Main

April 1, 2020

1 Project 3 Predictive analytics: model evaluation and selection

1.0.1 Baseline Model : GBM (Claimed test accuracy: 38%)

1.0.2 Advanced Model : Neural Network (Claimed test accuracy: 54%)

1.1 Step 0 : Set up and Import

[1]:
```
import warnings
warnings.filterwarnings("ignore")
```

If you do not have xgboost installation in your python environment.

Please install first, otherwise, ignoring the following step : conda install -c conda-forge xgboost

[1]:
```
conda install -c conda-forge xgboost
```

```
Collecting package metadata (current_repodata.json): done
Solving environment: done

# All requested packages already installed.


Note: you may need to restart the kernel to use updated packages.
```

[3]:
```
import os
import pandas as pd
import numpy as np
import scipy.io as scio
from matplotlib import pyplot as plt
import seaborn as sns
import time
from datetime import datetime
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
```

```
import time
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.externals import joblib
from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
import xgboost
from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, roc_auc_score
from scipy.io import loadmat
import scipy
```

1.2 Step 1: Feature Extraction

[4]:
```
class load():
    def load_data(filename):
        raw_data = pd.read_csv(filename)
        raw_data['filename'] = [str(i).zfill(4)+'.jpg' for i in␣
↪raw_data['Index'].tolist()]
        raw_data['pointsname'] = [str(i).zfill(4)+'.mat' for i in␣
↪raw_data['Index'].tolist()]
        return raw_data

    #read points data from mat data
    def load_points(points_path,data):
        n = data.shape[0]
        points_data = np.zeros([n,3003,2])
        start_time = time.time()
        for i in range(n):
            result = loadmat(points_path+data['pointsname'][i])
            key = sorted(result.keys())[-1]
            points = result[key]
            distance_h = []
            distance_v = []
            for d in range(points.shape[0]-1):
                for j in range(d+1,points.shape[0]):
                    distance_h.append(abs(points[d,0]-points[j,0]))
                    distance_v.append(abs(points[d,1]-points[j,1]))

            points_data[i,:,0]=distance_h
            points_data[i,:,1]=distance_v
        print("--- %s seconds ---" % (time.time() - start_time))
```

```
        return points_data.reshape([2500,6006])
```

[5]:
```
path = '/Users/zhaoziqin/Desktop/train_set/'  # Please modify your own path
data = load.load_data(path+'label.csv')
points_path = '/Users/zhaoziqin/Desktop/train_set/points/'
X = load.load_points(points_path,data)
X = np.round(X,0)
y= data['emotion_idx'].to_numpy()
```

```
--- 14.46776294708252 seconds ---
```

1.3 Step 2: Train Test Split

[6]:
```
train_x,test_x,train_y,test_y=train_test_split(X,y,test_size=0.
↪2,random_state=123)
```

[7]:
```
print(train_x.shape)
print(test_x.shape)
print(train_y.shape)
print(test_y.shape)
```

```
(2000, 6006)
(500, 6006)
(2000,)
(500,)
```

1.4 Step 3: Baseline Model — GBM

We used Cross-Validation to find the more efficient estimator combination for baseline model and fitted by using this combination.

[8]:
```
start_time = time.time()
baseline = GradientBoostingClassifier(n_estimators=100,max_depth=␣
↪1,learning_rate=0.1)
gbm_model = baseline.fit(train_x,train_y)
print("--- %s minutes ---" % ((time.time() - start_time)/60))
```

```
--- 8.974443185329438 minutes ---
```

[9]:
```
baseline_train_accuracy = gbm_model.score(train_x,train_y)
baseline_test_accuracy = gbm_model.score(test_x,test_y)
print("Training Accuracy on the baseline model: %.4f" %␣
↪(baseline_train_accuracy))
print("Testing Accuracy on the baseline model: %.4f" % (baseline_test_accuracy))

baseline_pred_train = gbm_model.predict(train_x)
t1 = time.time()
baseline_pred_test = gbm_model.predict(test_x)
```

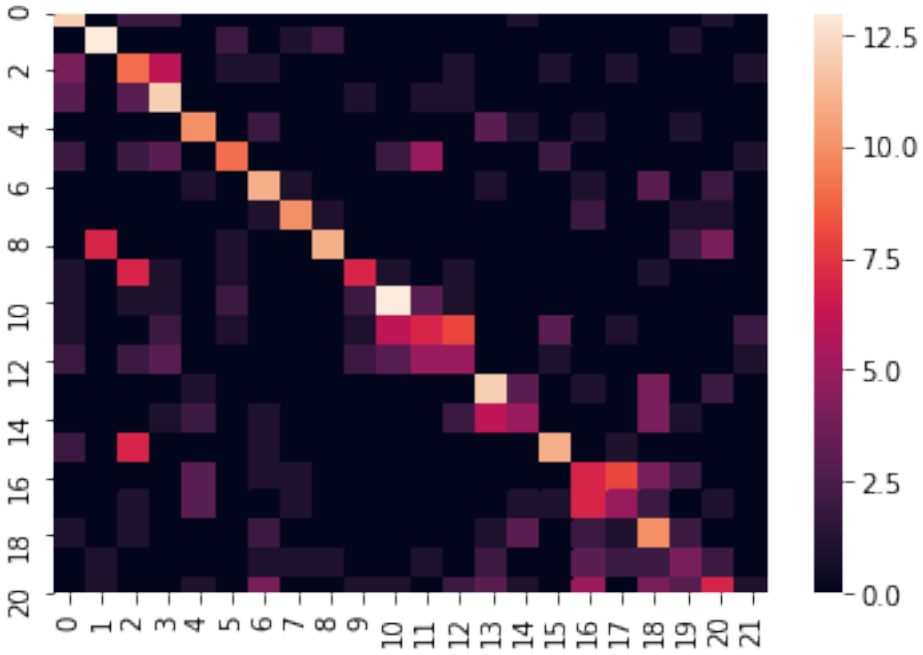
```
print("Prediction on Baseline: %s seconds" % (time.time() - t1))

print(classification_report(train_y, baseline_pred_train))
print(classification_report(test_y, baseline_pred_test))
```

```
Training Accuracy on the baseline model: 0.8380
Testing Accuracy on the baseline model: 0.3820
Prediction on Baseline: 0.029536008834838867 seconds
              precision    recall  f1-score   support

           1       0.87      0.97      0.91        94
           2       0.93      0.99      0.96        95
           3       0.80      0.90      0.85       110
           4       0.80      0.82      0.81        97
           5       0.90      0.95      0.92        91
           6       0.81      0.83      0.82        78
           7       0.88      0.88      0.88        94
           8       0.94      0.95      0.95       104
           9       0.91      0.95      0.93        84
          10       0.79      0.77      0.78        87
          11       0.89      0.82      0.85        99
          12       0.85      0.75      0.79        80
          13       0.81      0.65      0.72        83
          14       0.78      0.90      0.84        99
          15       0.86      0.77      0.81        73
          16       0.82      0.88      0.85        91
          17       0.79      0.83      0.81       101
          18       0.90      0.77      0.83        84
          19       0.70      0.75      0.73        88
          20       0.74      0.74      0.74        95
          21       0.86      0.78      0.82        83
          22       0.85      0.69      0.76        90

    accuracy                           0.84      2000
   macro avg       0.84      0.83      0.83      2000
weighted avg       0.84      0.84      0.84      2000

              precision    recall  f1-score   support

           1       0.41      0.67      0.51        18
           2       0.59      0.68      0.63        19
           3       0.24      0.36      0.29        25
           4       0.38      0.57      0.45        21
           5       0.48      0.56      0.51        18
           6       0.47      0.35      0.40        26
           7       0.42      0.55      0.48        20
           8       0.67      0.62      0.65        16
```

