STA 6106 : Project 2
Report By: Anmol Sureshkumar Panchal

Problem 1:-

a)

First import the dataset using where we are importing x1,x2,x3 and x4.

```
df = pd.read_csv(r"C:\Users\anmol\Downloads\charlie.csv")
df = df[['x1', 'x2', 'x3', 'x4']]
X = np.array(df)
print(type(df)
```

Declaring data to generate test data with first 4 rows :

```
x_test = X[0:4,:]
print(x_test)
```

Now we have to check the shape of rows and columns of the test data so that when we perform sequential updates we make sure (-1) we insert at the end of last row of last update.

```
n_rows, n_cols = x_test.shape
print(n_rows,n_cols)
updates = X[4:-1,:]
```

Now we will define function to calculate parameters c,s from givens (https://en.wikipedia.org/wiki/Givens_rotation) . Givens rotation is used to yield an upper triangular matrix in order to compute the QR decomposition which returns c,s.

```
def Givens(a, b):
    if b == 0:
        c = 1
        s = 0
    elif np.abs(b) >= np.abs(a):
        t = -a/b
        s = 1/np.sqrt(1 + t**2)
        c = s*t
    else:
        t = -b/a
        c = 1/np.sqrt(1 + t**2)
        s = c*t
    return c, s
```

Now we define update rows function to update rows sequentially using algorithm from qr_update file from webcourses:

```
def update_rows(Q, R, u, data):
    next_row = u
    n_rows, n_cols = data.shape
    for j in range(n_cols):
        c, s  = Givens(R[j,j], next_row[j])
        R[j, j] = c*R[j, j] - s*next_row[j]
        #Update jth row of R and u
        t1 = R[j, j+1:n_cols]
        t2 = next_row[j + 1:n_cols]
        R[j, j+1:n_cols] = c*t1 - s*t2
        next_row[j+1:n_cols] = s*t1 + c*t2
    R_up = np.zeros((len(data)+1, len(data)+1))
    R_up[0:len(data), 0:len(data)] = R
#     R_up.shape

    m, n = Q.shape
    Q_up = np.zeros((m+1, m+1))
    Q_up[-1][-1] = 1
    Q_up[0:m, 0:m] = Q
    for j in range(n_cols):
        c, s = Givens(R[j, j], next_row[j])
        t1 = Q_up[0:m+1, j]
        t2 = Q_up[0:m+1, m]
        Q_up[0:m+1, j] = c*t1 - s*t2
        Q_up[0:m+1, m] = s*t1 + c*t2
    Q_up.shape

    data = np.vstack([data, np.zeros(n_cols)])
    data[-1] = u

#     A = np.delete(A, -1, 1)
#     print(A - data)
    print("Q update :", Q_up)
    print("R update: ", R_up)
#     print(np.dot(Q_up, R_up) - data)
    return Q_up, R_up

for i in range(4, len(X)):
    Q_up, R_up = update_rows(Q_up, R_up, X[i], x_test)
    A = np.dot(Q_up, R_up)
    r, c = np.shape(A)
    while c != nc:
        A = np.delete(A, -1, 1)
        c = c-1
```

```
c = nc
print("A: ",A)
x_test = np.vstack([x_test, np.zeros(nc)])
x_test[-1] = df.iloc[i,:]
print("X_test = ", x_test, "\n")
```

This above update function gives Q_up,R_up which are frequent sequential QR updates of the rows inserted. **np.linalg.qr** is used to calculate Q,R and perform QR decomposition.

```
Q update : [[-0.13802532  0.15613408  0.07206437 ... -0.04385967 -0.08160296
   0.08907631]
 [-0.14492658 -0.3690783   0.16227286 ... -0.04349659  0.07879631
  -0.18767628]
 [-0.13388456  0.12850264 -0.18449905 ...  0.02404815 -0.0716289
   0.05083207]
 ...
 [-0.15827532  0.02844674  0.2314337  ...  0.87250926 -0.03380239
   0.02464166]
 [-0.11908655  0.20262748  0.04429196 ...  0.          0.86198827
   0.11137467]
 [-0.14173917 -0.51849484 -0.073928   ...  0.          0.
   0.51868778]]
R update:  [[ -81.71260613 -153.76288103 -104.03053437 ...    0.
     0.           0.        ]
 [   0.          36.27032779   18.14377352 ...    0.
     0.           0.        ]
 [   0.           0.         -26.05287269 ...    0.
     0.           0.        ]
```

This gives us the test data (x_test) from sequential row updates using qr decomposition of the data each time rows are updated sequentially.

```
X_test =  [[10.  20.7 13.6 15.5]
 [10.5 19.9 18.1 14.8]
 [ 9.7 20.  16.1 16.5]
 [ 9.8 20.2 19.1 17.1]
 [11.7 21.5 19.8 18.3]
 [11.  20.9 10.3 13.8]
 [ 8.7 18.8 16.9 16.8]
 [ 9.5 19.3 15.3 12.2]
 [10.1 19.4 16.2 15.8]
 [ 9.5 19.6 13.6 14.5]
 [10.5 20.3 17.  16.5]
 [ 9.2 19.  11.5 16.3]
 [11.3 21.6 14.  18.7]
 [10.  19.8 14.  15.9]
 [ 8.5 19.2 17.4 15.8]
 [ 9.7 20.1 10.  16.6]
 [ 8.3 18.4 12.5 14.2]
 [11.9 21.8 14.1 16.2]
 [10.3 20.5 15.6 15.1]
```

b)

First we will import our datasets to use x1,x2,x3,x4 columns and perform qr decomposition with first 2 columns and sequentially adding next 2 columns.

```
dataset = pd.read_csv(r"C:\Users\anmol\Downloads\charlie.csv")
x1 = np.array(dataset.x1)
x2 = np.array(dataset.x2)
x3 = np.array(dataset.x3)
x4 = np.array(dataset.x4)
dataset = np.array(dataset)
```

We are defining a function to calculate qr decomposition of columns where each column is a input to our function as a vector whose orthogonal vectors q1,q2… are found by this function.

First orthogonalize each vector w.r.t. previous ones; then normalize result to have norm one using**(np.linalg.norm).** The algorithm implemented here works on the logic of algorithm given in qr_update file and which was more similar to Gram Schmidt process given here(http://ee263.stanford.edu/lectures/qr.pdf).

```
def gs_column_update(col_vector):
    update_array = []
    for v in col_vector:
        w = v - np.sum( np.dot(v,b)*b  for b in update_array )
        if (w > 1e-10).any():
            update_array.append(w/np.linalg.norm(w))
#     return linalg.qr(np.array(update_array))
    return linalg.qr(np.cov(np.array(update_array)))
```

You can remove **np.cov** from above line if you dont want the covariance matrix of the set obtained with sequential column updates and comment it and use this instead:
return linalg.qr(np.array(update_array))
To just perform QR decomposition without covariance matrix.

Now we will see the output when we calculate QR decomposition having first two columns only.

```
gs_column_update([x1,x2])
# Output genearted with np.cov
```

```
C:\Users\anmol\Anaconda3\lib\site-packages\ipykernel_launcher.py:4: DeprecationWarning: Calling np.sum(generator) is deprec
ated, and in the future will give a different result. Use np.sum(np.from_iter(generator)) or the python sum builtin instea
d.
  after removing the cwd from sys.path.
```

```
(array([[-0.21603084,  0.97638654],
        [ 0.97638654,  0.21603084]]), array([[-0.00437793,  0.03405998],
        [ 0.        ,  0.00315802]]))
```

Then we are passing third column as third vector:

```
gs_column_update([x1,x2,x3])
# Output genearted with np.cov
```

```
C:\Users\anmol\Anaconda3\lib\site-packages\ipykernel_launcher.py:4: DeprecationWarning: Calling np.sum(generator) is deprec
ated, and in the future will give a different result. Use np.sum(np.from_iter(generator)) or the python sum builtin instea
d.
  after removing the cwd from sys.path.
```

```
(array([[-0.19907418, -0.14160026,  0.96970038],
        [ 0.89974819, -0.4185418 ,  0.12359592],
        [ 0.38835893,  0.89709092,  0.21072545]]),
 array([[-0.00475083,  0.03129522,  0.01350798],
        [ 0.        , -0.01381013,  0.03120279],
        [ 0.        ,  0.        ,  0.00542682]]))
```

Now passing final x4 column as vector:

```
gs_column_update([x1,x2,x3,x4])
# Output genearted with np.cov
```

```
C:\Users\anmol\Anaconda3\lib\site-packages\ipykernel_launcher.py:4: DeprecationWarning: Calling np.sum(generator) is deprec
ated, and in the future will give a different result. Use np.sum(np.from_iter(generator)) or the python sum builtin instea
d.
  after removing the cwd from sys.path.
```

```
(array([[-0.18128004, -0.04561866, -0.04675425,  0.98125966],
        [ 0.81932467, -0.55949018, -0.0059592 ,  0.12506924],
        [ 0.35364567,  0.53807995, -0.76321063,  0.05398372],
        [ 0.41325461,  0.6287763 ,  0.64442848,  0.1362825 ]]),
 array([[-0.00521717,  0.02838435,  0.01225156,  0.01431662],
        [ 0.        , -0.01909227,  0.01864102,  0.02178307],
        [ 0.        ,  0.        , -0.02622891,  0.02232532],
        [ 0.        ,  0.        ,  0.        ,  0.00252412]]))
```

Note: All this values are calculated using covariance matrix of columns using np.cov, if we don't use np.cov results will vary.

c & d)

Now moving to the part we have to fit data for classification by help of calculating H's and alpha's.

Here we have to use row update thing from first part so our X will be x_test data and Data column will be as y. We are using C = 10, sigma = 0.001 as input for C and sigma for our kernel function which is used to calculate H matrix and alpha values.

```
data = pd.read_csv(r"C:\Users\anmol\Downloads\charlie1.csv")
X = x_test
y = data['Data']
X = np.array(X)
y = np.array(y)
C,sigma = 10,0.001
```

We are going to define a gaussian kernel function as given below:

```
def Kernel(x, y, sigma):
    return np.exp(-np.linalg.norm(x-y)**2 / ( (sigma ** 2)))
```

Then calculate Gram Matrix which will be required to calculate H matrix :

```
def Gram_Matrix(x):
    K = np.zeros((len(x),len(x)))
    for i in range(0, len(x)):
        for j in range(0, len(x)):
            K[i, j] = Kernel(x[i], x[j], sigma)

    return K


def H(x):
    mat = np.zeros((len(x), len(x)))
    mat[0:len(x), 0:len(x)] = Gram_Matrix(x) + np.eye(len(x))/2*C
    return mat
```

Above we defined H function as given in question and below e_func is a vector of one's of size of x_test data.

```
def e_func(x):
    return np.ones(len(x))
def k_func(x):
    k = np.zeros(len(x_test))
```

```
    for j in range(0, len(x)):
        k[j] = Kernel(x[j], x[j], sigma)
    return k

e = e_func(x_test)
k = k_func(x_test)
```

Now we will calculate alpha matrix which has all alpha values corresponding to each H matrix.

```
def alpha(H_mat):
    p1 = np.dot(np.dot(np.linalg.inv(H_mat), e.T),k)
    p2 = np.dot(np.dot(np.linalg.inv(H_mat), e.T), e)
    p3 = (2-p1)/p2
    p3 = k + np.dot(p3, e)
    a = 0.5*np.dot(np.linalg.inv(H_mat),p3)
    return a
```

You can check first 4 rows getting updated as below which return H4 and alpha(1,2,3,4), with their qr decomposition.

```
x_test = X[0:4,:]
x_test
H_mat = H(x_test)
al = alpha(H_mat)
```

```
In [375]: H_mat

Out[375]: array([[6., 0., 0., 0.],
                 [0., 6., 0., 0.],
                 [0., 0., 6., 0.],
                 [0., 0., 0., 6.]])
```

```
In [376]: al

Out[376]: array([0.25, 0.25, 0.25, 0.25])
```

```
In [377]: Q_H, R_H = np.linalg.qr(H_mat)
```

```
In [378]: Q_H

Out[378]: array([[ 1.,  0.,  0.,  0.],
                 [-0.,  1.,  0.,  0.],
                 [-0., -0.,  1.,  0.],
                 [-0., -0., -0.,  1.]])
```

```
In [379]: R_H

Out[379]: array([[6., 0., 0., 0.],
                 [0., 6., 0., 0.],
                 [0., 0., 6., 0.],
                 [0., 0., 0., 6.]])
```

Similarly you can see H1,H2,H3….Hn matrix where each time you go from i to i+1 till n you will one row and one column is getting added. This can be observed from the screenshot below:

```
H(X[0:2,:])
```

```
array([[6., 0.],
       [0., 6.]])
```

```
H(X[0:3,:])
```

```
array([[6., 0., 0.],
       [0., 6., 0.],
       [0., 0., 6.]])
```

```
alpha()
```

```
array([0.02671756, 0.02671756, 0.02671756, 0.02671756, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038, 0.01145038, 0.01145038, 0.01145038,
       0.01145038, 0.01145038])
```

Now we need to calculate radius function return radius value which will be used to classify data for anomalies.

```
def R_square(al, X):
    p1 = 0
    p2 = 0
    total = 0
    for s in range(0, len(X)):
        k = Kernel(X[s], X[s], sigma)
        for j in range(0, len(X)):
            p1 = p1 + al[j]*Kernel(X[s], X[j], sigma)
          for l in range(0, len(X)):
```

```
            p2 = p2 + al[j]*al[l]*Kernel(X[j], X[l], sigma)
        total = total + (k - 2 * p1 + p2)
    final = total/len(X)
    return final

final = R_square(al, x_test)
```

```
final
```

```
0.375
```

So our **final** is the R - radius distance obtained from x_test data.
Now we will write a classification function and fit this test data and calculate classification rate. Our data classification depends on if x_test data has a distance less than radius - final if not then it will be considered as an outlier.

```
def classification(x):
    t_out = []
    t_in = []
    p = 0
    p1 = 0
    for z in range(0, len(x)):
        k = Kernel(x[z], x[z], sigma)
        for j in range(0, len(X)):
            p = p + al[j]*Kernel(x, X[j], sigma)
            for l in range(0, len(X)):
                p1 = p1 + al[j]*al[l]*Kernel(X[j], X[l], sigma)
        d = k - 2*p + p1
        if d <= final:
            t_in.append(x[z])
        else:
            t_out.append(x[z])

    return t_out, t_in

t_out, t_in = classification(X)
```

```
t_out,t_in
```

```
([array([10. , 20.7, 13.6, 15.5]),
  array([10.5, 19.9, 18.1, 14.8]),
  array([ 9.7, 20. , 16.1, 16.5]),
  array([ 9.8, 20.2, 19.1, 17.1]),
  array([11.7, 21.5, 19.8, 18.3]),
  array([11. , 20.9, 10.3, 13.8]),
  array([ 8.7, 18.8, 16.9, 16.8]),
  array([ 9.5, 19.3, 15.3, 12.2]),
  array([10.1, 19.4, 16.2, 15.8]),
  array([ 9.5, 19.6, 13.6, 14.5]),
  array([10.5, 20.3, 17. , 16.5]),
  array([ 9.2, 19. , 11.5, 16.3]),
  array([11.3, 21.6, 14. , 18.7]),
  array([10. , 19.8, 14. , 15.9]),
  array([ 8.5, 19.2, 17.4, 15.8]),
  array([ 9.7, 20.1, 10. , 16.6]),
  array([ 8.3, 18.4, 12.5, 14.2]),
  array([11.9, 21.8, 14.1, 16.2]),
  array([10.3, 20.5, 15.6, 15.1]),
  array([ 8.9, 19. ,  8.5, 14.7]),
  array([ 9.9, 20. , 15.4, 15.9]),
  array([ 8.7, 19. ,  9.9, 16.8]),
  array([11.5, 21.8, 19.3, 12.1]),
  array([15.9, 24.6, 14.7, 15.3]),
  array([12.6, 23.9, 17.1, 14.2]),
  array([14.9, 25. , 16.3, 16.6]),
```

Then we fit our data using sklearn svm and calculate number of error(or outliers) which came out to be 4 out of every 10 data points.

```
import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn import svm

clf = svm.OneClassSVM(kernel = 'rbf', gamma = 'auto')
clf.fit(t_out,t_in)
```

```
OneClassSVM(cache_size=200, coef0=0.0, degree=3, gamma='auto', kernel='rbf',
      max_iter=-1, nu=0.5, random_state=None, shrinking=True, tol=0.001,
      verbose=False)
```

```
n_error_outliers = t_out[t_out == 1].size
print("Number of errors = ",n_error_outliers,"/",y_out.size)
#classification rate
rate = n_error_outliers/y_out.size
print("Classification rate = ",100*(1-rate),"%")
```

```
Number of errors =  4 / 10
Classification rate =  60.0 %
```

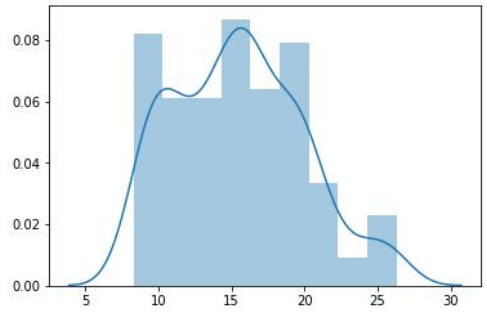
Then we converted our t_out data in dataframe as shown below so that we can use seaborn to plot the data.

plot_data = pd.DataFrame(np.array(t_out).reshape(328,))

```
sns.distplot(plot_data)
```

```
C:\Users\anmol\Anaconda3\lib\site-packages\scipy\stats\stats.py:1713: FutureWarning: Using a non-tuple sequence for multidi
mensional indexing is deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2224cc86828>
```



Problem 2:

Now for Problem 2 we will require some dataset to check and select our best parameter value for C and gamma. So lets load the digit dataset from python library itself.

Loading the digit data:

```
digits = datasets.load_digits()
```

Viewing the features of the first observation

```
digits.data[0:1]
```

Viewing the target of the first observation

```
digits.target[0:1]
```

Now creating dataset 1 and 2:

```
# Create dataset 1
data1_features = digits.data[:1000]
data1_target = digits.target[:1000]

# Create dataset 2
data2_features = digits.data[1000:]
data2_target = digits.target[1000:]
```

Now set possible parameter values on which we will perform Grid Search function to select the best parametric value of C and gamma.

```
parameter_candidates = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001,0.1,1,0.00001], 'kernel': ['rbf','linear']},
]
```

Now we will Create a classifier object with the classifier and parameter candidates

```
clf = GridSearchCV(estimator=svm.SVC(), param_grid=parameter_candidates, n_jobs=-1)
```

Then we will need to Train the classifier on data1's feature and target data

```
clf.fit(data1_features, data1_target)
```

```
GridSearchCV(cv='warn', error_score='raise-deprecating',
       estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid=[{'C': [1, 10, 100, 1000], 'kernel': ['linear']}, {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001, 0.1,
1, 1e-05], 'kernel': ['rbf', 'linear']}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

Now we need to see the accuracy score

```
print('Best score for data1:', clf.best_score_)
```

>>>Best score for data1: 0.942

Seeing the best parameters for the model found using grid search functions **(clf.best_estimator.(parameter_to_check)):**

```
print('Best C:',clf.best_estimator_.C)
print('Best Kernel:',clf.best_estimator_.kernel)
print('Best Gamma:',clf.best_estimator_.gamma)
```

```
Best C: 10
Best Kernel: rbf
Best Gamma: 0.001
```

We can also verify these results with different program implemented here: (http://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html)

Which gives output as:

Automatically created module for IPython interactive environment

Tuning hyper-parameters for precision

Best parameters set found on development set:

{'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}

Grid scores on development set:

0.986 (+/-0.016) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.959 (+/-0.029) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.026) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.025) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.025) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.975 (+/-0.014) for {'C': 1, 'kernel': 'linear'}
0.975 (+/-0.014) for {'C': 10, 'kernel': 'linear'}
0.975 (+/-0.014) for {'C': 100, 'kernel': 'linear'}
0.975 (+/-0.014) for {'C': 1000, 'kernel': 'linear'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	89
1	0.97	1.00	0.98	90
2	0.99	0.98	0.98	92
3	1.00	0.99	0.99	93
4	1.00	1.00	1.00	76
5	0.99	0.98	0.99	108
6	0.99	1.00	0.99	89
7	0.99	1.00	0.99	78
8	1.00	0.98	0.99	92
9	0.99	0.99	0.99	92
micro avg	0.99	0.99	0.99	899
macro avg	0.99	0.99	0.99	899
weighted avg	0.99	0.99	0.99	899

Tuning hyper-parameters for recall

Best parameters set found on development set:

{'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}

Grid scores on development set:

0.986 (+/-0.019) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.957 (+/-0.029) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.981 (+/-0.028) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.981 (+/-0.026) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.981 (+/-0.026) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.972 (+/-0.012) for {'C': 1, 'kernel': 'linear'}
0.972 (+/-0.012) for {'C': 10, 'kernel': 'linear'}
0.972 (+/-0.012) for {'C': 100, 'kernel': 'linear'}
0.972 (+/-0.012) for {'C': 1000, 'kernel': 'linear'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	89
1	0.97	1.00	0.98	90
2	0.99	0.98	0.98	92
3	1.00	0.99	0.99	93
4	1.00	1.00	1.00	76
5	0.99	0.98	0.99	108
6	0.99	1.00	0.99	89
7	0.99	1.00	0.99	78
8	1.00	0.98	0.99	92
9	0.99	0.99	0.99	92
micro avg	0.99	0.99	0.99	899
macro avg	0.99	0.99	0.99	899
weighted avg	0.99	0.99	0.99	899