hw3

March 13, 2023

1 Problem 5

[15]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

[1]:
```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.linear_model import LinearRegression, LogisticRegression, Ridge,␣
↪RidgeCV, Lasso, LassoCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,␣
↪QuadraticDiscriminantAnalysis
from sklearn.metrics import mean_squared_error, accuracy_score, r2_score
```

[2]:
```
train = pd.read_csv('mnist_train.csv')
test = pd.read_csv('mnist_test.csv')
```

[3]:
```
X_train = train.iloc[:, 1:]/255
y_train = train.iloc[:, 0]
X_test = test.iloc[:, 1:]/255
y_test = test.iloc[:, 0]
```

[4]:
```
# X_train = StandardScaler().fit_transform(X_train)
# pca = PCA()
# X_train = pca.fit_transform(X_train)
```

1.0.1 1. The pixel values in both the training set ranges in [0,255], thus divide them by 255 to normalize them between [0, 1]

1.0.2 2. Individually train Logistic Regression, LDA and QDA on the training set

Logistic regression

[19]:
```
lr = LogisticRegression().fit(X_train, y_train)
lrpred = lr.predict(X_test)
```

```
/Users/colin/opt/miniconda3/lib/python3.10/site-
packages/sklearn/linear_model/_logistic.py:814: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

[20]:
```
accuracy_score(lrpred, y_test)
```

[20]: 0.9258

LDA

[21]:
```
lda = LinearDiscriminantAnalysis().fit(X_train, y_train)
ldapred = lda.predict(X_test)
```

[22]:
```
accuracy_score(ldapred, y_test)
```

[22]: 0.873

QDA

[9]:
```
qda = QuadraticDiscriminantAnalysis().fit(X_train, y_train)
qdapred = qda.predict(X_test)
accuracy_score(qdapred, y_test)
```

```
/Users/colin/opt/miniconda3/lib/python3.10/site-
packages/sklearn/discriminant_analysis.py:878: UserWarning: Variables are
collinear
  warnings.warn("Variables are collinear")
```

[9]: 0.5553

Naive Bayes

[10]:
```
# prepare the cross-validation procedure
cv = KFold(n_splits=10, random_state=1, shuffle=True)
# create model
naive_bayes = MultinomialNB()
# evaluate model
scores = cross_val_score(naive_bayes, X_train, y_train, scoring='accuracy',␣
 ↪cv=cv, n_jobs=-1)
# report performance
```

```
print('Accuracy: %.4f' % (np.mean(scores)))
```

```
Accuracy: 0.8235
```

It seems that logistic regression performed the best all of four algorithems and QDA performed worst. Logistics regression is the simplest model of all, which does not seem to underfit the mnist dataset, while quadratic discriminant analysis performs the worst, probably due to possible over-fitting on the training data.

2 Problem 6

[38]:
```
college = pd.read_csv('College.csv')
X = college.drop(['Unnamed: 0', 'Apps'], axis = 1)
y = college['Apps']
```

2.0.1 1. Split the data set into a training set and a test set, normalize data

[39]:
```
# only categorical variable: private (yes/no), convert it (1/0)
X = pd.get_dummies(data=X, drop_first=True)
```

[41]:
```
# normalized all columns except categorical feature 'Private_Yes'
scaler = StandardScaler()
X.loc[:, X.columns != 'Private_Yes'] = scaler.fit_transform(X.loc[:, X.columns !
 ↪= 'Private_Yes'])
# check result: mean=zero, variance=1
# X.mean(axis=0)
# X.var(axis=0)
```

[43]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
 ↪random_state=42)
```

2.0.2 2. Fit linear regression and report test error

[44]:
```
regr = LinearRegression().fit(X_train, y_train)  # fit linear regression
print("Mean squared error: %.2f" % mean_squared_error(y_test, regr.
 ↪predict(X_test)))  # MSE
```

```
Mean squared error: 1492443.38
```

2.0.3 3. Fit a ridge regression model on the training set, with chosen by cross-validation

[45]:
```
ridgecv = RidgeCV()  # leave-one-out cv
ridgecv.fit(X_train, y_train)  # fit ridge
ridgecv.alpha_  # best alpha
```

[45]: 0.1

[46]:
```
ridge = Ridge(alpha = ridgecv.alpha_)  # using best alpha
ridge.fit(X_train, y_train)  # fit ridge
print("Mean squared error: %.2f" % mean_squared_error(y_test, ridge.
 ↪predict(X_test)))  # MSE
```

```
Mean squared error: 1490697.78
```

2.0.4 4. Fit LASSO regression

[47]:
```
lassocv = LassoCV()  # leave-one-out cv
lassocv.fit(X_train, y_train)  # fit lasso
print('lasso alpha: ', lassocv.alpha_)  # best alpha
```

```
lasso alpha:  3.7424119368188964
```

[48]:
```
lasso = Lasso(alpha=lassocv.alpha_)  # using best alpha
lasso.fit(X_train, y_train)   # fit lasso
print("Mean squared error: %.2f" % mean_squared_error(y_test, lasso.
 ↪predict(X_test)))  # MSE
```

```
Mean squared error: 1474198.20
```

[49]:
```
print('lasso nonzero coefficients:', lasso.coef_)
```

```
lasso nonzero coefficients: [4027.14242701 -780.41613793  869.74492015
-259.00089067  146.69769831
  -20.64162858 -277.05730676  164.69712511   17.28003815   21.88049098
 -160.61002607  -17.41857252   14.06946747   -8.6408197   206.9888744
  137.63842589 -609.42279556]
```

2.0.5 5. Comment on results obtained

[50]:
```
r2_score(y_test, regr.predict(X_test))
```

[50]: 0.8877583168400993

[51]:
```
r2_score(y_test, ridge.predict(X_test))
```

[51]: 0.8878895973260851

[52]:
```
r2_score(y_test, lasso.predict(X_test))
```

[52]: 0.8891304757579555

It seems that LASSO performed the best, where the 88.91% of the variation in the response variable is explained by the predictors. While ridge regression, 88.79% of the variation is explained by the predictors. Both penalized regression performed similarly. This is probably because even though, ridge did not select variables, it still shrinks the coefficient to be fairly small numbers. Thus, producing similar results.

```
[ ]:
```