9	0.73	0.44	0.55	25
10	0.44	0.35	0.39	20
11	0.48	0.54	0.51	24
12	0.32	0.22	0.26	32
13	0.24	0.21	0.22	24
14	0.43	0.52	0.47	23
15	0.31	0.23	0.26	22
16	0.55	0.50	0.52	22
17	0.24	0.27	0.25	26
18	0.26	0.23	0.24	22
19	0.28	0.43	0.34	23
20	0.20	0.20	0.20	20
21	0.32	0.21	0.25	34
22	0.14	0.05	0.07	20
accuracy			0.38	500
macro avg	0.39	0.40	0.38	500
weighted avg	0.38	0.38	0.37	500

[10]:
```
cm_baseline = confusion_matrix(test_y,baseline_pred_test)
sns.heatmap(cm_baseline)
```

[10]: <matplotlib.axes._subplots.AxesSubplot at 0x1a25c1db90>



1.5 Step 4 : Candidate Advanced Model — Part I

For our advanced model, we chose different models to fit the train dataset. Here is the following models we picked as our candidate advanced models firstly.

1. XGBoost
2. Logistic Regression
3. Support Vector Machine (SVM)
4. LDA
5. Random Forest

For each of these models, We have different python files for each model which contains the completed process. In each of model, we used Cross-Validation to find the best parameter combination and fit the training set.

1.5.1 1. XGBoost

We used GridSearch cross-validation to find the best parameter combination. 'max_depth': range (1, 5, 1)

'n_estimators': range(1, 200, 20)

'learning_rate': [0.1, 0.01, 0.05]

Results: Best Max Depth: 2 / Best N.estimators: 181 / Best Learning Rate: 0.1

[11]:
```
start_time = time.time()
xgb = XGBClassifier(n_estimators = 181,max_depth=2,learning_rate=0.1)
xgb_model = xgb.fit(train_x,train_y)
print("--- %s minutes ---" % ((time.time() - start_time)/60))
```

```
--- 32.34239470163981 minutes ---
```

[12]:
```
xgb_train_accuracy = xgb_model.score(train_x,train_y)
xgb_test_accuracy = xgb_model.score(test_x,test_y)
print("Training Accuracy on the xgb model: %.4f" % (xgb_train_accuracy))
print("Testing Accuracy on the xgb model: %.4f" % (xgb_test_accuracy))

xgb_pred_train = xgb_model.predict(train_x)
t1 = time.time()
xgb_pred_test = xgb_model.predict(test_x)
print("Prediction on XGBoost: %s seconds" % (time.time() - t1))

print(classification_report(train_y, xgb_pred_train))
print(classification_report(test_y, xgb_pred_test))
```

```
Training Accuracy on the xgb model: 1.0000
Testing Accuracy on the xgb model: 0.4920
Prediction on XGBoost: 1.2528421878814697 seconds
              precision    recall  f1-score   support
```

1	1.00	1.00	1.00	94
2	1.00	1.00	1.00	95
3	1.00	1.00	1.00	110
4	1.00	1.00	1.00	97
5	1.00	1.00	1.00	91
6	1.00	1.00	1.00	78
7	1.00	1.00	1.00	94
8	1.00	1.00	1.00	104
9	1.00	1.00	1.00	84
10	1.00	1.00	1.00	87
11	1.00	1.00	1.00	99
12	1.00	1.00	1.00	80
13	1.00	1.00	1.00	83
14	1.00	1.00	1.00	99
15	1.00	1.00	1.00	73
16	1.00	1.00	1.00	91
17	1.00	1.00	1.00	101
18	1.00	1.00	1.00	84
19	1.00	1.00	1.00	88
20	1.00	1.00	1.00	95
21	1.00	1.00	1.00	83
22	1.00	1.00	1.00	90
accuracy			1.00	2000
macro avg	1.00	1.00	1.00	2000
weighted avg	1.00	1.00	1.00	2000

	precision	recall	f1-score	support
1	0.58	0.83	0.68	18
2	0.63	0.63	0.63	19
3	0.46	0.52	0.49	25
4	0.44	0.52	0.48	21
5	0.50	0.61	0.55	18
6	0.73	0.42	0.54	26
7	0.47	0.40	0.43	20
8	0.75	0.75	0.75	16
9	0.68	0.52	0.59	25
10	0.55	0.60	0.57	20
11	0.52	0.58	0.55	24
12	0.43	0.41	0.42	32
13	0.29	0.29	0.29	24
14	0.60	0.65	0.63	23
15	0.56	0.41	0.47	22
16	0.65	0.77	0.71	22
17	0.42	0.50	0.46	26
18	0.50	0.32	0.39	22
19	0.36	0.57	0.44	23

```
           20       0.27      0.35      0.30        20
           21       0.46      0.32      0.38        34
           22       0.17      0.10      0.12        20

     accuracy                           0.49       500
    macro avg       0.50      0.50      0.49       500
 weighted avg       0.50      0.49      0.49       500
```

1.5.2 2. Logistic Regression

We used GridSearchCV to fine the best parameter combination. 'C': [0.001,0.01, 1, 25,50,100]

Result:

'C': 0.01, 'dual': False. 'fit_intercept': True, 'intercept_scaling': 1, 'max_iter': 300, 'multi_class': 'multinomial', 'penalty': 'l2', 'solver': 'lbfgs', 'tol': 0.0001}

[13]:
```
start_time = time.time()
lr = LogisticRegression(C=0.01, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=300,
                   multi_class='multinomial', penalty='l2',
                   solver='lbfgs', tol=0.0001)
lr_model = lr.fit(train_x,train_y)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 16.357911109924316 seconds ---
```

[14]:
```
lr_train_accuracy = lr_model.score(train_x,train_y)
lr_test_accuracy = lr_model.score(test_x,test_y)
print("Training Accuracy on the logistic regression model: %.4f" %␣
 ↪(lr_train_accuracy))
print("Testing Accuracy on the logistic regression model: %.4f" %␣
 ↪(lr_test_accuracy))

lr_pred_train = lr_model.predict(train_x)
t1 = time.time()
lr_pred_test = lr_model.predict(test_x)
print("Prediction on Logistic Regression: %s seconds" % (time.time() - t1))

print(classification_report(train_y, lr_pred_train))
print(classification_report(test_y, lr_pred_test))
```

```
Training Accuracy on the logistic regression model: 0.7725
Testing Accuracy on the logistic regression model: 0.5140
Prediction on Logistic Regression: 0.013192892074584961 seconds
              precision    recall  f1-score   support
```

