Exercise 6 – Evaluation II

Introduction to Machine Learning

Hint: Useful libraries

R

```r
# Consider the following libraries for this exercise sheet:

library(mlbench)
library(mlr3)
library(mlr3learners)
```

Python

```python
# Consider the following libraries for this exercise sheet:

# general
import numpy as np
import pandas as pd

# sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import RepeatedStratifiedKFold
```

Exercise 1: Overfitting & underfitting

Learning goals

Discuss for given situations if they might lead to overfitting / underfitting

Assume a polynomial regression model with a continuous target variable y, a continuous, p-dimensional feature vector $\mathbf{x}$ and polynomials of degree d, i.e.,

$$f\left(\mathbf{x}^{(i)}\right) = \sum_{j=1}^{p} \sum_{k=0}^{d} \theta_{j,k} (\mathbf{x}_j^{(i)})^k.$$

For each of the following situations, indicate whether we would generally expect the performance of a flexible polynomial learner (high d) to be better or worse than an inflexible one (low d). Justify your answer.

NB

We can only state tendencies here; performance strongly depends on the specific data situation.

---

i. The sample size n is extremely large, and the number of features p is small.

Solution

We expect the flexible learner to perform better because it covers a large number of hypotheses and we have enough data to tell them apart, so there is little risk of overfitting, while the inflexible learner might underfit.

---

ii. The number of features p is extremely large, and the number of observations n is small.

Solution

With a flexible learner we might quickly run into overfitting here. The data situation, which we frequently encounter in biostatistics (e.g., genomics data), creates a high-dimensional and sparsely populated input space with few training points (relatively speaking) that are easy to overfit. Therefore, a low-degree polynomial should fare better (but note that it is not immune to overfitting either in this challenging setting). Some sort of feature selection could be advisable first.

iii. The true relationship between the features and the response is highly non-linear.

Solution

The inflexible learner will likely underfit here, so the flexible variant can be expected to perform better.

iv. The data could only be observed with a high level of noise.

Solution

The flexible learner might interpolate between the noisy training samples, creating a wiggly curve that generalizes poorly, so we expect better performance from a less complex learner.

Are overfitting and underfitting properties of a learner or of a fixed model? Explain your answer.

Solution

Overfitting and underfitting are always connected to a particular fixed *model*, even though attributes of the underlying hypothesis space typically influence the tendency toward one or the other behavior, as we have seen in the previous question. In order to understand this, think of a classification problem with linearly separable data. Applying a QDA learner, which is able to learn more complex decision boundaries, poses a risk of overfitting with the chosen model, but the degree of overfitting depends on the model itself. In theory, the QDA learner is free to set equal covariances for the Gaussian class densities, amounting to LDA and *not* overfitting the data. Under- and overfitting are therefore properties of a specific model and not of an entire learner.

A common strategy is to choose a rather flexible model class and encourage simplicity in the actual model by *regularization* (e.g., take a higher-degree polynomial but drive as many coefficients a possible toward zero, which you might know as LASSO regression).

Should we aim to completely avoid both overfitting and underfitting?

Solution

That will be hardly possible. Recall how we defined the two variants of data fit:

$$UF(\hat{f}, L) = GE(\hat{f}, L) - GE(f^*, L)$$

$$OF(\hat{f}, L) = GE(\hat{f}, L) - \mathcal{R}_{\text{emp}}(\hat{f}, L)$$

In order to avoid underfitting completely we would need to always find the universally loss-optimal model across arbitrary hypothesis spaces (the so-called *Bayes-optimal* model), which is obviously not something we can hope to achieve in general. Zero overfitting would mean to exactly balance theoretical generalization error and empirical risk, but the way empirical risk minimization is designed, our model will likely fare a bit worse on unseen test data.

In practice we will always experience these phenomena to some degree and finding a model that trades them off well is the holy grail in machine learning.

Exercise 2: Resampling strategies

Learning goals

1. Implement resampling procedures in R/Python
2. Understand how the choice of resampling strategy affects the quality of the GE estimator

---

Why would we apply resampling rather than a single holdout split?

Solution

The two main advantages of resampling are:

- We are able to use larger training sets (at the expense of test set size) because the high variance this incurs for the resulting estimator is smoothed out by averaging across repetitions.
- Repeated sampling reduces the risk of getting lucky (or not so lucky) with a particular data split, which is especially relevant with few observations.

Classify the `german_credit` data into solvent and insolvent debtors using logistic regression. Compute the training error w.r.t. MCE.

Python Hint

Read the already preprocessed file german_credit_for_py.csv

Solution

R

```
# create task and learner
(task <- tsk("german_credit"))
learner <- lrn("classif.log_reg")

# train, predict and compute train error
learner$train(task)
preds <- learner$predict(task)
preds$score()
```

```
<TaskClassif:german_credit> (1000 x 21): German Credit
* Target: credit_risk
* Properties: twoclass
* Features (20):
  - fct (14): credit_history, employment_duration, foreign_worker,
    housing, job, other_debtors, other_installment_plans,
    people_liable, personal_status_sex, property, purpose, savings,
    status, telephone
  - int (3): age, amount, duration
  - ord (3): installment_rate, number_credits, present_residence
```

classif.ce: 0.211

Python

Read data

```
german_credit = pd.read_csv("../data/german_credit_for_py.csv")
german_credit.head()
```

	credit_risk	status_... >= 200 DM / salary for at least 1 year	status_0<= ... < 200 DM	status_no c
0	good	0.0	0.0	1.0
1	bad	0.0	0.0	0.0
2	good	1.0	0.0	0.0
3	good	0.0	0.0	1.0
4	bad	0.0	0.0	1.0

