Machine Learning Exercise - II - Solutions

May 3, 2022

Kannan Singaravelu

* * *

Exercise 3

Generate a training dataset containing 30 observations with two predictors centered around -0.5 and 0.5 with a cluster standard deviation of 0.4 and one qualitative response variable. Define classes that takes 'Red' when response variable is positive and 'Blue' otherwise. Use this generated dataset to make a prediction for y when X1 = X2 = 0.25 using K-nearest neighbours.

a) Compute the Euclidean distance between each observations and the test points.
b) What is the class prediction with K = 1?
c) What is the class prediction with K = 5?
d) Plot the classification points with decision boundary for K = 5.

Solution

Generate dataset

We use `scikit-learn` to generate datasets with centers [0.5,0.5], [-0.5,-0.5] and cluster standard deviation of 0.4.

Finding the neighbors

Distance can be thought of as a measure of similarity. Euclidean distance is the most commonly used but other distance metrics such as Manhattan work as well. The generalized distance metric is called the Minkowski distance, defined as

$$d = \left(\sum_{n=i}^{n} |x_i - y_i|^p \right)^{1/p},$$

where x_i and y_i are the two observations for which distance d is being calculated with a hyperparameter, integer p.

When $p = 1$, the Minkowski distance is the Manhattan distance and when $p = 2$, the Minkowski distance is the just the standard Euclidean distance. With the K neighbors identified using distance metrics, the algorithm can make a classification or prediction with the label values of the neighbors.

[1]:
```
# Import libraries
import pandas as pd
import numpy as np

# Import plotting library
import matplotlib.pyplot as plt

# Preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

from sklearn.datasets import make_blobs
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics.pairwise import euclidean_distances
```

[2]:
```
X, y = make_blobs(n_samples=30, centers=[[0.5,0.5], [-0.5,-0.5]], cluster_std=0.
 ↪4, random_state=110)
```

[3]:
```
df = pd.DataFrame({'X1': X[:,0],
                   'X2': X[:,1],
                   'Y': y})
# specify classes
df['Class'] = df['Y'].apply(lambda x: 'Red' if x>0 else 'Blue')
```

a) Compute the Euclidean distance between each observations and the test points.

[4]:
```
edist = pd.Series(euclidean_distances(df[['X1', 'X2']], [[0.25, 0.25]]).
 ↪flatten(), name='Euclidean')
df  = pd.concat([df, edist], axis=1)
df.head()
```

[4]:
```
         X1        X2  Y Class  Euclidean
0 -0.621282 -0.331191  1   Red   1.047337
1  0.410545 -0.552438  0  Blue   0.818341
2 -0.046838 -0.632821  1   Red   0.931389
3 -0.455832 -0.826784  1   Red   1.287502
4 -0.087054  1.217604  0  Blue   1.024628
```

b) What is the class prediction with K = 1?

[5]:
```
df.nsmallest(1,'Euclidean')
```

[5]:
```
          X1        X2  Y Class  Euclidean
15  0.434864  0.202033  0  Blue   0.190986
```

c) What is the class prediction with K = 5?

