop over each layer
      for l in range(1, L):
      # Initialize W and b
      # He weight initialization best for ReLu
      parameters["W" + str(l)] = np.random.normal(
            0,
            np.sqrt(2 / layer_dimensions[l - 1]),
            (layer_dimensions[l], layer_dimensions[l - 1]),
      )
      # Zero initialization for biases
      parameters["b" + str(l)] = np.zeros((layer_dimensions[l], 1))

      return parameters


def get_induced_local_field(A_previous, W, b):
      """Compute the induced local field.

      Args:
      A_previous (ndarray): Outputs of the previous layer or inputs.
            Has shape (previous layer, number of data points).
      W (ndarray): Weight matrix. Has shape: (current layer, previous layer)
      b (ndarray): Bias vector. Has shape: (current layer, 1)
```

```
        Returns:
        ndarray: Induced local field.
              Has shape (current layer, number of data points).
        tuple of ndarrays: Storage to be used in backward propagation
        """
        # Calculate induced local field.
        V = np.dot(W, A_previous) + b
        # Store results for faster backward propagation
        induced_local_field_storage = (A_previous, W, b)

        return V, induced_local_field_storage


def relu(V):
        """Calculate the output of the ReLu activation function given the
induced
        local field.

        Args:
        V (ndarray): Induced local field.
              Has shape (current layer, number of data points).

        Returns:
        ndarray: Output of ReLu. Has shape (current layer, number of data
points).
        ndarray: Storage to be used in backward propagation. Contains V.
        """
        # Get activation results of current layer
        A = np.maximum(0, V)
        # Store induced local field
        storage = V

        return A, storage


def sigmoid(V):
        """Calculate the output of the sigmoid activation function given the
induced
        local field.

        Args:
        V (ndarray): Induced local field.
              Has shape (current layer, number of data points).

        Returns:
        ndarray: Output of sigmoid. Has shape (current layer, number of data
points).
```

```
        ndarray: Storage to be used in backward propagation. Contains V.
        """
        # Get activation results of current layer
        A = 1 / (1 + np.exp(-V))
        # Store induced local field
        storage = V

        return A, storage


def relu_backward(dA, storage):
        """Backward propagation version of ReLu.

        Args:
        dA (ndarray): Gradient of activation.
        storage (ndarray): Only contains the induced local field.

        Returns:
        ndarray: Gradient of cost with respect to induced local field.
        """
        # Get induced local field from storage
        V = storage
        # Convert dA to dV
        dV = np.array(dA, copy=True)

        # Set dV to 0 when V <= 0
        dV[V <= 0] = 0

        return dV


def sigmoid_backward(dA, storage):
        """Backward propagation version of sigmoid.

        Args:
        dA (ndarray): Gradient of activation.
        storage (ndarray): Only contains the induced local field.

        Returns:
        ndarray: Gradient of cost with respect to induced local field.
        """
        # Get induced local field from storage
        V = storage

        # Compute gradient of cost with respect to induced local field
        s = 1 / (1 + np.exp(-V))
        dV = dA * s * (1 - s)
```

```
    return dV


def forward_one_layer(A_previous, W, b, activation):
    """Forward propagate for one layer.

    Args:
    A_previous (ndarray): Outputs of the previous layer or inputs.
        Has shape (previous layer, number of data points).
    W (ndarray): Weight matrix. Has shape: (current layer, previous layer)
    b (ndarray): Bias vector. Has shape: (current layer, 1)
    activation (function): Activation function to be used.
        ReLu for hidden layers, sigmoid for output layer.

    Returns:
    ndarray: Output of the activation function.
        Has shape (current layer, number of data points).
    tuple of ndarrays: Storage to be used in backward propagation.
    """
    V, induced_local_field_storage = get_induced_local_field(A_previous, W,
b)
    A, activation_storage = activation(V)

    storage = (induced_local_field_storage, activation_storage)

    return A, storage


def forward_propagation(X, parameters):
    """Complete network forward propagation.

    Args:
    X (ndarray): Feature values for each data point. Has shape (p, n)
    parameters (dictionary or ndarrays): Output of initialize_parameters.
        Contains weights and biases.