1	0.84	0.89	0.87	94
2	0.93	0.94	0.93	95
3	0.83	0.85	0.84	110
4	0.76	0.76	0.76	97
5	0.85	0.89	0.87	91
6	0.77	0.77	0.77	78
7	0.80	0.79	0.80	94
8	0.93	0.95	0.94	104
9	0.92	0.93	0.92	84
10	0.67	0.64	0.66	87
11	0.81	0.81	0.81	99
12	0.68	0.66	0.67	80
13	0.64	0.64	0.64	83
14	0.77	0.83	0.80	99
15	0.71	0.67	0.69	73
16	0.80	0.80	0.80	91
17	0.80	0.77	0.79	101
18	0.70	0.71	0.71	84
19	0.62	0.66	0.64	88
20	0.76	0.64	0.70	95
21	0.67	0.64	0.65	83
22	0.61	0.63	0.62	90
accuracy			0.77	2000
macro avg	0.77	0.77	0.77	2000
weighted avg	0.77	0.77	0.77	2000

	precision	recall	f1-score	support
1	0.58	0.78	0.67	18
2	0.85	0.58	0.69	19
3	0.55	0.64	0.59	25
4	0.42	0.52	0.47	21
5	0.62	0.89	0.73	18
6	0.48	0.46	0.47	26
7	0.52	0.60	0.56	20
8	0.59	0.81	0.68	16
9	0.90	0.72	0.80	25
10	0.45	0.45	0.45	20
11	0.52	0.50	0.51	24
12	0.44	0.34	0.39	32
13	0.43	0.38	0.40	24
14	0.56	0.65	0.60	23
15	0.55	0.55	0.55	22
16	0.57	0.73	0.64	22
17	0.54	0.54	0.54	26
18	0.64	0.41	0.50	22
19	0.19	0.17	0.18	23

```
           20       0.32      0.35      0.33        20
           21       0.52      0.35      0.42        34
           22       0.20      0.20      0.20        20

     accuracy                           0.51       500
    macro avg       0.52      0.53      0.52       500
 weighted avg       0.52      0.51      0.51       500
```

1.5.3 3. Support Vector Machine (SVM)

We used GridSearchCV to find the best parameter combination of SVM.

param= {'C': [0.0000001,0.000001,0.00001,0.0001,0.001,0.01,1]}

Result: 'C':0.00001

[15]:
```
start_time = time.time()
svc = SVC(kernel= 'linear', random_state = 123, C = 0.00001)
svm = svc.fit(train_x,train_y)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 18.52547812461853 seconds ---
```

[16]:
```
svm_train_accuracy = svm.score(train_x,train_y)
svm_test_accuracy = svm.score(test_x,test_y)
print("Training Accuracy on the logistic regression model: %.4f" %␣
 ↪(svm_train_accuracy))
print("Testing Accuracy on the logistic regression model: %.4f" %␣
 ↪(svm_test_accuracy))

svm_pred_train = svm.predict(train_x)
t1 = time.time()
svm_pred_test = svm.predict(test_x)
print("Prediction on SVM: %s seconds" % (time.time() - t1))

print(classification_report(train_y, svm_pred_train))
print(classification_report(test_y, svm_pred_test))
```

```
Training Accuracy on the logistic regression model: 0.8145
Testing Accuracy on the logistic regression model: 0.4840
Prediction on SVM: 5.348582744598389 seconds
              precision    recall  f1-score   support

           1       0.88      0.97      0.92        94
           2       0.96      0.96      0.96        95
           3       0.87      0.82      0.85       110
           4       0.75      0.81      0.78        97
           5       0.90      0.95      0.92        91
```

6	0.79	0.78	0.79	78
7	0.87	0.87	0.87	94
8	0.94	0.98	0.96	104
9	0.93	0.95	0.94	84
10	0.70	0.79	0.74	87
11	0.86	0.76	0.81	99
12	0.66	0.71	0.69	80
13	0.65	0.61	0.63	83
14	0.75	0.89	0.81	99
15	0.85	0.71	0.78	73
16	0.89	0.89	0.89	91
17	0.81	0.78	0.79	101
18	0.82	0.74	0.78	84
19	0.69	0.70	0.70	88
20	0.82	0.68	0.75	95
21	0.72	0.78	0.75	83
22	0.78	0.68	0.73	90
accuracy			0.81	2000
macro avg	0.81	0.81	0.81	2000
weighted avg	0.82	0.81	0.81	2000

	precision	recall	f1-score	support
1	0.59	0.72	0.65	18
2	0.68	0.68	0.68	19
3	0.40	0.56	0.47	25
4	0.41	0.57	0.48	21
5	0.54	0.83	0.65	18
6	0.50	0.46	0.48	26
7	0.45	0.45	0.45	20
8	0.59	0.62	0.61	16
9	0.78	0.72	0.75	25
10	0.38	0.40	0.39	20
11	0.55	0.46	0.50	24
12	0.54	0.44	0.48	32
13	0.36	0.33	0.35	24
14	0.62	0.70	0.65	23
15	0.65	0.59	0.62	22
16	0.48	0.59	0.53	22
17	0.50	0.46	0.48	26
18	0.37	0.32	0.34	22
19	0.17	0.17	0.17	23
20	0.33	0.30	0.32	20
21	0.52	0.35	0.42	34
22	0.14	0.10	0.12	20
accuracy			0.48	500

```
   macro avg       0.48      0.49      0.48       500
weighted avg       0.48      0.48      0.48       500
```

1.5.4 4. Linear Discriminant Analysis (LDA)

While LDA is often times used for dimensionality reduction, it can also be a powerful classifying model which performed quite well with the image data.

For the LDA we used the following key parameters:

n_components: 2 This determines the amount of features that will be created by LDA.

shrinkage:.1 This determines the steps taken by the model to improve covariance matrix estimation.

[17]:
```
start_time = time.time()
lda = LDA(solver='eigen', shrinkage=.1, n_components=2)
lda_model = lda.fit(train_x, train_y)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 58.26339888572693 seconds ---
```

[18]:
```
lda_train_accuracy = lda_model.score(train_x,train_y)
lda_test_accuracy = lda_model.score(test_x,test_y)
print("Training Accuracy on the LDA model: %.4f" % (lda_train_accuracy))
print("Testing Accuracy on the LDA model: %.4f" % (lda_test_accuracy))

lda_pred_train = lda_model.predict(train_x)
t1 = time.time()
lda_pred_test = lda_model.predict(test_x)
print("Prediction on LDA: %s minutes" % (time.time() - t1))

print(classification_report(train_y, lda_pred_train))
print(classification_report(test_y, lda_pred_test))
```

```
Training Accuracy on the LDA model: 0.8580
Testing Accuracy on the LDA model: 0.5380
Prediction on LDA: 0.010159015655517578 minutes
              precision    recall  f1-score   support

           1       0.91      0.99      0.95        94
           2       0.93      0.96      0.94        95
           3       0.87      0.89      0.88       110
           4       0.86      0.84      0.85        97
           5       0.96      0.90      0.93        91
           6       0.84      0.82      0.83        78
           7       0.90      0.87      0.89        94
           8       0.95      0.92      0.94       104
           9       0.94      0.96      0.95        84
```

