Connect-4 Neural Shrub-Classes

July 11, 2019

In [1]:
```
import time

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

from keras.models import Sequential
from keras.layers import Dense
from sklearn.preprocessing import LabelEncoder
from keras.utils import np_utils

import numpy as np
```

```
/home/shashwati/anaconda3/envs/py35/lib/python3.5/site-packages/h5py/__init__.py:36: FutureWarn
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

In [2]:
```
# fix random seed for reproducibility
seed = 7
np.random.seed(seed)
```

In [3]:
```
# function returns the data in the right format
def get_data():
    dataset = np.genfromtxt("connect-4.csv", dtype='str', delimiter=",")

    preX = dataset[:,0:42]
    preY = dataset[:,42]

    X = np.zeros(preX.shape)

    for i, row in enumerate(preX):
        for j, col in enumerate(row):
            if col == 'x':
                X[i,j] = 1.0
            if col == 'o':
                X[i,j] = -1.0
```

```
                if col == 'b':
                    X[i,j] = 0.0


        encoder = LabelEncoder()
        # code: 0 - draw; 1 - loss; 2 -win
        encoded_Y = encoder.fit_transform(preY)


        # splitting the dataset into 80% training and 20% test data set
        train, test, label_train, label_test = \
                train_test_split(X, encoded_Y,test_size = 0.2)

        return train, label_train, test, label_test
```

In [4]:
```
from sklearn.tree._tree import TREE_LEAF

def prune_index(inner_tree, index, threshold):
    if inner_tree.value[index].min() < threshold:
        # turn node into a leaf by "unlinking" its children
        inner_tree.children_left[index] = TREE_LEAF
        inner_tree.children_right[index] = TREE_LEAF
    # if there are shildren, visit them as well
    if inner_tree.children_left[index] != TREE_LEAF:
        prune_index(inner_tree, inner_tree.children_left[index], threshold)
        prune_index(inner_tree, inner_tree.children_right[index], threshold)
```

In [5]:
```
# builds the decision tree of depth 12
def decision_tree(train, label):
    dt = DecisionTreeClassifier(max_depth = 12, min_samples_leaf=100)
    dt.fit(train, label)
    prune_index(dt.tree_, 0, 5)
    end = time.time()
    return dt
```

In [6]:
```
# builds the neural network for a given class
def neural_network(class_data):
    num_train = []
    num_label = []
    for x in class_data:
        num_train.append(x[0])
        num_label.append(x[1])

    num_train = np.array(num_train)
    num_label = np.array(num_label)

    # converting categorical variable into numerical values
    encoder = LabelEncoder()
```

```
    encoder.fit(num_label)

    # code: 0 - draw; 1 - loss; 2 -win
    encoded_Y = encoder.transform(num_label)
    final_label = np_utils.to_categorical(encoded_Y, 3)

    model = Sequential()
    model.add(Dense(8, input_dim=42, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', \
                  optimizer='adam', metrics=['accuracy'])
    model.fit(num_train, final_label, epochs=5, batch_size=5)
    return model
```

```
In [7]: # builds the neural shrub
    def neural_shrubs(tree, train, label):
        train = np.array(train)
        label = np.array(label)

        # leave_id: index of the leaf that cantains the instance
        leave_id = tree.apply(train)

        num_class = 3
        classes = [[] for i in range(0, num_class)]

        for x in range(len(train)):
            leaf = leave_id[x]

            # Gets the class for each leaf
            #.value: returns the distributition at the leaf,
            #        i.e number of instance in each class at that leaf
            #.argmax(): returns the class which has the max instance
            #        i.e here: (0, 1, 2) - it is 0-indexed
            idx = np.array(tree.tree_.value[leaf]).argmax()

            # insert the instance into the class
            classes[idx].append([train[x], label[x]])

        # stores the neural network for each class
        nn_models = []

        #stores the max time taken to build a neural network
        max_time = 0;

        for x in range(num_class):

            start = time.time()
            model = neural_network(classes[x])
```

```
            end = time.time()

            time_taken = end - start
            if max_time < time_taken:
                max_time = time_taken

            nn_models.append(model)

        # returns a neural network for each class and the max
        # time taken to build the neural network
        return nn_models, max_time
```

In [8]:
```
# The algorithm to build the neural shrub
train, train_label, test, test_label = get_data()

dt_start = time.time()
tree = decision_tree(train, train_label)
dt_end = time.time()

shrubs, max_time = neural_shrubs(tree, train, train_label)
```

```
Epoch 1/5
211/211 [==============================] - 0s 2ms/step - loss: 1.1439 - acc: 0.3175
Epoch 2/5
211/211 [==============================] - 0s 467us/step - loss: 1.1121 - acc: 0.3318
Epoch 3/5
211/211 [==============================] - 0s 402us/step - loss: 1.0909 - acc: 0.3412
Epoch 4/5
211/211 [==============================] - 0s 429us/step - loss: 1.0761 - acc: 0.3697
Epoch 5/5
211/211 [==============================] - 0s 472us/step - loss: 1.0619 - acc: 0.3981
Epoch 1/5
12442/12442 [==============================] - 5s 401us/step - loss: 0.8540 - acc: 0.6231
Epoch 2/5
12442/12442 [==============================] - 5s 378us/step - loss: 0.7480 - acc: 0.6857
Epoch 3/5
12442/12442 [==============================] - 5s 383us/step - loss: 0.7267 - acc: 0.6926
Epoch 4/5
12442/12442 [==============================] - 5s 379us/step - loss: 0.7152 - acc: 0.6968
Epoch 5/5
12442/12442 [==============================] - 5s 387us/step - loss: 0.7084 - acc: 0.7009
Epoch 1/5
41392/41392 [==============================] - 17s 421us/step - loss: 0.5751 - acc: 0.7783
Epoch 2/5
41392/41392 [==============================] - 17s 405us/step - loss: 0.5110 - acc: 0.7998
Epoch 3/5
41392/41392 [==============================] - 17s 406us/step - loss: 0.5002 - acc: 0.8026
Epoch 4/5
```

```
41392/41392 [==============================] - 17s 402us/step - loss: 0.4935 - acc: 0.8048
Epoch 5/5
41392/41392 [==============================] - 17s 403us/step - loss: 0.4885 - acc: 0.8071
```

In [9]:
```
# predicts using the neural shrub
def neural_shrub_predict(tree, nn_model, test, label):
    label_test = np.array(label)
    test = np.array(test)

    #row - actual; col - pred
    confusion_matrix = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
    correct = 0

    for i in range(len(test)):
        x = test[i]
        pred_class = tree.predict([x])
        x = np.array([x])
        pred = pred_class[0]
        nn_model_class = nn_model[pred_class[0]]
        pred = np.argmax(nn_model_class.predict(x))

        confusion_matrix[label[i]][pred] = \
            confusion_matrix[label[i]][pred] + 1
        if pred == label[i]: correct = correct + 1

    acc_score = correct/len(test)

    return confusion_matrix, acc_score
```

In [10]:
```
# Predicting
cm, acc_score = neural_shrub_predict(tree, shrubs, test, test_label)
print("Confusion Matrix:\n\n", cm)
```

```
Confusion Matrix:

 [[  62  345  838]
 [  46 2261 1083]
 [  35  674 8168]]
```

In [11]:
```
# function used to calcultate the metrics for each class
def metrics(cm, cls, size):
    cm = np.array(cm)
    tp = cm[cls][cls]
    fp = sum(cm[x, cls] for x in range(3))-cm[cls][cls]
    fn = sum(cm[cls, x] for x in range(3))-cm[cls][cls]
    tn = size - tp - fp - fn
    precision = tp/(tp+fp)
```

```
            recall = tp/(tp+fn)
            fmeasure = 2*(precision*recall)/(precision + recall)
            accuracy = (tp + tn)/size

            return precision, recall, fmeasure, accuracy
```

```
In [12]: # metrics for class 0 (draw)
         precision0, recall0, f0, acc0 = metrics(cm, 0, len(test))
         print("                Precision Recall F-measure Accuracy")
         print("Class 0 (draw): ", round(precision0, 3), "  ", round(recall0, 3), \
               " ", round(f0, 3), "   ", round(acc0,3))
```

```
                Precision Recall F-measure Accuracy
Class 0 (draw):  0.434    0.05   0.089     0.906
```

```
In [13]: # metrics for class 1 (lose)
         precision1, recall1, f1, acc1 = metrics(cm, 1, len(test))
         print("                Precision Recall F-measure Accuracy")
         print("Class 1 (loss): ", round(precision1, 3), "  ", round(recall1, 3), \
               " ", round(f1, 3), "   ", round(acc1,3))
```

```
                Precision Recall F-measure Accuracy
Class 1 (loss):  0.689    0.667   0.678     0.841
```

```
In [14]: # metrics for class 2 (win)
         precision2, recall2, f2, acc2 = metrics(cm, 2, len(test))
         print("                Precision Recall F-measure Accuracy")
         print("Class 2 (win): ", round(precision2, 3), "  ", round(recall2, 3), \
               " ", round(f2, 3), "   ", round(acc2,3))
```

```
                Precision Recall F-measure Accuracy
Class 2 (win):  0.81    0.92   0.861     0.805
```

```
In [15]: # average metrics
         avg_p = (precision0 + precision1 + precision2)/3.0
         avg_r = (recall0 + recall1 + recall2) / 3.0
         avg_f = (f0 + f1 + f2) / 3.0
         avg_a = (acc0 + acc1 + acc2)/ 3.0
         print("        Precision Recall F-measure Accuracy")
         print("Average: ", round(avg_p, 3), "  ", round(avg_r, 3), \
               " ", round(avg_f, 3), "   ", round(avg_a,3))
```

```
        Precision Recall F-measure Accuracy
Average:  0.644    0.546   0.543     0.851
```

```
In [16]: # training time
         total_time_taken = dt_end - dt_start + max_time
         print("Training Time: %s sec" % round(total_time_taken, 5))
```

```
Training Time: 84.82363 sec
```

```
In [17]: # Number of instances correctly classified
         print("Accuracy_score: ", round(acc_score, 5))
```

```
Accuracy_score:  0.77642
```