**SOURCE CODE**

User Side views.py

**from** django.shortcuts **import** render,HttpResponse  
**from** django.contrib **import** messages  
**from** .forms **import** UserRegistrationForm  
**from** .models **import** UserRegistrationModel  
**from** django.conf **import** settings  
**import** pandas **as** pd  
**from** .algorithms.DeepLearningAlgorithms **import** MyDeepLearning  
**from** .algorithms.BackPropoagationAlgorithm **import** MyBPAlgorithm  
**from** .algorithms.MyAlgorithmTest **import** AlgorithmExecution  
**import** matplotlib  
*#matplotlib.use("Agg")  
  
# Create your views here.***def** UserRegisterActions(request):  
 **if** request.method == **'POST'**:  
 form = UserRegistrationForm(request.POST)  
 **if** form.is\_valid():  
 print(**'Data is Valid'**)  
 form.save()  
 messages.success(request, **'You have been successfully registered'**)  
 form = UserRegistrationForm()  
 **return** render(request, **'UserRegistrations.html'**, {**'form'**: form})  
 **else**:  
 messages.success(request, **'Email or Mobile Already Existed'**)  
 print(**"Invalid form"**)  
 **else**:  
 form = UserRegistrationForm()  
 **return** render(request, **'UserRegistrations.html'**, {**'form'**: form})  
**def** UserLoginCheck(request):  
 **if** request.method == **"POST"**:  
 loginid = request.POST.get(**'loginname'**)  
 pswd = request.POST.get(**'pswd'**)  
 print(**"Login ID = "**, loginid, **' Password = '**, pswd)  
 **try**:  
 check = UserRegistrationModel.objects.get(loginid=loginid, password=pswd)  
 status = check.status  
 print(**'Status is = '**, status)  
 **if** status == **"activated"**:  
 request.session[**'id'**] = check.id  
 request.session[**'loggeduser'**] = check.name  
 request.session[**'loginid'**] = loginid  
 request.session[**'email'**] = check.email  
 print(**"User id At"**, check.id, status)  
 **return** render(request, **'users/UserHome.html'**, {})  
 **else**:  
 messages.success(request, **'Your Account Not at activated'**)  
 **return** render(request, **'UserLogin.html'**)  
 **except** Exception **as** e:  
 print(**'Exception is '**, str(e))  
 **pass** messages.success(request, **'Invalid Login id and password'**)  
 **return** render(request, **'UserLogin.html'**, {})  
**def** UserHome(request):  
 **return** render(request, **'users/UserHome.html'**, {})  
  
**def** UserViewDataset(request):  
 path = settings.MEDIA\_ROOT + **"\\"** + **"Indian\_Liver\_Patients\_Dataset.csv"** df = pd.read\_csv(path)  
 df = df.to\_html  
 **return** render(request, **'users/UserViewDataset.html'**,{**'data'**:df})  
  
**def** UserResults(request):  
 *#obj = MyDeepLearning()  
 #obj.startDeepLearningProcess()* bp = MyBPAlgorithm()  
 bp\_accuracy=bp.startBackProp()  
 **return** HttpResponse(**"User Results On Process"**)  
  
**def** UserClassification(request):  
 obj = AlgorithmExecution()  
 cls\_accuracy = obj.startProcess()  
 **return** render(request,**"users/classificationResults.html"**,{**'DT'**:cls\_accuracy[0],**'Kmeans'**:cls\_accuracy[1],**'knn'**:cls\_accuracy[2],**'svm'**:cls\_accuracy[3],**'gnb'**:cls\_accuracy[4]})  
  
**def** UserNeuralNetwork(request):  
 obj = MyDeepLearning()  
 ann,j48 = obj.startDeepLearningProcess()  
 bp = MyBPAlgorithm()  
 bp\_accuracy = bp.startBackProp()  
 **return** render(request,**"users/userNeuralresults.html"**,{**"ann"**:ann/100,**'j48'**:j48,**'bp'**:bp\_accuracy/100})

user Side **Models.py**

**from** django.db **import** models  
  
*# Create your models here.***class** UserRegistrationModel(models.Model):  
 name = models.CharField(max\_length=100)  
 loginid = models.CharField(unique=**True**, max\_length=100)  
 password = models.CharField(max\_length=100)  
 mobile = models.CharField(unique=**True**, max\_length=100)  
 email = models.CharField(unique=**True**, max\_length=100)  
 locality = models.CharField(max\_length=100)  
 address = models.CharField(max\_length=1000)  
 city = models.CharField(max\_length=100)  
 state = models.CharField(max\_length=100)  
 status = models.CharField(max\_length=100)  
  
