CS3920/CS5920 Lab Worksheet 6: In support of Assignment 2

Volodymyr Vovk

November 3, 2022

This lab worksheet should be completed during the lab session of 7 November and your independent study time. There are no exercises (Assignment 2 can be considered to be a set of exercises for this lab worksheet).

The topics that are covered in this worksheet are:

- Data normalization.
- Parameter selection.

For further details of the functions used in this worksheet, see [1, Chapters 3 and 5] and [2].

If you are getting results that are not exactly the same as given below, usually there is no need to worry: you may be using a different version of `scikit-learn`.

1 Normalization

Let's see how normalization works in `scikit-learn`. In this section we will use a new dataset, the Wisconsin Breast Cancer dataset (`cancer`, for short), which records clinical measurements of breast cancer tumors. Each tumor is labeled as "benign" (for harmless tumors) or "malignant" (for cancerous tumors), and the task is to learn to predict whether a tumor is malignant based on the measurements of the tissue. We will see how normalization affects the performance of the support vector machine (SVM) on this dataset. We will study SVM in detail in Chapter 8, and here will use it as a black box. Do not worry about this; in practice, you will often need to explore the performance of all kinds of black boxes (before deciding whether it makes sense for you to spend your time studying them in detail).

We start by loading our dataset and splitting it into a training set and a test set:

```
In[1]:
  from sklearn.datasets import load_breast_cancer
  from sklearn.model_selection import train_test_split
  cancer = load_breast_cancer()
  X_train, X_test, y_train, y_test = train_test_split(cancer.data,
    cancer.target, random_state=0)
  print(X_train.shape)
  print(X_test.shape)

  (426, 30)
  (143, 30)
```

The dataset contains 569 data points, each represented by 30 measurements. We split the dataset into 426 samples for the training set and 143 samples for the test set.

To do preprocessing, we first import the class that implements our chosen method (`MinMaxScaler`) and then instantiate it:

```
In[2]:
  from sklearn.preprocessing import MinMaxScaler
  scaler = MinMaxScaler()
```

We then fit the scaler using the `fit` method, applied to the training data only. For the `MinMaxScaler`, the `fit` method computes the minimum and maximum value of each feature on the training set. It is only provided with `X_train`, and `y_train` is not used:

```
In[3]:
  scaler.fit(X_train)
```

To apply the transformation that we just learned—that is, to actually scale the data—we use the `transform` method of the scaler.

```
In[4]:
  # transform data
  X_train_scaled = scaler.transform(X_train)
  # print dataset properties before and after scaling
  print(X_train_scaled.shape)
  print(X_train.min(axis=0))
  print(X_train.max(axis=0))
  print(X_train_scaled.min(axis=0))
  print(X_train_scaled.max(axis=0))
```

You can see that the transformed data has the same shape as the original data—the features are simply shifted and scaled. The method `min` used in `X_train.min(axis=0)` computes the minimum of the entries in the matrix `X_train` over the axis 0 (which is the rows; for each fixed column you compute the minimum over the rows). You can see that all of the features are now between 0 and 1, as desired.

To apply the SVM to the scaled data, we also need to transform the test set. This is again done by calling the `transform` method, this time on `X_test`:

```
In[5]:
  # transform test data
  X_test_scaled = scaler.transform(X_test)
```

```
# print test data properties after scaling
print(X_test_scaled.min(axis=0))
print(X_test_scaled.max(axis=0))
```

You can see that for the test set, after scaling, the minimum and maximum are not 0 and 1. The explanation is that the `MinMaxScaler` (and all the other scalers) always applies exactly the same transformation to the training and the test set. This means the `transform` method always subtracts the training set minimum and divides by the training set range, which might be different from the minimum and range for the test set.

Now let's see the effect of using the `MinMaxScaler` on learning the SVM. First, let's fit the SVM on the original data for comparison:

```
In[6]:
  from sklearn.svm import SVC
  X_train, X_test, y_train, y_test = train_test_split(cancer.data,
    cancer.target, random_state=0)
  svm = SVC(C=100)
  svm.fit(X_train, y_train)
  print(svm.score(X_test, y_test))

  0.944
```

The interface for the SVM is the standard one and uses the class `SVC` of the module `svm`. When creating an SVM, we specify `C=100`; as almost any machine-learning algorithm, SVM has several parameters to select, and `C` is among the most important ones; selecting parameters will be the topic of the next section.