10	0.75	0.86	0.80	87
11	0.96	0.70	0.81	99
12	0.79	0.82	0.80	80
13	0.75	0.81	0.78	83
14	0.84	0.86	0.85	99
15	0.88	0.77	0.82	73
16	0.95	0.84	0.89	91
17	0.89	0.81	0.85	101
18	0.81	0.88	0.85	84
19	0.74	0.89	0.81	88
20	0.86	0.77	0.81	95
21	0.78	0.88	0.83	83
22	0.76	0.82	0.79	90
accuracy			0.86	2000
macro avg	0.86	0.86	0.86	2000
weighted avg	0.86	0.86	0.86	2000

	precision	recall	f1-score	support
1	0.61	0.94	0.74	18
2	0.79	0.58	0.67	19
3	0.58	0.60	0.59	25
4	0.50	0.57	0.53	21
5	0.85	0.61	0.71	18
6	0.64	0.54	0.58	26
7	0.60	0.60	0.60	20
8	0.68	0.81	0.74	16
9	0.79	0.88	0.83	25
10	0.60	0.60	0.60	20
11	0.77	0.42	0.54	24
12	0.43	0.50	0.46	32
13	0.37	0.46	0.41	24
14	0.54	0.57	0.55	23
15	0.52	0.50	0.51	22
16	0.73	0.73	0.73	22
17	0.55	0.46	0.50	26
18	0.50	0.45	0.48	22
19	0.25	0.35	0.29	23
20	0.33	0.40	0.36	20
21	0.52	0.35	0.42	34
22	0.17	0.15	0.16	20
accuracy			0.54	500
macro avg	0.56	0.55	0.55	500
weighted avg	0.55	0.54	0.54	500

1.5.5 5. Random Forest

[19]:
```
start_time = time.time()
rf = RandomForestClassifier(n_estimators = 100, criterion = 'gini',␣
↪min_samples_leaf=1, max_features='sqrt')
rf_model = rf.fit(train_x, train_y)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 7.26177978515625 seconds ---
```

[20]:
```
rf_train_accuracy = rf_model.score(train_x,train_y)
rf_test_accuracy = rf_model.score(test_x,test_y)
print("Training Accuracy on the random forest model: %.4f" %␣
↪(rf_train_accuracy))
print("Testing Accuracy on the random forest model: %.4f" % (rf_test_accuracy))

rf_pred_train = rf_model.predict(train_x)
t1 = time.time()
rf_pred_test = rf_model.predict(test_x)
print("Prediction on Random Forest: %s seconds" % (time.time() - t1))

print(classification_report(train_y, rf_pred_train))
print(classification_report(test_y, rf_pred_test))
```

```
Training Accuracy on the random forest model: 1.0000
Testing Accuracy on the random forest model: 0.4280
Prediction on Random Forest: 0.04013514518737793 seconds
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95
           3       1.00      1.00      1.00       110
           4       1.00      1.00      1.00        97
           5       1.00      1.00      1.00        91
           6       1.00      1.00      1.00        78
           7       1.00      1.00      1.00        94
           8       1.00      1.00      1.00       104
           9       1.00      1.00      1.00        84
          10       1.00      1.00      1.00        87
          11       1.00      1.00      1.00        99
          12       1.00      1.00      1.00        80
          13       1.00      1.00      1.00        83
          14       1.00      1.00      1.00        99
          15       1.00      1.00      1.00        73
          16       1.00      1.00      1.00        91
          17       1.00      1.00      1.00       101
          18       1.00      1.00      1.00        84
          19       1.00      1.00      1.00        88
```

20	1.00	1.00	1.00	95
21	1.00	1.00	1.00	83
22	1.00	1.00	1.00	90
accuracy			1.00	2000
macro avg	1.00	1.00	1.00	2000
weighted avg	1.00	1.00	1.00	2000

	precision	recall	f1-score	support
1	0.35	0.39	0.37	18
2	0.48	0.68	0.57	19
3	0.30	0.52	0.38	25
4	0.39	0.62	0.48	21
5	0.55	0.67	0.60	18
6	0.53	0.38	0.44	26
7	0.39	0.35	0.37	20
8	0.61	0.88	0.72	16
9	0.62	0.52	0.57	25
10	0.48	0.50	0.49	20
11	0.34	0.42	0.38	24
12	0.41	0.28	0.33	32
13	0.07	0.04	0.05	24
14	0.46	0.83	0.59	23
15	0.64	0.41	0.50	22
16	0.70	0.64	0.67	22
17	0.36	0.46	0.41	26
18	0.27	0.14	0.18	22
19	0.26	0.26	0.26	23
20	0.35	0.30	0.32	20
21	0.60	0.35	0.44	34
22	0.14	0.05	0.07	20
accuracy			0.43	500
macro avg	0.42	0.44	0.42	500
weighted avg	0.42	0.43	0.41	500

1.6 Step 5:

1.6.1 Ensemle with VotingClassifier : Combining the effective models to get the better prediction

From our candidate advanced models 1-5, summary is in the following :

[4]:
```
chart = {'Candidate Model':['Baseline(GBM)', 'XGBoost', 'Logistic Regression',␣
↪'LDA','SVM','Random Forest'],
        'Training Accuracy(%)':['83.8','100','77.25','85.8','81.45','100'],
```

```
        'Testing Accuracy(%)':['38.2','49.2','51.4','53.8','48.4','43'],
         'Fitting Time':['521s','1987s','15s','68.77s','18.18s','7.9s'],
         'Prediction Time':['0.036s','1.18s','0.0084s','0.0077s','5.26s','0.
↪04s']
        }

df = pd.DataFrame(chart,index = ['1','2','3','4','5','6'])
df
```

[4]:

	Candidate Model	Training Accuracy(%)	Testing Accuracy(%)	Fitting Time	Prediction Time
1	Baseline(GBM)	83.8	38.2	521s	0.036s
2	XGBoost	100	49.2	1987s	1.18s
3	Logistic Regression	77.25	51.4	15s	0.0084s
4	LDA	85.8	53.8	68.77s	0.0077s
5	SVM	81.45	48.4	18.18s	5.26s
6	Random Forest	100	43	7.9s	0.04s

Now, we want to combine the effiective models showing above and to do a ensemble prediction.

From all the candidate models, XGBoost, Logistic Regression, LDA and SVM have the higher testing accuracy. At the same time, XGBoost has the longer fitting time, so we did not pick XGBoost in our ensemble model.

1.6.2 Ensemble : Logistic Regression, LDA and SVM

[22]:
```
start_time = time.time()
eclf = VotingClassifier(
    estimators=[('lr', lr), ('lda', lda), ('svm', svc)],
    voting='hard',
    weights=[1.5,1.5,1])
eclf_model = eclf.fit(train_x, train_y)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 89.48586392402649 seconds ---
```

