Chiao-Hsi Joshua Wang

Food Classification through Machine Learning Algorithms

Introduction

This report aims to explore and analyse food nutrient data obtained from the Food Standards Australia and New Zealand at *https://www.foodstandards.gov.au/science/monitoringnutrients/afcd/Pages/default.aspx* by making use of a variety of machine learning techniques, models and algorithms. The code used throughout this report has been completed using Python 3.10.9, and requires the following dependencies:

- Pandas
- Numpy
- Scikit-learn
- Matplotlib

This report will use look into three main machine learning techniques – K-Nearest Neighbours, Logistic Regression, and Neural Networks. The problem we wish to tackle using these techniques is to classify foods as either solids or liquids, based off information about the nutrients that exist in each food.

Dataset Pre-processing

The dataset contains information about the nutrient composition in a wide range of foods in an Excel spreadsheet file, with all the data split into two sheets – "All solids & liquids per 100g" and "Liquids only per 100mL". There were 1616 listed items, each represented by 293 attributes, in the "All solids & liquids per 100g" category, and there were 189 items under the "Liquids only per 100mL" category, each having 204 attributes. The attributes for each item included aspects such as its name, energy, protein and moisture. This dataset was loaded by using Python and the Pandas library:

```
PATH = '/Users/joshuawang/Documents/University/Courses/COMP4702/Assignment/'
all_solids_liquids = pd.read_excel(PATH + 'Rel_2_Nutrient_file.xlsx', sheet_name='All solids &
liquids per 100g')
all_liquids = pd.read_excel(PATH + 'Rel_2_Nutrient_file.xlsx', sheet_name='Liquids only per
100mL')
```

The first step in pre-processing this data was to address the many missing values throughout the dataset. For many of the foods listed in the dataset, information on nutrients such as glucose, glycogen and citric acid had no values, while only a few entries have actual values or 0 values recorded. Since the dataset had many columns already, it was decided that if all columns with NaN values were removed, there would still be enough relevant data to analyse. Thus, we obtain a new dataset with 62 columns rather than the original 204 with the following code:

```
for col in all_solids_liquids:
    if all_solids_liquids[col].isna().sum() != 0:
        all_solids_liquids = all_solids_liquids.drop(col, axis=1)
```

The next step in pre-processing the dataset was categorising the data according to either a solid or liquid, since the original data had two Excel sheets split – one for solids & liquids, and another for only liquids. We achieved this by appending on an additional attribute named "Category" and for each of the records in the "All solids & liquids per 100g" Excel sheet, if its "Public Food Key" was present in the "Liquids only per 100mL" sheet, then it was classified as a liquid. To make the binary classification analysis process simpler for later, the "Category" attribute was created to be a binary digit – either a 0 or 1, for solids and liquids, respectively.

```
categories = []
for key in all_solids_liquids["Public Food Key"]:
    if key in all_liquids["Public Food Key"].to_numpy():
        categories.append(1)
    else:
        categories.append(0)
all_solids_liquids["Category"] = categories
```

The final step in pre-processing our data is to normalize our values. In the remaining attributes of our dataset, we find that the scales can vary greatly, with attributes such as moisture ranging from 0 to 100, whilst sodium (in milligrams) ranged from 0 to 38178. Since the scale of features can affect the performance of machine learning techniques, a method to solve this problem is through the use of either normalization or standardization. In this case, since we cannot suppose that the distribution of each of our features follow a Gaussian distribution, it was decided that normalization would be performed on all features, apart from those that contained string or binary digit values. By taking away the minimum value for each column from the original values, then dividing by the difference between the minimum and maximum values of that attribute, we are able have all features on the same scale so that the minimum value is 0 and the maximum is 1.

However, before performing normalization, we will first need to split our dataset into training and testing sets, since if we were to normalize the whole dataset at once, information from our testing set would be leaked into the training set. It was decided that 80% of the data would be used for training and the remaining 20% would be used for testing. Our report will also investigate the effects that hyperparameters have on our results, and so it was decided that k-fold cross validation would be required. Since we have a small dataset, 4-fold cross validation would be used, and on top of this, since we have 1616 total food items but out of these only 189 are liquids, stratified sampling was used to ensure that we have a proportionate number of liquids in each of our samples, which would help us make sure our results would be less biased towards the solid foods. So, to split our datasets and normalize the data, the following code was used:

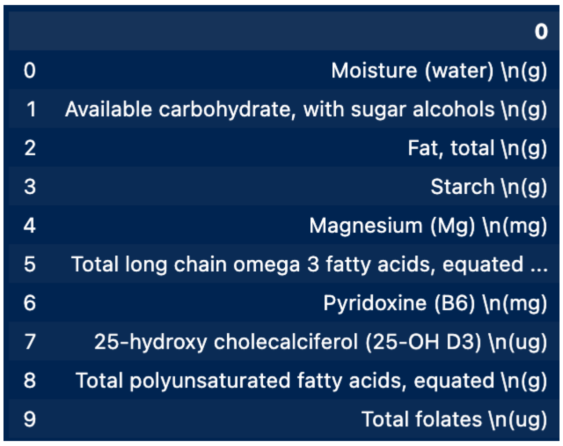
```
# Get training and test sets - We use 4-fold cross-validation, with stratified sampling to fix
class imbalance
train_set, test_set = train_test_split(all_solids_liquids, train_size=0.8, shuffle=True,
random_state=18, stratify=all_solids_liquids["Category"])

# Get each of the training folds
train_1, train_2 = train_test_split(train_set, train_size=0.5, shuffle=True,
stratify=train_set["Category"])
train_1, train_3 = train_test_split(train_1, train_size=0.5, shuffle=True,
stratify=train_1["Category"])
train_2, train_4 = train_test_split(train_2, train_size=0.5, shuffle=True,
stratify=train_2["Category"])

def normalize(dataset):
    for col in dataset:
        if dataset[col].dtypes in ['int64', 'float64'] and col != 'Category':
            dataset[col] = (
                (dataset[col]-dataset[col].min()) /
                (dataset[col].max() - dataset[col].min())
            )
normalize(train_1)
normalize(train_2)
normalize(train_3)
normalize(train_4)
normalize(test_set)
```

This results in 5 datasets for us to use, 4 of which will be used in 4-fold cross validation and 1 hold-out set will be used for testing models once hyperparameters have been decided. The datasets to be used in 4-fold cross validation have a length of 323 while the test set has 324 items, and each have 38 liquid items in the set, apart from the `train_2` set which only has 37.

Principal Component Analysis

After our pre-processing, our data still contains many features (63 to be exact), and so before we proceed with any other models, we should identify the most important nutrients to narrow down the machine learning models we build later. To do this, we will use principal component analysis, which is a dimensionality reduction method. It was decided that for our analysis, we would use the top 10 most important features, and to find these features, we would create 10 principal components, which explains a total of 89.15% of variation in our data, and take the top contributor to these principal components (greatest eigenvalues) as the features we use for analysis. To perform the principal component analysis, the Sklearn library was used:

```
pca = PCA(n_components=10)
pca.fit(dataset_attributes)

# Get names of the top features
feature_names = []
for i in range(10):
    feature_names.append(dataset_attributes.columns[np.abs(pca.components_[i]).argmax()])

df = pd.DataFrame(feature_names)
```



	0
0	Moisture (water) \n(g)
1	Available carbohydrate, with sugar alcohols \n(g)
2	Fat, total \n(g)
3	Starch \n(g)
4	Magnesium (Mg) \n(mg)
5	Total long chain omega 3 fatty acids, equated ...
6	Pyridoxine (B6) \n(mg)
7	25-hydroxy cholecalciferol (25-OH D3) \n(ug)
8	Total polyunsaturated fatty acids, equated \n(g)
9	Total folates \n(ug)

From the output above, we can observe the top 10 features which explained the most variance in our dataset, and so these are the features we will continue using for the remainder of this report.

