ECE 449 - Intelligent Systems Engineering

Lab 1-D41: Neural Networks, Perceptrons and Hyperparameters

Lab date: *Thursday, September 16, 2021 -- 2:00 - 4:50 PM*
Room: *ETLC E5-013*
Lab report due: *Wednesday, September 29, 2021 -- 3:50 PM*

1. Objective:

The objective of this lab is to gain familiarity with the concepts of linear models and to gain a feeling for how changing hyperparameters affects the performance of the model. The exercises in the lab will help bring to light the weaknesses and strengths of linear models and how to work with them.

2. Expectation:

Complete the pre-lab, and hand it in before the lab starts. A formal lab report is required for this lab, which will be the completed version of this notebook. There is a marking guide at the end of the lab manual. If figures are required, label all the axies and provide a legend when appropriate. An abstract, introduction, and conclusion are required as well, for which cells are provided at the end of the notebook. The abstract should be a brief description of the topic, the introduction a description of the goals of the lab, and the conclusion a summary of what you learned, what you found difficult, and your own ideas and observations.

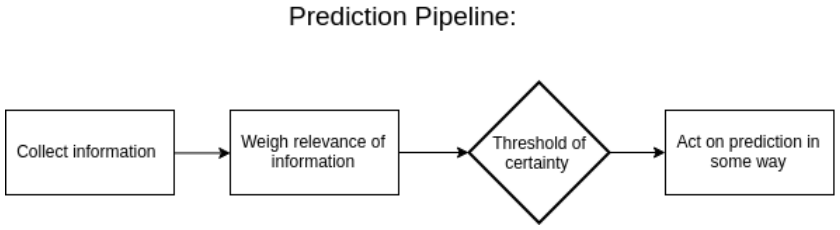
3. Pre lab:

1. Read through the code. What kind of models will be used in this lab?
2. Explain why the differentiability of an activation function plays an important role in the learning of these neural networks. Why might the linear activation function be a poor choice in some cases?

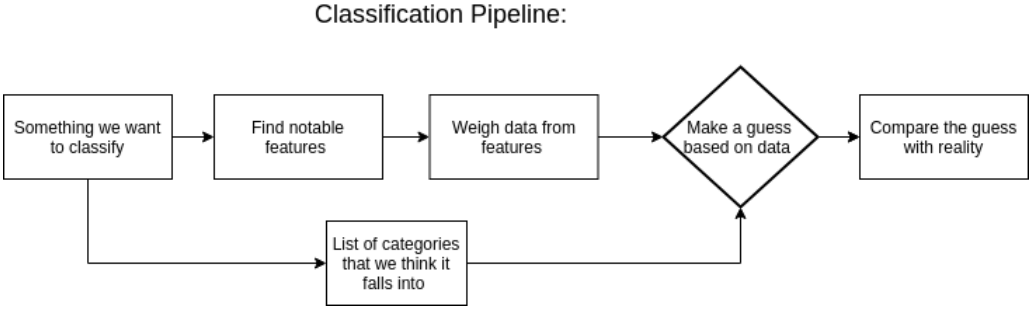
4. Introduction:

During this lab, you will be performing a mix of 2 common machine learning tasks: regression and classification. Before defining these tasks mathematically, it is important to understand the core process behind the two tasks. Regression is defined as reasoning backwards. In the context of machine learning, regression is about predicting the future based on the past. Classification is defined as the act of arranging things based on their properties. These definitions give insight into how these problems are broken down.

Suppose you, a human being, want to make a prediction. What are the steps that you take: You first collect data on the subject that you want to predict. Then you weigh the relevance of each piece of information that you get, attributing varying levels of importance to each piece of data. Once you have enough relevant data, you become certain of an outcome. Finally, you act on that certainty. This pipeline is shown in the figure below:



Now the classification task, one usually begins this with a topic that they want to classify. This is usually accompanied by a list of candidate categories, one of which is the correct category for the topic in question. Since classification relies on properties of the topic, the next step is to list the notable features that may help in the discerning the correct category. Similarly to the prediction, the relevance of each piece of information is then weighed and a decision is made when you have enough data. Once this is done, the guess is compared to reality in order to judge if the classification was correct. This pipeline is shown in the figure below:



The mathematical model that we will use in this lab to describe such behaviors are called linear models. The simplest linear model is the perceptron.

A *perceptron* is a simple type of neural network that uses *supervised learning*, where the expected values, or *targets*, are provided to the network in addition to the inputs. This simple network operates by first calculating the weighted sum of its inputs. These weights are typically randomly assigned. Then, the sum is processed with an *activation function* that "squashes" the summed total to a smaller range, such as (0, 1). A *bias*, externally applied, can increase or lower the output of the weighted sum.

The perceptron's functioning is inspired on human's learning model. It takes in input data in the form of the x vector. It then weighs the relevance of each input using the multiplication of x by their respective weight w. Following this, the total sum of all weighted inputs or transmitted signal is passed through a link, which can be whether a synaptic link or an activation link. A synaptic link keeps a linear input-output relationship, while an activation link is determined by the before mentioned *activation function*. This function keeps a nonlinear input-output relationship. The output value is y, which is effectively the action that you take based on your prediction.

The math behind the perceptron's operations is described by the following formulae:

$$\mathbf{v}_m = \sum_{i=1}^{n} \mathbf{w}_{mi}\mathbf{x}_i + b_m$$
$$\mathbf{y}_m = \phi(\mathbf{v}_m)$$

$\mathbf{v}_m$ - activation potential
$\mathbf{x}_i$ - input vector
$\mathbf{w}$ - weight vector
b - bias term
$\phi(\ )$ - activation function
$\mathbf{y}_m$ - output of neuron m

Training a perceptron involves calculating the error by taking the difference between the target or the desired output value **d** and the actual output **y**. This allows it to determine how to update its weights such that a closer output value to the desired output value is obtained. Perceptrons are commonly employed to solve two-class classification problems where the classes are linearly separable. However, the applications for this are evidently very limited. Therefore, a more practical extension of the perceptron is the *multi-layer perceptron (MLP)*, which adds extra hidden layer(s) between the inputs and outputs to yield more flexibility in what the MLP can classify.
The most common learning algorithm used is *backpropagation (BP)*. It employs gradient descent in an attempt to minimize the squared error between the network outputs and the targets.

$$E(k) = \frac{1}{2}\sum_{k=1}^{K}\sum_{m}[d_m(k) - y_m(k)]^2$$

$E(k)$ - Error over the training sample
K - number of samples in the training set
$d(k)_m$ - desired output value of neuron m
$y(k)_m$ - output of neuron m at iteration or epoch k

This error value is propagated backwards through the network, and small changes are made to the weights in each layer. Using a gradient descent approach, the weights in the network are updated as follows:

$$\Delta w_{mi}(k) = -\alpha\frac{\partial E(k)}{\partial w_{mi}(k)}$$

where $\alpha > 0$ is the *learning rate*.

