Import Libraries

In [1]:
```
import os
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
from cv2 import resize as cv2_resize
from librosa import load as lib_load
from librosa import feature as lib_feature
import seaborn as sns

# Import sci-kit models
from sklearn.preprocessing import OneHotEncoder , RobustScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import confusion_matrix
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier, NearestCentroid
```

Load Data

In [2]:
```
# Launch Jupyter Notebook from the project folder
root_path = os.getcwd()
# Data is stored in the folder named "Data"
path = os.path.join(root_path, 'Edited Data')
```

In [3]:
```
def load_data(path, sample_rate):
    # Create data lists
    samples = []
    labels = []
    classes = os.listdir(path)
    print('Loading data...')

    for file in os.listdir(path):
        class_path = os.path.join(path, file)
        for filename in os.listdir(class_path):
            # Load data
            y, s = lib_load(os.path.join(class_path,filename), sr=sample_rate)
            # Append data and label
            labels.append([file])
            samples.append(y)
            #print('Loaded {}'.format(filename))

    return samples, classes, labels
```

In [4]:
```
# Define sample rate
sample_rate = 44_100
# Load data
samples, classes, labels = load_data(path, sample_rate)
# Save full data set if samples is overwritten (sub-sampling) later
full_samples = samples
```

```
full_labels = labels

# Print data information
print('Loaded data from the directory {} \n'.format(path))
print('Loaded the classes: {} \n'.format(classes))
print('Loaded {} samples with {} labels \n'.format(len(samples), len(labels)))

# Check lengths of samples and their labels
for idx, sample in enumerate(samples):
  print('Sample {:2}: \t {:.1f}s \t {}'.format(idx, len(sample)/sample_rate, labels
```

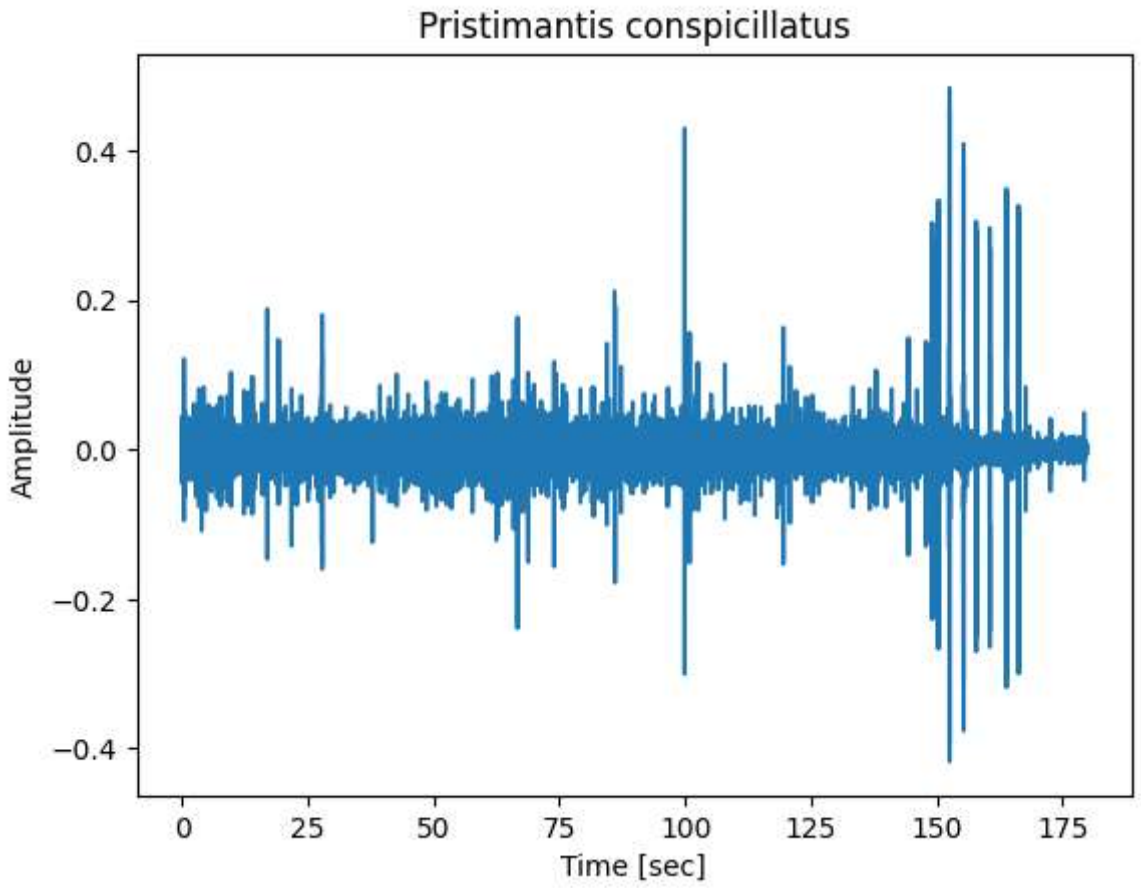
```
Loading data...
Loaded data from the directory C:\Users\14w\Documents\CS522_final\Edited Data

Loaded the classes: ['Dendropsophus bifurcus', 'Engystomops petersi', 'Pristimanti
s conspicillatus']

Loaded 29 samples with 29 labels

Sample  0:       105.8s          ['Dendropsophus bifurcus']
Sample  1:       497.8s          ['Dendropsophus bifurcus']
Sample  2:       45.9s   ['Dendropsophus bifurcus']
Sample  3:       29.4s   ['Dendropsophus bifurcus']
Sample  4:       105.8s          ['Dendropsophus bifurcus']
Sample  5:       208.6s          ['Dendropsophus bifurcus']
Sample  6:       247.6s          ['Dendropsophus bifurcus']
Sample  7:       366.1s          ['Dendropsophus bifurcus']
Sample  8:       204.9s          ['Dendropsophus bifurcus']
Sample  9:       236.5s          ['Dendropsophus bifurcus']
Sample 10:       439.7s          ['Engystomops petersi']
Sample 11:       220.5s          ['Engystomops petersi']
Sample 12:       253.4s          ['Engystomops petersi']
Sample 13:       248.4s          ['Engystomops petersi']
Sample 14:       381.5s          ['Engystomops petersi']
Sample 15:       838.1s          ['Engystomops petersi']
Sample 16:       67.1s   ['Engystomops petersi']
Sample 17:       79.8s   ['Engystomops petersi']
Sample 18:       100.3s          ['Engystomops petersi']
Sample 19:       167.1s          ['Engystomops petersi']
Sample 20:       140.2s          ['Engystomops petersi']
Sample 21:       201.3s          ['Pristimantis conspicillatus']
Sample 22:       179.7s          ['Pristimantis conspicillatus']
Sample 23:       164.8s          ['Pristimantis conspicillatus']
Sample 24:       266.3s          ['Pristimantis conspicillatus']
Sample 25:       89.7s   ['Pristimantis conspicillatus']
Sample 26:       160.1s          ['Pristimantis conspicillatus']
Sample 27:       168.4s          ['Pristimantis conspicillatus']
Sample 28:       230.6s          ['Pristimantis conspicillatus']
```