    Returns:
    ndarray: Output of the activation function of the output layer.
        Has shape (output layer, number of data points).
    list of tuples: Storages to be used in backward propagation.
        Has length one fewer than the number of layers.
    """
    # For backpropagation
    storages = []
    # Inputs
    A = X
    # Get the number of layers
    L = len(parameters) // 2
```

```
        # Forward propagation for hidden layers
        for l in range(1, L):
        A_previous = A
        A, storage = forward_one_layer(
              A_previous, parameters["W" + str(l)], parameters["b" + str(l)],
relu
        )
        storages.append(storage)

        # Forward propagation for output layer
        A_last, storage = forward_one_layer(
        A, parameters["W" + str(L)], parameters["b" + str(L)], sigmoid
        )
        storages.append(storage)

        return A_last, storages


def get_cost(A_last, y_true):
        """Compute cross-entropy cost.

        Args:
        A_last (ndarray): Prediction probabilities. Has shape (1, n).
        y_true (ndarray): Correct label vector. Has shape (1, n)

        Returns:
        double: Cross-entropy cost
        """
        # Get number of data points
        m = y_true.shape[1]

        # Compute cost
        cost = (-1 / m) * (
        np.dot(y_true, np.log(A_last).T) + np.dot((1 - y_true), np.log(1 -
A_last).T)
        )
        cost = np.squeeze(cost)

        return cost


def get_backward_linear(dV, storage):
        """Compute the linear part of the backward propagation.

        Args:
        dV (ndarray): Derivative of induced local field.
        storage (tuple of ndarrays): "induced_local_field_storage" coming from
```

```
            the function "forward_one_layer".

      Returns:
      ndarray: Gradient of the cost with respect to activation.
            Has shape(previous layer, number of data points).
      ndarray: Gradient of the cost with respect to weights. Has same shape
as W.
      ndarray: Gradient of the cost with respect to biases. Has same shape as
b.
      """
      # Prepare values from storage for derivatives
      A_previous, W, _ = storage
      # Get the number of data points
      m = A_previous.shape[1]

      # Compute necessary derivatives
      dW = (1 / m) * np.dot(dV, A_previous.T)
      db = (1 / m) * np.sum(dV, axis=1, keepdims=True)
      dA_previous = np.dot(W.T, dV)

      return dA_previous, dW, db


def backward_one_layer(dA, storage, activation_backward):
      """Compute one layer of backward propagation.

      Args:
      dA (ndarray): Gradient of activation of current layer.
      storage (tuple of ndarrays): "induced_local_field_storage"
            and "activation_storage".
      activation_backward (function): Function corresponding to this layer.

      Returns:
      ndarray: Gradient of the cost with respect to activation.
            Has shape(previous layer, number of data points).
      ndarray: Gradient of the cost with respect to weights. Has same shape
as W.
      ndarray: Gradient of the cost with respect to biases. Has same shape as
b.
      """
      # Prepare values from storage for derivatives
      induced_local_field_storage, activation_storage = storage
      # Compute the gradient of the cost with respect to activation
      dV = activation_backward(dA, activation_storage)
      # Compute other gradients
      dA_previous, dW, db = get_backward_linear(dV,
induced_local_field_storage)
```

```
    return dA_previous, dW, db


def backward_propagation(A_last, y_true, storages):
    """Compute backward propagation for the whole network.

    Args:
    A_last (ndarray): Output of "forward_propagation". Probability vector.
        Has shape (output layer, number of data points).
    y_true (ndarray): Correct label vector. Has shape (1, n)
    storages (list of ndarrays): Storages of "get_induced_local_field" of
        relu and sigmoid layers.

