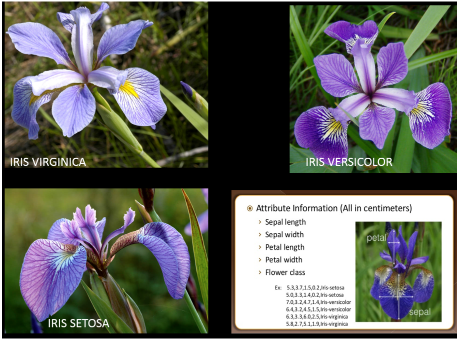
Lab 3

Gil Cohen

Biostatistics, Fall 2020

This data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.



Data Visualization and Preparation

```
#reading the Iris.cvs dataset into a dataframe called dataset
dataset = pd.read_csv('/content/drive/My Drive/Iris.csv')

'''Plotting pairwise relationships between features as scatter plots
   and the marginal distributions of each feature as univariate plots
   along the diagonal'''
g = sns.pairplot(dataset)
```

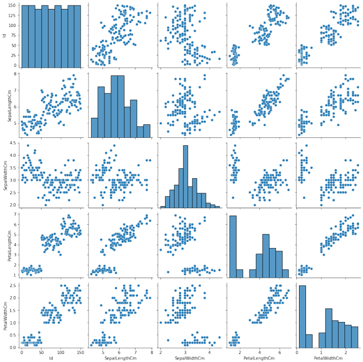


Figure: Pairwise relationships between features

Data Visualization and Preparation

```
#Plotting 5 by 5 subplot grid for pairwise relationships
g = sns.PairGrid(dataset, hue="Species")

#Setting the diagonal plots to be histograms
g = g.map_diag(plt.hist)
#Setting the non-diagonal plots to be scatterplots
g = g.map_offdiag(plt.scatter)
#adding a legend
g = g.add_legend()
```

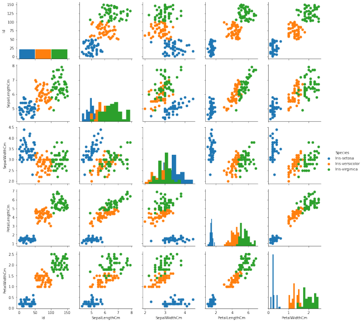


Figure: Pairwise relationships with legend

Data Visualization and Preparation

```
'''creating a new dataframe from dataset called target_data that contains
only the 'Species' column'''
target_data = dataset[['Species']]

'''creating a new dataframe called input_data from the dataset
dataframe by dropping the "Id" and 'Species' columns'''
input_data = dataset.drop(['Id','Species'],axis=1)

#converting input_data into an array
input_data = np.array(input_data)

#converting input_data into an array
input_data = np.array(input_data)

'''displaying the first 10 instances in the input_data array as
a colormap image'''
plt.imshow(input_data[0:10,:])
```

Data Visualization and Preparation

```
'''Converting the categorical data 'Species' into  a dataframe
of indicator variables'''
target_data = pd.get_dummies(target_data.Species)

'''Changing target_data to be an array of the indices of the
columns of target_data at which each instance has a value of 1'''
_, target_data = np.where(target_data==1)

#Random permutation of indices 0, 1, ... ,149 as an array
r = np.random.permutation(input_data.shape[0])

#Cutting the length of r by 20 percent and assigning the value to cut
cut = int(0.8*len(r))

input_data.shape
--> (150, 4)

#Setting X to be the input_data for a random selection of 120 instances
X = input_data[r[:cut],:]
#Setting X_test to be the input_data for the remaining 30 instances
X_test = input_data[r[cut:],:]
#Setting Y to be the target_data for the same instances used in X
Y = target_data[r[:cut]]
#Setting Y_test to be the target_data for the remaining 30 instances
Y_test = target_data[r[cut:]]
```

Defining Functions

```
'''Defining a function called softmax that takes a tensor as input
and returns one containing column values that sum to 1 for each row'''
def softmax(x):
    s1 = torch.exp(x - torch.max(x,1)[0][:,None])
    s = s1 / s1.sum(1)[:,None]
    return s

#Defining a cross entropy loss function
def cross_entropy(outputs, labels):
    return -torch.sum(softmax(outputs).log()[range(outputs.size()[0]),
    labels.long()])/outputs.size()[0]

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)

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

Defining Functions

```
#Defining functions that each construct a tensor from "data"
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'))

#Defining a function that returns a batch of size b for the input_data and target_data
def get_batch(mode):
    b = c.b  #size of batch
    if mode == "train":
        r = np.random.randint(X.shape[0]-b)
        x = X[r:r+b,:] #getting batch from training input_data
        y = Y[r:r+b]
    elif mode == "test":
        r = np.random.randint(X_test.shape[0]-b)
        x = X_test[r:r+b,:] #getting batch from test input_data
        y = Y_test[r:r+b]
    return x,y

#Defining a function that performs a gradient step on w
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_()
```

Defining Functions

```
'''Defining a function that makes plots for the accuracy of a
model over its training data and a batch of its test data'''
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})

'''Calling GPU_data function on the input data and target data
for the training and testing instances to form tensors'''
X = GPU_data(X)
Y = GPU_data(Y)
X_test = GPU_data(X_test)
Y_test = GPU_data(Y_test)

#Function that returns the input value if its greater than 0, and 0 otherwise
def relu(x):
    return x * (x > 0)

def model(x,w):
    for j in range(len(w)):
        '''re-assign x to be the matrix product x * w[j] if
        its positive and 0 otherwise'''
        x = relu(matmul(x,w[j]))
    return x
```

Training and Testing

```
wb.init(project="Iris");
c = wb.config

c.h = 0.05   #increment value for gradient step
c.b = 20   #size of batch
c.layers = 3   #number of layers for w
c.epochs = 2500   #number of epochs for training

c.f_n = [4,16,16,3]   #initial values for randn_trunc

#initial weights
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')   #getting a batch of training instances

    loss = cross_entropy(softmax(model(x,w)),y)

    loss.backward()

    gradient_step(w)   #performing a weight update

    if (i+1) % 1 == 0:
        make_plots()
```

Training and Testing

```
acc(model(X,w),Y) #accuracy of model over the training data
−> 0.675

acc(model(X_test,w),Y_test) #accuracy of model over the test data
−> 0.5666666666666667

X[0] #first instance in training data
−> tensor([5.6000, 3.0000, 4.1000, 1.3000], device='cuda:0')

model(X[0],w) #output of the model for the first instance
−> tensor([−0., −0., −0.], device='cuda:0', grad_fn=<MulBackward0>)

#returns index of maximum value of model output for the first instance
torch.argmax(model(X[0],w))

#Displaying the three tensors in w as color images
for i in range(len(w)):
    plt.imshow(w[i].cpu().detach().numpy())
    plt.show()
```

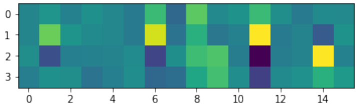


Figure: w[0]

Training and Testing

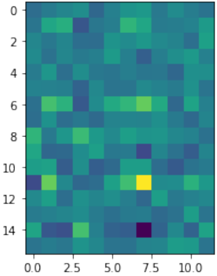


Figure: w[1]

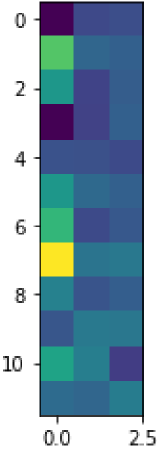


Figure: w[2]