Music Information Retrieval A3

James Davidson - V00812527

March 17^{th}, 2017

All up-to-date code for this can be found at https://github.com/jamesthomasdavidson/Music-Classifier

Question 1

```python
import numpy as np
from sklearn import svm, datasets, linear_model
from sklearn.externals.joblib import Memory
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.neighbors import KNeighborsClassifier

def get_data():
    return datasets.load_svmlight_file('genres3.libsvm')

def SVC():
    X, y = get_data()
    print('Support Vector Machine')
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
    classifier = svm.SVC(kernel='linear', C=.8)
    y_pred = classifier.fit(X_train, y_train).predict(X_test)
    print("Confusion matrix: \n" + str(confusion_matrix(y_test, y_pred)))
    print("Accuracy: " + str(classifier.score(X,y)) + '\n\n')

def SGD():
    X, y = get_data()
    print('Stochastic Gradient Descent')
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
    classifier = linear_model.SGDClassifier()
    y_pred = classifier.fit(X_train, y_train).predict(X_test)
    print("Confusion matrix: \n" + str(confusion_matrix(y_test, y_pred)))
    print("Accuracy: " + str(classifier.score(X,y)) + '\n\n')

def NN():
    X, y = get_data()
    print('Nearest Neighbours')
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
    classifier = KNeighborsClassifier(n_neighbors=2)
    y_pred = classifier.fit(X_train, y_train).predict(X_test)
    print("Confusion matrix: \n" + str(confusion_matrix(y_test, y_pred)))
    print("Accuracy: " + str(classifier.score(X,y)) + '\n\n')

SVC()
SGD()
NN()
```

```
./mkcollection -c classical.mf -l classical ../../genres/classical
./mkcollection -c rock.mf -l rock ../../genres/rock
./mkcollection -c hiphop.mf -l hiphop ../../genres/hiphop
cat cl.mf hi.mf ro.mf > genres3.mf
bextract -sv genres3.mf -w genres3.arff
```

Using Weka:

===========**ZeroR**==========

```
Correctly Classified Instances        100              33.3333 %
Incorrectly Classified Instances      200              66.6667 %
Kappa statistic                        0
Mean absolute error                    0.4444
Root mean squared error                0.4714
Relative absolute error                100 %
Root relative squared error            100 %
Total Number of Instances              300

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    1.000    0.333      1.000    0.500      0.000    0.500     0.333     classical
                 0.000    0.000    0.000      0.000    0.000      0.000    0.500     0.333     hiphop
                 0.000    0.000    0.000      0.000    0.000      0.000    0.500     0.333     rock
Weighted Avg.    0.333    0.333    0.111      0.333    0.167      0.000    0.500     0.333

=== Confusion Matrix ===

   a   b   c   <-- classified as
 100   0   0 |   a = classical
 100   0   0 |   b = hiphop
 100   0   0 |   c = rock
```

======**NaiveBayesSimple**======

```
Correctly Classified Instances        253              84.6154 %
Incorrectly Classified Instances       46              15.3846 %
Kappa statistic                        0.7692
Mean absolute error                    0.1021
Root mean squared error                0.3158
Relative absolute error               22.9815 %
Root relative squared error           66.99   %
Total Number of Instances             299
```

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.949	0.035	0.931	0.949	0.940	0.910	0.987	0.975	classical
	0.710	0.025	0.934	0.710	0.807	0.742	0.974	0.938	hiphop
	0.880	0.171	0.721	0.880	0.793	0.681	0.921	0.778	rock
Weighted Avg.	0.846	0.077	0.862	0.846	0.846	0.777	0.961	0.897	

=== Confusion Matrix ===

```
  a  b  c   <-- classified as
 94  0  5 |  a = classical
  0 71 29 |  b = hiphop
  7  5 88 |  c = rock
```

===========**J48**============

Correctly Classified Instances	248	82.6667 %
Incorrectly Classified Instances	52	17.3333 %
Kappa statistic	0.74	
Mean absolute error	0.1225	
Root mean squared error	0.3324	
Relative absolute error	27.5682 %	
Root relative squared error	70.5044 %	
Total Number of Instances	300	

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.880	0.045	0.907	0.880	0.893	0.842	0.923	0.846	classical
	0.850	0.085	0.833	0.850	0.842	0.761	0.895	0.763	hiphop
	0.750	0.130	0.743	0.750	0.746	0.618	0.796	0.654	rock
Weighted Avg.	0.827	0.087	0.828	0.827	0.827	0.740	0.871	0.754	

=== Confusion Matrix ===

```
  a  b  c   <-- classified as
 88  1 11 |  a = classical
  0 85 15 |  b = hiphop
  9 16 75 |  c = rock
```

