

ECE 449 - Intelligent Systems Engineering

Lab 1-D42: Neural Networks, Perceptrons and Hyperparameters

Lab date: *Thursday, September 23, 2021 -- 2:00 - 4:50 PM*

Room: *ETLC E5-013*

Lab report due: *Wednesday, October 06, 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 axes 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:

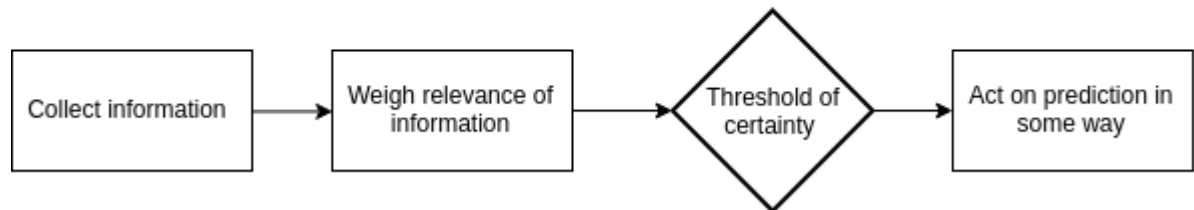
1. Read through the code. What kind of models will be used in this lab?
2. Explain why the differentiability of an activation function plays an important role in the learning of these neural networks. Why might the linear activation function be a poor choice in some cases?

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.

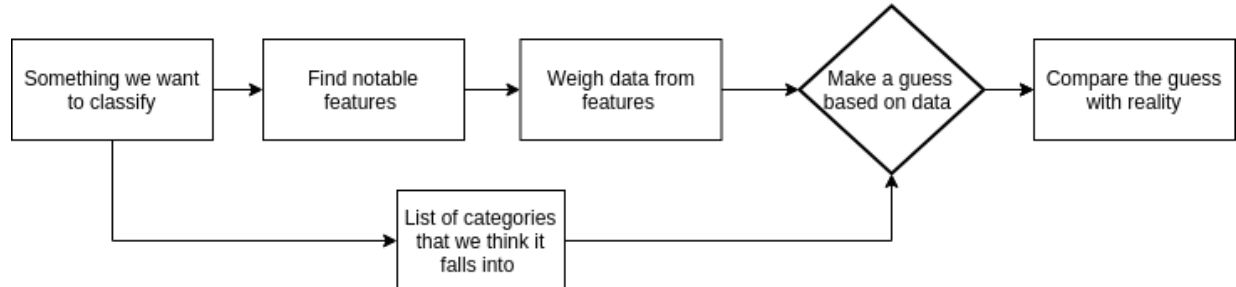
Suppose you, a human being, want to make a prediction. What are the steps that you take: You first collect data on the subject that you want to predict. Then you weigh the relevance of each piece of information that you get, attributing varying levels of importance to each piece of data. Once you have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

Prediction Pipeline:



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:

Classification Pipeline:



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 \mathbf{d} and the actual output \mathbf{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 heavily on the selection of hyperparameters and some advances in the field have even been solely due to changes in hyperparameters. 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 [2]: ▶ %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:

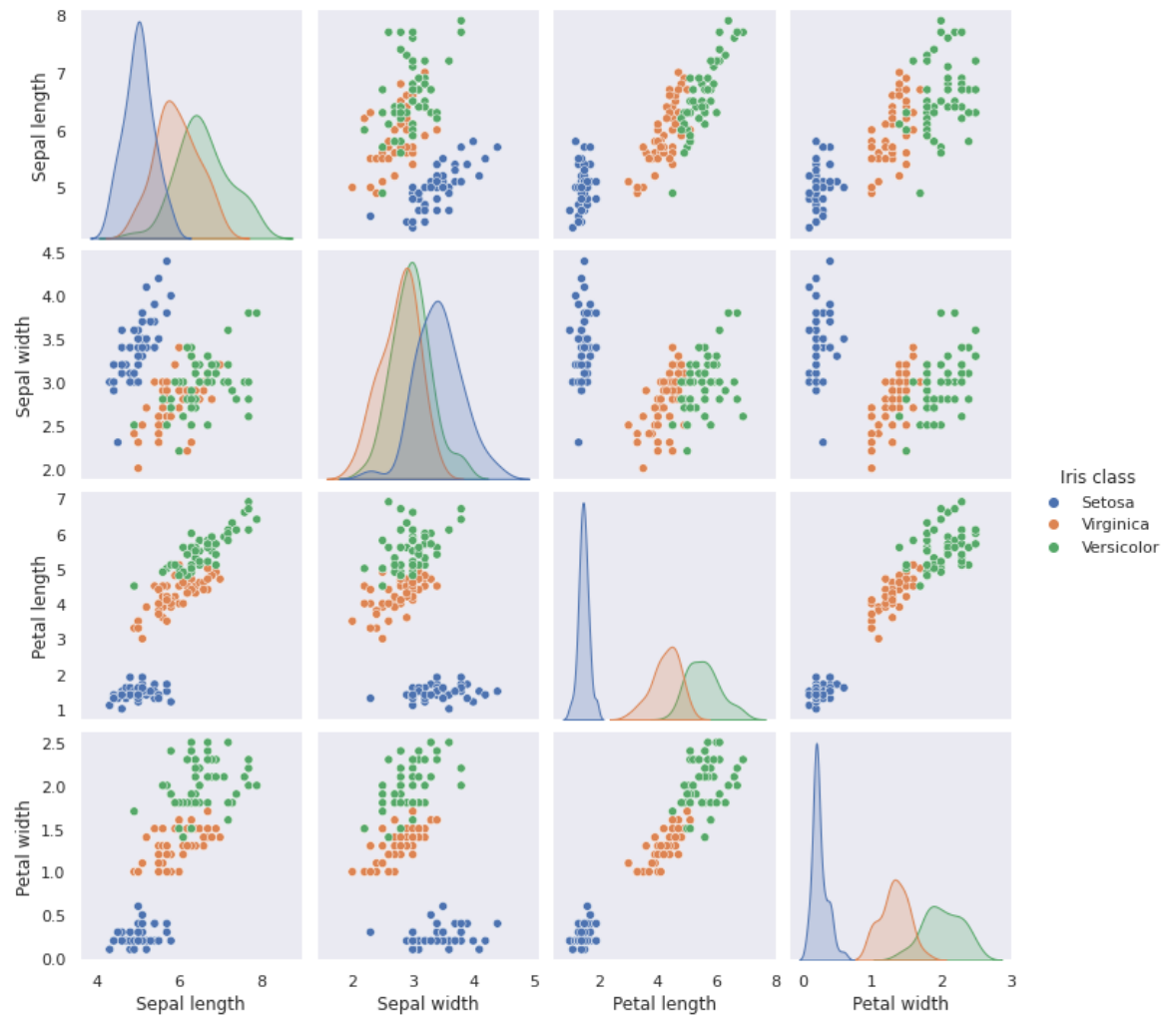
1. Sepal Length
2. Sepal Width
3. Petal Length
4. Petal Width

