hw5 - lydiayu

April 14, 2022

[37]:
```
%env IAI_DISABLE_COMPILED_MODULES=True
```

```
env: IAI_DISABLE_COMPILED_MODULES=True
```

[38]:
```
# import interpretableai
# interpretableai.install_julia()
# interpretableai.install_system_image()
```

[39]:
```
from interpretableai import iai
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as mse
```

0.0.1 Problem 1

a)

[40]:
```
admission = pd.read_csv("admission.csv")
```

[41]:
```
admission = admission.rename(columns={"Chance of Admit ": "Chance of Admit"})
```

[42]:
```
y = admission["Chance of Admit"]
X = admission.drop(columns=["Serial No.", "Chance of Admit"])
```

[43]:
```
(train_X, train_y), (test_X, test_y) = iai.split_data('regression', X, y,␣
↪train_proportion = 0.8, seed=1)
```

b)

[44]:
```
grid = iai.GridSearch(
    iai.OptimalTreeRegressor(
        random_seed=1,
    ),
    max_depth=range(1, 7),
    minbucket=range(1, 5),
)
grid.fit(train_X, train_y)
```

```
grid.get_learner()
```

[44]:
```
Fitted OptimalTreeRegressor:
  1) Split: CGPA < 8.325
    2) Split: GRE Score < 309.5
      3) Split: CGPA < 7.885
        4) Predict: 0.5005, 42 points, error 0.005819
        5) Predict: 0.6134, 47 points, error 0.005882
      6) Split: LOR  < 2.75
        7) Predict: 0.5767, 18 points, error 0.006011
        8) Predict: 0.6748, 31 points, error 0.002399
    9) Split: CGPA < 9.055
      10) Split: CGPA < 8.655
        11) Predict: 0.7062, 77 points, error 0.003852
        12) Predict: 0.7695, 85 points, error 0.0034
      13) Split: CGPA < 9.225
        14) Predict: 0.8444, 39 points, error 0.002148
        15) Predict: 0.9267, 61 points, error 0.0006811
```

Because there are only 500 observations, `max_depth` should not be too high because otherwise the model would be overfitting (eg even a depth of 9 would be too much because $2^9 = 512$. I would also expect `minbucket` to not be very low because there would be a risk of overfitting. If `max_depth` is too high and `minbucket` is too low, then the model is likely overfitting; if `max_depth` is too low and `minbucket` is too high, then the model is likely not detailed enough.

The best parameter combination seems to be `max_depth` $= 3$ (3 edges max from root to leaf) and `minbucket` $= 2$.

[45]:
```
# predicted values for test and train sets
pred_test = grid.predict(test_X)
pred_train = grid.predict(train_X)
```

[46]:
```
# TRAIN MSE
mse(y_true = train_y, y_pred = pred_train)
```

[46]:
```
0.0035357804855505903
```

[47]:
```
# TEST MSE
mse(y_true = test_y, y_pred = pred_test)
```

[47]:
```
0.006916174584340078
```

c) Interpret the trees by detailing the variables in the splits, the position of each splits (i.e. root/leaf), as well as the values of each feature split (i.e. Split on GRE Score 300). Use IAI.variableimportance(lnr) to report the most important features. What have you observed that are the three most important factor of graduate admission? Do you think the admission metric has anything that can be improved upon?

The first split occurs on whether CGPA is greater than 8.325. - If CGPA is less than 8.325, the tree then splits on whether GRE score is greater than 309.5. - If GRE score is less than 309.5, the tree splits on whether CGPA is greater than 7.885. - The chances of admission are 0.5005 if CGPA is less than 7.885 and 0.6134 if CGPA is greater than or equal to 7.885. - If GRE score is greater than or equal to 309.5, the tree splits on whether LOR is greater than 2.75. - The chances of admission are 0.5767 if the LOR is less than 2.75 and 0.6748 if the LOR is greater than or equal to 2.75. - If CGPA is greater than or equal to 8.325, the tree splits on whether CGPA is greater than 9.055. - If CGPA is less than 9.055, the tree splits on whether CGPA is greater than 8.655. - The chances of admission are 0.7062 if the CGPA is less than 8.655 and 0.7695 if the CGPA is greater than or equal to 8.655. - If CGPA is greater than or equal to 9.055, the tree splits on whether CGPA is greater than 9.225. - The chances of admission are 0.8444 if the CGPA is less than 9.225 and 0.9267 if the CGPA is greater than or equal to 9.225.

