# LAB 7: Machine Learning using the Scikit-learn and the MNIST dataset
# Part B Experiments with Classifiers

Yanling Wu
yw996

November 15, 2018

### 0.1 1: K-Neraest-Neighbor (KNN)

In this section, we will evaluate KNN classifier performance and explore how to improve its performance.

#### 0.1.1 Different Size of Dataset

In order to control the computing time with 10 minutes, I tried to use *StandardScaler* to standardlize the dataset, it did decrease the computing time, but the prediction accuracy decreased more. Hence, I chose to narrow the size of dataset and tried different sizes of training set and test set and we need to take the running time and predicted accuracy into consideration. Intuitively, decreasing the size of training dataset will lead to the decrease of prediction accuracy. Shown as below figures, we can see the prediction score increasd as the datasize increases.

After several attempt, I set the training set to 40,000 and the test set to 6,000, which can guarantee the running time with 10 minutes and the accuracy is not that low.

```
The number of neighbors is 3
    Training time 17.646 seconds
    Test score with 3NN is: 0.9653
    Test time 395.543 seconds
    Test score with 3NN is: 0.9653
```

Figure 3. 30k training dataset and 6k test dataset

```
The number of neighbors is 3
    Training time 2.422 seconds
    Test score with 3NN is: 0.9545
    Test time 50.832 seconds
    Test score with 3NN is: 0.9545
```

Figure 1. 10k training dataset and 2k test dataset

```
The number of neighbors is 3
    Training time 8.107 seconds
    Test score with 3NN is: 0.9613
    Test time 203.844 seconds
    Test score with 3NN is: 0.9613
```

Figure 2.20k training dataset and 4k test dataset

```
The number of neighbors is 3
    Training time 29.882 seconds
    Test score with 3NN is: 0.9662
    Test time 606.459 seconds
    Test score with 3NN is: 0.9662
```

Figure 4. 40k training dataset and 6k test dataset

### 0.1.2 Different Number of Near Neighbor

Next, I tried different number of near neighbors from 1 to 5. The following figure shows the result. Compared 1NN with 3NN, the accuracy of 1NN is bigger than that of 3NN. And the change trend from 1 to 5 near neighbors is down, up and down. The good amount of near neighbors is one or four. But there are higher possibility for 1NN to overfit the data since it needs to consider all of the near neighbors during fitting the model.

```
The number of neighbors is 3
    Training time 29.882 seconds
    Test score with 3NN is: 0.9662
    Test time 606.459 seconds
    Test score with 3NN is: 0.9662
```

Figure 5. The result of different number of near neighboring

## 0.2 2: Multi-Layer Perceptron (MLP)

In this section, we used the MLP classifier to recognize the digits and compared the results of model with one MLP layer to that of model with two MLP layers. I ran the program in Google Colaboratory whose computation speed is very fast. So I used all minist dataset whose total amount is 70,000 and 10,000 of them are as test set. In order to optimal the MLP classifier to get the better score of prediction, we need to change some vital parameters of MLPClassifier including *learning_rate_init*, *max_iter*, *alpha*, *tol*, *alpha*, *solver*.

Specificly, the parameter of *learning_rate_init* will control the size of every step to calculate, which can affect the score of prediction hugely and I changed this parameters to 0.0001. Besides, the parameter of *max_iter* will decide the number of epoches and the times of iterations and will influence the length of calculation. Since we need to control the length of computation of this program within 10 minutes, I changed it to 300 steps. Additionally, the parameter of *tol* will give a criterion to end the calculation and optimization, meaning that when the decreasing value of the loss between every calculation is more than value of *tol*, the training will be ended. The parameter of *solver* will decide what mathematic function and method will be used to train the classifier.

