Linear SVM - Email Spam Classifier

October 22, 2020

1 Linear SVM - Email Spam Classifier

In this section, we'll build a linear SVM classifier to classify emails into spam and ham. The dataset, taken from the UCI ML repository, contains about 4600 emails labelled as **spam** or **ham**.

1.1 Data Understanding

Let's first load the data and understand the attributes meanings, shape of the dataset etc.

```
In [30]: import numpy as np
         import pandas as pd
         from sklearn.model_selection import train_test_split
         from sklearn.svm import SVC
         from sklearn.metrics import confusion_matrix
         from sklearn.model_selection import validation_curve
         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
```

```
In [2]: # load the data
        email_rec = pd.read_csv("Spam.txt",  sep = ',', header= None )
        print(email_rec.head())
```

```
     0     1     2    3     4     5     6     7     8     9  ...    48     49  \
0  0.00  0.64  0.64  0.0  0.32  0.00  0.00  0.00  0.00  0.00  ...  0.00  0.000
1  0.21  0.28  0.50  0.0  0.14  0.28  0.21  0.07  0.00  0.94  ...  0.00  0.132
2  0.06  0.00  0.71  0.0  1.23  0.19  0.19  0.12  0.64  0.25  ...  0.01  0.143
3  0.00  0.00  0.00  0.0  0.63  0.00  0.31  0.63  0.31  0.63  ...  0.00  0.137
4  0.00  0.00  0.00  0.0  0.63  0.00  0.31  0.63  0.31  0.63  ...  0.00  0.135

    50     51     52     53     54   55    56  57
0  0.0  0.778  0.000  0.000  3.756   61   278   1
1  0.0  0.372  0.180  0.048  5.114  101  1028   1
2  0.0  0.276  0.184  0.010  9.821  485  2259   1
3  0.0  0.137  0.000  0.000  3.537   40   191   1
```

```
4  0.0  0.135  0.000  0.000  3.537   40   191   1

[5 rows x 58 columns]
```

As of now, the columns are named as integers. Let's manually name the columns appropriately (column names are available at the UCI website here: https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/spambase.names)

```
In [3]: # renaming the columns
        email_rec.columns  = ["word_freq_make", "word_freq_address", "word_freq_all", "word_fr
                              "word_freq_our", "word_freq_over", "word_freq_remove", "word_fre
                              "word_freq_order", "word_freq_mail", "word_freq_receive", "word_
                              "word_freq_people", "word_freq_report", "word_freq_addresses", "
                              "word_freq_business", "word_freq_email", "word_freq_you", "word_
                              "word_freq_your", "word_freq_font", "word_freq_000", "word_freq_
                              "word_freq_hpl", "word_freq_george", "word_freq_650", "word_freq
                              "word_freq_telnet", "word_freq_857", "word_freq_data", "word_fre
                              "word_freq_technology", "word_freq_1999", "word_freq_parts", "wo
                              "word_freq_cs", "word_freq_meeting", "word_freq_original", "word
                              "word_freq_edu", "word_freq_table", "word_freq_conference", "cha
                              "char_freq_[", "char_freq_!", "char_freq_$", "char_freq_hash", "
                              "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]
```

```
In [4]: # look at dimensions of the df
        print(email_rec.shape)
```

```
(4601, 58)
```

```
In [5]: # ensure that data type are correct
        email_rec.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4601 entries, 0 to 4600
Data columns (total 58 columns):
word_freq_make                4601 non-null float64
word_freq_address             4601 non-null float64
word_freq_all                 4601 non-null float64
word_freq_3d                  4601 non-null float64
word_freq_our                 4601 non-null float64
word_freq_over                4601 non-null float64
word_freq_remove              4601 non-null float64
word_freq_internet            4601 non-null float64
word_freq_order               4601 non-null float64
word_freq_mail                4601 non-null float64
word_freq_receive             4601 non-null float64
word_freq_will                4601 non-null float64
word_freq_people              4601 non-null float64
word_freq_report              4601 non-null float64
word_freq_addresses           4601 non-null float64
```

```
word_freq_free              4601 non-null float64
word_freq_business          4601 non-null float64
word_freq_email             4601 non-null float64
word_freq_you               4601 non-null float64
word_freq_credit            4601 non-null float64
word_freq_your              4601 non-null float64
word_freq_font              4601 non-null float64
word_freq_000               4601 non-null float64
word_freq_money             4601 non-null float64
word_freq_hp                4601 non-null float64
word_freq_hpl               4601 non-null float64
word_freq_george            4601 non-null float64
word_freq_650               4601 non-null float64
word_freq_lab               4601 non-null float64
word_freq_labs              4601 non-null float64
word_freq_telnet            4601 non-null float64
word_freq_857               4601 non-null float64
word_freq_data              4601 non-null float64
word_freq_415               4601 non-null float64
word_freq_85                4601 non-null float64
word_freq_technology        4601 non-null float64
word_freq_1999              4601 non-null float64
word_freq_parts             4601 non-null float64
word_freq_pm                4601 non-null float64
word_freq_direct            4601 non-null float64
word_freq_cs                4601 non-null float64
word_freq_meeting           4601 non-null float64
word_freq_original          4601 non-null float64
word_freq_project           4601 non-null float64
word_freq_re                4601 non-null float64
word_freq_edu               4601 non-null float64
word_freq_table             4601 non-null float64
word_freq_conference        4601 non-null float64
char_freq_;                 4601 non-null float64
char_freq_(                 4601 non-null float64
char_freq_[                 4601 non-null float64
char_freq_!                 4601 non-null float64
char_freq_$                 4601 non-null float64
char_freq_hash              4601 non-null float64
capital_run_length_average  4601 non-null float64
capital_run_length_longest  4601 non-null int64
capital_run_length_total    4601 non-null int64
spam                        4601 non-null int64
dtypes: float64(55), int64(3)
memory usage: 2.0 MB
```

```
In [6]: # there are no missing values in the dataset
```

```python
email_rec.isnull().sum()
```