The network is trained using the same data, multiple times in *epochs*. Typically, this continues until the network has reached a convergence point that is defined by the user through a *tolerance* value. For the case of this lab, the tolerance value is ignored and training will continue until the specified number of epochs is reached. More details of backpropagation can be found in the lecture notes.

Neural networks have two types of parameters that affect the performance of the network, parameters and hyperparameters. Parameters have to do with the characteristics that the model learns during the training process. Hyperparameters are values that are set before training begins. The parameters of linear models are the weights. The hyperparameters include:

- Learning algorithm
- Loss function
- Learning rate
- Activation function

Hyperparameter selection is very important in the field of AI in general. The performance of the learning systems that are deployed relies hevily on the selection of hyperparameters and some advances in the field have even been soley due to changes in hyperparametes. More on hyperparameters can be found in the literature.

5. Experimental Procedure:

Exercise 1: Perceptrons and their limitations

The objective of this exercise is to show how adding depth to the network makes it learn better. This exercise will involve running the following cells and examining the data. This exercise will showcase the classification task and it will be performed on the Iris dataset. Also, ensure that all files within "Lab 1 Resources" is placed in the same directory as this Jupyter notebook.
Run the following cell to import all the required libraries.

In [1]:
```
%matplotlib inline

import numpy as np                                # General math operations
import scipy.io as sio                            # Loads .mat variables
import matplotlib.pyplot as plt                   # Data visualization
from sklearn.linear_model import Perceptron       # Perceptron toolbox
from sklearn.neural_network import MLPRegressor   # MLP toolbox
import seaborn as sns
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
from sklearn import preprocessing
from sklearn import linear_model                # Linear models
from sklearn.tree import DecisionTreeRegressor

import warnings
warnings.filterwarnings('ignore')
```

The Iris dataset: This dataset contains data points on three different species of Iris, a type of flower. The dataset has 50 entries for each of the species and has 4 different features:

1. Sepal Length
2. Sepal Width
3. Petal Length
4. Petal Width

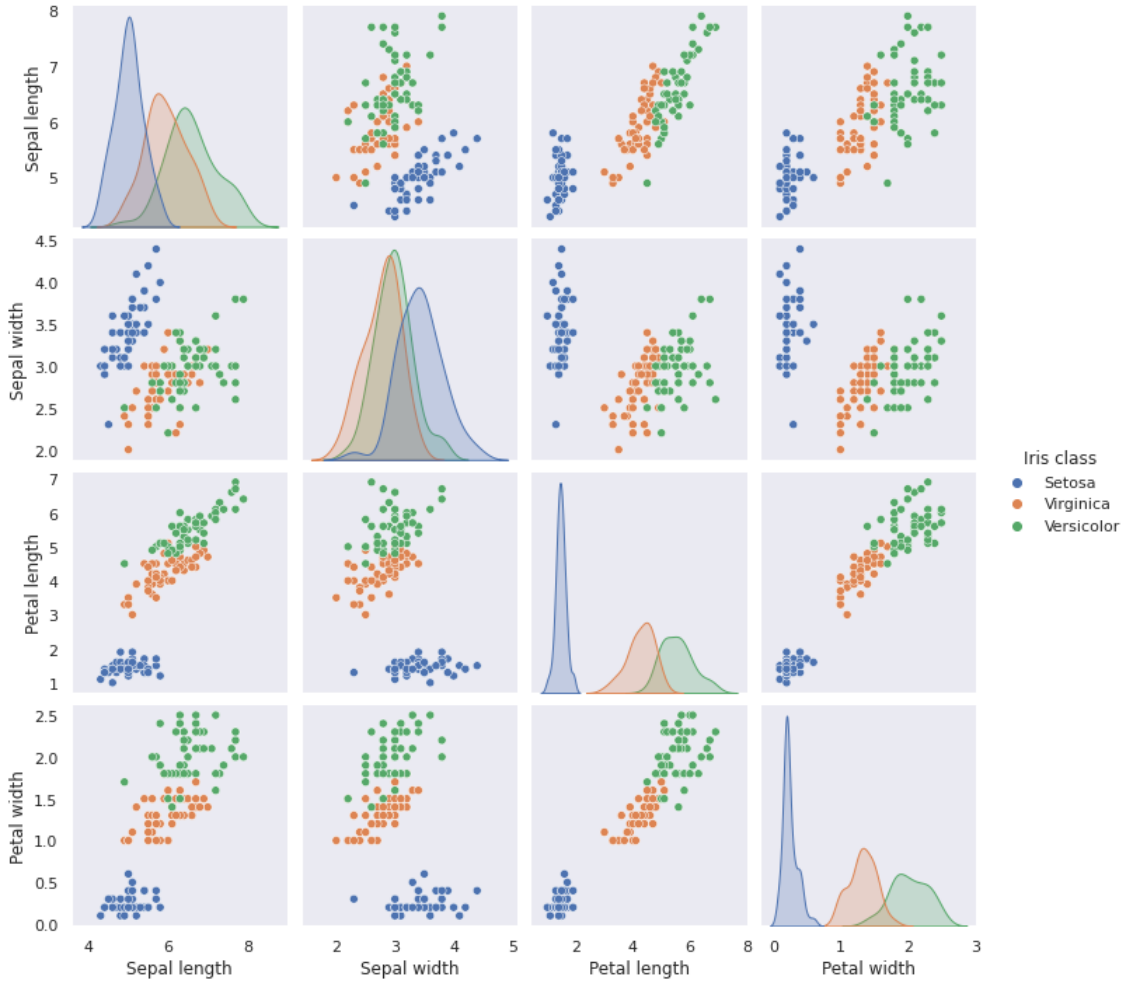
This dataset has one obvious class that is separate from a cluster of the other two classes, making it a typical exercise in classification for machine learning. The next cell loads the dataset into 2 variables, one for the features and one for the classes.

In [4]:
```
# load the data
iris = datasets.load_iris()
Y = iris.target
X = iris.data

# set up the pandas dataframes
X_df = pd.DataFrame(X, columns = ['Sepal length','Sepal width', 'Petal length', 'Petal width'] )
Y_df = pd.DataFrame(Y, columns = ['Iris class'])

# this code changes the class labels from numerical values to strings
Y_df = Y_df.replace({
0:'Setosa',
1:'Virginica',
2:'Versicolor'
})
#Joins the two dataframes into a single data frame for ease of use
Z_df = X_df.join(Y_df)
```

Visualizing the data is an important tool for data exploration. Visualizing the data will allow you to intuitively understand obvious relationships that are present in the data, even before you begin to analyse it. The next cell will plot all of the features against each other.