#### 0.2.1 MLP Classifier with One Layer

After every iteration, the loss will decrease and I set the number of iteration to 300 and we can see the loss value of last iteration have already decreased to 0.097 and taining time is 372 seconds. The score of prediction of training set is 0.9709 and the score of predicting of test set is 0.9445. This result is not bad.

```
Iteration 300, loss = 0.09790995
    Training time 372.610 seconds
/usr/local/lib/python3.6/dist-packages/sklearn/neural
  % self.max_iter, ConvergenceWarning)
Training set score: 0.970900
Test set score: 0.944500
    Test time 0.075 seconds
```

Figure 6.the Result of MLP Classifier with One Layer

#### 0.2.2 MLP Classifier with two Layers

In this step, we add one more layer to recognize the digits and try to get better score of prediction. It took 453 seconds to iterate 300 times to calculate two layers model and the loss value of last iteration have declined to 0.067 and the score of prediction of training set is 0.981467 and the score of predicting of test set is 0.9457, which is better than the one layer model, especially the socre of training set.

```
Iteration 300, loss = 0.06736598
    Training time 435.474 seconds
/usr/local/lib/python3.6/dist-packages/sklearn,
  % self.max_iter, ConvergenceWarning)
Training set score: 0.981467
Test set score: 0.945700
    Test time 0.108 seconds
```

Figure 7.the Result of MLP Classifier with One Layer

### 0.3 3: Support Vector Machine (SVM)

Support vector machines are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis.And we could choose different kernel of classifiers to improve the performance of it. Besides linear classificaiton, there are several non-linear classification method.

In this section, we will apply three kinds of classifiers and compare their performances and control their running time with 10 minutes.

**SVM with a linear funciton** We tried to build a SVM Classifiers with a linear function using 30,000 training data and 6000 test data.Training time is 308.556 seconds and the accuracy is 0.9038, which is not so high.

```
Linear SVM Training time 308.556 seconds
Linear SVM Test set score: 0.901833
Linear SVM Test time 49.752 seconds
```

Figure 8.the Result of SVM Classifier with Linear function

**SVM with a polynomial funciton** We tried to build a SVM Classifiers with a polynomial function using 30,000 training data and 6000 test data. Training time is 177.658 seconds and the accuracy is 0.971667, which is far higher than that of classifier with a linear function.

```
Training time 177.658 seconds
Test set score: 0.971667
Test time 43.641 seconds
```

Figure 9.the Result of SVM Classifier with Polynomial function

**SVM with radial basis function** We tried to build a SVM Classifiers with a polynomial function using 40,000 training data and 6000 test data. Training time is 372.252 seconds and the accuracy is 0.93133, which is far higher than that of classifier with a linear function.

I also found two interesting things. One is that the realtionship between the size of dataset and the length of running time is noe linear and more like exponential function. Another is that the radial basis function is very sensitive to standarlization of dataset. If there is no syntax like *X /= 255.* , this *rbf* kernel will give very bad prediction and the score will only be 0.11.

```
rbf SVM Training time 376.252 seconds
rbf SVM Test set score: 0.931333
rbf SVM Test time 88.793 seconds
```

Figure 10.the Result of SVM Classifier with radial basis function

**Reminder: There might be no output results in below cell, this is becasue I ran these code in different file and copied codes together.**

In [3]:
```
#import libraries

%matplotlib inline
import io
import time
from scipy.io.arff import loadarff
import matplotlib.pyplot as plt
from sklearn.datasets import get_data_home
from sklearn.externals.joblib import Memory
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
```

In [4]:
```
#Read the dataset
try:
    from urllib.request import urlopen
```

```
except ImportError:
    # Python 2
    from urllib2 import urlopen

print(__doc__)

memory = Memory(get_data_home())


@memory.cache()
def fetch_mnist():
    content = urlopen(
        'https://www.openml.org/data/download/52667/mnist_784.arff').read()
    data, meta = loadarff(io.StringIO(content.decode('utf8')))
    data = data.view([('pixels', '<f8', 784), ('class', '|S1')])
    return data['pixels'], data['class']

X, y = fetch_mnist()
X /= 255.
# rescale the data, use the traditional train/test split
#the size of train set is 60k and the size of test size is 10k
X_train, X_test = X[:40000], X[40000:46000]
y_train, y_test = y[:40000], y[40000:46000]
```

```
Automatically created module for IPython interactive environment
```