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 [3]:  # 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 [4]: # 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 [5]: RANDOM_SEED = 6
xTrain, xTest, yTrain, yTest = train_test_split(X_df, Y_df, test_size =0.3,\
                                                random_state=RANDOM_SEED)

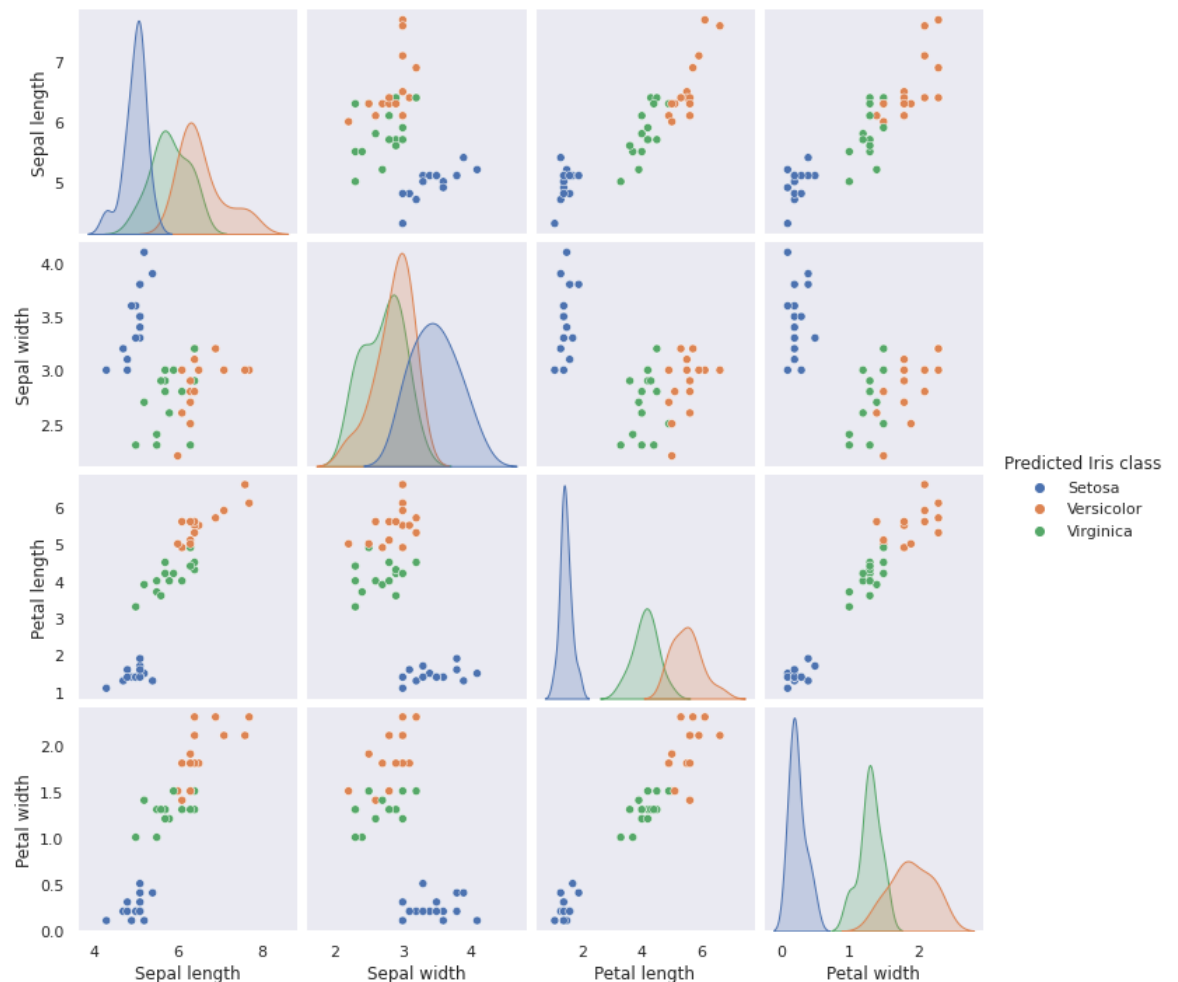
#plot the testing data
test_df = xTest.join(yTest)
# print(test_df.head)
# perceptron training
percep = Perceptron(max_iter = 1000)
percep.fit(xTrain, yTrain)
prediction = percep.predict(xTest)

# print(prediction)
# display the classifiers performance
prediction_df = pd.DataFrame(prediction, columns=['Predicted Iris class'], in
# print(prediction_df.head)