[6]:
```
df.nsmallest(5,'Euclidean')
```

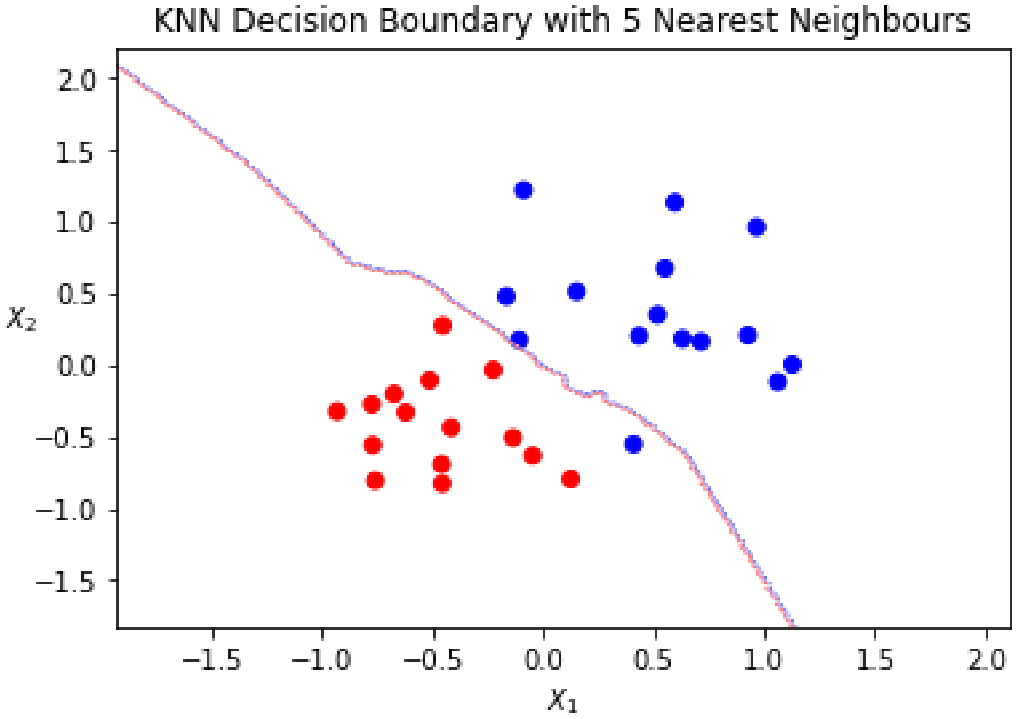
[6]:
```
          X1        X2  Y Class  Euclidean
15  0.434864  0.202033  0  Blue   0.190986
9   0.152672  0.510864  0  Blue   0.278429
6   0.518747  0.349231  0  Blue   0.286482
20 -0.107496  0.173775  0  Blue   0.365532
11  0.631439  0.181521  0  Blue   0.387537
```

d) Draw decision doundary with K=5.

[7]:
```
def plot_boundary(x, y, k):

    # Instantiate the model object
    knn = KNeighborsClassifier(n_neighbors=k)

    # Fits the model
    knn.fit(x, y)

    # Step size of the mesh.
    h = .02

    # Plot the decision boundary.
    x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
    y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1

    # Create Meshgrid
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Predict labels for each point in mesh
    Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])

    # Reshape to match dimensions
    Z = Z.reshape(xx.shape)

    # Plotting
    plt.contour(xx, yy, Z, cmap=plt.cm.bwr, linestyles = 'dashed', linewidths=0.
 ↪5)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.bwr)
    plt.title(f'KNN Decision Boundary with {k} Nearest Neighbours')
    plt.xlabel('$X_1$')
    plt.ylabel('$X_2$', rotation='horizontal')
    plt.show()
```

[8]:

```
# Plot KNN decision boundary with K=5
plot_boundary(X, y, 5)
```



Exercise 4

For this exercise, use the admission dataset from https://stats.idre.ucla.edu/stat/data/binary.csv. The dataset contains three predictor variables: gre, gpa and rank and one binary response variable called admit.

a) List all tunable hyperparameters.
b) Select the best model by searching over a range of hyperparameters based on cross validation score using an Exhaustive Search.

Solution

GridSearch

The conventional way of performing hyperparameter optimization has been a grid search (aka parameter sweep). It is an exhaustive search through a manually specified subset of the hyperparameter space of a learning algorithm. A grid search algorithm must be guided by some performance metric, typically measured by cross-validation on the training set or evaluation on a validation set.