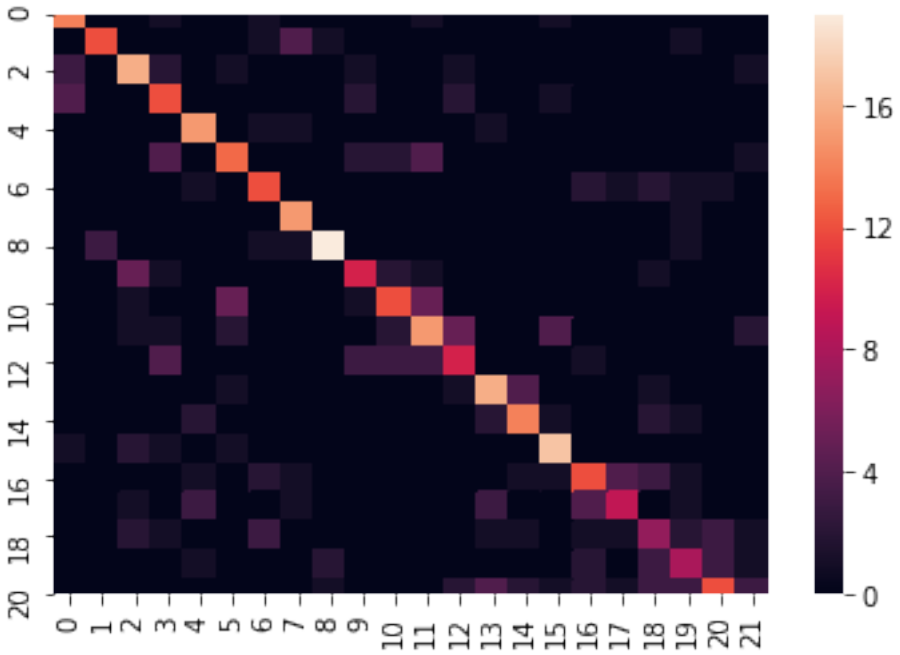
[23]:
```
eclf_train_accuracy = eclf_model.score(train_x,train_y)
eclf_test_accuracy = eclf_model.score(test_x,test_y)
print("Training Accuracy on the Ensemble model: %.4f" % (eclf_train_accuracy))
print("Testing Accuracy on the Ensemble model: %.4f" % (eclf_test_accuracy))
```

```
eclf_pred_train = eclf_model.predict(train_x)
t1 = time.time()
eclf_pred_test = eclf_model.predict(test_x)
print("Prediction on Ensemble model: %s seconds" % (time.time() - t1))
```

```
Training Accuracy on the Ensemble model: 0.8405
Testing Accuracy on the Ensemble model: 0.5460
Prediction on Ensemble model: 5.4597392082214355 seconds
```

[24]:
```
cm_eclf = confusion_matrix(test_y,eclf_pred_test)
sns.heatmap(cm_eclf)
```

[24]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a41708c10>
```



1.7 Step 6 : Candidate Advanced Model — Deep Learning

1.7.1 Neural Network

At the beginning of this model, install the required packages.

If you do not have keras and tensorflow in your python environment. Please install first.

Otherwise, just ignore : 1. conda install keras 2. conda install tensorflow

[25]:
```
conda install keras
```

```
Collecting package metadata (current_repodata.json): done
```

```
Solving environment: done

# All requested packages already installed.


Note: you may need to restart the kernel to use updated packages.
```

[26]:
```
conda install tensorflow
```

```
Collecting package metadata (current_repodata.json): done
Solving environment: done

# All requested packages already installed.


Note: you may need to restart the kernel to use updated packages.
```

[27]:
```
import tensorflow
import keras
from keras.utils import to_categorical
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from keras.layers import Dense, Activation, Flatten, Input, Dropout
from keras.layers import BatchNormalization
from keras.models import Model
from keras import initializers
from keras.optimizers import Adam
from keras.utils import to_categorical
```

```
Using TensorFlow backend.
```

Model Fitting

[28]:
```
Y = to_categorical(y)
Y = Y[:,1:]
X_train,X_test,y_train,y_test = train_test_split(X,Y,test_size=0.
 ↪2,random_state=1106)
```

[29]:
```
input_shape = [6006]
input_layer = Input(input_shape)
x = BatchNormalization(momentum = 0.88)(input_layer)
x = Dense(220,activation='relu',kernel_initializer=initializers.
 ↪glorot_normal(seed=6))(x)
x = Dropout(0.25)(x)
x = BatchNormalization()(x)
x = Dense(176,activation='relu',kernel_initializer=initializers.
 ↪glorot_normal(seed=6))(x)
x = Dropout(0.25)(x)
```

```
x = Dense(88,activation='relu',kernel_initializer=initializers.
 ↪glorot_normal(seed=6))(x)
x = Dropout(0.25)(x)
x = Dense(44,activation='relu',kernel_initializer=initializers.
 ↪glorot_normal(seed=6))(x)
output_layer = Dense(22,activation='softmax',kernel_initializer=initializers.
 ↪glorot_normal(seed=6))(x)
model = Model(input_layer,output_layer)
```

[30]:
```
start_time = time.time()
model.compile(loss='categorical_crossentropy',optimizer = Adam(lr=0.
 ↪001),metrics=['accuracy'])
model_nn = model.fit(X_train,y_train,epochs=200)
print("--- %s minutes ---" % ((time.time() - start_time)/60))
```

```
Epoch 1/200
2000/2000 [==============================] - 7s 4ms/step - loss: 3.0549 -
accuracy: 0.0795
Epoch 2/200
2000/2000 [==============================] - 4s 2ms/step - loss: 2.6785 -
accuracy: 0.1685
Epoch 3/200
2000/2000 [==============================] - 4s 2ms/step - loss: 2.3393 -
accuracy: 0.2415
Epoch 4/200
2000/2000 [==============================] - 4s 2ms/step - loss: 2.1456 -
accuracy: 0.2950
Epoch 5/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.9831 -
accuracy: 0.3410
Epoch 6/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.9012 -
accuracy: 0.3645
Epoch 7/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.7745 -
accuracy: 0.4000
Epoch 8/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.7166 -
accuracy: 0.4050
Epoch 9/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.6490 -
accuracy: 0.4230
Epoch 10/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.5701 -
accuracy: 0.4690
Epoch 11/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.5239 -
```

```
accuracy: 0.4835
Epoch 12/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.4611 -
accuracy: 0.4935
Epoch 13/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.4383 -
accuracy: 0.5115
Epoch 14/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.4093 -
accuracy: 0.5240
Epoch 15/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.3798 -
accuracy: 0.5160
Epoch 16/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.3092 -
accuracy: 0.5480
Epoch 17/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.2975 -
accuracy: 0.5495
Epoch 18/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.3439 -
accuracy: 0.5350
Epoch 19/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.2519 -
accuracy: 0.5805
Epoch 20/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.2313 -
accuracy: 0.5715
Epoch 21/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.2178 -
accuracy: 0.5885
Epoch 22/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.1734 -
accuracy: 0.5850
Epoch 23/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.1797 -
accuracy: 0.5910
Epoch 24/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.1367 -
accuracy: 0.6030
Epoch 25/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.1402 -
accuracy: 0.6010
Epoch 26/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.0854 -
accuracy: 0.6190
Epoch 27/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.0431 -
```

```
accuracy: 0.6410
Epoch 28/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.0588 -
accuracy: 0.6255
Epoch 29/200
2000/2000 [==============================] - 4s 2ms/step - loss: 1.0688 -
accuracy: 0.6285
Epoch 30/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.9888 -
accuracy: 0.6560
Epoch 31/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.9808 -
accuracy: 0.6675
Epoch 32/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.9526 -
accuracy: 0.6675
Epoch 33/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.9252 -
accuracy: 0.6875
Epoch 34/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.9527 -
accuracy: 0.6570
Epoch 35/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.9191 -
accuracy: 0.6810
Epoch 36/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.9312 -
accuracy: 0.6750
Epoch 37/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.9023 -
accuracy: 0.6900
Epoch 38/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.9059 -
accuracy: 0.6885
Epoch 39/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.8809 -
accuracy: 0.6885
Epoch 40/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.8648 -
accuracy: 0.7005
Epoch 41/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.8133 -
accuracy: 0.7210
Epoch 42/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.8088 -
accuracy: 0.7205
Epoch 43/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.8133 -
```