===========**SMO**============

Correctly Classified Instances	288	96 %
Incorrectly Classified Instances	12	4 %
Kappa statistic	0.94	
Mean absolute error	0.2311	
Root mean squared error	0.288	
Relative absolute error	52 %	

Root relative squared error 61.101 %
Total Number of Instances 300

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.990	0.000	1.000	0.990	0.995	0.993	0.999	0.998	classical
	0.920	0.015	0.968	0.920	0.944	0.917	0.979	0.938	hiphop
	0.970	0.045	0.915	0.970	0.942	0.912	0.963	0.898	rock
Weighted Avg.	0.960	0.020	0.961	0.960	0.960	0.941	0.980	0.944	

=== Confusion Matrix ===

```
  a  b  c   <-- classified as
 99  0  1 |  a = classical
  0 92  8 |  b = hiphop
  0  3 97 |  c = rock
```

Using scikit-learn:

	Support Vector Machine	Stochastic Gradient Descent	Nearest Neighbours
Confusion matrix:	[[23 0 1] [0 23 3] [0 0 25]]	[[24 0 0] [3 23 0] [7 18 0]]	[[23 0 1] [2 22 2] [3 6 16]]
Accuracy:	0.966666666667	0.64	0.886666666667

Question 2

```
69  #start
70  tag = {'Rap' : 12, 'Pop_Rock' : 1, 'Country' : 3}
71  words = get_words()
72  tracks = get_tracks()
73  labels = genres()
```

```
75  def extract_vocabulary(D):
76      V = []
77      for word in words:
78          V.append(word)
79      return V
80
81  def count_documents(D):
82      return len(D)
83
84  def count_docs_in_class(D, c):
85      return len([get_tracks(genre = c)])
86
87  def concatenate_all_text_in_docs(D, c):
88      text, tracks = [], get_tracks(genre = c)
89      for word in words:
90          for track in tracks:
91              n = track.num_word(words.index(word))
92              while n > 0:
93                  text.append(word)
94                  n = n - 1
95      return text
96
97  def count_tokens_of_terms(text_c, t):
98      return text_c.count(t)
99
100 def extract_tokens_from_doc(V, d):
101     text = []
102     for word in words:
103         n = d.num_word(words.index(word))
104         while n > 0:
105             text.append(word)
106             n = n - 1
107     return text
```

```
109 #train the multinomial model
110 def train_multinomial(C, D):
111     V = extract_vocabulary(D)
112     N = count_documents(D)
113     prior, condprob = dict.fromkeys(C), {}
114     for c in C:
115         N_c = count_docs_in_class(D, c)
116         prior[c] = float(N_c)/N
117         text_c = concatenate_all_text_in_docs(D, c)
118         T = {}
119         for t in V:
120             T[(t,c)] = count_tokens_of_terms(text_c, t)
121         for t in V:
122             condprob[(t,c)] = float(T[(t,c)] + 1)
123                                   /(sum([T[tp,c] + 1 for tp in V]))
124     return V, prior, condprob
125
126 #apply the multinomial on new instance d
127 def apply_multinomial(C, V, prior, condprob, d):
128     W = extract_tokens_from_doc(V, d)
129     score = dict.fromkeys(C, 0)
130     for c in C:
131         score[c] = np.log(prior[c])
132         for t in W:
133             score[c] += np.log(condprob[(t,c)])
134     argmax, max_score = '', -np.inf
135     for c in C:
136         if score[c] > max_score:
137             max_score = score[c]
138             argmax = c
139     return argmax
```

The probability of a word given a genre, pr(word | genre), is obtaining by condprob[(word,genre)] which can be obtained by calling train_multinomial(C, D). (lines 110 - 124 above) (Resulting table below)