In [0]:
```
#Evaluate a K-NN classifier
for i in range(1,6):
  print("The number of neighbors is %d: " % i)
  start_time = time.time()
  #Classifier Declaration
  KNN = KNeighborsClassifier(n_neighbors=i)
  #Train the classifier
  KNN.fit(X_train,y_train)
  train_time = time.time() - start_time
  start_time = time.time()
  print("     Training time %.3f seconds" % train_time)
  #Evaluate the result
  score = KNN.score(X_test,y_test)
  print("     Test score with %dNN is: %.4f" % (i, score))
  test_time = time.time() - start_time
  print("     Test time %.3f seconds" % test_time)
  print("     Test score with %dNN is: %.4f" % (i,score))
```

```
The number of neighbors is 1:
     Training time 23.620 seconds
     Test score with 1NN is: 0.9682
```

```
    Test time 407.259 seconds
    Test score with 1NN is: 0.9682
The number of neighbors is 2:
    Training time 23.884 seconds
    Test score with 2NN is: 0.9620
    Test time 407.892 seconds
    Test score with 2NN is: 0.9620
The number of neighbors is 3:
    Training time 23.654 seconds
    Test score with 3NN is: 0.9662
    Test time 457.245 seconds
    Test score with 3NN is: 0.9662
The number of neighbors is 4:
    Training time 23.705 seconds
    Test score with 4NN is: 0.9667
    Test time 421.913 seconds
    Test score with 4NN is: 0.9667
The number of neighbors is 5:
    Training time 23.669 seconds
    Test score with 5NN is: 0.9663
    Test time 406.560 seconds
    Test score with 5NN is: 0.9663
```