```
accuracy: 0.7200
Epoch 44/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.8278 -
accuracy: 0.7145
Epoch 45/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.8193 -
accuracy: 0.7140
Epoch 46/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.7431 -
accuracy: 0.7370
Epoch 47/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.7627 -
accuracy: 0.7415
Epoch 48/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.7410 -
accuracy: 0.7460
Epoch 49/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.7910 -
accuracy: 0.7215
Epoch 50/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.7170 -
accuracy: 0.7585
Epoch 51/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.7191 -
accuracy: 0.7460
Epoch 52/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.6650 -
accuracy: 0.7660
Epoch 53/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.6986 -
accuracy: 0.7485
Epoch 54/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.6751 -
accuracy: 0.7580
Epoch 55/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.6978 -
accuracy: 0.7555
Epoch 56/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.6981 -
accuracy: 0.7675
Epoch 57/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.7151 -
accuracy: 0.7560
Epoch 58/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.6801 -
accuracy: 0.7755
Epoch 59/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.6096 -
```

```
accuracy: 0.7840
Epoch 60/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5897 -
accuracy: 0.7920
Epoch 61/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5886 -
accuracy: 0.7920
Epoch 62/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.6059 -
accuracy: 0.7930
Epoch 63/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5914 -
accuracy: 0.7885
Epoch 64/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5639 -
accuracy: 0.8020
Epoch 65/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5875 -
accuracy: 0.8055
Epoch 66/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5728 -
accuracy: 0.7985
Epoch 67/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5770 -
accuracy: 0.8035
Epoch 68/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.5228 -
accuracy: 0.8275
Epoch 69/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5309 -
accuracy: 0.8185
Epoch 70/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5442 -
accuracy: 0.8195
Epoch 71/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5320 -
accuracy: 0.8255
Epoch 72/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5318 -
accuracy: 0.8260
Epoch 73/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5395 -
accuracy: 0.8080
Epoch 74/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5308 -
accuracy: 0.8250
Epoch 75/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.5134 -
```

```
accuracy: 0.8295
Epoch 76/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4583 -
accuracy: 0.8410
Epoch 77/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4463 -
accuracy: 0.8455
Epoch 78/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4737 -
accuracy: 0.8375
Epoch 79/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4599 -
accuracy: 0.8400
Epoch 80/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4598 -
accuracy: 0.8405
Epoch 81/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4454 -
accuracy: 0.8505
Epoch 82/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4623 -
accuracy: 0.8400
Epoch 83/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4191 -
accuracy: 0.8540
Epoch 84/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4961 -
accuracy: 0.8375
Epoch 85/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4453 -
accuracy: 0.8425
Epoch 86/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4906 -
accuracy: 0.8360
Epoch 87/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3959 -
accuracy: 0.8660
Epoch 88/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4320 -
accuracy: 0.8500
Epoch 89/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4257 -
accuracy: 0.8590
Epoch 90/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4352 -
accuracy: 0.8585
Epoch 91/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.4078 -
```

```
accuracy: 0.8625
Epoch 92/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3814 -
accuracy: 0.8690
Epoch 93/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3619 -
accuracy: 0.8785
Epoch 94/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3578 -
accuracy: 0.8720
Epoch 95/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3568 -
accuracy: 0.8760
Epoch 96/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3548 -
accuracy: 0.8815
Epoch 97/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3635 -
accuracy: 0.8795
Epoch 98/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3866 -
accuracy: 0.8725
Epoch 99/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.3878 -
accuracy: 0.8740
Epoch 100/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.3710 -
accuracy: 0.8795
Epoch 101/200
2000/2000 [==============================] - 6s 3ms/step - loss: 0.3740 -
accuracy: 0.8675
Epoch 102/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.3340 -
accuracy: 0.8820
Epoch 103/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.3404 -
accuracy: 0.8875
Epoch 104/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3454 -
accuracy: 0.8865
Epoch 105/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3589 -
accuracy: 0.8865
Epoch 106/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3219 -
accuracy: 0.8875
Epoch 107/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3399 -
```

```
accuracy: 0.8840
Epoch 108/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3271 -
accuracy: 0.8930
Epoch 109/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3596 -
accuracy: 0.8895
Epoch 110/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3444 -
accuracy: 0.8810
Epoch 111/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2999 -
accuracy: 0.8945
Epoch 112/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3301 -
accuracy: 0.8880
Epoch 113/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3296 -
accuracy: 0.8915
Epoch 114/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3665 -
accuracy: 0.8790
Epoch 115/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3201 -
accuracy: 0.9040
Epoch 116/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3058 -
accuracy: 0.9015
Epoch 117/200
2000/2000 [==============================] - 3s 2ms/step - loss: 0.2567 -
accuracy: 0.9140
Epoch 118/200
2000/2000 [==============================] - 3s 2ms/step - loss: 0.3238 -
accuracy: 0.8945
Epoch 119/200
2000/2000 [==============================] - 3s 2ms/step - loss: 0.3186 -
accuracy: 0.8965
Epoch 120/200
2000/2000 [==============================] - 3s 2ms/step - loss: 0.3192 -
accuracy: 0.8975
Epoch 121/200
2000/2000 [==============================] - 35s 18ms/step - loss: 0.2937 -
accuracy: 0.9055
Epoch 122/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.2886 -
accuracy: 0.8990
Epoch 123/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3057 -
```

```
accuracy: 0.8975
Epoch 124/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2921 -
accuracy: 0.9030
Epoch 125/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3099 -
accuracy: 0.9030
Epoch 126/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2654 -
accuracy: 0.9120
Epoch 127/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2416 -
accuracy: 0.9160
Epoch 128/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2322 -
accuracy: 0.9250
Epoch 129/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3033 -
accuracy: 0.9065
Epoch 130/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2561 -
accuracy: 0.9135
Epoch 131/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2250 -
accuracy: 0.9275
Epoch 132/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2741 -
accuracy: 0.9130
Epoch 133/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2569 -
accuracy: 0.9135
Epoch 134/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2388 -
accuracy: 0.9190
Epoch 135/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2827 -
accuracy: 0.9145
Epoch 136/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2642 -
accuracy: 0.9210
Epoch 137/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2749 -
accuracy: 0.9110
Epoch 138/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2286 -
accuracy: 0.9240
Epoch 139/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2243 -
```

```
accuracy: 0.9315
Epoch 140/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2198 -
accuracy: 0.9255
Epoch 141/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2459 -
accuracy: 0.9180
Epoch 142/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2224 -
accuracy: 0.9285
Epoch 143/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2555 -
accuracy: 0.9170
Epoch 144/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.3203 -
accuracy: 0.9030
Epoch 145/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2356 -
accuracy: 0.9275
Epoch 146/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2202 -
accuracy: 0.9320
Epoch 147/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2310 -
accuracy: 0.9245
Epoch 148/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2150 -
accuracy: 0.9315
Epoch 149/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2701 -
accuracy: 0.9115
Epoch 150/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2223 -
accuracy: 0.9280
Epoch 151/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2242 -
accuracy: 0.9325
Epoch 152/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2386 -
accuracy: 0.9310
Epoch 153/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.1917 -
accuracy: 0.9420
Epoch 154/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2142 -
accuracy: 0.9335
Epoch 155/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2591 -
```