In [5]:

```
# show the data using seaborn
sns.set(style='dark', palette= 'deep')
pair = sns.pairplot(Z_df, hue = 'Iris class')
plt.show()
```



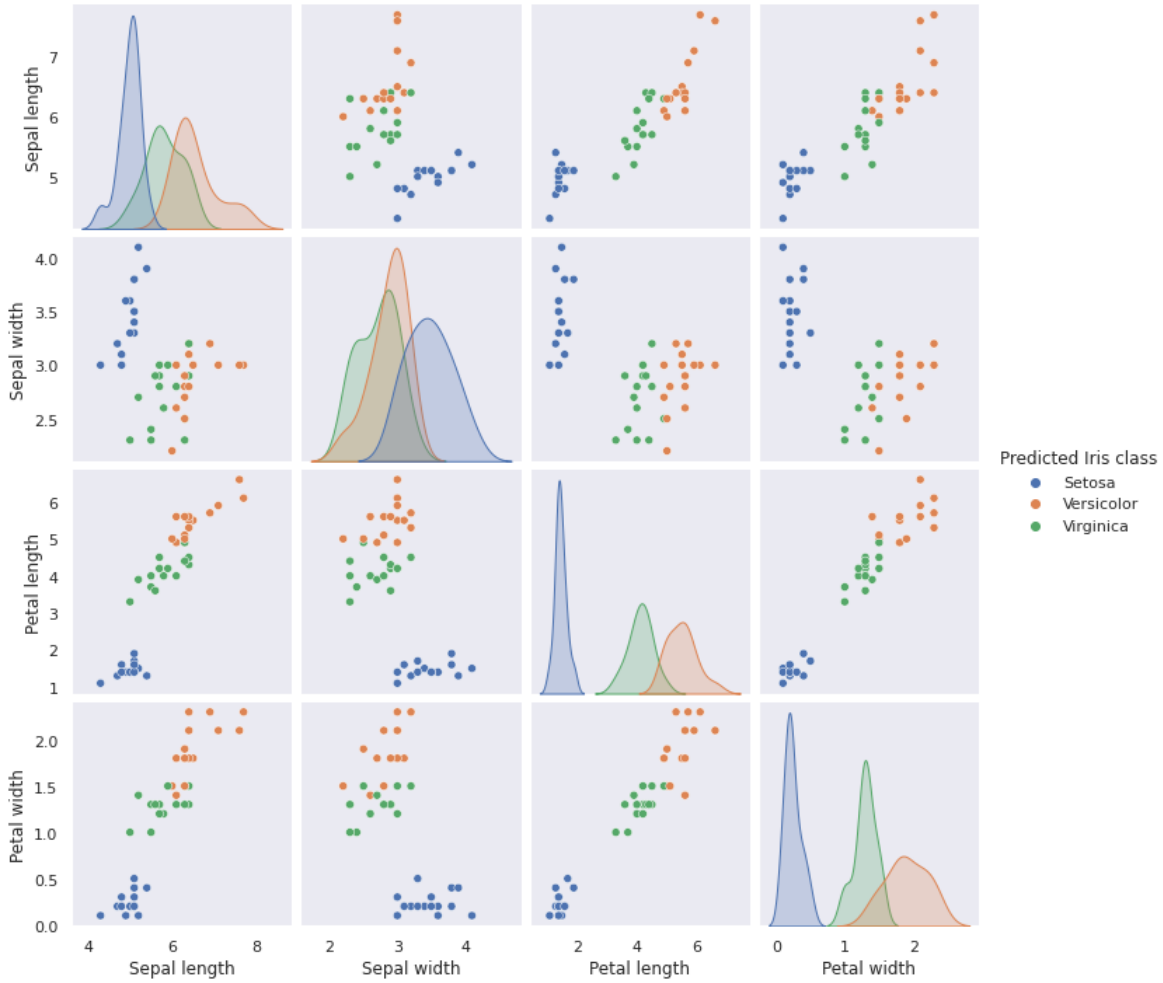
This type of plot is called a pairplot. It plots each feature against all other features including itself; this is done for all four features. This results in 2 different types of plots being present in the plot, scatter and histogram.

The following cell will train a perceptron on the features and labels and display the result on the test set in a pairplot.

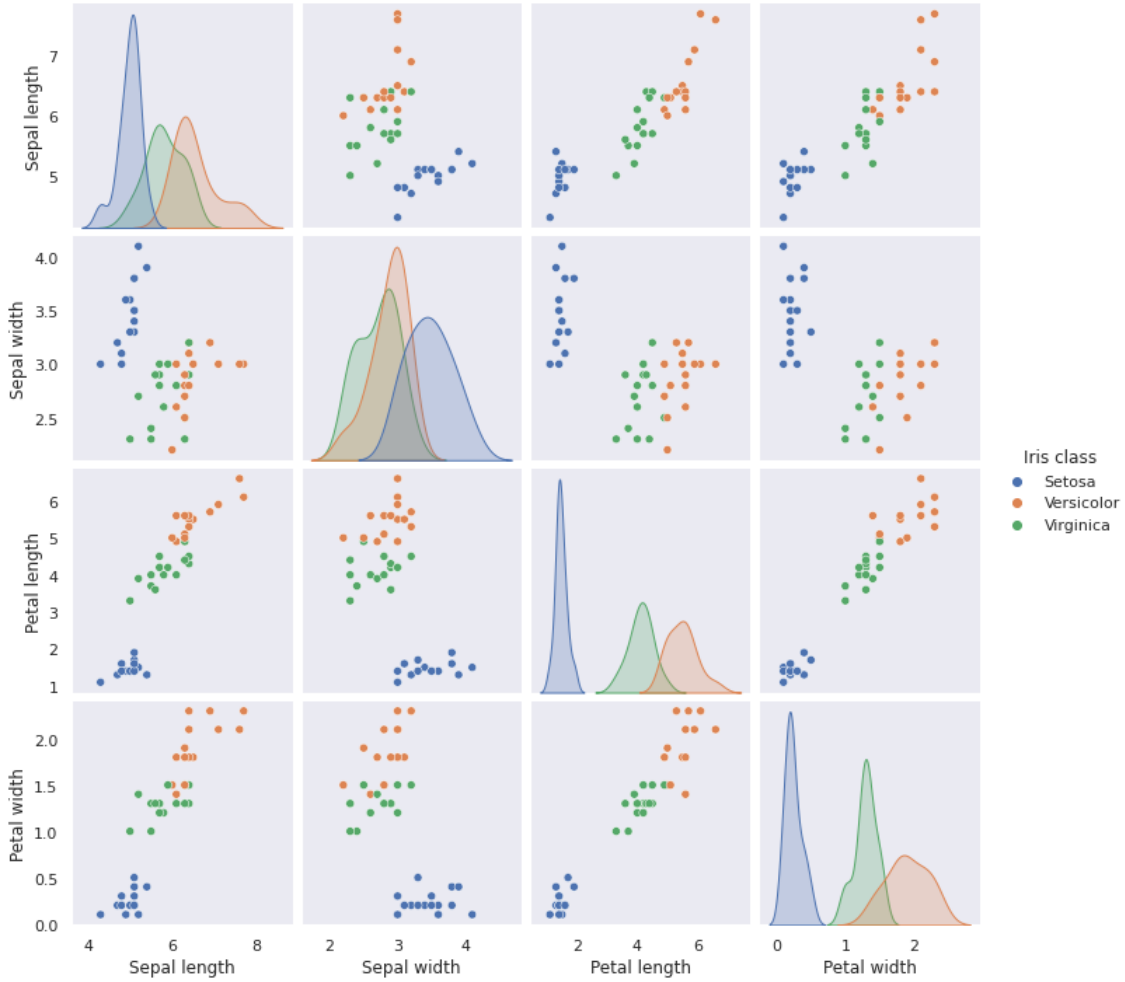
In [13]:

```
RANDOM_SEED = 6
xTrain, xTest, yTrain, yTest = train_test_split(X_df, Y_df, test_size =0.3,\
                                                random_state=RANDOM_SEED)
#plot the testing data
test_df = xTest.join(yTest)

# print(test_df.head)
# perceptron training
percep = Perceptron(max_iter = 1000)
percep.fit(xTrain, yTrain)
prediction = percep.predict(xTest)

print(percep.score(xTest,yTest))
# display the classifiers performance
prediction_df = pd.DataFrame(prediction, columns=['Predicted Iris class'], index = test_df.index)
# print(prediction_df.head)

prediction_df_index_df = prediction_df.join(xTest)
# print(prediction_df_index_df.head)

pair = sns.pairplot(prediction_df_index_df, hue = 'Predicted Iris class')
#pair_test = sns.pairplot(test_df, hue ='Iris class')
plt.show()
```

```
1.0
```