Out[6]:
```
word_freq_make                0
word_freq_address             0
word_freq_all                 0
word_freq_3d                  0
word_freq_our                 0
word_freq_over                0
word_freq_remove              0
word_freq_internet            0
word_freq_order               0
word_freq_mail                0
word_freq_receive             0
word_freq_will                0
word_freq_people              0
word_freq_report              0
word_freq_addresses           0
word_freq_free                0
word_freq_business            0
word_freq_email               0
word_freq_you                 0
word_freq_credit              0
word_freq_your                0
word_freq_font                0
word_freq_000                 0
word_freq_money               0
word_freq_hp                  0
word_freq_hpl                 0
word_freq_george              0
word_freq_650                 0
word_freq_lab                 0
word_freq_labs                0
word_freq_telnet              0
word_freq_857                 0
word_freq_data                0
word_freq_415                 0
word_freq_85                  0
word_freq_technology          0
word_freq_1999                0
word_freq_parts               0
word_freq_pm                  0
word_freq_direct              0
word_freq_cs                  0
word_freq_meeting             0
word_freq_original            0
word_freq_project             0
word_freq_re                  0
word_freq_edu                 0
```

```
word_freq_table             0
word_freq_conference        0
char_freq_;                 0
char_freq_(                 0
char_freq_[                 0
char_freq_!                 0
char_freq_$                 0
char_freq_hash              0
capital_run_length_average  0
capital_run_length_longest  0
capital_run_length_total    0
spam                        0
dtype: int64
```

Let's also look at the fraction of spam and ham emails in the dataset.

```
In [7]: # look at fraction of spam emails
        # 39.4% spams
        email_rec['spam'].describe()
```

```
Out[7]: count    4601.000000
        mean        0.394045
        std         0.488698
        min         0.000000
        25%         0.000000
        50%         0.000000
        75%         1.000000
        max         1.000000
        Name: spam, dtype: float64
```

