linear_models_regularization

May 29, 2021

1 Regularization of linear regression model

In this notebook, we will see the limitations of linear regression models and the advantage of using regularized models instead.

Besides, we will also present the preprocessing required when dealing with regularized models, furthermore when the regularization parameter needs to be tuned.

We will start by highlighting the over-fitting issue that can arise with a simple linear regression model.

1.1 Effect of regularization

We will first load the California housing dataset.

Note

If you want a deeper overview regarding this dataset, you can refer to the Appendix - Datasets description section at the end of this MOOC.

[ ]:
```
from sklearn.datasets import fetch_california_housing

data, target = fetch_california_housing(as_frame=True, return_X_y=True)
target *= 100  # rescale the target in k$
data.head()
```

In one of the previous notebook, we showed that linear models could be used even in settings where `data` and `target` are not linearly linked.

We showed that one can use the `PolynomialFeatures` transformer to create additional features encoding non-linear interactions between features.

Here, we will use this transformer to augment the feature space. Subsequently, we will train a linear regression model. We will use the out-of-sample test set to evaluate the generalization capabilities of our model.

[ ]:
```
from sklearn.model_selection import cross_validate
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

linear_regression = make_pipeline(PolynomialFeatures(degree=2),
```

```
                                    LinearRegression())
cv_results = cross_validate(linear_regression, data, target,
                            cv=10, scoring="neg_mean_squared_error",
                            return_train_score=True,
                            return_estimator=True)
```

We can compare the mean squared error on the training and testing set to assess the generalization performance of our model.

[ ]:
```
train_error = -cv_results["train_score"]
print(f"Mean squared error of linear regresion model on the train set:\n"
      f"{train_error.mean():.3f} +/- {train_error.std():.3f}")
```

[ ]:
```
test_error = -cv_results["test_score"]
print(f"Mean squared error of linear regresion model on the test set:\n"
      f"{test_error.mean():.3f} +/- {test_error.std():.3f}")
```

The score on the training set is much better. This statistical performance gap between the training and testing score is an indication that our model overfitted our training set.

Indeed, this is one of the danger when augmenting the number of features with a `PolynomialFeatures` transformer. Our model will focus on some specific features. We can check the weights of the model to have a confirmation. Let's create a dataframe: the columns will contain the name of the feature while the line the coefficients values stored by each model during the cross-validation.

Since we used a `PolynomialFeatures` to augment the data, we will create feature names representative of the feature combination. Scikit-learn provides a `get_feature_names` method for this purpose. First, let's get the first fitted model from the cross-validation.

[ ]:
```
model_first_fold = cv_results["estimator"][0]
```

Now, we can access to the fitted `PolynomialFeatures` to generate the feature names

[ ]:
```
feature_names = model_first_fold[0].get_feature_names(
    input_features=data.columns)
feature_names
```

Finally, we can create the dataframe containing all the information.

[ ]:
```
import pandas as pd

coefs = [est[-1].coef_ for est in cv_results["estimator"]]
weights_linear_regression = pd.DataFrame(coefs, columns=feature_names)
```

Now, let's use a box plot to see the coefficients variations.

[ ]:
```
import matplotlib.pyplot as plt
```

```
color = {"whiskers": "black", "medians": "black", "caps": "black"}
weights_linear_regression.plot.box(color=color, vert=False, figsize=(6, 16))
_ = plt.title("Linear regression coefficients")
```

We can force the linear regression model to consider all features in a more homogeneous manner. In fact, we could force large positive or negative weight to shrink toward zero. This is known as regularization. We will use a ridge model which enforces such behavior.

[ ]:
```
from sklearn.linear_model import Ridge

ridge = make_pipeline(PolynomialFeatures(degree=2),
                      Ridge(alpha=100))
cv_results = cross_validate(ridge, data, target,
                            cv=10, scoring="neg_mean_squared_error",
                            return_train_score=True,
                            return_estimator=True)
```

[ ]:
```
train_error = -cv_results["train_score"]
print(f"Mean squared error of linear regresion model on the train set:\n"
      f"{train_error.mean():.3f} +/- {train_error.std():.3f}")
```

[ ]:
```
test_error = -cv_results["test_score"]
print(f"Mean squared error of linear regresion model on the test set:\n"
      f"{test_error.mean():.3f} +/- {test_error.std():.3f}")
```

We see that the training and testing scores are much closer, indicating that our model is less overfitting. We can compare the values of the weights of ridge with the un-regularized linear regression.

[ ]:
```
coefs = [est[-1].coef_ for est in cv_results["estimator"]]
weights_ridge = pd.DataFrame(coefs, columns=feature_names)
```

[ ]:
```
weights_ridge.plot.box(color=color, vert=False, figsize=(6, 16))
_ = plt.title("Ridge weights")
```

By comparing the magnitude of the weights on this plot compared to the previous plot, we see that the magnitude of the weights are shrunk towards zero in comparison with the linear regression model.

However, in this example, we omitted two important aspects: (i) the need to scale the data and (ii) the need to search for the best regularization parameter.

1.2 Scale your data!

Regularization will add constraints on weights of the model. We saw in the previous example that a ridge model will enforce that all weights have a similar magnitude. Indeed, the larger alpha is, the larger this enforcement will be.

This procedure should make us think about feature rescaling. Let's consider the case where features

have an identical data dispersion: if two features are found equally important by the model, they will be affected similarly by regularization strength.

Now, let's consider the scenario where features have completely different data dispersion (for instance age in years and annual revenue in dollars). If two features are as important, our model will boost the weights of features with small dispersion and reduce the weights of features with high dispersion.

We recall that regularization forces weights to be closer. Therefore, we get an intuition that if we want to use regularization, dealing with rescaled data would make it easier to find an optimal regularization parameter and thus an adequate model.

As a side note, some solvers based on gradient computation are expecting such rescaled data. Unscaled data will be detrimental when computing the optimal weights. Therefore, when working with a linear model and numerical data, it is generally good practice to scale the data.

Thus, we will add a `StandardScaler` in the machine learning pipeline. This scaler will be placed just before the regressor.

[ ]:
```
from sklearn.preprocessing import StandardScaler

ridge = make_pipeline(PolynomialFeatures(degree=2), StandardScaler(),
                      Ridge(alpha=0.5))
cv_results = cross_validate(ridge, data, target,
                            cv=10, scoring="neg_mean_squared_error",
                            return_train_score=True,
                            return_estimator=True)
```

[ ]:
```
train_error = -cv_results["train_score"]
print(f"Mean squared error of linear regresion model on the train set:\n"
      f"{train_error.mean():.3f} +/- {train_error.std():.3f}")
```

[ ]:
```
test_error = -cv_results["test_score"]
print(f"Mean squared error of linear regresion model on the test set:\n"
      f"{test_error.mean():.3f} +/- {test_error.std():.3f}")
```

We observe that scaling data has a positive impact on the test score and that the test score is closer to the train score. It means that our model is less overfitted and that we are getting closer to the best generalization sweet spot.

Let's have an additional look to the different weights.

[ ]:
```
coefs = [est[-1].coef_ for est in cv_results["estimator"]]
weights_ridge = pd.DataFrame(coefs, columns=feature_names)
```

[ ]:
```
weights_ridge.plot.box(color=color, vert=False, figsize=(6, 16))
_ = plt.title("Ridge weights with data scaling")
```

Compare to the previous plots, we see that now all weight manitudes are closer and that all weights are more equally contributing.

In the previous analysis, we did not study if the parameter `alpha` will have an effect on the performance. We chose the parameter beforehand and fix it for the analysis.

In the next section, we will check the impact of this hyperparameter and how it should be tuned.

1.3 Fine tuning the regularization parameter

As mentioned, the regularization parameter needs to be tuned on each dataset. The default parameter will not lead to the optimal model. Therefore, we need to tune the `alpha` parameter.

Model hyperparameter tuning should be done with care. Indeed, we want to find an optimal parameter that maximizes some metrics. Thus, it requires both a training set and testing set.

However, this testing set should be different from the out-of-sample testing set that we used to evaluate our model: if we use the same one, we are using an `alpha` which was optimized for this testing set and it breaks the out-of-sample rule.

Therefore, we should include search of the hyperparameter `alpha` within the cross-validation. As we saw in previous notebooks, we could use a grid-search. However, some predictor in scikit-learn are available with an integrated hyperparameter search, more efficient than using a grid-search. The name of these predictors finishes by `CV`. In the case of `Ridge`, scikit-learn provides a `RidgeCV` regressor.

Therefore, we can use this predictor as the last step of the pipeline. Including the pipeline a cross-validation allows to make a nested cross-validation: the inner cross-validation will search for the best alpha, while the outer cross-validation will give an estimate of the testing score.

[ ]:
```
import numpy as np
from sklearn.linear_model import RidgeCV

alphas = np.logspace(-2, 0, num=20)
ridge = make_pipeline(PolynomialFeatures(degree=2), StandardScaler(),
                      RidgeCV(alphas=alphas, store_cv_values=True))
```

[ ]:
```
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=5, random_state=1)
cv_results = cross_validate(ridge, data, target,
                            cv=cv, scoring="neg_mean_squared_error",
                            return_train_score=True,
                            return_estimator=True, n_jobs=-1)
```

[ ]:
```
train_error = -cv_results["train_score"]
print(f"Mean squared error of linear regresion model on the train set:\n"
      f"{train_error.mean():.3f} +/- {train_error.std():.3f}")
```

[ ]:
```
test_error = -cv_results["test_score"]
print(f"Mean squared error of linear regresion model on the test set:\n"
      f"{test_error.mean():.3f} +/- {test_error.std():.3f}")
```

By optimizing `alpha`, we see that the training an testing scores are closed. It indicates that our model is not overfitting.

When fitting the ridge regressor, we also requested to store the error found during cross-validation (by setting the parameter `store_cv_values=True`). We will plot the mean squared error for the different `alphas` regularization strength that we tried.

[ ]:
```
mse_alphas = [est[-1].cv_values_.mean(axis=0)
              for est in cv_results["estimator"]]
cv_alphas = pd.DataFrame(mse_alphas, columns=alphas)
cv_alphas
```

[ ]:
```
cv_alphas.mean(axis=0).plot(marker="+")
plt.ylabel("Mean squared error\n (lower is better)")
plt.xlabel("alpha")
_ = plt.title("Error obtained by cross-validation")
```

As we can see, regularization is just like salt in cooking: one must balance its amount to get the best statistical performance. We can check if the best `alpha` found is stable across the cross-validation fold.

[ ]:
```
best_alphas = [est[-1].alpha_ for est in cv_results["estimator"]]
best_alphas
```

In this notebook, you learned about the concept of regularization and the importance of preprocessing and parameter tuning.