Part 1: KNN Implementation from scratch

1. Explanation of the KNN algorithm and its implementation.

To implement the k nearest neighbors' algorithm the initial steps are to load the breast cancer dataset, apply standardization, and split the dataset as 70% training and 30% testing. This is shown below in Figure 1.

```
# (part a) load the breast cancer dataset
data = load_breast_cancer()
x = data.data
y = data.target

# (part b) apply standardization
scaler = StandardScaler()
scaled = scaler.fit_transform(x)

# (part c) split the dataset into training and test sets
x_train, x_test, y_train, y_test = train_test_split(scaled, y, test_size=0.3, random_state=42)
```

Figure 1: Load breast cancer dataset, apply standardization and split the dataset

It can be seen that the *test_size* parameter is set to 0.3 and the *random_state* is equal to 42. 42 is a common value selection for random state as it produces the same training and test sets across different executions [1].

To implement the knn algorithm from scratch and establish a new instance x*, the necessary steps are as follows:

a) Select a value for k
b) Calculate the distance between x* and all other x from $i = 1, \ldots, n$
c) Sort the distance from smallest to largest and identify i for the k smallest distances. The instances with the smallest distances to x* are the nearest neighbors
d) Calculate the proportion of times each class occurs in the k nearest neighbors. For each class, the proportion of times the class occurs are the predicted probabilities of class membership
e) The predicted class for instance x* is the class that occurs most often in the nearest neighbors

Since we will be testing several different k values to determine optimal performance, explanation of step a) will be pushed to question 2. The following step is to calculate distances between the instances which will be implemented with the Euclidean distance, determined in its own function as follows in Figure 2.

```
# (part a) function defn for euclidean distance calculations
def euclidean_distance(x1, x2):
    euc_dist = np.sqrt(np.sum((x1 - x2) ** 2))
    return euc_dist
```

Figure 2: Euclidean distance function

The next step c) is implemented in the following function displayed as Figure 3 where distances get sorted by length and k nearest neighbors are determined.

```python
# function for finding the nearest neighbor
def nearest_neighbor(x_train, y_train, Xtest, k):
    distances = []
    for i, Xtrain in enumerate(x_train):
        dist = euclidean_distance(Xtest, Xtrain)
        distances.append((dist, y_train[i]))

    distances.sort(key=lambda x: x[0])
    return [label for _, label in distances[:k]]
```

Figure 3: Function determining nearest neighbors

Step d) is to calculate the proportion of times each class appears within all the k nearest neighbors. Implementation can be seen as prediction in Figure 4 below.

```python
# function for predicting the majority class
def prediction(neighbors):
    class_count = Counter(neighbors)
    majority = class_count.most_common(1)[0][0]
    return majority
```

Figure 4: Function calculating the majority class between the k nearest neighbors

The final knn function ties all steps together and returns the final y_prediction for the new instance x*, see Figure 5.

```python
# K nearest neighbors algorithm
def knn(x_train, y_train, x_test, k=3):
    y_prediction = []
    for Xtest in x_test:
        neighbors = nearest_neighbor(x_train, y_train, Xtest, k)
        predict = prediction(neighbors)
        y_prediction.append(predict)
    return y_prediction
```

Figure 5: K nearest neighbors' function

2. Description of the optimal k value found and its justification.

The optimal k value is determined by testing various k values and calculating each of their respective accuracies. We first implement the function to calculate prediction accuracy shown in Figure 6.

```python
# compare the correct predictions against total predictions made
def acc(true_val, y_prediction):
    correct = np.sum(true_val == y_prediction)
    accuracy = correct / len(true_val)
    return accuracy
```

Figure 6: Accuracy function

The k values tested include all odd numbers ranging 1 through 15. A function was written which takes, train and test sets as well as all the available k values organized into an array. The highest accuracy is

initially set to zero. We then loop through all potential k values, calculating their accuracies with the function returning both the optimal k value along with its success rate. The implementation is shown below in Figure 7.

```
def k_testing(x_train, y_train, x_test, y_test, k_vals):
    optimal_k = None
    highest_acc = 0
    for k in k_vals:
        y_prediction = knn(x_train, y_train, x_test, k)
        acc_dec = acc(y_test, y_prediction)
        acc_perc = acc_dec * 100
        print(f"Accuracy using k={k}: {acc_perc:.2f}%")
        if acc_perc > highest_acc:
            highest_acc = acc_perc
            optimal_k = k

    print(f"The highest accuracy of the model is {highest_acc:.2f}% with the optimal k value being {optimal_k}")
    return optimal_k, highest_acc

k_vals = [1, 3, 5, 7, 9, 11, 13, 15]
optimal_k, highest_acc = k_testing(x_train, y_train, x_test, y_test, k_vals)
```

Figure 7: Function determining the optimal k value

Output from the above is displayed below in Figure 8. It can be observed that the k value which produces the highest accuracy percentage is k=9. All accuracy values land at approximately 95% with the exception of k=9 that produces a success of ~97%.

```
Accuracy using k=1: 95.32%
Accuracy using k=3: 95.91%
Accuracy using k=5: 95.91%
Accuracy using k=7: 95.91%
Accuracy using k=9: 97.08%
Accuracy using k=11: 95.91%
Accuracy using k=13: 95.91%
Accuracy using k=15: 95.32%
The highest accuracy of the model is 97.08% with the optimal k value being 9
```