GridSearch performs exhaustive search over specified parameter values for an estimator. It implements a “fit” and a “score” method among other methods. The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

[9]:

```
# Import Library
import io
import requests

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV, cross_val_score
```

[10]:

```
response = requests.get('https://stats.idre.ucla.edu/stat/data/binary.csv')
df = pd.read_csv(io.StringIO(response.text))
df
```

[10]:

```
     admit  gre   gpa  rank
0        0  380  3.61     3
1        1  660  3.67     3
2        1  800  4.00     1
3        1  640  3.19     4
4        0  520  2.93     4
..     …    …     …     …
395      0  620  4.00     2
396      0  560  3.04     3
397      0  460  2.63     2
398      0  700  3.65     2
399      0  600  3.89     3

[400 rows x 4 columns]
```

[11]:

```
features = df.drop('admit', axis=1)
target = df['admit']
```

[12]:

```
features
```

[12]:

```
     gre   gpa  rank
0    380  3.61     3
1    660  3.67     3
2    800  4.00     1
3    640  3.19     4
4    520  2.93     4
..   …     …     …
395  620  4.00     2
396  560  3.04     3
397  460  2.63     2
398  700  3.65     2
399  600  3.89     3

[400 rows x 3 columns]
```

```
[13]: # convert to arrays
      X = features.values
      y = target.values
```

```
[14]: # Scale and fit the model
      pipe = Pipeline([("scaler", StandardScaler()),
                       ("logistic", LogisticRegression(solver='liblinear'))])
      pipe.fit(X, y)
```

```
[14]: Pipeline(steps=[('scaler', StandardScaler()),
                      ('logistic', LogisticRegression(solver='liblinear'))])
```

a) List of all tunable hyper-parameters

```
[15]: # get model params
      pipe.get_params()
```

```
[15]: {'memory': None,
       'steps': [('scaler', StandardScaler()),
        ('logistic', LogisticRegression(solver='liblinear'))],
       'verbose': False,
       'scaler': StandardScaler(),
       'logistic': LogisticRegression(solver='liblinear'),
       'scaler__copy': True,
       'scaler__with_mean': True,
       'scaler__with_std': True,
       'logistic__C': 1.0,
       'logistic__class_weight': None,
       'logistic__dual': False,
       'logistic__fit_intercept': True,
       'logistic__intercept_scaling': 1,
       'logistic__l1_ratio': None,
       'logistic__max_iter': 100,
       'logistic__multi_class': 'auto',
       'logistic__n_jobs': None,
       'logistic__penalty': 'l2',
       'logistic__random_state': None,
       'logistic__solver': 'liblinear',
       'logistic__tol': 0.0001,
       'logistic__verbose': 0,
       'logistic__warm_start': False}
```

b) Select the best model by searching over a range of hyperparameters based on cross validation score using an Exhaustive Search.

[16]:
```
# penalty hyperparamter values
penalty = ['l1', 'l2']

# regularization hyperparamter
C = np.linspace(0.01,10,10)
C

# subsume into one dict
param_grid = dict(logistic__C=C, logistic__penalty=penalty)
```

[17]:
```
# create a grid search with cv=5
gridsearch = GridSearchCV(pipe, param_grid, n_jobs=-1, cv=5, verbose=1)

# fit grid search
best_model = gridsearch.fit(X, y)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
```

[18]:
```
# best model parameters
best_model.best_params_
```

[18]:
```
{'logistic__C': 2.23, 'logistic__penalty': 'l2'}
```

[19]:
```
# best score
best_model.best_score_
```

[19]:
```
0.7075000000000001
```

[20]:
```
pipe['logistic'].coef_
```

[20]:
```
array([[ 0.26139396,  0.29067213, -0.51864053]])
```

[21]:
```
best_model.best_estimator_.named_steps
```

[21]:
```
{'scaler': StandardScaler(),
 'logistic': LogisticRegression(C=2.23, solver='liblinear')}
```

