ml_lab_03_02_class_with_tuning_nocode

October 14, 2020

1 Classification with hyperparameter tuning

1.0.1 Aim: Show classification with different strategies for the tuning and evaluation of the classifier

1. simple **holdout**
2. **holdout with validation** train and validate repeatedly changing a hyperparameter, to find the value giving the best score, then test for the final score
3. **cross validation** on training set, then score on test set
4. **bagging** it is an *ensemble* method made available in `scikit-learn`

NB: You should not interpret those experiments as a way to find the *best* evaluation method, but simply as examples of *how* to do the evaluation.

If you look at the final report, methods **1** to **3** are meant for increasing evaluation reliability, method **3** is the more reliable, but it requires several repetitions for cross validation, therefore, if the learning method is expensive, it requires long processing time. If, due to intrinsic variation caused by random sampling, it turns out that methods **1** or **2** give higher accuracy, this means simply that the forecast towards generalisation is less reliable.

Method **4** is on a different dimension, it simply shows that a good result can be obtained with an *ensemble* of simpler classifiers (the best value for the hyperparameter `max_depth` is smaller than in the other cases)

1.0.2 Workflow

- download the data
- drop the useless data
- separe the predicting attributes X from the class attribute y
- split X and y into training and test

- part 1 - single run with default parameters
  - initialise an estimator with the chosen model generator
  - fit the estimator with the training part of X
  - show the tree structure
  - part 1.1
    - predict the y values with the fitted estimator and the train data
      - compare the predicted values with the true ones and compute the accuracy on the training set

    - part 1.2
        * predict the y values with the fitted estimator and the test data
            · compare the predicted values with the true ones and compute the accuracy on the test set

- part 2 - multiple runs changing a parameter
    - split the train part in to `train_t` and `val`
    - prepare the structure to hold the accuracy data for the multiple runs
    - repeat for all the values of the parameter
        * initialise an estimator with the current parameter value
        * fit the estimator with the train_t part
        * predict the class for the val part
        * compute the accuracy and store the value
    - find the parameter value for the top accuracy
    - fit the model for the entire train part using the best parameter
    - evaluate the fitted model with the test part

- part 3 - compute accuracy with cross validation
    - prepare the structure to hold the accuracy data for the multiple runs
    - repeat for all the values of the parameter
        * initialise an estimator with the current parameter value
        * compute the accuracy with cross validation and store the value
    - find the parameter value for the top accuracy
    - fit the estimator with the entire X
    - show the resulting tree and classification report

The data are already in your folder, use the name `winequality-red.csv`

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import tree
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import BaggingClassifier

%matplotlib inline
plt.rcParams['figure.figsize'] = [20, 20]
random_state = 15
np.random.seed(random_state)
# the random state is reset here in numpy, all the scikit-learn procedure use␣
 ↪the numpy random state
# obviously the experiment can be repeated exactly only with a complete run of␣
 ↪the program

data_url = "winequality-red.csv"
target_name = 'quality'
```

Read the data into a dataframe and show the size