Figure 8: Output from the k testing function

3. Evaluation metrics and their interpretation.

The metric to be evaluated for the k nearest neighbors' algorithm is accuracy. Accuracy is determined by comparing the total number of predictions made to the correct predictions made. As discussed in the previous question the optimal k value was determined to be k=9. Accuracy reported with k=9 yielded a success rate of approximately 97.08%. This is a relatively high accuracy and can be seen below in Figure 10. A few other metrics were determined as well such that their results could be compared to the logistic regression model. Implementation and output of these metrics, including precision, recall, and f1 score are all also shown in Figure 9 and Figure 10 respectively.

```
precision_knn = precision_score(y_test, y_prediction_knn)
recall_knn = recall_score(y_test, y_prediction_knn)
f1_score_knn = f1_score(y_test, y_prediction_knn)

print(f"Knn accuracy: {highest_acc:2f}%")
print(f"Knn precision: {precision_knn * 100:2f}%")
print(f"Knn recall: {recall_knn * 100:2f}%")
print(f"Knn f1 score: {f1_score_knn * 100:2f}%")
```

Figure 9: Metric calculations for the knn algorithm

```
Knn accuracy: 97.076023%
Knn precision: 97.247706%
Knn recall: 98.148148%
Knn f1 score: 97.695853%
```

Figure 10: Metric outputs for the knn algorithm

Part 2: Logistic Regression using sklearn

1. Explanation of Logistic Regression and its implementation using sklearn.

To implement logistic regression, the model is first initialized with penalty = None, meaning there is no regularization used. Then the model gets fit with the x_train and y_test data. Finally, the predictions are made with x_test data. Implementation in python is shown below in Figure 11.

```
# initialize a logistic regression model
logisticModel = LogisticRegression(penalty=None)

# fit the model
logisticModel.fit(x_train, y_train)

# make predictions on the test data
y_prediction_logr = logisticModel.predict(x_test)
```

Figure 11: Logistic regression implementation with sklearn

2. Evaluation metrics and their interpretation.

The metrics to be evaluated for the logistic regression model are accuracy, precision, recall, and f1 score. Each of these metric equations are summarized below:

$$Accuracy = \frac{Num\ Correct\ Predictions}{Num\ Total\ Predictions}$$

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

To calculate these metrics the sklearn library can be used and is implemented in python as shown below in Figure 12.

```
acc_logr = accuracy_score(y_test, y_prediction_logr)
precision_logr = precision_score(y_test, y_prediction_logr)
recall_logr = recall_score(y_test, y_prediction_logr)
f1_score_logr = f1_score(y_test, y_prediction_logr)

print(f"Log reg accuracy: {acc_logr * 100:2f}%")
print(f"Log reg precision: {precision_logr * 100:2f}%")
print(f"Log reg recall: {recall_logr * 100:2f}%")
print(f"Log reg f1 score: {f1_score_logr * 100:2f}%")
```

Figure 12: Metric calculations for the logistic regression model

The output for the code shown above is displayed in Figure 13. As we can see the resulting metrics are fairly high overall for the logistic regression model, with accuracy = 95.32%, precision = 99.02%, recall = 93.52% and finally an f1 score of 96.19%.

```
Log reg accuracy: 95.321637%
Log reg precision: 99.019608%
Log reg recall: 93.518519%
Log reg f1 score: 96.190476%
```

Figure 13: Metric outputs for the logistic regression model

3. A comparison of Logistic Regression and KNN performance.

When comparing the metric results back-to-back we can see that overall accuracy of the knn algorithm is higher than the accuracy acquired using logistic regression. However, the precision on log reg is 2% higher than for knn. This tells us that even though knn makes more successful predictions, output for the log reg model yields higher true positives to false positives than knn does. Recall on the knn model is higher than log reg, therefore we know that knn produces higher true positives to false positives. Finally, the f1 score is calculated for each model and yields very similar values. Overall, these statistical metrics tell us that the k nearest neighbors' algorithm has a higher likelihood of producing correct predictions. Output is displayed below in Figure 14.

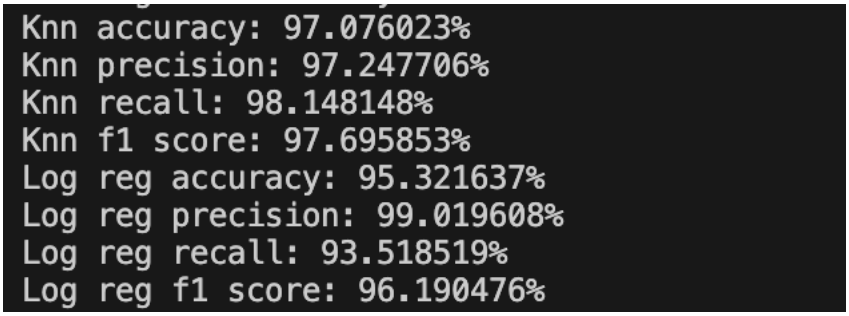

Figure 14: Both metric outputs for the knn algorithm and the logistic regression model

Works Cited

[1] R. Pramoditha, "Why do we set random state in machine learning models?," Medium, 30 April 2022. [Online]. Available: https://towardsdatascience.com/why-do-we-set-a-random-state-in-machine-learning-models-bb2dc68d8431#:~:text=With%20random_state%3D42%20%2C%20we%20get,affect%20the%20model's%20performance%20score. [Accessed 2 October 2024].