Letter Recognition Using SVM

Let's now tackle a slightly more complex problem - letter recognition. We'll first explore the dataset a bit, prepare it (scale etc.) and then experiment with linear and non-linear SVMs with various hyperparameters.

Data Understanding

Let's first understand the shape, attributes etc. of the dataset.

In [1]:
```
# libraries
import pandas as pd
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import scale

# dataset
letters = pd.read_csv("letter-recognition.csv")
```

In [2]:

```
# about the dataset

# dimensions
print("Dimensions: ", letters.shape, "\n")

# data types
print(letters.info())

# head
letters.head()
```

```
Dimensions:  (20000, 17)

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20000 entries, 0 to 19999
Data columns (total 17 columns):
letter    20000 non-null object
xbox      20000 non-null int64
ybox      20000 non-null int64
width     20000 non-null int64
height    20000 non-null int64
onpix     20000 non-null int64
xbar      20000 non-null int64
ybar      20000 non-null int64
x2bar     20000 non-null int64
y2bar     20000 non-null int64
xybar     20000 non-null int64
x2ybar    20000 non-null int64
xy2bar    20000 non-null int64
xedge     20000 non-null int64
xedgey    20000 non-null int64
yedge     20000 non-null int64
yedgex    20000 non-null int64
dtypes: int64(16), object(1)
memory usage: 2.6+ MB
None
```

Out[2]:

	letter	xbox	ybox	width	height	onpix	xbar	ybar	x2bar	y2bar	xybar	x2ybar	xy2ba
0	T	2	8	3	5	1	8	13	0	6	6	10	8
1	I	5	12	3	7	2	10	5	5	4	13	3	9
2	D	4	11	6	8	6	10	6	2	6	10	3	7
3	N	7	11	6	6	3	5	9	4	6	4	4	10
4	G	2	1	3	1	1	8	6	6	6	6	5	9

In [3]:
```
# a quirky bug: the column names have a space, e.g. 'xbox ', which throws and
 error when indexed
print(letters.columns)
```

```
Index(['letter', 'xbox ', 'ybox ', 'width ', 'height', 'onpix ', 'xbar ',
       'ybar ', 'x2bar', 'y2bar ', 'xybar ', 'x2ybar', 'xy2bar', 'xedge ',
       'xedgey', 'yedge ', 'yedgex'],
      dtype='object')
```

In [4]:
```
# let's 'reindex' the column names
letters.columns = ['letter', 'xbox', 'ybox', 'width', 'height', 'onpix', 'xba
r',
       'ybar', 'x2bar', 'y2bar', 'xybar', 'x2ybar', 'xy2bar', 'xedge',
       'xedgey', 'yedge', 'yedgex']
print(letters.columns)
```

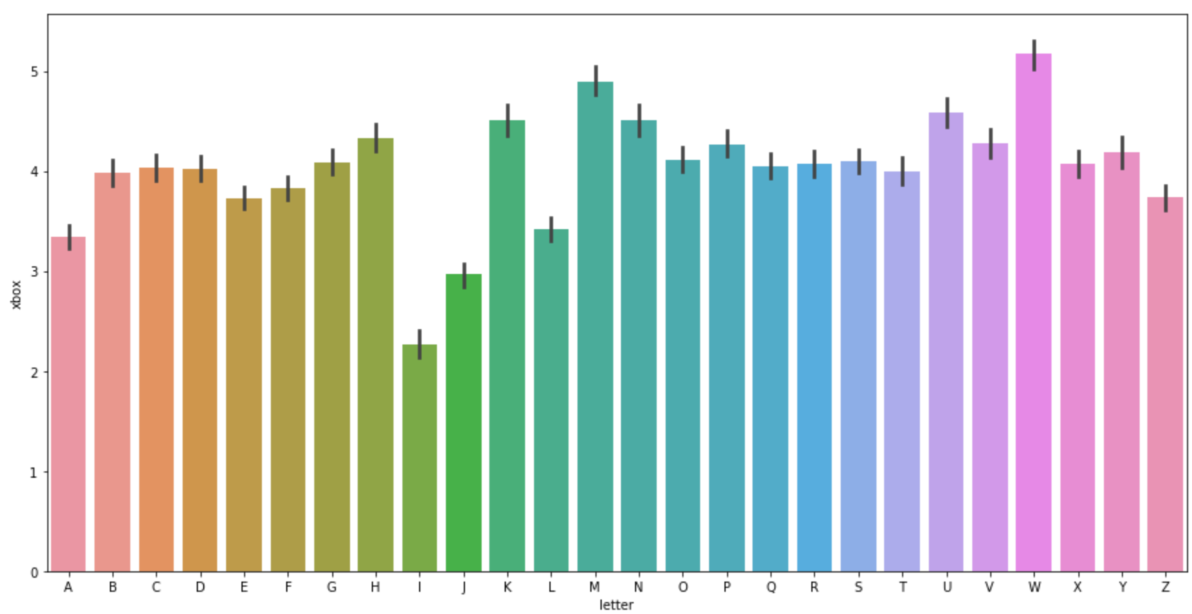
```
Index(['letter', 'xbox', 'ybox', 'width', 'height', 'onpix', 'xbar', 'ybar',
       'x2bar', 'y2bar', 'xybar', 'x2ybar', 'xy2bar', 'xedge', 'xedgey',
       'yedge', 'yedgex'],
      dtype='object')
```

In [5]:
```
order = list(np.sort(letters['letter'].unique()))
print(order)
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

In [6]:
```
# basic plots: How do various attributes vary with the letters

plt.figure(figsize=(16, 8))
sns.barplot(x='letter', y='xbox',
            data=letters,
            order=order)
```

Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x104cca5f8>



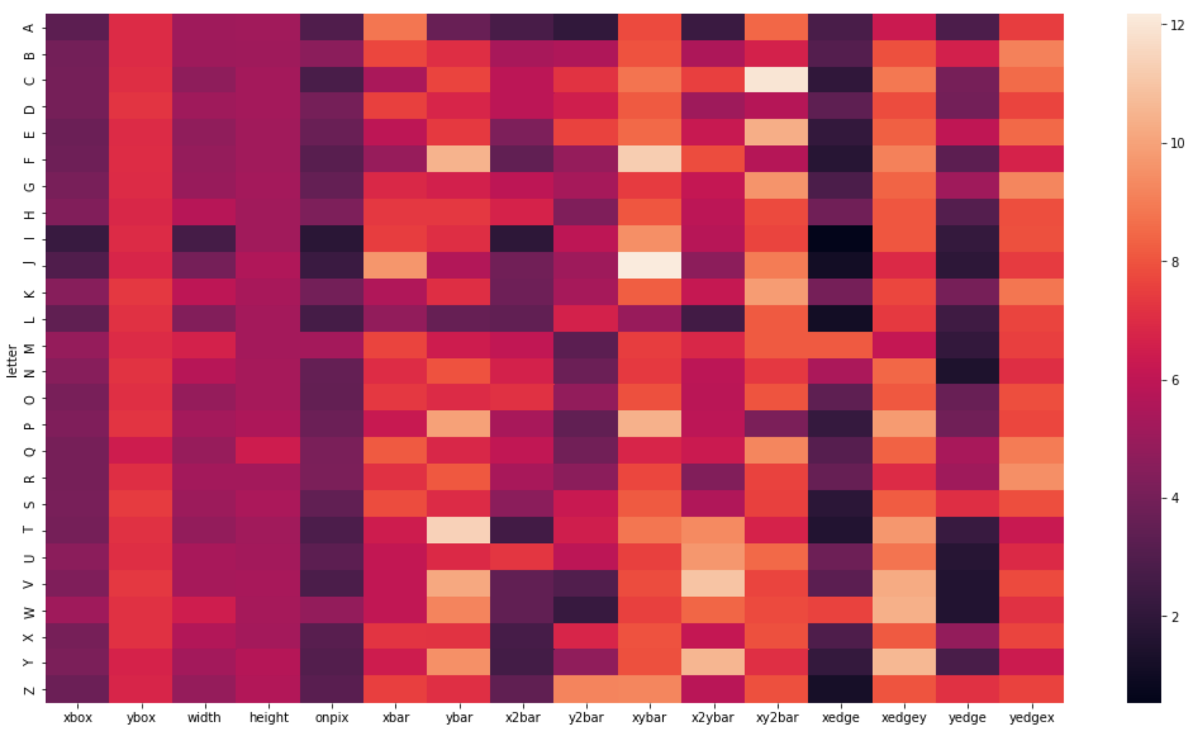
In [7]:
```
letter_means = letters.groupby('letter').mean()
letter_means.head()
```

Out[7]:

	xbox	ybox	width	height	onpix	xbar	ybar	x2bar	
letter									
A	3.337136	6.975919	5.128010	5.178707	2.991128	8.851711	3.631179	2.755387	2.
B	3.985640	6.962141	5.088773	5.169713	4.596606	7.671018	7.062663	5.366841	5.
C	4.031250	7.063859	4.701087	5.296196	2.775815	5.437500	7.627717	5.927989	7.
D	4.023602	7.244720	5.170186	5.288199	4.026087	7.539130	6.806211	5.921739	6.
E	3.727865	6.944010	4.756510	5.201823	3.679688	5.966146	7.352865	4.223958	7.

In [8]:
```
plt.figure(figsize=(18, 10))
sns.heatmap(letter_means)
```

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x10af66a20>



Data Preparation

Let's conduct some data preparation steps before modeling. Firstly, let's see if it is important to **rescale** the features, since they may have varying ranges. For example, here are the average values:

In [9]:
```
# average feature values
round(letters.drop('letter', axis=1).mean(), 2)
```

Out[9]:
```
xbox      4.02
ybox      7.04
width     5.12
height    5.37
onpix     3.51
xbar      6.90
ybar      7.50
x2bar     4.63
y2bar     5.18
xybar     8.28
x2ybar    6.45
xy2bar    7.93
xedge     3.05
xedgey    8.34
yedge     3.69
yedgex    7.80
dtype: float64
```

In this case, the average values do not vary a lot (e.g. having a diff of an order of magnitude). Nevertheless, it is better to rescale them.

In [10]:
```
# splitting into X and y
X = letters.drop("letter", axis = 1)
y = letters['letter']
```

In [11]:
```
# scaling the features
X_scaled = scale(X)

# train test split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size =
0.3, random_state = 101)
```

Model Building

Let's fist build two basic models - linear and non-linear with default hyperparameters, and compare the accuracies.

In [12]:
```
# linear model

model_linear = SVC(kernel='linear')
model_linear.fit(X_train, y_train)

# predict
y_pred = model_linear.predict(X_test)
```

In [13]:
```
# confusion matrix and accuracy

# accuracy
print("accuracy:", metrics.accuracy_score(y_true=y_test, y_pred=y_pred), "\n")

# cm
print(metrics.confusion_matrix(y_true=y_test, y_pred=y_pred))
```

```
accuracy: 0.8523333333333334

[[198   0   0   0   0   0   1   1   0   1   1   1   0   0   0   0   0   1
    0   1   1   0   0   0   3   0]
 [  0 188   0   3   0   1   3   3   1   0   1   0   0   2   0   1   1   9
    3   0   0   1   0   1   0   0]
 [  1   0 200   0   7   0  12   1   0   0   5   0   0   0   3   0   0   0
    0   0   1   0   0   0   0   0]
 [  1  15   0 210   0   1   2   2   1   0   1   0   1   5   3   0   0   5
    0   1   0   0   0   0   0   0]
 [  0   1   3   0 204   2   6   1   0   0   1   5   0   0   0   0   2   2
    1   2   0   0   0   1   0   3]
 [  0   0   0   1   1 201   1   2   1   1   0   0   0   2   0   2   0   0
    3   7   1   0   1   0   1   0]
 [  0   1   9   4   2   2 167   1   0   1   4   3   1   0   1   0   9   1
    8   0   0   2   3   0   0   0]
 [  0   7   3  11   0   4   3 141   0   2   4   1   2   0  12   0   4  12
    0   0   4   2   0   4   1   0]
 [  0   0   2   3   0   6   0   0 184   9   0   0   0   0   1   0   0   0
    3   0   0   0   0   4   0   3]
 [  2   0   0   3   0   2   0   2  10 187   0   0   0   1   2   0   0   1
    5   0   1   0   0   0   0   4]
 [  0   1   5   2   0   0   1   3   0   0 198   2   2   0   0   0   0  19
    0   0   0   0   0  12   0   0]
 [  2   1   3   2   5   0   8   1   0   0   1 206   0   0   0   0   5   0
    2   1   0   0   0   0   0   0]
 [  0   3   0   0   0   0   0   3   0   0   0   0 222   1   0   0   0   2
    0   0   0   0   3   0   0   0]
 [  1   0   0   4   0   0   0   6   0   0   0   0   1 235   1   1   0   0
    0   0   0   1   1   0   0   0]
 [  3   0   4   7   0   0   0  21   0   0   0   0   2   0 163   3   2   3
    0   0   3   0  10   0   0   0]
 [  0   2   0   2   0  16   5   1   0   1   3   0   0   0   1 225   0   0
    0   0   0   1   0   0   8   0]
 [  3   1   0   0   4   0   9   0   0   2   0   1   0   0   6   0 198   0
    8   0   0   0   1   0   0   2]
 [ 11  11   0   2   0   1   6   3   0   0  10   0   0   3   4   0   2 188
    0   1   0   0   0   1   0   0]
 [  1  13   0   0   9   5   8   0   7   1   0   2   0   0   0   1   6   1
  155   6   0   0   0   3   0  10]
 [  0   0   0   1   1   4   2   4   1   0   0   0   0   0   0   0   0   1
    3 214   0   0   0   1   1   6]
 [  2   0   1   2   0   0   0   2   0   0   0   0   1   1   3   0   0   0
    0   0 211   0   1   0   0   0]
 [  2   2   0   0   0   0   1   3   0   0   0   0   0   1   0   1   0   3
    0   0   0 190   6   0   2   0]
 [  0   0   0   0   0   0   1   0   0   0   0   0   6   1   2   0   0   0
    0   0   0   0 212   0   0   0]
 [  0   2   0   4   5   0   1   0   2   3   3   3   0   0   1   0   0   1
    2   2   1   0   0 212   1   1]
 [  2   0   0   0   0   2   0   1   0   0   0   0   1   0   0   0   3   0
    0   4   1  10   0   2 211   0]
 [  1   0   0   0   3   0   0   0   0   6   0   0   0   0   0   0   5   0
   18   1   0   0   0   1   0 194]]
```

The linear model gives approx. 85% accuracy. Let's look at a sufficiently non-linear model with randomly chosen hyperparameters.

In [14]:
```
# non-linear model
# using rbf kernel, C=1, default value of gamma

# model
non_linear_model = SVC(kernel='rbf')

# fit
non_linear_model.fit(X_train, y_train)

# predict
y_pred = non_linear_model.predict(X_test)
```

In [15]:
```
# confusion matrix and accuracy

# accuracy
print("accuracy:", metrics.accuracy_score(y_true=y_test, y_pred=y_pred), "\n")

# cm
print(metrics.confusion_matrix(y_true=y_test, y_pred=y_pred))
```

```
accuracy: 0.9383333333333334

[[205   0   0   1   0   0   0   0   0   1   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   2   0]
 [  0 205   0   3   1   0   1   0   0   0   0   0   0   0   0   0   0   6
    1   0   0   0   0   1   0   0]
 [  0   0 213   0   5   0   7   1   0   0   0   0   0   0   4   0   0   0
    0   0   0   0   0   0   0   0]
 [  0   4   0 234   0   0   1   3   0   0   0   0   0   3   1   0   0   2
    0   0   0   0   0   0   0   0]
 [  0   0   0   0 221   1   9   0   0   0   0   0   0   0   0   0   0   1
    0   0   0   0   0   0   0   2]
 [  0   0   0   1   0 215   1   1   1   0   0   0   0   1   0   1   0   0
    1   3   0   0   0   0   0   0]
 [  0   0   3   4   1   1 202   0   0   0   0   1   1   0   2   0   0   1
    0   0   0   1   2   0   0   0]
 [  0   7   0   5   0   0   4 177   0   0   2   0   1   0   3   0   4  13
    0   0   1   0   0   0   0   0]
 [  0   0   1   1   0   3   0   0 194  11   0   0   0   0   0   1   0   0
    2   0   0   0   0   2   0   0]
 [  1   0   0   1   0   0   0   0   6 206   0   0   0   1   2   0   0   0
    2   0   0   0   0   1   0   0]
 [  0   4   0   2   0   0   0   4   0   0 217   0   1   0   0   0   0  14
    0   0   0   0   0   3   0   0]
 [  0   0   1   0   2   0   6   0   0   0   1 222   0   0   0   0   0   3
    0   0   0   0   0   2   0   0]
 [  0   5   0   0   0   0   0   2   0   0   0   0 225   0   0   0   0   0
    0   0   0   0   2   0   0   0]
 [  0   2   0   1   0   0   0   2   0   0   0   0   1 239   3   0   0   2
    0   0   0   0   0   0   1   0]
 [  0   0   0   1   0   0   0   0   0   0   0   0   0   0 209   0   1   1
    0   0   1   0   8   0   0   0]
 [  0   2   0   3   3  11   1   1   0   0   0   0   0   0   1 237   1   0
    0   0   0   0   0   0   5   0]
 [  0   0   0   0   2   0   2   0   0   0   0   0   0   0   6   0 222   0
    1   0   0   0   2   0   0   0]
 [  0  10   0   2   0   0   0   0   0   0   1   0   0   4   0   0   2 224
    0   0   0   0   0   0   0   0]
 [  0   3   0   0   2   3   0   0   0   0   0   0   0   0   0   0   0   0
  220   0   0   0   0   0   0   0]
 [  0   0   0   1   0   2   0   2   0   0   0   0   0   0   0   1   0   1
    0 228   0   0   0   3   1   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   1   1   0   0   0
    0   0 222   0   0   0   0   0]
 [  0   7   0   0   0   0   0   1   0   0   0   0   1   4   0   1   0   0
    0   0   0 193   1   0   3   0]
 [  0   1   0   0   0   0   1   1   0   0   0   0   0   1   0   0   0   0
    0   0   1   0 217   0   0   0]
 [  0   2   0   3   2   0   0   0   1   0   2   0   0   0   0   0   0   1
    0   0   0   0   0 233   0   0]
 [  2   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   1   0
    0   1   2   2   0   0 228   0]
 [  0   0   0   0   2   0   0   0   0   0   0   0   0   0   0   0   4   0
    1   0   0   0   0   0   0 222]]
```

The non-linear model gives approx. 93% accuracy. Thus, going forward, let's choose hyperparameters corresponding to non-linear models.

Grid Search: Hyperparameter Tuning

Let's now tune the model to find the optimal values of C and gamma corresponding to an RBF kernel. We'll use 5-fold cross validation.

In [16]:
```
# creating a KFold object with 5 splits
folds = KFold(n_splits = 5, shuffle = True, random_state = 101)

# specify range of hyperparameters
# Set the parameters by cross-validation
hyper_params = [ {'gamma': [1e-2, 1e-3, 1e-4],
                     'C': [1, 10, 100, 1000]}]


# specify model
model = SVC(kernel="rbf")

# set up GridSearchCV()
model_cv = GridSearchCV(estimator = model,
                        param_grid = hyper_params,
                        scoring= 'accuracy',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# fit the model
model_cv.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits

[Parallel(n_jobs=1)]: Done  60 out of  60 | elapsed: 10.0min finished
```

Out[16]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=101, shuffle=True),
       error_score='raise',
       estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid=[{'gamma': [0.01, 0.001, 0.0001], 'C': [1, 10, 100, 100
0]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring='accuracy', verbose=1)
```

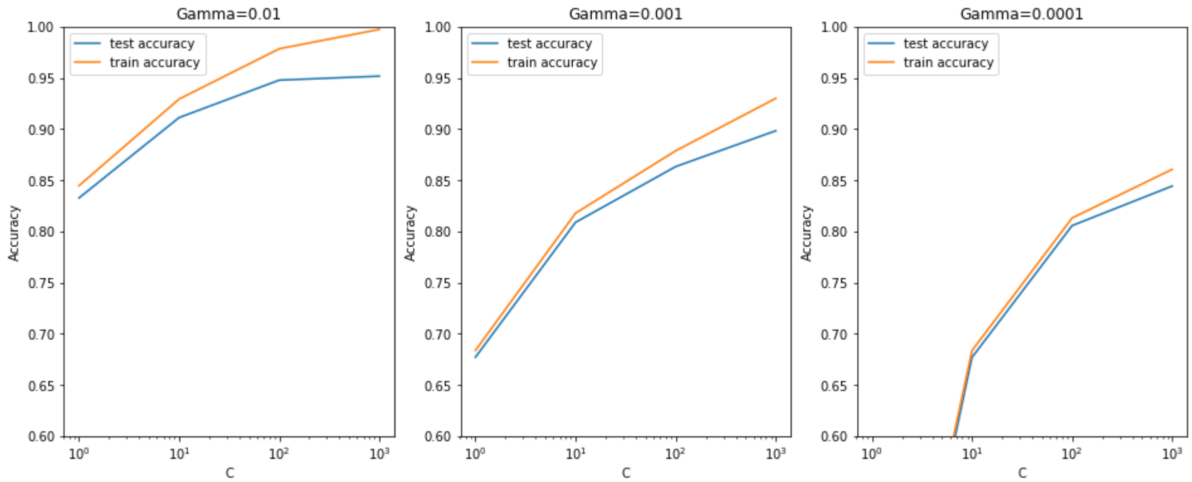
In [17]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[17]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_C	par
0	2.531680	1.421701	0.832714	0.844679	1	0.0
1	5.496493	1.896336	0.677214	0.683821	1	0.0
2	10.647400	2.109896	0.217571	0.227982	1	0.0
3	1.609691	0.936208	0.911214	0.929304	10	0.0
4	2.501903	1.427731	0.808929	0.817857	10	0.0
5	5.405686	1.914143	0.677000	0.683643	10	0.0
6	2.070304	0.750065	0.947786	0.978411	100	0.0
7	1.858425	0.965986	0.863357	0.878768	100	0.0
8	2.490351	1.424067	0.805714	0.813179	100	0.0
9	2.196414	0.660221	0.951714	0.997357	1000	0.0
10	2.385556	0.784292	0.898357	0.929982	1000	0.0
11	2.175660	1.081910	0.844357	0.860679	1000	0.0

12 rows × 22 columns

In [18]:
```
# converting C to numeric type for plotting on x-axis
cv_results['param_C'] = cv_results['param_C'].astype('int')

# # plotting
plt.figure(figsize=(16,6))

# subplot 1/3
plt.subplot(131)
gamma_01 = cv_results[cv_results['param_gamma']==0.01]

plt.plot(gamma_01["param_C"], gamma_01["mean_test_score"])
plt.plot(gamma_01["param_C"], gamma_01["mean_train_score"])
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title("Gamma=0.01")
plt.ylim([0.60, 1])
plt.legend(['test accuracy', 'train accuracy'], loc='upper left')
plt.xscale('log')

# subplot 2/3
plt.subplot(132)
gamma_001 = cv_results[cv_results['param_gamma']==0.001]

plt.plot(gamma_001["param_C"], gamma_001["mean_test_score"])
plt.plot(gamma_001["param_C"], gamma_001["mean_train_score"])
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title("Gamma=0.001")
plt.ylim([0.60, 1])
plt.legend(['test accuracy', 'train accuracy'], loc='upper left')
plt.xscale('log')


# subplot 3/3
plt.subplot(133)
gamma_0001 = cv_results[cv_results['param_gamma']==0.0001]

plt.plot(gamma_0001["param_C"], gamma_0001["mean_test_score"])
plt.plot(gamma_0001["param_C"], gamma_0001["mean_train_score"])
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title("Gamma=0.0001")
plt.ylim([0.60, 1])
plt.legend(['test accuracy', 'train accuracy'], loc='upper left')
plt.xscale('log')
```



The plots above show some useful insights:

- Non-linear models (high gamma) perform *much better* than the linear ones
- At any value of gamma, a high value of C leads to better performance
- None of the models tend to overfit (even the complex ones), since the training and test accuracies closely follow each other

This suggests that the problem and the data is **inherently non-linear** in nature, and a complex model will outperform simple, linear models in this case.

Let's now choose the best hyperparameters.

In [19]:
```
# printing the optimal accuracy score and hyperparameters
best_score = model_cv.best_score_
best_hyperparams = model_cv.best_params_

print("The best test score is {0} corresponding to hyperparameters {1}".format
(best_score, best_hyperparams))
```

```
The best test score is 0.9517142857142857 corresponding to hyperparameters
{'C': 1000, 'gamma': 0.01}
```

Building and Evaluating the Final Model

Let's now build and evaluate the final model, i.e. the model with highest test accuracy.

In [20]:
```
# model with optimal hyperparameters

# model
model = SVC(C=1000, gamma=0.01, kernel="rbf")

model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# metrics
print("accuracy", metrics.accuracy_score(y_test, y_pred), "\n")
print(metrics.confusion_matrix(y_test, y_pred), "\n")
```

```
accuracy 0.9596666666666667

[[206   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0
    0   0   0   0   0   0   2   0]
 [  0 211   0   2   1   0   1   0   0   0   0   0   0   0   0   0   0   1
    2   0   0   0   0   0   0   0]
 [  0   0 220   0   3   0   4   1   0   0   0   0   0   0   2   0   0   0
    0   0   0   0   0   0   0   0]
 [  0   3   0 236   0   1   0   1   0   1   0   0   0   2   2   0   0   1
    1   0   0   0   0   0   0   0]
 [  0   0   1   0 225   1   4   0   0   0   0   0   0   0   0   0   0   0
    1   0   0   0   0   0   0   2]
 [  0   0   0   1   0 217   0   0   1   1   0   0   0   1   0   3   0   0
    0   0   0   1   0   0   0   0]
 [  0   0   2   3   1   0 209   0   0   0   0   0   1   0   1   0   0   0
    0   0   0   1   1   0   0   0]
 [  0   1   3   5   0   0   2 195   1   1   2   1   1   0   1   0   3   1
    0   0   0   0   0   0   0   0]
 [  0   0   0   1   0   1   0   0 203   8   0   0   0   0   0   0   1   0
    0   0   0   0   0   1   0   0]
 [  0   0   0   0   0   0   0   0   7 209   0   0   0   1   0   0   0   1
    0   0   2   0   0   0   0   0]
 [  0   1   0   0   2   0   0   5   0   0 228   0   0   0   0   0   0   5
    0   0   0   0   0   4   0   0]
 [  0   0   0   0   0   0   1   1   0   0   0 232   0   0   0   1   0   1
    0   1   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 230   2   0   0   0   0
    0   0   0   0   2   0   0   0]
 [  0   3   0   1   0   0   0   0   0   0   0   0   1 244   0   0   0   1
    0   0   0   0   1   0   0   0]
 [  0   0   2   0   0   0   2   0   0   1   0   0   2   0 210   0   1   1
    0   0   1   0   1   0   0   0]
 [  0   0   0   0   1   8   0   1   1   0   0   0   0   0   0 252   1   0
    0   0   0   0   0   0   1   0]
 [  0   0   0   0   3   0   1   0   0   0   0   0   0   0   2   1 226   0
    0   0   0   0   0   0   0   2]
 [  0   8   0   1   0   0   1   3   0   0   3   0   0   3   0   0   0 224
    0   0   0   0   0   0   0   0]
 [  0   1   0   0   1   2   1   0   0   0   0   0   0   0   0   0   0   0
  223   0   0   0   0   0   0   0]
 [  0   0   0   1   0   0   0   1   0   0   0   0   0   0   0   1   0   0
    0 235   0   0   0   1   0   0]
 [  2   0   0   0   0   0   1   2   0   0   0   0   0   2   0   0   0   0
    0   0 217   0   0   0   0   0]
 [  0   4   0   0   0   0   0   0   0   0   0   0   1   0   0   1   0   0
    0   0   0 203   1   0   1   0]
 [  0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   1   0   0
    0   0   0   0 220   0   0   0]
 [  0   1   0   3   2   0   0   0   0   0   3   1   0   0   0   0   0   1
    0   0   0   0   0 232   1   0]
 [  1   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0
    1   0   1   2   1   0 230   0]
 [  0   0   0   0   2   1   0   0   0   0   0   0   0   0   0   0   3   0
    1   1   0   0   0   0   0 221]]
```

Conclusion

The accuracy achieved using a non-linear kernel (~0.95) is mush higher than that of a linear one (~0.85). We can conclude that the problem is highly non-linear in nature. Accuarcy is changed from 92% to 94% at the value of hyperparameters i.e. c= 2 & Sigma = '0.05'