(de,Rap,0.072)	(de,Pop_Rock,0.026)	(de,Country,0.002)
(niggaz,Rap,0.042)	(niggaz,Pop_Rock,0.003)	(niggaz,Country,0.001)
(ya,Rap,0.093)	(ya,Pop_Rock,0.012)	(ya,Country,0.012)
(und,Rap,0.047)	(und,Pop_Rock,0.019)	(und,Country,0.000)
(yall,Rap,0.042)	(yall,Pop_Rock,0.002)	(yall,Country,0.005)
(ich,Rap,0.067)	(ich,Pop_Rock,0.030)	(ich,Country,0.000)
(fuck,Rap,0.069)	(fuck,Pop_Rock,0.022)	(fuck,Country,0.002)
(shit,Rap,0.093)	(shit,Pop_Rock,0.006)	(shit,Country,0.003)
(yo,Rap,0.078)	(yo,Pop_Rock,0.009)	(yo,Country,0.005)
(bitch,Rap,0.059)	(bitch,Pop_Rock,0.004)	(bitch,Country,0.001)
(end,Rap,0.014)	(end,Pop_Rock,0.037)	(end,Country,0.022)
(wait,Rap,0.013)	(wait,Pop_Rock,0.045)	(wait,Country,0.026)
(again,Rap,0.019)	(again,Pop_Rock,0.048)	(again,Country,0.053)
(light,Rap,0.016)	(light,Pop_Rock,0.044)	(light,Country,0.032)
(eye,Rap,0.023)	(eye,Pop_Rock,0.056)	(eye,Country,0.042)
(noth,Rap,0.012)	(noth,Pop_Rock,0.038)	(noth,Country,0.021)
(lie,Rap,0.009)	(lie,Pop_Rock,0.038)	(lie,Country,0.017)
(fall,Rap,0.011)	(fall,Pop_Rock,0.050)	(fall,Country,0.031)
(our,Rap,0.023)	(our,Pop_Rock,0.062)	(our,Country,0.043)
(away,Rap,0.017)	(away,Pop_Rock,0.079)	(away,Country,0.054)
(gone,Rap,0.016)	(gone,Pop_Rock,0.035)	(gone,Country,0.044)
(good,Rap,0.029)	(good,Pop_Rock,0.033)	(good,Country,0.062)
(night,Rap,0.023)	(night,Pop_Rock,0.063)	(night,Country,0.071)
(blue,Rap,0.007)	(blue,Pop_Rock,0.015)	(blue,Country,0.037)

(home,Rap,0.015) (long,Rap,0.017) (littl,Rap,0.025) (well,Rap,0.022) (heart,Rap,0.015) (old,Rap,0.011)	(home,Pop_Rock,0.034) (long,Pop_Rock,0.037) (littl,Pop_Rock,0.038) (well,Pop_Rock,0.044) (heart,Pop_Rock,0.052) (old,Pop_Rock,0.019)	(home,Country,0.055) (long,Country,0.065) (littl,Country,0.075) (well,Country,0.065) (heart,Country,0.087) (old,Country,0.066)

More code used in Q2:

```
def run_multinomial():

    C, D, k = ['Rap', 'Pop_Rock', 'Country'], get_tracks(), 10

    #applies the multinomial and prints the data
    def print_statistics(data):
        confusion_matrix = [[0,0,0,C[2]],[0,0,0,C[1]],[0,0,0,C[0]]]
        total = 0
        for d in [d for d in data if d.genre == tag['Rap']]:
            c = apply_multinomial(C, V, prior, condprob, d)
            if c == C[0]:
                total += 1
                confusion_matrix[2][0] += 1
            elif c == C[1]:
                confusion_matrix[2][1] += 1
            elif c == C[2]:
                confusion_matrix[2][2] += 1

        for d in [d for d in data if d.genre == tag['Pop_Rock']]:
            c = apply_multinomial(C, V, prior, condprob, d)
            if c == C[0]:
                confusion_matrix[1][0] += 1
            elif c == C[1]:
                total += 1
                confusion_matrix[1][1] += 1
            elif c == C[2]:
                confusion_matrix[1][2] += 1

        for d in [d for d in data if d.genre == tag['Country']]:
            c = apply_multinomial(C, V, prior, condprob, d)
            if c == C[0]:
                confusion_matrix[0][0] += 1
            elif c == C[1]:
                confusion_matrix[0][1] += 1
            elif c == C[2]:
                total += 1
                confusion_matrix[0][2] += 1

        print("classification accuracy: " + str(total*1.0/count_documents(data)))
        print('|%10s |%10s |%10s |' % (C[0],C[1],C[2]))
        print('-----------------------------------------------')
        for row in confusion_matrix:
            for col in row:
                print('|%10s' % col),
            print('')
        print('\n')

    #test on kth fold and train on the remaining k-1 folds for each k
    random.shuffle(D)
    V, prior, condprob = train_multinomial(C, D)
    for i in range(k):
        k_folds = np.split(D, k)
        test_data = k_folds.pop(i)
        train_data = [j for i in k_folds for j in i]
        V, prior, condprob = train_multinomial(C, train_data)
        print_statistics(test_data)
```

Confusion Matrix for Naive Bayes classifier.