[48]:
```
grid.get_learner().variable_importance()
```

[48]:

	Feature	Importance
0	CGPA	0.765328
1	GRE Score	0.214815
2	TOEFL Score	0.010849
3	LOR	0.006057
4	Research	0.002952
5	SOP	0.000000
6	University Rating	0.000000

The three most important features are CGPA, GRE Score, and TOEFL Score. Based on the tree, it seems that for anyone with a higher CGPA (above 8.325), the only metric the model uses to evaluate their chances of admission is CGPA. This could be oversimplifying the admissions process and leaving out important features that play a role in admissions for students with higher CGPAs. Also, even though TOEFL Score is one of the top three most important features, it is not used in any of the splits of the tree, while LOR is used even though it is not as important.

d)

[49]:
```
X = X[["GRE Score", "CGPA"]]
```

[50]:
```
(train_X, train_y), (test_X, test_y) = iai.split_data('regression', X, y,␣
↪train_proportion = 0.8, seed=1)
```

[51]:
```
grid = iai.GridSearch(
    iai.OptimalTreeRegressor(
        random_seed=1,
    ),
    max_depth=3,
    minbucket=2,
)
grid.fit(train_X, train_y)
grid.get_learner()
```

[51]:
```
Fitted OptimalTreeRegressor:
  1) Split: CGPA < 8.645
    2) Split: GRE Score < 306.5
      3) Split: CGPA < 8.005
        4) Predict: 0.508, 50 points, error 0.006684
        5) Predict: 0.6409, 32 points, error 0.001958
      6) Split: CGPA < 8.325
        7) Predict: 0.634, 67 points, error 0.006099
        8) Predict: 0.7132, 62 points, error 0.004067
    9) Split: CGPA < 9.055
      10) Predict: 0.7678, 89 points, error 0.003341
      11) Split: CGPA < 9.225
        12) Predict: 0.8444, 39 points, error 0.002148
        13) Predict: 0.9267, 61 points, error 0.0006811
```

[52]:
```
# predicted values for test and train sets
pred_test = grid.predict(test_X)
pred_train = grid.predict(train_X)
```

[53]:
```
# TRAIN MSE
mse(y_true = train_y, y_pred = pred_train)
```

[53]:
```
0.003700731131149985
```

[54]:
```
# TEST MSE
mse(y_true = test_y, y_pred = pred_test)
```

[54]:
```
0.007299861487725458
```

e)

[55]:
```
# assuming we are using the same train/test data as in part d) (e.g. only GRE␣
↪and CGPA)
grid = iai.GridSearch(
    iai.OptimalTreeRegressor(
        random_seed=1,
        hyperplane_config={'sparsity': 'all'},
    ),
    max_depth=3,
    minbucket=2
)
grid.fit(train_X, train_y)
grid.get_learner()
```

[55]:
```
Fitted OptimalTreeRegressor:
  1) Split: 0.01154 * GRE Score + 0.2059 * CGPA < 5.428
    2) Split: 0.000002764 * GRE Score + 0.003919 * CGPA < 0.03223
```

```
      3) Split: 0.009033 * GRE Score + 0.2688 * CGPA < 4.807
        4) Predict: 0.4825, 32 points, error 0.005087
        5) Predict: 0.5784, 43 points, error 0.005562
      6) Split: 0.003142 * GRE Score + 0.3052 * CGPA < 3.511
        7) Predict: 0.6415, 60 points, error 0.004266
        8) Predict: 0.7021, 68 points, error 0.00259
    9) Split: CGPA < 9.055
      10) Split: 0.005854 * GRE Score + 0.251 * CGPA < 4.084
        11) Predict: 0.7447, 45 points, error 0.0041
        12) Predict: 0.7863, 52 points, error 0.003254
      13) Split: 0.002817 * GRE Score + 0.1519 * CGPA < 2.329
        14) Predict: 0.8482, 45 points, error 0.001908
        15) Predict: 0.9325, 55 points, error 0.0004517
```

[56]:
```
# predicted values for test and train sets
pred_test = grid.predict(test_X)
pred_train = grid.predict(train_X)
```

[57]:
```
# TRAIN MSE
mse(y_true = train_y, y_pred = pred_train)
```

[57]:
```
0.003246224986474234
```

[58]:
```
# TEST MSE
mse(y_true = test_y, y_pred = pred_test)
```

[58]:
```
0.007170527598937293
```

Both train and test MSE are lower than they were in the previous problem.

