Biostatistics (STA5195) Lab 3 Report

Iris Dataset Analysis

Christopher Rutherford

September 18, 2020

Abstract

The Iris dataset is a commonly used dataset used for demonstrating machine/deep learning algorithms. It contains 150 data points, 50 each for 3 species of Iris: Iris setosa, Iris virginica, and Iris versicolor. It includes the measurements, in cm, of each flower's sepal length and width along with the petal length and width. We use a neural network to predict which subspecies of Iris a flower belongs to. We will also use the `wandb` (**w**eights **and** **b**iases) service to keep track of our neural network's metrics, hyperparameters, CPU/GPU utilization, and more.

1 Introduction

The Iris dataset is one of the most commonly used datasets for the demonstration of classification algorithms in machine learning. The measurements were collected by Edgar Anderson, and the dataset introduced by Ronald Fisher in 1936 to present an example of linear discriminant analysis [1, 2]. R includes the dataset in the base package, and the scikit-learn library includes it for Python. By today's "big data" standards, it is a relatively simple dataset containing 150 samples of three species of the Iris plant. There are 50 samples each of Iris setosa, Iris virginica, and Iris setosa. Each sample contains a measurement for the plant's sepal length, sepal width, petal length, and petal width, along with the species of Iris it belongs to.

2 Theory

With the independent variables all being numeric and the dependent variable being categorical, there is lots of flexibility with which model we can choose for our analysis. We will demonstrate how a neural network can be used to predict which species of Iris a flower belongs to given the four measurements. Figure 1 provides an overview of how this neural network can be visualized. We provide the four variables as an input to the model (yellow circles). This data is fed to the "hidden" layers (blue circles), which perform the necessary calculations. This information is fed to the output layer (orange circles) to determine which species of Iris the flower belongs to.

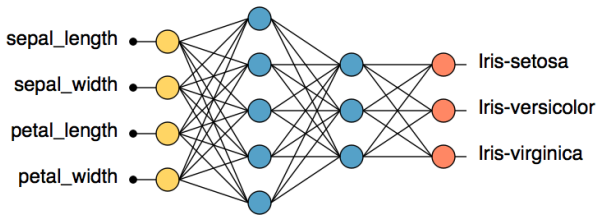

Figure 1: Overview of neural network for Iris species prediction

3 Procedures

As usual, we begin by importing the necessary libraries. This package is extremely convenient due to the inclusion of common libraries such as pandas, numpy, matplotlib, and more.

```
!pip install git+https://github.com/williamedwardhahn/mpcr
from mpcr import *
```

Once this is done importing and installing, we can mount our google drive to our colab notebook:

```
drive.mount('/content/drive')
```

The next code block assumes the file is named `Iris.csv` and is saved in a folder named Data. Adjust this as necessary.

```
dataset = pd.read_csv('/content/drive/My Drive/Data/Iris.csv')
dataset
```

The printout of the dataset shows us our columns and the head/tail of our rows:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...	...
145	146	6.7	3.0	5.2	2.3	Iris-virginica
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 6 columns

Figure 2: Iris dataset overview

We are interested in using the four measurements in order to predict the species, but first, we create some visualizations to see if we can detect some possible patterns in our data.

```
g = sns.pairplot(dataset) #create a histogram for each variable and
    a scatterplot for each pair of variables
```

