PROJECT: NAÏVE BEES: PREDICT SPECIES FROM IMAGES

BİLGİSAYARLI GÖRME İÇİN DERİN ÖĞRENME UYGULAMALARI

İstanbul Üniversitesi

Rayees Ahmad Lone

## 1. Import Python libraries



A honey bee (Apis).

Can a machine identify a bee as a honey bee or a bumble bee? These bees have different behaviors and appearances, but given the variety of backgrounds, positions, and image resolutions, it can be a challenge for machines to tell them apart.

Being able to identify bee species from images is a task that ultimately would allow researchers to more quickly and effectively collect field data. Pollinating bees have critical roles in both ecology and agriculture, and diseases like colony collapse disorder threaten these species. Identifying different species of bees in the wild means that we can better understand the prevalence and growth of these important insects.



A bumble bee (Bombus).

After loading and pre-processing images, this notebook walks through building a model that can automatically detect honey bees and bumble bees.

# used to change filepaths  
import os  
  
import matplotlib as mpl  
import matplotlib.pyplot as plt  
from IPython.display import display  
%matplotlib inline  
import pandas as pd  
import numpy as np  
  
# import Image from PIL  
from PIL import Image  
  
from skimage.feature import hog  
from skimage.color import rgb2grey  
  
from sklearn.preprocessing import StandardScaler  
from sklearn.decomposition import PCA  
  
# import train\_test\_split from sklearn's model selection module  
from sklearn.model\_selection import train\_test\_split  
  
# import SVC from sklearn's svm module  
from sklearn.svm import SVC  
  
# import accuracy\_score from sklearn's metrics module  
from sklearn.metrics import roc\_curve, auc, accuracy\_score

## 2. Display image of each bee type

Now that we have all of our imports ready, it is time to look at some images. We will load our labels.csv file into a dataframe called labels, where the index is the image name (e.g. an index of 1036 refers to an image named 1036.jpg) and the genus column tells us the bee type. genus takes the value of either 0.0 (Apis or honey bee) or 1.0 (Bombus or bumble bee).

The function get\_image converts an index value from the dataframe into a file path where the image is located, opens the image using the Image object in Pillow, and then returns the image as a numpy array.

We'll use this function to load the sixth Apis image and then the sixth Bombus image in the dataframe.

# load the labels using pandas  
labels = pd.read\_csv("datasets/labels.csv", index\_col=0)  
  
# show the first five rows of the dataframe using head  
display(labels.head)  
  
def get\_image(row\_id, root="datasets/"):  
 """  
 Converts an image number into the file path where the image is located,   
 opens the image, and returns the image as a numpy array.  
 """  
 filename = "{}.jpg".format(row\_id)  
 file\_path = os.path.join(root, filename)  
 img = Image.open(file\_path)  
 return np.array(img)  
  
# subset the dataframe to just Apis (genus is 0.0) get the value of the sixth item in the index  
apis\_row = labels[labels.genus == 0.0].index[5]  
  
# show the corresponding image of an Apis  
plt.imshow(get\_image(apis\_row))  
plt.show()  
  
# subset the dataframe to just Bombus (genus is 1.0) get the value of the sixth item in the index  
bombus\_row = labels[labels.genus == 1.0].index[5]  
  
# show the corresponding image of a Bombus  
plt.imshow(get\_image(bombus\_row))  
plt.show()

## # Results

id   
520 1.0  
3800 1.0  
3289 1.0  
2695 1.0  
4922 1.0  
1934 1.0  
3704 1.0  
4017 1.0  
3331 1.0  
173 1.0  
2983 1.0  
116 1.0  
4938 1.0  
3524 1.0  
346 1.0  
631 1.0  
3588 1.0  
3321 1.0  
2237 1.0  
1180 1.0  
1725 1.0  
4067 1.0  
2675 1.0  
728 1.0  
1815 1.0  
1722 1.0  
1377 1.0  
432 1.0  
1506 1.0  
1705 1.0  
... ...  
3229 0.0  
2157 0.0  
3926 0.0  
3515 0.0  
4645 0.0  
1106 0.0  
2851 0.0  
630 0.0  
938 0.0  
2794 0.0  
4612 0.0  
3644 0.0  
3001 0.0  
3464 0.0  
4618 0.0  
2142 0.0  
1087 0.0  
2220 0.0  
701 0.0  
1279 0.0  
134 0.0  
3680 0.0  
3987 0.0  
2113 0.0  
3015 0.0  
650 0.0  
3782 0.0  
4026 0.0  
1347 0.0  
807 0.0  
  
