Solutions 12

Jumping Rivers

Alternative Dimension Reduction

We will use the same data set as the previous example so that we can compare the results

```
from sklearn.datasets import make_regression

X, y, ground_truth = make_regression(
  n_samples=1000, n_features=1000,
  n_informative=100, effective_rank=20,
  random_state=2019, noise=0.2, coef=True
)
```

a) Create a pipline for fitting a PCR

```
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, make_scorer
import numpy as np

pca_pipe = Pipeline([
  ('preprocess', StandardScaler()),
  ('pca', PCA()),
  ('regression', LinearRegression())
])

pca_pipe.fit(X,y)
```

```
## Pipeline(memory=None,
##          steps=[('preprocess',
##                  StandardScaler(copy=True, with_mean=True, with_std=True)),
##                 ('pca',
##                  PCA(copy=True, iterated_power='auto', n_components=None,
##                      random_state=None, svd_solver='auto', tol=0.0,
##                      whiten=False)),
##                 ('regression',
##                  LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
##                                   normalize=False))],
##          verbose=False)
```

b) Use a grid search cross validation to find the optimal number of components to use.

```
params = {
  'pca__n_components': np.arange(1,100)
}
search = GridSearchCV(
  pca_pipe, params, cv=10,
  scoring=make_scorer(mean_squared_error, greater_is_better=False),
  return_train_score=False
)
results = search.fit(X, y)
results.best_score_
```

```
## -3.5957437806800945
```

```
results.best_params_
```

```
## {'pca__n_components': 98}
```

c) From your `GridSearchCV` object you can retrieve the best estimator as the `.best_estimator_` attribute. Use this to get a cross validation estimate of the mean squared error for this model.

```
from sklearn.model_selection import cross_validate
from sklearn.metrics import mean_squared_error, make_scorer

score_fn = make_scorer(mean_squared_error)

pcr_model = cross_validate(
  search.best_estimator_, X, y,
  scoring=score_fn, cv=10,
  return_train_score=False
)

pcr_model['test_score'].mean()
```

```
## 3.685193649865466
```

d) How does this compare to the linear model from the previous practical

```
print("""
It looks like performance is worse in this instance
""")
```

```
##
## It looks like performance is worse in this instance
```

e) Why do you think this is?

```
print("""
In the training data, the input variables all have similar variation.
PCR finds components while ignoring the response variable
So we struggle here because only a few of the inputs are important, but the
principal component decomposition has no way of knowing this.
Performance gets worse because we are reducing the dimension, throwing away information,
but are not picking out the important variables.
""")
```

```
##
## In the training data, the input variables all have similar variation.
## PCR finds components while ignoring the response variable
## So we struggle here because only a few of the inputs are important, but the
## principal component decomposition has no way of knowing this.
## Performance gets worse because we are reducing the dimension, throwing away information,
## but are not picking out the important variables.
```

f) Fit a PLS regression and compare

```
from sklearn.cross_decomposition import PLSRegression

pls_pipe = Pipeline([
  ('preprocess', StandardScaler()),
  ('pls', PLSRegression())
])

pls_pipe.fit(X,y)
```

```
## Pipeline(memory=None,
##          steps=[('preprocess',
##                  StandardScaler(copy=True, with_mean=True, with_std=True)),
##                 ('pls',
##                  PLSRegression(copy=True, max_iter=500, n_components=2,
##                                scale=True, tol=1e-06))],
##          verbose=False)
```

```
params = {
  'pls__n_components': np.arange(1,100)
}
search = GridSearchCV(
  pls_pipe, params, cv=10,
  scoring=make_scorer(mean_squared_error, greater_is_better=False),
  return_train_score=False
)
```

```
results = search.fit(X, y)
results.best_score_
```

```
## -0.2090607549638625
```

```
results.best_params_
```

```
## {'pls__n_components': 58}
```

```
pls_model = cross_validate(
  search.best_estimator_, X, y,
  scoring=score_fn, cv=10,
  return_train_score=False
)
```

```
pls_model['test_score'].mean()
```

```
## 0.2090607549638625
```

```
print("""
Performance is better than PCR since we now consider the response when
choosing the components. We are still not seeing great improvement over linear regression
because we are including useless parameters.
""")
```

```
##
## Performance is better than PCR since we now consider the response when
## choosing the components. We are still not seeing great improvement over linear regression
## because we are including useless parameters.
```

Diagnosing progression of diabetes

Here we will put together all of the elements we have seen so far to build a model for predicting the progression of diabetes in patients. The data is available as part of the sklearn package.

```
from sklearn.datasets import load_diabetes
import pandas as pd
```

```
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target
feature_names = diabetes.feature_names
```

```
full_df = pd.DataFrame(X, columns=feature_names)
full_df['target'] = y
```

a) Start by producing some exploratory graphics and summary statistics. As a hint for producing plots quickly, you can use the **seaborn** library `pairplot()` function.

b) Given what we have discussed I want you to tell me about the diabetes data. What interesting things are there? What are important predictors in disease progression. What is the best model you can come up with for predicting disease progression for a new patient.