1.2 Data Preparation

Let's now conduct some prelimininary data preparation steps, i.e. rescaling the variables, splitting into train and test etc. To understand why rescaling is required, let's print the summary stats of all columns - you'll notice that the columns at the end (capital_run_length_longest, capital_run_length_total etc.) have much higher values (means = 52, 283 etc.) than most other columns which represent fraction of word occurrences (no. of times word appears in email/total no. of words in email).

```
In [8]: email_rec.describe()
```

```
Out[8]:        word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
        count     4601.000000        4601.000000    4601.000000   4601.000000
        mean         0.104553           0.213015       0.280656      0.065425
        std          0.305358           1.290575       0.504143      1.395151
        min          0.000000           0.000000       0.000000      0.000000
        25%          0.000000           0.000000       0.000000      0.000000
        50%          0.000000           0.000000       0.000000      0.000000
        75%          0.000000           0.000000       0.420000      0.000000
```

```
max         4.540000          14.280000       5.100000     42.810000

       word_freq_our  word_freq_over  word_freq_remove  word_freq_internet  \
count    4601.000000     4601.000000       4601.000000         4601.000000
mean        0.312223        0.095901          0.114208            0.105295
std         0.672513        0.273824          0.391441            0.401071
min         0.000000        0.000000          0.000000            0.000000
25%         0.000000        0.000000          0.000000            0.000000
50%         0.000000        0.000000          0.000000            0.000000
75%         0.380000        0.000000          0.000000            0.000000
max        10.000000        5.880000          7.270000           11.110000

       word_freq_order  word_freq_mail     ...       char_freq_;  char_freq_(  \
count      4601.000000     4601.000000     ...       4601.000000  4601.000000
mean          0.090067        0.239413     ...          0.038575     0.139030
std           0.278616        0.644755     ...          0.243471     0.270355
min           0.000000        0.000000     ...          0.000000     0.000000
25%           0.000000        0.000000     ...          0.000000     0.000000
50%           0.000000        0.000000     ...          0.000000     0.065000
75%           0.000000        0.160000     ...          0.000000     0.188000
max           5.260000       18.180000     ...          4.385000     9.752000

       char_freq_[  char_freq_!  char_freq_$  char_freq_hash  \
count  4601.000000  4601.000000  4601.000000     4601.000000
mean      0.016976     0.269071     0.075811        0.044238
std       0.109394     0.815672     0.245882        0.429342
min       0.000000     0.000000     0.000000        0.000000
25%       0.000000     0.000000     0.000000        0.000000
50%       0.000000     0.000000     0.000000        0.000000
75%       0.000000     0.315000     0.052000        0.000000
max       4.081000    32.478000     6.003000       19.829000

       capital_run_length_average  capital_run_length_longest  \
count                 4601.000000                 4601.000000
mean                     5.191515                   52.172789
std                     31.729449                  194.891310
min                      1.000000                    1.000000
25%                      1.588000                    6.000000
50%                      2.276000                   15.000000
75%                      3.706000                   43.000000
max                   1102.500000                 9989.000000

       capital_run_length_total         spam
count               4601.000000  4601.000000
mean                 283.289285     0.394045
std                  606.347851     0.488698
min                    1.000000     0.000000
25%                   35.000000     0.000000
```

```
50%                    95.000000     0.000000
75%                   266.000000     1.000000
max                 15841.000000     1.000000

[8 rows x 58 columns]
```

```
In [9]: # splitting into X and y
        X = email_rec.drop("spam", axis = 1)
        y = email_rec.spam.values.astype(int)
```

```
In [10]: # scaling the features
         # note that the scale function standardises each column, i.e.
         # x = x-mean(x)/std(x)

         from sklearn.preprocessing import scale
         X = scale(X)
```

```
In [11]: # split into train and test
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_sta
```

```
In [12]: # confirm that splitting also has similar distribution of spam and ham
         # emails
         print(y_train.mean())
         print(y_test.mean())
```

```
0.3978260869565217
0.38522809558291093
```