Nearest Neighbours Clustering

The first model we will be testing is using k-Nearest Neighbours Classification to see if we can classify our foods as either solids or liquids based off the properties recorded by the dataset. We first try building a k-Nearest Neighbours classifier by using all 10 of the features we have selected previously using principal component analysis, with an initial k of 3, and using 4-fold cross validation to attain the results across our 4 training sets. To gauge the performance of this classifier, there are many possible metrics we could use, though since our classes are significantly imbalance, we will look at the F_1 score, precision and recall, to ensure that our model is capable of handling the class imbalances. Since we use 4-fold cross validation to judge the performance of our models, we will not use the held-out testing set at all until we have finalised model parameters. Instead, we use the 4 "training" sets, where we rotate through the sets, using three of the sets to train the model and the other set to test the model's performance on previously unseen data. The code and results are as follows:

```
train_precision = []
train_recall = []
train_f1 = []
test_precision = []
test_recall = []
test_f1 = []

for i in range(len(train_sets)):
    classifier = KNeighborsClassifier(n_neighbors=3)

    # Get training sets for 4-fold cross validation
    sets = []
    for j in range(len(train_sets)):
        if i != j:
            sets.append(train_sets[j])
    fold_train = pd.concat(sets)

    # Get metrics
    classifier.fit(fold_train[feature_names], fold_train["Category"])
    train_precision.append(precision_score(fold_train["Category"],
classifier.predict(fold_train[feature_names])))
    train_recall.append(recall_score(fold_train["Category"],
classifier.predict(fold_train[feature_names])))
    train_f1.append(f1_score(fold_train["Category"],
classifier.predict(fold_train[feature_names])))
    test_precision.append(precision_score(train_sets[i]["Category"],
classifier.predict(train_sets[i][feature_names])))
    test_recall.append(recall_score(train_sets[i]["Category"],
classifier.predict(train_sets[i][feature_names])))
    test_f1.append(f1_score(train_sets[i]["Category"],
classifier.predict(train_sets[i][feature_names])))

result = pd.DataFrame(
    {
        'Training': [
            np.sum(train_precision)/len(train_precision),
            np.sum(train_recall)/len(train_recall), np.sum(train_f1)/len(train_f1)
        ],
        'Testing': [
            np.sum(test_precision)/len(test_precision),
            np.sum(test_recall)/len(test_recall), np.sum(test_f1)/len(test_f1)
        ]
    },
    index = ['Precision', 'Recall', 'F1']
)

print(result)
```

Outputs:

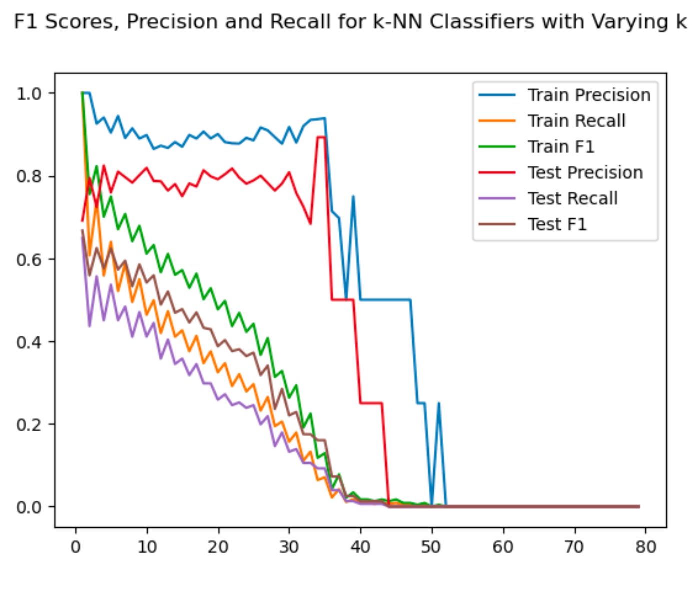
```
           Training   Testing
Precision  0.925853  0.723691
Recall     0.741636  0.556543
F1         0.823053  0.625105
```

From the above, we can see that overall, the model performs relatively well, with a high training F1 score, and relatively high precision and recall, given our hugely imbalanced classes. However, the model attained a testing F1 score of 0.62, with low precision and very low recall. Although this classifier is functional and able to classify our foods into either solids or liquids, adjusting the k value could help us achieve better results, since our algorithm would then consider more of the surrounding points around each data point when classifying. With a k value of 5, the metrics become:

```
           Training   Testing
Precision  0.904032  0.759386
Recall     0.640176  0.536629
F1         0.749452  0.623839
```

Here, we can see that by increasing the number of neighbours we consider in our classifier, we have actually achieved a slightly worse result, with both our training and testing datasets achieving lower overall F1 scores. This may be because when our model decides on the class for each of our data points, it takes into account the distances between more points around it, and it may currently be considering many points which are all of the wrong class, when deciding the class for our data points. Our points which should be part of class 1 may be close to many class 1 and class 0 points, with the surrounding class 1 points being closer than the surrounding class 0 points. So, when using a smaller k value, the closest class 1 points will result in our point being labelled class 1. But, when using a larger k, if the surrounding class 0 points outnumber the surrounding class 1 points, our point will be misclassified as class 0.

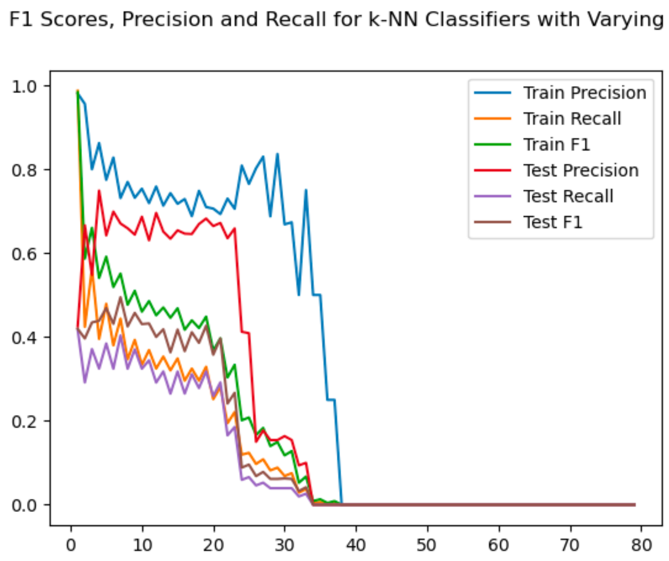
We want to try and optimise the k-value that we pick, so we will try different k values on our models and plot the F1, precision and recall scores so that we can identify the optimal value. By looping through all k-values from 1 to 80, and plotting the metrics of the model as the hyperparameter changes, we get the following result:



The code used to obtain the graph above can be found in Appendix A. We find above that with an increase in k, our precision, recall and F1 scores all decrease significantly, plateauing at about $k = 50$. Using the graph above, it seems that our ideal k-value would be 1.

