Lab:11-Hyper-Parameter Optimization with PCA (Wine Partial) dht258

November 24, 2020

1 Lab: Hyper-Parameter Optimization with PCA

PCA is often applied as a pre-processing step with classifiers. When using PCA in this manner, one must select the number of PC components to use along with parameters in classifier. In this lab, we will demonstrate how to performing this *hyper-parameter optimization*. In doing the lab, you will learn to:

- Combine PCA with data scaling.
- Compute and visualize PC components
- Select the number of PCs with K-fold cross validation
- Implement the multi-stage classifier pipeline in sklearn
- Perform automatic parameter search using `GridSearchCV` in combination with a pipeline.

We first download the basic packages.

[1]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

1.1 Downloading the Data

We will use a very simple `wine` dataset, commonly used in teaching machine learning class. The problem is to classify the type of red wine from features of the wine such as the `alchohol` and other chemical components. There are three possible wine types.

[2]:
```
from sklearn.datasets import load_wine
from sklearn.model_selection import KFold
data = load_wine()

# TODO print the features names in data.feature_names and data.target_names

print('Features Names' , data.feature_names)
print('Target names', data.target_names)
```

```
Features Names ['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash',
'magnesium', 'total_phenols', 'flavanoids', 'nonflavanoid_phenols',
'proanthocyanins', 'color_intensity', 'hue', 'od280/od315_of_diluted_wines',
```

```
'proline']
Target names ['class_0' 'class_1' 'class_2']
```

Get the data matrix X from data.data and the target values y from data.target. Print the number of samples, number of features and number of classes.

[3]:
```
# TODO
X = data.data
y = data.target
print('Number_Samples: ',X.shape[0])
print('Number_Features: ',X.shape[1])
print('Number_Classes: ',len(np.unique(y)))
```

```
Number_Samples:  178
Number_Features:  13
Number_Classes:  3
```

1.2 Perform PCA for Visualization

Before performing PCA, you should scale the data matrix to remove the mean and normalize the variance of the different components. For this purpose, create a StandardScaling object scaling. Then fit the scaling with the entire data X. Transform the data and let Xs be the scaled data.

[4]:
```
from sklearn.preprocessing import StandardScaler

# TODO
scaling = StandardScaler(with_mean = True , with_std = True)
scaling.fit(X)
Xs = scaling.transform(X)
```

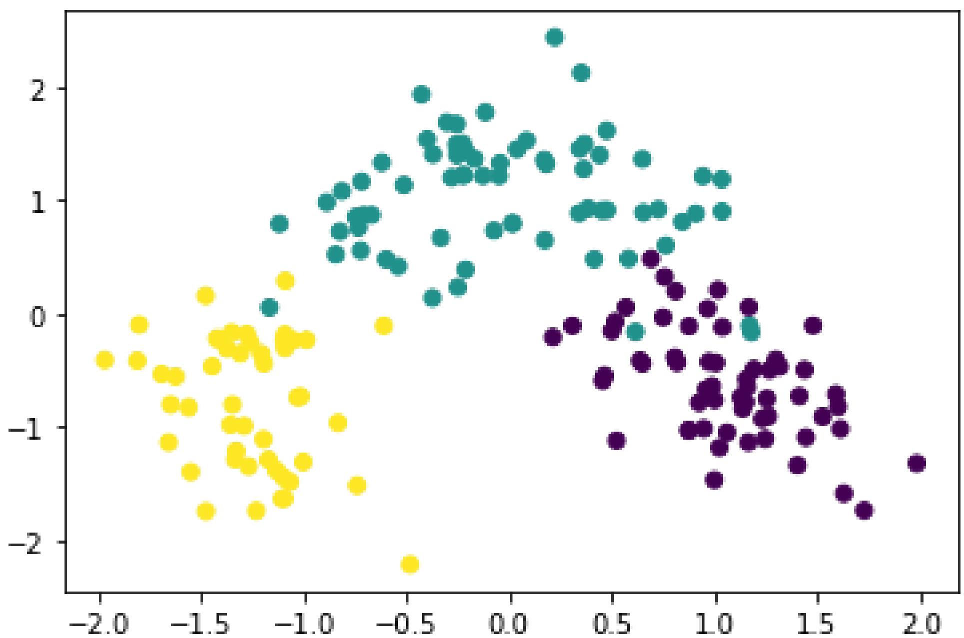
Now, fit a PCA on the scaled data matrix Xs. You can use the sklearn PCA method. In order that we can visualize the results set n_components=2. Select svd_solver='randomized' and whiten=True. Use the the pca.transform method to find, Z, the coefficients of Xs in the PCA basis.

[5]:
```
from sklearn.decomposition import PCA
# TODO
# Construct the PCA object
ncomp = 2
pca = PCA(n_components = ncomp , svd_solver = 'randomized' , whiten = True)
pca.fit(Xs)
Z = pca.transform(Xs)
```

In the transformed basis, each data sample is represented by a two dimensional vector, Z[i,0], Z[i,1]. Plot a scatter plot of the transformed data. Use different marker colors for the different classes. If you did everything, you should see that the classes are quite well separated with even two PCs.

[6]:
```
# TODO
plt.scatter(Z[:,0],Z[:,1],c = y)
```

[6]: <matplotlib.collections.PathCollection at 0x241b91b51c8>



Now, refit the scaled data `Xs` using `n_components=nfeatures` where `nfeatures` is the number of features. This is the maximum number of PCs. Get the singular values from `pca.singular_values_` and plot the portion of variation as a function of the number of PCs. The PoV for using `n` PCs is:

```
PoV[n]  = \sum_{i=0}^{n-1}  s[i]**2 / \sum_{i=0}^{d-1}  s[i]**2
```

where `s[i]` is the `i`-th singular value and `d` is the number of features. You should see that the 4 PCs contains more than 70% of the variance.

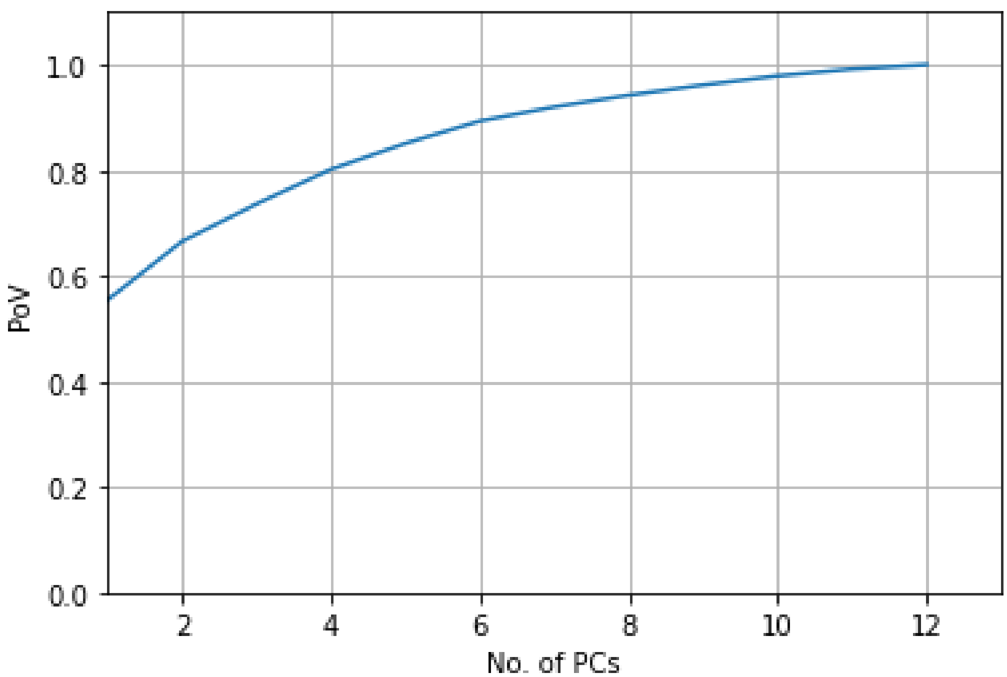
[8]:
```
# TODO
numfeatures=13
pca_n = PCA(n_components=numfeatures, svd_solver='randomized', whiten=True)
pca_n.fit(Xs)
Z_n = pca_n.transform(Xs)
s = pca_n.singular_values_
d = numfeatures
PoV = []
sum_all = np.sum(s**2)

for n in range(1,d+1,1):
    PoV1 = np.sum(s[:n]**2)/sum_all
    PoV.append(PoV1)
    print(n,PoV1)
plt.plot(PoV)
```

```
plt.grid()
plt.axis([1,d,0,1.1])
plt.xlabel('No. of PCs')
plt.ylabel('PoV')
```

```
1 0.3619884809992635
2 0.5540633835693531
3 0.6652996889318524
4 0.7359899907589929
5 0.8016229275554787
6 0.8509811607477042
7 0.8933679539739373
8 0.9201754434577261
9 0.9423969775056232
10 0.961697168445064
11 0.9790655253449633
12 0.9920478511010055
13 1.0
```

[8]: Text(0, 0.5, 'PoV')



1.3 Using PCA with Classification

We will now use data scaling and PCA as a pre-processing step for logistic classification. The number of PCs to use can be found with cross-validation. Complete the code below which tries different number of PCs components to use and measures the test accuracy for each value.

```
[9]: from sklearn.preprocessing import StandardScaler
     from sklearn.linear_model import LogisticRegression
     nfold = 5

     # Create a K-fold object
     kf = KFold(n_splits=nfold)
     kf.get_n_splits(X)

     # Number of PCs to try
     ncomp_test = np.arange(2,12)
     num_nc = len(ncomp_test)

     # Accuracy:  acc[icomp,ifold]  is test accuracy when using `ncomp =␣
      ↪ncomp_test[icomp]` in fold `ifold`.
     acc = np.zeros((num_nc,nfold))

     # Loop over number of components to test
     for icomp, ncomp in enumerate(ncomp_test):

         # Look over the folds
         for ifold, I in enumerate(kf.split(X)):
             Itr, Its = I

             # TODO: Split data into training
             Xtr, Xts, ytr, yts = X[Itr] , X[Its] , y[Itr] , y[Its]


             # TODO: Create a scaling object and fit the scaling on the training data
             scale2 = StandardScaler()
             scale2.fit(Xtr)
             Xtr_fitted = scale2.transform(Xtr)
             # print('Xtr Fitted', Xtr_fitted)


             # TODO: Fit the PCA on the scaled training data
             pca = PCA(n_components = ncomp , svd_solver = 'randomized' , whiten =␣
      ↪True)
             pca.fit(Xtr_fitted)
             Z = pca.transform(Xtr_fitted)


             # TODO: Train a classifier on the transformed training data
             # Use a logistic regression classifier
             logreg = LogisticRegression(multi_class='ovr', solver='lbfgs')
             logreg.fit(Z,ytr)
```

```
        # TODO: Transform the test data through data scaler and PCA
        Xts_fitted = scale2.transform(Xts)
        Zs = pca.transform(Xts_fitted)


        # TODO: Predict the labels the test data
        yhat = logreg.predict(Zs)


        # TODO: Measure the accuracy
        acc[icomp, ifold] = np.mean(yhat == yts)
```

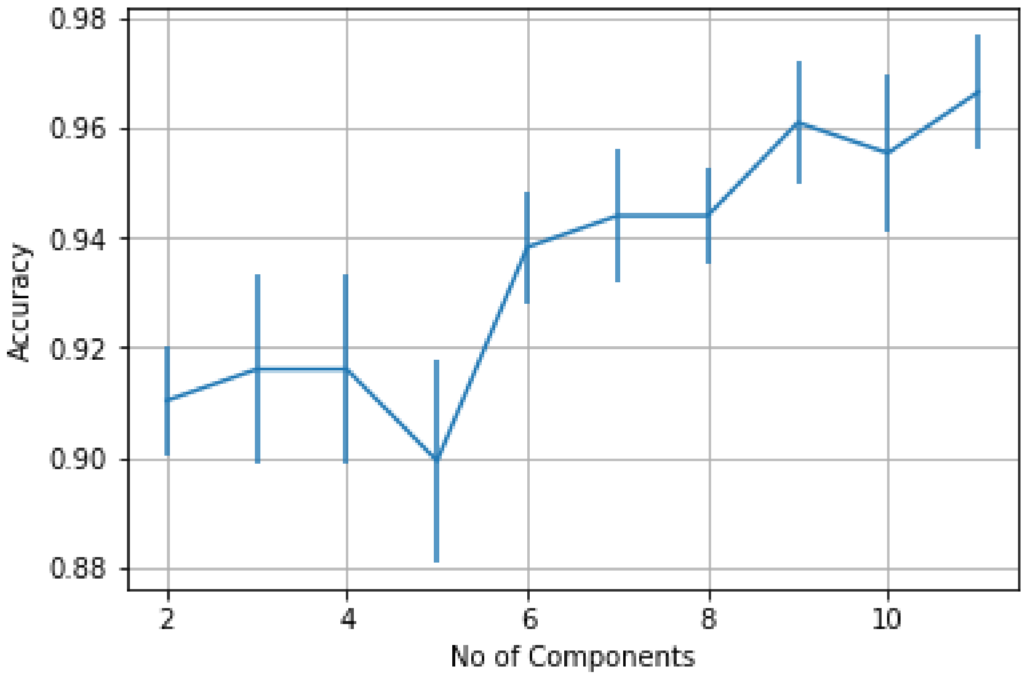
Use the `plt.errorbar` function to plot the mean accuracy with error bars corresponding to the standard error of the accuracy as a function of the number of components. Find the optimal number of PCs to use according to the normal rule and one SE rule. If you did it correctly, you should get an accuracy of around 96%.

[10]:
```
# TODO:
print(acc)
print(acc.shape)

acc_mean = np.mean(acc,axis=1)
print('Accuracy mean: ',acc_mean)
acc_se = np.std(acc,axis=1)/np.sqrt(nfold-1)
print('Accuracy standard error: ', acc_se)
plt.errorbar(ncomp_test, acc_mean, yerr = acc_se, fmt='-')
plt.xlabel('No of Components')
plt.ylabel('Accuracy')
plt.grid()
```

```
[[0.88888889 0.91666667 0.88888889 0.91428571 0.94285714]
 [0.86111111 0.94444444 0.88888889 0.94285714 0.94285714]
 [0.86111111 0.94444444 0.88888889 0.94285714 0.94285714]
 [0.86111111 0.86111111 0.88888889 0.94285714 0.94285714]
 [0.97222222 0.91666667 0.91666667 0.94285714 0.94285714]
 [0.97222222 0.91666667 0.91666667 0.94285714 0.97142857]
 [0.94444444 0.94444444 0.91666667 0.94285714 0.97142857]
 [0.94444444 0.94444444 0.94444444 0.97142857 1.        ]
 [0.94444444 0.91666667 0.94444444 0.97142857 1.        ]
 [0.97222222 0.94444444 0.94444444 0.97142857 1.        ]]
(10, 5)
Accuracy mean:  [0.91031746 0.91603175 0.91603175 0.89936508 0.93825397
0.94396825
 0.94396825 0.96095238 0.95539683 0.96650794]
Accuracy standard error:  [0.0100828  0.01731978 0.01731978 0.01846565
0.01031563 0.01233763
```

```
 0.00866353 0.0110725  0.01411806 0.01037347]
```



[11]:

```
# Find the minimum RSS target
imax = np.argmax(acc_mean)
acc_tgt = acc_mean[imax] - acc_se[imax]
I = np.where(acc_mean >=acc_tgt)[0]
iopt = I[0]
dopt = ncomp_test[iopt]
# Print results
print('Iopt: ',iopt)
print("Optimal no. of Components:" , dopt)
print("Optimal Acc:", acc_mean[iopt]*100)
```

```
Iopt:  7
Optimal no. of Components: 9
Optimal Acc: 96.09523809523809
```

1.4 Hyper-Parameter Optimization with GridCV.

We will now try a more complex classifier – a support vector classifier with a radial basis function. When we use such a classifer, there will be a number of parameters to tune. When the number of parameters to tune becomes large, writing a loop over multiple parameters as we did above becomes cumbersome. The `sklearn` package has a very nice routine, `GridSearchCV` to perform this sort of parameter search.

Before, we do this we need to create an estimator `Pipeline`. An estimator pipeline is a sequence of transformations followed by an estimator that will operate on the transformed data. Create the following pipeline:

- Create a `StandardScaler()` object called `scaler` for the first transformation
- Create a `PCA()` object called `pca` for the second transformation
- Create a `SVC()` object called `svc` for the final SVM classifier. Set the parameter `kernel='rbf'`.

Once you have the three steps defined, you can create the pipeline with the command:

```
pipe = Pipeline(steps=[('scaler', scaler), ('pca', pca), ('svc', svc)])
```

[12]:
```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

# TODO
scaler = StandardScaler()
pca = PCA()
svc = SVC(kernel='rbf')
pipe = Pipeline(steps=[('scaler', scaler), ('pca', pca), ('svc', svc)])
```

We next define all the parameters that we want to search over. Define the following arrays:

- `ncomp_test`: values from 3 to 10 representing number of PCs to test
- `C_test`: values of `C` in the SVC to test. Use `10^{-2}, 10^{-1}, ... ,10^{3}`
- `gam_test`: values of `gamma` in the SVC to test. Use `10^{-3}, 10^{-2}, ... ,10^{1}`

[13]:
```
# TODO
ncomp_test = np.arange(3,11)
c_test = np.logspace(-2, 3 , num = 6 , base = 10)
gam_test = np.logspace(-3 , 1, num = 5 , base = 10)
```

Next, we create a dictionary `params` of the form:

```
params =  {'pca__n_components': ncomp_test, 'svc__C' : c_test, ...}
```

Each key in the dictionary is the of the form `estimator__param` and the value is the values to be tested.

[14]:
```
# TODO
params = {
    'pca__n_components' : ncomp_test ,
    'svc__C' : c_test ,
    'svc__gamma' : gam_test
}
```

Finally, an object `estimator = GridSearchCV(...)` from `pipe` and `params`. Set `cv=5`, `train_score=True` and `iid=False`. Fit the estimator from the data `X,y`. Then the estimator will perform the cross-validation over all the parameters. This may take a minute since we are search over so many parameters.

[17]:
```
# TODO
estimator = GridSearchCV(pipe , cv = 5 , return_train_score = True , iid =␣
 ↪False, param_grid = params)
estimator.fit(X,y)
```

```
C:\Users\deept\anaconda3\envs\tensorflow\lib\site-
packages\sklearn\model_selection\_search.py:849: FutureWarning: The parameter
'iid' is deprecated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

[17]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('pca', PCA()), ('svc', SVC())]),
             iid=False,
             param_grid={'pca__n_components': array([ 3,  4,  5,  6,  7,  8,  9,
10]),
                         'svc__C': array([1.e-02, 1.e-01, 1.e+00, 1.e+01,
1.e+02, 1.e+03]),
                         'svc__gamma': array([1.e-03, 1.e-02, 1.e-01, 1.e+00,
1.e+01])},
             return_train_score=True)
```

Print the best test score and best parameters. They are fields in `estimator`. If you did it correctly, it should be a little higher than the logistic regression (about 0.97 to 0.98 accuracy).

[18]:
```
# TODO
print('Best Params',estimator.best_params_)
print('Best Score' , estimator.best_score_)
```

```
Best Params {'pca__n_components': 5, 'svc__C': 1.0, 'svc__gamma': 0.1}
Best Score 0.9777777777777779
```

Finally, you can get the test score for all the parameter choices from

```
test_score = estimator.cv_results_['mean_test_score']
```

Use the `imshow` command to plot the mean test score over `gamma` and `C` for the value `n_components=5`.

[19]:
```
# TODO
test_score = estimator.cv_results_['mean_test_score']
print(test_score.shape)
pc_n=estimator.cv_results_['param_pca__n_components'].data.astype(int)
tsc=estimator.cv_results_['mean_test_score'][np.where(pc_n==5)]
print(estimator.cv_results_['param_svc__C'][np.where(pc_n==5)])
print(estimator.cv_results_['param_svc__gamma'][np.where(pc_n==5)])
print(tsc.shape)
print(tsc)
tsc=tsc.reshape(len(c_test),len(gam_test))
```

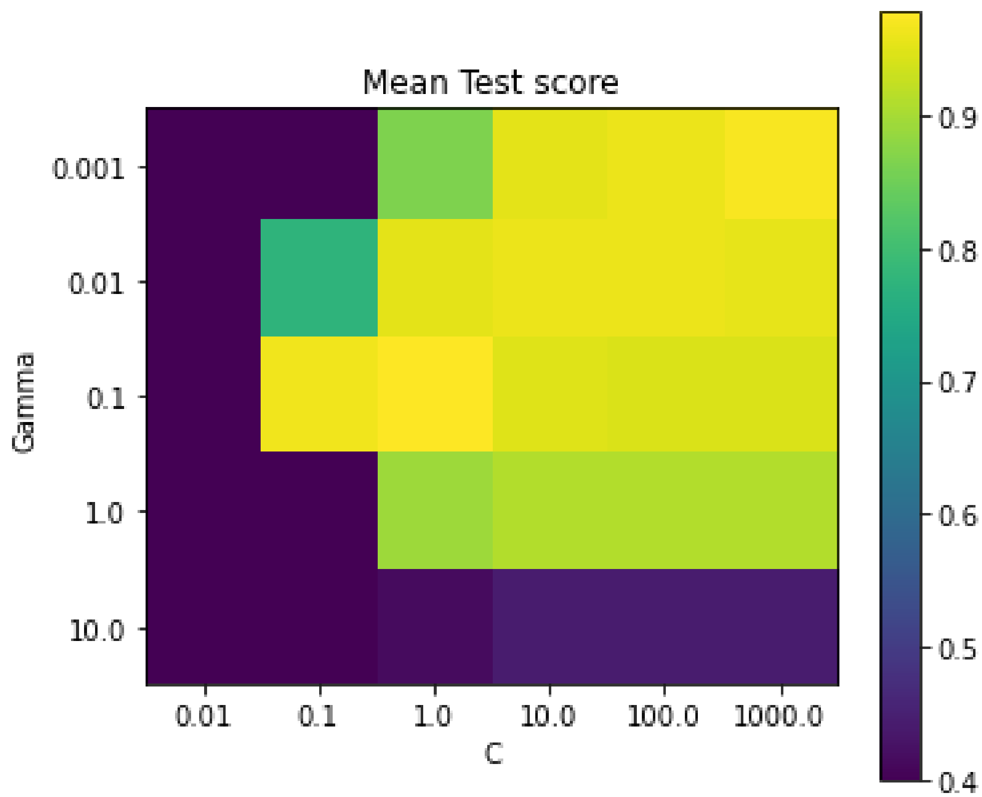
```
print(tsc)
```

```
(240,)
[0.01 0.01 0.01 0.01 0.01 0.1 0.1 0.1 0.1 0.1 1.0 1.0 1.0 1.0 1.0 10.0
 10.0 10.0 10.0 10.0 100.0 100.0 100.0 100.0 100.0 1000.0 1000.0 1000.0
 1000.0 1000.0]
[0.001 0.01 0.1 1.0 10.0 0.001 0.01 0.1 1.0 10.0 0.001 0.01 0.1 1.0 10.0
 0.001 0.01 0.1 1.0 10.0 0.001 0.01 0.1 1.0 10.0 0.001 0.01 0.1 1.0 10.0]
(30,)
[0.39904762 0.39904762 0.39904762 0.39904762 0.39904762 0.39904762
 0.77603175 0.96619048 0.39904762 0.39904762 0.86587302 0.95507937
 0.97777778 0.89365079 0.41603175 0.95507937 0.96095238 0.95
 0.91047619 0.44412698 0.96111111 0.96095238 0.94428571 0.91047619
 0.44412698 0.97206349 0.95555556 0.94428571 0.91047619 0.44412698]
[[0.39904762 0.39904762 0.39904762 0.39904762 0.39904762]
 [0.39904762 0.77603175 0.96619048 0.39904762 0.39904762]
 [0.86587302 0.95507937 0.97777778 0.89365079 0.41603175]
 [0.95507937 0.96095238 0.95       0.91047619 0.44412698]
 [0.96111111 0.96095238 0.94428571 0.91047619 0.44412698]
 [0.97206349 0.95555556 0.94428571 0.91047619 0.44412698]]
```

[25]:
```
print(c_test.astype(object))
print(gam_test.astype(object))
```

```
[0.01 0.1 1.0 10.0 100.0 1000.0]
[0.001 0.01 0.1 1.0 10.0]
```

[26]:
```
plt.figure(figsize=(6, 5))
plt.subplots_adjust(left=.2, right=0.95, bottom=0.15, top=0.95)
plt.imshow(tsc.T,interpolation='nearest')
plt.xlabel('C')
plt.ylabel('Gamma')
plt.colorbar()
plt.yticks(np.arange(len(gam_test)), gam_test)
plt.xticks(np.arange(len(c_test)), c_test)
plt.title('Mean Test score')
plt.show()
```


Mean Test score
0.001
0.01
0.1
1.0
10.0
Gamma
0.01
0.1
1.0
10.0
100.0
1000.0
C
0.9
0.8
0.7
0.6
0.5
0.4