    Returns:
    dictionary of ndarrays: Gradients of activations, weights and biases.
        for every layer. Formatted as dA1, dW1, db1, dA2, dW2, db2 etc.
    """
    # Initialize necessary variables
    gradients = {}
    L = len(storages)  # Number of layers
    m = A_last.shape[1]  # Number of data points
    y_true = y_true.reshape(A_last.shape)  # y_true same shape as A_last

    # Initialization of backward propagation
    dA_last = -(np.divide(y_true, A_last) - np.divide(1 - y_true, 1 -
A_last))

    # Process last layer
    current_storage = storages[L - 1]
    (
    gradients["dA" + str(L - 1)],
    gradients["dW" + str(L)],
    gradients["db" + str(L)],
    ) = backward_one_layer(dA_last, current_storage, sigmoid_backward)

    # Process all other layers
    for l in reversed(range(L - 1)):
    current_storage = storages[l]
    dA_previous_temp, dW_temp, db_temp = backward_one_layer(
        gradients["dA" + str(l + 1)], current_storage, relu_backward
    )
    gradients["dA" + str(l)] = dA_previous_temp
    gradients["dW" + str(l + 1)] = dW_temp
    gradients["db" + str(l + 1)] = db_temp

    return gradients
```

```
def update_parameters(parameters, gradients, learning_rate):
      """Update weights and biases based on gradient descent.

      Args:
      parameters (dictionary of ndarrays): Contains weights and biases.
      gradients (dictionary of ndarrays): Contains gradients.
            Output of "backward_propagation".
      learning_rate (double): Learning rate parameter of gradient descent.

      Returns:
      dictionary of ndarrays: Updated weights and biases.
      """
      # Get number of layers in the network
      L = len(parameters) // 2

      # Loop over each layer and update weights and biases using gradient
descent
      for l in range(L):
      parameters["W" + str(l + 1)] = (
            parameters["W" + str(l + 1)] - learning_rate * gradients["dW" +
str(l + 1)]
      )
      parameters["b" + str(l + 1)] = (
            parameters["b" + str(l + 1)] - learning_rate * gradients["db" +
str(l + 1)]
      )

      return parameters


def train(
      X_train,
      y_train,
      X_valid,
      y_valid,
      layer_dimensions,
      learning_rate,
      max_iterations,
      min_cost_delta,
      validate,
      print_cost,
      plot,
):
      """Create and train a feedforward neural network.

      Args:
      X_train (ndarray): Feature values for each data point. Has shape (p,
n).
```

```
      y_train (ndarray): Labels of each data point. Has shape (1, n)
      X_valid (ndarray): Feature values for each data point. Has shape (p,
n).
      y_valid (ndarray): Labels of each data point. Has shape (1, n)
      layer_dimensions (list of ints): Contains the dimension of every layer.
            Ex: [4, 7, 4] Represents a network with 4 input, 7 hidden_2
            and 4 output layer neurons.
      learning_rate (double): Gradient descent learning rate
      max_iterations (int): Maximum number of training iterations.
      min_cost_delta (double): If cost delta is smaller than this value, stop
            training.
      validate (bool): Simultaneously validate the model.
      print_cost (bool): Print current training cost occasionally.
      plot (bool): Plot loss vs epochs at the end of training.

      Returns:
      dictionary of ndarrays: Trained parameters.
      """
      # List to keep track of costs
      costs_train = np.zeros(max_iterations)
      if validate and plot:
      costs_valid = np.zeros(max_iterations)

      # Intialize parameters
      parameters = initialize(layer_dimensions)

      # Loop for predetermined number of max iterations
      for i in range(max_iterations):
      # Forward propagation for training
      A_last_train, storages = forward_propagation(X_train, parameters)
      # Forward propagation for validation
      if validate and plot:
            A_last_valid, _ = forward_propagation(X_valid, parameters)

      # Compute cost for training
      cost_train = get_cost(A_last_train, y_train)
      # Compute cost for validation
      if validate and plot:
            cost_valid = get_cost(A_last_valid, y_valid)

      # Backward propagation
      gradients = backward_propagation(A_last_train, y_train, storages)

      # Update parameters
      parameters = update_parameters(parameters, gradients, learning_rate)

      # Save the cost
      costs_train[i] = cost_train
```

```
	if validate and plot:
		costs_valid[i] = cost_valid

	# Calculate cost delta and break if necessary
	if i > 0:
		cost_delta = abs(costs_train[i - 1] - costs_train[i])
		if cost_delta < min_cost_delta:
			break