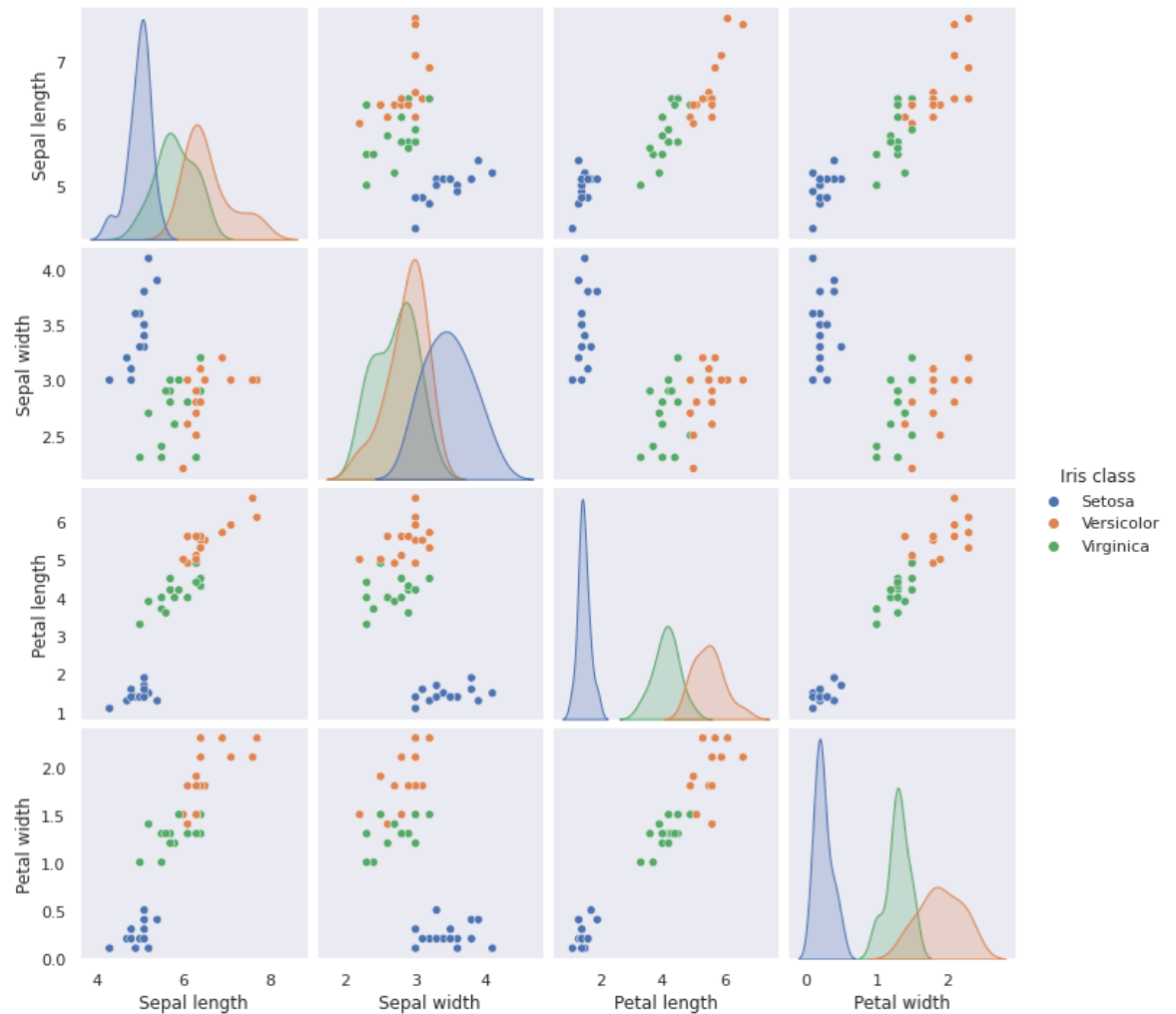
prediction_df_index_df = prediction_df.join(xTest)
# print(prediction_df_index_df.head)

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

```




```
In [6]: pair_test = sns.pairplot(test_df, hue='Iris class') #test data from the data
```



Question 1:

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

A perceptron is a linear classification algorithm which means that it learns a decision boundary that separates two classes using a line (called a hyperplane) in the feature space. This is appropriate to use when the classes are linearly separable. As mentioned above, the given dataset has one obvious class that is separate from a cluster of the other two classes.

The perceptron does a good job in classifying that one obvious class i.e., Setosa separate from a cluster of the other two Versicolor and Virginica classes. This is because we can see clearly from the plots above that there exists a clear decision boundary that separates the Setosa class from the cluster of the other two classes. We also see that the data points for classes - Versicolor and Virginica overlap with each other depicting that they form a cluster. The decision boundary can be classified as a "good" decision boundary as it clearly classifies the Setosa class different from the cluster of the other two classes because it has a low error value. Thus, overall the perceptron does a good job at handling the task.

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 [9]: # change the layers, retrain the mlp
cls = MLPClassifier(solver = 'sgd', activation = 'relu', \
                    hidden_layer_sizes = (8,3,), max_iter = 100000)

for i in range(0,5):
    cls.fit(xTrain, yTrain)

mlp_z = cls.predict(xTest)

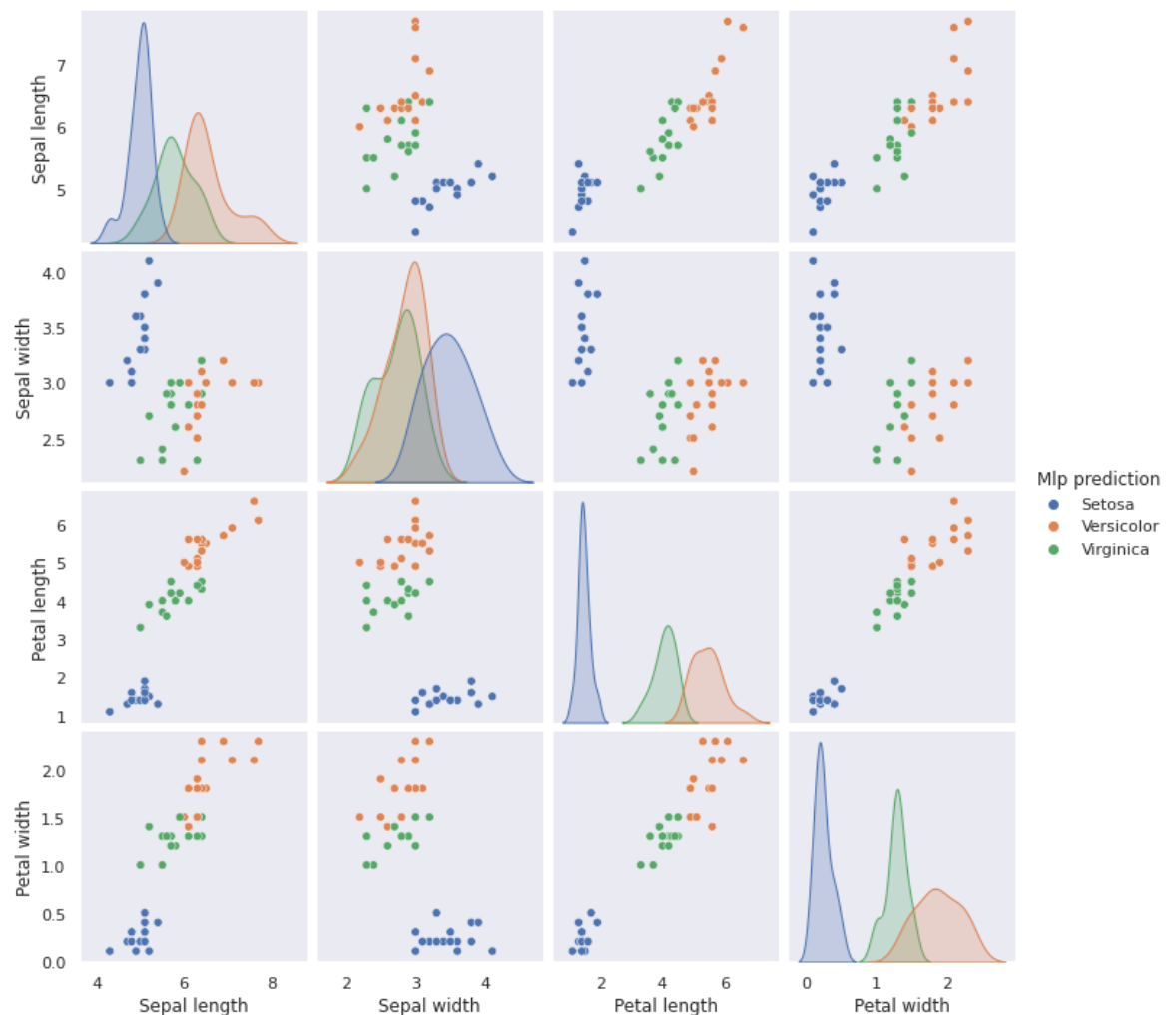
mlp_z.reshape(-1,1)

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')