 **def** \_\_str\_\_(self):  
 **return** self.loginid  
  
 **class** Meta:  
 db\_table = **'UserRegistrations'**

User Side **Forms.py**

**from** django **import** forms  
**from** .models **import** UserRegistrationModel  
  
  
**class** UserRegistrationForm(forms.ModelForm):  
 name = forms.CharField(widget=forms.TextInput(attrs={**'pattern'**: **'[a-zA-Z]+'**}), required=**True**, max\_length=100)  
 loginid = forms.CharField(widget=forms.TextInput(attrs={**'pattern'**: **'[a-zA-Z]+'**}), required=**True**, max\_length=100)  
 password = forms.CharField(widget=forms.PasswordInput(attrs={**'pattern'**: **'(?=.\*\d)(?=.\*[a-z])(?=.\*[A-Z]).{8,}'**,  
 **'title'**: **'Must contain at least one number and one uppercase and lowercase letter, and at least 8 or more characters'**}),  
 required=**True**, max\_length=100)  
 mobile = forms.CharField(widget=forms.TextInput(attrs={**'pattern'**: **'[56789][0-9]{9}'**}), required=**True**,  
 max\_length=100)  
 email = forms.CharField(widget=forms.TextInput(attrs={**'pattern'**: **'[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,}$'**}),  
 required=**True**, max\_length=100)  
 locality = forms.CharField(widget=forms.TextInput(), required=**True**, max\_length=100)  
 address = forms.CharField(widget=forms.Textarea(attrs={**'rows'**: 4, **'cols'**: 22}), required=**True**, max\_length=250)  
 city = forms.CharField(widget=forms.TextInput(  
 attrs={**'autocomplete'**: **'off'**, **'pattern'**: **'[A-Za-z ]+'**, **'title'**: **'Enter Characters Only '**}), required=**True**,  
 max\_length=100)  
 state = forms.CharField(widget=forms.TextInput(  
 attrs={**'autocomplete'**: **'off'**, **'pattern'**: **'[A-Za-z ]+'**, **'title'**: **'Enter Characters Only '**}), required=**True**,  
 max\_length=100)  
 status = forms.CharField(widget=forms.HiddenInput(), initial=**'waiting'**, max\_length=100)  
  
 **class** Meta():  
 model = UserRegistrationModel  
 fields = **'\_\_all\_\_'**

ClassificationAlgorithmCode.py

**from** django.conf **import** settings  
**import** seaborn **as** sns  
**from** sklearn.metrics **import** accuracy\_score  
**class** AlgorithmExecution:  
 **def** startProcess(self):  
 **import** pandas **as** pd  
 **import** itertools  
 **import** numpy **as** np  
 **import** matplotlib.pyplot **as** plt  
 **from** sklearn **import** preprocessing  
 *#% matplotlib inline  
 # import dataset* path = settings.MEDIA\_ROOT+**"\\"**+**"Indian\_Liver\_Patients\_Dataset.csv"** *#liver\_data\_train = pd.read\_csv('DataSet/Indian\_Liver\_Patients\_Dataset.csv')* liver\_data\_train = pd.read\_csv(path)  
 print(liver\_data\_train.head())  
 print(liver\_data\_train.shape)  
  
 *# create a column based on dataset to see if a patient has disease or not* **def** label\_disease(liver\_data\_train):  
 **if** liver\_data\_train[**"Dataset"**] == 1:  
 **return 'yes'  
 return 'no'** liver\_data\_train[**'HasDisease'**] = liver\_data\_train.apply(  
 **lambda** liver\_data\_train: label\_disease(liver\_data\_train), axis=1)  
  
 print(liver\_data\_train[**'HasDisease'**].value\_counts())  
 **import** seaborn **as** sns  
 bins = np.linspace(liver\_data\_train.Age.min(), liver\_data\_train.Age.max(), 10)  
 grids = sns.FacetGrid(liver\_data\_train, col=**"Gender"**, hue=**"HasDisease"**, palette=**"Set1"**, col\_wrap=2)  
 grids.map(plt.hist, **'Age'**, bins=bins, ec=**"k"**)  
  
 grids.axes[-1].legend()  
 plt.show()  
 bins = np.linspace(liver\_data\_train.Total\_Bilirubin.min(), liver\_data\_train.Total\_Bilirubin.max(), 10)  
 grids = sns.FacetGrid(liver\_data\_train, col=**"Gender"**, hue=**"HasDisease"**, palette=**"Set1"**, col\_wrap=2)  
 grids.map(plt.hist, **'Total\_Bilirubin'**, bins=bins, ec=**"k"**)  
  
 grids.axes[-1].legend()  
 plt.show()  
 bins = np.linspace(liver\_data\_train.Total\_Protiens.min(), liver\_data\_train.Total\_Protiens.max(), 10)  
 grids = sns.FacetGrid(liver\_data\_train, col=**"Gender"**, hue=**"HasDisease"**, palette=**"Set1"**, col\_wrap=2)  
 grids.map(plt.hist, **'Total\_Protiens'**, bins=bins, ec=**"k"**)  
  