[22]:
```
best_model.best_estimator_.named_steps['logistic'].coef_
```

[22]:
```
array([[ 0.26317473,  0.29320696, -0.52387746]])
```

[23]:
```
best_model.best_estimator_.named_steps['logistic'].intercept_
```

[23]:
```
array([-0.85285979])
```

[24]:

```
# best model params after hypertuning
best_model.get_params()
```

[24]:

```
{'cv': 5,
 'error_score': nan,
 'estimator__memory': None,
 'estimator__steps': [('scaler', StandardScaler()),
  ('logistic', LogisticRegression(solver='liblinear'))],
 'estimator__verbose': False,
 'estimator__scaler': StandardScaler(),
 'estimator__logistic': LogisticRegression(solver='liblinear'),
 'estimator__scaler__copy': True,
 'estimator__scaler__with_mean': True,
 'estimator__scaler__with_std': True,
 'estimator__logistic__C': 1.0,
 'estimator__logistic__class_weight': None,
 'estimator__logistic__dual': False,
 'estimator__logistic__fit_intercept': True,
 'estimator__logistic__intercept_scaling': 1,
 'estimator__logistic__l1_ratio': None,
 'estimator__logistic__max_iter': 100,
 'estimator__logistic__multi_class': 'auto',
 'estimator__logistic__n_jobs': None,
 'estimator__logistic__penalty': 'l2',
 'estimator__logistic__random_state': None,
 'estimator__logistic__solver': 'liblinear',
 'estimator__logistic__tol': 0.0001,
 'estimator__logistic__verbose': 0,
 'estimator__logistic__warm_start': False,
 'estimator': Pipeline(steps=[('scaler', StandardScaler()),
                 ('logistic', LogisticRegression(solver='liblinear'))]),
 'n_jobs': -1,
 'param_grid': {'logistic__C': array([ 0.01,  1.12,  2.23,  3.34,  4.45,  5.56,
6.67,  7.78,  8.89,
         10.  ]),
  'logistic__penalty': ['l1', 'l2']},
 'pre_dispatch': '2*n_jobs',
 'refit': True,
 'return_train_score': False,
 'scoring': None,
 'verbose': 1}
```