Plot Data

In [5]:
```
# Choose series to plot
# Dendropsophus bifurcus = 7
# Engystomops petersi = 10
# Pristimantis conspicillatus = 22
```

```
sample = samples[22]
plt.plot(np.linspace(0, len(sample)/sample_rate, len(sample)), sample)
plt.xlabel('Time [sec]')
plt.ylabel('Amplitude')
plt.title('Pristimantis conspicillatus')
#plt.savefig('PC.png', bbox_inches='tight')
```

Out[5]: Text(0.5, 1.0, 'Pristimantis conspicillatus')



Down-sample Data to 18 Seconds

In [6]:
```
## Random sub-sampling
# If True, data is down sampled. Specify subsampled time below.
if False:
    # Specify down-sample time in seconds and number of samples
    subsample_time = 18
    subsample_samples = subsample_time * sample_rate
    # Create sub-sampled data
    subsamples = []
    validation_samples = []
    for sample in full_samples:
        # Get correct indicies
        start = np.random.randint(0, len(sample) - subsample_samples)
        end = start + subsample_samples
        if end >= len(sample):
            start = start - 1
            end = end - 1
        # Create sub-sampled array
```

```
        subsamples.append(sample[start:end])
        # Create training matrix from remaining data by taking the longer of the re
        # Training samples are not added to avoid creating splicing artifacts
        train1 = sample[0:start]
        train2 = sample[end:-1]
        if len(train1) > len(train2):
            validation_samples.append(train1)
        else:
            validation_samples.append(train2)

    # Overwrite samples data with subsamples
    samples = subsamples
```

In [7]:
```
## Multi-index sub-sampling
# If True, data is down sampled. Specify subsampled time below.
if True:
    # Number of samples
    num_samples = 2
    # Specify down-sample time in seconds and number of samples
    subsample_time = 18
    subsample_samples = subsample_time * sample_rate
    # Create sub-sampled data
    subsamples = []
    sublabels = []
    for idx, sample in enumerate(full_samples):
        # Get correct indicies to start
        start = 0
        end = start + subsample_samples
        for iidx in range(num_samples):
            if end >= len(sample):
                break
            # Create sub-sampled array
            subsamples.append(sample[start:end])
            # Create new label array
            sublabels.append(full_labels[idx])
            # Increment
            start = end + 1
            end = start + subsample_samples

    # Overwrite samples and labels data with subsamples and sublabels
    samples = subsamples
    labels = sublabels
```

Encode labels

In [8]:
```
# Reshape classes list
unique_classes = [[c] for c in classes]

# Create encoder, fit, and check encoding
enc = OneHotEncoder()
enc.fit(unique_classes)
enc_labels = enc.transform(labels).toarray()
for idx, item in enumerate(enc_labels):
    print('Encoding is {}, label is {}'.format(enc_labels[idx], labels[idx]))
```

```
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [1. 0. 0.], label is ['Dendropsophus bifurcus']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 1. 0.], label is ['Engystomops petersi']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
```

```
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
Encoding is [0. 0. 1.], label is ['Pristimantis conspicillatus']
```

In [9]:
```
print(enc.categories_)
```

```
[array(['Dendropsophus bifurcus', 'Engystomops petersi',
       'Pristimantis conspicillatus'], dtype=object)]
```

In [10]:
```
enc_orders = ['Dendropsophus bifurcus', 'Engystomops petersi','Pristimantis conspic
```

Get Spectrograms

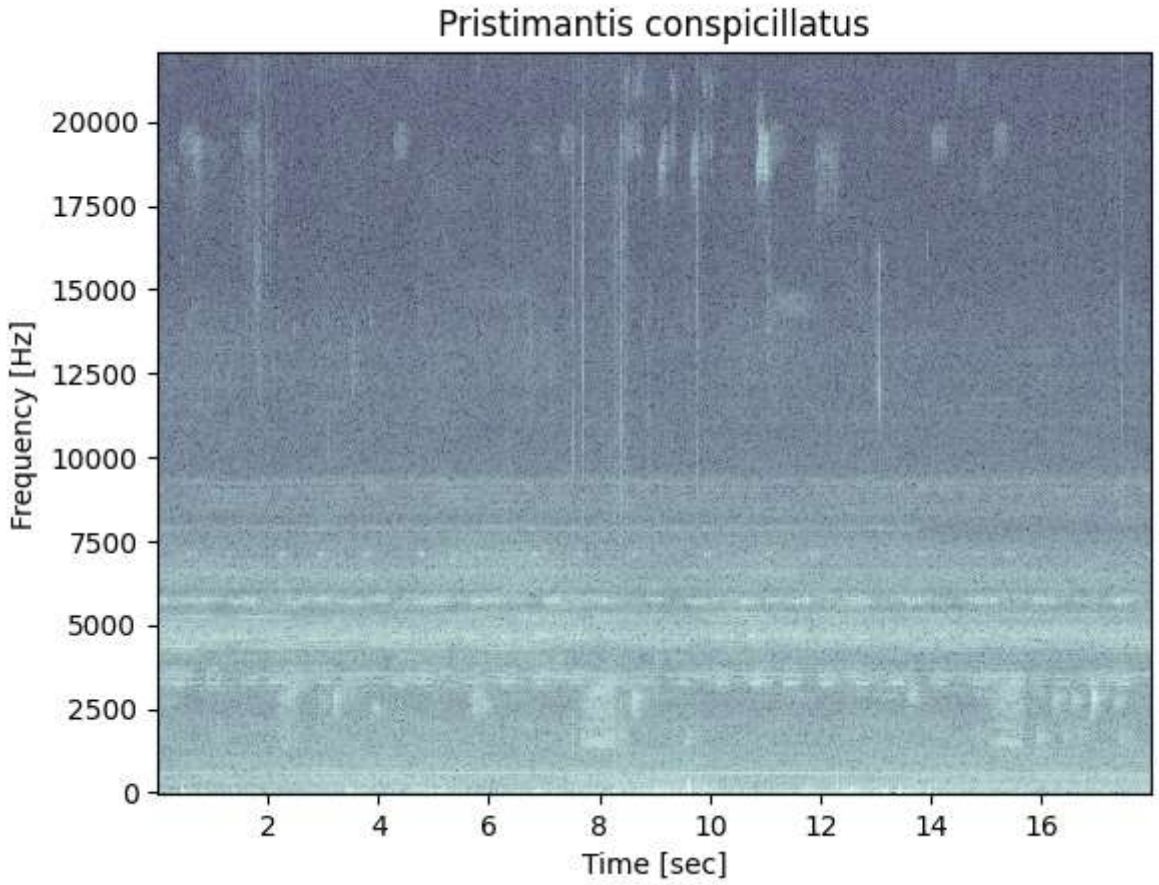
In [11]:
```
def get_spectrograms(data, fft_size=1_024, sample_rate=44_100):
    ## Compute Fast Fourier Transform of data
    # Create FFT list
    f = []
    t = []
    pxx = []

    print("Calculating FFTs for data ...")
    for datum in data:
        f_idx, t_idx, pxx_idx = signal.spectrogram(datum, nperseg=fft_size, fs=samp

        f.append(f_idx)
        t.append(t_idx)
        pxx.append(pxx_idx)

    return f, t, pxx
```