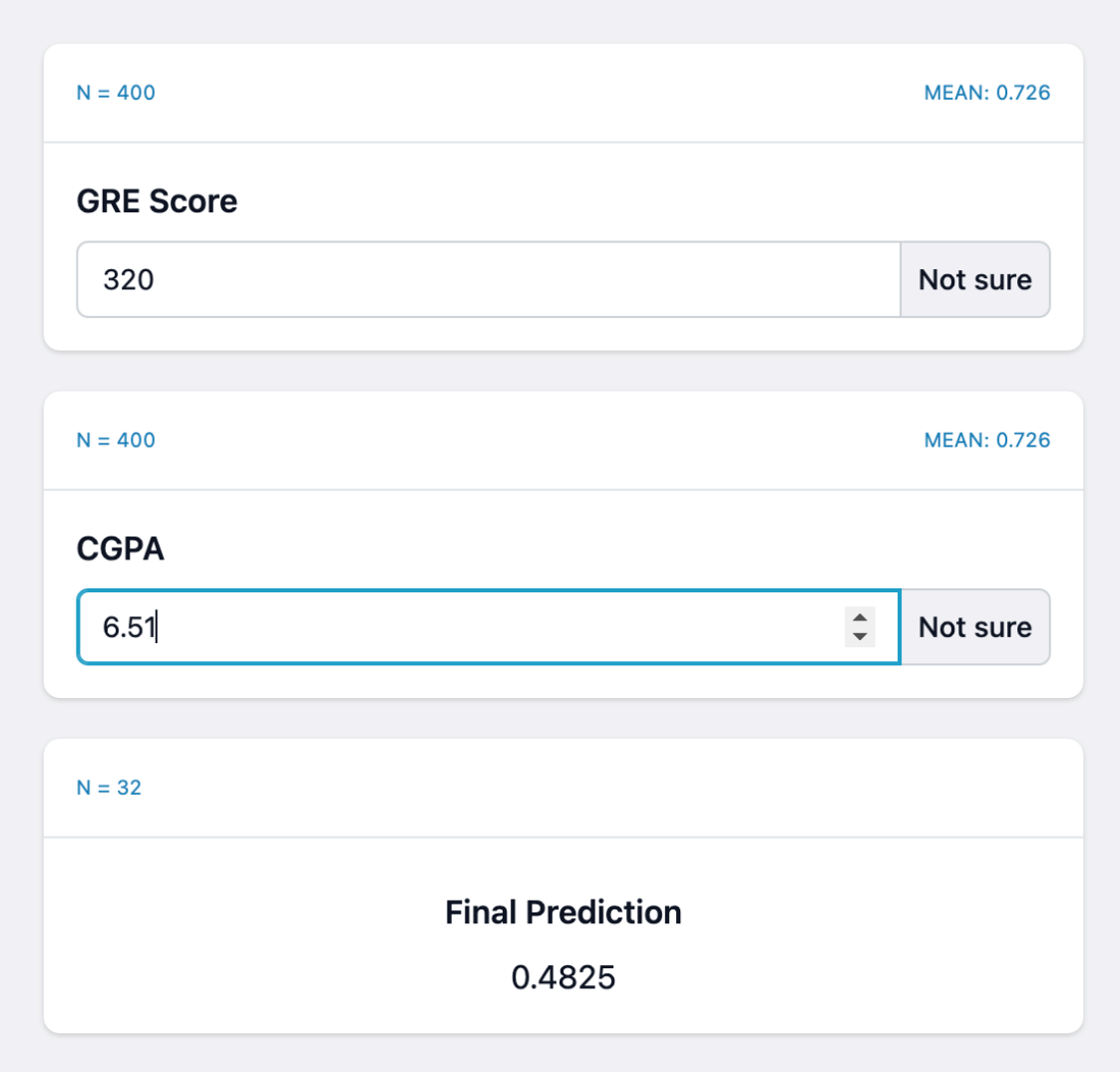
f)