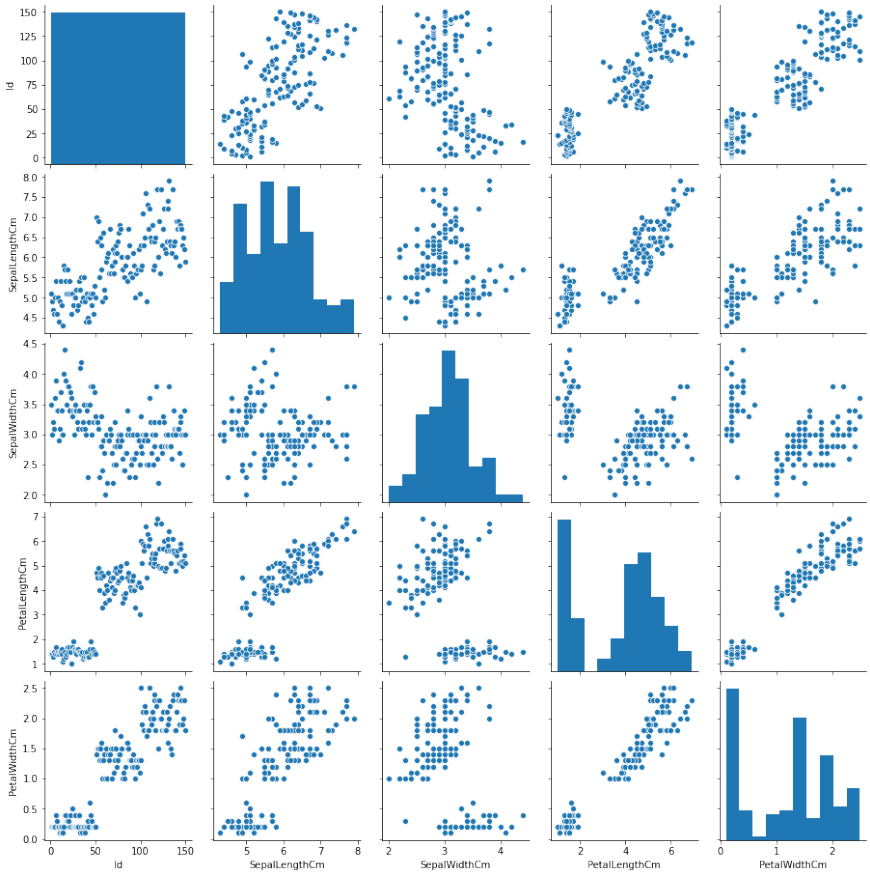

Figure 3: Histograms and scatterplots for each variable

Along the diagonal, we have histograms for each variable, and off of the diagonal, we have scatterplots between each pair of variables. Some of the plots show some very evident clusters of data. We can color each data point based on what species it belongs to to get a better look at the data:

```
g = sns.PairGrid(dataset, hue="Species") #same as above, but color
    each dot based on its species
g = g.map_diag(plt.hist) #histograms along the diagonal
g = g.map_offdiag(plt.scatter) #scatterplots off the diagonal
g = g.add_legend()
```

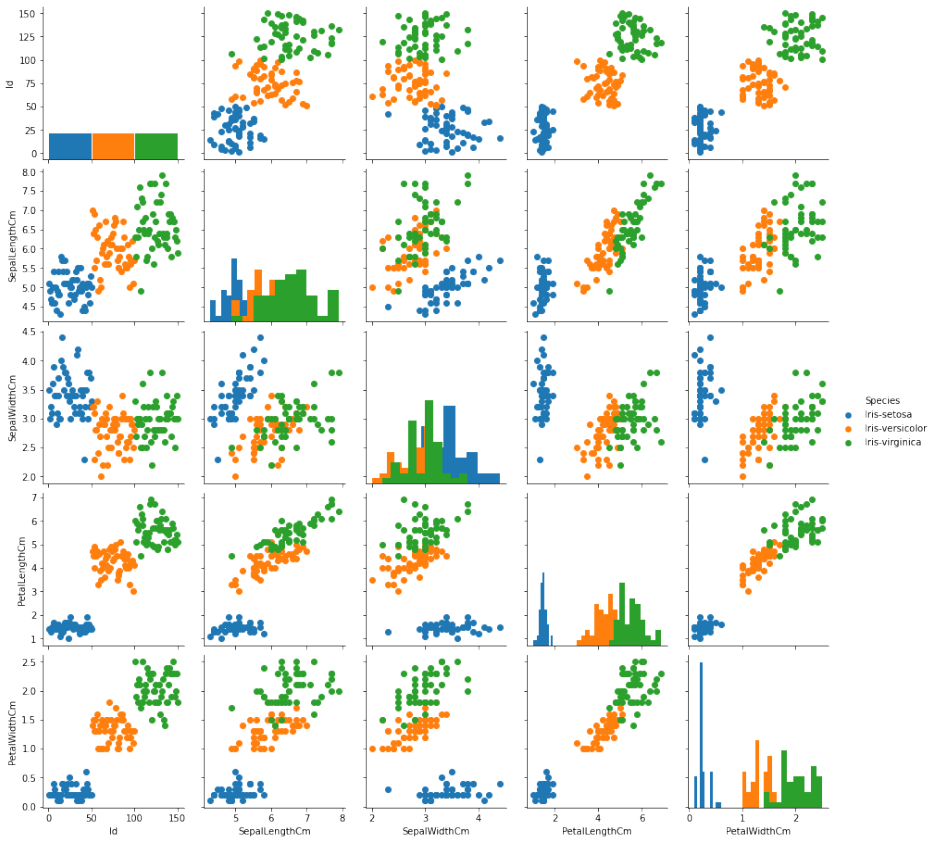

Figure 4: Histograms and scatterplots for each variable, colored by species

Providing a color for each species makes the clusters much more apparent in most of the scatterplots.