In [12]:
```
# Compute spectrograms if True
if True:
    # Specify FFT size
    fft_size = 1024
    # Load data
    freqs, times, specs = get_spectrograms(samples, fft_size, sample_rate)

    # Check spectrogram data shapes
    print('Spectrograms: \t {}'.format(len(specs)))
    print('Frequency bins: {} \n'.format(len(specs[0])))
    for idx, spec in enumerate(specs):
        print('Spectrogram {} time bins: \t {}'.format(idx, len(spec[0])))
```

```
Calculating FFTs for data ...
Spectrograms:    57
Frequency bins: 513

Spectrogram 0 time bins:        1549
Spectrogram 1 time bins:        1549
Spectrogram 2 time bins:        1549
Spectrogram 3 time bins:        1549
Spectrogram 4 time bins:        1549
Spectrogram 5 time bins:        1549
Spectrogram 6 time bins:        1549
Spectrogram 7 time bins:        1549
Spectrogram 8 time bins:        1549
Spectrogram 9 time bins:        1549
Spectrogram 10 time bins:       1549
Spectrogram 11 time bins:       1549
Spectrogram 12 time bins:       1549
Spectrogram 13 time bins:       1549
Spectrogram 14 time bins:       1549
Spectrogram 15 time bins:       1549
Spectrogram 16 time bins:       1549
Spectrogram 17 time bins:       1549
Spectrogram 18 time bins:       1549
Spectrogram 19 time bins:       1549
Spectrogram 20 time bins:       1549
Spectrogram 21 time bins:       1549
Spectrogram 22 time bins:       1549
Spectrogram 23 time bins:       1549
Spectrogram 24 time bins:       1549
Spectrogram 25 time bins:       1549
Spectrogram 26 time bins:       1549
Spectrogram 27 time bins:       1549
Spectrogram 28 time bins:       1549
Spectrogram 29 time bins:       1549
Spectrogram 30 time bins:       1549
Spectrogram 31 time bins:       1549
Spectrogram 32 time bins:       1549
Spectrogram 33 time bins:       1549
Spectrogram 34 time bins:       1549
Spectrogram 35 time bins:       1549
Spectrogram 36 time bins:       1549
Spectrogram 37 time bins:       1549
Spectrogram 38 time bins:       1549
Spectrogram 39 time bins:       1549
Spectrogram 40 time bins:       1549
Spectrogram 41 time bins:       1549
Spectrogram 42 time bins:       1549
Spectrogram 43 time bins:       1549
Spectrogram 44 time bins:       1549
Spectrogram 45 time bins:       1549
Spectrogram 46 time bins:       1549
Spectrogram 47 time bins:       1549
Spectrogram 48 time bins:       1549
Spectrogram 49 time bins:       1549
Spectrogram 50 time bins:       1549
Spectrogram 51 time bins:       1549
```

```
Spectrogram 52 time bins:        1549
Spectrogram 53 time bins:        1549
Spectrogram 54 time bins:        1549
Spectrogram 55 time bins:        1549
Spectrogram 56 time bins:        1549
```

In [13]:
```
# Plot spectrogram for data
fig = plt.figure()
fft_size = 1024
f, t, pxx = signal.spectrogram(samples[15], nperseg=fft_size, fs=sample_rate, nover

cmap=plt.cm.bone
cmap = plt.cm.get_cmap("bone").copy()
cmap.set_under(color='k', alpha=None)
plt.pcolormesh(t, f, np.log10(pxx), cmap=cmap)

plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [sec]')
plt.title('Pristimantis conspicillatus')

plt.show()

title = 'PCSpec.png'
#fig.savefig(title, bbox_inches='tight')
```



Binning Data

In [14]:
```
def get_bins(data, num_freq_bins=5, num_time_bins=5):
    binned_data = []

    # Get individual samples
    for datum in data:
        #Open CV's resize takes (columns,rows) as the input for desired size
        resized_pxx=cv2_resize(datum[:,:],(num_time_bins,num_freq_bins))
        binned_data.append(resized_pxx.flatten())

    return binned_data
```

In [15]:
```
# Get binned data if True
if True:
    # Define bins
    freq_bins = 5
    time_bins = 5
    # Get binned data
    binned_specs = get_bins(specs, freq_bins, time_bins)

    # Print bin shapes
    print('Binned spectrograms: \t {}'.format(len(binned_specs)))
    print('Number of bins: \t {}'.format(len(binned_specs[0])))
```

```
Binned spectrograms:     57
Number of bins:          25
```

Get Windows

In [16]:
```
def get_time_windows(data, len_window=1, overlap=0, sample_rate=44_100):
    # Print values of windows
    print('Extracting time windows {}s long with {}% overlap. \n'.format(len_window
    # Define window interval lengths and number of samples per interval
    interval_samples = len_window * sample_rate

    windows = []
    for idx, datum in enumerate(data):
        # Define sample time values
        total_time = len(datum) / sample_rate
        start_time = len_window * overlap
        overlap_time = len_window * overlap
        # Define vector of window center time values
        times = np.arange(start_time, total_time - overlap_time, len_window)

        # Create windows
        datum_windows = []
        for iidx, time in enumerate(times):
            low_time = interval_samples * iidx
            high_time = interval_samples * (iidx+1) - 1
            window = datum[low_time:high_time]
            datum_windows.append(window)

        windows.append(datum_windows)

    return windows
```

In [17]:
```
def get_freq_windows(data, len_window=1, overlap=0, sample_rate=44_100, fft_window=
    # Print values of windows
    print('Extracting frequency windows {}s long with {}% overlap. \n'.format(len_w
    # Define samples per interval
    interval_samples = round(len_window * sample_rate / fft_window // 1)

    windows = []
    for idx, datum in enumerate(data):
        # Define sample time values
        total_time = datum.shape[1] * fft_window / sample_rate
        start_time = len_window * overlap
        overlap_time = len_window * overlap
        # Define vector of window center time values
        times = np.arange(start_time, total_time - overlap_time, len_window)

        # Create windows
        datum_windows = []
        for iidx, time in enumerate(times):
            low_time = interval_samples * iidx
            high_time = interval_samples * (iidx+1) - 1
            window = np.array(datum[:, low_time:high_time])
            datum_windows.append(window)

        windows.append(datum_windows)

    return windows
```

In [18]:
```
def get_windowed_labels(labels, windowed_data):
    # Initialize output list and iterate over non-windowed labels
    windowed_labels = []
    for idx, label in enumerate(labels):
        # Compute number of windows for sample associated with label
        num_windows = len(windowed_samples[idx])
        # Create list of label repeated for each window
        windowed_label = []
        for idx in range(0, num_windows):
            windowed_label.append(label)

        windowed_labels.append(windowed_label)

    return windowed_labels
```

In [19]:
```
# Get windows if true
if False:
    # Define time windows of interest
    len_window = 5
    window_overlap = 0.5

    # Get time windows
    windowed_samples = get_time_windows(samples, len_window, window_overlap, sample
    # Get frequency windows
    windowed_specs = get_freq_windows(specs, len_window, window_overlap, sample_rat
    # Get windowed labels
    windowed_labels = get_windowed_labels(labels, windowed_samples)
    # Get encoded windowed labels
    enc_windowed_labels = get_windowed_labels(enc_labels, windowed_samples)

    # Print data shapes
    for idx, sample in enumerate(windowed_samples):
        print('Windows in sample {}: \t {}'.format(idx, len(sample)))
```

Flatten windowed arrays, optional

In [20]:
```
def flatten_windows(samples):
    flat = []
    for sample in samples:
        for data in sample:
            flat.append(data)

    return flat
```

In [21]:
```
# Flatten windowed time series data, spectrograms, and labels if True
if False:
    windowed_samples = flatten_windows(windowed_samples)
    windowed_specs = flatten_windows(windowed_specs)
    windowed_labels = flatten_windows(windowed_labels)
    enc_windowed_labels = flatten_windows(enc_windowed_labels)
    print('Samples length: \t {}'.format(len(windowed_samples)))
    print('Spectrograms length: \t {}'.format(len(windowed_specs)))
    print('Labels length: \t {}'.format(len(windowed_labels)))
```

Feature Extraction

Define features

In [22]:
```
# Extract MFCC features
def extract_mfcc(samples, sample_rate=44_100, num_mfcc=10, fft_size=1_024, window_n
    print('Extracting MFCCs...')
    features = []
    for i in range(len(samples)):
        # Compute MFCCs
        mfccs = lib_feature.mfcc(y=samples[i], sr=sample_rate, n_mfcc=10, win_lengt
        # Get mean for each value
        mfccs_mean = np.mean(mfccs.T, axis=0)
        # Re-cast to list
        mfccs_mean = mfccs_mean.tolist()

        # Get features for all samples
        features.append(mfccs_mean)

    print('MFCCs extracted.')

    return features
```

In [23]:
```
# Extract Spectral Centorid features
def extract_sc(samples, sample_rate=44_100, fft_size=1_024, window_numbers=20):
    print('Extracting spectral centroid...')
    features = []
    for i in range(len(samples)):
        # Compute Spectral Centroid
        sc = lib_feature.spectral_centroid(y=samples[i], sr=sample_rate, win_length
        reshape_sc=[]
        for x in sc:
            for j in x:
                reshape_sc.append(j)
        max_sc = max(reshape_sc)
        min_sc = min(reshape_sc)
        mean_sc = np.mean(reshape_sc)
        sc_fv = [max_sc, min_sc, mean_sc]

        # Get features for all samples
        features.append(sc_fv)

    print('Spectral Centroid extracted.')

    return features
```

In [24]:
```
# Extract Bandwidth features
def extract_bw(samples, sample_rate=44_100, fft_size=1_024, window_numbers=20):
    print('Extracting bandwidth...')
    features = []
    for i in range(len(samples)):
        # Compute Bandwidth
```

```
        bw = lib_feature.spectral_bandwidth(y=samples[i], sr=sample_rate, win_lengt
        reshape_bw=[]
        for x in bw:
            for j in x:
                reshape_bw.append(j)
        max_bw = max(reshape_bw)
        min_bw = min(reshape_bw)
        mean_bw = np.mean(reshape_bw)
        bw_fv = [max_bw, min_bw, mean_bw]

        # Get features for all samples
        features.append(bw_fv)

    print('Bandwidth extracted.')

    return features
```

Extract Features

In [28]:
```
# Define feature vector for classification
features = []

# Create initial feature vectors
fvs = []
# Enable features you wish to extract
fvs.append(extract_mfcc(samples))
fvs.append(extract_sc(samples))
fvs.append(extract_bw(samples))

# Loop over each data sample
for idx, sample in enumerate(samples):
    # Concatenate all feature types for a sample
    sample_features = []
    for fv in fvs:
        sample_features.append(fv[idx])
    # Flatten list of features
    sample_features = sum(sample_features, [])
    # Append sample features to feature vector for classification
    features.append(sample_features)
```

```
Extracting MFCCs...
MFCCs extracted.
Extracting spectral centroid...
Spectral Centroid extracted.
Extracting bandwidth...
Bandwidth extracted.
```

In [29]:
```
# Normalize feature
scaler = RobustScaler()
normalized_features = scaler.fit_transform(features)
```

Train Model

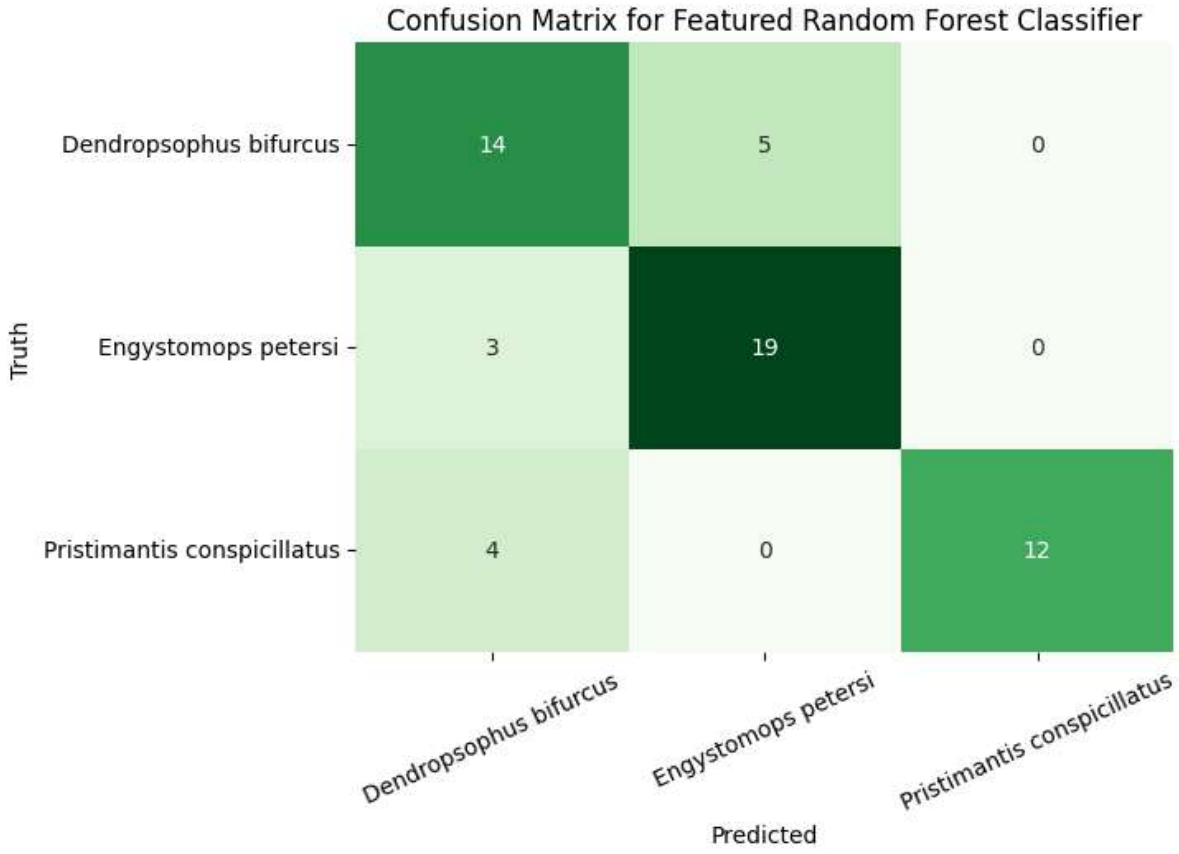
I . Using only feature data

Random Forest

In [30]:
```
# Specify which data to use, these are the only parameters that should change, the
X = normalized_features
y = enc_labels
depth = 5

# Convert X to numpy array if not imputing
X = np.asarray(X)
y = np.asarray(y)

# Define model
clf = RandomForestClassifier(n_estimators=100, max_depth=depth, random_state=0)

# Define number of folds
cv = StratifiedKFold(n_splits=10, shuffle=False)

# Split data, train and test model on 10 folds
split = 1
scores = []
confuse_mat = []
for train_index, test_index in cv.split(np.array(X), np.array(y.argmax(axis=1))):
    print(f"Split {split}/10...")
    x_train, y_train = X[train_index], y[train_index]
    x_test, y_test = X[test_index], y[test_index]


    # Fit model and evaluate it
    clf.fit(x_train, y_train)
    scores.append(clf.score(x_test, y_test))

    # Construct confusion matrix
    y_predict = clf.predict(x_test)
    confuse_mat.append(confusion_matrix(y_test.argmax(axis=1), y_predict.argmax(axi

    # Iterate
    split = split + 1

print("Mean accuracy:" + str(np.mean(scores)))

# Plot heatmap of results
confuse_mat = np.add.reduce(confuse_mat)
#print(confuse_mat)
fig = plt.figure()
sns.heatmap(confuse_mat, annot=True, cmap=plt.cm.Greens, cbar=False)
ticks = np.arange(len(classes))
ticks = ticks + 0.5
plt.xticks(ticks, [i for i in enc_orders], rotation=25)
plt.yticks(ticks, [i for i in enc_orders], rotation=0)
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.title('Confusion Matrix for Featured Random Forest Classifier')
```

```
plt.show()

title = 'FeaturedRandomForest.png'
fig.savefig(title, bbox_inches='tight', dpi = 300)
```

```
Split 1/10...
Split 2/10...
Split 3/10...
Split 4/10...
Split 5/10...
Split 6/10...
Split 7/10...
Split 8/10...
Split 9/10...
Split 10/10...
Mean accuracy:0.75
```



SVM

In [31]:
```
# Specify which data to use, these are the only parameters that should change, the
X = normalized_features
y = enc_labels
y = y.argmax(axis=1)

# Convert X to numpy array if not imputing
X = np.asarray(X)
y = np.asarray(y)

# Define model
clf = SVC(C=1.0, kernel='linear', random_state = 0)
```

```
# Define number of folds
cv = StratifiedKFold(n_splits=10, shuffle=False)

# Split data, train and test model on 10 folds
split = 1
scores = []
confuse_mat = []
for train_index, test_index in cv.split(X,y):
    print(f"Split {split}/10...")
    x_train, y_train = X[train_index], y[train_index]
    x_test, y_test = X[test_index], y[test_index]


    # Fit model and evaluate it
    clf.fit(x_train, y_train)
    scores.append(clf.score(x_test, y_test))

    # Construct confusion matrix
    y_predict = clf.predict(x_test)
    confuse_mat.append(confusion_matrix(y_test, y_predict))

    # Iterate
    split = split + 1

print("Mean accuracy:" + str(np.mean(scores)))

# Plot heatmap of results
confuse_mat = np.add.reduce(confuse_mat)
#print(confuse_mat)
fig = plt.figure()
sns.heatmap(confuse_mat, annot=True, cmap=plt.cm.Greens, cbar=False)
ticks = np.arange(len(classes))
ticks = ticks + 0.5
plt.xticks(ticks, [i for i in enc_orders], rotation=25)
plt.yticks(ticks, [i for i in enc_orders], rotation=0)
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.title('Confusion Matrix for Featured SVM Classifier')
plt.show()

title = 'FeaturedSVM.png'
fig.savefig(title, bbox_inches='tight', dpi = 300)
```

```
Split 1/10...
Split 2/10...
Split 3/10...
Split 4/10...
Split 5/10...
Split 6/10...
Split 7/10...
Split 8/10...
Split 9/10...
Split 10/10...
Mean accuracy:0.7166666666666666
```

KNN

In [42]:
```
# Specify which data to use, these are the only parameters that should change, the
X = normalized_features
y = enc_labels
y = y.argmax(axis=1)
depth = 5

# Convert X to numpy array if not imputing
X = np.asarray(X)
y = np.asarray(y)

# Define model
clf = KNeighborsClassifier(n_neighbors=1, weights='uniform')

# Define number of folds
cv = StratifiedKFold(n_splits=10, shuffle=False)

# Split data, train and test model on 10 folds
split = 1
scores = []
confuse_mat = []
for train_index, test_index in cv.split(X,y):
    print(f"Split {split}/10...")
    x_train, y_train = X[train_index], y[train_index]
    x_test, y_test = X[test_index], y[test_index]
```

```
    # Fit model and evaluate it
    clf.fit(x_train, y_train)
    scores.append(clf.score(x_test, y_test))

    # Construct confusion matrix
    y_predict = clf.predict(x_test)
    confuse_mat.append(confusion_matrix(y_test, y_predict))

    # Iterate
    split = split + 1

print("Mean accuracy:" + str(np.mean(scores)))

# Plot heatmap of results
confuse_mat = np.add.reduce(confuse_mat)
#print(confuse_mat)
fig = plt.figure()
sns.heatmap(confuse_mat, annot=True, cmap=plt.cm.Greens, cbar=False)
ticks = np.arange(len(classes))
ticks = ticks + 0.5
plt.xticks(ticks, [i for i in enc_orders], rotation=25)
plt.yticks(ticks, [i for i in enc_orders], rotation=0)
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.title('Confusion Matrix for Featured KNN Classifier')
plt.show()

title = 'FeaturedKNN.png'
fig.savefig(title, bbox_inches='tight', dpi = 300)
```

```
Split 1/10...
Split 2/10...
Split 3/10...
Split 4/10...
Split 5/10...
Split 6/10...
Split 7/10...
Split 8/10...
Split 9/10...
Split 10/10...
Mean accuracy:0.8633333333333333
```

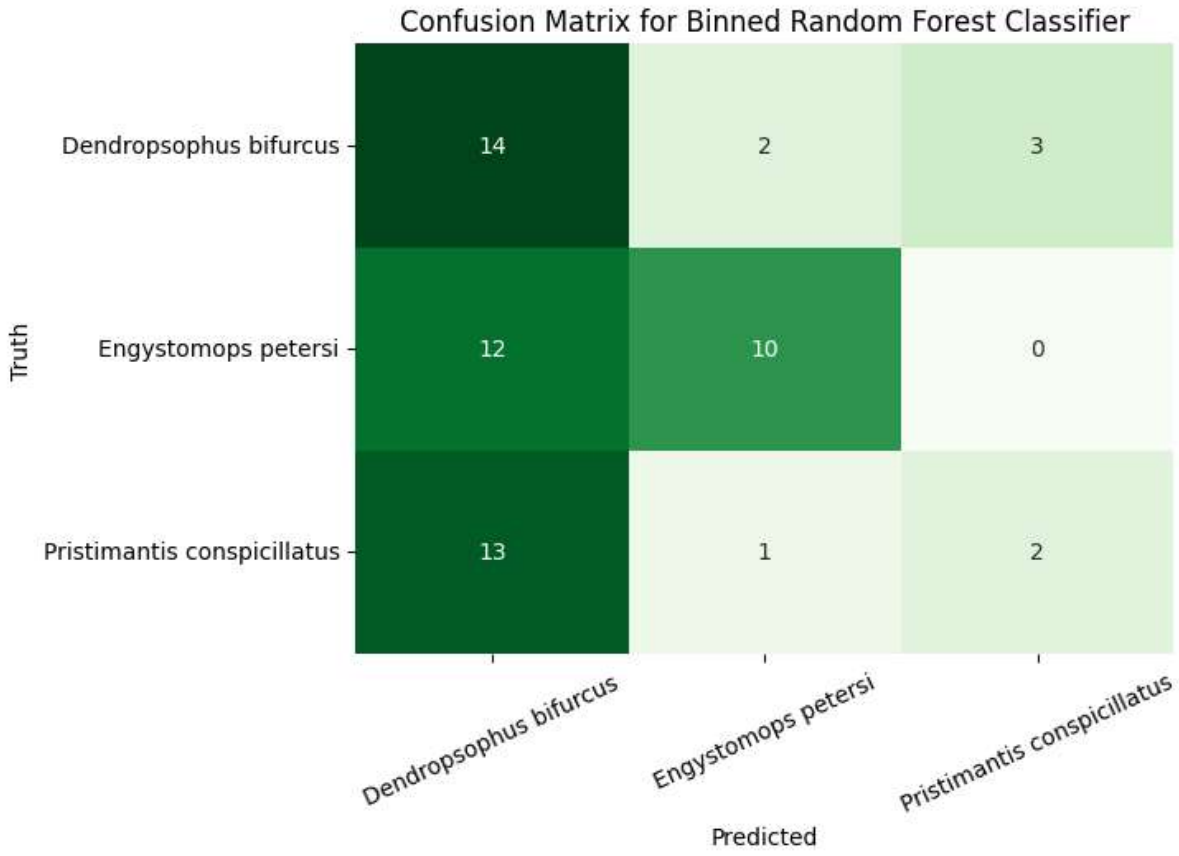
II. Using binned data

In [33]:
```
#Normalize binned data
scaler = RobustScaler()
normalized_bin = scaler.fit_transform(binned_specs)
```

In [34]:
```
# Specify which data to use, these are the only parameters that should change, the
X = normalized_bin
y = enc_labels
depth = 5

# Convert X to numpy array if not imputing
X = np.asarray(X)
y = np.asarray(y)

# # Create our imputer to replace missing values with the mean
# imp = SimpleImputer(missing_values=np.nan, strategy='mean')
# imp = imp.fit(X)

# # Impute our data
# X_imp = imp.transform(X)

# Define model
clf = RandomForestClassifier(max_depth=depth, random_state=0)

# Define number of folds
cv = StratifiedKFold(n_splits=10, shuffle=False)

# Split data, train and test model on 10 folds
```

```
split = 1
scores = []
confuse_mat = []
for train_index, test_index in cv.split(np.array(X), np.array(y.argmax(axis=1))):
    print(f"Split {split}/10...")
    x_train, y_train = X[train_index], y[train_index]
    x_test, y_test = X[test_index], y[test_index]


    # Fit model and evaluate it
    clf.fit(x_train, y_train)
    scores.append(clf.score(x_test, y_test))

    # Construct confusion matrix
    y_predict = clf.predict(x_test)
    confuse_mat.append(confusion_matrix(y_test.argmax(axis=1), y_predict.argmax(axi

    # Iterate
    split = split + 1

print("Mean accuracy:" + str(np.mean(scores)))

# Plot heatmap of results
confuse_mat = np.add.reduce(confuse_mat)
#print(confuse_mat)
fig = plt.figure()
sns.heatmap(confuse_mat, annot=True, cmap=plt.cm.Greens, cbar=False)
ticks = np.arange(len(classes))
ticks = ticks + 0.5
plt.xticks(ticks, [i for i in enc_orders], rotation=25)
plt.yticks(ticks, [i for i in enc_orders], rotation=0)
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.title('Confusion Matrix for Binned Random Forest Classifier')
plt.show()

title = 'BinnedRandomForest.png'
fig.savefig(title, bbox_inches='tight', dpi = 300)
```

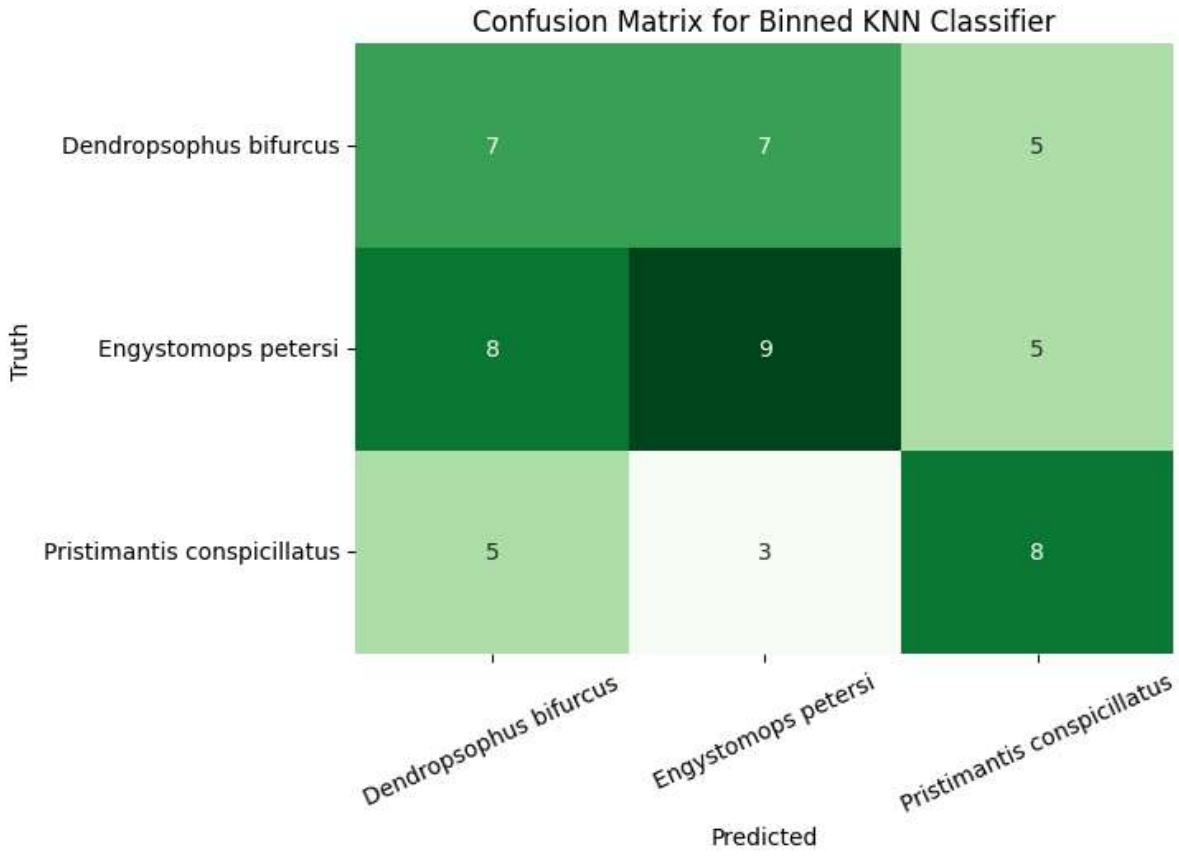
```
Split 1/10...
Split 2/10...
Split 3/10...
Split 4/10...
Split 5/10...
Split 6/10...
Split 7/10...
Split 8/10...
Split 9/10...
Split 10/10...
Mean accuracy:0.33333333333333337
```



In [35]:
```
# Specify which data to use, these are the only parameters that should change, the
X = normalized_bin
y = enc_labels
y = y.argmax(axis=1)
depth = 5

# Convert X to numpy array if not imputing
X = np.asarray(X)
y = np.asarray(y)

# Define model
clf = KNeighborsClassifier(n_neighbors=1, weights='uniform')

# Define number of folds
cv = StratifiedKFold(n_splits=10, shuffle=False)

# Split data, train and test model on 10 folds
split = 1
scores = []
confuse_mat = []
for train_index, test_index in cv.split(X,y):
    print(f"Split {split}/10...")
    x_train, y_train = X[train_index], y[train_index]
    x_test, y_test = X[test_index], y[test_index]


    # Fit model and evaluate it
    clf.fit(x_train, y_train)
    scores.append(clf.score(x_test, y_test))
```

```
    # Construct confusion matrix
    y_predict = clf.predict(x_test)
    confuse_mat.append(confusion_matrix(y_test, y_predict))

    # Iterate
    split = split + 1

print("Mean accuracy:" + str(np.mean(scores)))

# Plot heatmap of results
confuse_mat = np.add.reduce(confuse_mat)
#print(confuse_mat)
fig = plt.figure()
sns.heatmap(confuse_mat, annot=True, cmap=plt.cm.Greens, cbar=False)
ticks = np.arange(len(classes))
ticks = ticks + 0.5
plt.xticks(ticks, [i for i in enc_orders], rotation=25)
plt.yticks(ticks, [i for i in enc_orders], rotation=0)
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.title('Confusion Matrix for Binned KNN Classifier')
plt.show()

title = 'BinnedKNN.png'
fig.savefig(title, bbox_inches='tight', dpi = 300)
```

```
Split 1/10...
Split 2/10...
Split 3/10...
Split 4/10...
Split 5/10...
Split 6/10...
Split 7/10...
Split 8/10...
Split 9/10...
Split 10/10...
Mean accuracy:0.42333333333333334
```



III. Using binned + feature data

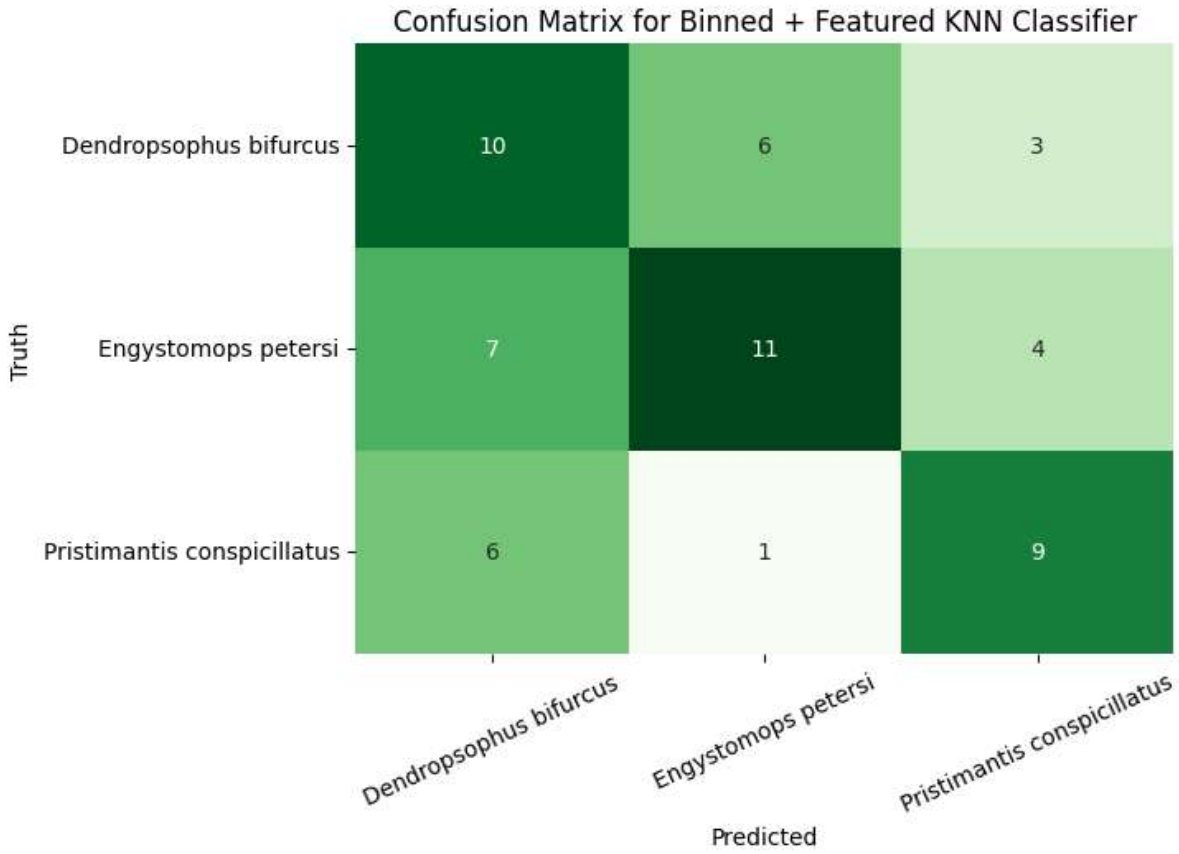
In [36]:
```
# Combine features and binned data

all_features =[]
for i in range(len(features)):
    l=[features[i],list(binned_specs[i])]
    l=[item for sublist in l for item in sublist]
    all_features.append(l)
all_features=np.array(all_features, dtype = object)
```

In [37]:
```
## Normalize all features

scaler = RobustScaler()
all_features = scaler.fit_transform(all_features)
```

In [38]:
```
# Specify which data to use, these are the only parameters that should change, the
X = all_features
y = enc_labels
y = y.argmax(axis=1)
depth = 5

# Convert X to numpy array if not imputing
X = np.asarray(X)
y = np.asarray(y)

# Define model
clf = KNeighborsClassifier(n_neighbors=1, weights='uniform')
```

```
# Define number of folds
cv = StratifiedKFold(n_splits=10, shuffle=False)

# Split data, train and test model on 10 folds
split = 1
scores = []
confuse_mat = []
for train_index, test_index in cv.split(X,y):
    print(f"Split {split}/10...")
    x_train, y_train = X[train_index], y[train_index]
    x_test, y_test = X[test_index], y[test_index]


    # Fit model and evaluate it
    clf.fit(x_train, y_train)
    scores.append(clf.score(x_test, y_test))

    # Construct confusion matrix
    y_predict = clf.predict(x_test)
    confuse_mat.append(confusion_matrix(y_test, y_predict))

    # Iterate
    split = split + 1

print("Mean accuracy:" + str(np.mean(scores)))

# Plot heatmap of results
confuse_mat = np.add.reduce(confuse_mat)
#print(confuse_mat)
fig = plt.figure()
sns.heatmap(confuse_mat, annot=True, cmap=plt.cm.Greens, cbar=False)
ticks = np.arange(len(classes))
ticks = ticks + 0.5
plt.xticks(ticks, [i for i in enc_orders], rotation=25)
plt.yticks(ticks, [i for i in enc_orders], rotation=0)
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.title('Confusion Matrix for Binned + Featured KNN Classifier')
plt.show()

title = 'Binned_Featured_KNN.png'
fig.savefig(title, bbox_inches='tight', dpi = 300)
```

```
Split 1/10...
Split 2/10...
Split 3/10...
Split 4/10...
Split 5/10...
Split 6/10...
Split 7/10...
Split 8/10...
Split 9/10...
Split 10/10...
Mean accuracy:0.5233333333333333
```



Save Model

In [39]:
```
import pickle

pkl_filename = "pickle_model.pkl"

# Note, for now this will only store the file in volitle memory in the notebook
# To save permanently, most add file path to drive
with open(pkl_filename, 'wb') as file:
  pickle.dump(clf, file)
```

Load Model

In [40]:
```
with open(pkl_filename, 'rb') as file:
  pickle_model = pickle.load(file)
```

In [41]:
```
# Test saved model
print(pickle_model.score(x_test, y_test))
```

```
0.4
```