	# Occasionally print the cost
	if print_cost and i % 5 == 0:
		print(f"Cost after iteration {i}: {cost_train}")

	# Plot cost over epochs
	if plot:
	# Trim the 0's at the end of costs ndarrays
	costs_train = np.trim_zeros(costs_train, "b")
	if validate:
		costs_valid = np.trim_zeros(costs_valid, "b")
	plt.plot(np.squeeze(costs_train), label="Training set")
	if validate:
		plt.plot(np.squeeze(costs_valid), ".", label="Validation set")
	plt.title("Cross-Entropy Loss vs Epochs")
	plt.xlabel("Epochs")
	plt.ylabel("Cross-Entropy Loss")
	if validate:
		plt.legend()
	plt.grid(True)
	plt.show()

	return parameters


# MODEL EVALUATION
def predict(X, parameters, threshold):
	"""Get predictions from a trained model.

	Args:
	X (ndarray): Feature values for each data point. Has shape (p, n)
	parameters (dictionary of ndarrays): Trained weights and biases.

	Returns:
	ndarray: Predicted labels of all data points. Has shape (1, n)
	"""
	# Forward propagation
	probabilities, _ = forward_propagation(X, parameters)
	# Convert probabilities to predictions
	predictions = probabilities >= threshold
```

```
    predictions = predictions.astype(np.uint8)

    return predictions


def get_confusion_matrix(true, pred):
    """Calculate confusion matrix for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    ndarray: Confusion matrix
    TN FP
    FN TP
    """
    result = np.zeros((2, 2))

    for i in range(true.shape[0]):
    result[true[i][0]][pred[i][0]] += 1

    return result


def get_accuracy(true, pred):
    """Calculate accuracy for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    double: Accuracy value.
    """
    TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
    return (TP + TN) / (TP + TN + FP + FN)


def get_precision(true, pred):
    """Calculate precision for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    double: Precision value.
```

```
    """
    TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
    return TP / (TP + FP)


def get_recall(true, pred):
    """Calculate recall for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    double: Recall value.
    """
    TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
    return TP / (TP + FN)


def get_f1_score(true, pred):
    """Calculate F1 score for performance evaluation.

    Args:
    true (ndarray): Array of true labels. Has shape (t, 1).
    pred (ndarray): Array of predicted labels. Has shape (t, 1).

    Returns:
    double: F1 score.
    """
    TN, FP, FN, TP = get_confusion_matrix(true, pred).ravel()
    return (2 * TP) / ((2 * TP) + FP + FN)


# HYPERPARAMETER TUNING
def validate(
    X_train,
    y_train,
    X_valid,
    y_valid,
    layer_dimensions,
    learning_rate,
    max_iterations,
    min_cost_delta,
    threshold,
):
    """Evaluate model performance on hold out validation set.

    Args:
```

```
      X_train (ndarray): Feature values for each data point. Has shape (p,
n).
      y_train (ndarray): Labels of each data point. Has shape (1, n)
      X_valid (ndarray): Feature values for each data point. Has shape (p,
n).
      y_valid (ndarray): Labels of each data point. Has shape (1, n)
      layer_dimensions (list of ints): Contains the dimension of every layer.
            Ex: [4, 7, 4] Represents a network with 4 input, 7 hidden_2
            and 4 output layer neurons.
      learning_rate (double): Gradient descent learning rate
      max_iterations (int): Maximum number of training iterations.
      min_cost_delta (double): If cost delta is smaller than this value, stop
            training.
      threshold (double): Threshold determining class of data point.

      Returns:
      double: F1 score of validation set.
      """
      # Train model on training set
      parameters = train(
      X_train,
      y_train,
      X_valid,
      y_valid,
      layer_dimensions,
      learning_rate,
      max_iterations,
      min_cost_delta,
      validate=False,
      print_cost=True,
      plot=False,
      )
      # Validate model on validation set
      predictions = predict(X_valid, parameters, threshold)
      # Return F1 score
      return get_f1_score(y_valid.T, predictions.T)


def tune_threshold(
      X_train,
      y_train,
      X_valid,
      y_valid,
      layer_dimensions,
      learning_rate,
      max_iterations,
      min_cost_delta,
      threshold_values,
```

```
):
      """Threshold tuning.