 grids.axes[-1].legend()  
 plt.show()  
 liver\_data\_train[**'Gender'**].replace(to\_replace=[**'Male'**, **'Female'**], value=[0, 1], inplace=**True**)  
 liver\_data\_train.isna().sum()  
 liver\_data\_train[**'Albumin\_and\_Globulin\_Ratio'**].fillna((liver\_data\_train[**'Albumin\_and\_Globulin\_Ratio'**].mean()),  
 inplace=**True**)  
 Feature = liver\_data\_train[[**'Age'**, **'Gender'**, **'Total\_Bilirubin'**, **'Direct\_Bilirubin'**,  
 **'Alkaline\_Phosphotase'**, **'Alamine\_Aminotransferase'**, **'Aspartate\_Aminotransferase'**,  
 **'Total\_Protiens'**, **'Albumin'**, **'Albumin\_and\_Globulin\_Ratio'**]]  
 X = Feature  
 y = liver\_data\_train[**'HasDisease'**].values  
 y[0:5]  
 X = preprocessing.StandardScaler().fit(X).transform(X)  
 X[0:5]  
 **from** sklearn.tree **import** DecisionTreeClassifier  
 **from** sklearn.model\_selection **import** train\_test\_split  
 X\_trainset, X\_testset, y\_trainset, y\_testset = train\_test\_split(X, y, test\_size=0.2, random\_state=5)  
 liverDiseaseTree = DecisionTreeClassifier(criterion=**'entropy'**, max\_depth=7)  
 liverDiseaseTree.fit(X\_trainset, y\_trainset)  
 liverDiseaseTree  
 dt\_yhat = liverDiseaseTree.predict(X\_testset)  
 **from** sklearn.metrics **import** f1\_score  
 dt\_f1 = f1\_score(y\_testset, dt\_yhat, average=**'weighted'**)  
 print(dt\_f1)  
 dt\_accuracy = accuracy\_score(y\_testset, dt\_yhat)  
 print(**"dt\_accuracy= "**,dt\_accuracy)  
 **from** sklearn.metrics **import** jaccard\_similarity\_score  
 dt\_ja = jaccard\_similarity\_score(y\_testset, dt\_yhat)  
 print(dt\_ja)  
 **from** sklearn.model\_selection **import** train\_test\_split  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=4)  
 print(**'Train Set:'**, X\_train.shape, y\_train.shape)  
 print(**'Test Set:'**, X\_test.shape, y\_test.shape)  
 Knn = 10  
 **from** sklearn.neighbors **import** KNeighborsClassifier  
 **from** sklearn **import** metrics  
 mean\_acc = np.zeros((Knn - 1))  
 std\_acc = np.zeros((Knn - 1))  
 confusionMx = [];  
 **for** n **in** range(1, Knn):  
 neigh = KNeighborsClassifier(n\_neighbors=n).fit(X\_train, y\_train)  
 knn\_hat = neigh.predict(X\_test)  
 mean\_acc[n - 1] = metrics.accuracy\_score(y\_test, knn\_hat)  
 std\_acc[n - 1] = np.std(knn\_hat == y\_test) / np.sqrt(knn\_hat.shape[0])  
 k\_cluster\_accuracy = mean\_acc.max()  
 print(**"The best accuracy was with"**, mean\_acc.max(), **"with k ="**, mean\_acc.argmax() + 1)  
 **from** sklearn.neighbors **import** KNeighborsClassifier  
 k\_nearest = KNeighborsClassifier(n\_neighbors=7).fit(X\_train, y\_train)  
 k\_nearest  
 **from** sklearn.metrics **import** f1\_score  
 knn\_f1 = f1\_score(y\_test, knn\_hat, average=**'weighted'**)  
 **from** sklearn.metrics **import** jaccard\_similarity\_score  
 knn\_ja = jaccard\_similarity\_score(y\_test, knn\_hat)  
 **from** sklearn.linear\_model **import** LogisticRegression  
 LogReg = LogisticRegression(C=0.01, solver=**'liblinear'**).fit(X\_trainset, y\_trainset)  
 LogReg  
 LogReg\_yhat = LogReg.predict(X\_test)  
 LogReg\_yhat[0:5]  
 LogReg\_yhat\_prob = LogReg.predict\_proba(X\_test)  
 LogReg\_yhat\_prob[0:5]  
  
 **from** sklearn.metrics **import** f1\_score  
 LR\_f1 = f1\_score(y\_test, LogReg\_yhat, average=**'weighted'**)  
  
 print(LR\_f1)  
 **from** sklearn.metrics **import** jaccard\_similarity\_score  
 LR\_ja = jaccard\_similarity\_score(y\_test, LogReg\_yhat)  
 print(LR\_ja)  
 **from** sklearn.metrics **import** log\_loss  
 LR\_log = log\_loss(y\_test, LogReg\_yhat\_prob)  
 print(LR\_log)  
  
 **from** sklearn **import** svm  
 **from** sklearn.metrics **import** f1\_score  
 **from** sklearn.metrics **import** jaccard\_similarity\_score  
  
 Svm = svm.SVC(kernel=**'rbf'**)  
 Svm.fit(X\_trainset, y\_trainset)  
 svm\_yhat = Svm.predict(X\_test)  
 **from** sklearn.metrics **import** f1\_score  
 svm\_f1 = f1\_score(y\_test, svm\_yhat, average=**'weighted'**)  
 svm\_accuracy = accuracy\_score(y\_test, svm\_yhat)  
 print(svm\_f1)  
 **from** sklearn.metrics **import** jaccard\_similarity\_score  
 svm\_ja = jaccard\_similarity\_score(y\_test, svm\_yhat)  
 print(svm\_ja)  
  
 **from** sklearn.naive\_bayes **import** GaussianNB  
 G\_NB = GaussianNB()  
 G\_NB.fit(X\_trainset, y\_trainset)  
 svm\_yhat = G\_NB.predict(X\_test)  
 **from** sklearn.metrics **import** f1\_score  
 svm\_f1 = f1\_score(y\_test, svm\_yhat, average=**'weighted'**)  
 Gnb\_accuracy = accuracy\_score(y\_test, svm\_yhat)  
 print(Gnb\_accuracy)  
  
 ja\_list = [knn\_ja, dt\_ja, svm\_ja, LR\_ja]  
 f1\_list = [knn\_f1, dt\_f1, svm\_f1, LR\_f1]  
 log\_list = [**'NA'**, **'NA'**, **'NA'**, LR\_log]  
  
 accuracyReport = pd.DataFrame(ja\_list, index=[**'KNN'**, **'Decision Tree'**, **'SVM'**, **'LogisticRegression'**])  
 accuracyReport.columns = [**'Jaccard'**]  
 accuracyReport.insert(loc=1, column=**'F1-score'**, value=f1\_list)  
 accuracyReport.insert(loc=2, column=**'LogLoss'**, value=log\_list)  
 accuracyReport.columns.name = **'Algorithm'** print(accuracyReport)  
 *#scores = [knn\_f1, dt\_f1, svm\_f1, LR\_f1]* scores = [dt\_accuracy, k\_cluster\_accuracy, knn\_ja, svm\_accuracy,Gnb\_accuracy]  
 algorithms = [**"Decision Tree"**, **"K Means"**, **"KNN"**, **"SVM"**,**"Naive Bayes"**]  
 plt.xlabel(**"Algorithms"**)  
 plt.ylabel(**"Accuracy score"**)  
 sns.barplot(algorithms,scores)  
 plt.show()  
 **return** scores

