Question 2:

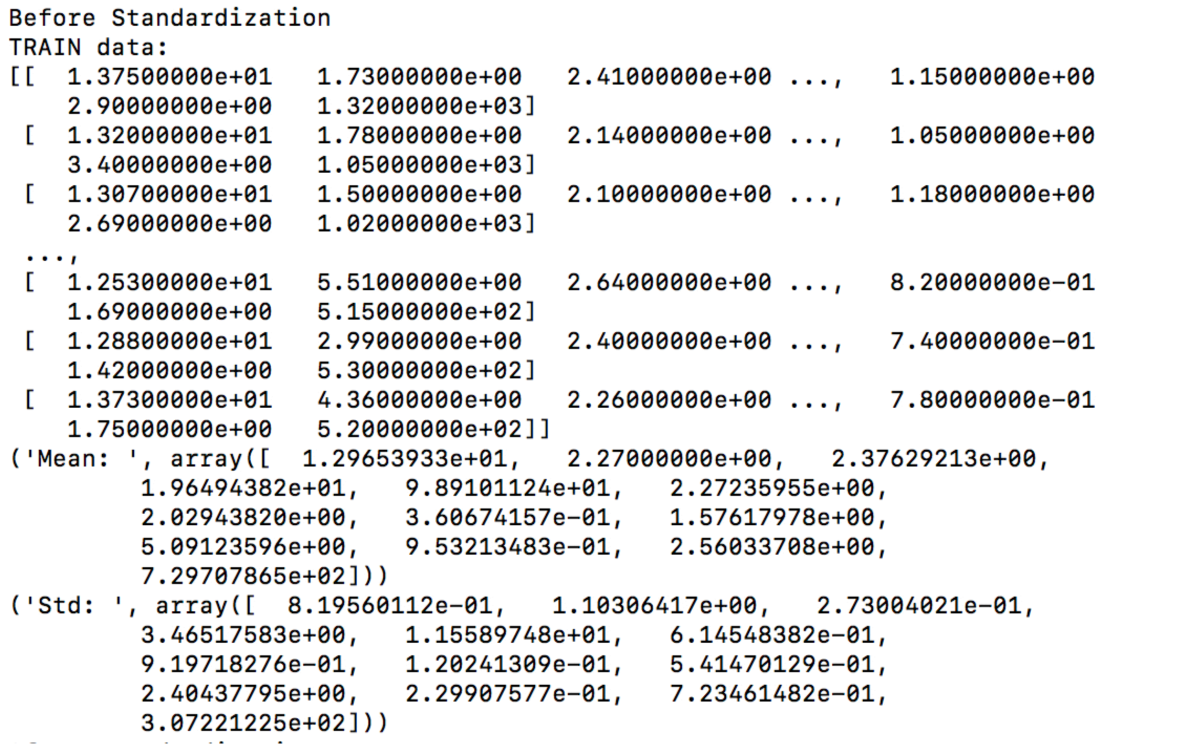
AIM:

To compare the classification performance of ***Perceptron*** and ***MSE classifiers*** for the given data set.
(a) To state the programming language used.
(b) To read and Display the un-normalized data and standardized (normalized) data for both Train and Test data sets. Explain why normalizing factor should be calculated using Training data only.
(c)
 (i) What is the default initial weight vector?
 (ii) What is halting condition? If the solution weight vector is not reached, what's the back-up.
(d)
 (i) Use the first 2 features and apply the perceptron learning algorithm. Report the weight vector and classification accuracy for both training and test data sets.
 (ii) Repeat (d)(i) but with all 13 features.
(e)
 (i) Use the first 2 features and apply the perceptron learning algorithm for random initial weight vector. Do this 100 times and find the run that has the *best performance on the training data set*. If there is a tie, pick the weight vector at random. Report the final weight vector, classification accuracy rates on both train and test data.
 (ii) Repeat (e)(i) with all 13 features.
(f) Compare the test results for 2 and 13 features in (d) and (e) part.
(g) Use the MSE classifier or the Pseudoinverse classifier on un-normalized data sets and report the training and test classification accuracy.
(h) Repeat (g) with normalized data.
(i) Compare (g) and (h)
(j) Compare (h) and (e)

PROGRAM CODE:

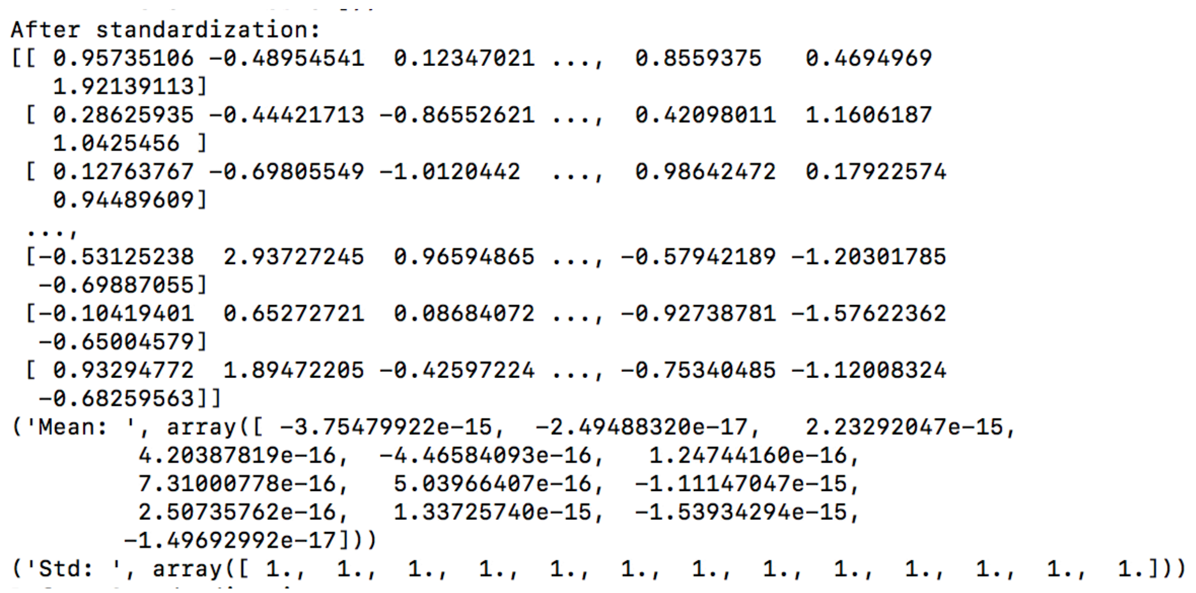
Attached in CODE_EE559_HW6_HarikrishnaPrabhu.
(i) Code 7.py
(ii) Code 8.py

OBSERVATION:



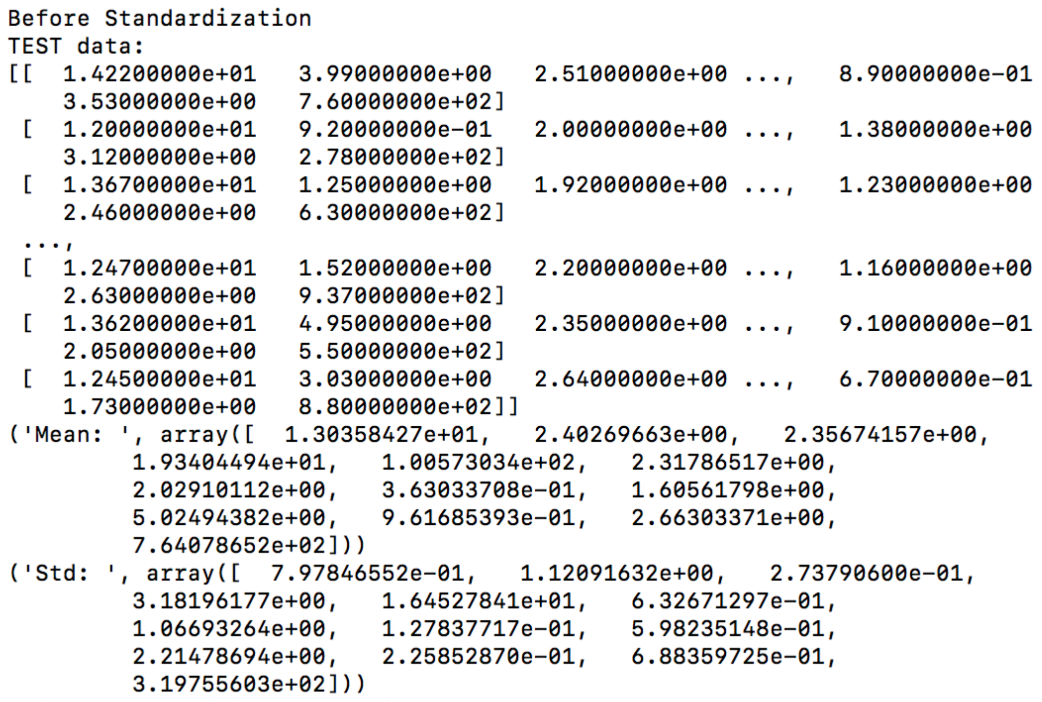
```
Before Standardization
TRAIN data:
[[  1.37500000e+01   1.73000000e+00   2.41000000e+00 ...,   1.15000000e+00
    2.90000000e+00   1.32000000e+03]
 [  1.32000000e+01   1.78000000e+00   2.14000000e+00 ...,   1.05000000e+00
    3.40000000e+00   1.05000000e+03]
 [  1.30700000e+01   1.50000000e+00   2.10000000e+00 ...,   1.18000000e+00
    2.69000000e+00   1.02000000e+03]
 ...,
 [  1.25300000e+01   5.51000000e+00   2.64000000e+00 ...,   8.20000000e-01
    1.69000000e+00   5.15000000e+02]
 [  1.28800000e+01   2.99000000e+00   2.40000000e+00 ...,   7.40000000e-01
    1.42000000e+00   5.30000000e+02]
 [  1.37300000e+01   4.36000000e+00   2.26000000e+00 ...,   7.80000000e-01
    1.75000000e+00   5.20000000e+02]]
('Mean: ', array([  1.29653933e+01,   2.27000000e+00,   2.37629213e+00,
         1.96494382e+01,   9.89101124e+01,   2.27235955e+00,
         2.02943820e+00,   3.60674157e-01,   1.57617978e+00,
         5.09123596e+00,   9.53213483e-01,   2.56033708e+00,
         7.29707865e+02]))
('Std: ', array([  8.19560112e-01,   1.10306417e+00,   2.73004021e-01,
         3.46517583e+00,   1.15589748e+01,   6.14548382e-01,
         9.19718276e-01,   1.20241309e-01,   5.41470129e-01,
         2.40437795e+00,   2.29907577e-01,   7.23461482e-01,
         3.07221225e+02]))
```

Figure 1: Train data Un-normalized.



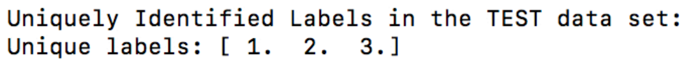
```
After standardization:
[[ 0.95735106 -0.48954541  0.12347021 ...,  0.8559375   0.4694969
   1.92139113]
 [ 0.28625935 -0.44421713 -0.86552621 ...,  0.42098011  1.1606187
   1.0425456 ]
 [ 0.12763767 -0.69805549 -1.0120442  ...,  0.98642472  0.17922574
   0.94489609]
 ...,
 [-0.53125238  2.93727245  0.96594865 ..., -0.57942189 -1.20301785
  -0.69887055]
 [-0.10419401  0.65272721  0.08684072 ..., -0.92738781 -1.57622362
  -0.65004579]
 [ 0.93294772  1.89472205 -0.42597224 ..., -0.75340485 -1.12008324
  -0.68259563]]
('Mean: ', array([ -3.75479922e-15,  -2.49488320e-17,   2.23292047e-15,
         4.20387819e-16,  -4.46584093e-16,   1.24744160e-16,
         7.31000778e-16,   5.03966407e-16,  -1.11147047e-15,
         2.50735762e-16,   1.33725740e-15,  -1.53934294e-15,
        -1.49692992e-17]))
('Std: ', array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]))
```

Figure 2: Train data Normalized.

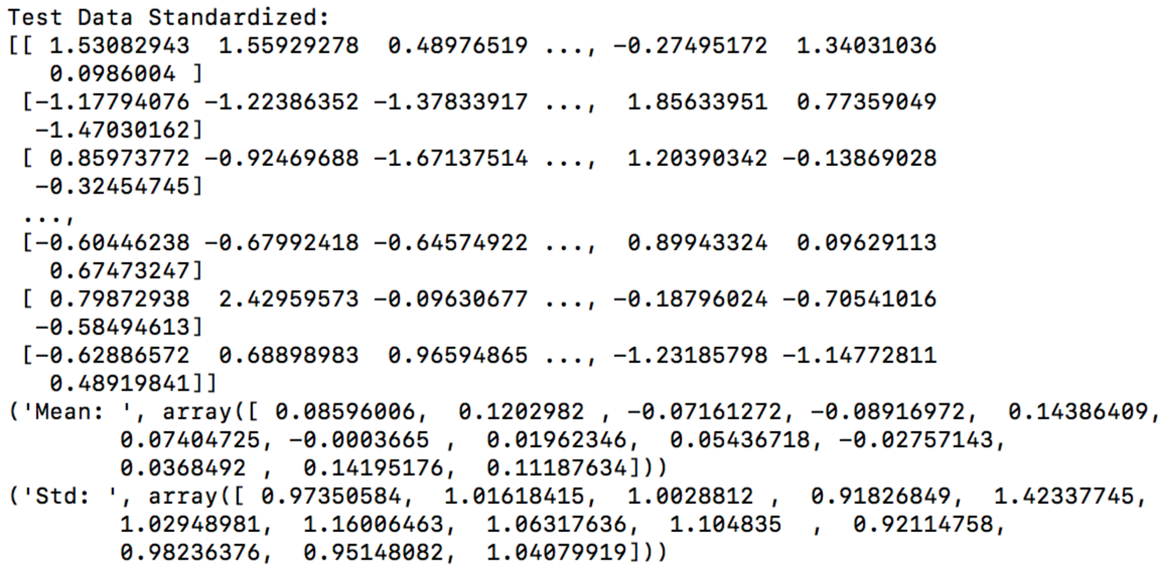


```
Before Standardization
TEST data:
[[  1.42200000e+01   3.99000000e+00   2.51000000e+00 ...,   8.90000000e-01
    3.53000000e+00   7.60000000e+02]
 [  1.20000000e+01   9.20000000e-01   2.00000000e+00 ...,   1.38000000e+00
    3.12000000e+00   2.78000000e+02]
 [  1.36700000e+01   1.25000000e+00   1.92000000e+00 ...,   1.23000000e+00
    2.46000000e+00   6.30000000e+02]
 ...,
 [  1.24700000e+01   1.52000000e+00   2.20000000e+00 ...,   1.16000000e+00
    2.63000000e+00   9.37000000e+02]
 [  1.36200000e+01   4.95000000e+00   2.35000000e+00 ...,   9.10000000e-01
    2.05000000e+00   5.50000000e+02]
 [  1.24500000e+01   3.03000000e+00   2.64000000e+00 ...,   6.70000000e-01
    1.73000000e+00   8.80000000e+02]]
('Mean: ', array([  1.30358427e+01,   2.40269663e+00,   2.35674157e+00,
         1.93404494e+01,   1.00573034e+02,   2.31786517e+00,
         2.02910112e+00,   3.63033708e-01,   1.60561798e+00,
         5.02494382e+00,   9.61685393e-01,   2.66303371e+00,
         7.64078652e+02]))
('Std: ', array([  7.97846552e-01,   1.12091632e+00,   2.73790600e-01,
         3.18196177e+00,   1.64527841e+01,   6.32671297e-01,
         1.06693264e+00,   1.27837717e-01,   5.98235148e-01,
         2.21478694e+00,   2.25852870e-01,   6.88359725e-01,
         3.19755603e+02]))
```

Figure 3: Test data Un-normalized.

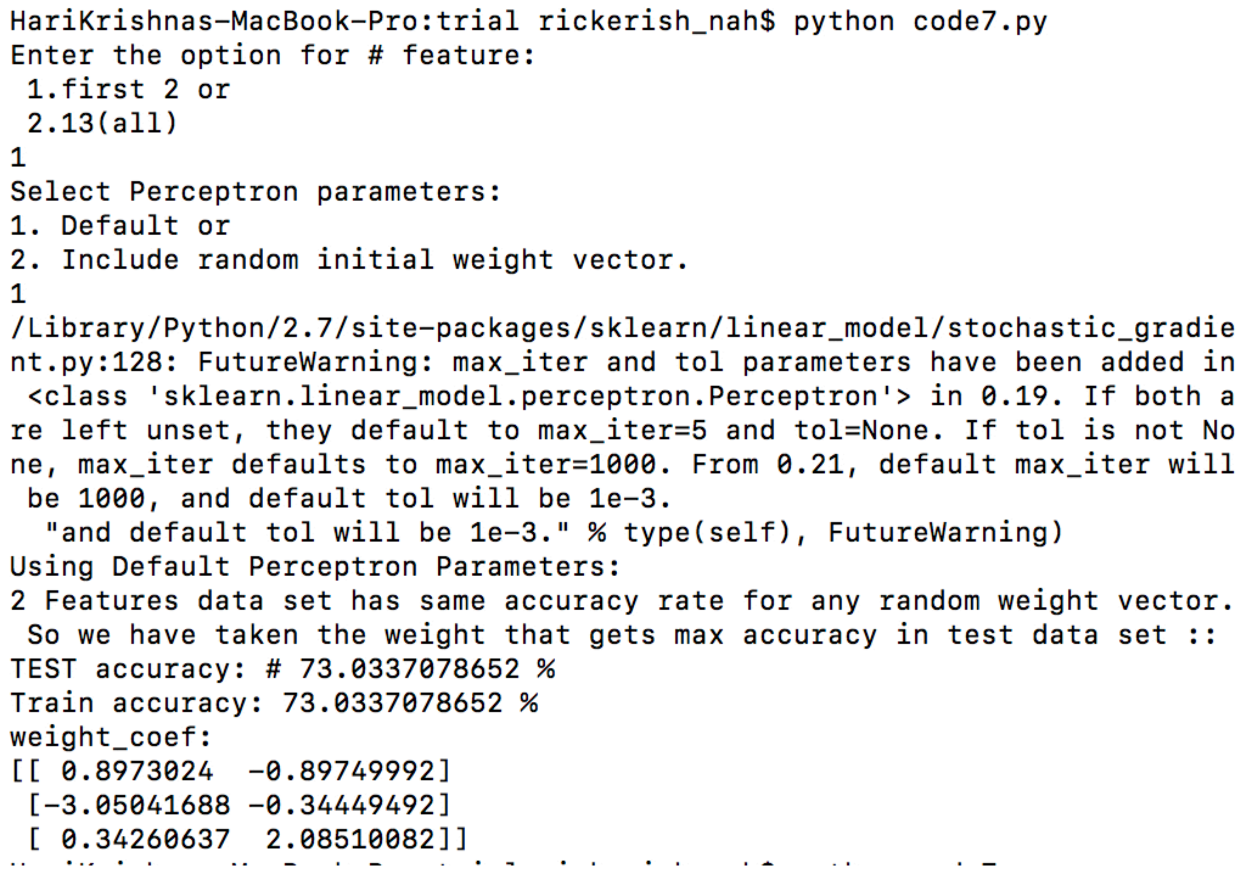


```
Uniquely Identified Labels in the TEST data set:
Unique labels: [ 1.  2.  3.]
```



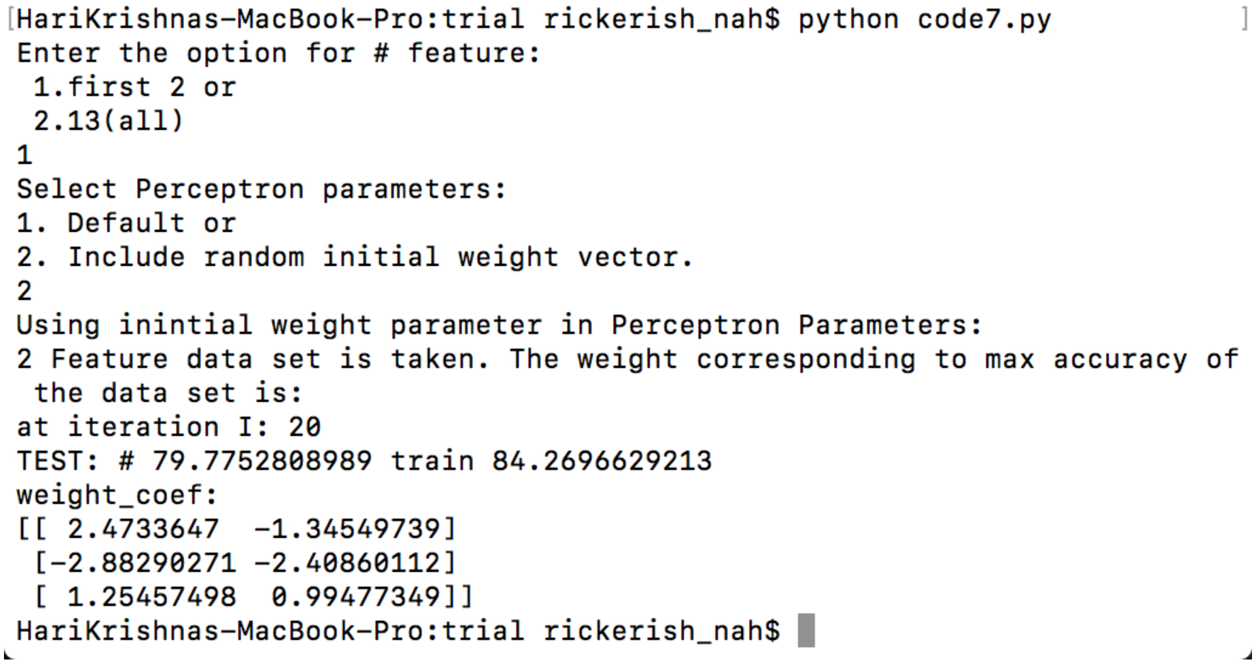
```
Test Data Standardized:
[[ 1.53082943  1.55929278  0.48976519 ..., -0.27495172  1.34031036
   0.0986004 ]
 [-1.17794076 -1.22386352 -1.37833917 ...,  1.85633951  0.77359049
  -1.47030162]
 [ 0.85973772 -0.92469688 -1.67137514 ...,  1.20390342 -0.13869028
  -0.32454745]
 ...,
 [-0.60446238 -0.67992418 -0.64574922 ...,  0.89943324  0.09629113
   0.67473247]
 [ 0.79872938  2.42959573 -0.09630677 ..., -0.18796024 -0.70541016
  -0.58494613]
 [-0.62886572  0.68898983  0.96594865 ..., -1.23185798 -1.14772811
   0.48919841]]
('Mean: ', array([ 0.08596006,  0.1202982 , -0.07161272, -0.08916972,  0.14386409,
        0.07404725, -0.0003665 ,  0.01962346,  0.05436718, -0.02757143,
        0.0368492 ,  0.14195176,  0.11187634]))
('Std: ', array([ 0.97350584,  1.01618415,  1.0028812 ,  0.91826849,  1.42337745,
        1.02948981,  1.16006463,  1.06317636,  1.104835  ,  0.92114758,
        0.98236376,  0.95148082,  1.04079919]))
```

Figure 4: Test data Normalized.



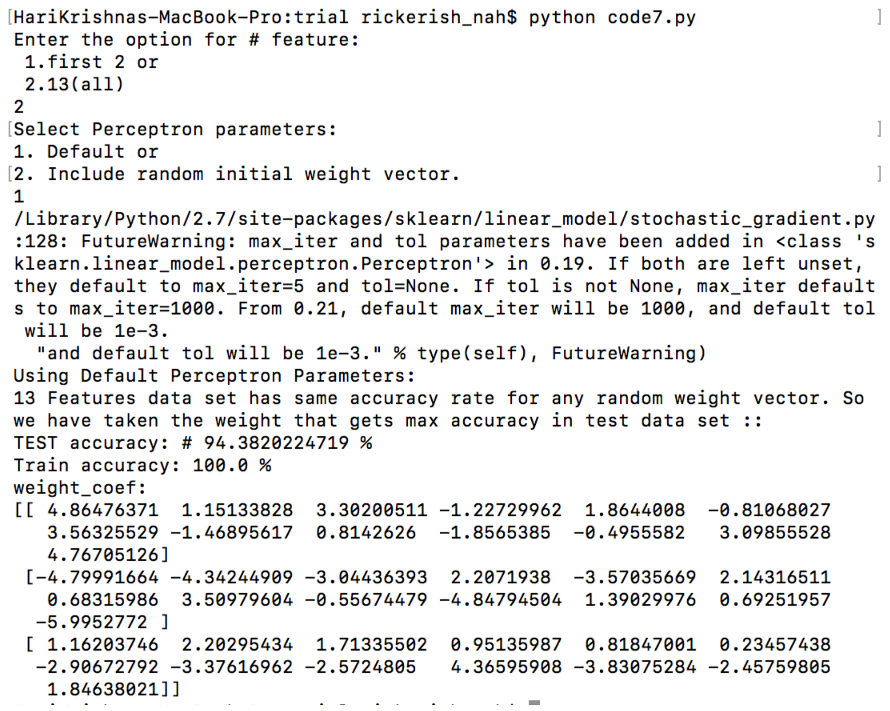
```
HariKrishnas-MacBook-Pro:trial rickerish_nah$ python code7.py
Enter the option for # feature:
 1.first 2 or
 2.13(all)
1
Select Perceptron parameters:
1. Default or
2. Include random initial weight vector.
1
/Library/Python/2.7/site-packages/sklearn/linear_model/stochastic_gradie
nt.py:128: FutureWarning: max_iter and tol parameters have been added in
 <class 'sklearn.linear_model.perceptron.Perceptron'> in 0.19. If both a
re left unset, they default to max_iter=5 and tol=None. If tol is not No
ne, max_iter defaults to max_iter=1000. From 0.21, default max_iter will
 be 1000, and default tol will be 1e-3.
  "and default tol will be 1e-3." % type(self), FutureWarning)
Using Default Perceptron Parameters:
2 Features data set has same accuracy rate for any random weight vector.
 So we have taken the weight that gets max accuracy in test data set ::
TEST accuracy: # 73.0337078652 %
Train accuracy: 73.0337078652 %
weight_coef:
[[ 0.8973024  -0.89749992]
 [-3.05041688 -0.34449492]
 [ 0.34260637  2.08510082]]
```

Figure 5: Perceptron Learning algorithm: 2 features (Default initial weight vector).



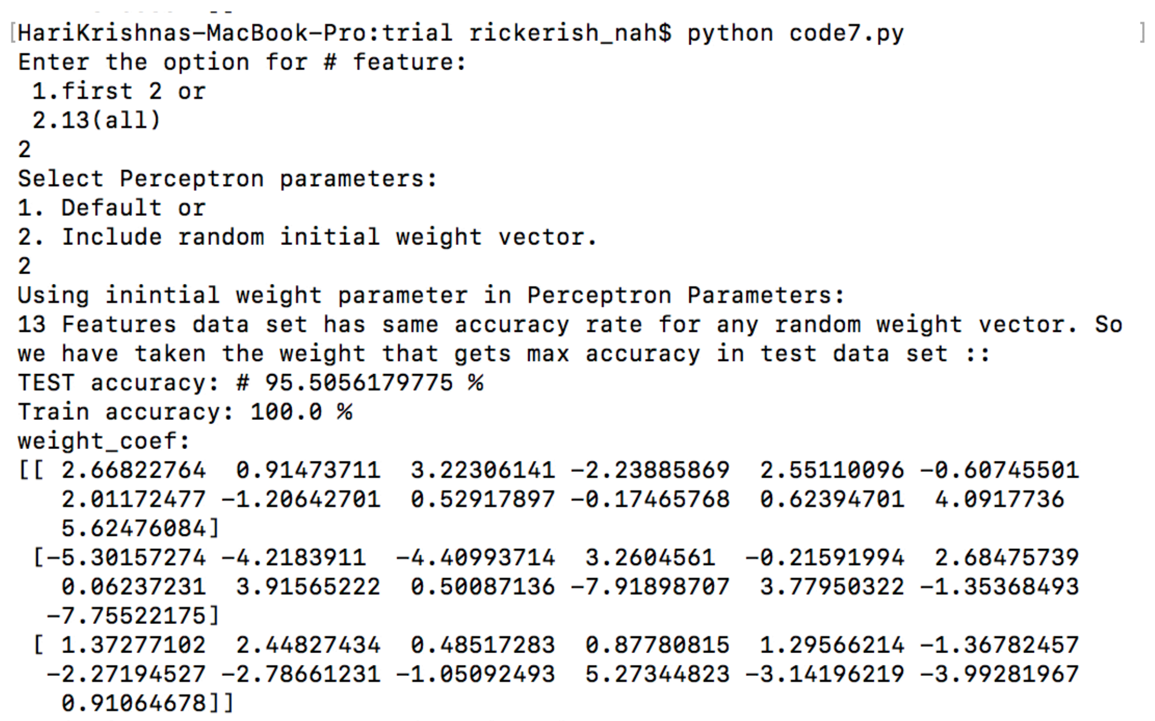
```
HariKrishnas-MacBook-Pro:trial rickerish_nah$ python code7.py
Enter the option for # feature:
 1.first 2 or
 2.13(all)
1
Select Perceptron parameters:
1. Default or
2. Include random initial weight vector.
2
Using inintial weight parameter in Perceptron Parameters:
2 Feature data set is taken. The weight corresponding to max accuracy of
 the data set is:
at iteration I: 20
TEST: # 79.7752808989 train 84.2696629213
weight_coef:
[[ 2.4733647  -1.34549739]
 [-2.88290271 -2.40860112]
 [ 1.25457498  0.99477349]]
HariKrishnas-MacBook-Pro:trial rickerish_nah$
```

Figure 6: Perceptron Learning algorithm: 2 features (Random initial weight vector).



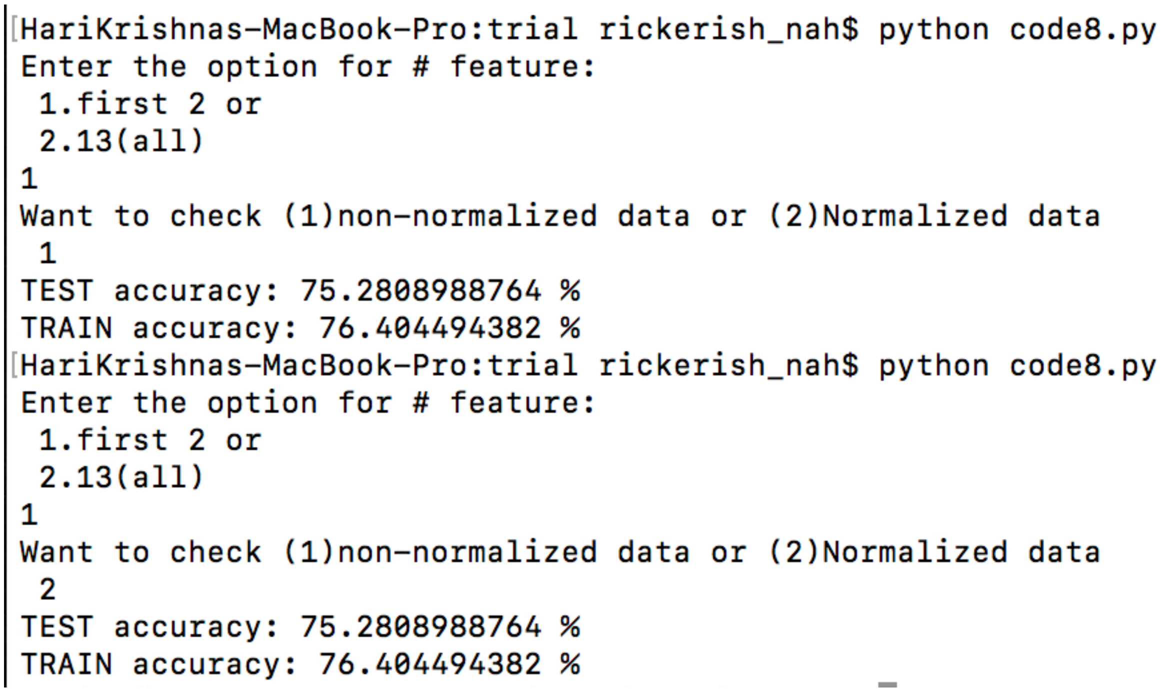
```
HariKrishnas-MacBook-Pro:trial rickerish_nah$ python code7.py
Enter the option for # feature:
 1.first 2 or
 2.13(all)
2
Select Perceptron parameters:
1. Default or
2. Include random initial weight vector.
1
/Library/Python/2.7/site-packages/sklearn/linear_model/stochastic_gradient.py
:128: FutureWarning: max_iter and tol parameters have been added in <class 's
klearn.linear_model.perceptron.Perceptron'> in 0.19. If both are left unset,
they default to max_iter=5 and tol=None. If tol is not None, max_iter default
s to max_iter=1000. From 0.21, default max_iter will be 1000, and default tol
 will be 1e-3.
  "and default tol will be 1e-3." % type(self), FutureWarning)
Using Default Perceptron Parameters:
13 Features data set has same accuracy rate for any random weight vector. So
we have taken the weight that gets max accuracy in test data set ::
TEST accuracy: # 94.3820224719 %
Train accuracy: 100.0 %
weight_coef:
[[ 4.86476371  1.15133828  3.30200511 -1.22729962  1.8644008  -0.81068027
   3.56325529 -1.46895617  0.8142626  -1.8565385  -0.4955582   3.09855528
   4.76705126]
 [-4.79991664 -4.34244909 -3.04436393  2.2071938  -3.57035669  2.14316511
   0.68315986  3.50979604 -0.55674479 -4.84794504  1.39029976  0.69251957
  -5.9952772 ]
 [ 1.16203746  2.20295434  1.71335502  0.95135987  0.81847001  0.23457438
  -2.90672792 -3.37616962 -2.5724805   4.36595908 -3.83075284 -2.45759805
   1.84638021]]
```

Figure 7: Perceptron Learning algorithm: 13 features (Default initial weight vector).



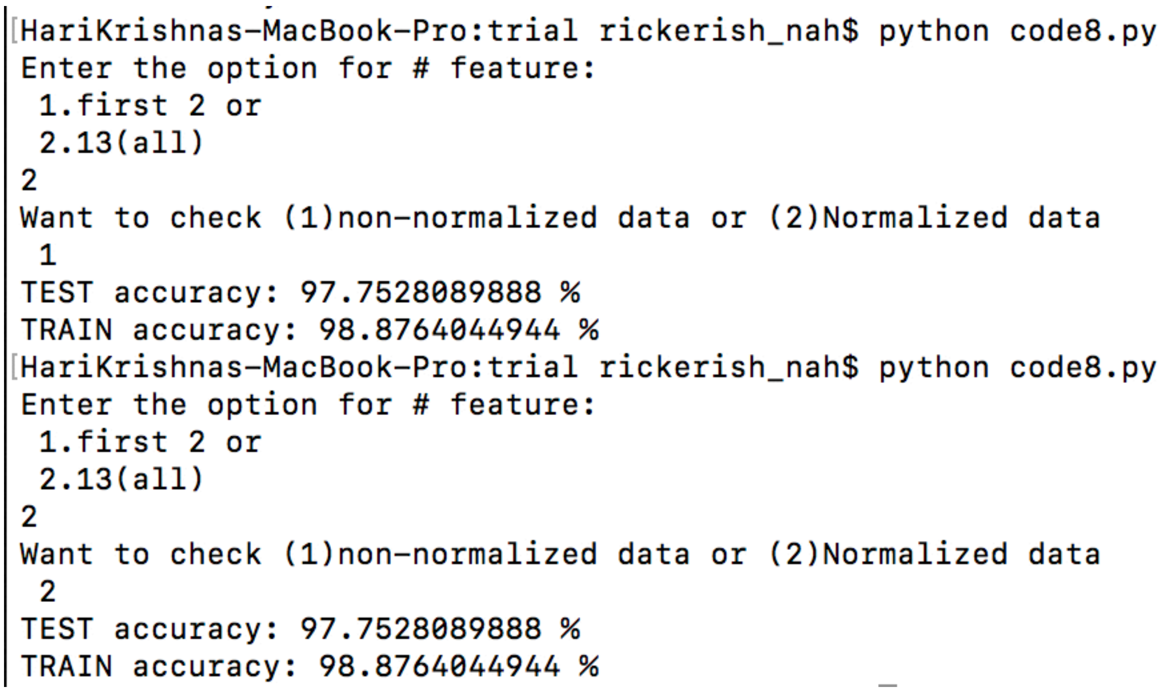
```
HariKrishnas-MacBook-Pro:trial rickerish_nah$ python code7.py
Enter the option for # feature:
 1.first 2 or
 2.13(all)
2
Select Perceptron parameters:
1. Default or
2. Include random initial weight vector.
2
Using inintial weight parameter in Perceptron Parameters:
13 Features data set has same accuracy rate for any random weight vector. So
we have taken the weight that gets max accuracy in test data set ::
TEST accuracy: # 95.5056179775 %
Train accuracy: 100.0 %
weight_coef:
[[ 2.66822764  0.91473711  3.22306141 -2.23885869  2.55110096 -0.60745501
   2.01172477 -1.20642701  0.52917897 -0.17465768  0.62394701  4.0917736
   5.62476084]
 [-5.30157274 -4.2183911  -4.40993714  3.2604561  -0.21591994  2.68475739
   0.06237231  3.91565222  0.50087136 -7.91898707  3.77950322 -1.35368493
  -7.75522175]
 [ 1.37277102  2.44827434  0.48517283  0.87780815  1.29566214 -1.36782457
  -2.27194527 -2.78661231 -1.05092493  5.27344823 -3.14196219 -3.99281967
   0.91064678]]
```

Figure 8: Perceptron Learning algorithm: 13 features (Random initial weight vector).



```
HariKrishnas-MacBook-Pro:trial rickerish_nah$ python code8.py
Enter the option for # feature:
 1.first 2 or
 2.13(all)
1
Want to check (1)non-normalized data or (2)Normalized data
 1
TEST accuracy: 75.2808988764 %
TRAIN accuracy: 76.404494382 %
HariKrishnas-MacBook-Pro:trial rickerish_nah$ python code8.py
Enter the option for # feature:
 1.first 2 or
 2.13(all)
1
Want to check (1)non-normalized data or (2)Normalized data
 2
TEST accuracy: 75.2808988764 %
TRAIN accuracy: 76.404494382 %
```

Figure 9: MSE classifier: 2 features un-normalized and normalized.



```
HariKrishnas-MacBook-Pro:trial rickerish_nah$ python code8.py
Enter the option for # feature:
 1.first 2 or
 2.13(all)
2
Want to check (1)non-normalized data or (2)Normalized data
 1
TEST accuracy: 97.7528089888 %
TRAIN accuracy: 98.8764044944 %
HariKrishnas-MacBook-Pro:trial rickerish_nah$ python code8.py
Enter the option for # feature:
 1.first 2 or
 2.13(all)
2
Want to check (1)non-normalized data or (2)Normalized data
 2
TEST accuracy: 97.7528089888 %
TRAIN accuracy: 98.8764044944 %
```

Figure 10: MSE classifier: 13 features un-normalized and normalized.

INFERENCE:

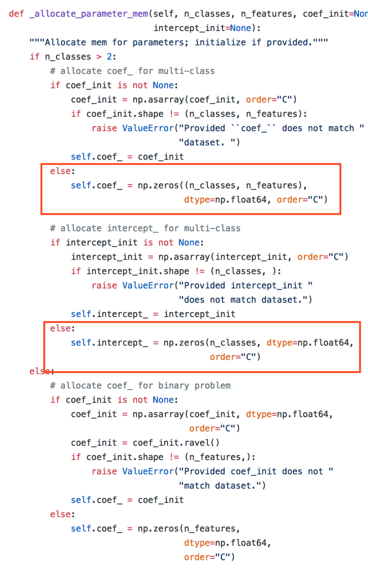
(a) Programming language used: Python
Library Used: Scikit-Learn, Numpy

(b)
The Un-normalized and the Normalized data set for both Training and the Test data is shown in the figures, Figure1, Figure2, Figure3 and Figure4.

In general, the normalization is applied only after the splitting, i.e only after we split the entire data set into training and test data. The normalization factors are calculated using the training data set only because, Normalizing the training data gives us the range and distribution within which we will be developing the classifier. In case we normalize the entire data, we are potentially leaking the features of the test data. Which in real world application is unknown. It not only deals with the leakage of information, but also it tends to give over-optimistic (not in all cases) result for our developed classifier. It is always better to split the data into training and test, normalize the training data and establish the normalizing factors and then apply these factors to the test data and see how well our classifier is developed.

(c) (i): The default initial weight vector is zero.

Perceptron imports from, `from .stochastic_gradient import BaseSGDClassifier` .



```python
def _allocate_parameter_mem(self, n_classes, n_features, coef_init=None,
                            intercept_init=None):
    """Allocate mem for parameters; initialize if provided."""
    if n_classes > 2:
        # allocate coef_ for multi-class
        if coef_init is not None:
            coef_init = np.asarray(coef_init, order="C")
            if coef_init.shape != (n_classes, n_features):
                raise ValueError("Provided ``coef_`` does not match "
                                 "dataset. ")
            self.coef_ = coef_init
        else:
            self.coef_ = np.zeros((n_classes, n_features),
                                  dtype=np.float64, order="C")

        # allocate intercept_ for multi-class
        if intercept_init is not None:
            intercept_init = np.asarray(intercept_init, order="C")
            if intercept_init.shape != (n_classes, ):
                raise ValueError("Provided intercept_init "
                                 "does not match dataset.")
            self.intercept_ = intercept_init
        else:
            self.intercept_ = np.zeros(n_classes, dtype=np.float64,
                                       order="C")
    else:
        # allocate coef_ for binary problem
        if coef_init is not None:
            coef_init = np.asarray(coef_init, dtype=np.float64,
                                   order="C")
            coef_init = coef_init.ravel()
            if coef_init.shape != (n_features,):
                raise ValueError("Provided coef_init does not "
                                 "match dataset.")
            self.coef_ = coef_init
        else:
            self.coef_ = np.zeros(n_features,
                                  dtype=np.float64,
                                  order="C")
```

Inside “Stochastic_gradient” we can see that the default initial weigh vector is zero.

(ii) Halting Condition:
In Perceptron learning algorithm, we see that the measure of error is to reduce or keep decreasing as we continue through the iterations. And we see that the measure of error in one state is lesser than the difference between the error produced in the previous state and a function 'b',

$$\mathcal{E}_{\underline{w}}(i+1) \leq \mathcal{E}_{\underline{w}}(i) - b^2$$

At one point (end where our error tends to zero) we reach a situation where the above given equation fails to comply. After that point, the classifier's operation ceases to exist. That point is called as the Halting point of the classifier. It is said that the weights have converged.

There are cases (in practical, most often) where the iteration doesn't stop and the operation takes place in a vicious manner. In those situation, we ought to introduce some ad- hoc conditions so that we can end the never-ending loop. In such cases, where the weight vector doesn't converge even after several epochs, we put a stop to it and take that weight vector over the entire iteration that produced the least classification error.

(d)

Perceptron learning algorithm is applied to both 2-features data set and all 13-features data set, and the corresponding ***resultant weight vector and classification accuracy of the classifier*** on both TRAIN and TEST data set is shown in the figures, ***Figure 5 and Figure 7***.

(e)

Perceptron learning algorithm is applied to both 2-features data set and all 13-features data set, where a random initial weight vector is loaded into the system (100 trials) and the corresponding ***resultant weight vector and classification accuracy of the classifier*** on both TRAIN and TEST data set is shown in the figures, ***Figure 6 and Figure 8***.

(f)

	TRAIN: Default	TRAIN: Random Initialization	TEST: Default	TEST: Random Initialization
2 Features	73	84	73	80
13 Features	100	100	94	95

From the above results, it is very evident that, a data set with more features shows better classification accuracy. May it be default initial weight vector or a random initialized weight vector, the accuracy rates are high for 13 feature data set. This explains that, more are the features present, complex is the learning but better is the developed classifier. This is the effect of dimensionality. As the number of dimensions increases, better is the classifier fit. (There are cases where a high dimension data set will over fit the training data but won't for the test)
In the non-default section, the changes made are,

- max_iter = 10000 (default is much downsized)

- weight vector = random at each iteration.

We see that in default initialization, the accuracy rate is same over any number of repeated iterations. But when we initialize a random vector at each iteration, we get different accuracy rates. For example, the accuracy rate for 13-features test data set varies from 91 to 98% depending on the initial random weight vector.

NOTE: The weight vector that I've implemented has its value varying in the range [0,1).

(g)

The Pseudoinverse Classifier or the MSE classifier learning algorithm is applied to both 2-features and 13-features data set. (Here the data set used is ***un-normalized/raw data***). The classification accuracy rate is reporter in Figure 9 and Figure 10 un-normalized part.

For 2 features data set,	For 13 features data set,
Training accuracy rate = 76%	Training accuracy rate = 99
Test accuracy rate = 75%	Test accuracy rate = 98

(h)

The Pseudoinverse Classifier or the MSE classifier learning algorithm is applied to both 2-features and 13-features data set. (Here the data set used is ***normalized/raw data***). The classification accuracy rate is reporter in Figure 9 and Figure 10 un-normalized part.

For 2 features data set,	For 13 features data set,
Training accuracy rate = 76%	Training accuracy rate = 99
Test accuracy rate = 75%	Test accuracy rate = 98

(i)

In developing a learning classifier, data normalization is not a mandatory step.

Data normalization is done because of the following reasons,

(1) Through data normalization we can interpret the data in an easy manner.

(2) In case we have data set involving large numbers, chances for numerical problems to occur is high. Example, manipulating with large number is a very big head-ache.

(3) Moreover, if the given data is not of uniform scale or bounded, then performing calculation on it becomes tedious.

Hence, we do normalization of data.

Here in our MSE case we see that the normalization of data doesn't bring about any change in the accuracy rates. i.e. the classification accuracy rates are same for both un-normalized and normalized data. This is because, MSE uses least square's regression method.

Let 'X' be the data set and 'A' be a matrix that can normalize X.

i.e X- un-normalized data set and XA- normalized data set.

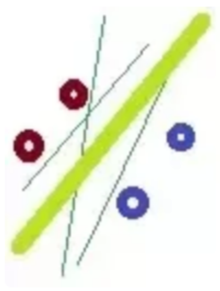
In $X b = y$, $b = [(X^T X)^{-1} X^T] y$ ....(1)

In $XA\, b^\wedge = y$, $b^\wedge = [(A^T X^T XA)^{-1} (XA)^T] y$ ....(2) $= A^{-1} y$

In both the cases we arrive at the same predictions, therefore using raw data or normalized data doesn't make a difference in MSE Classifiers/ Pseudoinverse Classifiers.

(j)

The difference between Perceptron and MSE is that, perceptron's default condition is One Vs Rest (we can override it using explicit citing of the classifier) and it can take more than 2 classes on its own, but the MSE which is based on linear regression, is a binary classifier and needs an explicit command structure to implement One vs Rest, One vs All etc. Perceptron is a classifier and MSE's is a linear regression operator mimicking as a classifier. Figuratively we can see that the difference between perceptron and MSE can be shown as



where the thin green lines refer to the possible perceptron generated boundaries and the thick green line refers to the MSE generated boundary.

The accuracy of the classifier in terms of training the classifier through training data says that perceptron is a better classifier as it has constantly kept 100% accuracy rate over 98-99% of MSE's. When it comes to classification of test data, they perform equally good as they have a classification accuracy of over 95%, with MSE having an edge over perceptron. Key area of usage of perceptron is in Machine learning and that of MSE's is Data mining.