```



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:

In the initial run of the code, the MLP did not perform the classification task well compared to the perceptron. Overall, it generated a plot only for the first class - Setosa and didn't generate a classification plot for all three classes. Thus, we can conclude that the MLP did not do well and it was not able to classify the three classes.

However, when the code is run again, we clearly see that the MLP classifies the three classes similar to the perceptron. It clearly classifies the class Setosa separate from the cluster of the other two classes. The plot also has clear good decision boundaries as were shown by the perceptron.

Sometimes, it is possible to get different results when we run the same algorithm on the same data due to the nature of the learning algorithm. The reason why the MLP displayed different classification in the first and second run is because of the stochastic nature of the training algorithm. This means that the algorithm is not deterministic i.e., given the same dataset, it may not learn the same model every time. The algorithm, on the other hand has a behavior that incorporates elements of randomness. Thus, we can say that the stochastic nature of the algorithm gives different results in the first and second run.

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 availability 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

$$RMSE = \sqrt{\frac{\sum (Approximated - observed)^2}{n}}$$

```

In [53]: # Obtain training data
moxeeData = sio.loadmat('moxeetrainingdata.mat') # Load variables from the
trainingInputs = moxeeData['pressureData'] # Pressure values and dif
trainingTargets = moxeeData['dataEstimate'] # Estimate of incoming so

# Preprocess the training inputs and targets
iScaler = preprocessing.StandardScaler() # Scaler that removes the mean an
scaledTrainingInputs = iScaler.fit_transform(trainingInputs) # Fit and scal

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 decis
mlp = MLPRegressor(
    hidden_layer_sizes = (50,), # One hidden layer with 50 neurons
    activation = 'tanh', # Logistic sigmoid activation function
    solver = 'sgd', # Gradient descent
    learning_rate_init = 0.01, # Initial learning rate
)

#
##### Create the de
dt_reg = DecisionTreeRegressor(criterion='mse', max_depth = 10)
dt_reg.fit(scaledTrainingInputs, scaledTrainingTargets)

### MODIFY THE VALUE BELOW ###
noIterations = 900 # Number of iterations (epochs) for which the MLP trains
### MODIFY THE VALUE ABOVE ###

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

# Train the MLP for the specified number of iterations
for i in range(noIterations):
    mlp.partial_fit(scaledTrainingInputs, np.ravel(scaledTrainingTargets)) #
    currentOutputs = mlp.predict(scaledTrainingInputs) # Obtain the outputs
    trainingError[i] = np.sum((scaledTrainingTargets.T - currentOutputs) ** 2

# Plot the error curve
plt.figure(figsize=(10,6))
ErrorHandle ,= plt.plot(range(noIterations), trainingError, label = 'Error 50
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
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) #
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)
testRMSE = np.sqrt(np.sum((testOutputs - testTargets[:, 0]) ** 2) / len(testC

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

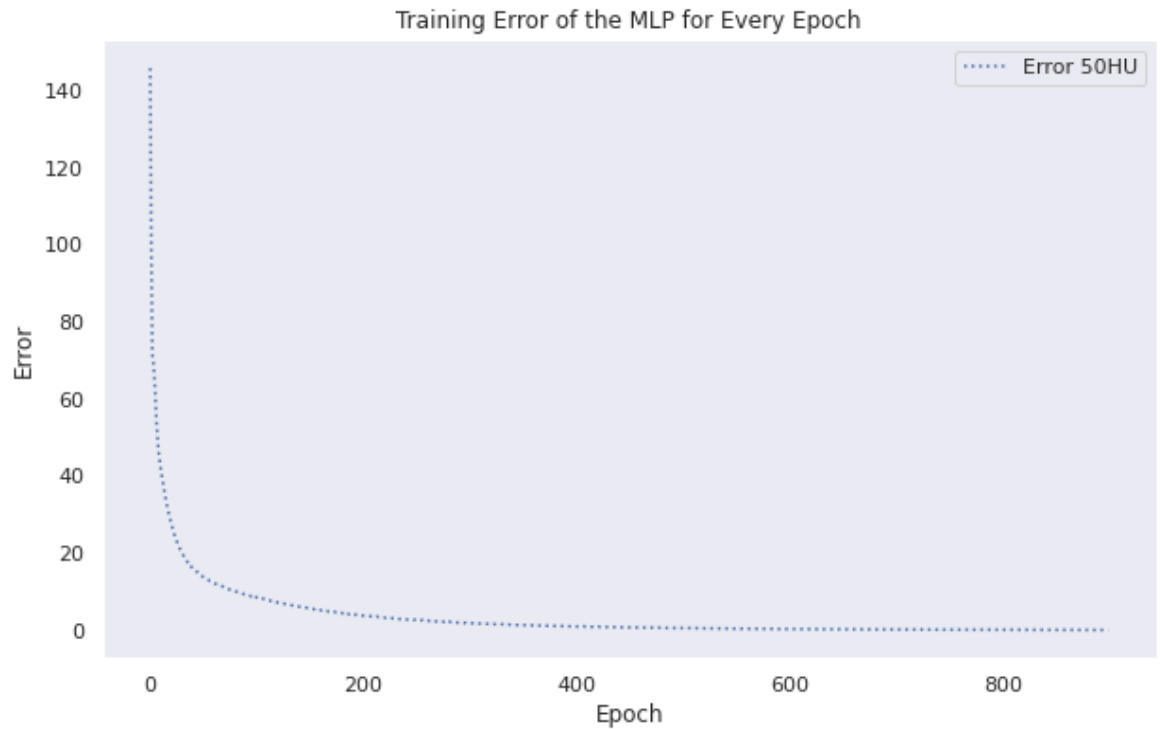
print("Training RMSE:", trainingRMSE, "MJ/m^2")
print("Test RMSE:", testRMSE, "MJ/m^2")
##### Print t
print("Decision Tree training RMSE:", trainingTreeRMSE, 'MJ/m^2')
print("Decision Tree Test RMSE:", testTreeRMSE, 'MJ/m^2')
day = np.array(range(1, len(testTargets) + 1))

# Plot training targets vs. training outputs
plt.figure(figsize=(10,6))
trainingTargetHandle ,= plt.plot(day, trainingTargets / 1000000, label = 'Tar
trainingOutputHandle ,= plt.plot(day, trainingOutputs / 1000000, label = 'Out
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 valu
testOutputHandle ,= plt.plot(day, testOutputs / 1000000, label = 'Outputs 50H
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 t
plt.figure(figsize=(10,6))
testTreeTargetHandle, = plt.plot(day, testTargets / 1000000, label = 'Target
testTreeOutputHandle, = plt.plot(day, treeTestingOutputs / 1000000, label = '
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()