[500 rows x 1 columns]>

A close up of a flower

Description automatically generated

A picture containing bird

Description automatically generated

## 3. Image manipulation with rgb2grey

Scikit-image has a number of image processing functions built into the library, for example, converting an image to greyscale. The rgb2grey function computes the luminance of an RGB image using the following formula Y = 0.2125 R + 0.7154 G + 0.0721 B.

Image data is represented as a matrix, where the depth is the number of channels. An RGB image has three channels (red, green, and blue) whereas the returned greyscale image has only one channel. Accordingly, the original color image has the dimensions 100x100x3 but after calling rgb2grey, the resulting greyscale image has only one channel, making the dimensions 100x100x1.

# load a bombus image using our get\_image function and bombus\_row from the previous cell  
bombus = get\_image(bombus\_row)  
  
# print the shape of the bombus image  
print('Color bombus image has shape: ', bombus.shape)  
  
# convert the bombus image to greyscale  
grey\_bombus = rgb2grey(bombus)  
  
# show the greyscale image  
plt.imshow(grey\_bombus, cmap=mpl.cm.gray)  
  
# greyscale bombus image only has one channel  
print('Greyscale bombus image has shape: ', grey\_bombus.shape)

## # Results

Color bombus image has shape: (100, 100, 3)  
Greyscale bombus image has shape: (100, 100)

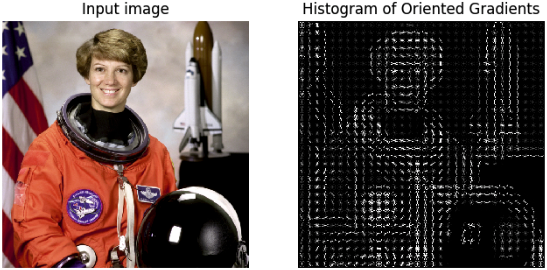
A picture containing bird

Description automatically generated

## 4. Histogram of oriented gradients

Now we need to turn these images into something that a machine learning algorithm can understand. Traditional computer vision techniques have relied on mathematical transforms to turn images into useful features. For example, you may want to detect edges of objects in an image, increase the contrast, or filter out particular colors.

We've got a matrix of pixel values, but those don't contain enough interesting information on their own for most algorithms. We need to help the algorithms along by picking out some of the salient features for them using the histogram of oriented gradients (HOG) descriptor. The idea behind HOG is that an object's shape within an image can be inferred by its edges, and a way to identify edges is by looking at the direction of intensity gradients (i.e. changes in luminescence).



An image is divided in a grid fashion into cells, and for the pixels within each cell, a histogram of gradient directions is compiled. To improve invariance to highlights and shadows in an image, cells are block normalized, meaning an intensity value is calculated for a larger region of an image called a block and used to contrast normalize all cell-level histograms within each block. The HOG feature vector for the image is the concatenation of these cell-level histograms.

# run HOG using our greyscale bombus image  
hog\_features, hog\_image = hog(grey\_bombus,  
 visualize=True,  
 block\_norm='L2-Hys',  
 pixels\_per\_cell=(16, 16))  
  
# show our hog\_image with a grey colormap  
plt.imshow(hog\_image, cmap=mpl.cm.gray)

A picture containing white

Description automatically generated

## 5. Create image features and flatten into a single row

Algorithms require data to be in a format where rows correspond to images and columns correspond to features. This means that all the information for a given image needs to be contained in a single row.

