CS3920/CS5920 Lab Worksheet 5: Linear models in `scikit-learn`

Volodymyr Vovk

October 28, 2022

This lab worksheet should be completed during the lab session of 31 October and your independent study time.

The topics that are briefly covered in this worksheet are:

- Least squares.
- Ridge regression.
- Lasso.

For further details of some functions used in this worksheet, see [1, Chapter 2] 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 Installing and importing `mglearn`

The library `mglearn` is written for the main textbook for this module, [1] (see [1, Chapter 1, p. 11]). You can find it at

`https://github.com/amueller/introduction_to_ml_with_python`

Place the folder `mglearn` in your home directory. (It might be easiest to download the whole GitHub book site by clicking on the green button "Code" and then choosing "Download ZIP".) Run

```
In[1]:
  import mglearn
```

Now you can use `mglearn`.

If for some reason you can't install the library, please ask for our help, and if nothing works, switch to the version of this worksheet that does not require `mglearn`.

2 Least squares

First we plot the figure you saw in the lectures:

```
In[2]:
  mglearn.plots.plot_linear_regression_wave()
Out[2]:
  w[0]: 0.393906  b: -0.031804
  [the picture]
```

Do the estimated coefficients `w[0]` and `b` look plausible to you?

Now we can create the `Wave` dataset (have a look at the code in the `mglearn` folder) and fit a linear model to it:

```
In[3]:
  from sklearn.linear_model import LinearRegression
  X,y = mglearn.datasets.make_wave(n_samples=60)
  X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=0)
  lr = LinearRegression().fit(X_train,y_train)
```

The estimated coefficients are (as we already know):

```
In[4]:
  lr.coef_
Out[4]:
  array([0.393906])
In[5]:
  lr.intercept_
Out[5]:
  -0.031804
```

The trailing underscores at the end of `coef_` and `intercept_` signal that these are derived from the training data (rather than being set by the user).

Let's look at the training set and test set performance:

```
In[6]:
  lr.score(X_train, y_train)
Out[6]:
  0.670089
In[7]:
  lr.score(X_test, y_test)
Out[7]:
  0.659337
```

Remember that the performance of a regressor is evaluated using R^2, as discussed in Lab 4 and Chapter 5. There is no overfitting (our model is too primitive for that!).

Let us now engineer another feature in the `wave` dataset, X^2:

```
In[8]:
  import numpy as np
  X_train_ext = np.concatenate((X_train,X_train**2),axis=1)
  X_test_ext = np.concatenate((X_test,X_test**2),axis=1)
```

The `NumPy` function `concatenate` joins the original feature X (a column vector) with X^2 (also a column vector, namely X with all its elements squared) along axis 1 (axis 0 runs along the rows and axis 1 runs along the columns). If we now fit `LinearRegression` to the extended training data `X_train_ext` and apply the resulting model to the extended test data `X_test_ext`, we can hardly notice the curvature:

```
In[9]:
  lr = LinearRegression().fit(X_train_ext, y_train)
  y_hat = lr.predict(X_test_ext)
  %matplotlib inline
  import matplotlib.pyplot as plt
  plt.scatter(X_test, y_hat)
```

There is still little overfitting:

```
In[10]:
  print(lr.score(X_train_ext,y_train))
  print(lr.score(X_test_ext,y_test))

  0.660182
  0.686192
```

To get a clear instance of overfitting, we will try a more complex dataset, namely the `diabetes` dataset introduced in Lab 4. To make the linear model more powerful, we will expand this dataset by using not only the 10 original features that you saw in Lab 4 (such as age and body mass index), but also looking at all products (also called interactions) between features. In other words, we will not only consider age and body mass index as features, but also the product of age and body mass index. This derived dataset can be created using the `PolynomialFeatures` function available in the module `preprocessing`:

```
In[11]:
  from sklearn.datasets import load_diabetes
  diabetes = load_diabetes()
  diabetes.data.shape
Out[11]:
  (442, 10)
In [12]:
  from sklearn.preprocessing import PolynomialFeatures
  X = PolynomialFeatures(degree=2,
    include_bias=False).fit_transform(diabetes.data)
  y = diabetes.target
  X.shape
Out[12]:
  (442, 65)
```

The resulting 55 new features are the 10 original features make up 65 features in the extended data set.

Remark. If you are curious where 65 is coming from (it's safe to ignore this remark if you are not!), 65 is the number of pairs $(\text{feature}_1, \text{feature}_2)$ for $\text{feature}_1 \neq$

feature_2, plus the number of squares feature^2, plus the number of original features:

$$\binom{10}{2} + 10 + 10 = 10 \times 9/2 + 10 + 10 = 65.$$

The term $\binom{n}{k}$ ("n choose k", in this case "10 choose 2") is the binomial coefficient, which is the number of combinations of k elements that can be selected from a set of n elements.

After loading the dataset and splitting it into a training and a test set, we build the linear regression model as before:

```
In[13]:
  X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=42)
  lr = LinearRegression().fit(X_train,y_train)
```

When comparing training set and test set scores, we find that we predict very accurately on the training set, but the R^2 on the test set is much worse:

```
In[14]:
  print(lr.score(X_train, y_train))
  print(lr.score(X_test, y_test))

  0.605
  0.424
```

This discrepancy between performance on the training set and the test set is a clear sign of overfitting, and therefore we should try to find a model that allows us to control complexity. In this module we discuss two solutions: ridge regression and lasso.