Using all sets as training data and test data:

classification accuracy: 0.68

Rap	Pop_Rock	Country	
25	206	769	Country
114	507	379	Pop_Rock
748	118	134	Rap

Using k = 10 folds, use 9 sets for training and 1 for testing. Run 10 times to try each k^{th} fold as test data with the remaining k-1 folds as training data:

classification accuracy: 0.68

Rap	Pop_Rock	Country	
3	23	70	Country
11	56	33	Pop_Rock
78	15	11	Rap

classification accuracy: 0.683333333333

Rap	Pop_Rock	Country	
4	23	84	Country
10	46	36	Pop_Rock
75	10	12	Rap

classification accuracy: 0.62

Rap	Pop_Rock	Country	
2	20	58	Country
15	52	46	Pop_Rock
76	15	16	Rap

classification accuracy: 0.68

Rap	Pop_Rock	Country	
2	23	86	Country
13	48	38	Pop_Rock
70	8	12	Rap

classification accuracy: 0.67

Rap	Pop_Rock	Country	
2	21	77	Country
14	44	37	Pop_Rock
80	13	12	Rap

classification accuracy: 0.71

Rap	Pop_Rock	Country	
3	18	81	Country
7	53	37	Pop_Rock
79	10	12	Rap

classification accuracy: 0.653333333333

Rap	Pop_Rock	Country	
2	23	81	Country
15	52	37	Pop_Rock
63	12	15	Rap

classification accuracy: 0.666666666667

Rap	Pop_Rock	Country	
1	13	77	Country
7	46	46	Pop_Rock
77	18	15	Rap

classification accuracy: 0.693333333333

Rap	Pop_Rock	Country	
2	23	72	Country
13	64	33	Pop_Rock
72	11	10	Rap

classification accuracy: 0.69

Rap	Pop_Rock	Country	
4	19	83	Country
9	46	36	Pop_Rock
78	6	19	Rap

Making randomly generated tracks using the probability distribution of a word occurring given a genre:

```
def get_probabilistic_word(genre = None):
    assert(genre is not None)
    prob_dist = []
    for word in words:
        prob_dist.append(condprob[(word, genre)])
    return np.random.choice(words, p = prob_dist)
n_lyrics, n_songs = 20, 5
generated_tracks = []
for e, c in enumerate(C):
    for i in range(n_songs):
        t = Track(n_songs*e+i, [0]*30, c)
        for j in range(n_lyrics):
            t.add_word(words.index(get_probabilistic_word(c)))
        generated_tracks.append(t)
for t in generated_tracks:
    t.print_track()
```

ID: 0 Genre: 12
Feature Vector: [0, 2, 0, 2, 0, 3, 2, 2, 1, 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 2, 0]

ID: 1 Genre: 12
Feature Vector: [0, 2, 1, 2, 2, 1, 3, 5, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]

ID: 2 Genre: 12
Feature Vector: [1, 1, 0, 0, 2, 1, 1, 4, 3, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0]

ID: 3 Genre: 12
Feature Vector: [3, 0, 1, 2, 1, 0, 3, 2, 1, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 2, 0, 0, 0, 0]

ID: 4 Genre: 12
Feature Vector: [2, 2, 1, 1, 2, 0, 1, 1, 2, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0]

ID: 5 Genre: 1
Feature Vector: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 2, 0, 2, 2, 1, 0, 3, 1, 1, 1, 2, 0, 0, 1, 0, 0, 1, 0]

ID: 6 Genre: 1
Feature Vector: [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 2, 1, 2, 3, 1, 0, 0, 3, 0, 2, 0, 1, 2, 1, 0, 0, 0, 0]

ID: 7 Genre: 1
Feature Vector: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 1, 0, 1, 0, 0, 2, 2, 1, 0, 0, 1, 0, 1, 1, 2, 2, 0, 3]

ID: 8 Genre: 1
Feature Vector: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 1, 0, 0, 2, 3, 2, 1, 1, 3, 1, 2, 0, 0, 0, 0, 1]

ID: 9 Genre: 1

Feature Vector: [0, 0, 0, 1, 0, 2, 0, 1, 1, 0, 1, 0, 2, 0, 0, 0, 2, 1, 0, 1, 1, 1, 1, 0, 2, 0, 1, 1, 1, 0]

ID: 10 Genre: 3
Feature Vector: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 3, 2, 0, 0, 1, 2, 0, 1, 1, 0, 0, 0, 0, 1, 2, 4, 0]

ID: 11 Genre: 3
Feature Vector: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 1, 0, 0, 0, 1, 2, 0, 2, 1, 2, 2, 0, 0, 1, 3, 1, 1]

ID: 12 Genre: 3
Feature Vector: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 1, 3, 0, 0, 0, 1, 3, 0, 0, 3, 0, 0, 0, 1, 1, 0, 4]

ID: 13 Genre: 3
Feature Vector: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 2, 0, 0, 1, 2, 0, 1, 2, 2, 0, 4, 3]

ID: 14 Genre: 3
Feature Vector: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 2, 2, 1, 1, 1, 0, 1, 2, 2, 3, 1]

Running classifier on generated data for fun :)

classification accuracy: 0.866666666667

Rap	Pop_Rock	Country	
0	0	5	Country
0	3	2	Pop_Rock
5	0	0	Rap

Happy Marking!