Deeplearnig Code:

**from** django.conf **import** settings  
**class** MyDeepLearning:  
 **def** startDeepLearningProcess(self):  
 print(**"Deep Learning Starting"**)  
 **from** keras.models **import** Sequential  
 **from** keras.layers **import** Dense  
 **import** pandas **as** pd  
 **from** sklearn.model\_selection **import** train\_test\_split  
 **from** sklearn.metrics **import** accuracy\_score  
 model = Sequential()  
 model.add(Dense(8, activation=**'relu'**, input\_dim=7))  
 model.add(Dense(1, activation=**'sigmoid'**))  
  
 path = settings.MEDIA\_ROOT + **"\\"** + **"Indian\_Liver\_Patients\_Dataset.csv"** liver\_data\_train = pd.read\_csv(path)  
 X = liver\_data\_train.iloc[:,2:9].values  
 y = liver\_data\_train.iloc[:,-1].values  
 X\_trainset, X\_testset, y\_trainset, y\_testset = train\_test\_split(X, y, test\_size=0.2, random\_state=5)  
  
  
 model.compile(loss=**'binary\_crossentropy'**, optimizer=**'adam'**, metrics=[**'accuracy'**])  
 *# model.fit(X\_train,Y\_train,epochs=300)* model.fit(X\_trainset, y\_trainset, epochs=100)  
 Y\_pred\_nn = model.predict(X\_testset)  
 Y\_pred\_nn.shape  
 print(Y\_pred\_nn.shape)  
 rounded = [round(x[0]) **for** x **in** Y\_pred\_nn]  
  
 Y\_pred\_nn = rounded  
 score\_nn = round(accuracy\_score(Y\_pred\_nn, y\_testset) \* 100, 2)  
 print(**"ANN accuracy Score "**, score\_nn)  
  
 *### J48 Implementation* **from** sklearn.tree **import** DecisionTreeClassifier  
 clf\_gini = DecisionTreeClassifier(criterion=**"gini"**,random\_state=100, max\_depth=3, min\_samples\_leaf=5)  
  
 *# Performing training* clf\_gini.fit(X\_trainset, y\_trainset)  
 y\_pred = clf\_gini.predict(X\_testset)  
 j48\_accuracy = accuracy\_score(y\_testset, y\_pred)  
 print(**"J48 Accuracy = "**,j48\_accuracy)  
  
 **return** score\_nn,j48\_accuracy

BackpropogationCode.py

**from** django.conf **import** settings  
**import** pandas **as** pd  
**import** matplotlib.pyplot **as** plt  
**import** numpy **as** np  
**class** MyBPAlgorithm:  
 **def** startBackProp(self):  
 path = settings.MEDIA\_ROOT + **"\\"** + **"liver.csv"  
 from** random **import** seed  
 **from** random **import** randrange  
 **from** random **import** random  
 **from** csv **import** reader  
 **from** math **import** exp, tanh  
 **import** matplotlib.pyplot **as** plt  
 **import** numpy **as** np  
  
 *# Initialize a network* **def** initialize\_network(n\_inputs, n\_hidden, n\_outputs):  
 *"""  
 Note: No layer is created for the input layer  
 """* network = list()  
 *# Create dictionary to store WEIGHTS as: [{'weights': [x x x] }, {...}, {'weights': [x x x]}]* hidden\_layer = [{**'weights'**: [random() **for** i **in** range(n\_inputs + 1)]} **for** i **in** range(n\_hidden)]  
 network.append(hidden\_layer)  
 output\_layer = [{**'weights'**: [random() **for** i **in** range(n\_hidden + 1)]} **for** i **in** range(n\_outputs)]  
 network.append(output\_layer)  
 **return** network  
 **def** initialize\_network\_custom(tab):  
 network = list()  
 **for** idx\_layer **in** range(1, len(tab)):  
 layer = []  
 *# For each neuron* **for** idx\_neuron **in** range(tab[idx\_layer]):  
 randomWeight = []  
 *# Create x weights + 1 (bias) for each neuron* **for** k **in** range(tab[idx\_layer - 1] + 1):  
 randomWeight.append(random())  
 *# Create dictionnary* temp = {**'weights'**: randomWeight}  
 layer.append(temp)  
 network.append(layer)  
 **return** network  
  
 **def** activate(weights, inputs):  
 *# Add the bias weight (last index)* activation = weights[-1]  
 *# Add other inputs\*weights linked to our neuron* **for** i **in** range(len(weights) - 1):  
 activation += weights[i] \* inputs[i]  
 **return** activation  
  
 *# Transfer neuron activation: sigmoid function or sigmoid' according to derivate arg* **def** transfer\_sigmoid(x, derivate):  
 **if** derivate == 0:  
 *# x = activation* **return** 1.0 / (1.0 + exp(-x))  
 **else**:  
 *# x = neuron outputs, calculate the derivative of an neuron output* **return** x \* (1.0 - x)  
  
 *# Transfer neuron activation: tanh function or tanh' according to derivate arg* **def** transfer\_tanh(x, derivate):  
 **if** derivate == 0:  
 **return** tanh(x)  
 **else**:  
 **return** 1.0 - tanh(x) \*\* 2  
  