[25]:

```
# cross validation results
df1 = pd.DataFrame(best_model.cv_results_)
df1
```

```
[25]:     mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
      0        0.006046      0.001014         0.001761        0.000657
      1        0.004002      0.000632         0.001875        0.001503
      2        0.004914      0.000525         0.001396        0.000489
      3        0.004189      0.000398         0.001396        0.000488
      4        0.005216      0.001083         0.001396        0.000489
      5        0.003610      0.001336         0.001943        0.001560
      6        0.004241      0.001208         0.001516        0.001037
      7        0.004189      0.000399         0.001596        0.000488
      8        0.004531      0.000454         0.001408        0.000479
      9        0.003922      0.000530         0.001197        0.000398
      10       0.003722      0.000403         0.001101        0.000207
      11       0.003264      0.001792         0.001654        0.000901
      12       0.003415      0.000505         0.001208        0.000395
      13       0.003591      0.000487         0.001197        0.000399
      14       0.003991      0.000630         0.000996        0.000003
      15       0.002624      0.000793         0.001216        0.000267
      16       0.003438      0.000618         0.000997        0.000629
      17       0.004300      0.001184         0.000996        0.000002
      18       0.003791      0.000399         0.001403        0.000483
      19       0.003591      0.000487         0.000997        0.000631

         param_logistic__C param_logistic__penalty  \
      0               0.01                      l1
      1               0.01                      l2
      2               1.12                      l1
      3               1.12                      l2
      4               2.23                      l1
      5               2.23                      l2
      6               3.34                      l1
      7               3.34                      l2
      8               4.45                      l1
      9               4.45                      l2
      10              5.56                      l1
      11              5.56                      l2
      12              6.67                      l1
      13              6.67                      l2
      14              7.78                      l1
      15              7.78                      l2
      16              8.89                      l1
      17              8.89                      l2
      18             10.00                      l1
      19             10.00                      l2

                                                 params  split0_test_score  \
      0    {'logistic__C': 0.01, 'logistic__penalty': 'l1'}             0.6875
      1    {'logistic__C': 0.01, 'logistic__penalty': 'l2'}             0.7125
```

```
2    {'logistic__C': 1.12, 'logistic__penalty': 'l1'}             0.7125
3    {'logistic__C': 1.12, 'logistic__penalty': 'l2'}             0.7125
4    {'logistic__C': 2.23, 'logistic__penalty': 'l1'}             0.7125
5    {'logistic__C': 2.23, 'logistic__penalty': 'l2'}             0.7125
6    {'logistic__C': 3.34, 'logistic__penalty': 'l1'}             0.7125
7    {'logistic__C': 3.34, 'logistic__penalty': 'l2'}             0.7125
8    {'logistic__C': 4.45, 'logistic__penalty': 'l1'}             0.7125
9    {'logistic__C': 4.45, 'logistic__penalty': 'l2'}             0.7125
10  {'logistic__C': 5.5600000000000005, 'logistic_…             0.7125
11  {'logistic__C': 5.5600000000000005, 'logistic_…             0.7125
12   {'logistic__C': 6.67, 'logistic__penalty': 'l1'}             0.7125
13   {'logistic__C': 6.67, 'logistic__penalty': 'l2'}             0.7125
14   {'logistic__C': 7.78, 'logistic__penalty': 'l1'}             0.7125
15   {'logistic__C': 7.78, 'logistic__penalty': 'l2'}             0.7125
16   {'logistic__C': 8.89, 'logistic__penalty': 'l1'}             0.7125
17   {'logistic__C': 8.89, 'logistic__penalty': 'l2'}             0.7125
18   {'logistic__C': 10.0, 'logistic__penalty': 'l1'}             0.7125
19   {'logistic__C': 10.0, 'logistic__penalty': 'l2'}             0.7125

    split1_test_score  split2_test_score  split3_test_score  \
0              0.6875             0.6875             0.6750
1              0.7500             0.7000             0.6875
2              0.7375             0.7000             0.6875
3              0.7375             0.7000             0.6875
4              0.7500             0.7000             0.6875
5              0.7375             0.7000             0.6875
6              0.7375             0.7000             0.6875
7              0.7375             0.7000             0.6875
8              0.7375             0.7000             0.6875
9              0.7375             0.7000             0.6875
10             0.7375             0.7000             0.6875
11             0.7375             0.7000             0.6875
12             0.7375             0.7000             0.6875
13             0.7375             0.7000             0.6875
14             0.7375             0.7000             0.6875
15             0.7375             0.7000             0.6875
16             0.7375             0.7000             0.6875
17             0.7375             0.7000             0.6875
18             0.7375             0.7000             0.6875
19             0.7375             0.7000             0.6875

    split4_test_score  mean_test_score  std_test_score  rank_test_score
0              0.6750           0.6825        0.006124               20
1              0.6875           0.7075        0.023184               14
2              0.6875           0.7050        0.018708               16
3              0.6875           0.7050        0.018708               16
4              0.6875           0.7075        0.023184               14
```

```
5              0.7000           0.7075        0.016956                1
6              0.6875           0.7050        0.018708               16
7              0.7000           0.7075        0.016956                1
8              0.6875           0.7050        0.018708               16
9              0.7000           0.7075        0.016956                1
10             0.7000           0.7075        0.016956                1
11             0.7000           0.7075        0.016956                1
12             0.7000           0.7075        0.016956                1
13             0.7000           0.7075        0.016956                1
14             0.7000           0.7075        0.016956                1
15             0.7000           0.7075        0.016956                1
16             0.7000           0.7075        0.016956                1
17             0.7000           0.7075        0.016956                1
18             0.7000           0.7075        0.016956                1
19             0.7000           0.7075        0.016956                1
```

For a combination of C and penality values, we have created 10 x 2 x 5 = 100 model candidates from which the best model was selected. On the basis of above cross validation results, we then choose the model that ranked number one.

[26]:
```python
# Model Params
print(f"Best Penalty: {best_model.best_params_['logistic__penalty']}")
print(f"Best C: {best_model.best_params_['logistic__C']}")
print(f"Best Score: {best_model.best_score_:.04}")
```

```
Best Penalty: l2
Best C: 2.23
Best Score: 0.7075
```

References

- Scikit-learn GridSearchCV
- Scikit-learn KNN
- Python resources