Encode data and train classifier

```
german_x_raw = german_credit.iloc[:,1:]
german_y_raw = german_credit.iloc[:,0]

# Initialize encoder for target
enc_target = LabelEncoder()

enc_target.fit(german_y_raw.values.ravel())
# .values will give the values in a numpy array (shape: (n,1))
# .ravel will convert that array shape to (n, ) (i.e. flatten it)

german_y = enc_target.transform(german_y_raw.values.ravel()) # now numpy array
# you can also use enc_target.fit_transform(X) to combine both steps

german_x = np.asarray(german_x_raw)

# Using whole data set to train and predict; increase max iterations for convergence
log_mod = LogisticRegression(max_iter=10000).fit(german_x, german_y)

print("Mean accuracy: %.2f" % log_mod.score(german_x, german_y))
print("Mean classification error : %.2f" % (1-log_mod.score(german_x, german_y)))
```

```
Mean Accuracy: 0.78
Mean Classification Error : 0.22
```

In order to evaluate your learner, compare the test MCE using

1. three times ten-fold cross validation (3x10-CV)
2. 10x3-CV

3. 3x10-CV with stratification for the feature `foreign_worker` to ensure equal representation in all folds
4. a single holdout split with 90% training data

Hint

R

You will need `rsmp`, `resample` and `aggregate`.

Python

You will need `RepeatedKFold`, `RepeatedStratifiedKFold` and `train_test_split`.

Solution

R

```r
# create different resampling strategies
set.seed(123)
resampling_3x10_cv <- rsmp("repeated_cv", folds = 10, repeats = 3)
resampling_10x3_cv <- rsmp("repeated_cv", folds = 3, repeats = 10)
resampling_ho <- rsmp("holdout", ratio = 0.9)

# evaluate without stratification
result_3x10_cv <- resample(task, learner, resampling_3x10_cv, store_models = TRUE)
result_10x3_cv <- resample(task, learner, resampling_10x3_cv, store_models = TRUE)
result_ho <- resample(task, learner, resampling_ho, store_models = TRUE)

# evaluate with stratification
task_stratified <- task$clone()
task_stratified$set_col_roles("foreign_worker", roles = "stratum")
result_stratified <- resample(
  task_stratified, learner, resampling_3x10_cv, store_models = TRUE)
```

foo

```r
# aggregate results over splits (mce is default)
print(sapply(
  list(result_3x10_cv, result_10x3_cv, result_stratified, result_ho),
  function(i) i$aggregate()))
```

```
classif.ce classif.ce classif.ce classif.ce
 0.2486667  0.2557977  0.2525512  0.1800000
```

Python

```python
random_state = 14
err = []
rkf_3x10 = RepeatedKFold(n_splits=10, n_repeats=3, random_state=random_state)
for train, test in rkf_3x10.split(german_x):
    log_mod = LogisticRegression(max_iter=10000).fit(
        german_x[train,:], german_y[train]
    )
    err.append(1-log_mod.score(german_X[test,:], german_y[test]))
    # score gives mean accuracy

res = np.array(err)
print("MCE of 3x10 CV: ", res.mean())
```

```
MCE of 3x10 CV:  0.24433333333333332
```

foo

```python
err = []
rkf_10x3 = RepeatedKFold(n_splits=3, n_repeats=10, random_state=random_state)
for train, test in rkf_10x3.split(german_x):
    log_mod = LogisticRegression(max_iter=10000).fit(
        german_x[train,:], german_y[train]
    )
    err.append(1-log_mod.score(german_x[test,:], german_y[test]))

res = np.array(err)
print("MCE of 10x3 CV: ",res.mean())
```

```
MCE of 10x3 CV:  0.253094112076148
```

foo

```python
err = []
strat_gkf_10 = RepeatedStratifiedKFold(
    n_splits=10, n_repeats=3, random_state=random_state
)
# Note that providing y in split(X, y) is sufficient to generate the splits,
# and hence np.zeros(n_samples) may be used as a placeholder for X instead
# of actual training data.
for train, test in strat_gkf_10.split(german_x, german_x[:,41]):
    # index 41 stands for column of foreign_workers_yes
    log_mod = LogisticRegression(max_iter=10000).fit(
        german_x[train,:], german_y[train]
    )
    err.append(1-log_mod.score(german_x[test,:], german_y[test]))

res = np.array(err)
print("MCE of 3x10-CV with stratification: ", res.mean())
```

```
MCE of 3x10-CV with stratification:  0.24966666666666665
```

foo

```
x_train, x_test, y_train, y_test = train_test_split(
    german_x, german_y, test_size = 0.1, random_state=random_state
)
log_mod = LogisticRegression(max_iter=10000).fit(x_train, y_train)
german_pred = log_mod.predict(x_test)
print("MCE of Hold-out split: ", 1-log_mod.score(x_test, y_test))
```

```
MCE of Hold-out split:  0.30000000000000004
```

Discuss and compare your findings and compare them to the training error computed previously.

Solution

Generalization error estimates are pretty stable across the different resampling strategies because we have a fairly large number (1000) of observations. Still, the pessimistic bias of small training sets is visible: 10x3-CV, using roughly 67% of data for training in each split, estimates a higher generalization error than 3x10-CV with roughly 90% training data. Stratification by `foreign_worker` does not seem to have much effect on the estimate. However, we see a glaring difference when we use a single 90%-10% split, where the estimated GE is quite different from 3x10-CV, meaning we got a different error just because of an (un)lucky split.

Comparing the results (except for the unreliable one produced by a single split) with the training error from before indicates no serious overfitting.

Would you consider LOO-CV to be a good alternative?

Solution

LOO is probably not a very good idea here – with 1000 observations this would take a long time. Also, LOO has high variance by nature. Repeated CV with a sufficient number of folds should give us a pretty good idea about the expected GE of our learner.