Now, let's apply SVM to the data normalized by `MinMaxScaler`:

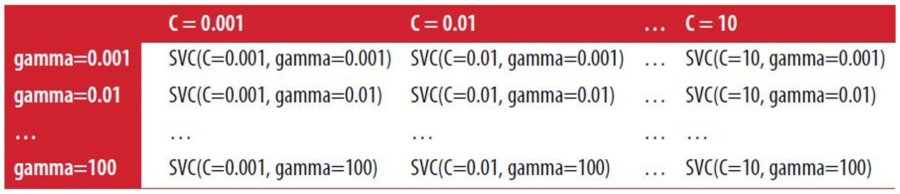
```
In[7]:
  X_test_scaled = scaler.transform(X_test)
  # learning an SVM on the scaled training data:
  svm.fit(X_train_scaled, y_train)
  # scoring on the scaled test set:
  print(svm.score(X_test_scaled, y_test))

  0.965
```

The effect of scaling the data is quite significant; data normalization really works in this case.

2 Parameter selection using a validation set and cross-validation

The two important parameters of SVM, as implemented in the SVC class, are the "kernel bandwidth" `gamma` and the "regularization parameter" `C` (which you have already seen in the previous section; it is akin to the regularization parameter α as used in Ridge Regression and Lasso). Suppose we want to try the values 0.001, 0.01, 0.1, 1, 10, and 100 for `C`, and the same for `gamma`.

Because we have 6 different settings for `C` and `gamma` that we want to try, we have 36 combinations of parameters in total. Looking at all possible combinations creates a table (or grid) of parameter settings for the SVM, like this one (but ending with $C = 100$):



	C = 0.001	C = 0.01	…	C = 10
gamma=0.001	SVC(C=0.001, gamma=0.001)	SVC(C=0.01, gamma=0.001)	…	SVC(C=10, gamma=0.001)
gamma=0.01	SVC(C=0.001, gamma=0.01)	SVC(C=0.01, gamma=0.01)	…	SVC(C=10, gamma=0.01)
…	…	…	…	…
gamma=100	SVC(C=0.001, gamma=100)	SVC(C=0.01, gamma=100)	…	SVC(C=10, gamma=100)

Let's now use the `iris` dataset. The naive implementation of grid search as nested `for` loops over the two parameters, training and evaluating a classifier for each combination, is:

```
In[8]:
  from sklearn.datasets import load_iris
  iris = load_iris()
  # naive grid search implementation
  X_train, X_test, y_train, y_test = train_test_split(iris.data,
    iris.target, random_state=0)
  best_score = 0
  for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
      # for each combination of parameters, train an SVM
      svm = SVC(gamma=gamma, C=C)
      svm.fit(X_train, y_train)
      # evaluate the SVM on the test set
      score = svm.score(X_test, y_test)
      # if we got a better score, store the score and parameters
      if score > best_score:
        best_score = score
        best_C = C
        best_gamma = gamma
  print("Best score:", best_score)
  print("Best parameters C and gamma:", best_C, best_gamma)

  Best score: 0.973684
  Best parameters C and gamma: 100 0.001
```

As discussed in the lectures, we should resist the temptation to report that we found a model that performs with 97% accuracy on our dataset; remember that we have tried 36 combinations of parameters, and one of them might have been very good just by accident.

Using a validation set

Let's repeat what we did, but this time using a separate validation set:

```
In[9]:
```

```
# split data into train set and test sets
X_train, X_test, y_train, y_test = train_test_split(iris.data,
  iris.target, random_state=0)
# split train set into train set proper and validation set
X_train_pr, X_valid, y_train_pr, y_valid = train_test_split(X_train,
  y_train, random_state=1)
print("Sizes of train_pr, valid, and test sets:",
  X_train_pr.shape[0], X_valid.shape[0], X_test.shape[0])
best_score = 0
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
  for C in [0.001, 0.01, 0.1, 1, 10, 100]:
    # for each combination of parameters, train an SVM
    svm = SVC(gamma=gamma, C=C)
    svm.fit(X_train_pr, y_train_pr)
    # evaluate the SVM on the validation set
    score = svm.score(X_valid, y_valid)
    # if we got a better score, store the score and parameters
    if score > best_score:
      best_score = score
      best_C = C
      best_gamma = gamma
# rebuild a model on the full training set,
# and evaluate it on the test set
svm = SVC(C=best_C, gamma=best_gamma)
svm.fit(X_train, y_train)
test_score = svm.score(X_test, y_test)
print("Best score on validation set:", best_score)
print("Best parameters C and gamma:", best_C, best_gamma)
print("Test set score with best parameters:", test_score)

Sizes of train_pr, valid, and test sets: 84 28 38
Best score on validation set: 0.964286
Best parameters C and gamma: 10 0.001
Test set score with best parameters: 0.921053
```

The best score on the validation set is 96%, slightly lower than before (probably because we used less data to train the model). However, the score on the test set—the score that actually tells us how well we generalize—is even lower, at 92%. So we can only claim to classify new data 92% correctly, not 97% correctly as we thought before!