1.3 Model Building

Let's build a linear SVM mode now. The `SVC()` class does that in sklearn. We highly recommend reading the documentation at least once.

```
In [13]: help(SVC)
```

```
Help on class SVC in module sklearn.svm.classes:

class SVC(sklearn.svm.base.BaseSVC)
 |  C-Support Vector Classification.
 |
 |  The implementation is based on libsvm. The fit time complexity
 |  is more than quadratic with the number of samples which makes it hard
 |  to scale to dataset with more than a couple of 10000 samples.
 |
 |  The multiclass support is handled according to a one-vs-one scheme.
 |
 |  For details on the precise mathematical formulation of the provided
 |  kernel functions and how `gamma`, `coef0` and `degree` affect each
```

```
|  other, see the corresponding section in the narrative documentation:
|  :ref:`svm_kernels`.
|
|  Read more in the :ref:`User Guide <svm_classification>`.
|
|  Parameters
|  ----------
|  C : float, optional (default=1.0)
|      Penalty parameter C of the error term.
|
|  kernel : string, optional (default='rbf')
|       Specifies the kernel type to be used in the algorithm.
|       It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or
|       a callable.
|       If none is given, 'rbf' will be used. If a callable is given it is
|       used to pre-compute the kernel matrix from data matrices; that matrix
|       should be an array of shape ``(n_samples, n_samples)``.
|
|  degree : int, optional (default=3)
|      Degree of the polynomial kernel function ('poly').
|      Ignored by all other kernels.
|
|  gamma : float, optional (default='auto')
|      Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
|      If gamma is 'auto' then 1/n_features will be used instead.
|
|  coef0 : float, optional (default=0.0)
|      Independent term in kernel function.
|      It is only significant in 'poly' and 'sigmoid'.
|
|  probability : boolean, optional (default=False)
|      Whether to enable probability estimates. This must be enabled prior
|      to calling `fit`, and will slow down that method.
|
|  shrinking : boolean, optional (default=True)
|      Whether to use the shrinking heuristic.
|
|  tol : float, optional (default=1e-3)
|      Tolerance for stopping criterion.
|
|  cache_size : float, optional
|      Specify the size of the kernel cache (in MB).
|
|  class_weight : {dict, 'balanced'}, optional
|      Set the parameter C of class i to class_weight[i]*C for
|      SVC. If not given, all classes are supposed to have
|      weight one.
|      The "balanced" mode uses the values of y to automatically adjust
```

```
 |      weights inversely proportional to class frequencies in the input data
 |      as ``n_samples / (n_classes * np.bincount(y))``
 |
 |  verbose : bool, default: False
 |      Enable verbose output. Note that this setting takes advantage of a
 |      per-process runtime setting in libsvm that, if enabled, may not work
 |      properly in a multithreaded context.
 |
 |  max_iter : int, optional (default=-1)
 |      Hard limit on iterations within solver, or -1 for no limit.
 |
 |  decision_function_shape : 'ovo', 'ovr', default='ovr'
 |      Whether to return a one-vs-rest ('ovr') decision function of shape
 |      (n_samples, n_classes) as all other classifiers, or the original
 |      one-vs-one ('ovo') decision function of libsvm which has shape
 |      (n_samples, n_classes * (n_classes - 1) / 2).
 |
 |      .. versionchanged:: 0.19
 |          decision_function_shape is 'ovr' by default.
 |
 |      .. versionadded:: 0.17
 |         *decision_function_shape='ovr'* is recommended.
 |
 |      .. versionchanged:: 0.17
 |         Deprecated *decision_function_shape='ovo' and None*.
 |
 |  random_state : int, RandomState instance or None, optional (default=None)
 |      The seed of the pseudo random number generator to use when shuffling
 |      the data.  If int, random_state is the seed used by the random number
 |      generator; If RandomState instance, random_state is the random number
 |      generator; If None, the random number generator is the RandomState
 |      instance used by `np.random`.
 |
 |  Attributes
 |  ----------
 |  support_ : array-like, shape = [n_SV]
 |      Indices of support vectors.
 |
 |  support_vectors_ : array-like, shape = [n_SV, n_features]
 |      Support vectors.
 |
 |  n_support_ : array-like, dtype=int32, shape = [n_class]
 |      Number of support vectors for each class.
 |
 |  dual_coef_ : array, shape = [n_class-1, n_SV]
 |      Coefficients of the support vector in the decision function.
 |      For multiclass, coefficient for all 1-vs-1 classifiers.
 |      The layout of the coefficients in the multiclass case is somewhat
```

```
|      non-trivial. See the section about multi-class classification in the
|      SVM section of the User Guide for details.
|
|  coef_ : array, shape = [n_class-1, n_features]
|      Weights assigned to the features (coefficients in the primal
|      problem). This is only available in the case of a linear kernel.
|
|      `coef_` is a readonly property derived from `dual_coef_` and
|      `support_vectors_`.
|
|  intercept_ : array, shape = [n_class * (n_class-1) / 2]
|      Constants in decision function.
|
|  Examples
|  --------
|  >>> import numpy as np
|  >>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
|  >>> y = np.array([1, 1, 2, 2])
|  >>> from sklearn.svm import SVC
|  >>> clf = SVC()
|  >>> clf.fit(X, y) #doctest: +NORMALIZE_WHITESPACE
|  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)
|  >>> print(clf.predict([[-0.8, -1]]))
|  [1]
|
|  See also
|  --------
|  SVR
|      Support Vector Machine for Regression implemented using libsvm.
|
|  LinearSVC
|      Scalable Linear Support Vector Machine for classification
|      implemented using liblinear. Check the See also section of
|      LinearSVC for more comparison element.