```
accuracy: 0.9220
Epoch 156/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2060 -
accuracy: 0.9320
Epoch 157/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2071 -
accuracy: 0.9305
Epoch 158/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2081 -
accuracy: 0.9315
Epoch 159/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2191 -
accuracy: 0.9300
Epoch 160/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2086 -
accuracy: 0.9330
Epoch 161/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2742 -
accuracy: 0.9105
Epoch 162/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2201 -
accuracy: 0.9315
Epoch 163/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.2440 -
accuracy: 0.9245
Epoch 164/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.1993 -
accuracy: 0.9365
Epoch 165/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.1936 -
accuracy: 0.9395
Epoch 166/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.1990 -
accuracy: 0.9300
Epoch 167/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.2060 -
accuracy: 0.9320
Epoch 168/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.2186 -
accuracy: 0.9350
Epoch 169/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.2081 -
accuracy: 0.9355
Epoch 170/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.1765 -
accuracy: 0.9400
Epoch 171/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.2097 -
```

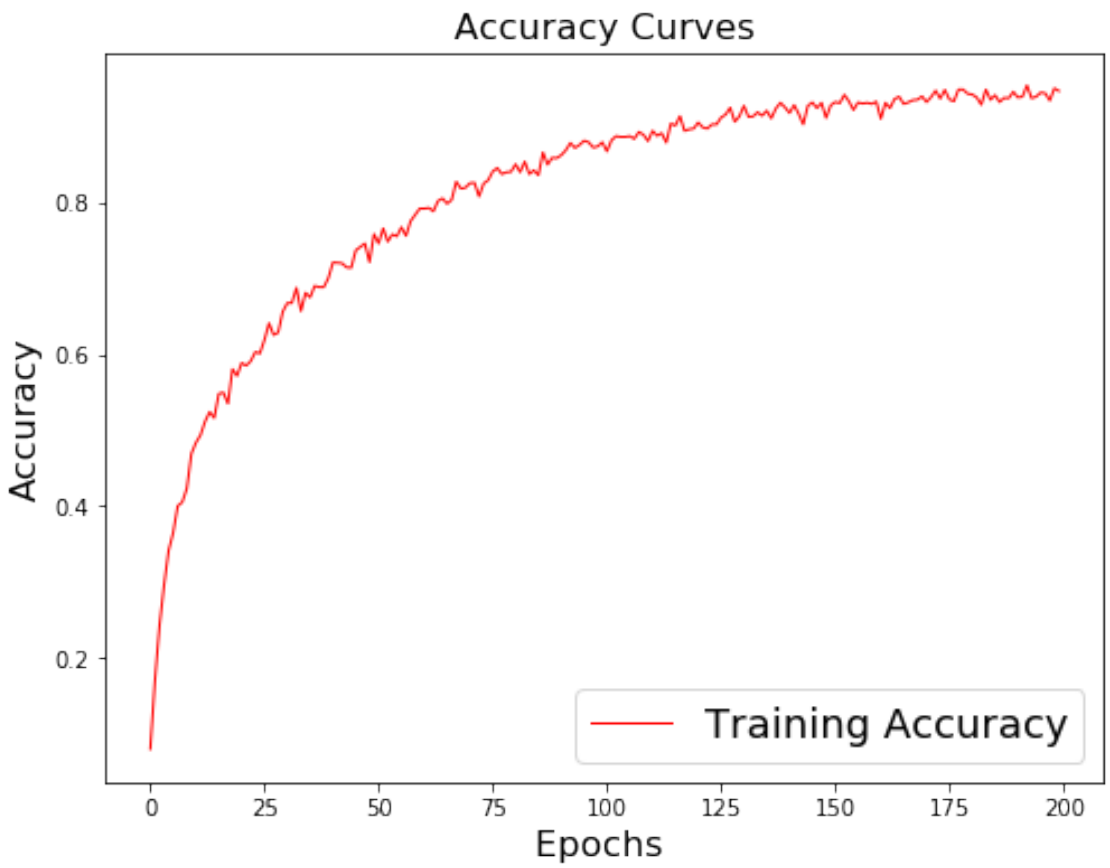
```
accuracy: 0.9325
Epoch 172/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1662 -
accuracy: 0.9390
Epoch 173/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1618 -
accuracy: 0.9475
Epoch 174/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1961 -
accuracy: 0.9370
Epoch 175/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1676 -
accuracy: 0.9485
Epoch 176/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1867 -
accuracy: 0.9360
Epoch 177/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.2161 -
accuracy: 0.9335
Epoch 178/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1657 -
accuracy: 0.9490
Epoch 179/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1592 -
accuracy: 0.9490
Epoch 180/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1797 -
accuracy: 0.9435
Epoch 181/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1687 -
accuracy: 0.9425
Epoch 182/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.2000 -
accuracy: 0.9385
Epoch 183/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.2072 -
accuracy: 0.9290
Epoch 184/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1685 -
accuracy: 0.9490
Epoch 185/200
2000/2000 [==============================] - 4s 2ms/step - loss: 0.1936 -
accuracy: 0.9355
Epoch 186/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1896 -
accuracy: 0.9415
Epoch 187/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.2286 -
```

```
accuracy: 0.9325
Epoch 188/200
2000/2000 [==============================] - 527s 263ms/step - loss: 0.2037 -
accuracy: 0.9380
Epoch 189/200
2000/2000 [==============================] - 6s 3ms/step - loss: 0.2061 -
accuracy: 0.9375
Epoch 190/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.1826 -
accuracy: 0.9460
Epoch 191/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1838 -
accuracy: 0.9385
Epoch 192/200
2000/2000 [==============================] - 6s 3ms/step - loss: 0.1965 -
accuracy: 0.9395
Epoch 193/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.1661 -
accuracy: 0.9545
Epoch 194/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.1951 -
accuracy: 0.9375
Epoch 195/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.1753 -
accuracy: 0.9395
Epoch 196/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.1797 -
accuracy: 0.9450
Epoch 197/200
2000/2000 [==============================] - 5s 2ms/step - loss: 0.1742 -
accuracy: 0.9440
Epoch 198/200
2000/2000 [==============================] - 5s 3ms/step - loss: 0.2026 -
accuracy: 0.9345
Epoch 199/200
2000/2000 [==============================] - 6s 3ms/step - loss: 0.1491 -
accuracy: 0.9505
Epoch 200/200
2000/2000 [==============================] - 6s 3ms/step - loss: 0.1481 -
accuracy: 0.9470
--- 23.48711793422699 minutes ---
```

[31]:
```
plt.figure(figsize=[8,6])
plt.plot(model_nn.history['accuracy'],'r',linewidth=1.0)
plt.legend(['Training Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
```

```
plt.title('Accuracy Curves',fontsize=16)
```

[31]:
```
Text(0.5, 1.0, 'Accuracy Curves')
```



[32]:
```
start_time = time.time()
pred = model.predict(X_test)
pred_list = []
for i in range(len(pred)):
    arr = pred[i]
    idx = np.argwhere(arr == np.max(arr))
    pred_list.append(idx[0][0])
tst_labl = np.argmax(y_test, axis=-1)
accuracy = accuracy_score(pred_list, tst_labl)
print("Test accuracy is %s percent" % round(accuracy*100,3))
print("testing model takes %s seconds" % (time.time() - start_time))
```