In [0]:
```
### Multi-layer perceptron Classifier
#
# Single hidden layer
start_time = time.time()
mlp = MLPClassifier(hidden_layer_sizes=(50,), max_iter=300, alpha=1e-5,
                    solver='sgd', verbose=10, tol=1e-5, random_state=1,
                    learning_rate_init=.0001)

mlp.fit(X_train, y_train)
train_time = time.time() - start_time
print("    Training time %.3f seconds" % train_time)
print("Training set score: %f" % mlp.score(X_train, y_train))
start_time = time.time()
print("Test set score: %f" % mlp.score(X_test, y_test))
test_time = time.time() - start_time
print("    Test time %.3f seconds" % test_time)

fig, axes = plt.subplots(4, 4)
# use global min / max to ensure all weights are shown on the same scale
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
               vmax=.5 * vmax)
    ax.set_xticks(())
```

```
            ax.set_yticks(())

        plt.show()
```

```
Iteration 1, loss = 2.31873444
Iteration 2, loss = 1.15623711
Iteration 3, loss = 0.90721258
Iteration 4, loss = 0.78289191
Iteration 5, loss = 0.70110037
Iteration 6, loss = 0.63781107
Iteration 7, loss = 0.59398188
Iteration 8, loss = 0.55855692
Iteration 9, loss = 0.52846498
Iteration 10, loss = 0.50488183
Iteration 11, loss = 0.48572582
Iteration 12, loss = 0.46790671
Iteration 13, loss = 0.45218744
Iteration 14, loss = 0.43800788
Iteration 15, loss = 0.42549022
Iteration 16, loss = 0.41308816
Iteration 17, loss = 0.40090918
Iteration 18, loss = 0.39053311
Iteration 19, loss = 0.37950472
Iteration 20, loss = 0.36989921
Iteration 21, loss = 0.36159076
Iteration 22, loss = 0.35360963
Iteration 23, loss = 0.34535371
Iteration 24, loss = 0.33673483
Iteration 25, loss = 0.32996003
Iteration 26, loss = 0.32164059
Iteration 27, loss = 0.31491139
Iteration 28, loss = 0.30793868
Iteration 29, loss = 0.30128505
Iteration 30, loss = 0.29551798
Iteration 31, loss = 0.29047461
Iteration 32, loss = 0.28619027
Iteration 33, loss = 0.28160394
Iteration 34, loss = 0.27757461
Iteration 35, loss = 0.27388301
Iteration 36, loss = 0.26995744
Iteration 37, loss = 0.26681359
Iteration 38, loss = 0.26371218
Iteration 39, loss = 0.26014049
Iteration 40, loss = 0.25740091
Iteration 41, loss = 0.25465179
Iteration 42, loss = 0.25163958
Iteration 43, loss = 0.24888187
Iteration 44, loss = 0.24654647
```

```
Iteration 45, loss = 0.24332008
Iteration 46, loss = 0.24069375
Iteration 47, loss = 0.23858355
Iteration 48, loss = 0.23568965
Iteration 49, loss = 0.23326901
Iteration 50, loss = 0.23102953
Iteration 51, loss = 0.22862093
Iteration 52, loss = 0.22654744
Iteration 53, loss = 0.22405408
Iteration 54, loss = 0.22240206
Iteration 55, loss = 0.21964725
Iteration 56, loss = 0.21734006
Iteration 57, loss = 0.21593882
Iteration 58, loss = 0.21356789
Iteration 59, loss = 0.21122427
Iteration 60, loss = 0.20986988
Iteration 61, loss = 0.20753796
Iteration 62, loss = 0.20552051
Iteration 63, loss = 0.20387871
Iteration 64, loss = 0.20274987
Iteration 65, loss = 0.20059544
Iteration 66, loss = 0.19914538
Iteration 67, loss = 0.19753161
Iteration 68, loss = 0.19604687
Iteration 69, loss = 0.19466012
Iteration 70, loss = 0.19347300
Iteration 71, loss = 0.19239917
Iteration 72, loss = 0.19083935
Iteration 73, loss = 0.18978491
Iteration 74, loss = 0.18869764
Iteration 75, loss = 0.18729116
Iteration 76, loss = 0.18639925
Iteration 77, loss = 0.18519216
Iteration 78, loss = 0.18374924
Iteration 79, loss = 0.18321397
Iteration 80, loss = 0.18150545
Iteration 81, loss = 0.18090267
Iteration 82, loss = 0.17938187
Iteration 83, loss = 0.17864592
Iteration 84, loss = 0.17723672
Iteration 85, loss = 0.17687366
Iteration 86, loss = 0.17565105
Iteration 87, loss = 0.17473061
Iteration 88, loss = 0.17402527
Iteration 89, loss = 0.17294163
Iteration 90, loss = 0.17243182
Iteration 91, loss = 0.17154515
Iteration 92, loss = 0.17099145
```

```
Iteration 93, loss = 0.16996444
Iteration 94, loss = 0.16914097
Iteration 95, loss = 0.16855611
Iteration 96, loss = 0.16781354
Iteration 97, loss = 0.16723710
Iteration 98, loss = 0.16641741
Iteration 99, loss = 0.16550781
Iteration 100, loss = 0.16494696
Iteration 101, loss = 0.16391333
Iteration 102, loss = 0.16285748
Iteration 103, loss = 0.16240395
Iteration 104, loss = 0.16193133
Iteration 105, loss = 0.16130471
Iteration 106, loss = 0.16059459
Iteration 107, loss = 0.15960714
Iteration 108, loss = 0.15919719
Iteration 109, loss = 0.15838710
Iteration 110, loss = 0.15843687
Iteration 111, loss = 0.15755216
Iteration 112, loss = 0.15665204
Iteration 113, loss = 0.15637297
Iteration 114, loss = 0.15538610
Iteration 115, loss = 0.15523950
Iteration 116, loss = 0.15458034
Iteration 117, loss = 0.15396650
Iteration 118, loss = 0.15348568
Iteration 119, loss = 0.15316056
Iteration 120, loss = 0.15267678
Iteration 121, loss = 0.15196448
Iteration 122, loss = 0.15140381
Iteration 123, loss = 0.15066570
Iteration 124, loss = 0.15044067
Iteration 125, loss = 0.14993841
Iteration 126, loss = 0.14935365
Iteration 127, loss = 0.14863132
Iteration 128, loss = 0.14849108
Iteration 129, loss = 0.14854981
Iteration 130, loss = 0.14746038
Iteration 131, loss = 0.14734307
Iteration 132, loss = 0.14678454
Iteration 133, loss = 0.14616585
Iteration 134, loss = 0.14544773
Iteration 135, loss = 0.14509118
Iteration 136, loss = 0.14499581
Iteration 137, loss = 0.14438060
Iteration 138, loss = 0.14394391
Iteration 139, loss = 0.14375013
Iteration 140, loss = 0.14296720
```

```
Iteration 141, loss = 0.14314319
Iteration 142, loss = 0.14256443
Iteration 143, loss = 0.14179873
Iteration 144, loss = 0.14133943
Iteration 145, loss = 0.14110915
Iteration 146, loss = 0.14115561
Iteration 147, loss = 0.14044585
Iteration 148, loss = 0.14001760
Iteration 149, loss = 0.13938400
Iteration 150, loss = 0.13901351
Iteration 151, loss = 0.13873104
Iteration 152, loss = 0.13859187
Iteration 153, loss = 0.13809624
Iteration 154, loss = 0.13754044
Iteration 155, loss = 0.13711536
Iteration 156, loss = 0.13676482
Iteration 157, loss = 0.13655506
Iteration 158, loss = 0.13622600
Iteration 159, loss = 0.13579527
Iteration 160, loss = 0.13538972
Iteration 161, loss = 0.13490434
Iteration 162, loss = 0.13452621
Iteration 163, loss = 0.13437402
Iteration 164, loss = 0.13399865
Iteration 165, loss = 0.13358385
Iteration 166, loss = 0.13306798
Iteration 167, loss = 0.13252260
Iteration 168, loss = 0.13214522
Iteration 169, loss = 0.13203767
Iteration 170, loss = 0.13189212
Iteration 171, loss = 0.13129969
Iteration 172, loss = 0.13070261
Iteration 173, loss = 0.13037170
Iteration 174, loss = 0.13038073
Iteration 175, loss = 0.13022410
Iteration 176, loss = 0.12958255
Iteration 177, loss = 0.12938374
Iteration 178, loss = 0.12915958
Iteration 179, loss = 0.12853625
Iteration 180, loss = 0.12764193
Iteration 181, loss = 0.12798824
Iteration 182, loss = 0.12729224
Iteration 183, loss = 0.12690774
Iteration 184, loss = 0.12693984
Iteration 185, loss = 0.12600704
Iteration 186, loss = 0.12600159
Iteration 187, loss = 0.12549985
Iteration 188, loss = 0.12552274
```

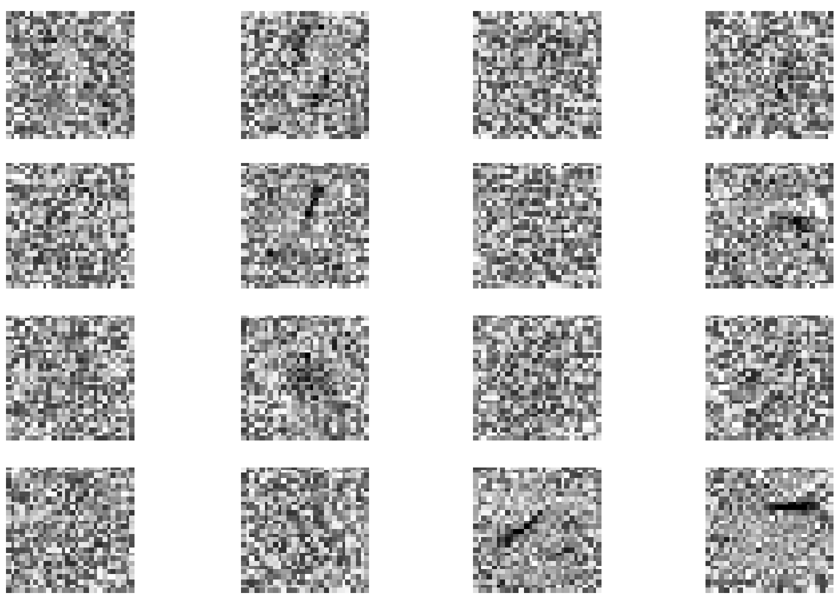
```
Iteration 189, loss = 0.12473630
Iteration 190, loss = 0.12456259
Iteration 191, loss = 0.12417117
Iteration 192, loss = 0.12389621
Iteration 193, loss = 0.12363118
Iteration 194, loss = 0.12333332
Iteration 195, loss = 0.12305087
Iteration 196, loss = 0.12265024
Iteration 197, loss = 0.12248247
Iteration 198, loss = 0.12203635
Iteration 199, loss = 0.12202520
Iteration 200, loss = 0.12152678
Iteration 201, loss = 0.12114582
Iteration 202, loss = 0.12109901
Iteration 203, loss = 0.12044461
Iteration 204, loss = 0.12025410
Iteration 205, loss = 0.11976158
Iteration 206, loss = 0.11974289
Iteration 207, loss = 0.11941659
Iteration 208, loss = 0.11929975
Iteration 209, loss = 0.11892507
Iteration 210, loss = 0.11832592
Iteration 211, loss = 0.11831234
Iteration 212, loss = 0.11815542
Iteration 213, loss = 0.11769222
Iteration 214, loss = 0.11747328
Iteration 215, loss = 0.11707318
Iteration 216, loss = 0.11687918
Iteration 217, loss = 0.11644429
Iteration 218, loss = 0.11629496
Iteration 219, loss = 0.11600984
Iteration 220, loss = 0.11565793
Iteration 221, loss = 0.11558371
Iteration 222, loss = 0.11531771
Iteration 223, loss = 0.11540476
Iteration 224, loss = 0.11455055
Iteration 225, loss = 0.11435280
Iteration 226, loss = 0.11391935
Iteration 227, loss = 0.11359069
Iteration 228, loss = 0.11346089
Iteration 229, loss = 0.11309546
Iteration 230, loss = 0.11277466
Iteration 231, loss = 0.11266508
Iteration 232, loss = 0.11236592
Iteration 233, loss = 0.11181770
Iteration 234, loss = 0.11194775
Iteration 235, loss = 0.11155141
Iteration 236, loss = 0.11148631
```

```
Iteration 237, loss = 0.11086880
Iteration 238, loss = 0.11048006
Iteration 239, loss = 0.11048949
Iteration 240, loss = 0.11011562
Iteration 241, loss = 0.10992562
Iteration 242, loss = 0.10982629
Iteration 243, loss = 0.10897588
Iteration 244, loss = 0.10913553
Iteration 245, loss = 0.10881733
Iteration 246, loss = 0.10843473
Iteration 247, loss = 0.10847295
Iteration 248, loss = 0.10839309
Iteration 249, loss = 0.10837179
Iteration 250, loss = 0.10761648
Iteration 251, loss = 0.10743634
Iteration 252, loss = 0.10747278
Iteration 253, loss = 0.10677149
Iteration 254, loss = 0.10662121
Iteration 255, loss = 0.10653065
Iteration 256, loss = 0.10610956
Iteration 257, loss = 0.10605380
Iteration 258, loss = 0.10553833
Iteration 259, loss = 0.10547874
Iteration 260, loss = 0.10547655
Iteration 261, loss = 0.10493371
Iteration 262, loss = 0.10464989
Iteration 263, loss = 0.10491113
Iteration 264, loss = 0.10463297
Iteration 265, loss = 0.10437494
Iteration 266, loss = 0.10384280
Iteration 267, loss = 0.10419341
Iteration 268, loss = 0.10374185
Iteration 269, loss = 0.10363650
Iteration 270, loss = 0.10304112
Iteration 271, loss = 0.10307877
Iteration 272, loss = 0.10299605
Iteration 273, loss = 0.10272854
Iteration 274, loss = 0.10247484
Iteration 275, loss = 0.10203062
Iteration 276, loss = 0.10214378
Iteration 277, loss = 0.10177714
Iteration 278, loss = 0.10178876
Iteration 279, loss = 0.10145863
Iteration 280, loss = 0.10115999
Iteration 281, loss = 0.10089446
Iteration 282, loss = 0.10091111
Iteration 283, loss = 0.10087082
Iteration 284, loss = 0.10035896
```

```
Iteration 285, loss = 0.10033624
Iteration 286, loss = 0.10046364
Iteration 287, loss = 0.10016383
Iteration 288, loss = 0.09995543
Iteration 289, loss = 0.09973476
Iteration 290, loss = 0.09943315
Iteration 291, loss = 0.09912233
Iteration 292, loss = 0.09911057
Iteration 293, loss = 0.09901436
Iteration 294, loss = 0.09876133
Iteration 295, loss = 0.09853774
Iteration 296, loss = 0.09865721
Iteration 297, loss = 0.09831805
Iteration 298, loss = 0.09819467
Iteration 299, loss = 0.09787703
Iteration 300, loss = 0.09790995
    Training time 372.610 seconds
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/neural_network/multilayer_perceptron.py:564: Conv
  % self.max_iter, ConvergenceWarning)
```

```
Training set score: 0.970900
Test set score: 0.944500
    Test time 0.075 seconds
```



In [0]:
```
### Multi-layer perceptron Classifier
#
# Two hidden layers
start_time = time.time()
# Two hidden layers, each has 50 elements
mlp_2 = MLPClassifier(hidden_layer_sizes=(50,50), max_iter=10, alpha=1e-4,
                      solver='sgd', verbose=10, tol=1e-4, random_state=1,
                      learning_rate_init=.1)

mlp.fit(X_train, y_train)
train_time = time.time() - start_time
print("    Training time %.3f seconds" % train_time)
print("Training set score: %f" % mlp_2.score(X_train, y_train))
start_time = time.time()
print("Test set score: %f" % mlp_2.score(X_test, y_test))
test_time = time.time() - start_time
print("    Test time %.3f seconds" % test_time)
```

In [0]:
```
### Support Vector Nachine (SVM) Classifier

### Linear

start_time = time.time()
## Linear
svc = svm.SVC(kernel = 'linear', C = 1)

svc.fit(X_train, y_train)
train_time = time.time() - start_time
print("Linear SVM Training time %.3f seconds" % train_time)

start_time = time.time()
score = svc.score(X_test, y_test)
print("Linear SVM Test set score: %f" % score)

test_time = time.time() - start_time
print("Linear SVM Test time %.3f seconds" % test_time)
```

In [0]:
```
### Support Vector Nachine (SVM) Classifier

### Poly
start_time = time.time()
# ## cubic polynomial
svc = svm.SVC(kernel = 'poly',degree = 3, C = 1)
# ##
# ## Radial basis functions
# svc = svm.SVC(kernel = 'rbf', C = 2, gamma = 0.0005)

svc.fit(X_train, y_train)
```

```
        train_time = time.time() - start_time
        print(" SVM Training time %.3f seconds" % train_time)

        start_time = time.time()
        score = svc.score(X_test, y_test)
        print(" SVM Test set score: %f" % score)

        test_time = time.time() - start_time
        print(" SVM Test time %.3f seconds" % test_time)
```

```
In [0]: ###  Support Vector Nachine (SVM) Classifier

        ### rbf
        start_time = time.time()
        ##
        # ## Radial basis functions
        svc = svm.SVC(kernel = 'rbf', C = 2, gamma = 'auto')

        svc.fit(X_train, y_train)
        train_time = time.time() - start_time
        print("rbf SVM Training time %.3f seconds" % train_time)

        start_time = time.time()
        score = svc.score(X_test, y_test)
        print("rbf SVM Test set score: %f" % score)

        test_time = time.time() - start_time
        print("rbf SVM Test time %.3f seconds" % test_time)
```