[59]:
```
grid.get_learner().write_html("best_tree.html")
```

[59]:
```
626529
```

[60]:
```
grid.get_learner().write_questionnaire("best_tree_questionnaire.html")
```

[60]:
```
636265
```



0.0.2 Problem 2

[61]:
```
import tarfile
tar = tarfile.open("CUB_200_2011.tgz")
tar.extractall()
tar.close()
```

[62]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[63]:
```
# These are the metadata of all bird species and images
# path to dataset
data_path = 'CUB_200_2011/'
# aggregate datasets
df_images = pd.read_csv(data_path+'images.txt',
```

```
                          sep = ' ',header = None,
                          names = ['img_num','img'])
df_labels = pd.read_csv(data_path+'image_class_labels.txt',
                          sep = ' ',header = None,
                          names = ['img_num','class_id'])
df_classes = pd.read_csv(data_path+'classes.txt',
                           sep = ' ', header = None,
                           names = ['class_id','bird_class'])
df_split = pd.read_csv(data_path +'train_test_split.txt',
                         sep = ' ', header = None,
                         names = ['img_num','dataset'])
df = pd.merge(df_images, df_labels, on = 'img_num', how = 'inner')
df = pd.merge(df, df_classes, on = 'class_id',how = 'inner')
df = pd.merge(df, df_split, on = 'img_num',how = 'inner')
```

[64]:
```
# Delete unncessary dataframes to save RAM and storage space
del df_images, df_labels, df_classes, df_split
```

[65]:
```
# Define all classes that belong to sparrow
sparrow_class = ['113.Baird_Sparrow', '114.Black_throated_Sparrow',
                 '115.Brewer_Sparrow', '116.Chipping_Sparrow', '117.
 ↪Clay_colored_Sparrow',
                 '118.House_Sparrow', '119.Field_Sparrow', '120.Fox_Sparrow',
                 '121.Grasshopper_Sparrow', '122.Harris_Sparrow', '123.
 ↪Henslow_Sparrow',
                 '124.Le_Conte_Sparrow', '125.Lincoln_Sparrow',
                 '126.Nelson_Sharp_tailed_Sparrow', '127.Savannah_Sparrow',
                 '128.Seaside_Sparrow', '129.Song_Sparrow', '130.Tree_Sparrow',
                 '131.Vesper_Sparrow', '132.White_crowned_Sparrow', '133.
 ↪White_throated_Sparrow']
```

[66]:
```
# Define the output label: Is this a sparrow or not?
df['OUTPUT_LABEL'] = (df.bird_class.isin(sparrow_class)).astype('int')
```

a) Split the data into 70% train and 30% test using random seed 42, write a function to calculate the percentage of sparrows in train and in test. Report the percentages.

[125]:
```
############### Fill in the following lines #############
df_train_all = df.sample(frac=0.7, random_state=42)
df_test = df.drop(df_train_all.index)
```

[68]:
```
############### Fill in the following lines #############
def calc_counts(y):
    # y = 1 if sparrow
    return sum(y) / len(y)
```

```
print('train all %.3f'%calc_counts(df_train_all.OUTPUT_LABEL))
print('test %.3f'%calc_counts(df_test.OUTPUT_LABEL))
```

```
train all 0.103
test 0.111
```

b) balance the dataset by generating augmentations of the Sparrow class

[71]:
```
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing.image import img_to_array, load_img
from tqdm import tqdm
```

[70]:
```
# List of all sparrow images directory
sparrow_imgs = df_train_all.loc[df_train_all.OUTPUT_LABEL == 1,'img'].values
```

[72]:
```
!mkdir CUB_200_2011/images/aug_sparrows
```

```
mkdir: CUB_200_2011/images/aug_sparrows: File exists
```

[93]:
```
# if you still encounter errors here, make a directory called aug_sparrows␣
 ↪first in the images folder
for bird_img in tqdm(sparrow_imgs):
  img = load_img(data_path + 'images/' + bird_img)
  img = img_to_array(img)
  img = np.expand_dims(img, axis = 0)

  ############### Fill in the following lines #############
  aug = ImageDataGenerator(rotation_range=30, fill_mode='nearest',  # rotation
                           width_shift_range=0.2, height_shift_range=0.2,  #␣
 ↪shift
                           horizontal_flip=True, vertical_flip=True,  # flip
                           brightness_range=[0.4,1.5])  # brightness

  img_gen = aug.flow(img, batch_size = 1,
                     save_to_dir = data_path + 'images/aug_sparrows',
                     save_prefix = 'image',
                     save_format = 'jpg')
  total = 0
  for image in img_gen:
    total += 1
    if total == 2:
      break
```

```
100%|      | 850/850 [02:00<00:00,  7.06it/s]
```

Types of augmentation used: 1. Rotation: rotate the image by a random amount (between 0 and 360 degrees), and fill in any empty pixels resulting from the rotation with the nearest pixels. 2. Shifts: shift the pixels horizontally and/or vertically. 3. Flips: flip the image along the vertical or

horizontal axis. 4. Brightness: change the brightness of the image to simulate different lighting conditions.

[78]:
```
# Aggregate augmented datasets and available training dataset
from os import listdir
sparrow_aug_files = ['aug_sparrows/'+ a for a in listdir(data_path+'images/
 ↪aug_sparrows/') if a.endswith('.jpg')]
df_aug = pd.DataFrame({'img':sparrow_aug_files, 'OUTPUT_LABEL':␣
 ↪[1]*len(sparrow_aug_files) })
df_c = pd.concat([df_train_all[['img','OUTPUT_LABEL']],df_aug],
                 axis = 0, ignore_index = True, sort = False)
```

[79]:
```
# Balance the data with a 1:1 ratio between sparrow and non-sparrow
rows_pos = df_c.OUTPUT_LABEL == 1
df_pos = df_c.loc[rows_pos]
df_neg = df_c.loc[~rows_pos]
n= min([len(df_pos), len(df_neg)])
df_train = pd.concat([df_pos.sample(n = n,random_state = 42),
                      df_neg.sample(n = n, random_state = 42)],
                     axis = 0)
df_train = df_train.sample(frac = 1, random_state = 42)
```

[80]:
```
del df, df_train_all, df_aug, df_c, df_pos, df_neg
```

[81]:
```
# To not crash our RAM, randomly only select half of the samples in train and␣
 ↪test
df_train = df_train.sample(frac = 0.5, random_state = 42)
df_test = df_test.sample(frac = 0.5, random_state = 42)
```