We want to provide our model with the raw pixel values from our images as well as the HOG features we just calculated. To do this, we will write a function called create\_features that combines these two sets of features by flattening the three-dimensional array into a one-dimensional (flat) array.

def create\_features(img):  
 # flatten three channel color image  
 color\_features = np.ndarray.flatten(img)  
 # convert image to greyscale  
 grey\_image = rgb2grey(img)  
 # get HOG features from greyscale image  
 hog\_features = hog(grey\_image, block\_norm='L2-Hys', pixels\_per\_cell=(16, 16))  
 # combine color and hog features into a single array  
 flat\_features = np.hstack((color\_features,hog\_features))  
 return flat\_features  
  
bombus\_features = create\_features(bombus)  
  
# print shape of bombus\_features  
print(bombus\_features.shape)

## # Results

(31296,)

## 6. Loop over images to preprocess

Above we generated a flattened features array for the bombus image. Now it's time to loop over all of our images. We will create features for each image and then stack the flattened features arrays into a big matrix we can pass into our model.

In the create\_feature\_matrix function, we'll do the following:

* Load an image
* Generate a row of features using the create\_features function above
* Stack the rows into a features matrix

In the resulting features matrix, rows correspond to images and columns to features.

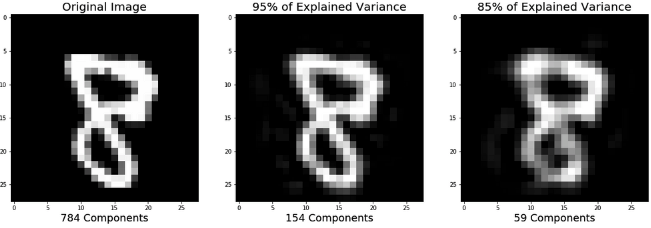
def create\_feature\_matrix(label\_dataframe):  
 features\_list = []  
   
 for img\_id in label\_dataframe.index:  
 # load image  
 img = get\_image(img\_id)  
 # get features for image  
 image\_features = create\_features(img)  
 features\_list.append(image\_features)  
   
 # convert list of arrays into a matrix  
 feature\_matrix = np.array(features\_list)  
 return feature\_matrix  
  
# run create\_feature\_matrix on our dataframe of images  
feature\_matrix = create\_feature\_matrix(labels)

## 7. Scale feature matrix + PCA

Our features aren't quite done yet. Many machine learning methods are built to work best with data that has a mean of 0 and unit variance. Luckily, scikit-learn provides a simple way to rescale your data to work well using StandardScaler. They've got a more thorough explanation of why that is in the linked docs.

Remember also that we have over 31,000 features for each image and only 500 images total. To use an SVM, our model of choice, we also need to reduce the number of features we have using principal component analysis (PCA).

PCA is a way of linearly transforming the data such that most of the information in the data is contained within a smaller number of features called components. Below is a visual example from an image dataset containing handwritten numbers. The image on the left is the original image with 784 components. We can see that the image on the right (post PCA) captures the shape of the number quite effectively even with only 59 components.



In our case, we will keep 500 components. This means our feature matrix will only have 500 columns rather than the original 31,296.

# get shape of feature matrix  
print('Feature matrix shape is: ', feature\_matrix.shape)  
  
# define standard scaler  
ss = StandardScaler()  
# run this on our feature matrix  
bees\_stand = ss.fit\_transform(feature\_matrix)  
  
pca = PCA(n\_components=500)  
# use fit\_transform to run PCA on our standardized matrix  
bees\_pca = pca.fit\_transform(bees\_stand)  
# look at new shape  
print('PCA matrix shape is: ', bees\_pca.shape)

## # Results

Feature matrix shape is: (500, 31296)  
PCA matrix shape is: (500, 500)

## 8. Split into train and test sets

Now we need to convert our data into train and test sets. We'll use 70% of images as our training data and test our model on the remaining 30%. Scikit-learn's train\_test\_split function makes this easy.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(bees\_pca,  
 labels.genus.values,  
 test\_size=.3,  
 random\_state=1234123)  
  