      Args:
      X_train (ndarray): Feature values for each data point. Has shape (p,
n).
      y_train (ndarray): Labels of each data point. Has shape (1, n)
      X_valid (ndarray): Feature values for each data point. Has shape (p,
n).
      y_valid (ndarray): Labels of each data point. Has shape (1, n)
      layer_dimensions (list of ints): Contains the dimension of every layer.
            Ex: [4, 7, 4] Represents a network with 4 input, 7 hidden_2
            and 4 output layer neurons.
      learning_rate (double): Gradient descent learning rate
      max_iterations (int): Maximum number of training iterations.
      min_cost_delta (double): If cost delta is smaller than this value, stop
            training.
      threshold_values (ndarray): Threshold values to try.

      Returns:
      double: Best threshold.
      """
      # Initialize results matrix
      # 2 columns: threshold, score
      results = np.empty((0, 2))

      # Perform grid search
      # Loop over threshold_values
      for threshold in threshold_values:
      # Perform validation
      f1_score = validate(
            X_train,
            y_train,
            X_valid,
            y_valid,
            layer_dimensions,
            learning_rate,
            max_iterations,
            min_cost_delta,
            threshold,
      )
      # Append threshold, f1_score row to results matrix
      row = np.array([threshold, f1_score])
      results = np.vstack((results, row))

      threshold_results = results[:, 0].flatten()
      f1_score_results = results[:, 1].flatten()
      # Plot scores against threshold values
```

```
      plt.plot(threshold_results, f1_score_results, ".")
      plt.title("F1 Score vs Decision Threshold")
      plt.xlabel("Threshold")
      plt.ylabel("F1 Score")
      plt.grid(True)
      plt.show()

      # Get best_threshold
      # Row index of max score
      row_index = np.argmax(results[:, 1])
      best_row = results[row_index]
      best_threshold = best_row[0]

      return best_threshold


def tune_neuron_count(
      X_train,
      y_train,
      X_valid,
      y_valid,
      learning_rate,
      max_iterations,
      min_cost_delta,
      threshold,
      neuron_count_values,
):
      """Neuron count tuning.

      Args:
      X_train (ndarray): Feature values for each data point. Has shape (p,
n).
      y_train (ndarray): Labels of each data point. Has shape (1, n)
      X_valid (ndarray): Feature values for each data point. Has shape (p,
n).
      y_valid (ndarray): Labels of each data point. Has shape (1, n)
      learning_rate (double): Gradient descent learning rate
      max_iterations (int): Maximum number of training iterations.
      min_cost_delta (double): If cost delta is smaller than this value, stop
            training.
      threshold (ndarray): Threshold value.
      neuron_count_values (ndarray): Neuron counts to try.

      Returns:
      double: Best neuron count.
      """
      # Initialize results matrix
      # 2 columns: neuron_count, score
```

```
    results = np.empty((0, 2))

    # Perform grid search
    # Loop over neuron_counts
    for neuron_count in neuron_count_values:
    layer_dimensions = [X_train.shape[0], neuron_count, 1]
    # Perform validation
    f1_score = validate(
        X_train,
        y_train,
        X_valid,
        y_valid,
        layer_dimensions,
        learning_rate,
        max_iterations,
        min_cost_delta,
        threshold,
    )
    # Append neuron_count, f1_score row to results matrix
    row = np.array([neuron_count, f1_score])
    results = np.vstack((results, row))

    neuron_count_results = results[:, 0].flatten()
    f1_score_results = results[:, 1].flatten()
    # Plot scores against neuron_count values
    plt.plot(neuron_count_results, f1_score_results, ".")
    plt.title("F1 Score vs Neuron Count")
    plt.xlabel("Neuron Count")
    plt.ylabel("F1 Score")
    plt.grid(True)
    plt.show()

    # Get best_neuron_count
    # Row index of max score
    row_index = np.argmax(results[:, 1])
    best_row = results[row_index]
    best_neuron_count = best_row[0]

    return best_neuron_count


def tune_layer_count(
    X_train,
    y_train,
    X_valid,
    y_valid,
    learning_rate,
    max_iterations,
```

```
      min_cost_delta,
      threshold,
      neuron_count,
      layer_count_values,
):
      """Layer count tuning.