```
[1]:
```

```
Shape of the input data (1599, 12)
```

Have a quick look to the data. - use the .shape attribute to see the size - use the `.head()` function to see column names and some data - use the `.hist()` method for an histogram of the columns - use the .unique method to see the class values

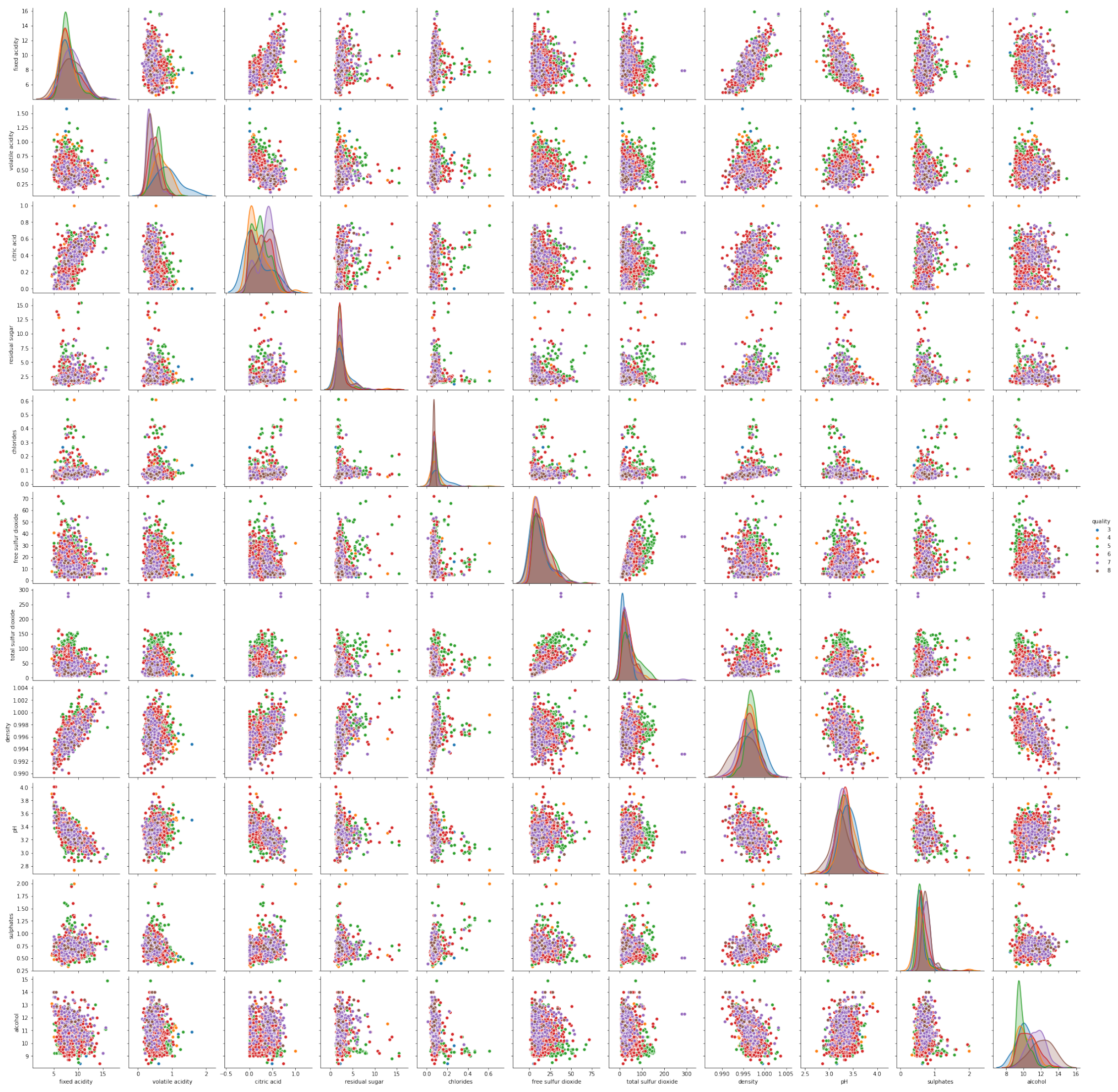
```
[2]:
```

```
[2]:    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
     0            7.4              0.70         0.00             1.9      0.076
     1            7.8              0.88         0.00             2.6      0.098
     2            7.8              0.76         0.04             2.3      0.092
     3           11.2              0.28         0.56             1.9      0.075
     4            7.4              0.70         0.00             1.9      0.076

        free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
     0                 11.0                  34.0   0.9978  3.51       0.56
     1                 25.0                  67.0   0.9968  3.20       0.68
     2                 15.0                  54.0   0.9970  3.26       0.65
     3                 17.0                  60.0   0.9980  3.16       0.58
     4                 11.0                  34.0   0.9978  3.51       0.56

        alcohol  quality
     0      9.4        5
     1      9.8        5
     2      9.8        5
     3      9.8        6
     4      9.4        5
```

Use `seaborn.pairplot` to show the the pairplots of the attributes, using the target as `hue`

NB: a semicolon at the end of a statement suppresses the `Out[]`

```
[3]:
```



Print the unique class labels (hint: use the `unique` method of pandas Series)

[4]:

```
```

```
[3 4 5 6 7 8]
```

Split the data into the predicting values X and the class y Drop also the columns which are not relevant for training a classifier, if any

The method “drop” of dataframes allows to drop either rows or columns - the “axis” parameter chooses between dropping rows (axis=0) or columns (axis=1)

[5]:

```
```

Another quick look to data

```
[6]: # X
```

```
[6]:    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
     0            7.4              0.70         0.00             1.9      0.076
     1            7.8              0.88         0.00             2.6      0.098
     2            7.8              0.76         0.04             2.3      0.092
     3           11.2              0.28         0.56             1.9      0.075
     4            7.4              0.70         0.00             1.9      0.076

        free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
     0                 11.0                  34.0   0.9978  3.51       0.56
     1                 25.0                  67.0   0.9968  3.20       0.68
     2                 15.0                  54.0   0.9970  3.26       0.65
     3                 17.0                  60.0   0.9980  3.16       0.58
     4                 11.0                  34.0   0.9978  3.51       0.56

        alcohol
     0      9.4
     1      9.8
     2      9.8
     3      9.8
     4      9.4
```

```
[7]: # y
```

```
[7]: 0    5
     1    5
     2    5
     3    6
     4    5
     Name: quality, dtype: int64
```

1.1 Prepare a simple model selection: holdout method

- Split X and y in train and test
- Show the number of samples in train and test, show the number of features

```
[8]:
```

```
There are 1055 samples in the training dataset
There are 544 samples in the testing dataset
Each sample has 11 features
```

1.2 Part 1

- Initialize an estimator with the required model generator `tree.DecisionTreeClassifier(criterion="entropy")`
- Fit the estimator on the train data and target

[9]:
```

```

1.2.1 Part 1.1

Let's see how it works on training data - predict the target using the fitted estimator on the training data - compute the accuracy on the training set using `accuracy_score(<target>,<predicted_target) * 100`

[10]:
```

```

```
The accuracy on training set is 100.0%
```

1.2.2 Part 1.2

Let's see how it works on test data, and, comparing with the result on training data, see if you can suspect *overfitting* - use the fitted estimator to predict using the test features - compute the accuracy and store it on a variable for the final summary - store the maximum depth of the tree, for later use - `fitted_max_depth = estimator.tree_.max_depth` - store the range of the parameter which will be used for tuning - `parameter_values = range(1,fitted_max_depth+1)` - print the accuracy on the test set and the maximum depth of the tree

[11]:
```

```

```
The accuracy on test set is 57.9%
The maximum depth of the tree fitted on X_train is 19
```

1.3 Part 2

Optimising the tree: limit the maximum tree depth. We will use the three way splitting: `train, validation, test`. For simplicity, since we already splitted in *train* and *test*, we will furtherly split the *train* - split the training set into two parts: **train_t** and **val** - max_depth - pruning the tree cutting the branches which exceed max_depth - the experiment is repeated varying the parameter from 1 to the depth of the unpruned tree - the scores for the various values are collected and plotted

[12]:
```

```

```
There are 791 samples in the train_t dataset
There are 264 samples in the val dataset
```

Initialize and fit an estimator.

Prepare the range `parameter_values` of values for the `max_dept` parameter of the estimator: the range will start with 1 and end with the maximum depth of the unconstrained fitted tree.

That maximum depth is found in the fitted model, in the attribute `tree_.max_depth`

[13]:
```

```

1.3.1 Loop for computing the score varying the hyperparameter

- initialise a list to contain the scores
- loop varying `par` in `parameter_values`
  - initialize an estimator with a DecisionTreeClassifier, using `par` as maximum depth and `entropy` as criterion
  - fit the estimator on the `train_t` part of the features and the target
  - predict with the estimator using the validation features
  - compute the score comparing the prediction with the validation target and append it to the end of the list

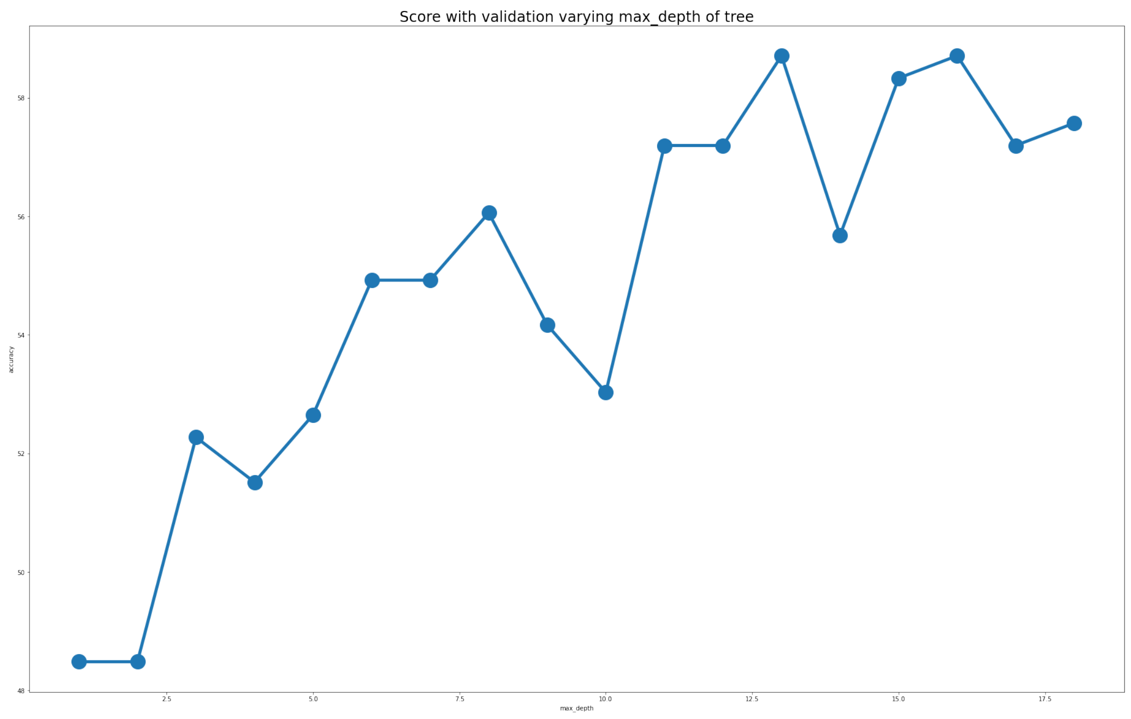
[14]:
```

```

1.3.2 Plot the results

Plot using the `parameter_values` and the list of `scores`

For your reference, the code below is a simple way to plot data

[15]:
```
plt.figure(figsize=(32,20))
plt.plot(parameter_values, scores, '-o', linewidth=5, markersize=24)
plt.xlabel('max_depth')
plt.ylabel('accuracy')
plt.title("Score with validation varying max_depth of tree", fontsize = 24)
plt.show();
```



The function np.argmax applied to a subscriptable object (such as a list, an array or a range) gives the index of the maximum value.

Apply argmax to the list of scores and use the returned index to extract from the `parameter_values` the value of the hyperparameter `max_depth` giving the best accuracy

[16]:

```
Validation: The best accuracy is obtained with MAX_DEPTH=13
```

1.3.3 Fit the tree after validation and print summary

- store the parameter value giving the best score with `np.argmax(scores)`
- initialize an estimator as a DecisionTreeClassifier, using the best parameter value computed above as maximum depth and `entropy` as criterion
- fit the estimator using the `train` part
- use the fitted estimator to predict the test data
- compute the accuracy on the test and store it on a variable for the final summary
- print the accuracy on the test set and the best parameter value

[17]:

```
The accuracy after optimization with validation is 57.9%
Obtained with max_depth = 13
```

1.4 Part 3 - Tuning with Cross Validation

Optimisation of the hyperparameter with **cross validation**. Now we will tune the hyperparameter looping on cross validation with the **training set**, then we will fit the estimator on the training set and evaluate the performance on the **test set**

- initialize an empty list for the scores
- loop varying `par` in `parameter_values`
  - initialize an estimator with a DecisionTreeClassifier, using `par` as maximum depth and `entropy` as criterion
  - compute the score using the estimator on the `train` part of the features and the target using
    - `cross_val_score(estimator, X_train, y_train, scoring='accuracy', cv = 5)`
    - the result is list of scores
  - compute the average of the scores and append it to the end of the list
- print the scores

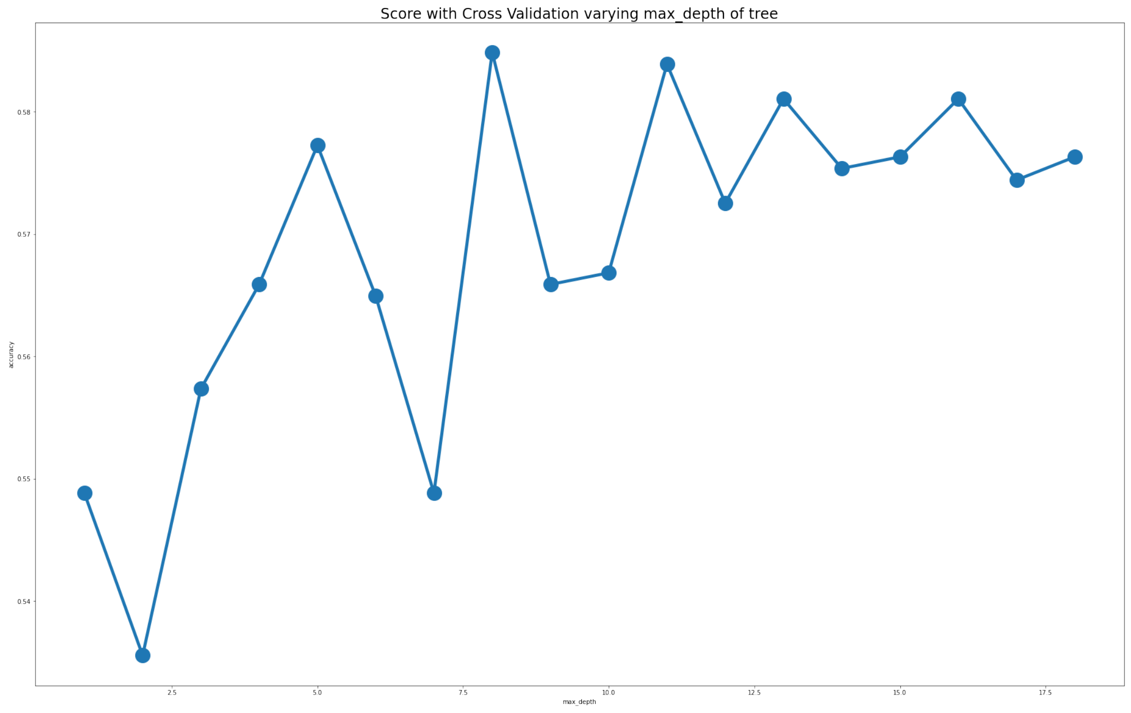
[18]:

```
[0.5488151658767773, 0.5355450236966824, 0.5573459715639811, 0.5658767772511849,
0.5772511848341232, 0.5649289099526066, 0.5488151658767773, 0.584834123222749,
0.5658767772511848, 0.5668246445497631, 0.5838862559241706, 0.5725118483412323,
0.581042654028436, 0.5753554502369668, 0.576303317535545, 0.581042654028436,
0.5744075829383886, 0.576303317535545]
```

Plot using the `parameter_values` and the list of `scores`

[19]:

```
```



1.4.1 Fit the tree after cross validation and print summary

- store the parameter value giving the best score with `np.argmax(scores)`
- initialize an estimator as a DecisionTreeClassifier, using the best parameter value computed above as maximum depth and `entropy` as criterion
- fit the estimator using the `train` part
- use the fitted estimator to predict using the test features
- compute the accuracy on the test and store it on a variable for the final summary
- print the accuracy on the test set and the best parameter value

[20]:

```
```

```
The accuracy on test set tuned with cross_validation is 55.7% with depth 8
```

Show a more detailed information using the `classification_report` function of `sklearn.metrics`, using the true and predicted target values

[21]:

```
```

```
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         5
           4       0.00      0.00      0.00        18
           5       0.64      0.66      0.65       227
           6       0.54      0.57      0.55       223
           7       0.42      0.38      0.40        65
           8       0.00      0.00      0.00         6

    accuracy                           0.56       544
   macro avg       0.27      0.27      0.27       544
weighted avg       0.54      0.56      0.55       544
```

- **micro**: Calculate metrics globally by counting the total true positives, false negatives and false positives.
- **macro**: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
- **weighted**: Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

Print the `confusion_matrix`, using the function of `sklearn.metrics`

[22]:

```
```

```
[[  0   0   5   0   0   0]
 [  0   0   8   9   1   0]
 [  0   7 150  64   6   0]
 [  1   2  67 128  23   2]
 [  0   0   3  36  25   1]
 [  0   0   0   2   4   0]]
```

1.4.2 Final report

Print a summary of the four experiments

[23]:

```
```

```
                                        Accuracy   Hyperparameter
Simple HoldOut and full tree        :   57.9%      18
HoldOut and tuning on validation set:   57.9%      13
CrossValidation and tuning          :   55.7%      8
```

[24]:

```
import sklearn
print('The scikit-learn version is {}.'.format(sklearn.__version__))
```

```
The scikit-learn version is 0.23.1.
```

1.4.3 Suggested exercises

- try other datasets
- try to optimise the parameters "min_impurity_decrease"

- try to optimise using ‘gini’ instead of ‘entropy’