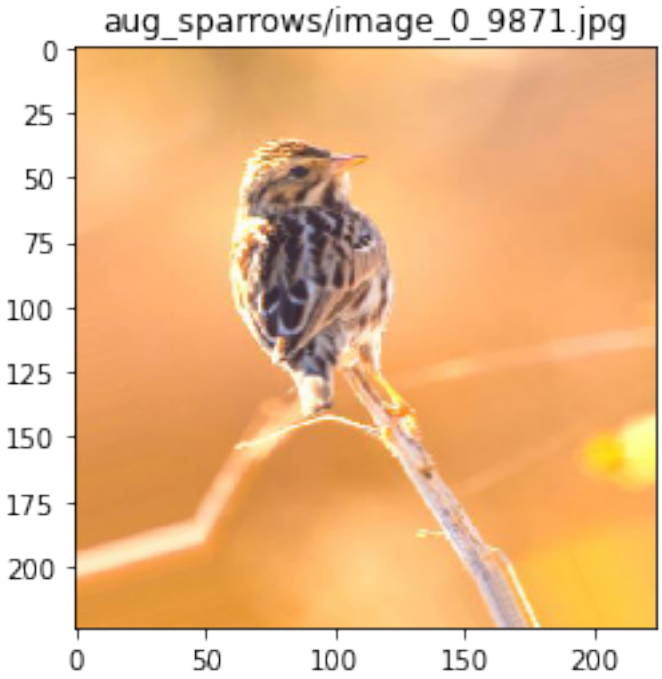
c) standardize the data

[85]:
```
IMG_SIZE = 224
def load_imgs(df):
    imgs = np.ndarray(shape = (len(df), IMG_SIZE, IMG_SIZE,3), dtype = np.
 ↪float32)
    for ii in range(len(df)):
        file = df.img.values[ii]

        ############### Fill in the following lines #############
        # convert to RGB
        img = load_img(data_path+'images/'+file,
                       color_mode='rgb',
                       target_size=(IMG_SIZE, IMG_SIZE)) # resize to 224x224

        ############### Fill in the following lines #############
        # Normalize pixel value to [0, 1] using img_to_array, then divide by 255
        img = img_to_array(img) / 255
```

```
        imgs[ii] = img
    return imgs
```

[86]:
```
# Will take a long time and a lot of RAM
X_train = load_imgs(df_train)
X_test = load_imgs(df_test)
y_train = df_train.OUTPUT_LABEL.values
y_test = df_test.OUTPUT_LABEL.values

# reshape
X_train = X_train.reshape(X_train.shape[0], IMG_SIZE,IMG_SIZE, 3)
X_test = X_test.reshape(X_test.shape[0], IMG_SIZE,IMG_SIZE, 3)
```

[132]:
```
ii = 3
plt.imshow(X_train[ii])
plt.title(df_train.img.iloc[ii])
plt.show()
```



d) build the neural network

[84]:
```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout
```

[94]:
```
model = Sequential()
model.add(Conv2D(filters = 64, kernel_size = (5,5),
                 activation = 'relu',
                 input_shape = X_train.shape[1:]))

############## Fill in the following lines #############
# Maxpooling 2D layer with a pool size of (3, 3)
model.add(MaxPool2D(pool_size=(3,3)))
# Dropout layer that random drops 25% of input units
model.add(Dropout(rate=0.25))
# Convolutional 2D layer with 64 output filers, (3, 3) convolution window, relu␣
 ↪activation
model.add(Conv2D(filters = 64, kernel_size = (3,3), activation = 'relu'))
# Maxpooling 2D layer with a pool size of (3, 3)
model.add(MaxPool2D(pool_size=(3,3)))
# Dropout layer that random drops 25% of input units
model.add(Dropout(rate=0.25))
# Flatten layer
model.add(Flatten())
# Dense layer, output space dimension of 64, relu activation
model.add(Dense(units=64, activation='relu'))
# Dropout layer that random drops 25% of input units
model.add(Dropout(rate=0.25))

model.add(Dense(1, activation = 'sigmoid'))
```

e) run the model

[97]:
```
import tensorflow as tf
```

[98]:
```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer = tf.keras.optimizers.Adam(),
              metrics = ['accuracy'])
```

[99]:
```
model.fit(X_train, y_train, batch_size = 64, epochs= 5, verbose = 1)
```

```
Epoch 1/5
59/59 [==============================] - 113s 2s/step - loss: 0.7368 - accuracy:
0.5614
Epoch 2/5
59/59 [==============================] - 105s 2s/step - loss: 0.6349 - accuracy:
0.6360
Epoch 3/5
59/59 [==============================] - 109s 2s/step - loss: 0.5699 - accuracy:
0.7033
Epoch 4/5
59/59 [==============================] - 101s 2s/step - loss: 0.5292 - accuracy:
```

```
0.7291
Epoch 5/5
59/59 [==============================] - 109s 2s/step - loss: 0.5101 - accuracy:
0.7548
```

[99]:
```
<keras.callbacks.History at 0x7f82db047c70>
```

Because this is a binary classification problem with output values in the set {0, 1}, a good loss function to use is Binary Crossentropy, which assigns a higher loss Sparrow observations that had a low predicted probability of being a Sparrow. The loss decreases (and accuracy increases) with each epoch as the weights are updated.

f) AUC and accuracy for train and test, accuracy for each sparrow class

[135]:
```
from sklearn import metrics
```

[101]:
```
y_train_preds = model.predict(X_train,verbose = 1)
y_test_preds = model.predict(X_test,verbose = 1)
```

```
117/117 [==============================] - 27s 231ms/step
56/56 [==============================] - 14s 253ms/step
```