The conclusion of the last paragraph is natural (and is actually made in [1, p. 265]), but you should be careful: replace the second `random_state` of 1 in the above snippet by 0, and your results will be very different.

Using cross-validation

Notice that grid search using a validation set selected `C` = 10 and `gamma` = 0.001 as the best parameters, while using the test set selected `C` = 100 and `gamma` = 0.001 as the best parameters.

```
In[10]:
  for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
      # for each combination of parameters, train an SVC
```

```
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_train, y_train, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
        # if we got a better score, store the score and parameters
        if score > best_score:
          best_score = score
          best_C = C
          best_gamma = gamma
  # rebuild a model on the full training set
  svm = SVC(C=best_C, gamma=best_gamma)
  svm.fit(X_train, y_train)
  test_score = svm.score(X_test, y_test)
  print("Best CV score:", best_score)
  print("Best parameters C and gamma:", best_C, best_gamma)
  print("Test set score with best parameters:", test_score)

  Best CV score: 0.972690
  Best parameters C and gamma: 100 0.01
  Test set score with best parameters: 0.973684
```

Correct syntactic errors if there are any. The test set score seems to improve as compared to using a validation set but you should at least try different `random_state`s before drawing conclusions.

To evaluate the accuracy of the SVM using a particular setting of `C` and `gamma` using 5-fold cross-validation, we need to train $36 \times 5 = 180$ models. The main downside of the use of cross-validation is the time it takes to train all these models.

The module `scikit-learn` provides the `GridSearchCV` class, which implements grid search with cross-validation. To use the `GridSearchCV` class, you first need to specify the parameters you want to search over using a dictionary. (Dictionaries will be covered in CS5800 shortly, but to use `GridSearchCV` you do not need to know the details of this data type in Python. You just need to know that each key, from a finite set of keys, is mapped to a value.) `GridSearchCV` will then perform all the necessary model fits. The keys of the dictionary are the names of parameters we want to adjust (in this case, `C` and `gamma`), and the values are the parameter settings we want to try out. Trying the values 0.001, 0.01, 0.1, 1, 10, and 100 for `C` and `gamma` translates to the following dictionary:

```
In[11]:
  param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

We can now instantiate the `GridSearchCV` class with the model (`SVC`), the parameter grid to search (`param_grid`), and the number of folds in cross-validation:

```
In[12]:
  from sklearn.model_selection import GridSearchCV
  grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

`GridSearchCV` will use cross-validation in place of the split into a training set proper and a validation set that we used before. However, we still need to split the data into a training and a test set:

```
In[13]:
  X_train, X_test, y_train, y_test = train_test_split(iris.data,
    iris.target, random_state=0)
```

The `grid_search` object that we created behaves just like a classifier; we can call the standard methods `fit`, `predict`, and `score` on it. However, when we call `fit`, it will run cross-validation for each combination of parameters we specified in `param_grid`:

```
In[14]:
  grid_search.fit(X_train, y_train)
```

Fitting the `GridSearchCV` object not only searches for the best parameters, but also automatically fits a new model on the whole training dataset with the parameters that yielded the best cross-validation performance. What happens in `fit` is therefore equivalent to the result of the `In[10]` code. The `GridSearchCV` class provides a convenient interface to access the retrained model using the `predict` and `score` methods. To evaluate how well the best found parameters generalize, we can call `score` on the test set:

```
In[15]:
  grid_search.score(X_test, y_test)
Out[15]:
  0.973684
```

Choosing the parameters using cross-validation, we actually found a model that achieves 97% accuracy on the test set. It is important that we did not use the test set to choose the parameters. The parameters that were found are stored in the `best_params_` attribute, and the best cross-validation accuracy (the mean accuracy over the different folds for this parameter setting) is stored in `best_score_`:

```
In[16]:
  print(grid_search.best_params_)
  print(grid_search.best_score_)

  {'C': 100, 'gamma': 0.01}
  0.97
```

Sometimes (as in Assignment 2) it is helpful to have access to the actual model that was found (e.g., to look at its coefficients in the case of a linear model such as `Lasso`). You can access the model with the best parameters trained on the whole training set using the `best_estimator_` attribute:

```
  grid_search.best_estimator_
```

is the best model. But because `grid_search` itself has `predict` and `score` methods, using `best_estimator_` is rarely needed; in particular, it is not needed to make predictions or evaluate the model.

References

[1] Andreas C. Müller and Sarah Guido. *An introduction to machine learning with Python.* O'Reilly, Beijing, 2017.

[2] `scikit-learn` tutorials. `http://scikit-learn.org/stable/tutorial/`, 2007–2022.