      Args:
      X_train (ndarray): Feature values for each data point. Has shape (p,
n).
      y_train (ndarray): Labels of each data point. Has shape (1, n)
      X_valid (ndarray): Feature values for each data point. Has shape (p,
n).
      y_valid (ndarray): Labels of each data point. Has shape (1, n)
      learning_rate (double): Gradient descent learning rate
      max_iterations (int): Maximum number of training iterations.
      min_cost_delta (double): If cost delta is smaller than this value, stop
            training.
      threshold (ndarray): Threshold value.
      layer_count_values (ndarray): Layer counts to try.

      Returns:
      double: Best layer count.
      """
      # Initialize results matrix
      # 2 columns: layer_count, score
      results = np.empty((0, 2))

      # Perform grid search
      # Loop over layer_counts
      for layer_count in layer_count_values:
      hidden_layers = [neuron_count] * layer_count
      layer_dimensions = [X_train.shape[0]] + hidden_layers + [1]
      # Perform validation
      f1_score = validate(
            X_train,
            y_train,
            X_valid,
            y_valid,
            layer_dimensions,
            learning_rate,
            max_iterations,
            min_cost_delta,
            threshold,
      )
      # Append layer_count, f1_score row to results matrix
      row = np.array([layer_count, f1_score])
      results = np.vstack((results, row))
```

```
        layer_count_results = results[:, 0].flatten()
        f1_score_results = results[:, 1].flatten()
        # Plot scores against layer_count values
        plt.plot(layer_count_results, f1_score_results, ".")
        plt.title("F1 Score vs Layer Count")
        plt.xlabel("Layer Count")
        plt.ylabel("F1 Score")
        plt.grid(True)
        plt.show()

        # Get best_layer_count
        # Row index of max score
        row_index = np.argmax(results[:, 1])
        best_row = results[row_index]
        best_layer_count = best_row[0]

        return best_layer_count

# %% [markdown]
# ## Dataset Preparation

# %%
# Import dataset into a pandas dataframe
# Data points classified as "diabetes or pre-diabetes"(1) or "no diabetes"(0)
dataset = pd.read_csv("datasets/binary.csv", dtype="uint8")
# Dataset overview
dataset.info(verbose=True, show_counts=True)

# Divide dataset into training and test sets
train_fraction = 0.8
training, test = train_test_split(dataset, train_fraction)
# Number of data points
print(f"\n\nNumber of data points in train set: {len(training.index)}")
print(f"Number of data points in test set: {len(test.index)}")

# Get X and Y from train and test datasets
# Train
y_train = training["Diabetes"]
X_train = training.drop("Diabetes", axis=1)
# Test
y_test = test["Diabetes"]
X_test = test.drop("Diabetes", axis=1)

# Standardize X
X_train_standardized, X_test_standardized = standardize(X_train, X_test)

# Convert everything from dataframe to ndarray
```

```
# Training
X_train_standardized = X_train_standardized.to_numpy()
y_train = y_train.to_numpy().reshape(-1, 1)
# Test
X_test_standardized = X_test_standardized.to_numpy()
y_test = y_test.to_numpy().reshape(-1, 1)

# Split validation set
X_train_standardized, y_train, X_valid_standardized, y_valid =
split_validation(
      X_train_standardized, y_train, 0.2
)

# Transpose everything for compatability
# Training
X_train_standardized = np.transpose(X_train_standardized)
y_train = np.transpose(y_train)
print(X_train_standardized.shape)
print(y_train.shape)
# Validation
X_valid_standardized = np.transpose(X_valid_standardized)
y_valid = np.transpose(y_valid)
print(X_valid_standardized.shape)
print(y_valid.shape)
# Test
X_test_standardized = np.transpose(X_test_standardized)
y_test = np.transpose(y_test)
print(X_test_standardized.shape)
print(y_test.shape)