[109]:
```
# train AUC
print("Train AUC:", metrics.roc_auc_score(y_train, y_train_preds))
```

```
Train AUC: 0.861225412557307
```

[110]:
```
# test AUC
print("Test AUC:", metrics.roc_auc_score(y_test, y_test_preds))
```

```
Test AUC: 0.800203964141788
```

[115]:
```
# create original df again with information on bird classes
# (no need to augment here because we are using the already-trained model)
df_images = pd.read_csv(data_path+'images.txt',
                        sep = ' ',header = None,
                        names = ['img_num','img'])
df_labels = pd.read_csv(data_path+'image_class_labels.txt',
                        sep = ' ',header = None,
                        names = ['img_num','class_id'])
df_classes = pd.read_csv(data_path+'classes.txt',
                         sep = ' ', header = None,
                         names = ['class_id','bird_class'])
df_split = pd.read_csv(data_path +'train_test_split.txt',
                       sep = ' ', header = None,
                       names = ['img_num','dataset'])
df = pd.merge(df_images, df_labels, on = 'img_num', how = 'inner')
df = pd.merge(df, df_classes, on = 'class_id',how = 'inner')
```

```
df = pd.merge(df, df_split, on = 'img_num',how = 'inner')

df['OUTPUT_LABEL'] = (df.bird_class.isin(sparrow_class)).astype('int')
```

[138]:
```
# accuracy for individual sparrow classes (using entire df)
for sparrow in sparrow_class:
    # subset df to just this class
    df_subset = df[df['bird_class'] == sparrow]

    # divide into X and y values and standardize
    X_subset = load_imgs(df_subset)
    y_subset = df_subset.OUTPUT_LABEL.values
    X_subset = X_subset.reshape(X_subset.shape[0], IMG_SIZE,IMG_SIZE, 3)

    # predict values for X_subset
    y_subset_prob = model.predict(X_subset)
    y_subset_pred_class = np.where(y_subset_prob > 0.5, 1,0)

    # get accuracy
    accuracy = metrics.accuracy_score(y_subset, y_subset_pred_class)

    print(f'accuracy for sparrow type {sparrow}:', accuracy)
```

```
accuracy for sparrow type 113.Baird_Sparrow: 0.96
accuracy for sparrow type 114.Black_throated_Sparrow: 0.65
accuracy for sparrow type 115.Brewer_Sparrow: 0.8813559322033898
accuracy for sparrow type 116.Chipping_Sparrow: 0.75
accuracy for sparrow type 117.Clay_colored_Sparrow: 0.9152542372881356
accuracy for sparrow type 118.House_Sparrow: 0.7666666666666667
accuracy for sparrow type 119.Field_Sparrow: 0.847457627118644
accuracy for sparrow type 120.Fox_Sparrow: 0.8333333333333334
accuracy for sparrow type 121.Grasshopper_Sparrow: 0.9
accuracy for sparrow type 122.Harris_Sparrow: 0.85
accuracy for sparrow type 123.Henslow_Sparrow: 0.95
accuracy for sparrow type 124.Le_Conte_Sparrow: 0.9152542372881356
accuracy for sparrow type 125.Lincoln_Sparrow: 0.9152542372881356
accuracy for sparrow type 126.Nelson_Sharp_tailed_Sparrow: 0.8983050847457628
accuracy for sparrow type 127.Savannah_Sparrow: 0.8666666666666667
accuracy for sparrow type 128.Seaside_Sparrow: 0.9
accuracy for sparrow type 129.Song_Sparrow: 0.8833333333333333
accuracy for sparrow type 130.Tree_Sparrow: 0.8666666666666667
accuracy for sparrow type 131.Vesper_Sparrow: 0.9
accuracy for sparrow type 132.White_crowned_Sparrow: 0.7833333333333333
accuracy for sparrow type 133.White_throated_Sparrow: 0.8333333333333334
```

The sparrow class that the model performs the best on is the Baird Sparrow. The class that the model performs the worst on is the Black Throated Sparrow. This may be because Baird Sparrows have more distinguishing features that make them more obviously sparrows while Black Throated

Sparrows have more ambiguous features.

[ ]: