Udacity Machine Learning Nanodegree: Capstone Project

A Comparative Study of Machine Learning Methods for Avian Speaker Recognition

Abstract

We perform speaker recognition on the calls of Thrush Nightingales (Luscinia luscinia) with a variety of machine learning algorithms. This involves data preparation and call extraction, feature generation and the use of standard libraries for supervised classification. The features used are the Mel-frequency Cepstral Coefficients (MFCCs), a widely employed feature in speech domains. We analyze our model by considering robustness to training data selection and size. Future directions and domain specific challenges and sensitivies are considered.

Contents

1. Data Exploration

In this section we import the data and give an initial exploration via some visualisations.

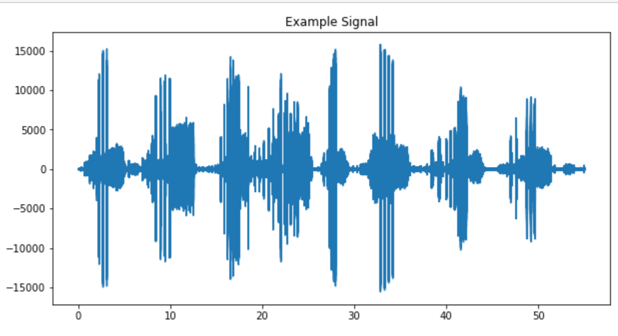
Example

In [1]:
```
import sklearn
import numpy as np
import pandas as pd
import scipy.io.wavfile as sp
import matplotlib.pyplot as plt
from os import listdir
import python_speech_features as psf
```

Plot of example signal.

In [2]:
```
raw_signal = sp.read("./data/XC247266.wav")
signal = (raw_signal[1][:, 0] + raw_signal[1][:, 1]) / 2
```

In [3]:
```
plt.figure(0, figsize = (10, 5))
time = np.linspace(0, len(signal), num=len(signal)) / raw_signal[0]
plt.title('Example Signal')
plt.plot(time, signal)
plt.show()
```



Call Detection

As illustrated by the above figure, we must also detect calls to identify the relevent portions of the time series.

In [4]:
```
def call_detector(signal, threshold, dispersion, rate, window):
    separation = dispersion * rate
    noise = separation + 1
    output = []
    for i in xrange(len(signal) - window - 1):
        if np.mean(signal[i : i + window]) < threshold:
            noise += 1
        elif np.mean(signal[i : i + window]) > threshold:
            if noise > separation:
                output.append(i)
            noise = 0
    return output
```

The above algorithm is used to compute call onset and offset times.

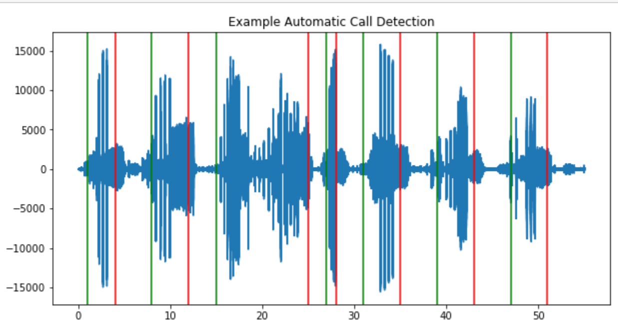
In [5]:
```
call_onsets = np.asarray(call_detector(signal, 200, 2, 44100, 30))
call_offsets = np.asarray(call_detector(signal[::-1], 200, 2, 44100, 30))
for i in xrange(len(call_offsets)):
    call_offsets[i] = len(signal) - call_offsets[i] - 1
call_offsets = np.asarray(call_offsets[::-1])
```

Example output from the call detection process.

In [6]:
```
[[call_onsets[i], call_offsets[i]] for i in xrange(min(len(call_onsets), len(call_offsets)))]
```

Out[6]:
```
[[64772, 208065],
 [356571, 553274],
 [683573, 1110755],
 [1205287, 1278861],
 [1401718, 1581712],
 [1728687, 1921742],
 [2072717, 2256330]]
```

In [7]:
```
plt.figure(0, figsize=(10, 5))
time = np.linspace(0, len(signal), num=len(signal)) / raw_signal[0]
plt.title('Example Automatic Call Detection')
plt.plot(time, signal)
for onset in call_onsets:
    plt.axvline(x = onset / 44100, color = 'g')
for offset in call_offsets:
    plt.axvline(x = offset / 44100, color = 'r')
plt.show()
```



Read audio and construct dictionary to store numeric representation

In [8]:
```
audio_files = listdir("./data")
rate = sp.read("./data/" + audio_files[1])[0]
ids = [x[0:7] for x in listdir("./data")]
signals = dict.fromkeys(ids)
for i, f in enumerate(audio_files):
    raw_signal = sp.read("./data/" + f)
    signals[ids[i]] = (raw_signal[1][:, 0] + raw_signal[1][:, 1]) / 2
signals
```

Out[8]:
```
{'XC24726': array([  0,  -1,  -1, ..., -16, -38, -34], dtype=int16),
 'XC24779': array([-26, -26,  -7, ..., -41, -63, -51], dtype=int16),
 'XC29044': array([ 51,  71,  61, ..., -60, -46, -13], dtype=int16),
 'XC29625': array([ -1,   1,  -1, ..., -48, -21, -17], dtype=int16),
 'XC30043': array([0, 0, 0, ..., 0, 0, 0], dtype=int16),
 'XC31658': array([  -1,   -1,    1, ...,    8,  -80, -111], dtype=int16),
 'XC31879': array([  0,   0,   0, ..., -24,  61, 108], dtype=int16),
 'XC36993': array([ 0,  0,  0, ..., -2, -2, -1], dtype=int16),
 'XC37067': array([    0,     0,     0, ..., -3882, -4112, -4262], dtype=int16),
 'XC37080': array([ 24,  -6, -47, ...,   0,   0,   0], dtype=int16)}
```

In [9]:
```
def extract_calls(signal, threshold, dispersion, rate, window):
    # detect call starts in forward and back direction
    call_onsets = np.asarray(call_detector(signal, threshold, dispersion, rate, window))
    call_offsets = np.asarray(call_detector(signal[::-1], threshold, dispersion, rate, window))

    # align indices and order of call offsets
    for i in xrange(len(call_offsets)):
        call_offsets[i] = len(signal) - call_offsets[i] - 1
    call_offsets = np.asarray(call_offsets[::-1])

    # combine to form a list of lists containing call onset and offset times
    return [[call_onsets[i], call_offsets[i]] for i in xrange(min(len(call_onsets), len(call_offsets)))]
```

The following collects the call offset and onset times for each recording/identity.

In [10]:
```
calls = dict.fromkeys(ids)
for identity in ids:
    calls[identity] = extract_calls(signals[identity], 200, 2, rate, 30)
```

Next, we remove the portions of noise or silence from the signal.

In [11]:
```
filt_signals = dict.fromkeys(ids)
for i in xrange(len(ids)):
    filt_signals[ids[i]] = np.concatenate([signals[ids[i]][x[0]:x[1]] for x in calls[ids[i]]])
filt_signals
```

Out[11]:
```
{'XC24726': array([-655, -156,  405, ...,  949,  667,  306], dtype=int16),
 'XC24779': array([ 437, 1923, 2374, ...,  855,  719,  528], dtype=int16),
 'XC29044': array([1696, 1942, 1784, ..., -961, 1308, 2552], dtype=int16),
 'XC29625': array([-2581,  -440,  1861, ...,   882,   816,   674], dtype=int16),
 'XC30043': array([  81,  330,  536, ..., 1098,  934,  652], dtype=int16),
 'XC31658': array([-1445,  -963,  -343, ...,   595,   293,   -38], dtype=int16),
 'XC31879': array([ 297,  228,  175, ..., -165, -308, -362], dtype=int16),
 'XC36993': array([  87,    5,  -87, ..., 1047, 1122, 1005], dtype=int16),
 'XC37067': array([15675, 14476, 14413, ...,  -248,  -268,  -279], dtype=int16),
 'XC37080': array([-647,  466, 1492, ..., 1891, 1913, 1376], dtype=int16)}
```

2. Feature Generation

In this section we transform the above signal time series data into suitable features for use in classification. In particular, we generate the Mel-frequency Cepstral Coefficients (MFCCs), a commonly used feature in speech classification. This is an important step, as a large time series passed to a machine learning algorithm would have prohibitively large dimension - as per the curse of dimensionality. However, the MFCC feature vector reduces this to have only 13 dimensions.

In [12]:
```
from sklearn.model_selection import train_test_split
```

In [13]:
```
features = np.zeros(14)
for i in xrange(len(ids)):
    mf = psf.mfcc(filt_signals[ids[i]], samplerate = rate, winlen=0.025, winstep=0.01, numcep=13,
                  nfilt=26, nfft=512, lowfreq=0, highfreq = None, preemph=0.97,
                  ceplifter=22, appendEnergy=True)
    a = np.zeros((np.shape(mf)[0], 14))
    a[:, : -1] = mf
    a[:, -1] = np.repeat(i, np.shape(mf)[0])
    features = np.vstack((features, a))
features = features[1:, ]
```

In [15]:
```
X_train, X_test, y_train, y_test = train_test_split(features[:, 0:12], features[:, 13],
                                                    test_size=0.33, random_state=42)
```

3. Classification and Optimisation

In [17]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn import svm
from sklearn.ensemble import AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics
from sklearn.model_selection import GridSearchCV
```

Decision Tree

In [16]:
```
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)
dt_acc = dt.score(X_test, y_test)
dt_f1 = metrics.f1_score(y_test, dt.predict(X_test), average = 'macro')
print [dt_acc, dt_f1]
```

```
[0.76317379338121771, 0.6957135133088832]
```

Exhaustive search parameter optimisation with GridSearchCV.

In [19]:
```
parameters = [
  {'criterion': ['gini', 'entropy'], 'min_samples_split' : [2, 3, 4, 5, 6, 7, 8],
   'splitter' : ['best', 'random']},
]
dt_gs = GridSearchCV(DecisionTreeClassifier(), parameters)
dt_gs.fit(X_train, y_train)
dt_opt_acc = dt_gs.score(X_test, y_test)
dt_opt_f1 = metrics.f1_score(y_test, dt_gs.predict(X_test), average = 'macro')
print [dt_opt_acc, dt_opt_f1]
```

```
[0.76639591864133716, 0.700308975399785O9]
```

Random Forest

In [21]:
```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
rf_acc = rf.score(X_test, y_test)
rf_f1 = metrics.f1_score(y_test, rf.predict(X_test), average = 'macro')
print [rf_acc, rf_f1]
```

```
[0.83808820567899578, 0.78196962371116441]
```

Exhaustive search parameter optimisation with GridSearchCV.

In [19]:
```
parameters = [
  {'criterion': ['gini', 'entropy'], 'min_samples_split' : [2, 4, 6, 8],
   'n_estimators': [10, 100, 200, 400, 500]}]
rf_gs = GridSearchCV(RandomForestClassifier(), parameters)
rf_gs.fit(X_train, y_train)
rf_opt_acc = rf_gs.score(X_test, y_test)
rf_opt_f1 = metrics.f1_score(y_test, rf_gs.predict(X_test), average = 'macro')
print [rf_opt_acc, rf_opt_f1]
```

```
[0.88574880848492987, 0.84150675534700559]
```

Support Vector Machine

In [20]:
```
sm = svm.SVC()
sm_fit = sm.fit(X_train, y_train)
print sm_fit.score(X_test, y_test)
```

```
0.201651339196
```

Exhaustive search parameter optimisation with GridSearchCV.

In [21]:
```
parameters = [{'C' : [0.5, 1.0, 1.5, 2.0]}]
sm_gs = GridSearchCV(svm.SVC(), parameters)
sm_gs.fit(X_train, y_train)
sm_opt_acc = sm_gs.score(X_test, y_test)
print sm_opt_acc
```

```
0.20265825334
```

Gaussian Naive Bayes

In [22]:
```
gnb = GaussianNB()
gnb.fit(X_train, y_train)
gnb_acc = gnb.score(X_test, y_test)
gnb_f1 = metrics.f1_score(y_test, gnb.predict(X_test), average = 'macro')
print [gnb_acc, gnb_f1]
```

```
[0.6198563469154863, 0.53774934409910324]
```

Optimisation not applicable, due to lack of informative priors.

AdaBoost

In [23]:
```
ab = AdaBoostClassifier()
ab.fit(X_train, y_train)
ab_acc = ab.score(X_test, y_test)
ab_f1 = metrics.f1_score(y_test, ab.predict(X_test), average = 'macro')
print [ab_acc, ab_f1]
```

```
[0.62858293616164329, 0.52879537935931309]
```

Exhaustive search parameter optimisation with GridSearchCV.

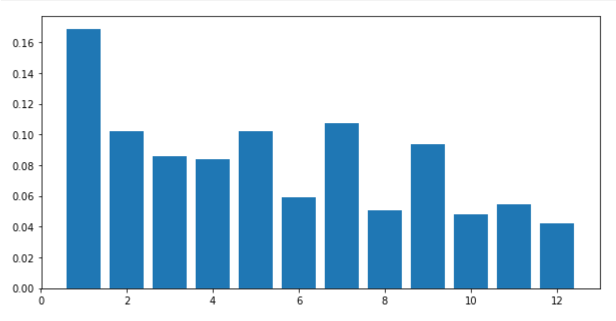
In [24]:
```
parameters = [{'n_estimators' : [50, 100, 200, 500], 'learning_rate' : [0.5, 1.0, 1.5, 2.0]}]
ab_gs = GridSearchCV(AdaBoostClassifier(), parameters)
ab_gs.fit(X_train, y_train)
ab_opt_acc = ab_gs.score(X_test, y_test)
ab_opt_f1 = metrics.f1_score(y_test, ab_gs.predict(X_test), average = 'macro')
print [ab_opt_acc, ab_opt_f1]
```

```
[0.64127005437336382, 0.49441287605529649]
```

5. Visualisations

The following bar chart visualises the relative feature importances for the fitted Random Forest classifier.

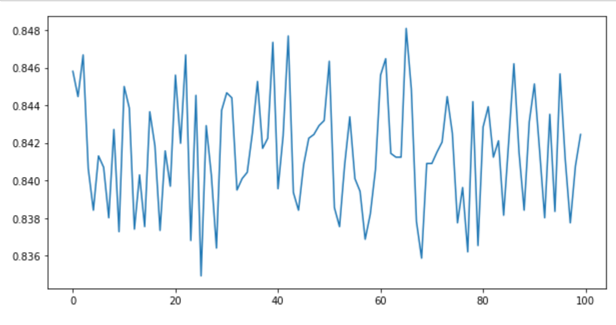
In [32]:
```
plt.figure(0, figsize = (10, 5))
plt.bar(range(1, 13), rf.feature_importances_)
plt.show()
```



Next, we visualise performance over successive distinct training and testing splits to test robustness.

In [33]:
```
out = np.zeros(100)
for i in xrange(100):
    X_train, X_test, y_train, y_test = train_test_split(features[:, 0:12], features[:, 13],
                                                        test_size=0.33, random_state= i)
    rf = RandomForestClassifier()
    rf.fit(X_train, y_train)
    out[i] = rf.score(X_test, y_test)
```

In [35]:
```
plt.figure(0, figsize = (10, 5))
plt.plot(range(100), out)
plt.show()
```



Finally, we consider how the quantity fo training data effects classification accuracy.

In [42]:
```
out2 = np.zeros(9)
for i in xrange(10, 100, 10):
    X_train, X_test, y_train, y_test = train_test_split(features[:, 0:12], features[:, 13],
                                                        test_size = i/100.0, random_state= 42)
    rf = RandomForestClassifier()
    rf.fit(X_train, y_train)
    out2[i/10 - 1] = rf.score(X_test, y_test)
```

In [43]:
```
plt.figure(0, figsize = (10, 5))
plt.plot(range(10, 100, 10), out2)
plt.show()
```