```

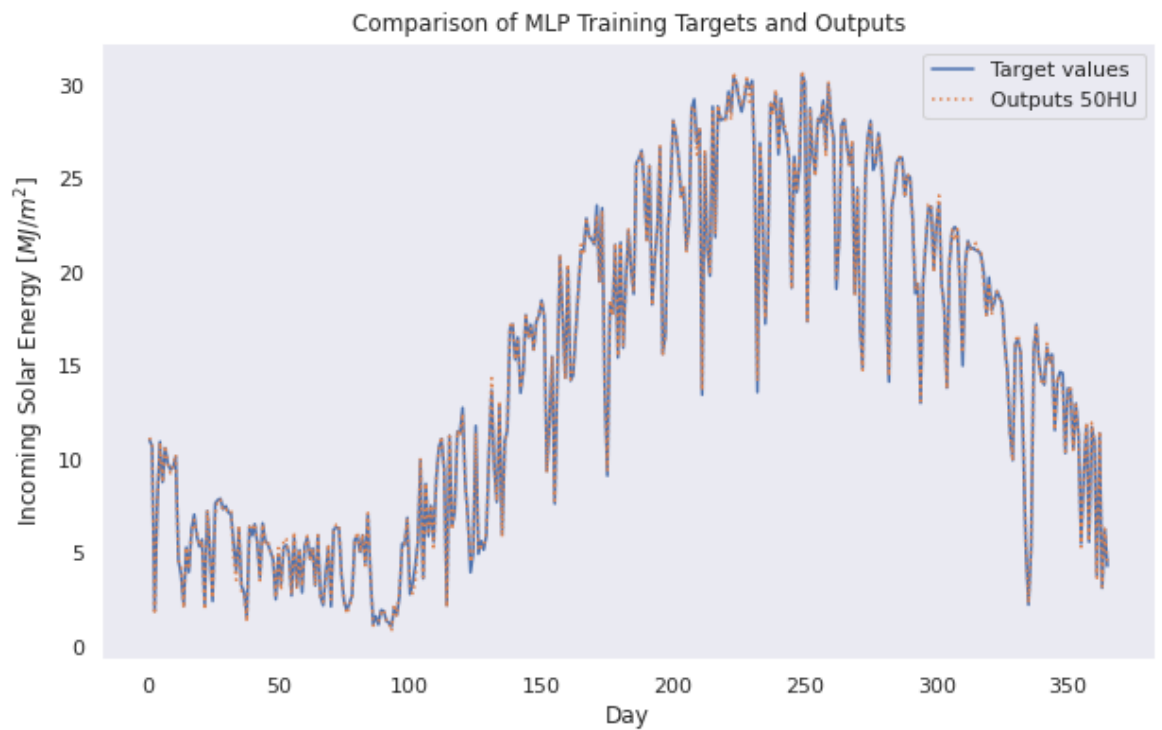


Training RMSE: 0.22188660110891786 MJ/m²

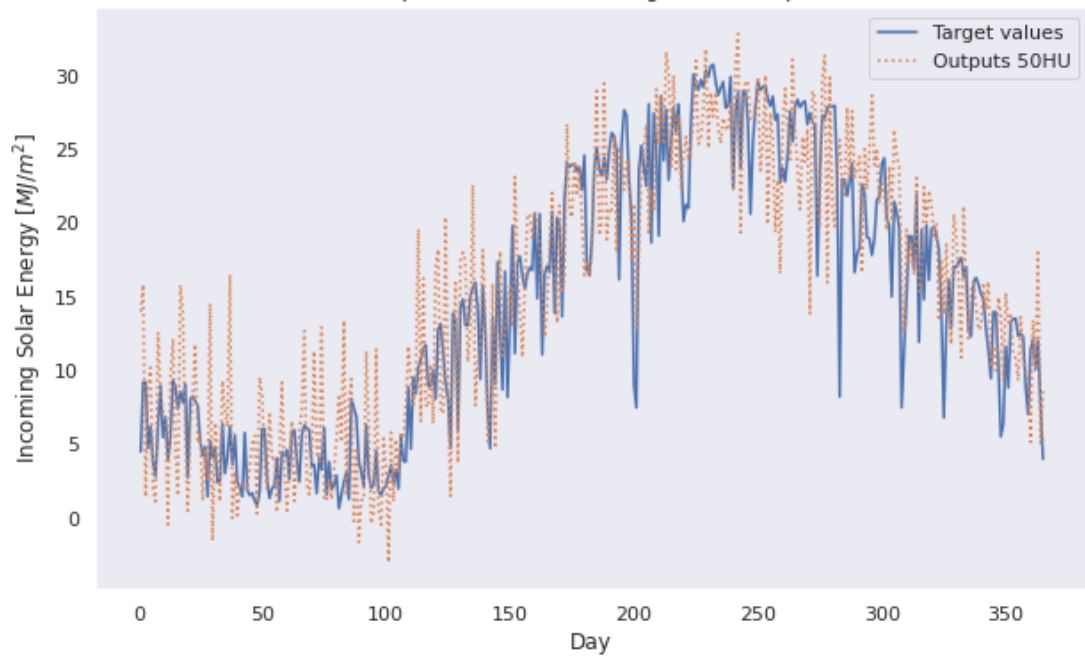
Test RMSE: 4.333005104447713 MJ/m²

Decision Tree training RMSE: 0.18092630438000787 MJ/m²

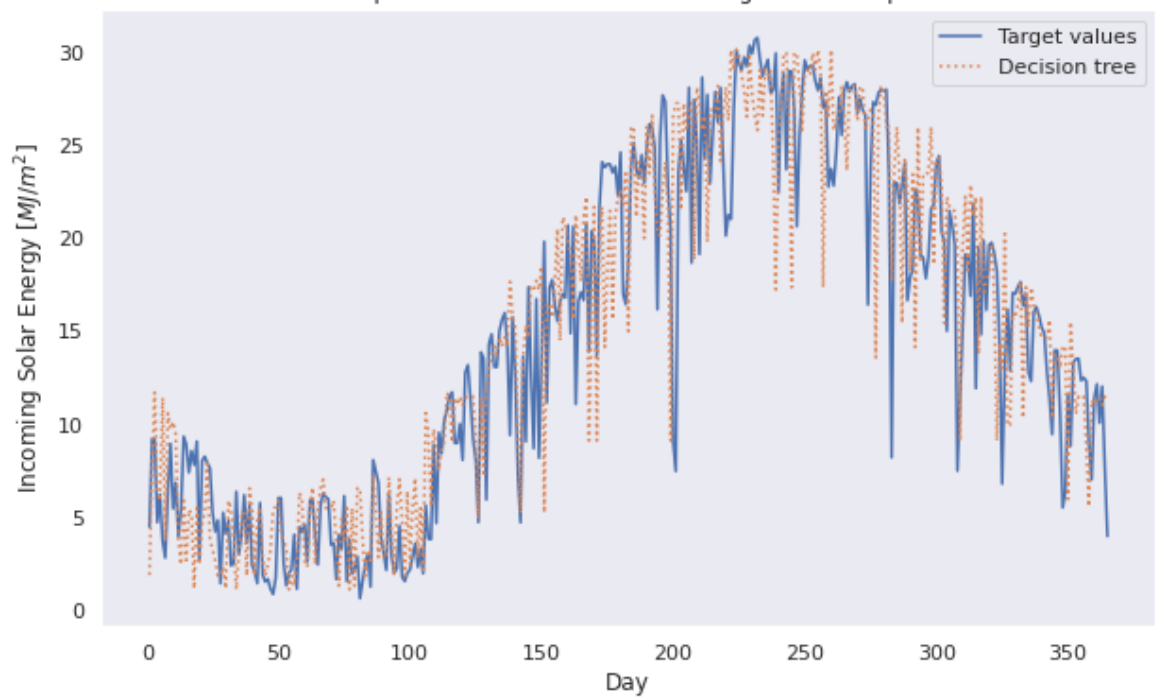
Decision Tree Test RMSE: 4.129775814248386 MJ/m²



Comparison of MLP Test Targets and Outputs



Comparison of Decision Tree Test Targets and Outputs



Fill the box below with your answer for question 1:

We observe that lower number of hidden layers give a higher RMSE (like 2.60 for 5 hidden layers for 600 iterations for a logistic activation function) when coupled with higher iterations. For a pair of small number of hidden layers and smaller number of iterations (eg: 5, 100) for different activation functions, we obtain a higher RMSE, which implies that we get an improper fit for the curves. As the number of hidden layers and number of iterations are increased, the value of RMSE becomes smaller and smaller for different activation functions which implies a better fit for the curve. All four types of activation functions were used to beat the decision table and their performance can be ranked as: 1. Tanh 2. Relu 3. Logistic and 4. Identity. I came up with this ranking based on the value of RMSE I got for different combinations of number of hidden layers and iterations for each activation function.

The tradeoffs associated with both lowering and raising the number of iterations can be explained as follows: when number of iterations are very low, it causes underfitting for the curve while when they are very high they lead to an overfitting for the curve.

After using a combination of number of hidden layers, number of iterations and activation functions, the best result obtained was for tanh function with 50 hidden layers and 900 number of iterations. This gave a training data RMSE of 0.366 which was the closest to the training data RMSE for the decision tree. More than 900 iterations could potentially lead to overfitting. While the logistic function gave an RMSE of 2.60 and relu gave 0.40, thus tanh was concluded as the best to use as activation function. Since, despite of all the combinations of number of hidden layers, number of iterations and activation functions, we couldn't generate an RMSE lesser than that of the decision tree, thus, we could not beat the decision tree.

Note: The identity function gave nan for RMSE whenever it was used in the code. The values obtained for the same data set gave slightly different results when they were run for more than once in the code.