|
|  Method resolution order:
|      SVC
|      sklearn.svm.base.BaseSVC
|      abc.NewBase
|      sklearn.svm.base.BaseLibSVM
|      abc.NewBase
|      sklearn.base.BaseEstimator
|      sklearn.base.ClassifierMixin
|      builtins.object
|
```

```
|  Methods defined here:
|
|  __init__(self, C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True, prob
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __abstractmethods__ = frozenset()
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.svm.base.BaseSVC:
|
|  decision_function(self, X)
|      Distance of the samples X to the separating hyperplane.
|
|      Parameters
|      ----------
|      X : array-like, shape (n_samples, n_features)
|
|      Returns
|      -------
|      X : array-like, shape (n_samples, n_classes * (n_classes-1) / 2)
|          Returns the decision function of the sample for each class
|          in the model.
|          If decision_function_shape='ovr', the shape is (n_samples,
|          n_classes)
|
|  predict(self, X)
|      Perform classification on samples in X.
|
|      For an one-class model, +1 or -1 is returned.
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix}, shape (n_samples, n_features)
|          For kernel="precomputed", the expected shape of X is
|          [n_samples_test, n_samples_train]
|
|      Returns
|      -------
|      y_pred : array, shape (n_samples,)
|          Class labels for samples in X.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.svm.base.BaseSVC:
|
|  predict_log_proba
```

```
|      Compute log probabilities of possible outcomes for samples in X.
|
|      The model need to have probability information computed at training
|      time: fit with attribute `probability` set to True.
|
|      Parameters
|      ----------
|      X : array-like, shape (n_samples, n_features)
|          For kernel="precomputed", the expected shape of X is
|          [n_samples_test, n_samples_train]
|
|      Returns
|      -------
|      T : array-like, shape (n_samples, n_classes)
|          Returns the log-probabilities of the sample for each class in
|          the model. The columns correspond to the classes in sorted
|          order, as they appear in the attribute `classes_`.
|
|      Notes
|      -----
|      The probability model is created using cross validation, so
|      the results can be slightly different than those obtained by
|      predict. Also, it will produce meaningless results on very small
|      datasets.
|
|  predict_proba
|      Compute probabilities of possible outcomes for samples in X.
|
|      The model need to have probability information computed at training
|      time: fit with attribute `probability` set to True.
|
|      Parameters
|      ----------
|      X : array-like, shape (n_samples, n_features)
|          For kernel="precomputed", the expected shape of X is
|          [n_samples_test, n_samples_train]
|
|      Returns
|      -------
|      T : array-like, shape (n_samples, n_classes)
|          Returns the probability of the sample for each class in
|          the model. The columns correspond to the classes in sorted
|          order, as they appear in the attribute `classes_`.
|
|      Notes
|      -----
|      The probability model is created using cross validation, so
|      the results can be slightly different than those obtained by
```

```
 |      predict. Also, it will produce meaningless results on very small
 |      datasets.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.svm.base.BaseLibSVM:
 |
 |  fit(self, X, y, sample_weight=None)
 |      Fit the SVM model according to the given training data.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix}, shape (n_samples, n_features)
 |          Training vectors, where n_samples is the number of samples
 |          and n_features is the number of features.
 |          For kernel="precomputed", the expected shape of X is
 |          (n_samples, n_samples).
 |
 |      y : array-like, shape (n_samples,)
 |          Target values (class labels in classification, real numbers in
 |          regression)
 |
 |      sample_weight : array-like, shape (n_samples,)
 |          Per-sample weights. Rescale C per sample. Higher weights
 |          force the classifier to put more emphasis on these points.
 |
 |      Returns
 |      -------
 |      self : object
 |          Returns self.
 |
 |      Notes
 |      ------
 |      If X and y are not C-ordered and contiguous arrays of np.float64 and
 |      X is not a scipy.sparse.csr_matrix, X and/or y may be copied.
 |
 |      If X is a dense array, then the other methods will not support sparse
 |      matrices as input.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.svm.base.BaseLibSVM:
 |
 |  coef_
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
```

```
 |  __repr__(self)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : boolean, optional
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |
 |      Returns
 |      -------
 |      self
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.BaseEstimator:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.ClassifierMixin:
 |
 |  score(self, X, y, sample_weight=None)
 |      Returns the mean accuracy on the given test data and labels.
 |
 |      In multi-label classification, this is the subset accuracy
 |      which is a harsh metric since you require for each sample that
```

```
 |      each label set be correctly predicted.
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape = (n_samples, n_features)
 |          Test samples.
 |
 |      y : array-like, shape = (n_samples) or (n_samples, n_outputs)
 |          True labels for X.
 |
 |      sample_weight : array-like, shape = [n_samples], optional
 |          Sample weights.
 |
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of self.predict(X) wrt. y.
```