In [12]:
```
pair_test = sns.pairplot(test_df, hue ='Iris class') #test data from the dataset
plt.show()
```



Question 1:

Comment on the performance of the perceptron, how well does it handle the task?

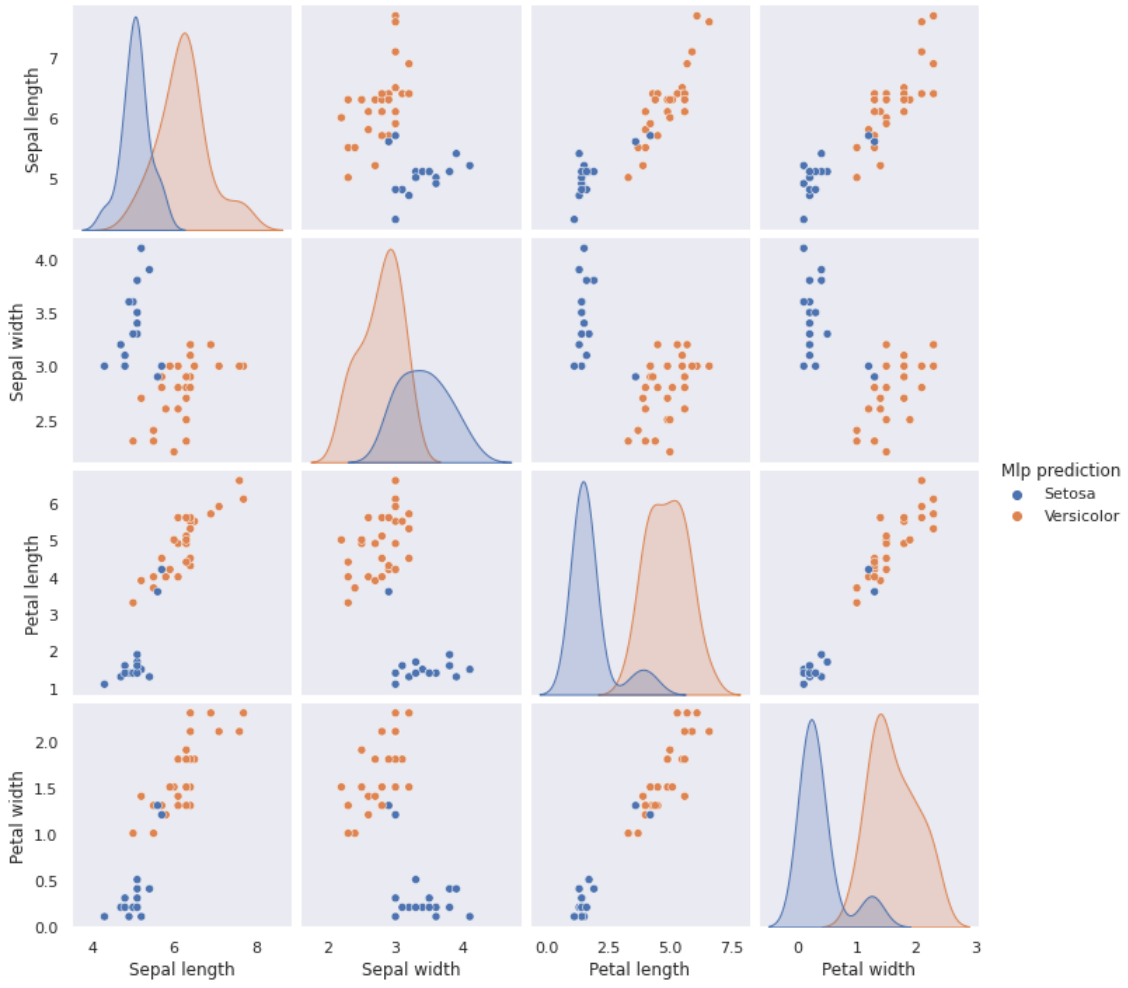
The perceptron performs well in this classification task as we can see from the pairplot graphs (the Predicted Iris class graph and the Test Iris class graph) they have performed exactly the same classifications. Moreover, if we compare the predicted classified data against the target values using the line "print(percep.score(xTest, yTest))", we obtain a score of 1.0 (100%) indicating that the perceptron's predicted classified data is the same as the target values (yTest).

The next cell will retrain the perceptron but with different parameters. This MLP consists of 2 hidden layers: one with 8 neurons and a second one with 3

In [19]:

```
# change the layers, retrain the mlp
cls = MLPClassifier(solver = 'sgd' ,activation = 'relu' ,  \
                    hidden_layer_sizes = (8,3,), max_iter = 100000)
# Overfitting
for i in range(0,5):
    cls.fit(xTrain, yTrain)

mlp_z = cls.predict(xTest)

mlp_z.reshape(-1,1)

cls_df = pd.DataFrame(mlp_z, columns = ["Mlp prediction"], index=xTest.index)

# cls_df_index = cls_df.join(Test_index_df).set_index('Test index')
# cls_df_index.index.name = None

# Join with the test_index frame
cls_prediction_df = cls_df.join(xTest)



# Display the MLP classifier
cls_pairplot = sns.pairplot(cls_prediction_df, hue = 'Mlp prediction')

print(cls.score(xTest, yTest))
```

```
0.6666666666666666
```



Question 2:

Answer the following questions:

- **How does the Mlp compare to the perceptron in the classification task?**
- **Did it do well?**
- **Was it able to classify the three classes?**
- **What happens when you run it again?**
- **Can you offer a explanation for what happened?**

Fill the box below with your answer:

```
The MLP does not perform as well as the perceptron in this classification task. The perceptron had an accuracy of
100% when evaluating the score, however the MLP upon consecutive runs had different accuracies each time. The MLP
had a score ranging from 33% to 98%, thus it did not do well. At times, it was able to classify 3 classes and during
other runs it was only able to classify all of the data as one class. When running it again, we notice that it's
accuracy fluctuates and the generated pairplot graph shows that the MLP classifies either all three, only two or one
iris type. An explanation to this random behavior stems from the weight and biases initialization. Upon successive
function calls, the weight and bias initialization changes each time, this can be stopped by using the random_state
parameter within the MLPClassifier class initialization.
```

Exercise 2: Getting your hands dirty with regression

NOTE: The code in this exercise is computationally intensive and may require up to 5 minutes to finish running.

In order to improve the energy management of monitoring stations, a team of climatologists would like to implement a more robust strategy of predicting the solar energy available for the next day, based on the current day's atmospheric pressure. They plan to test this with a station situated in Moxee, and are designing a multi-layer perceptron that will be trained with the previous year's worth of Moxee data. They have access to the following values:

- **Inputs:** Pressure values for each hour, along with the absolute differences between them
- **Targets:** Recorded solar energy for the day after

The individual who was in charge of this project before had created a traditional machine learning approach to predict the solar energy availiabilty of the next day. The individual recently retired and you have been brought on to the team to try to implement a more accurate system. You find some code that was left over that uses a MLP. The MLP is initially formed using one hidden layer of 50 neurons, a logistic sigmoid activation function, and a total of 500 iterations. Once it is trained, the MLP is used to predict the results of both the training cases and new test cases. As a measure of accuracy, the root mean square error (RMSE) is displayed after inputting data to the MLP.

First, read through the code to understand what results it produces, and then run the script.

Question 1:

Your objective is to play with the parameters of the regressor to see if you can beat the decision tree. There are parameters that you can change to try to beat it. You can change:

- Size of the Hidden Layers: **between 1 and 50**
- Activation Function:
  - **Identity**
  - **Logistic**
  - **tanh**
  - **relu**
- Number of Iterations, to different values (both lower and higher): **Between 1 and 1000**

Comment on how this affects the results. Include plots of your final results (using any one of your values for the parameters). Describe some of the tradeoffs associated with both lowering and raising the number of iterations.

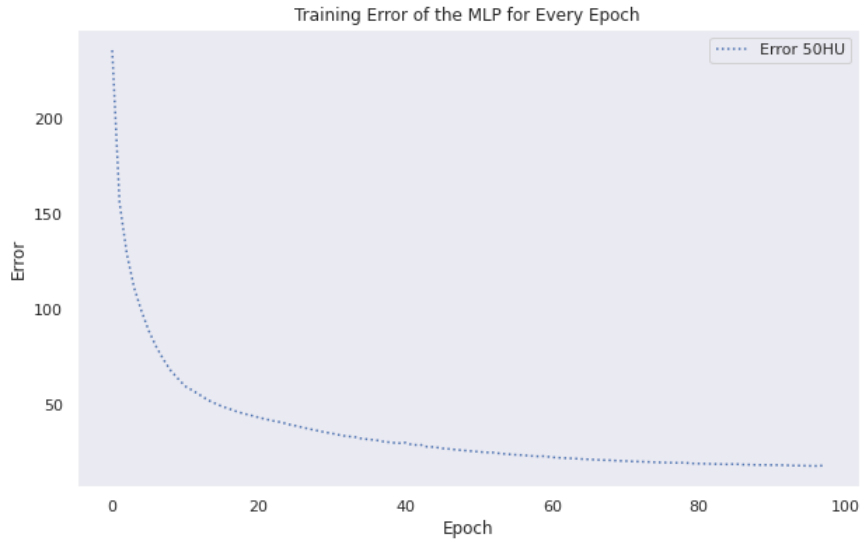
In order to determine the accuracy of the methods, you will be using RMSE

$$RMSE = \sqrt{\frac{\sum (Approximated - observed)}{n}}$$

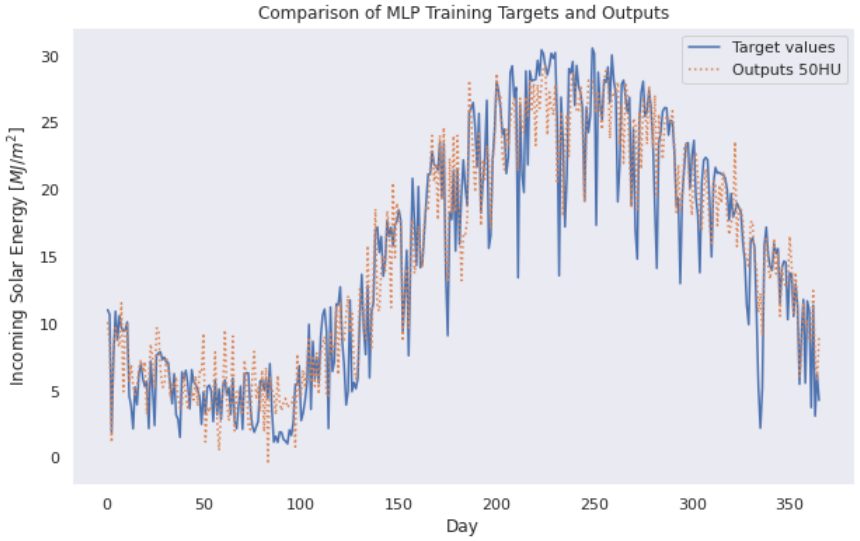
In [31]:

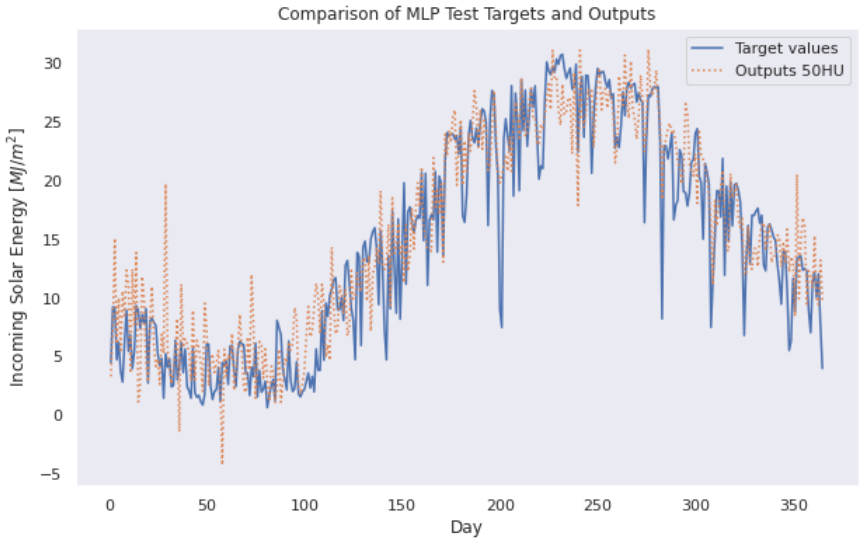
```
# Obtain training data
moxeeData = sio.loadmat('moxeetrainingdata.mat')    # Load variables from the Moxee dataset
trainingInputs = moxeeData['pressureData']          # Pressure values and differences for every hour in a year
trainingTargets = moxeeData['dataEstimate']         # Estimate of incoming solar energy based on observed data

# Preprocess the training inputs and targets
iScaler = preprocessing.StandardScaler()    # Scaler that removes the mean and scales to unit variance
scaledTrainingInputs = iScaler.fit_transform(trainingInputs)   # Fit and scale the training inputs

tScaler = preprocessing.StandardScaler()
scaledTrainingTargets = tScaler.fit_transform(trainingTargets)

# Create the multilayer perceptron.
# This is where you will be modifying the regressor to try to beat the decision tree
mlp = MLPRegressor(
    hidden_layer_sizes = (5,),     # One hidden layer with 50 neurons
    activation = 'relu',        # Logistic sigmoid activation function
    solver = 'sgd',                # Gradient descent
    learning_rate_init = 0.01 ,# Initial learning rate
    )
#
################################################################## Create the decision tree:
dt_reg = DecisionTreeRegressor(criterion='mse', max_depth = 10)
dt_reg.fit(scaledTrainingInputs, scaledTrainingTargets)


### MODIFY THE VALUE BELOW ###
noIterations = 98  # Number of iterations (epochs) for which the MLP trains
### MODIFY THE VALUE ABOVE ###

trainingError = np.zeros(noIterations)  # Initialize array to hold training error values

# Train the MLP for the specified number of iterations
for i in range(noIterations):
    mlp.partial_fit(scaledTrainingInputs, np.ravel(scaledTrainingTargets))  # Partial fit is used to obtain the ou
    currentOutputs = mlp.predict(scaledTrainingInputs)  # Obtain the outputs for the current MLP using the trainin
    trainingError[i] = np.sum((scaledTrainingTargets.T - currentOutputs) ** 2) / 2  # Keep track of the error thro

# Plot the error curve
plt.figure(figsize=(10,6))
ErrorHandle ,= plt.plot(range(noIterations), trainingError, label = 'Error 50HU',  linestyle = 'dotted')
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Training Error of the MLP for Every Epoch')
plt.legend(handles = [ErrorHandle])
plt.show()

# Obtain test data
testdataset = sio.loadmat('moxeetestdata.mat')
testInputs = testdataset['testInputs']
testTargets = testdataset['testTargets']
scaledTestInputs = iScaler.transform(testInputs)  # Scale the test inputs

# Predict incoming solar energy from the training data and the test cases
scaledTrainingOutputs = mlp.predict(scaledTrainingInputs)
scaledTestOutputs = mlp.predict(scaledTestInputs)

######################################################################## Predict using the bad guy:
scaledTreeTrainingOutputs = dt_reg.predict(scaledTrainingInputs)
scaledTreeTestOutputs = dt_reg.predict(scaledTestInputs)

# Transform the outputs back to the original values
trainingOutputs = tScaler.inverse_transform(scaledTrainingOutputs)
testOutputs = tScaler.inverse_transform(scaledTestOutputs)
## DT outputs
treeTrainingOutputs = tScaler.inverse_transform(scaledTreeTrainingOutputs) # -- transform the tree back to real da
treeTestingOutputs = tScaler.inverse_transform(scaledTreeTestOutputs)

# Calculate and display training and test root mean square error (RMSE)
trainingRMSE = np.sqrt(np.sum((trainingOutputs - trainingTargets[:, 0]) ** 2) / len(trainingOutputs)) / 1000000  #
testRMSE = np.sqrt(np.sum((testOutputs - testTargets[:, 0]) ** 2) / len(testOutputs)) / 1000000

## need to add this for the decision tree
trainingTreeRMSE = np.sqrt(np.sum((treeTrainingOutputs - trainingTargets[:, 0]) ** 2) / len(trainingOutputs)) / 10
testTreeRMSE = np.sqrt(np.sum((treeTestingOutputs - testTargets[:, 0]) ** 2) / len(testOutputs)) / 1000000

print("Training RMSE:", trainingRMSE, "MJ/m^2")
print("Test RMSE:", testRMSE, "MJ/m^2")
######################################################################## Print the tree RMSE:
print("Decision Tree training RMSE:", trainingTreeRMSE, 'MJ/m^2')
print("Decision Tree Test RMSE:", testTreeRMSE, 'MJ/m^2')
day = np.array(range(1, len(testTargets) + 1))

# Plot training targets vs. training outputs
plt.figure(figsize=(10,6))
trainingTargetHandle ,= plt.plot(day, trainingTargets / 1000000, label = 'Target values')
trainingOutputHandle ,= plt.plot(day, trainingOutputs / 1000000, label = 'Outputs 50HU',  linestyle = 'dotted')
```

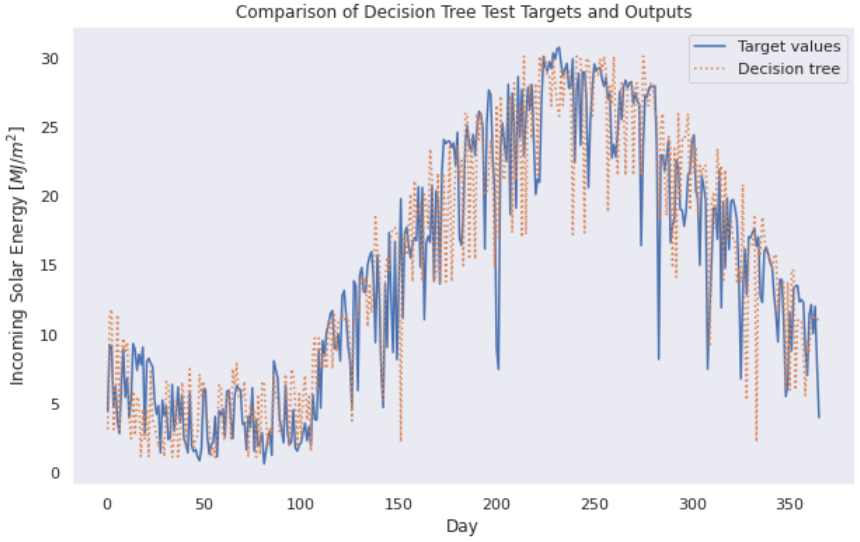
```
plt.xlabel('Day')
plt.ylabel(r'Incoming Solar Energy [$MJ / m^2$]')
plt.title('Comparison of MLP Training Targets and Outputs')
plt.legend(handles = [trainingTargetHandle, trainingOutputHandle])
plt.show()

# Plot test targets vs. test outputs -- student
plt.figure(figsize=(10,6))
testTargetHandle ,= plt.plot(day, testTargets / 1000000, label = 'Target values')
testOutputHandle ,= plt.plot(day, testOutputs / 1000000, label = 'Outputs 50HU',  linestyle = 'dotted')
plt.xlabel('Day')
plt.ylabel(r'Incoming Solar Energy [$MJ / m^2$]')
plt.title('Comparison of MLP Test Targets and Outputs')
plt.legend(handles = [testTargetHandle, testOutputHandle])
plt.show()

########################################################################## Plot the tree regressor vs. test outputs
plt.figure(figsize=(10,6))
testTreeTargetHandle, = plt.plot(day, testTargets / 1000000, label = 'Target values')
testTreeOutputHandle, = plt.plot(day, treeTestingOutputs / 1000000, label = 'Decision tree', linestyle = 'dotted')
plt.xlabel('Day')
plt.ylabel(r'Incoming Solar Energy [$MJ / m^2$]')
plt.title('Comparison of Decision Tree Test Targets and Outputs')
plt.legend(handles = [testTreeTargetHandle, testTreeOutputHandle])
plt.show()
```




```
Training RMSE: 2.80897605244897 MJ/m^2
Test RMSE: 3.613221989548918 MJ/m^2
Decision Tree training RMSE: 0.18092630438000787 MJ/m^2
Decision Tree Test RMSE: 4.129187112937238 MJ/m^2
```


Comparison of MLP Training Targets and Outputs
Target values
Outputs 50HU
Incoming Solar Energy [MJ/m²]
Day


Comparison of MLP Test Targets and Outputs
Target values
Outputs 50HU
Incoming Solar Energy [MJ/m²]
Day


Comparison of Decision Tree Test Targets and Outputs
Target values
Decision tree
Incoming Solar Energy [MJ/m²]
Day

Fill the box below with your answer for question 1:

```
By changing the number of hidden layers to 5 and changing the activation function to relu, I was able to get a lower Test RMSE of 3.61 than that of the decision tree. By lowering the number of iterations to too low a value you risk underfitting and by highering the number of iterations to too high a value, you risk overfitting the model to the data.
```

Repeat the same process but against this SVM:

During a coffee break, you get talking to one of your friends from a different department. He mentioned that at one point there was an intern that was also tasked with predicting the solar energy and they tried a Support Vector machine.

When you tell your superiors, they suggest that you try to beat this interns work as well since it seems to work better than the decision tree that your predecessor left.

Question 2

Your objective again is to play with the parameters of the regressor to see if you can beat the Support Vector Machine. There are parameters that you can change to try to beat it. You can change:

- Size of the Hidden Layers: **between 1 and 50**
- Activation Function:
  - **Identity**
  - **Logistic**
  - **tanh**
  - **relu**
- Number of Iterations, to different values (both lower and higher): **Between 1 and 1000**

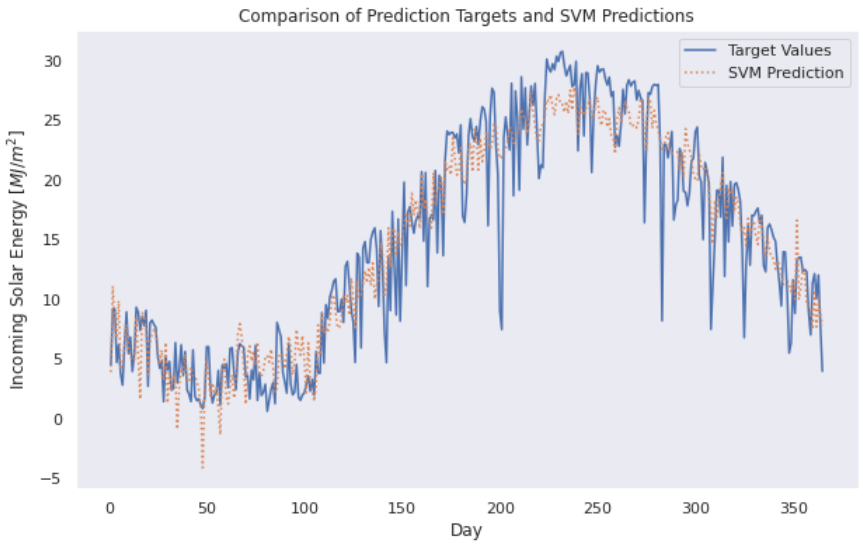
Comment on how this affects the results. Include plots of your final results (using any one of your values for the parameters). Describe some of the tradeoffs associated with both lowering and raising the number of iterations.

In order to determine the accuracy of the methods, you will be using the RMSE again.

$$RMSE = \sqrt{\frac{\sum (Approximated - observed)}{n}}$$

In [43]:

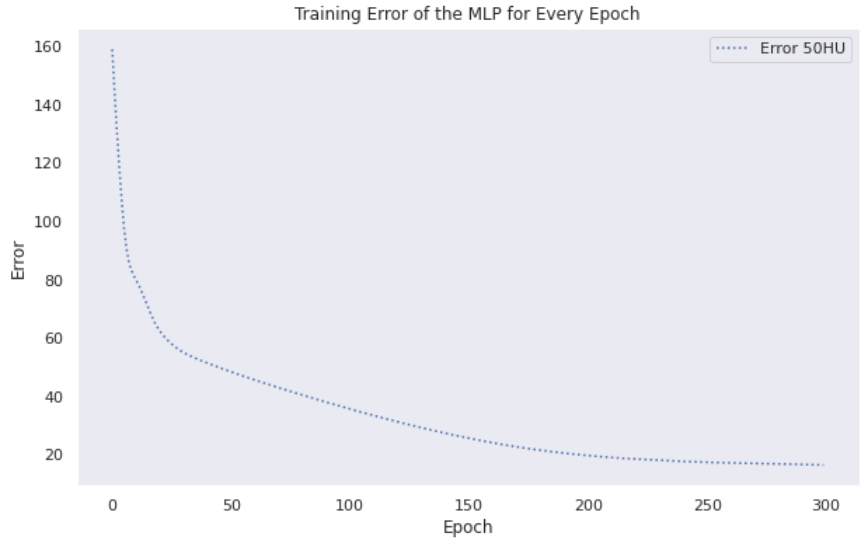
```
#INITIALIZE
from sklearn.svm import LinearSVR
svm_clf = LinearSVR(C=0.6, loss='squared_epsilon_insensitive')
svm_clf.fit(scaledTrainingInputs, np.ravel(scaledTrainingTargets))

# PREDICT the training outputs and the test outputs
scaledTrainingOutputs = svm_clf.predict(scaledTrainingInputs)
scaledTestOutputs = svm_clf.predict(scaledTestInputs)


trainingOutputs = tScaler.inverse_transform(scaledTrainingOutputs)
testOutputs = tScaler.inverse_transform(scaledTestOutputs)

 #Calculate and display training and test root mean square error (RMSE)
trainingsvmRMSE = np.sqrt(np.sum((trainingOutputs - trainingTargets[:, 0]) ** 2) / len(trainingOutputs)) / 1000000
testsvmRMSE = np.sqrt(np.sum((testOutputs - testTargets[:, 0]) ** 2) / len(testOutputs)) / 1000000

#### PLOTTING
plt.rcParams["figure.figsize"] = (10,6)
day = np.array(range(1, len(testTargets) + 1))

testTargetHandle, = plt.plot(day, testTargets / 1000000, label = 'Target Values')
testsvmOutputHandle, = plt.plot(day, testOutputs / 1000000, label = 'SVM Prediction', linestyle = 'dotted')
plt.xlabel('Day')
plt.ylabel(r'Incoming Solar Energy [$MJ / m^2$]')
plt.title('Comparison of Prediction Targets and SVM Predictions')
plt.legend(handles = [testTargetHandle, testsvmOutputHandle])
plt.show()

print("Support Vector Machine RMSE values and Plots")
print("Training RMSE:", trainingsvmRMSE, "MJ/m^2")
print("Test RMSE:", testsvmRMSE, "MJ/m^2")
```




```
Support Vector Machine RMSE values and Plots
Training RMSE: 2.9713640047460594 MJ/m^2
Test RMSE: 2.9944000422134693 MJ/m^2
```

In [79]:

```
# Modify this neural network
mlp = MLPRegressor(
    hidden_layer_sizes = (30,),     # One hidden layer with 50 neurons
    activation = 'logistic',        # Logistic sigmoid activation function
    solver = 'sgd',                 # Gradient descent
    learning_rate_init = 0.01 ,# Initial learning rate
    )

### MODIFY THE VALUE BELOW ###
noIterations = 300 # Number of iterations (epochs) for which the MLP trains
### MODIFY THE VALUE ABOVE ###

trainingError = np.zeros(noIterations)  # Initialize array to hold training error values

# Train the MLP for the specified number of iterations
for i in range(noIterations):
    mlp.partial_fit(scaledTrainingInputs, np.ravel(scaledTrainingTargets))  # Partial fit is used to obtain the ou
    currentOutputs = mlp.predict(scaledTrainingInputs)  # Obtain the outputs for the current MLP using the trainin
    trainingError[i] = np.sum((scaledTrainingTargets.T - currentOutputs) ** 2) / 2  # Keep track of the error thro

# Predict
scaledTrainingOutputs = mlp.predict(scaledTrainingInputs)
scaledTestOutputs = mlp.predict(scaledTestInputs)
#Training output conversion
trainingOutputs = tScaler.inverse_transform(scaledTrainingOutputs)
testOutputs = tScaler.inverse_transform(scaledTestOutputs)

#RMSE calculation
trainingRMSE = np.sqrt(np.sum((trainingOutputs - trainingTargets[:, 0]) ** 2) / len(trainingOutputs)) / 1000000  #
testRMSE = np.sqrt(np.sum((testOutputs - testTargets[:, 0]) ** 2) / len(testOutputs)) / 1000000

# Plot the error curve
plt.figure(figsize=(10,6))
ErrorHandle ,= plt.plot(range(noIterations), trainingError, label = 'Error 50HU',  linestyle = 'dotted')
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Training Error of the MLP for Every Epoch')
plt.legend(handles = [ErrorHandle])
plt.show()

print("MLP Training and test RMSE values:")
print("Training RMSE: " , trainingRMSE)
print("Test RMSE: " , testRMSE)
```




```
MLP Training and test RMSE values:
Training RMSE:  2.6696063316133274
Test RMSE:  2.927394251711981
```

Fill the box below with your answer for question 2:

For beating the Support Vector Machine, I used the same activation function, however I increased the number of hidden layers to 30 and the number of iterations to 300. The Support Vector Machine got a Test RMSE of 2.99 MJ/m^2 and my MLP Regressor was able to obtain a Test RMSE of 2.93 MJ/m^2. By lowering the number of iterations to too low a value you risk underfitting and by highering the number of iterations to too high a value, it becomes a case of diminishing returns.

Abstract

In this lab we performed two tasks, regression and classification. For classification, we explored the use of the perceptron neural network and the multi-layer perceptron neural network to classify 3 different iris species using four properties: Sepal Length, Sepal Width, Petal Length and Petal Width. The perceptron performed exceptionally well for this task, while the multi-layer perceptron had unpredictable behavior upon multiple function calls due to random bias and weight assignments. The regression task involved using pressure values per hour and absolute differeneces between two successive pressure values to predict the solar energy the day after. For this task we used a multi-layer perceptron and we compared it's performance in terms of root mean squared error values against the Support Vector Machine and the Decision Tree. By changing parameters of the multi-layer perceptron such as the number of hidden layers and the activation function we were able to obtain a test root mean squared error of 3.61 MJ/m^2 which was lower than that of the Decision Tree which was 4.12 MJ/m^2. Afterwards, by changing the number of hidden layers and the number of iterations, we were able to obtain a test root mean squared error of 2.93 MJ/m^2 which was lower than that of the Support Vector Machine which was 2.99 MJ/m^2.

Introduction

In this lab, we will be performing two tasks regression and classification. Classification involves the use of properties provided from data to categorize that data into group(s). In this task, we used the perceptron neural network and the multilayer perceptron neural network to classify different species of the iris flower using four different properties: Sepal Length, Sepal Width, Petal Length and Petal Width. The perceptron is an algorithm which involves the use of supervised learning where inputs and desired values are provided and it learns how to predict if future data belongs to a certain criteria or not (using its learning algorithm). The multi-layer perceptron is the same concept as the perceptron however it has hidden layer(s) and it uses backpropagation to assign it's weights and biases. Regression is the task of using past data to determine relationships between input and output variables (evaluating dependency). For this task, we used a multi-layer perceptron and compared it against a decision tree and a support vector machine.

Conclusion

For the classification task, the perceptron performed very well and had a accuracy of 100% when comparing the classified data to the test target values. On the other hand the multi-layer perceptron performed poorly on this task as upon consecutive function calls we would get different accuracies ranging from 33% to 98% and at times it wasnt able to classify more than one iris specie. The cause of this was the fact that every time we ran the code, the weights and bias' values that were assigned were done in a random manner (this can be stopped using a parameter that exists in the class called random_state). For the regression task, we were able to create a multi-layer perceptron model by increasing the number of hidden layers to 5 and the activation function to ReLu and that was able to obtain an test RMSE score of 3.61 MJ/m^2 which beat the decision tree which had a test RMSE score of 4.13 MJ/m^2. To beat the support vector machine which had a test RMSE score of 2.99 MJ/m^2 we increased the number of iterations to 300 and the number of hidden layers to 30 to obtain an test RMSE score of 2.93 MJ/m^2.

Lab 3 Marking Guide

Exercise	Item	Total Marks	Earned Marks
	Pre − lab	10	
	Abstract	3	
	Introduction	3	
	Conclusion	4	
1	*Classification*	35	
2	*Regression*	45	
	TOTAL	100	