# look at the distrubution of labels in the train set  
pd.Series(y\_train).value\_counts()

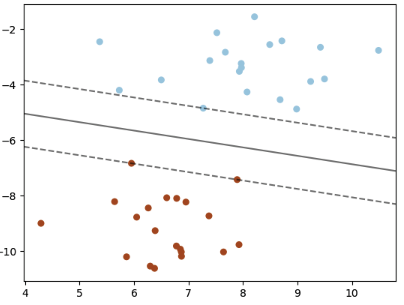
## # Results

0.0 175  
1.0 175  
dtype: int64

## 9. Train model

It's finally time to build our model! We'll use a support vector machine (SVM), a type of supervised machine learning model used for regression, classification, and outlier detection." An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall."

Here's a visualization of the maximum margin separating two classes using an SVM classifier with a linear kernel.



Since we have a classification task -- honey or bumble bee -- we will use the support vector classifier (SVC), a type of SVM. We imported this class at the top of the notebook.

# define support vector classifier  
svm = SVC(kernel='linear',probability=True,random\_state=42)  
  
# fit model  
svm.fit(X\_train,y\_train)

## # Results

SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0,  
 decision\_function\_shape='ovr', degree=3, gamma='auto', kernel='linear',  
 max\_iter=-1, probability=True, random\_state=42, shrinking=True,  
 tol=0.001, verbose=False)

## 10. Score model

Now we'll use our trained model to generate predictions for our test data. To see how well our model did, we'll calculate the accuracy by comparing our predicted labels for the test set with the true labels in the test set. Accuracy is the number of correct predictions divided by the total number of predictions. Scikit-learn's accuracy\_score function will do math for us. Sometimes accuracy can be misleading, but since we have an equal number of honey and bumble bees, it is a useful metric for this problem.

# generate predictions  
y\_pred = svm.predict(X\_test)  
  
# calculate accuracy  
accuracy = accuracy\_score(y\_pred, y\_test)  
print('Model accuracy is: ', accuracy)

## # Results

Model accuracy is: 0.68

## 11. ROC curve + AUC

Above, we used svm.predict to predict either 0.0 or 1.0 for each image in X\_test. Now, we'll use svm.predict\_proba to get the probability that each class is the true label. For example, predict\_proba returns [0.46195176, 0.53804824] for the first image, meaning there is a 46% chance the bee in the image is an Apis (0.0) and a 53% chance the bee in the image is a Bombus (1.0). Note that the two probabilities for each image always sum to 1.

Using the default settings, probabilities of 0.5 or above are assigned a class label of 1.0 and those below are assigned a 0.0. However, this threshold can be adjusted. The receiver operating characteristic curve (ROC curve) plots the false positive rate and true positive rate at different thresholds. ROC curves are judged visually by how close they are to the upper lefthand corner.

The area under the curve (AUC) is also calculated, where 1 means every predicted label was correct. Generally, the worst score for AUC is 0.5, which is the performance of a model that randomly guesses. See the scikit-learn documentation for more resources and examples on ROC curves and AUC.

# predict probabilities for X\_test using predict\_proba  
probabilities = svm.predict\_proba(X\_test)  
  
# select the probabilities for label 1.0  
y\_proba = probabilities[:, 1]  
  
# calculate false positive rate and true positive rate at different thresholds  
false\_positive\_rate, true\_positive\_rate, thresholds = roc\_curve(y\_test, y\_proba, pos\_label=1)  
  
# calculate AUC  
roc\_auc = auc(false\_positive\_rate, true\_positive\_rate)  
  
plt.title('Receiver Operating Characteristic')  
# plot the false positive rate on the x axis and the true positive rate on the y axis  
roc\_plot = plt.plot(false\_positive\_rate,  
 true\_positive\_rate,  
 label='AUC = {:0.2f}'.format(roc\_auc))  
  
plt.legend(loc=0)  
plt.plot([0,1], [0,1], ls='--')  
plt.ylabel('True Positive Rate')  
plt.xlabel('False Positive Rate');