4 Analysis

Now we begin preparing our data for our model.

```
target_data = dataset[['Species']]
input_data = dataset.drop(['Id','Species'],axis=1)
```

The `target_data` variable stores the Y variable, which is the data for which species/class each flowers belongs to. `input_data` stores our X variables, or the ones used to predict the class. Since `input_data` is currently a DataFrame, we need to convert it to a numpy array for our model.

```
input_data = np.array(input_data)
```

We can create a visualization of this array:

```
plt.imshow(input_data[0:10,:])
```

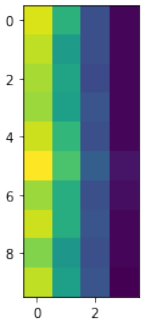

Figure 5: Heatmap of values of our input array

This heatmap indicates that the first column (sepal length) has the largest values (yellow) in the array, and the last column (petal width) has the smallest values (purple). Now, we one hot encode our target variable to make it easier for our network to work with:

```
target_data = pd.get_dummies(target_data.Species)
target_data
```

	Iris-setosa	Iris-versicolor	Iris-virginica
0	1	0	0
1	1	0	0
2	1	0	0
3	1	0	0
4	1	0	0
...	...	...	...
145	0	0	1
146	0	0	1
147	0	0	1
148	0	0	1
149	0	0	1

150 rows × 3 columns

Figure 6: One-hot encoded output variables

Instead of using one `species` column with labels, each species gets its own column. Data points with a 1 belong to that species, and 0 if they do not. As such, each data point will have a 1 in only one of the three columns. We can then convert this back into a single array where Iris setosa flowers will have a value of 0, Iris versicolor 1, and Iris virginica 2.

```
_, target_data = np.where(target_data==1)
target_data
```

Output: `array([0, 0, 0, ..., 1, 1, 1, ..., 2, 2, 2,])`

Now that we have our input and output data formatted for the model, we can create our train/test splits.

```
r = np.random.permutation(input_data.shape[0])
cut = int(0.8*len(r))
input_data.shape
```

Output: `(150, 4)`

The above code generates a random permutation of the integers $1, 2, ..., 150$. Since we are using 80% of the data for training the model, we set the cutoff to be 80% of the length of `r`. This leaves us with 120 data points for training and 30 for testing. We create our splits:

```
X = input_data[r[:cut],:]
X_test = input_data[r[cut:],:]
Y = target_data[r[:cut]]
Y_test = target_data[r[cut:]]
```

Our `X` and `Y` variables both contain 120 random data points. `X` has the first 120 measurements and `Y` has the first 120 corresponding species. `X_test` and `Y_test` contain the remaining 30 data points.

We now define the softmax function, which will convert our output values to probabilities to determine which species a flower most likely belongs to:

```
def softmax(x):
    s1 = torch.exp(x - torch.max(x,1)[0][:,None])
    s = s1 / s1.sum(1)[:,None]
    return s
```

Mathematically, the softmax function is defined as follows:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{1}$$

where $\sigma(\mathbf{z})_i \in (0, 1)$ is the array (or vector) of probabilities of the ith element belonging to each of the output classes.

Since applying the softmax function to an array turns it into a probability distribution [3], we can use cross entropy as our loss function.

```
def cross_entropy(outputs, labels):
    return -torch.sum(softmax(outputs).log()[range(outputs.size()
   [0]), labels.long()])/outputs.size()[0]
```

The following block of code generates random normal variables with a mean of 0 and standard deviation of 0.1, but only values within 2 standard deviations of the mean are generated.

```
def randn_trunc(s): #Truncated Normal Random Numbers
    mu = 0
    sigma = 0.1
    R = stats.truncnorm((-2*sigma - mu) / sigma, (2*sigma - mu) /
   sigma, loc=mu, scale=sigma)
    return R.rvs(s)
```

This allows us to generate random normal variables without getting extremely large/small values.

Now, we define a function to calculate the model's accuracy:

```
def acc(out,y):
    with torch.no_grad():
        return (torch.sum(torch.max(out,1)[1] == y).item())/y.shape
    [0]
```

This calculates how many flowers had their species labeled correctly, and divided that count by the total number of flowers (150).

