Non-Linear SVM - Email Spam Classifier

In this section, we'll build a non-linear SVM classifier to classify emails and compare the performance with the linear SVM model.

The dataset can be downloaded here: https://archive.ics.uci.edu/ml/datasets/spambase (https://archive.ics.uci.edu/ml/datasets/spambase)

To reiterate, the performance of the linear model was as follows:

- accuracy 0.93
- precision 0.92
- recall 0.89

In this section, we will build a non-linear model (using non-linear kernels) and then find the optimal hyperparameters (the choice of kernel, C, gamma).

In [1]:
```
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
```

Loading Data

In [2]:
```
email_rec = pd.read_csv("Spam.txt",  sep = ',', header= None )
```

Renaming the column names

In [3]:
```
# renaming the columns
email_rec.columns  = ["word_freq_make", "word_freq_address", "word_freq_all", 
"word_freq_3d", 
                      "word_freq_our", "word_freq_over", "word_freq_remove", 
"word_freq_internet", 
                      "word_freq_order", "word_freq_mail", "word_freq_receive"
, "word_freq_will", 
                      "word_freq_people", "word_freq_report", "word_freq_addre
sses", "word_freq_free", 
                      "word_freq_business", "word_freq_email", "word_freq_you"
, "word_freq_credit", 
                      "word_freq_your", "word_freq_font", "word_freq_000", "wo
rd_freq_money", "word_freq_hp", 
                      "word_freq_hpl", "word_freq_george", "word_freq_650", "w
ord_freq_lab", "word_freq_labs", 
                      "word_freq_telnet", "word_freq_857", "word_freq_data", 
"word_freq_415", "word_freq_85", 
                      "word_freq_technology", "word_freq_1999", "word_freq_par
ts", "word_freq_pm", "word_freq_direct", 
                      "word_freq_cs", "word_freq_meeting", "word_freq_origina
l", "word_freq_project", "word_freq_re", 
                      "word_freq_edu", "word_freq_table", "word_freq_conferenc
e", "char_freq_;", "char_freq_(", 
                      "char_freq_[", "char_freq_!", "char_freq_$", "char_freq_
hash", "capital_run_length_average", 
                      "capital_run_length_longest", "capital_run_length_total"
, "spam"]
print(email_rec.head())
```

```
   word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
0            0.00               0.64           0.64           0.0
1            0.21               0.28           0.50           0.0
2            0.06               0.00           0.71           0.0
3            0.00               0.00           0.00           0.0
4            0.00               0.00           0.00           0.0

   word_freq_our  word_freq_over  word_freq_remove  word_freq_internet  \
0           0.32            0.00              0.00                0.00
1           0.14            0.28              0.21                0.07
2           1.23            0.19              0.19                0.12
3           0.63            0.00              0.31                0.63
4           0.63            0.00              0.31                0.63

   word_freq_order  word_freq_mail  ...   char_freq_;  char_freq_(  \
0             0.00            0.00  ...          0.00        0.000
1             0.00            0.94  ...          0.00        0.132
2             0.64            0.25  ...          0.01        0.143
3             0.31            0.63  ...          0.00        0.137
4             0.31            0.63  ...          0.00        0.135

   char_freq_[  char_freq_!  char_freq_$  char_freq_hash  \
0          0.0        0.778        0.000           0.000
1          0.0        0.372        0.180           0.048
2          0.0        0.276        0.184           0.010
3          0.0        0.137        0.000           0.000
4          0.0        0.135        0.000           0.000

   capital_run_length_average  capital_run_length_longest  \
0                       3.756                          61
1                       5.114                         101
2                       9.821                         485
3                       3.537                          40
4                       3.537                          40

   capital_run_length_total  spam
0                       278     1
1                      1028     1
2                      2259     1
3                       191     1
4                       191     1

[5 rows x 58 columns]
```

Data Preparation

In [4]:
```
# splitting into X and y
X = email_rec.drop("spam", axis = 1)
y = email_rec.spam.values.astype(int)
```

In [5]:
```
# 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 = 4)
```

Model Building

In [6]:
```
# using rbf kernel, C=1, default value of gamma

model = SVC(C = 1, kernel='rbf')
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Model Evaluation Metrics

In [7]:
```
# confusion matrix
confusion_matrix(y_true=y_test, y_pred=y_pred)
```

Out[7]:
```
array([[811,  38],
       [ 61, 471]])
```

In [8]:
```
# accuracy
print("accuracy", metrics.accuracy_score(y_test, y_pred))

# precision
print("precision", metrics.precision_score(y_test, y_pred))

# recall/sensitivity
print("recall", metrics.recall_score(y_test, y_pred))
```

```
accuracy 0.9283128167994207
precision 0.925343811394892
recall 0.8853383458646616
```

Hyperparameter Tuning

Now, we have multiple hyperparameters to optimise -

- The choice of kernel (linear, rbf etc.)
- C
- gamma

We'll use the `GridSearchCV()` method to tune the hyperparameters.

Grid Search to Find Optimal Hyperparameters

Let's first use the RBF kernel to find the optimal C and gamma (we can consider the kernel as a hyperparameter as well, though training the model will take an exorbitant amount of time).

In [9]:
```
# creating a KFold object with 5 splits
folds = KFold(n_splits = 5, shuffle = True, random_state = 4)

# 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:   26.3s finished
```

Out[9]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=4, 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)
```

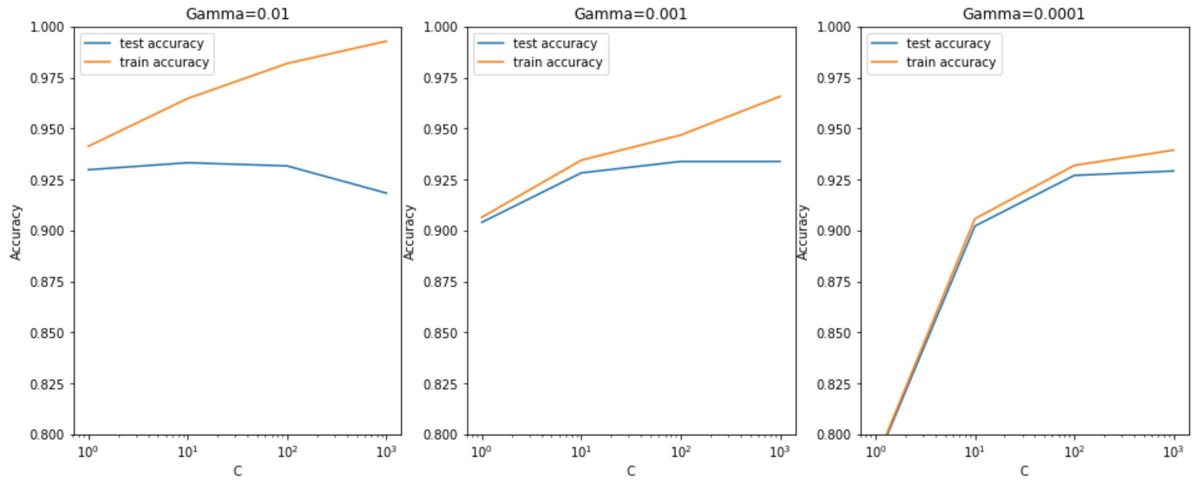
In [10]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[10]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_C	par
0	0.206314	0.039982	0.929814	0.941304	1	0.0
1	0.302772	0.063263	0.904037	0.906522	1	0.0
2	0.437604	0.098419	0.786025	0.786957	1	0.0
3	0.169503	0.028794	0.933230	0.964752	10	0.0
4	0.180548	0.034636	0.928261	0.934472	10	0.0
5	0.254190	0.051907	0.902174	0.905745	10	0.0
6	0.199619	0.024034	0.931677	0.981910	100	0.0
7	0.168072	0.026588	0.933851	0.946817	100	0.0
8	0.181463	0.034266	0.927019	0.931910	100	0.0
9	0.304800	0.023001	0.918323	0.992857	1000	0.0
10	0.297760	0.021695	0.933851	0.965761	1000	0.0
11	0.189173	0.026324	0.929193	0.939441	1000	0.0

12 rows × 22 columns

In [11]:
```
# 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([0.80, 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([0.80, 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([0.80, 1])
plt.legend(['test accuracy', 'train accuracy'], loc='upper left')
plt.xscale('log')
```



This plot reveals some interesting insights:

- **High values of gamma** lead to **overfitting** (especially at high values of C); note that the training accuracy at gamma=0.01 and C=1000 reaches almost 99%
- The **training score increases with higher gamma**, though the **test scores are comparable** (at sufficiently high cost, i.e. C > 10)
- The least amount of overfitting (i.e. difference between train and test accuracy) occurs at low gamma, i.e. a quite *simple non-linear model*

In [12]:
```
# 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 0.9338509316770186 corresponding to hyperparameters
{'C': 100, 'gamma': 0.001}
```

Though sklearn suggests the optimal scores mentioned above (gamma=0.001, C=100), one could argue that it is better to choose a simpler, more non-linear model with gamma=0.0001. This is because the optimal values mentioned here are calculated based on the average test accuracy (but not considering subjective parameters such as model complexity).

We can achieve comparable average test accuracy (~92.5%) with gamma=0.0001 as well, though we'll have to increase the cost C for that. So to achieve high accuracy, there's a tradeoff between:

- High gamma (i.e. high non-linearity) and average value of C
- Low gamma (i.e. less non-linearity) and high value of C

We argue that the model will be simpler if it has as less non-linearity as possible, so we choose gamma=0.0001 and a high C=100.

Building and Evaluating the Final Model

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

In [13]:
```
# specify optimal hyperparameters
best_params = {"C": 100, "gamma": 0.0001, "kernel":"rbf"}

# model
model = SVC(C=100, gamma=0.0001, kernel="rbf")

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

# metrics
print(metrics.confusion_matrix(y_test, y_pred), "\n")
print("accuracy", metrics.accuracy_score(y_test, y_pred))
print("precision", metrics.precision_score(y_test, y_pred))
print("sensitivity/recall", metrics.recall_score(y_test, y_pred))
```

```
[[810  39]
 [ 60 472]]

accuracy 0.9283128167994207
precision 0.923679060665362
sensitivity/recall 0.8872180451127819
```

Conclusion

The accuracy achieved using a non-linear kernel is comparable to that of a linear one. Thus, it turns out that for this problem, **you do not really need a non-linear kernel**.