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 [59]: #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)

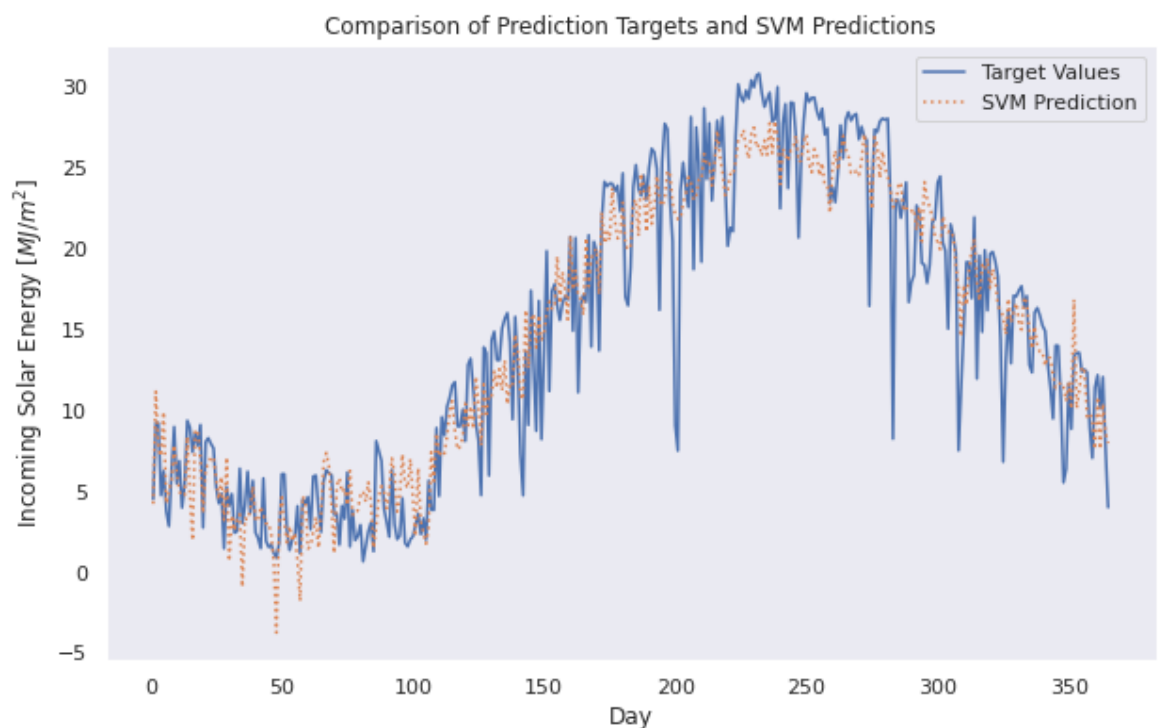
#Calculate and display training and test root mean square error (RMSE)
trainingsvmRMSE = np.sqrt(np.sum((trainingOutputs - trainingTargets[:, 0]) ** 2) / len(trainingTargets))
testsvmRMSE = np.sqrt(np.sum((testOutputs - testTargets[:, 0]) ** 2) / len(testTargets))

#### 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 Predictions')
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: 2.987745431398657 MJ/m²

