HW9 Bonus

Dylan Zhao dxz365

MLP Classifier

MLP Classifier from the scikit-learn library was used to classify the dataset. MLP Classifier is a feed forward artificial neural network used for supervised classification. This model is able utilize linear as well as non-linear activation functions to create decisions boundaries for the data. At each layer, the model computes:

$$z = W \cdot x + b$$

$$\text{output} = f(z)$$

with W being the weights, x being the data input, and b being the bias.

The following is code used for classifying the iris dataset:

```
data = pd.read_csv("irisdata.csv")

X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values


encoder = LabelEncoder()
y = encoder.fit_transform(y)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Scale Features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# ------------------- RELU --------------------
relu_clf = MLPClassifier(
    hidden_layer_sizes=(10,10),
    activation='relu',
    max_iter=2000,
    random_state=42
```

```
)
relu_clf.fit(X_train, y_train)
y_pred = relu_clf.predict(X_test)

print("\nReLU")
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred, target_names=encoder.classes_))
```

The first section of the code reads the iris data and separates the features from the labels. Since, MLP Classifier cannot use string values for labels, an encoder is used to convert them to a format that is usable. Then the data is split into the training data and test data with 70% of the data being used for training. Next, due to the varying values of each dimension, the features were scaled in order to make it so that not one dimension dominates simply due to the nature of that feature having large values compared to the rest. If scaling was not done, this would result in the model to being unable to accurate results as it would simply mainly use one feature for classification.

For this initial model, the ReLu activation model was used. The ReLu activation model uses this formula to calculate its output:

$$f(z) = max(0, z)$$

This function simply returns z if z is positive and 0 if not. As result, this allows more efficient computation and is better used for networks with a lot of hidden layers. This is because if an output is 0, it is similar to as if that neuron didn't fire at all. Additionally, this activation model helps mitigate the vanishing gradient problems, where the gradient values for other functions may end up being too small to make a difference in backpropagation. Instead, the derivative is either 1 or 0, preventing this issue from occurring.

The current model is using 2000 iterations, and trains using the training set and gets tested on the prediction set. The final accuracy and classification report is shown below:

```
ReLU
Accuracy: 0.9111111111111111
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.82      0.93      0.88        15
   virginica       0.92      0.80      0.86        15

    accuracy                           0.91        45
   macro avg       0.92      0.91      0.91        45
weighted avg       0.92      0.91      0.91        45
```

Precision determines how many of the predicted positives were correct while recall determines how many of the actual positives were found. F1 score is the harmonic mean that only is high if both precision and recall are both high. This score is determined by this formula:

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

The macro average treats all classes equally while the weighted average weighs each classes based on sample size. Since all classes have equal number of samples, these values are the same.

As a result the ReLu had an overall accuracy of 91%, with it having issues separating the versicolor and virginica classes.

Other Non-Linearities:

Other non-linear activation functions were also used to determine the performance differences between functions. A linear, sigmoid, and tanh model was used for comparison with all models having 2 hidden layers with 10 neurons each. The following is the code used for all the models:

```
def train_mlp(X_train, y_train, X_test, y_test, activation):

    clf = MLPClassifier(hidden_layer_sizes=(10,10), activation=activation,
max_iter=2000, random_state=42)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    print(f"\nActivation: {activation}")
    print("Accuracy:", accuracy_score(y_test, y_pred))
    print(classification_report(y_test, y_pred,
target_names=encoder.classes_))
```

```
    return y_pred

for act in ['tanh','logistic','identity']:
    train_mlp(X_train, y_train, X_test, y_test, act)
```

Tanh using the following function:

$$f(z) = tanh(z) = \frac{2}{1 + e^{-2z}} - 1$$

This function zero centered meaning outputs are ranging from -1 to 1. This helps deal with bias in the subsequent layers as well as help with the vanishing gradient problem better than sigmoid since the derivatives are larger near 0.

Sigmoid using the following function:

$$f(z) = \frac{1}{(1 + e^{-z})}$$

The outputs for this function ranges from 0 to 1. It S shaped and can handle non-linear decision boundaries.

Linear uses the following function:

$$f(z) = z$$

This is because z is already a linear combination of the weights, resulting the activation function just being an identity of itself. This function can only produce linear boundaries, meaning it may have poor performance in heavily overlapping or complex datasets.

The following are the results:

```
ReLU
Accuracy: 0.9111111111111111
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.82      0.93      0.88        15
   virginica       0.92      0.80      0.86        15

    accuracy                           0.91        45
   macro avg       0.92      0.91      0.91        45
weighted avg       0.92      0.91      0.91        45


Tanh
Accuracy: 0.9555555555555556
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.93      0.93      0.93        15
   virginica       0.93      0.93      0.93        15

    accuracy                           0.96        45
   macro avg       0.96      0.96      0.96        45
weighted avg       0.96      0.96      0.96        45


Sigmoid
Accuracy: 0.9333333333333333
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.88      0.93      0.90        15
   virginica       0.93      0.87      0.90        15

    accuracy                           0.93        45
   macro avg       0.93      0.93      0.93        45
weighted avg       0.93      0.93      0.93        45


Linear (Identity)
Accuracy: 0.9333333333333333
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.88      0.93      0.90        15
   virginica       0.93      0.87      0.90        15

    accuracy                           0.93        45
   macro avg       0.93      0.93      0.93        45
weighted avg       0.93      0.93      0.93        45
```

As seen, sigmoid and linear seem to have the similar performances overall (93.3% vs 93.3%) indicating that the data can be fairly separable using linear decision boundaries. The sigmoid had worse performance compared to the tanh function (93.3% vs 95.6, this can possibly attributed to the vanishing gradient problem, where the sigmoid function's become too small to make a difference while the tanh function's gradients are still making a difference. Thus, tanh is able learn more and have a better result. ReLu had the worst performance (91.1%), and this can be attributed to some neurons effectively dying due to 0 output. As a result, for a small number of hidden layers and with only 10 neurons, this can result in a lot of processing and outputs being lost in future layers. As a result, it produces the worst decision boundary.

All 4 models were able to classify setosa 100% correctly, indicating that setosa data points are completely separated from the other data points. The only trouble in classifcation happened in classifying between the versicolor and the viriginca classes. Most labels seem to have high precision on virginica and low recall with this being vice versa for versicolor. This might seem to indicate that these models are labeling more points as versicolor than viriginica, indicating heavy overlap between these classes with more bias towards versicolor.

Comparison to K-Means Clustering

This model was then compared to K-means clustering. K-means clustering is an unsupervised model in which clusters are determine solely based on distance from centroid with no input from the labels. Additionally, no weights are placed upon features, meaning that clusters are determined solely by raw distance and labels are assigned to each cluster. The goal of the algorithm is to minimize the distances of the between the data points and the center of their assigned clusters.

K-means clustering was implemented for this comparison using scikit-learn as well.

The following is the code used for K-means clustering:

```
# Load data
data = pd.read_csv("irisdata.csv")

X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Encode class labels
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y)

# Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Apply K-Means (3 clusters since 3 classes)
```

```
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
kmeans.fit(X_scaled)

# K-Means cluster assignments
clusters = kmeans.labels_

# Convert cluster labels → true class labels using majority vote
mapped_labels = np.zeros_like(clusters)

for cluster in range(3):
    mask = clusters == cluster
    mapped_labels[mask] = mode(y_encoded[mask], keepdims=True)[0]

# Evaluation
print("\nK-Means Clustering Results")
print("Accuracy:", accuracy_score(y_encoded, mapped_labels))
print(classification_report(y_encoded, mapped_labels,
target_names=encoder.classes_))
```

The true labels were encoded and the features were all scaled like before. K means were applied to three clusters since we know that 3 classes were given. To label these clusters, a majority vote system was implemented where an array containing all the datapoint's cluster assignments were created, and for each cluster, the label that appeared the most for those data points will be the label for that cluster.

The following are the results:

```
K-Means Clustering Results
Accuracy: 0.8333333333333334
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.74      0.78      0.76        50
   virginica       0.77      0.72      0.74        50

    accuracy                           0.83       150
   macro avg       0.83      0.83      0.83       150
weighted avg       0.83      0.83      0.83       150
```

The algorithm had a worse performance overall compared to any form MLP Classifier with an overall accuracy of 83.3% compared to the worst activation functions in MLP Classifier being 91.1%. K-means clustering had a low precision and recall on versicolor and virginica indicating that it had trouble separating these classes out while setosa had 100% accuracy.

This can be explained with how K-means will have a hard time separating labels out if classes end up overlapping. When this happens, data from one class will be attributed to another due to those data points simply being closer to the centroid of the other class. This results in many data points where classes are overlapped being misattributed, and thus, resulting in the low accuracy of the model.

MLP Classifier has an advantage as all its models are supervised to learn true class boundaries to which it can produce a more accurate boundary line compared to K-means clustering.

Sources/Tutorials used:

For MLP:
https://www.geeksforgeeks.org/machine-learning/classification-using-sklearn-multi-layer-perceptron/

For K-means:
https://www.datacamp.com/tutorial/k-means-clustering-python