# Get correlation matrix between each feature
corr_coef = np.corrcoef(X_train_standardized)
corr_coef = np.vstack((np.zeros((1, 21)), corr_coef))
corr_coef = np.hstack((np.zeros((22, 1)), corr_coef))
plt.matshow(corr_coef)
plt.title("Correlation Between Features")
plt.xlabel("Features")
plt.ylabel("Features")
ax = plt.gca()
ax.set_xlim([0.5, 21.5])
ax.set_ylim([21.5, 0.5])
plt.show()

# Get correlation vector between every feature and label
correlations = []
for i in range(X_train_standardized.shape[0]):
      corr_matrix = np.corrcoef(X_train_standardized[i,:], y_train[0,:])
      # The correlation coefficient we're interested in is the off-diagonal
```

```
element
      correlations.append(corr_matrix[0, 1])
plt.stem(range(1, 22), correlations)
plt.title("Correlation Between Each Feature and the Label Vector")
plt.xlabel("Features")
plt.xticks(range(1,22))
plt.ylabel("Pearson Product-Moment Correlation Coefficients")
plt.grid(True)
plt.show()

# %% [markdown]
# ## Hyperparameter Tuning

# %%
best_threshold = tune_threshold(
      X_train_standardized,
      y_train,
      X_valid_standardized,
      y_valid,
      layer_dimensions=[X_train_standardized.shape[0], 21, 1],
      learning_rate=1,
      max_iterations=1000,
      min_cost_delta=0.0001,
      threshold_values=np.arange(0.1, 1.0, 0.1),  # 0.1, 0.2, ..., 0.8, 0.9
)
print(f"Best threshold: {best_threshold}")
best_neuron_count = tune_neuron_count(
      X_train_standardized,
      y_train,
      X_valid_standardized,
      y_valid,
      learning_rate=0.1,
      max_iterations=1000,
      min_cost_delta=0.0001,
      threshold=best_threshold,
      neuron_count_values=np.arange(1, 102, 10),
)
print(f"Best neuron count: {best_neuron_count}")
best_layer_count = tune_layer_count(
      X_train_standardized,
      y_train,
      X_valid_standardized,
      y_valid,
      learning_rate=0.1,
      max_iterations=1000,
      min_cost_delta=0.0001,
      threshold=best_threshold,
      neuron_count=10,
```

```
      layer_count_values=np.arange(1, 11, 1),
)
print(f"Best layer count: {best_layer_count}")


# %% [markdown]
# ## Final Model and Results

# %%
# Stack validation and training sets
X_train_standardized = np.hstack((X_train_standardized,
X_valid_standardized))
y_train = np.hstack((y_train, y_valid))


parameters = train(
      X_train_standardized,
      y_train,
      X_test_standardized,
      y_test,
      layer_dimensions=[X_train_standardized.shape[0], 21, 1],
      learning_rate=1,
      max_iterations=1000,
      min_cost_delta=0.0001,
      validate=True,
      print_cost=True,
      plot=True,
)

predictions = predict(X_train_standardized, parameters, threshold=0.2)
trained_accuracy = get_accuracy(y_train.T, predictions.T)
trained_confusion_matrix = get_confusion_matrix(y_train.T, predictions.T)
trained_f1_score = get_f1_score(y_train.T, predictions.T)
TN, FP, FN, TP = get_confusion_matrix(y_train.T, predictions.T).ravel()
print(
      f"Accuracy: {trained_accuracy}\n"
      f"F1-score: {trained_f1_score}\n"
      f"TN, FP, TP, FN: {TN}, {FP}, {TP}, {FN}\n"
)

predictions = predict(X_test_standardized, parameters, threshold=0.2)
trained_accuracy = get_accuracy(y_test.T, predictions.T)
trained_confusion_matrix = get_confusion_matrix(y_test.T, predictions.T)
trained_f1_score = get_f1_score(y_test.T, predictions.T)
TN, FP, FN, TP = get_confusion_matrix(y_test.T, predictions.T).ravel()
print(
      f"Accuracy: {trained_accuracy}\n"
      f"F1-score: {trained_f1_score}\n"
```

```
        f"TN, FP, TP, FN: {TN}, {FP}, {TP}, {FN}\n"
)
```