 **def** forward\_propagate(network, row, transfer):  
 *# first input is set by the dataset array* inputs = row  
 **for** layer **in** network:  
 *# Array of neuron value after applying activate+Transfer (for 1 layer)* new\_inputs = []  
 **for** neuron **in** layer:  
 activation = activate(neuron[**'weights'**], inputs)  
 *# Add the neuron output into the neuron item (before 'weight')* neuron[**'output'**] = transfer(activation, 0)  
 new\_inputs.append(neuron[**'output'**])  
 inputs = new\_inputs  
 *# returns the outputs from the last layer* **return** inputs  
  
 *# Backpropagate error and store it into delta of neurons* **def** backward\_propagate\_error(network, expected, transfer):  
 *# Start from last layer* **for** idx\_layer **in** reversed(range(len(network))):  
 layer = network[idx\_layer]  
 errors = list()  
 *# --- (2) Error computed for the hidden layers: error = (weight\_k \* error\_j) \* transfer\_derivative(output)* **if** idx\_layer != len(network) - 1:  
 **for** idx\_neuron\_layer\_N **in** range(len(layer)):  
 error = 0.0  
 *# --- (A) error = Sum(delta \* weight linked to this delta)  
 # for each neuron[LAYER N+1] linked to this neuron[LAYER N] (current layer)* **for** neuron\_layer\_M **in** network[idx\_layer + 1]:  
 error += (neuron\_layer\_M[**'weights'**][idx\_neuron\_layer\_N] \* neuron\_layer\_M[**'delta'**])  
 errors.append(error)  
 *# --- (1) Error computed for the last layer: error = (expected - output) \* transfer\_derivative(output)* **else**:  
 *# --- (A) Store the difference between expected and output for each output neuron in errors[]* **for** idx\_neuron **in** range(len(layer)):  
 neuron = layer[idx\_neuron]  
 errors.append(expected[idx\_neuron] - neuron[**'output'**])  
 *# --- (B) Store the error signal in delta for each neuron* **for** idx\_neuron **in** range(len(layer)):  
 neuron = layer[idx\_neuron]  
 neuron[**'delta'**] = errors[idx\_neuron] \* transfer(neuron[**'output'**], 1)  
  
  
 **def** update\_weights(network, row, l\_rate):  
 **for** idx\_layer **in** range(len(network)):  
 inputs = row[:-1]  
 **if** idx\_layer != 0:  
 *# --- (1) Store the outputs of the layer N-1 into inputs[]* inputs = [neuron[**'output'**] **for** neuron **in** network[idx\_layer - 1]]  
 **for** neuron **in** network[idx\_layer]:  
 *# --- (2) Compute the new weights for each neuron of the layer N* **for** idx\_input **in** range(len(inputs)):  
 neuron[**'weights'**][idx\_input] += l\_rate \* neuron[**'delta'**] \* inputs[idx\_input]  
 *# --- (3) Update the bias of the neuron (input=1 below)* neuron[**'weights'**][-1] += l\_rate \* neuron[**'delta'**] \* 1  
  
 *# expected = one-hot encoding, one class for one output (one output = unic binary value)  
 # if 2 classes (2 integer out values), expected = [0, 1] and = [1, 0]  
 # write '1' at the index given by the integer output (if output = 2, set at the index 2: [0,0,1] )* **def** one\_hot\_encoding(n\_outputs, row\_in\_dataset):  
 expected = [0 **for** i **in** range(n\_outputs)]  
 expected[row\_in\_dataset[-1]] = 1  
 **return** expected  
  
 *# Train a network for a fixed number of epochs, it is updated using stochastic gradient descent.  
 # input: (network)Our neural network  
 # input: (train)Dataset to train the neural network  
 # input: (l\_rate)learning rate controls how much to change the weight to correct for the error.  
 # For example, a value of 0.1 will update the weight 10% of the amount that it possibly could be updated.  
 # input: (n\_epoch)Within each epoch, update the network for each row in the training dataset  
 # input: (n\_outputs)* **def** train\_network(network, train, test, l\_rate, n\_epoch, n\_outputs, transfer):  
 accuracy = []  
 **for** epoch **in** range(n\_epoch):  
 sum\_error = 0  
 *# Apply for each row of the dataset the backprop* **for** row **in** train:  
 outputs = forward\_propagate(network, row, transfer)  
 expected = one\_hot\_encoding(n\_outputs, row)  
 sum\_error += sum([(expected[i] - outputs[i]) \*\* 2 **for** i **in** range(len(expected))])  
 backward\_propagate\_error(network, expected, transfer)  
 update\_weights(network, row, l\_rate)  
 accuracy.append(get\_prediction\_accuracy(network, test, transfer))  
 accuracies.append(accuracy)  
  
 *# Make a prediction with a network* **def** predict(network, row, transfer):  
 outputs = forward\_propagate(network, row, transfer)  
 *# return the index with max value for each output (ex: output[i]=[0.1, 0.9, 0.2], prediciont[i] = 1)* **return** outputs.index(max(outputs))  
  
 **def** get\_prediction\_accuracy(network, train, transfer):  
 predictions = list()  
 **for** row **in** train:  
 prediction = predict(network, row, transfer)  
 predictions.append(prediction)  
 expected\_out = [row[-1] **for** row **in** train]  
 accuracy = accuracy\_metric(expected\_out, predictions)  
 **return** accuracy  
  
 *# Backpropagation Algorithm With Stochastic Gradient Descent* **def** back\_propagation(train, test, l\_rate, n\_epoch, n\_hidden, transfer):  
 n\_inputs = len(train[0]) - 1  
 n\_outputs = len(set([row[-1] **for** row **in** train]))  
 *# network = initialize\_network(n\_inputs, n\_hidden, n\_outputs)* network = initialize\_network\_custom([n\_inputs, 5, n\_outputs])  
 layerPrint = []  
 **for** i **in** range(len(network)):  
 layerPrint.append(len(network[i]))  
 print(**'network created: %d layer(s):'** % len(network), layerPrint)  
 train\_network(network, train, test, l\_rate, n\_epoch, n\_outputs, transfer)  
 predictions = list()  
 print(**"perform predictions on %d set of inputs:"** % len(test))  
 **for** row **in** test:  
 prediction = predict(network, row, transfer)  
 predictions.append(prediction)  
 print(**"pred ="**, predictions)  
 **return** (predictions)  
  
 *# Load a CSV file* **def** load\_csv(filename):  
 dataset = list()  
 **with** open(filename, **'r'**) **as** file:  
 csv\_reader = reader(file)  
 **for** row **in** csv\_reader:  
 **if not** row:  
 **continue** dataset.append(row)  
 **return** dataset  
  
 *# Convert string column to float* **def** str\_column\_to\_float(dataset, column):  
 **for** row **in** dataset:  
 row[column] = float(row[column].strip())  
  
 *# Convert string column to integer* **def** str\_column\_to\_int(dataset, column):  
 class\_values = [row[column] **for** row **in** dataset]  
 unique = set(class\_values)  
 lookup = dict()  
 **for** i, value **in** enumerate(unique):  
 lookup[value] = i  
 **for** row **in** dataset:  
 row[column] = lookup[row[column]]  
 **return** lookup  
  
 *# Find the min and max values for each column* **def** dataset\_minmax(dataset):  
 minmax = list()  
 stats = [[min(column), max(column)] **for** column **in** zip(\*dataset)]  
 **return** stats  
  
 *# Rescale dataset columns to the range 0-1* **def** normalize\_dataset(dataset, minmax):  
 **for** row **in** dataset:  
 **for** i **in** range(len(row) - 1):  
 row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])  
  
 *# Split a dataset into k folds* **def** cross\_validation\_split(dataset, n\_folds):  
 dataset\_split = list()  
 dataset\_copy = list(dataset)  
 fold\_size = int(len(dataset) / n\_folds)  
 **for** i **in** range(n\_folds):  
 fold = list()  
 **while** len(fold) < fold\_size:  
 index = randrange(len(dataset\_copy))  
 fold.append(dataset\_copy.pop(index))  
 dataset\_split.append(fold)  
 **return** dataset\_split  
  
 *# Calculate accuracy percentage* **def** accuracy\_metric(actual, predicted):  
 correct = 0  
 **for** i **in** range(len(actual)):  
 **if** actual[i] == predicted[i]:  
 correct += 1  
 **return** correct / float(len(actual)) \* 100.0  
  
 *# Evaluate an algorithm using a cross validation split  
 # input (dataset)train patterns  
 # input (algorithm)function to init and train the network + perform predictions  
 # input (n\_folds)dataset spliter number  
 # input (\*args) = (l\_rate, n\_epoch, n\_hidden)* **def** evaluate\_algorithm(dataset, algorithm, n\_folds, \*args):  
 *# k-fold cross-validation* folds = cross\_validation\_split(dataset, n\_folds)  
 scores = list()  
 **for** fold **in** folds:  
 *# --- (1) Prepare inputs to train the network* train\_set = list(folds) *# Set all dataset* train\_set.remove(fold) *# Remove the fold used to test the network* train\_set = sum(train\_set, [])  
 *# --- (2) Prepare inputs to test the network (to make predictions)* test\_set = list()  
 **for** row **in** fold: *# Store each row of fold (test) in test\_set[]* row\_copy = list(row)  
 test\_set.append(row\_copy)  
 row\_copy[-1] = **None** *# --- (3) create network + train it + return its predictions* predicted = algorithm(train\_set, fold, \*args)  
 *# --- (4) compare each prediction with the corresponding expected output in dataset* actual = [row[-1] **for** row **in** fold]  
 print(actual)  
 accuracy = accuracy\_metric(actual, predicted)  
 scores.append(accuracy)  
 print(**'- Training[%d] performed'** % len(scores))  
 print(**'---------------------------------------'**)  
 **return** scores  
  
 seed(1)  
  
 accuracies = list()  
  
  
 filename = path *#'liver.csv'* dataset = load\_csv(filename)  
 *# convert string numbers to floats* **for** i **in** range(len(dataset[0]) - 1):  
 str\_column\_to\_float(dataset, i)  
 *# convert class column to integers* str\_column\_to\_int(dataset, len(dataset[0]) - 1)  
 *# normalize input variables to the range of 0 and 1 (range of the transfer function)* minmax = dataset\_minmax(dataset)  
 normalize\_dataset(dataset, minmax)  
 *# evaluate algorithm:  
 # 5-fold cross validation is used.  
 # That means that 5 models are fit and evaluated on 5 different hold out sets. Each model is trained for 500 epochs.* n\_folds = 5  
 l\_rate = 0.3  
 n\_epoch = 500  
 n\_hidden = 5  
 print(**'---------------------------------------'**)  
 scores = evaluate\_algorithm(dataset, back\_propagation, n\_folds, l\_rate, n\_epoch, n\_hidden, transfer\_sigmoid)  
 print(**'Scores (per fold): %s'** % scores)  
 bp\_accuracy = (sum(scores) / float(len(scores)))  
 print(**'Mean Accuracy: %.3f%%'** % (sum(scores) / float(len(scores))))  
 subplot = ((n\_folds / 2) + (n\_folds % 2)) \* 100 + 21  
 **for** i **in** range(len(accuracies)):  
 plt.subplot(subplot + i)  
 plt.plot(accuracies[i])  
 plt.grid(**True**)  
 plt.ylabel(**'Accuracy (%)'**)  
 plt.xlabel(**'epoch number'**)  
 plt.show()  
 **return** bp\_accuracy

All Urls.py

*"""LiverDisease URL Configuration  
  
The `urlpatterns` list routes URLs to views. For more information please see:  
 https://docs.djangoproject.com/en/2.0/topics/http/urls/  
Examples:  
Function views  
 1. Add an import: from my\_app import views  
 2. Add a URL to urlpatterns: path('', views.home, name='home')  
Class-based views  
 1. Add an import: from other\_app.views import Home  
 2. Add a URL to urlpatterns: path('', Home.as\_view(), name='home')  
Including another URLconf  
 1. Import the include() function: from django.urls import include, path  
 2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))  
"""***from** django.contrib **import** admin  
**from** django.urls **import** path  
  
**from** django.contrib **import** admin  
**from** django.urls **import** path  
**from** LiverDisease **import** views **as** mainView  
**from** users **import** views **as** usr  
**from** admins **import** views **as** admins  
  
urlpatterns = [  
 path(**'admin/'**, admin.site.urls),  
 path(**""**, mainView.index, name=**'index'**),  
 path(**"index/"**, mainView.index, name=**"index"**),  
 path(**"logout/"**, mainView.logout, name=**"logout"**),  
 path(**"UserLogin/"**, mainView.UserLogin, name=**"UserLogin"**),  
 path(**"AdminLogin/"**, mainView.AdminLogin, name=**"AdminLogin"**),  
 path(**"UserRegister/"**, mainView.UserRegister, name=**"UserRegister"**),  
 path(**"workingProcess/"**, mainView.workingProcess, name=**"workingProcess"**),  
  
 *### User Side Views* path(**"UserRegisterActions/"**, usr.UserRegisterActions, name=**"UserRegisterActions"**),  
 path(**"UserLoginCheck/"**, usr.UserLoginCheck, name=**"UserLoginCheck"**),  
 path(**"UserHome/"**, usr.UserHome, name=**"UserHome"**),  
 path(**"UserViewDataset/"**, usr.UserViewDataset, name=**"UserViewDataset"**),  
 path(**"UserResults/"**, usr.UserResults, name=**"UserResults"**),  
 path(**"UserClassification/"**, usr.UserClassification, name=**"UserClassification"**),  
 path(**"UserNeuralNetwork/"**, usr.UserNeuralNetwork, name=**"UserNeuralNetwork"**),  
  
  
 *### Admin Side Views* path(**"AdminLoginCheck/"**, admins.AdminLoginCheck, name=**"AdminLoginCheck"**),  
 path(**"AdminHome/"**, admins.AdminHome, name=**"AdminHome"**),  
 path(**"ViewRegisteredUsers/"**, admins.ViewRegisteredUsers, name=**"ViewRegisteredUsers"**),  
 path(**"AdminActivaUsers/"**, admins.AdminActivaUsers, name=**"AdminActivaUsers"**),  
 path(**"adminClassification/"**,admins.adminClassification, name=**"adminClassification"**),  
 path(**"adminNeuralNetwork/"**, admins.adminNeuralNetwork, name=**"adminNeuralNetwork"**),  
  
]

Base.html

<!doctype **html**>  
{%load static%}  
<**html lang="en"**>  
  
<**head**>  
 *<!-- Required meta tags -->* <**meta charset="utf-8"**>  
 <**meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no"**>  
 <**link rel="icon" href="{%static 'img/favicon.png'%}" type="image/png"**>  
 <**title**>Medicare Medical</**title**>  
 *<!-- Bootstrap CSS -->* <**link rel="stylesheet" href="{%static 'css/bootstrap.css'%}"**>  
 <**link rel="stylesheet" href="{%static 'vendors/linericon/style.css'%}"**>  
 <**link rel="stylesheet" href="{%static 'css/font-awesome.min.css'%}"**>  
 <**link rel="stylesheet" href="{%static 'vendors/owl-carousel/owl.carousel.min.css'%}"**>  
 <**link rel="stylesheet" href="{%static 'vendors/lightbox/simpleLightbox.css'%}"**>  
 <**link rel="stylesheet" href="{%static 'vendors/nice-select/css/nice-select.css'%}"**>  
 <**link rel="stylesheet" href="{%static 'vendors/animate-css/animate.css'%}"**>  
 *<!-- main css -->* <**link rel="stylesheet" href="{%static 'css/style.css'%}"**>  
</**head**>  
  
<**body**>  
  
 *<!--================ Start Header Menu Area =================-->* <**header class="header\_area"**>  
 <**div class="header-top"**>  
  
 </**div**>  
 <**div class="main\_menu"**>  
 <**div class="search\_input" id="search\_input\_box"**>  
 <**div class="container"**>  
 <**form class="d-flex justify-content-between"**>  
 <**input type="text" class="form-control" id="search\_input" placeholder="Search Here"**>  
 <**button type="submit" class="btn"**></**button**>  
 <**span class="lnr lnr-cross" id="close\_search" title="Close Search"**></**span**>  
 </**form**>  
 </**div**>  
 </**div**>  
 <**nav class="navbar navbar-expand-lg navbar-light"**>  
 <**div class="container"**>  
 *<!-- Brand and toggle get grouped for better mobile display -->* <**a class="navbar-brand logo\_h" href="{%url 'index'%}"**><**h2**>Diagnosis of Liver Disease</**h2**></**a**>  
 <**button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent"  
 aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation"**>  
 <**span class="icon-bar"**></**span**>  
 <**span class="icon-bar"**></**span**>  
 <**span class="icon-bar"**></**span**>  
 </**button**>  
 *<!-- Collect the nav links, forms, and other content for toggling -->* <**div class="collapse navbar-collapse offset" id="navbarSupportedContent"**>  
 <**ul class="nav navbar-nav menu\_nav ml-auto"**>  
 <**li class="nav-item"**><**a class="nav-link" href="{%url 'index'%}"**>Home</**a**></**li**>  
 <**li class="nav-item"**><**a class="nav-link" href="{%url 'UserLogin'%}"**>Users</**a**></**li**>  
 <**li class="nav-item"**><**a class="nav-link" href="{%url 'AdminLogin'%}"**>Admin</**a**></**li**>  
 <**li class="nav-item"**><**a class="nav-link" href="{%url 'UserRegister'%}"**>Registrations</**a**></**li**>  
 </**ul**>  
 </**div**>  
 </**div**>  
 </**nav**>  
 </**div**>  
 </**header**>  
 {%block contents%}  
  
 {%endblock%}  
  
  
  
 <**footer class="footer-area section-gap"**>  
  
 <**div class="footer-bottom"**>  
 <**div class="container"**>  
 <**div class="d-flex justify-content-between align-items-center flex-wrap"**>  
 <**p class="footer-text"**>*<!-- Link back to Colorlib can't be removed. Template is licensed under CC BY 3.0. -->*Copyright **&copy;**<**script**>document.write(new Date().getFullYear());</**script**> All rights reserved | This template is made with <**i class="fa fa-heart-o" aria-hidden="true"**></**i**> by Alex Hales  
*<!-- Link back to Colorlib can't be removed. Template is licensed under CC BY 3.0. -->*</**p**>  
  
 </**div**>  
 </**div**>  
 </**div**>  
 </**footer**>  
 *<!--================ End footer Area =================-->  
  
 <!-- Optional JavaScript -->  
 <!-- jQuery first, then Popper.js, then Bootstrap JS -->* <**script src="{%static 'js/jquery-3.2.1.min.js'%}"**></**script**>  
 <**script src="{%static 'js/popper.js'%}"**></**script**>  
 <**script src="{%static 'js/bootstrap.min.js'%}"**></**script**>  
 <**script src="{%static 'js/stellar.js'%}"**></**script**>  
 <**script src="{%static 'vendors/lightbox/simpleLightbox.min.js'%}"**></**script**>  
 <**script src="{%static 'vendors/nice-select/js/jquery.nice-select.min.js'%}"**></**script**>  
 <**script src="{%static 'vendors/owl-carousel/owl.carousel.min.js'%}"**></**script**>  
 <**script src="{%static 'js/jquery.ajaxchimp.min.js'%}"**></**script**>  
 <**script src="{%static 'vendors/counter-up/jquery.waypoints.min.js'%}"**></**script**>  
 <**script src="{%static 'vendors/counter-up/jquery.counterup.js'%}"**></**script**>  
 <**script src="{%static 'js/mail-script.js'%}"**></**script**>  
 *<!--gmaps Js-->* <**script src="https://maps.googleapis.com/maps/api/js?key=AIzaSyCjCGmQ0Uq4exrzdcL6rvxywDDOvfAu6eE"**></**script**>  
 <**script src="{%static 'js/gmaps.min.js'%}"**></**script**>  
 <**script src="{%static 'js/theme.js'%}"**></**script**>  
</**body**>  
  
</**html**>

UserRegistration.html

{%extends 'base.html'%}  
{%block contents%}  
 <**section class="home\_banner\_area"**>  
 <**div class="banner\_inner"**>  
 <**div class="container"**>  
 <**div class="banner\_content"**>  
 <**h2**>User Register Form</**h2**>  
 <**p**>  
 <**form action="{%url 'UserRegisterActions'%}" method="POST" style="**width:100%**"**>  
  
 {% csrf\_token %}  
 <**table**>  
 <**tr**><**td**></**td**>  
 <**td**>User Name</**td**>  
 <**td**>{{form.name}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**>Login ID</**td**>  
 <**td**>{{form.loginid}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**>Password</**td**>  
 <**td**>{{form.password}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**>Mobile</**td**>  
 <**td**>{{form.mobile}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**>email</**td**>  
 <**td**>{{form.email}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**>Locality</**td**>  
 <**td**>{{form.locality}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**>Address</**td**>  
 <**td**>{{form.address}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**>City</**td**>  
 <**td**>{{form.city}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**>State</**td**>  
 <**td**>{{form.state}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**></**td**>  
 <**td**>{{form.status}}</**td**>  
 </**tr**>  
 <**tr**><**td**></**td**>  
 <**td**></**td**>  
 <**td**><**button type="submit" value="Register" class="primary-btn text-uppercase"**>Register</**button**></**td**>  
 </**tr**>  
 <**tr**>  
 <**td**>  
 <**div class="form-group mt-3"**>  
 <**span** >**&nbsp;**</**span**>  
 </**div**>  
 </**td**>  
 </**tr**>  
  
 {% if messages %}  
 {% for message in messages %}  
 <**font color='WHITE'**> {{ message }}</**font**>  
 {% endfor %}  
 {% endif %}  
  
 </**table**>  
  
 </**form**>  
 </**p**>  
 *<!--<a class="primary-btn text-uppercase" href="#">Learn More</a>-->* </**div**>  
 </**div**>  
 </**div**>  
 </**section**>  
 {%endblock%}