The following functions tell Python to perform the computations on the GPU instead of the CPU, which greatly reduces model training time.

```
def GPU(data):
    return torch.tensor(data, requires_grad=True, dtype=torch.float,
    device=torch.device('cuda'))

def GPU_data(data):
    return torch.tensor(data, requires_grad=False, dtype=torch.float
   , device=torch.device('cuda'))
```

This function tells the model whether or not is currently training the model or evaluating it on the test set and gives it a random amount of data from the training/test set. The subset of data starts at the random integer `r` and is of length `b`, which is the batch size.

```
def get_batch(mode):
    b = c.b
    if mode == "train":
        r = np.random.randint(X.shape[0]-b)
        x = X[r:r+b,:]
        y = Y[r:r+b]
    elif mode == "test":
        r = np.random.randint(X_test.shape[0]-b)
        x = X_test[r:r+b,:]
        y = Y_test[r:r+b]
    return x,y
```

The following function tells us how much to adjust our gradient descent based on the learning rate, `c.h`.

```
def gradient_step(w):
    for j in range(len(w)):
            w[j].data = w[j].data - c.h*w[j].grad.data
            w[j].grad.data.zero_()
```

An easy way to view our model's performance is by plotting the accuracy for both the training and testing sets:

```
def make_plots():
    acc_train = acc(model(x,w),y)
    xt,yt = get_batch('test')
    acc_test = acc(model(xt,w),yt)
    wb.log({"acc_train": acc_train, "acc_test": acc_test})
```

The following commands will tell Pythonto place our input and output data on the GPU by converting them to Tensor objects.

```
X = GPU_data(X)
Y = GPU_data(Y)
X_test = GPU_data(X_test)
Y_test = GPU_data(Y_test)
```

We define the ReLU layer, which is just $max(0, x)$. That is, negative values are mapped to zero, while positive values remain the same.

```
def relu(x):
    return x * (x > 0)
```

```
def model(x,w):
    for j in range(len(w)):
        x = relu(matmul(x,w[j]))
    return x
```

```
wb.init(project="Iris");
c = wb.config
c.h = 0.05
c.b = 20
c.layers = 3
c.epochs = 2500
c.f_n = [4,16,16,3]

w = [ GPU(randn_trunc((c.f_n[i],c.f_n[i+1]))) for i in range(c.
   layers) ]

for i in range(c.epochs):
    x,y = get_batch('train')
    loss = cross_entropy(softmax(model(x,w)),y)
    loss.backward()
    gradient_step(w)
    if (i+1) % 1 == 0:
        make_plots()
```

```
acc(model(X,w),Y)
```

Output: `0.30833333333333335`

```
acc(model(X_test,w),Y_test)
```

Output: `0.4666666666666667`

```
for i in range(len(w)):
    plt.imshow(w[i].cpu().detach().numpy())
    plt.show()
```

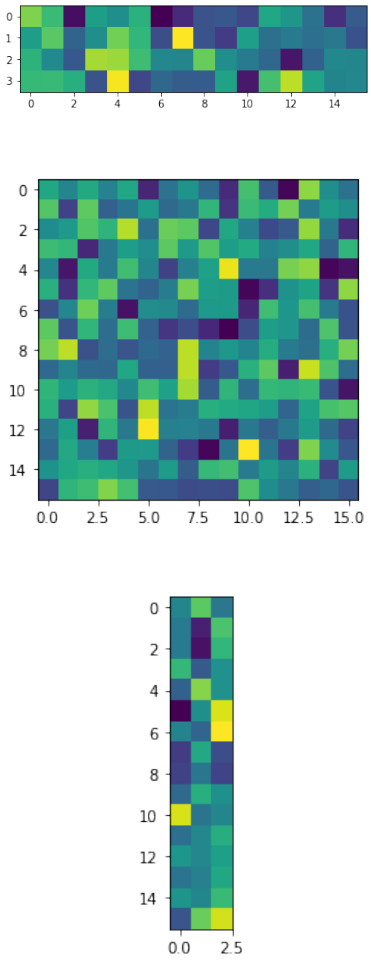

Figure 7: Visualization of weights

5 Conclusions

Overall, our model did not seem to be able to perform very well on this dataset. The model was only able to correctly label about 31% of the training data and 46.7% of the testing data, or 14 of the 30 testing flowers.

References

[1] *Iris (plant)*, available at https://en.wikipedia.org/wiki/Iris_(plant)

[2] *Iris flower data set*, available at https://en.wikipedia.org/wiki/Iris_flower_data_set

[3] Ramsundar, B., Eastman, P., Walters, P., Pande, V. (2019). *Deep learning for the life sciences: Applying deep learning to genomics, microscopy, drug discovery and more* (1st ed.). Sebastopol, CA: O'Reilly Media.