3 Ridge regression

Ridge regression is implemented in `linear_model.Ridge`.

Let's see how well it does on the extended `diabetes` dataset:

```
In[15]:
  from sklearn.linear_model import Ridge
  ridge = Ridge().fit(X_train,y_train)
  ridge.score(X_train,y_train)
Out[15]:
  0.428
In[16]:
  ridge.score(X_test,y_test)
Out[16]:
  0.439
```

The training set score of Ridge is lower than for `LinearRegression`, while the test set score is higher. We are no longer overfitting our data (the test score is even better than the training score, which is a statistical fluke). As we are only

interested in test set performance, we should choose the `Ridge` model over the `LinearRegression` model.

The default regularization parameter is $\alpha = 1$ (used implicitly in `In[15]` and `In[16]`). There is no reason why this will give us the best trade-off, though; the optimum α depends on the dataset. For example:

```
In[17]:
  ridge10 = Ridge(alpha=10).fit(X_train,y_train)
  ridge10.score(X_train,y_train)
Out[17]:
  0.151
In[18]:
  ridge10.score(X_test,y_test)
Out[18]:
  0.156
```

In this case the test set score deteriorates, but you can check that setting $\alpha = 0.1$ improves it. (The choice of parameters such as α is one of the topics of Chapter 6.)

Let's see how α changes the model by inspecting the `coef_` attribute of models with different values of `alpha`.

```
In[19]:
  plt.plot(lr.coef_, 'o', label="Least Squares")
  plt.plot(ridge01.coef_, '^', label="Ridge alpha=0.1")
  plt.plot(ridge.coef_,'v',label="alpha=1")
  plt.plot(ridge10.coef_,'s',label="alpha=10")
  plt.xlabel("Coefficient index")
  plt.ylabel("Coefficient magnitude")
  plt.hlines(0,0,len(lr.coef_))
  plt.ylim(-100,100)
  plt.legend()
```

See Exercise 1.

4 Lasso

Let's apply the lasso to the extended `diabetes` dataset:

```
In[20]:
  from sklearn.linear_model import Lasso
  lasso = Lasso().fit(X_train,y_train)
  lasso.score(X_train,y_train)
Out[20]:
  0.347
In[21]:
  lasso.score(X_test,y_test)
Out[21]:
  0.379
In[22]:
  import numpy as np
  np.sum(lasso.coef_ != 0)
```

```
Out[22]:
  3
```

Lasso works poorly both on the training and the test set. We are underfitting, and 3 of the 65 features is clearly not enough. In the previous cells, we used the default of $\alpha = 1.0$. To reduce underfitting, let's decrease α. When we do this, we also need to increase the default setting of `max_iter` (the maximum number of iterations to run):

```
In[23]:
  # we increase the default setting of "max_iter",
  # otherwise the model would warn us that we should increase max_iter.
  lasso001 = Lasso(alpha=0.01,max_iter=100000).fit(X_train,y_train)
  lasso001.score(X_train,y_train)
Out[23]:
  0.538
In[24]:
  lasso001.score(X_test,y_test)
Out[24]:
  0.514
In[25]:
  np.sum(lasso001.coef_ != 0)
Out[25]:
  16
```

The performance is slightly better than using `Ridge`, and we are using only 16 of the 65 features. This makes this model potentially easier to understand.

Remark. One difficulty with `scikit-learn` is that the regularization parameter α in Lasso is different from the one in Ridge Regression (and the one in Chapter 5). The objective function in `scikit-learn` is

$$\frac{\text{RSS}}{2n} + \alpha \sum_{j=0}^{p-1} |w[j]|\,,$$

where n is the size of the training set.

We can set α too low, which leads to overfitting:

```
In[26]:
  lasso00001 = Lasso(alpha=0.0001,max_iter=100000).fit(X_train,y_train)
  lasso00001.score(X_train,y_train)
Out[26]:
  0.601
In[27]:
  lasso00001.score(X_test,y_test)
Out[27]:
  0.448
In[28]:
  np.sum(lasso00001.coef_ != 0)
Out[28]:
  55
```

Again, we can plot the coefficients of the different models:

```
In[29]:
  plt.plot(lasso00001.coef_, '^', label="Lasso alpha=0.0001")
  plt.plot(lasso001.coef_, 'v', label="Lasso alpha=0.01")
  plt.plot(lasso.coef_, 's', label="Lasso alpha=1")
  plt.legend(ncol=2,loc=(0,1.05))
  plt.ylim(-1000,1000)
  plt.xlabel("Coefficient index")
  plt.ylabel("Coefficient magnitude")
```

See Exercise 3.

5 Exercises

Write all your answers in your Jupyter notebook.

1. Comment on the size of the coefficients for different values of α in `Out[19]`. (You might need to vary the range in `plt.ylim`.) Is this what you expected?

2. What is the role of the command `plt.hlines(0,0,len(lr.coef_))` in `In[19]`? (If needed, use `help()` or [2].)

3. Comment on the size of the coefficients for different values of α in `Out[29]`. (Do not forget to vary the range in `plt.ylim`.) Is this what you expected? How is this picture different from the one in `In[19]`?

4. Explain what is going on in `np.sum(lasso.coef_ != 0)` in `In[22]`. HINT: see Exercise 1 of Lab 2.

5. What is the role of `plt.legend(ncol=2,loc=(0,1.05))` in `In[29]`?

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.