```
Test accuracy is 54.2 percent
testing model takes 0.9099469184875488 seconds
```

1.8 Step 7: Comparing : Ensemble Model VS Neural Network

[33]:
```
compare = {'Candidate Model':['Ensemble Model','Neutral Network'],
        'Testing Accuracy(%)':['54.6','55'],
         'Fitting Time':['106s','994s'],
         'Prediction Time':['6.7s','0.35s']}
df_c = pd.DataFrame(compare,index = ['1','2'])
df_c
```

[33]:

	Candidate Model	Testing Accuracy(%)	Fitting Time	Prediction Time
1	Ensemble Model	54.6	106s	6.7s
2	Neutral Network	55	994s	0.35s

[34]:
```
eclf
```

[34]:
```
VotingClassifier(estimators=[('lr',
                              LogisticRegression(C=0.01, class_weight=None,
                                                 dual=False, fit_intercept=True,
                                                 intercept_scaling=1,
                                                 l1_ratio=None, max_iter=300,
                                                 multi_class='multinomial',
                                                 n_jobs=None, penalty='l2',
                                                 random_state=None,
                                                 solver='lbfgs', tol=0.0001,
                                                 verbose=0, warm_start=False)),
                             ('lda',
                              LinearDiscriminantAnalysis(n_components=2,
                                                         priors=None,
                                                         shrinkage…
                                                         solver='eigen',
                                                         store_covariance=False,
                                                         tol=0.0001)),
                             ('svm',
                              SVC(C=1e-05, cache_size=200, class_weight=None,
                                  coef0=0.0, decision_function_shape='ovr',
                                  degree=3, gamma='auto_deprecated',
                                  kernel='linear', max_iter=-1,
                                  probability=False, random_state=123,
                                  shrinking=True, tol=0.001, verbose=False))],
                 flatten_transform=True, n_jobs=None, voting='hard',
                 weights=[1.5, 1.5, 1])
```

[39]:
```
# save ensemble model
joblib.dump(eclf,'eclf_model.m')
```

[39]:
```
['eclf_model.m']
```

[41]:
```
# save nn model
joblib.dump(model,'NN.m')
```

[41]:
```
['NN.m']
```

We wanted to compare these two models and using cross-validation (cv=10) to calculate the standard deviation of each one. The details (code and process is in the file cv.ipynb)

Here is the results

[46]:
```
data = {'Model':['NN','Ensemble'],
        'Training Accuracy':['99.92% (sd 0.06%)', '83.57% (sd 0.73%)'],
        'Test Accuracy':['53.86% (sd 1.29%)', '53.48% (sd 1.85%)'],
       'Fit CPU':['90%','71%  50% 14%'],
       'Predict CPU':['"0"%','13.7%'],
       'Fit memory':['+1GB','+2GB +1GB +1GB'],
       'Predict memory':['"0"','+1GB'],
       'Saved Model Size':['16mb','653mb']}
pd.DataFrame(data,index = [1,2])
```

[46]:

	Model	Training Accuracy	Test Accuracy	Fit CPU	Predict CPU	Fit memory	Predict memory	Saved Model Size
1	NN	99.92% (sd 0.06%)	53.86% (sd 1.29%)	90%	"0"%	+1GB	"0"	16mb
2	Ensemble	83.57% (sd 0.73%)	53.48% (sd 1.85%)	71% 50% 14%	13.7%	+2GB +1GB +1GB	+1GB	653mb

From this data frame, we decided to use Neural Network as our final model, the reason is in the following: 1. Both of them have the same test accuracy. 2. Test Accuracy for NN is smaller than Ensemble model. 3. Predict CPU is better than Ensemble model and Predict memory is really small. The memory of ensemble model is huge. 4. Save Model size for NN is much smaller than ensemble model.

Therefore, We choose NN as our advanced model finally.

1.9 Step 8: Advanced Model (Final) — Neural Network

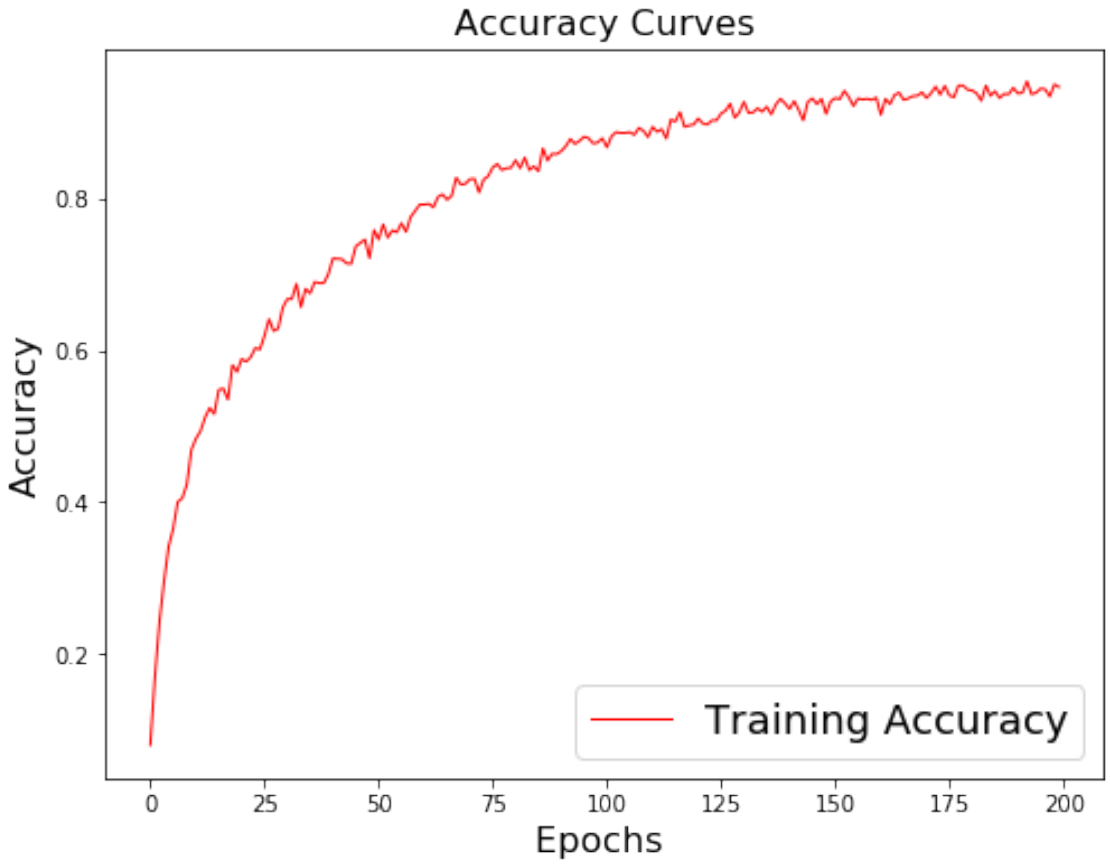
1.9.1 We have already done the Neural Network model in Step 6, so we do not do the fit again. Just show the result here.

[47]:
```
# NN model
model
```

[47]:
```
<keras.engine.training.Model at 0x1a9c312d90>
```

[48]:
```
plt.figure(figsize=[8,6])
plt.plot(model_nn.history['accuracy'],'r',linewidth=1.0)
plt.legend(['Training Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```

[48]:
```
Text(0.5, 1.0, 'Accuracy Curves')
```



[49]:
```
start_time = time.time()
pred = model.predict(X_test)
pred_list = []
for i in range(len(pred)):
    arr = pred[i]
    idx = np.argwhere(arr == np.max(arr))
    pred_list.append(idx[0][0])
tst_labl = np.argmax(y_test, axis=-1)
accuracy = accuracy_score(pred_list, tst_labl)
print("Test accuracy is %s percent" % round(accuracy*100,3))
print("testing model takes %s seconds" % (time.time() - start_time))
```

```
Test accuracy is 54.2 percent
testing model takes 0.4950728416442871 seconds
```

[ ]: