Non Linear SVM - Email Spam Classifier

October 22, 2020

1 Non-Linear SVM - Email Spam Classifier

We'll build a non-linear SVM classifier to classify emails and compare the performance with the linear SVM model.

To reiterate, the performance of the linear model was as follows: - accuracy 0.93 - precision 0.92 - recall 0.89

```
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.csv",  sep = ',')
```

1.1 Data Preparation

```
In [3]: # splitting into X and y
        X = email_rec.drop("spam", axis = 1)
        y = email_rec.spam.values.astype(int)
```

```
In [4]: # 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, rando
```

```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: DataConversionWarning: Data with input
```

1.2 Model Building

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

```
C:\Anaconda3\lib\site-packages\sklearn\svm\base.py:196: FutureWarning: The default value of gam
  "avoid this warning.", FutureWarning)
```

1.3 Model Evaluation Metrics

```
In [6]: # confusion matrix
        confusion_matrix(y_true=y_test, y_pred=y_pred)

Out[6]: array([[811,  38],
               [ 61, 471]], dtype=int64)
```

```
In [7]: # 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
```

1.4 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.

1.5 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 [8]: # 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)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  60 out of  60 | elapsed:   46.5s finished
```

```
Out[8]: GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
               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=None,
               param_grid=[{'gamma': [0.01, 0.001, 0.0001], 'C': [1, 10, 100, 1000]}],
               pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
               scoring='accuracy', verbose=1)
```

```
In [9]: # cv results
        cv_results = pd.DataFrame(model_cv.cv_results_)
        cv_results
```

```
Out[9]:     mean_fit_time  std_fit_time  mean_score_time  std_score_time param_C  \
        0        0.426284      0.020828         0.060368        0.003716       1
        1        0.474971      0.022318         0.093137        0.009366       1
        2        0.876207      0.103492         0.146991        0.002362       1
        3        0.296357      0.018458         0.046820        0.003405      10
        4        0.314254      0.010734         0.054949        0.004604      10
        5        0.453887      0.010305         0.083359        0.002161      10
        6        0.405114      0.097553         0.039683        0.007592     100
```

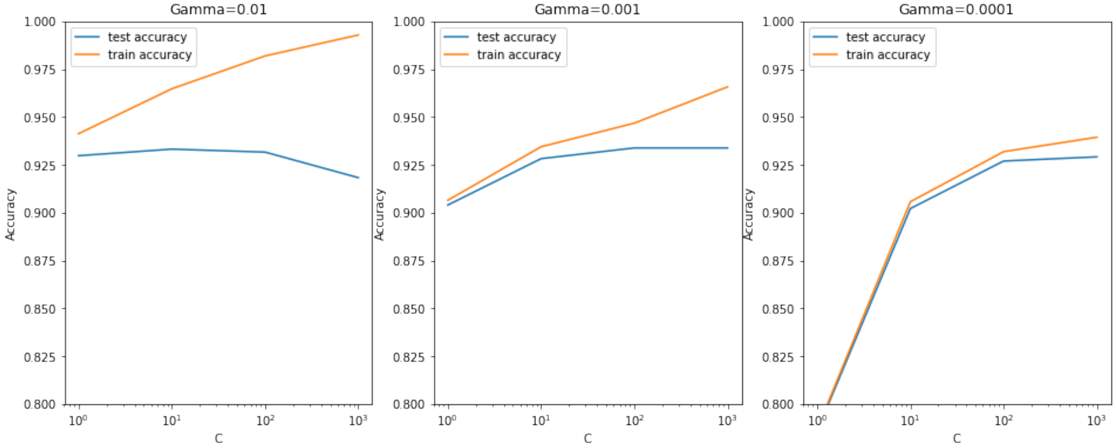
```
7        0.390674      0.050723         0.049930        0.006212     100
8        0.368195      0.017466         0.059163        0.001172     100
9        0.545270      0.038658         0.041219        0.009998    1000
10       0.540438      0.045874         0.035787        0.002707    1000
11       0.419198      0.083070         0.048977        0.010300    1000

   param_gamma                        params  split0_test_score  \
0         0.01       {'C': 1, 'gamma': 0.01}           0.917702   
1        0.001      {'C': 1, 'gamma': 0.001}           0.886646   
2       0.0001     {'C': 1, 'gamma': 0.0001}           0.770186   
3         0.01      {'C': 10, 'gamma': 0.01}           0.909938   
4        0.001     {'C': 10, 'gamma': 0.001}           0.917702   
5       0.0001    {'C': 10, 'gamma': 0.0001}           0.883540   
6         0.01     {'C': 100, 'gamma': 0.01}           0.913043   
7        0.001    {'C': 100, 'gamma': 0.001}           0.923913   
8       0.0001   {'C': 100, 'gamma': 0.0001}           0.919255   
9         0.01    {'C': 1000, 'gamma': 0.01}           0.908385   
10       0.001   {'C': 1000, 'gamma': 0.001}           0.919255   
11      0.0001  {'C': 1000, 'gamma': 0.0001}           0.920807   

    split1_test_score  split2_test_score  ...  mean_test_score  \
0            0.939441           0.922360  ...         0.929814   
1            0.919255           0.899068  ...         0.904037   
2            0.802795           0.791925  ...         0.786025   
3            0.944099           0.934783  ...         0.933230   
4            0.934783           0.916149  ...         0.928261   
5            0.914596           0.899068  ...         0.902174   
6            0.937888           0.934783  ...         0.931677   
7            0.940994           0.925466  ...         0.933851   
8            0.934783           0.917702  ...         0.927019   
9            0.922360           0.920807  ...         0.918323   
10           0.944099           0.930124  ...         0.933851   
11           0.936335           0.925466  ...         0.929193   

    std_test_score  rank_test_score  split0_train_score  split1_train_score  \
0         0.008528                5            0.943323            0.940994   
1         0.013080               10            0.910326            0.903339   
2         0.015322               12            0.789208            0.779503   
3         0.012266                3            0.966227            0.966615   
4         0.009491                7            0.937112            0.932453   
5         0.013749               11            0.909938            0.902174   
6         0.010159                4            0.982531            0.979814   
7         0.008482                1            0.950311            0.949534   
8         0.007349                8            0.934006            0.931289   
9         0.005607                9            0.993789            0.992624   
10        0.009033                1            0.966615            0.966227   
11        0.005777                6            0.940606            0.940994   
```

```
    split2_train_score  split3_train_score  split4_train_score  \
0             0.945264            0.937112            0.939829
1             0.908773            0.906056            0.904115
2             0.785326            0.791925            0.788820
3             0.967003            0.961568            0.962345
4             0.936335            0.935171            0.931289
5             0.908773            0.905280            0.902562
6             0.982531            0.982143            0.982531
7             0.948758            0.945652            0.939829
8             0.934006            0.930901            0.929348
9             0.992624            0.993012            0.992236
10            0.966615            0.963121            0.966227
11            0.940994            0.937112            0.937500

    mean_train_score  std_train_score
0           0.941304         0.002814
1           0.906522         0.002672
2           0.786957         0.004277
3           0.964752         0.002308
4           0.934472         0.002242
5           0.905745         0.003158
6           0.981910         0.001059
7           0.946817         0.003835
8           0.931910         0.001831
9           0.992857         0.000527
10          0.965761         0.001331
11          0.939441         0.001753

[12 rows x 22 columns]
```

In [10]:

```
# 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 [11]: # 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_s
```

```
The best test score is 0.9338509316770186 corresponding to hyperparameters {'C': 100, 'gamma':
```

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.

1.5.1 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
```

1.6 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**.