Now, the models we have been considering so far have been using all 10 dimensions (features) that we chose earlier. However, the performance of K-Nearest Neighbour Models can be negatively affected when we use a greater number of dimensions, referred to as the "Curse of Dimensionality". The reason for this is because as we increase dimensions, the distance between our points can vary much more significantly, and since the K-Nearest Neighbours algorithm performs classification based off the distances between points, with our points potentially further apart due to more dimensions being used, the performance can be worse. So, even though our classification model has been performing relatively well, we want to evaluate whether using fewer dimensions can result in better classification accuracy.

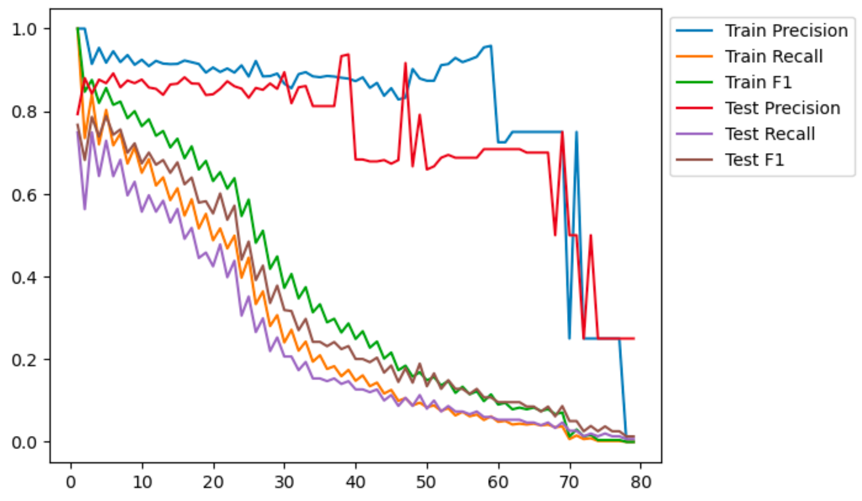
To pick the dimensions to use, we choose the top two dimensions which explained the most variability from the Principal Component Analysis, which were “Moisture” and “Available carbohydrate, with sugar alcohols”. Using these two dimensions, we test all k-values from 1 to 80 to check the metrics and performance of this new model:



From our metrics above, we observe that in our case, the performance of all the classifiers do not improve when we use fewer dimensions, and in fact, they actually worsen. Previously, our testing F1 score could reach a maximum of close to 0.7 with a training F1 score of 1.0. However, by using fewer dimensions, our F1 score for our testing has fallen drastically to a peak of about 0.5 at $k = 8$. This shows that although K-Nearest Neighbours is an algorithm whose accuracy can generally suffer when using greater dimensions, in our case, having greater dimensions actually benefits the model instead. The reason for this may be because our data has obvious clusters (such as in the Moisture feature), resulting in the majority of solid and liquid classes being classified correctly quite easily, and by adding more dimensions (all of which probably do not add a significant amount of noisy space between our data points), the algorithm has more information to distinguish between cases which may be on the border between the two classes. Another possible reason is that 10 features is not a “significant” number of features, since we do have 1616 total data points, and so using more features is not detrimental to our models’ performance.

Since we find that using fewer dimensions in our model actually results in worse results, let’s test the model’s performance using all 59 numerical features we have available (excluding the “Category” column since these contain the true classes).

F1 Scores, Precision and Recall for k-NN Classifiers with Varying k



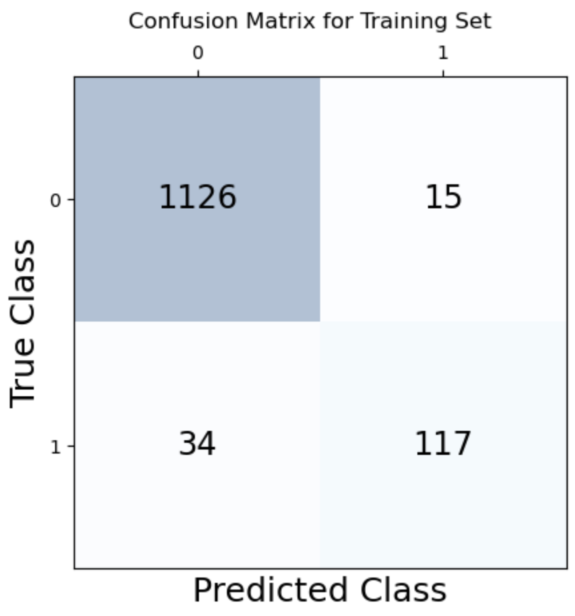
From our plots above, we can see that by including all of our features, we achieve the highest overall training and testing F1 scores of 0.86 and 0.79, respectively, at $k = 5$. Thus, we can conclude that overall, the most optimal K-Nearest Neighbours model for this classification task uses all of the dimensions available to us, and has $k = 5$. Now that we have identified the ideal hyperparameters for our model, we can use our hold-out set to test the model:

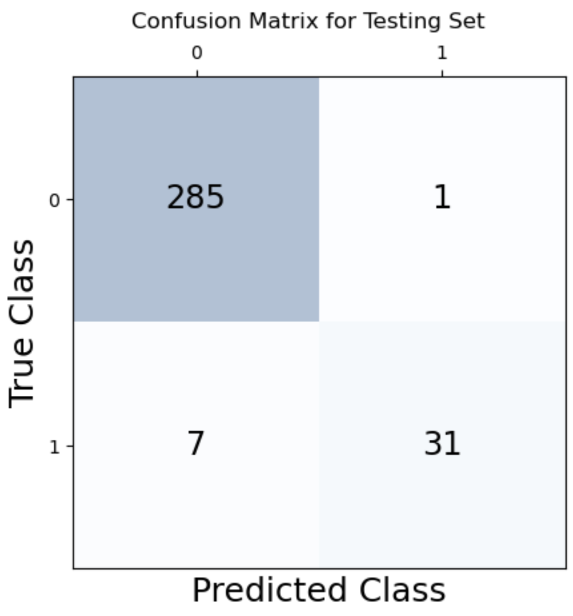
```
classifier = KNeighborsClassifier(n_neighbors=5)
fold_sets = pd.concat(train_sets)
classifier.fit(fold_sets[num_labels], fold_sets["Category"])

print("Testing Set Precision:", precision_score(test_set["Category"],
classifier.predict(test_set[num_labels])))
print("Testing Set Recall:", recall_score(test_set["Category"],
classifier.predict(test_set[num_labels])))
print("Testing Set F1 Score:", f1_score(test_set["Category"],
classifier.predict(test_set[num_labels])))
```

```
Testing Set Precision: 0.9705882352941176
Testing Set Recall: 0.868421052631579
Testing Set F1 Score: 0.9166666666666667
```

From the outputs above, these are very strong results, with the K-Nearest Neighbours classifier finding the foods which are liquids very precisely, although the recall still seems to be quite low, meaning that the classifier is still not yet correctly classifying all the liquid foods that are in our dataset. However, overall, with a F1 score of 0.91, this classifier performs well at its task. To judge how the K-Nearest Neighbours classifier is categorising the classes, we can create a confusion matrix:

```
def create_confusion_matrix(X, Y, model, title):
    matrix = confusion_matrix(Y, model.predict(X))
    fig, ax = plt.subplots()
    ax.matshow(matrix, cmap=plt.cm.Blues, alpha=0.3)
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[1]):
            ax.text(x=j, y=i, s=matrix[i, j], va='center', ha='center', size='xx-large')
    ax.tick_params(top=True, labeltop=True, bottom=False, labelbottom=False)
    plt.xticks(ticks=[0, 1], labels=[0, 1])
    plt.yticks(ticks=[0, 1], labels=[0, 1])
    plt.xlabel('Predicted Class', fontsize=18)
    plt.ylabel('True Class', fontsize=18)
    plt.suptitle(title)
    plt.show()

create_confusion_matrix(fold_sets[num_labels], fold_sets["Category"], classifier, 'Confusion 
Matrix for Training Set')
create_confusion_matrix(test_set[num_labels], test_set["Category"], classifier, 'Confusion 
Matrix for Testing Set')
```





From our confusion matrices above, we can see that the model has quite strong performance on both training and testing sets. These are relatively low false positives and false negatives, especially when compared to the true positive and true negative cases reported.

Logistic Regression

From the K-Nearest Neighbours models that we implemented above, we can see that the maximum F1 score that we achieved as about 0.91, with a recall rate of 86%. Now, we would like to try the same classification problem, but using logistic regression as the method to compare the results we can attain, and hopefully improve on the recall score. Again, since logistic regression also suffers from the "Curse of Dimensionality", where a greater number of predictors can lead to an overfitting on data and less accuracy, we will test our logistic regression models with two different feature spaces – one using the top 10, and one using all available features. The following code was used for creating and evaluating the models:

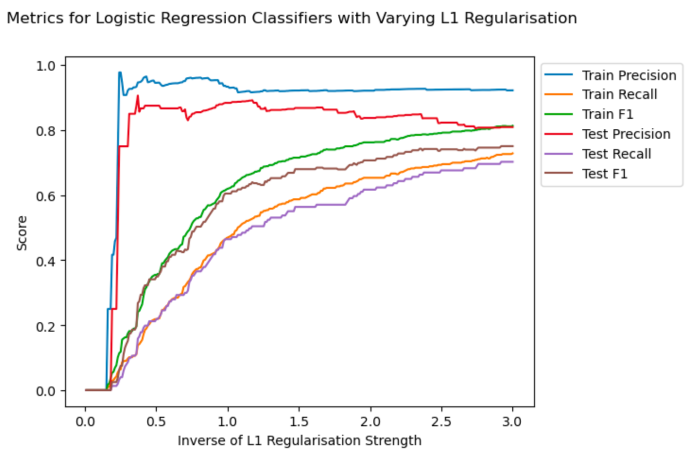
```
train_precision = []
train_recall = []
train_f1 = []
test_precision = []
test_recall = []
test_f1 = []

for i in range(len(train_sets)):
    log_regr = LogisticRegression(penalty=None, max_iter=5000, solver='saga')

    # Get training sets for 4-fold cross validation
    sets = []
    for j in range(len(train_sets)):
        if i != j:
            sets.append(train_sets[j])
    fold_train = pd.concat(sets)

    # Get metrics
    log_regr.fit(fold_train[feature_names], fold_train["Category"])
    train_precision.append(precision_score(fold_train["Category"],
log_regr.predict(fold_train[feature_names])))
    train_recall.append(recall_score(fold_train["Category"],
log_regr.predict(fold_train[feature_names])))
    train_f1.append(f1_score(fold_train["Category"],
log_regr.predict(fold_train[feature_names])))
    test_precision.append(precision_score(train_sets[i]["Category"],
log_regr.predict(train_sets[i][feature_names])))
    test_recall.append(recall_score(train_sets[i]["Category"],
log_regr.predict(train_sets[i][feature_names])))
    test_f1.append(f1_score(train_sets[i]["Category"],
log_regr.predict(train_sets[i][feature_names])))

result = pd.DataFrame(
    {
        'Training': [
            np.sum(train_precision)/len(train_precision),
            np.sum(train_recall)/len(train_recall), np.sum(train_f1)/len(train_f1)
        ],
        'Testing': [
            np.sum(test_precision)/len(test_precision),
            np.sum(test_recall)/len(test_recall), np.sum(test_f1)/len(test_f1)
        ]
    },
    index = ['Precision', 'Recall', 'F1']
)

print(result)
```

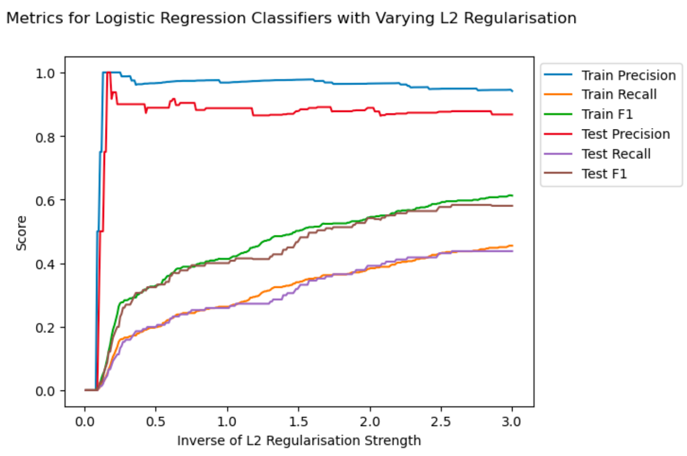
```
           Training   Testing
Precision  0.728557  0.725165
Recall     0.388449  0.370733
F1         0.500100  0.480634
```

Here, this logistic regression model with only the top 10 features performs significantly worse than our initial K-Nearest Neighbours model, with a F1 score of about 0.5 for both training and testing. Testing the logistic regression model on all of the available features, we get:

```
           Training   Testing
Precision  0.895702  0.726905
Recall     0.852158  0.742532
F1         0.873102  0.725643
```

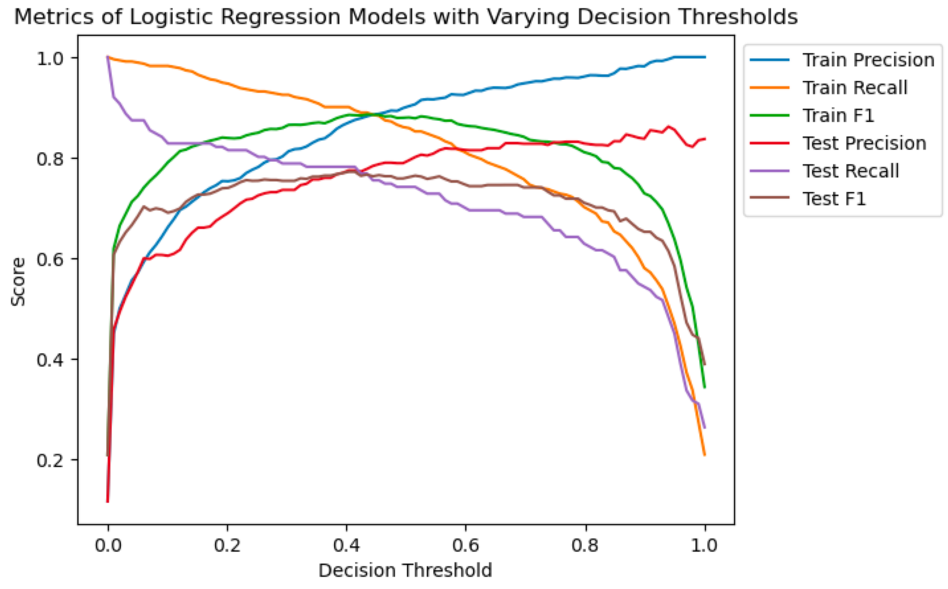
Now, this is an improvement and our model still performs much better with more dimensions, so we will continue using all of the features. We can also see that the model is not overfitting on the training data, since it is still able to provide relatively good classification results on the testing set, even though the training F1 score is higher. Thus, regularisation techniques are unlikely to help us achieve better model performance. We can also observe this by plotting the metrics that we achieve as we change the regularisation strengths:





The C argument that we pass to the logistic regression classifier is the inverse of the regularisation strength, meaning that as we increase this value, the weaker the regularisation effects become. We can see from both plots above that as we increase C, our model F1 scores improve greatly, and thus we can conclude that both $L1$ and $L2$ regularisation are not necessary for our model. A possible reason for why $L1$ regularisation is not needed is that it penalises the sum of absolute value of weights and therefore favours sparse solutions. This likely leads to many of our features getting low or 0 coefficients. However, as we have shown, our models tend to perform better with more features, and so $L1$ regularisation defeats the purpose of including all of our features in the model. The reason for $L2$ regularisation not helping with improving model performance is that it combats overfitting by forcing our weights to be smaller, and this could be resulting in the model underfitting the data.

Now, although our logistic regression model is achieving a testing accuracy of 0.73, it is suffering from a class imbalance. Because there are many more class 0 solids than class 1 liquids and we can see that our recall is quite low, there is a large number of false negatives (class 1 being predicted as class 0). So, to improve on our existing logistic regression classifier, we can try to vary the decision threshold. By changing the threshold, we can change the decision that the logistic regression classifier outputs. If we lower this threshold, more class 1's will be predicted, whilst raising this threshold will result in more class 0's being predicted. Since we would like more class 1's to be predicted, we can expect that the ideal threshold will be lower than 0.5. To find the ideal threshold, we plot the metrics we achieve as we change the decision threshold:



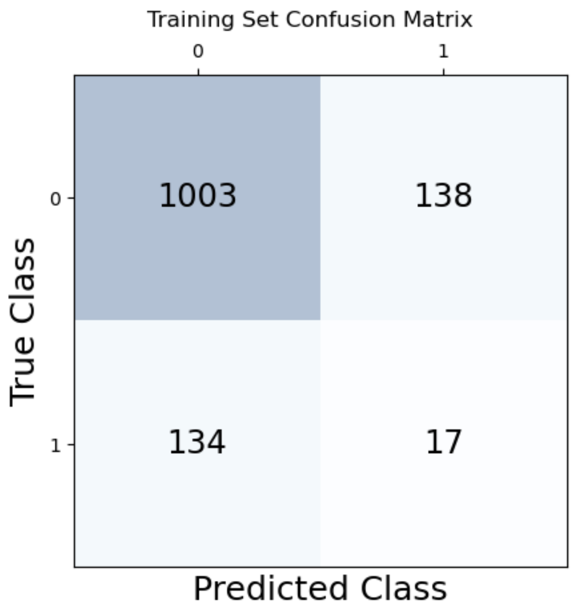
The code for producing the above plot can be found in Appendix B. So, from the plot above, we can see that with a low decision threshold, our recall for both training and testing is very high. This is due to us classifying the majority of our data points as class 1, therefore meaning that we are finding the majority of class 1 cases (which is the definition of recall). Inversely, with a low decision threshold, our precision is very low and this is because we are classifying all cases as class 1 and therefore our classifier has a large number of false positives, resulting in low precision. As we increase our decision threshold, precision starts to increase and recall begins decreasing since we are now classifying more class 0's correctly, but that results in an increase in false negatives (where a class 1 is incorrectly categorised as class 0).

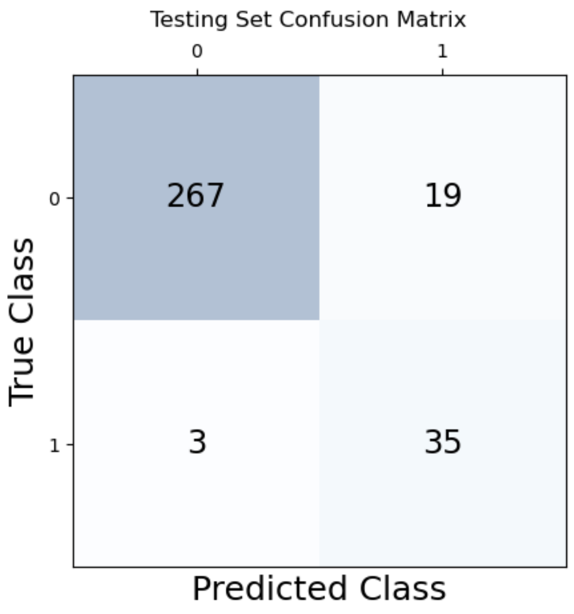
Overall from the plot of varying decision thresholds, we find that testing accuracy is maximised at 0.4, making this the ideal threshold for our model. Now that we have determined the optimal hyperparameters for our logistic regression classifier, we can use the held-out testing set to determine the performance of our optimal logistic regression classifier:

```
log_regr = LogisticRegression(penalty=None, max_iter=5000, solver='saga')

# Get training sets for 4-fold cross validation
all_train = pd.concat(train_sets)

log_regr.fit(all_train[num_labels], all_train["Category"])

# Get correct classes
train_predict_probs = log_regr.predict_proba(all_train[num_labels])
train_predict_probs_class_1 = train_predict_probs[:,1]
test_predict_probs = log_regr.predict_proba(test_set[num_labels])
test_predict_probs_class_1 = test_predict_probs[:,1]

train_predicted_labels = [1 if prob > 0.4 else 0 for prob in train_predict_probs_class_1]
test_predicted_labels = [1 if prob > 0.4 else 0 for prob in test_predict_probs_class_1]

result = pd.DataFrame(
    {
        'Training': [
            precision_score(all_train["Category"], train_predicted_labels),
            recall_score(all_train["Category"], train_predicted_labels),
            f1_score(all_train["Category"], train_predicted_labels)
        ],
        'Testing': [
            precision_score(test_set["Category"], test_predicted_labels),
            recall_score(test_set["Category"], test_predicted_labels),
            f1_score(test_set["Category"], test_predicted_labels)
        ]
    },
    index = ['Precision', 'Recall', 'F1']
)
print(result)
```

```
           Training   Testing
Precision  0.845161  0.648148
Recall     0.867550  0.921053
F1         0.856209  0.760870
```

From these results, we can see that we have been able to achieve a 0.76 F1 score on the unseen testing set, with a precision of 0.64 and recall of 0.92. These results are quite low compared to the optimal results that our K-Nearest Neighbour model was able to achieve, and are not a significant improvement from the logistic model that had no change in decision threshold. Building a confusion matrix, the code for both of which can be found in Appendix C:





We can observe that the performance of the logistic regression classifier is not ideal, with a significant number of false positives and false negatives in training. However, it seems to perform quite well on the testing set, with very little false negatives, though there is still room for improvement with regards to precision.

Neural Network (Multi-Layer Perceptron)

The final model we will try for our classification problem between solids and liquids is to use a neural network, specifically a multi-layer perceptron, which is a type of neural network consisting of an input layer, any number of hidden (dense) layers, and an output layer. Here, we will be exploring the effects of different numbers of hidden layers, activation functions, optimizers and learning rates. To decide on the initial model we will begin exploring with, we will use the GridSearchCV class provided in the Sklearn library to perform grid search to find the optimal parameters. Sklearn also provides us with the MLPClassifier class that we will use, so that we can simply specify the parameters for our multi-layer perceptron for testing. Since we have a classification problem, the outputs of our neural network need to be the probabilities that the input is a certain class, so that we can assume that the input has the class with the highest predicted probability. Additionally, since our problem is binary classification (because we only have two classes – liquids or solids), we will use the logistic activation function in the final layer.

```
mlp = MLPClassifier()

# Parameters to test for grid search
parameter_space = {
    'hidden_layer_sizes': [(50), (50, 50), (50, 50, 50), (100), (100, 100), (100, 100, 100),
(200, 200, 200)],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['sgd', 'adam', 'lbfgs'],
    'learning_rate': ['constant', 'adaptive']
}
# Grid Search with built-in stratified cross validation of k=4
clf = GridSearchCV(mlp, parameter_space, n_jobs=-1 , cv=4)
```

```
# Test MLP on all parameters, using only the train data (model will not see held-out test set)
clf.fit(pd.concat(train_sets)[num_labels].to_numpy(),
pd.concat(train_sets)["Category"].to_numpy())

print("Best Parameters:", clf.best_params_)
```

```
Best Parameters: {'activation': 'relu', 'hidden_layer_sizes': (200, 200, 200), 'learning_rate':
'adaptive', 'solver': 'adam'}
```

From the output of the code above, we can see that by using 4-fold cross validation and grid search, the multi-layer perceptron that had the greatest number of hidden layers and greatest number of neurons in each layer performed the best. This is likely because with more neurons, the model can update each neuron's weights more precisely and is therefore able to be more complex and find the patterns it needs in order to classify our foods more accurately. We can also see that the ReLU activation function, along with using an adaptive learning rate and the Adam optimizer yielded the most optimal model. The ReLU activation function works by deactivating neurons in the hidden layers with a value less than 0, and since none of our nutrient features should contain negative values, this likely helped to increase performance of the model by ignoring any negative values that resulted from the model weights which may have influenced the final class probabilities. Learning rate, which influences the step size when minimizing the loss function, is set to be adaptive. With a larger learning rate, the changes to weights in the neural network will be greater in each iteration, while with a smaller learning rate, the changes become smaller and thus the model learns the "finer" patterns in the data. By using an adaptive learning rate, where the learning rate is only changed to become smaller when the training loss does not decrease for two consecutive iterations, our model is able to learn the more general patterns initially by making larger model updates, before finding the more "detailed" patterns with the smaller learning rate to optimise results.

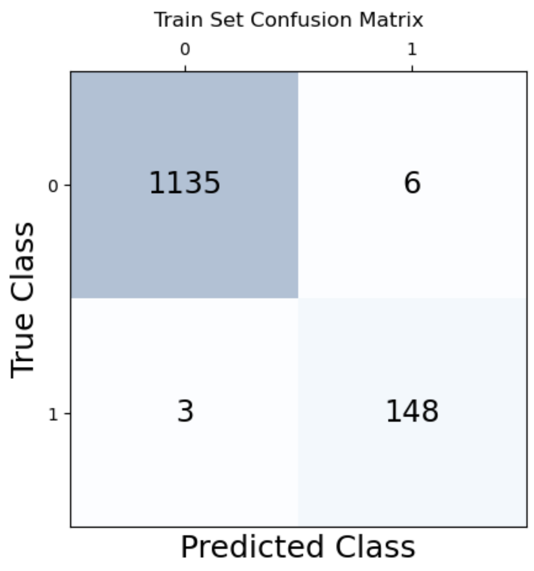
To judge the performance of the ideal model we have found using grid search and the other models which were not chosen, we will train separate models to find their precision, recall and F1 scores. The optimal model found by grid search was trained as follows:

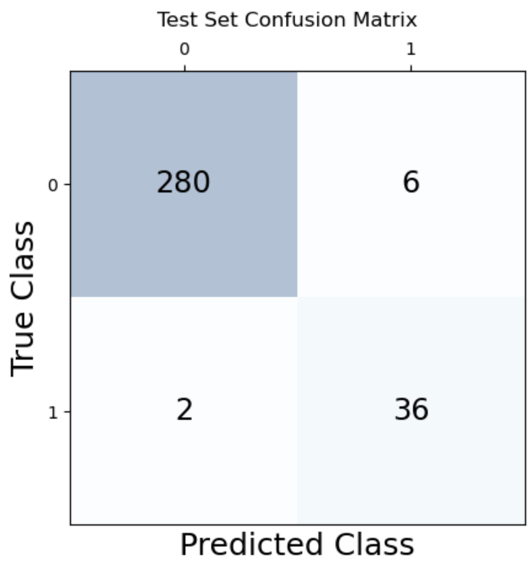
```
mlp_classifier = MLPClassifier(hidden_layer_sizes=(200, 200, 200), activation='relu',
solver='adam', learning_rate='adaptive')
mlp_classifier.fit(pd.concat(train_sets)[num_labels], pd.concat(train_sets)["Category"])

print(
    "Training Precision:", precision_score(pd.concat(train_sets)["Category"],
mlp_classifier.predict(pd.concat(train_sets)[num_labels]),
    "\nTraining Recall:", recall_score(pd.concat(train_sets)["Category"],
mlp_classifier.predict(pd.concat(train_sets)[num_labels])),
    "\nTraining F1:", f1_score(pd.concat(train_sets)["Category"],
mlp_classifier.predict(pd.concat(train_sets)[num_labels])),
    "\nTesting Precision:", precision_score(test_set["Category"],
mlp_classifier.predict(test_set[num_labels])),
    "\nTesting Recall:", recall_score(test_set["Category"],
mlp_classifier.predict(test_set[num_labels])),
    "\nTesting F1:", f1_score(test_set["Category"],
mlp_classifier.predict(test_set[num_labels]))
)
```

```
Training Precision: 0.9735099337748344
Training Recall: 0.9735099337748344
Training F1: 0.9735099337748344
Testing Precision: 0.9459459459459459
Testing Recall: 0.9210526315789473
Testing F1: 0.9333333333333332
```

Here, we can see that this model has very high F1, precision and recall, and performs better than all our previous models overall, for both the training and testing sets. The confusion matrices for our training and testing sets can be seen below.

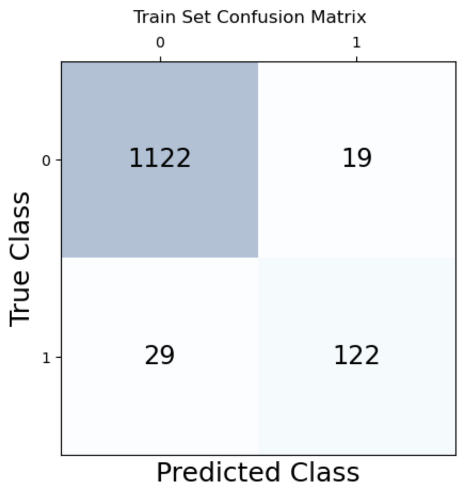
```
create_confusion_matrix(pd.concat(train_sets)[num_labels], pd.concat(train_sets)["Category"],
mlp_classifier, "Train Set Confusion Matrix")
create_confusion_matrix(test_set[num_labels], test_set["Category"], mlp_classifier, "Test Set
Confusion Matrix")
```

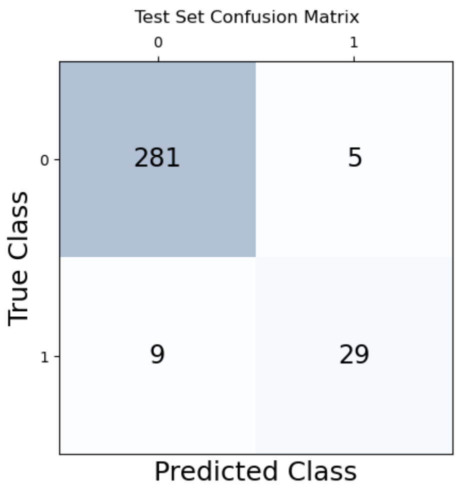




Here, we can observe from the confusion matrices of the training and testing set that the model performs well, with very few false negatives and false positives. There is also no sign of overfitting on the training set, as the performance of the classifier on both datasets are quite good. Now, let us see the performance of a model with significantly fewer layers and neurons. By using only one hidden layer with 50 neurons (with the same activation function, learning rate and optimizer), our metrics and confusion matrices become:

```
Training Precision: 0.8652482269503546
Training Recall: 0.8079470198675497
Training F1: 0.8356164383561644
Testing Precision: 0.8529411764705882
Testing Recall: 0.7631578947368421
Testing F1: 0.8055555555555555
```



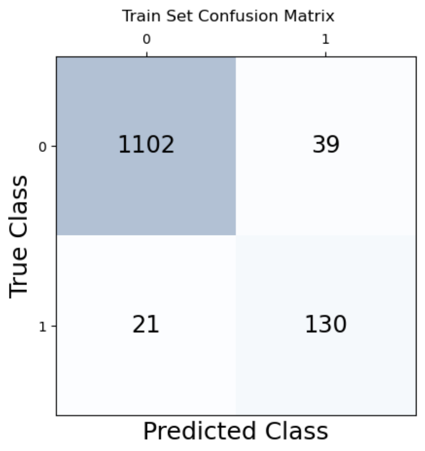


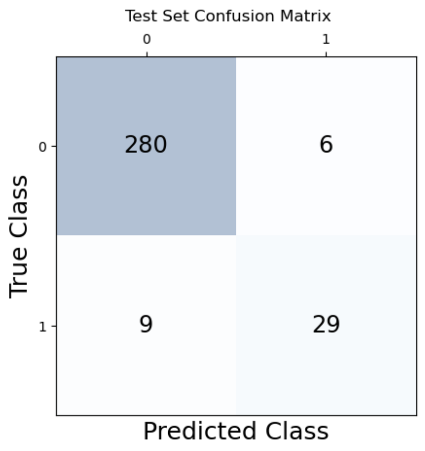
It is obvious that significantly decreasing the complexity of our model by decreasing the depth and number of neurons results in a great drop in performance from our model. Our F1 score drops by 0.1 and we have a 24% decrease in recall. The results are also obvious in our confusion matrices, where we see a much greater number of false positives and false negatives. If we were to increase the complexity of our model significantly, we could also expect to see overfitting on our training set, where the model would have close to 100% precision and recall when using the training set, but significantly low performance on the held-out testing dataset. To further improve on this model, we could try to find the optimal number of layers and number of neurons in each layer, though the search space for this task would be very large as there are many possible combinations for multi-layer perceptron architectures.

Examining the effects that activation functions have on the performance of a feed-forward neural network, we will try the same model with three hidden layers, each with 200 neurons, but change the activation functions between ReLU, logistic, identity (no activation) and tanh. We have already seen the performance of the ReLU function, with a testing F1 score of 0.933. The metrics and confusion matrices of the three other activation functions are displayed below:

Logistic:

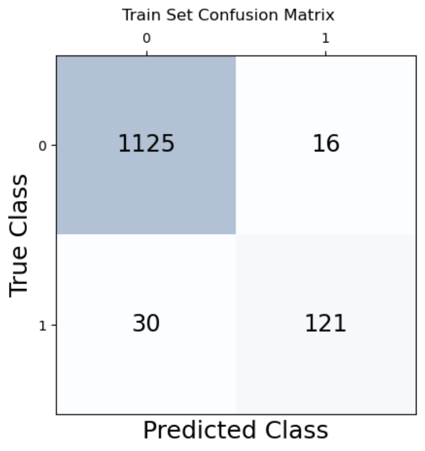
```
Training Precision: 0.8968253968253969
Training Recall: 0.7483443708609272
Training F1: 0.8158844765342962
Testing Precision: 0.8529411764705882
Testing Recall: 0.7631578947368421
Testing F1: 0.8055555555555555
```

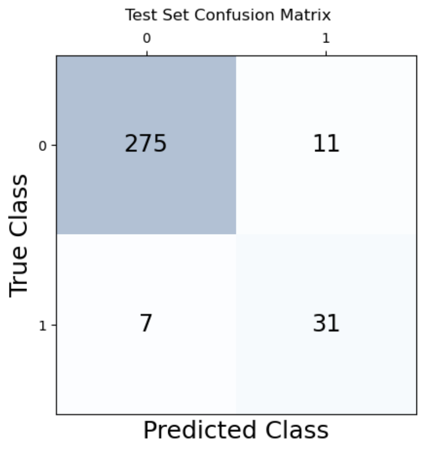




Identity:

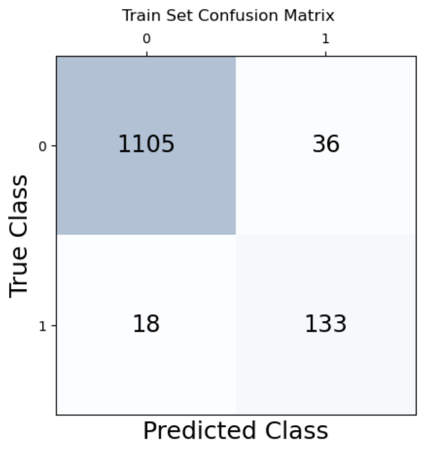
```
Training Precision: 0.8832116788321168
Training Recall: 0.8013245033112583
Training F1: 0.8402777777777778
Testing Precision: 0.7380952380952381
Testing Recall: 0.8157894736842105
Testing F1: 0.775
```

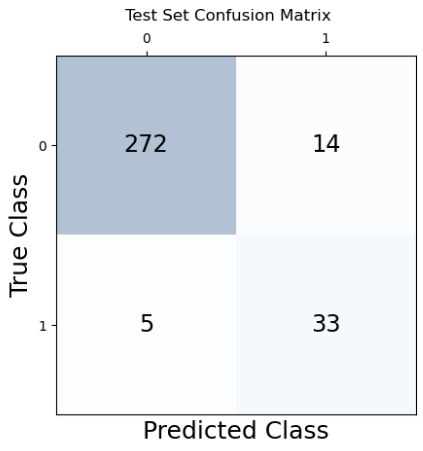




Tanh:

```
Training Precision: 0.7869822485207101
Training Recall: 0.8807947019867549
Training F1: 0.8312499999999999
Testing Precision: 0.7021276595744681
Testing Recall: 0.868421052631579
Testing F1: 0.7764705882352942
```





From the three activation functions tested above, we can see that all perform worse than the ReLU activation function. We see that the logistic activation function performs the best between the three extra functions tested, whilst no activation and the Tanh activation function seem to have roughly the same performance. It is interesting to note that the two best performing activation functions, ReLU and logistic both do not activate neurons with negative values, while identity and tanh do allow negative values to be passed through the network and resulted in worse performance. Earlier, it was mentioned that the ReLU activation may have been chosen by grid search due to its characteristic of not considering negative values, and by testing these different activation functions, this could be the case for this model.

Overall, after varying many sets of hyperparameters, the best performing model that has been found for our classification problem was the multi-layer perceptron, using three hidden layers with 200 neurons each, as well as the ReLU activation on all hidden layers, logistic activation on the output layer, adaptive learning rate and the Adam optimizer. With this model, we were able to achieve a F1 score of 0.933, which is a significant improvement compared to the previous K-Nearest Neighbours and logistic regression models that we tested.

Appendix A

```
all_train_precision = []
all_train_recall = []
all_train_f1 = []
all_test_precision = []
all_test_recall = []
all_test_f1 = []
for k in range(1, 80):
    train_precision = []
    train_recall = []
    train_f1 = []
    test_precision = []
    test_recall = []
    test_f1 = []
    for i in range(len(train_sets)):
        classifier = KNeighborsClassifier(n_neighbors=k)
        # Get training sets for 4-fold cross validation
        sets = []
        for j in range(len(train_sets)):
            if i != j:
                sets.append(train_sets[j])
        fold_train = pd.concat(sets)
        # Get metrics
        classifier.fit(fold_train[feature_names], fold_train["Category"])
        train_precision.append(precision_score(fold_train["Category"],
classifier.predict(fold_train[feature_names])))
        train_recall.append(recall_score(fold_train["Category"],
classifier.predict(fold_train[feature_names])))
        train_f1.append(f1_score(fold_train["Category"],
classifier.predict(fold_train[feature_names])))
        test_precision.append(precision_score(train_sets[i]["Category"],
classifier.predict(train_sets[i][feature_names])))
        test_recall.append(recall_score(train_sets[i]["Category"],
classifier.predict(train_sets[i][feature_names])))
        test_f1.append(f1_score(train_sets[i]["Category"],
classifier.predict(train_sets[i][feature_names])))

    all_train_precision.append(np.sum(train_precision)/len(train_precision))
    all_train_recall.append(np.sum(train_recall)/len(train_recall))
    all_train_f1.append(np.sum(train_f1)/len(train_f1))
    all_test_precision.append(np.sum(test_precision)/len(test_precision))
    all_test_recall.append(np.sum(test_recall)/len(test_recall))
    all_test_f1.append(np.sum(test_f1)/len(test_f1))

plt.plot(range(1, 80), all_train_precision)
plt.plot(range(1, 80), all_train_recall)
plt.plot(range(1, 80), all_train_f1)
plt.plot(range(1, 80), all_test_precision)
plt.plot(range(1, 80), all_test_recall)
plt.plot(range(1, 80), all_test_f1)
plt.suptitle("F1 Scores, Precision and Recall for k-NN Classifiers with Varying k")
plt.legend(["Train Precision", "Train Recall", "Train F1", "Test Precision", "Test Recall",
"Test F1"], bbox_to_anchor=(1, 1))
plt.show()
```

Appendix B

```
all_train_precision = []
all_train_recall = []
all_train_f1 = []
all_test_precision = []
all_test_recall = []
all_test_f1 = []

r = 0
while r <= 1:
    train_precision = []
    train_recall = []
    train_f1 = []
    test_precision = []
    test_recall = []
    test_f1 = []

    for i in range(len(train_sets)):
        log_regr = LogisticRegression(penalty=None, max_iter=5000, solver='saga')

        # Get training sets for 4-fold cross validation
        sets = []
        for j in range(len(train_sets)):
            if i != j:
                sets.append(train_sets[j])
        fold_train = pd.concat(sets)

        log_regr.fit(fold_train[num_labels], fold_train["Category"])

        # Get correct classes
        train_predict_probs = log_regr.predict_proba(fold_train[num_labels])
        train_predict_probs_class_1 = train_predict_probs[:,1]
        test_predict_probs = log_regr.predict_proba(train_sets[i][num_labels])
        test_predict_probs_class_1 = test_predict_probs[:,1]

        train_predicted_labels = [1 if prob > r else 0 for prob in train_predict_probs_class_1]
        test_predicted_labels = [1 if prob > r else 0 for prob in test_predict_probs_class_1]

        train_precision.append(precision_score(fold_train["Category"], train_predicted_labels, 
zero_division=0))
        train_recall.append(recall_score(fold_train["Category"], train_predicted_labels, 
zero_division=0))
        train_f1.append(f1_score(fold_train["Category"], train_predicted_labels, 
zero_division=0))
        test_precision.append(precision_score(train_sets[i]["Category"], test_predicted_labels, 
zero_division=0))
        test_recall.append(recall_score(train_sets[i]["Category"], test_predicted_labels, 
zero_division=0))
        test_f1.append(f1_score(train_sets[i]["Category"], test_predicted_labels, 
zero_division=0))

    r += 0.01

    all_train_precision.append(np.sum(train_precision)/len(train_precision))
    all_train_recall.append(np.sum(train_recall)/len(train_recall))
    all_train_f1.append(np.sum(train_f1)/len(train_f1))
    all_test_precision.append(np.sum(test_precision)/len(test_precision))
    all_test_recall.append(np.sum(test_recall)/len(test_recall))
    all_test_f1.append(np.sum(test_f1)/len(test_f1))

plt.plot(np.linspace(0, 1, 100), all_train_precision)
plt.plot(np.linspace(0, 1, 100), all_train_recall)
plt.plot(np.linspace(0, 1, 100), all_train_f1)
plt.plot(np.linspace(0, 1, 100), all_test_precision)
plt.plot(np.linspace(0, 1, 100), all_test_recall)
plt.plot(np.linspace(0, 1, 100), all_test_f1)
plt.legend(["Train Precision", "Train Recall", "Train F1", "Test Precision", "Test Recall",
"Test F1"], bbox_to_anchor=(1, 1))
plt.title('Metrics of Logistic Regression Models with Varying Decision Thresholds')
```

Appendix C

```
# Confusion Matrix of Training Set with altered threshold
matrix = confusion_matrix(train_set["Category"], train_predicted_labels)
fig, ax = plt.subplots()
ax.matshow(matrix, cmap=plt.cm.Blues, alpha=0.3)
for i in range(matrix.shape[0]):
    for j in range(matrix.shape[1]):
        ax.text(x=j, y=i, s=matrix[i, j], va='center', ha='center', size='xx-large')
ax.tick_params(top=True, labeltop=True, bottom=False, labelbottom=False)
plt.xticks(ticks=[0, 1], labels=[0, 1])
plt.yticks(ticks=[0, 1], labels=[0, 1])
plt.xlabel('Predicted Class', fontsize=18)
plt.ylabel('True Class', fontsize=18)
plt.suptitle('Training Set Confusion Matrix')
plt.show()

# Confusion Matrix of Testing Set with altered threshold
matrix = confusion_matrix(test_set["Category"], test_predicted_labels)
fig, ax = plt.subplots()
ax.matshow(matrix, cmap=plt.cm.Blues, alpha=0.3)
for i in range(matrix.shape[0]):
    for j in range(matrix.shape[1]):
        ax.text(x=j, y=i, s=matrix[i, j], va='center', ha='center', size='xx-large')
ax.tick_params(top=True, labeltop=True, bottom=False, labelbottom=False)
plt.xticks(ticks=[0, 1], labels=[0, 1])
plt.yticks(ticks=[0, 1], labels=[0, 1])
plt.xlabel('Predicted Class', fontsize=18)
plt.ylabel('True Class', fontsize=18)
plt.suptitle('Testing Set Confusion Matrix')
plt.show()
```