In [14]:
```python
# Model building

# instantiate an object of class SVC()
# note that we are using cost C=1
model = SVC(C = 1)

# fit
model.fit(X_train, y_train)

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

In [15]:
```python
# Evaluate the model using confusion matrix
from sklearn import metrics
metrics.confusion_matrix(y_true=y_test, y_pred=y_pred)
```

Out[15]:
```
array([[811,  38],
       [ 61, 471]])
```

In [16]:
```python
# print other metrics

# 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
```

```
In [17]: # specificity (% of hams correctly classified)
         print("specificity", 811/(811+38))
```

```
specificity 0.9552414605418139
```

The SVM we have built so far gives decently good results - an accuracy of 92%, sensitivity/recall (TNR) of 88%.

1.3.1 Interpretation of Results

In the confusion matrix, the elements at (0, 0) and (1,1) correspond to the more frequently occurring class, i.e. ham emails. Thus, it implies that: - 92% of all emails are classified correctly - 88.5% of spams are identified correctly (sensitivity/recall) - Specificity, or % of hams classified correctly, is 95%

1.4 Hyperparameter Tuning

```
In [18]: help(metrics.confusion_matrix)
```

```
Help on function confusion_matrix in module sklearn.metrics.classification:

confusion_matrix(y_true, y_pred, labels=None, sample_weight=None)
    Compute confusion matrix to evaluate the accuracy of a classification

    By definition a confusion matrix :math:`C` is such that :math:`C_{i, j}`
    is equal to the number of observations known to be in group :math:`i` but
    predicted to be in group :math:`j`.

    Thus in binary classification, the count of true negatives is
    :math:`C_{0,0}`, false negatives is :math:`C_{1,0}`, true positives is
    :math:`C_{1,1}` and false positives is :math:`C_{0,1}`.

    Read more in the :ref:`User Guide <confusion_matrix>`.

    Parameters
    ----------
    y_true : array, shape = [n_samples]
        Ground truth (correct) target values.

    y_pred : array, shape = [n_samples]
        Estimated targets as returned by a classifier.

    labels : array, shape = [n_classes], optional
```

```
    List of labels to index the matrix. This may be used to reorder
    or select a subset of labels.
    If none is given, those that appear at least once
    in ``y_true`` or ``y_pred`` are used in sorted order.

sample_weight : array-like of shape = [n_samples], optional
    Sample weights.

Returns
-------
C : array, shape = [n_classes, n_classes]
    Confusion matrix

References
----------
.. [1] `Wikipedia entry for the Confusion matrix
       <https://en.wikipedia.org/wiki/Confusion_matrix>`_

Examples
--------
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])

>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])

In the binary case, we can extract true positives, etc as follows:

>>> tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel()
>>> (tn, fp, fn, tp)
(0, 2, 1, 1)
```

1.4.1 K-Fold Cross Validation

Let's first run a simple k-fold cross validation to get a sense of the **average metrics** as computed over multiple *folds*. the easiest way to do cross-validation is to use the `cross_val_score()` function.

```
In [19]: # creating a KFold object with 5 splits
         folds = KFold(n_splits = 5, shuffle = True, random_state = 4)

         # instantiating a model with cost=1
         model = SVC(C = 1)

In [20]: # computing the cross-validation scores
         # note that the argument cv takes the 'folds' object, and
         # we have specified 'accuracy' as the metric

         cv_results = cross_val_score(model, X_train, y_train, cv = folds, scoring = 'accuracy

In [21]: # print 5 accuracies obtained from the 5 folds
         print(cv_results)
         print("mean accuracy = {}".format(cv_results.mean()))

[0.91770186 0.93944099 0.91925466 0.93012422 0.94254658]
mean accuracy = 0.9298136645962731
```

1.5 Grid Search to Find Optimal Hyperparameter C

K-fold CV helps us compute average metrics over multiple folds, and that is the best indication of the 'test accuracy/other metric scores' we can have.

But we want to use CV to compute the optimal values of hyperparameters (in this case, the cost C is a hyperparameter). This is done using the `GridSearchCV()` method, which computes metrics (such as accuracy, recall etc.)

In this case, we have only one hyperparameter, though you can have multiple, such as C and gamma in non-linear SVMs. In that case, you need to search through a *grid* of multiple values of C and gamma to find the optimal combination, and hence the name GridSearchCV.

```
In [22]: # specify range of parameters (C) as a list
         params = {"C": [0.1, 1, 10, 100, 1000]}

         model = SVC()

         # set up grid search scheme
         # note that we are still using the 5 fold CV scheme we set up earlier
         model_cv = GridSearchCV(estimator = model, param_grid = params,
                                  scoring= 'accuracy',
                                  cv = folds,
                                  verbose = 1,
                                 return_train_score=True)

In [23]: # fit the model - it will fit 5 folds across all values of C
         model_cv.fit(X_train, y_train)

Fitting 5 folds for each of 5 candidates, totalling 25 fits
```

```
[Parallel(n_jobs=1)]: Done  25 out of  25 | elapsed:   10.5s finished
```

Out[23]:
```
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={'C': [0.1, 1, 10, 100, 1000]}, pre_dispatch='2*n_jobs',
       refit=True, return_train_score=True, scoring='accuracy', verbose=1)
```

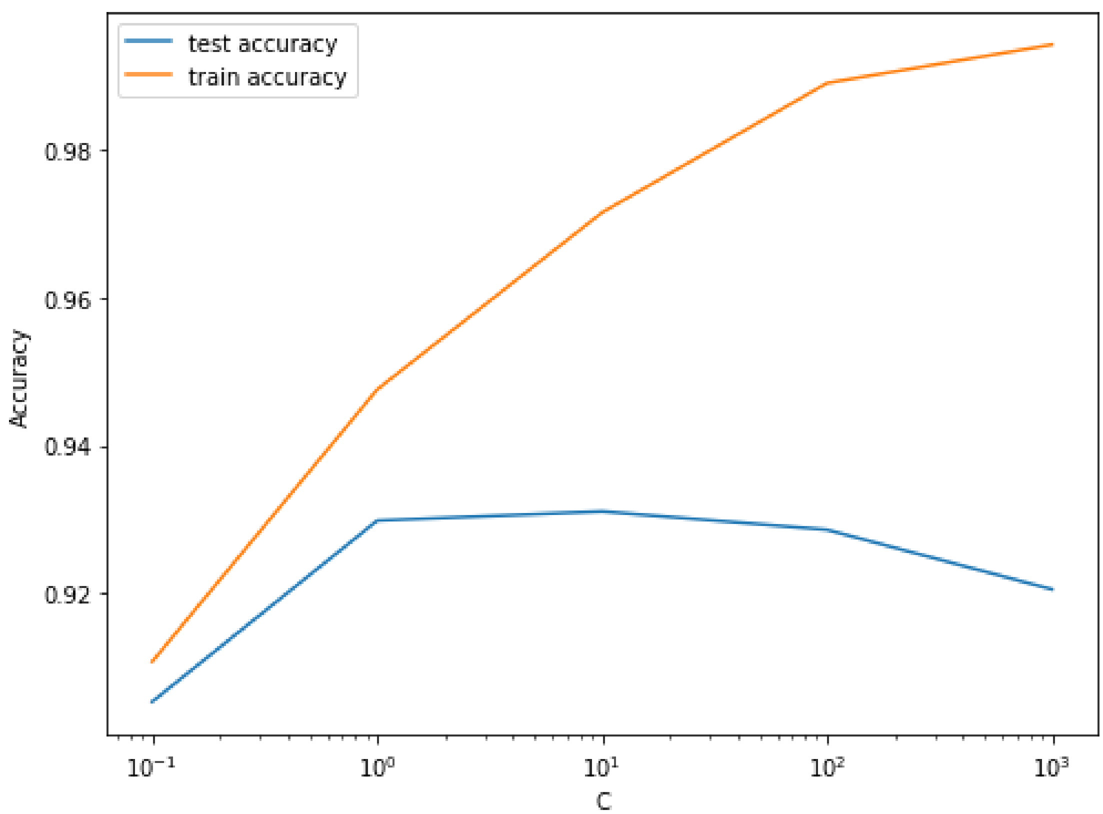
In [24]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[24]:
```
   mean_fit_time  mean_score_time  mean_test_score  mean_train_score param_C  \
0       0.297553         0.063188         0.905280          0.910714     0.1
1       0.197679         0.037653         0.929814          0.947593       1
2       0.179199         0.030673         0.931056          0.971584      10
3       0.219374         0.026815         0.928571          0.989130     100
4       0.297710         0.024977         0.920497          0.994332    1000

        params  rank_test_score  split0_test_score  split0_train_score  \
0   {'C': 0.1}                5           0.895963            0.912267
1     {'C': 1}                2           0.917702            0.951863
2    {'C': 10}                1           0.909938            0.973991
3   {'C': 100}                3           0.914596            0.989519
4  {'C': 1000}                4           0.908385            0.996118

   split1_test_score       ...         split2_test_score  split2_train_score  \
0           0.900621       ...                  0.906832            0.912267
1           0.939441       ...                  0.919255            0.950699
2           0.944099       ...                  0.933230            0.973602
3           0.925466       ...                  0.936335            0.989907
4           0.931677       ...                  0.923913            0.994177

   split3_test_score  split3_train_score  split4_test_score  \
0           0.902174            0.911491           0.920807
1           0.930124            0.946040           0.942547
2           0.928571            0.968944           0.939441
3           0.930124            0.988354           0.936335
4           0.919255            0.993789           0.919255

   split4_train_score  std_fit_time  std_score_time  std_test_score  \
0            0.906056      0.003776        0.001645        0.008505
1            0.943711      0.005545        0.000852        0.010130
```

```
2           0.970885      0.004506        0.001983        0.011809
3           0.988354      0.014443        0.001334        0.008098
4           0.993789      0.037398        0.002470        0.007569

   std_train_score
0         0.002355
1         0.003135
2         0.001924
3         0.000650
4         0.000905

[5 rows x 21 columns]
```

To get a better sense of how training and test accuracy varies with C, let's plot the tranining and test accuracies against C.

In [25]:
```python
# plot of C versus train and test scores

plt.figure(figsize=(8, 6))
plt.plot(cv_results['param_C'], cv_results['mean_test_score'])
plt.plot(cv_results['param_C'], cv_results['mean_train_score'])
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.legend(['test accuracy', 'train accuracy'], loc='upper left')
plt.xscale('log')
```



Though the training accuracy monotonically increases with C, the test accuracy gradually reduces. Thus, we can conclude that higher values of C tend to **overfit** the model. This is because a high C value aims to classify all training examples correctly (since C is the *cost of misclassification* - if you impose a high cost on the model, it will avoid misclassifying any points by overfitting the data).

Let's finally look at the optimal C values found by GridSearchCV.

```
In [26]: best_score = model_cv.best_score_
         best_C = model_cv.best_params_['C']

         print(" The highest test accuracy is {0} at C = {1}".format(best_score, best_C))
```

```
 The highest test accuracy is 0.931055900621118 at C = 10
```

Let's now look at the metrics corresponding to C=10.

```
In [27]: # model with the best value of C
         model = SVC(C=best_C)

         # fit
         model.fit(X_train, y_train)

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

```
In [28]: # metrics
         # print other metrics

         # 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 0.9304851556842868
precision 0.9241245136186771
recall 0.8928571428571429
```

1.6 Optimising for Other Evaluation Metrics

In this case, we had optimised (tuned) the model based on overall accuracy, though that may not always be the best metric to optimise. For example, if you are concerned more about catching all spams (positives), you may want to maximise TPR or sensitivity/recall. If, on the other hand, you want to avoid classifying hams as spams (so that any important mails don't get into the spam box), you would maximise the TNR or specificity.

```
In [29]: # specify params
         params = {"C": [0.1, 1, 10, 100, 1000]}

         # specify scores/metrics in an iterable
         scores = ['accuracy', 'precision', 'recall']

         for score in scores:
             print("# Tuning hyper-parameters for {}".format(score))

             # set up GridSearch for score metric
             clf = GridSearchCV(SVC(),
                                params,
                                cv=folds,
                                scoring=score,
                                return_train_score=True)
             # fit
             clf.fit(X_train, y_train)

             print(" The highest {0} score is {1} at C = {2}".format(score, clf.best_score_, c
             print("\n")
```

```
# Tuning hyper-parameters for accuracy
 The highest accuracy score is 0.931055900621118 at C = {'C': 10}


# Tuning hyper-parameters for precision
 The highest precision score is 0.936509856470386 at C = {'C': 0.1}


# Tuning hyper-parameters for recall
 The highest recall score is 0.8994650196111064 at C = {'C': 10}
```

Thus, you can see that the optimal value of the hyperparameter varies significantly with the choice of evaluation metric.