Test RMSE: 2.988144742787433 MJ/m²

```

In [61]: # Modify this neural network
mlp = MLPRegressor(
    hidden_layer_sizes = (50,),      # One hidden layer with 50 neurons
    activation = 'tanh',             # Logistic sigmoid activation function
    solver = 'sgd',                  # Gradient descent
    learning_rate_init = 0.01,       # Initial learning rate
)

### MODIFY THE VALUE BELOW ###
noIterations = 900 # Number of iterations (epochs) for which the MLP trains
### MODIFY THE VALUE ABOVE ###

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

# Train the MLP for the specified number of iterations
for i in range(noIterations):
    mlp.partial_fit(scaledTrainingInputs, np.ravel(scaledTrainingTargets)) #
    currentOutputs = mlp.predict(scaledTrainingInputs) # Obtain the outputs
    trainingError[i] = np.sum((scaledTrainingTargets.T - currentOutputs) ** 2)

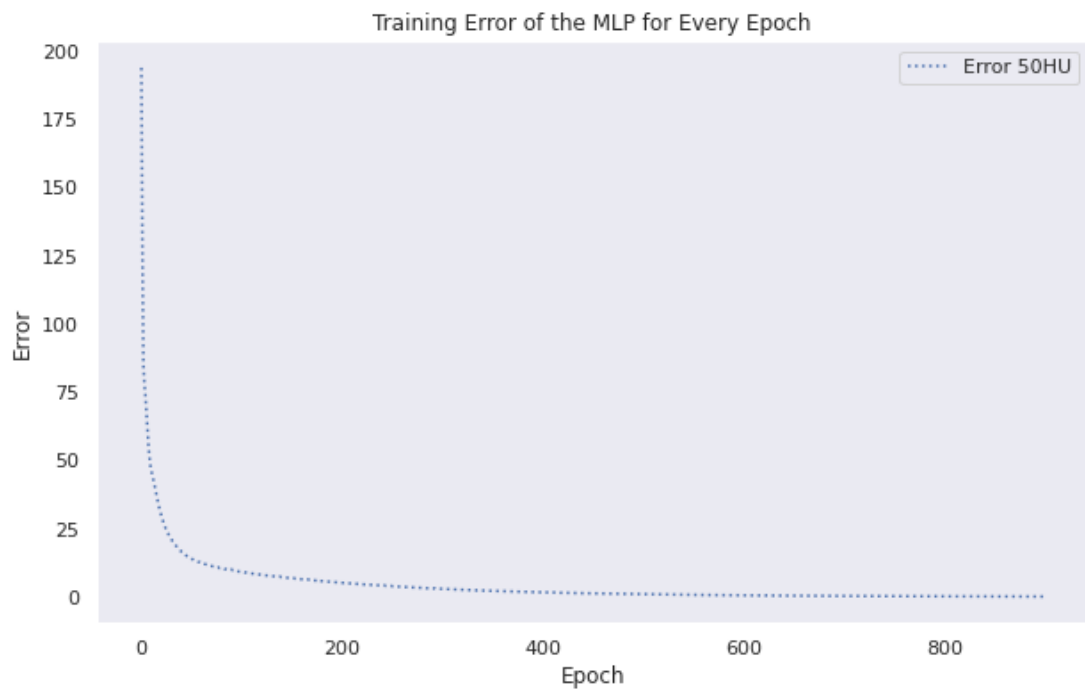
# Predict
scaledTrainingOutputs = mlp.predict(scaledTrainingInputs)
scaledTestOutputs = mlp.predict(scaledTestInputs)
# Training output conversion
trainingOutputs = tScaler.inverse_transform(scaledTrainingOutputs)
testOutputs = tScaler.inverse_transform(scaledTestOutputs)

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

# Plot the error curve
plt.figure(figsize=(10,6))
ErrorHandle, = plt.plot(range(noIterations), trainingError, label = 'Error 50')
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: 0.2825320193341544
Test RMSE: 4.483580020238104

Fill the box below with your answer for question 2:

In []: We observe that lower number of hidden layers give a higher training data RMSE (like 1.08 for 20 hidden layers for iterations for a tanh activation function) when coupled with higher iterations. For a pair of small number of hidden layers and smaller number of iterations (eg: 20, 600) for different activation functions, we obtain a higher RMSE, which implies that we get an improper fit for the curves. However, the training data RMSE values that we obtain from such values is still smaller than the training data RMSE for the SVM. As the number of hidden layers and number of iterations are increased, the value of RMSE becomes smaller and smaller for different activation functions which implies a better fit for the curve. All four types of activation functions were used to beat the decision table and their performance can be ranked as: 1. Tanh 2. Logistic 3. Relu and 4. Identity. I came up with this ranking based on the value of RMSE I got for different combinations of number of hidden layers and iterations for each activation function. The tradeoffs associated with both lowering and raising the number of iterations can be explained as follows: when number of iterations are very low, it causes underfitting for the curve while when they are very high they lead to an overfitting for the curve. After using a combination of number of hidden layers, number of iterations and activation functions, the best result obtained was for tanh function with 50 hidden layers and 900 number of iterations. This gave a training data RMSE of 0.27 which was the smallest RMSE that could be perhaps be obtained. More than 900 iterations could potentially lead to overfitting. While the logistic function gave an RMSE of 2.12 and relu gave 2.97, thus tanh was concluded as the best to use as activation function. After using all the combinations of number of hidden layers, number of iterations and activation functions, we could generate RMSE values lesser than that of the SVM, and thus, we could beat the SVM. Note: The identity function gave nan for RMSE whenever it was used in the code. The values obtained for the same data set gave slightly different results when they were run for more than once in the code.

Abstract

A perceptron is a simplified model of the biological neurons in our brain. It is the simplest neural network which is comprised of just one neuron. In its functioning, a perceptron takes the input as a vector (say x) and weighs the relevance of each input using the multiplication of the input by its respective weight (say w). This leads to the generation of the total sum of all weighted inputs/transmitted signal. This signal is passed through a synaptic (keeps linear input-output relation) or activation link (determined by an activation function which keeps a non-linear input-output relation). The output (say y) is generally an action that we take based on our prediction. The training of a perceptron involves error calculation which is the difference of desired output value and actual output. The error helps determine how to update its weights such that an output value closer to the desired value is obtained.

Since the applications of a perceptron are very limited thus, its more practical extension - multi-layer perceptron (MLP) is used for flexibility in classification problems. This is because an MLP adds an extra hidden layer(s) between the inputs and outputs. It uses the backpropagation learning algorithm which employs a gradient descent to minimize the squared error between the network outputs and the targets. MLP is trained using the same data, multiple times in epochs and this continues until the network reaches a convergence point that is defined by the user through a tolerance value.

Introduction

In this lab two machine learning tasks were performed: classification and regression. Classification is the act of arranging things based on their properties while regression is essentially reasoning backwards. Classification is the process of predicting the class of given data points. Classes are sometimes called as targets/ labels or categories. In this lab, we used perceptron and MLP to classify a class of Iris flower separate from a cluster of the other two Iris classes. A perceptron is a linear classification algorithm which means that it learns a decision boundary that separates two classes using a line (called a hyperplane) in the feature space. This is appropriate to use when the classes are linearly separable. Thus, a perceptron was used for the classification exercise. Since the applications of a perceptron are very limited thus, its more practical extension - multi-layer perceptron (MLP) was also used for flexibility in the classification exercise.

For the regression exercise, we played with certain parameters such as the - size of hidden layers (1-50), activation functions (identity, logistic, tanh and relu) and number of iterations (1-1000) to beat a decision tree and a Support Vector Machine (SVM). The results were analyzed on the basis of the fitting of the curve and the RMSE.

Conclusion

Two machine learning tasks were performed in the lab: classification and regression. A perceptron and MLP were used for the classification exercise and for the regression exercise, we played with certain parameters such as the – size of hidden layers (1-50), activation functions (identity, logistic, tanh and relu) and number of iterations (1-1000) to beat a decision tree and a Support Vector Machine (SVM).

We observed that the perceptron and MLP could successfully help in categorizing the Setosa class from the cluster of the other two classes – Virginica and Versicolor. Clear “good” decision boundaries were visible in the classification for the perceptron and were visible for the MLP but in the second run for the code. This is because in the first run, the MLP only displayed results for the Setosa class, possibly because of its stochastic behaviour. Overall, both perceptron and MLP behaved well in the classification problem.

For the regression exercise, after using a combination of number of hidden layers, number of iterations and activation functions, we observed that the smaller number of hidden layers lead to improper fitting while smaller number of iterations lead to underfitting and larger number of iterations lead to overfitting. Thus, for beating the decision tree, a high number of hidden layers and potentially large (but not too large to avoid overfitting) were described as the best data set to get the lowest RMSE. Lowest RMSE was obtained for tanh function. However, we could not find any RMSE value smaller than the one for the decision tree, and so we could not beat the decision tree.

For beating the SVM, we again used a high number of hidden layers and potentially large (but not too large to avoid overfitting) to describe the best data set to get the lowest RMSE. Lowest RMSE was obtained for tanh function and it was smaller than the one for the SVM so, we could beat the decision tree.

Lab 3 Marking Guide

Exercise	Item	Total Marks	Earned Marks
	<i>Pre – lab</i>	10	
	<i>Abstract</i>	3	
	<i>Introduction</i>	3	
	<i>Conclusion</i>	4	
1	<i>Classification</i>	35	
2	<i>Regression</i>	45	
	TOTAL	100	

