PennState World Campus

DAAN862: Analytics Programming in Python

Lesson 8: Supervised Learning with SciKit-Learn I: Classification Models

---

Lesson 8: Supervised Learning with SciKit-Learn I: Classification Models

---

Since the popular classification algorithms have been introduced in the previous course IE 575, we will not go over how these algorithms work.

At the end of this lesson, using scikit-learn, the student will be able to:

Build and evaluate Logistic Regression models

Build and evaluate Decision tree models

Build and evaluate Naive bayes models

Build and evaluate Neural network models

By the end of this lesson, please complete all readings and assignments found in the Lesson 8 Course Schedule.

Lesson 8.1: Logistic Regression

The class *LogisticRegression* in scikit-learn can be used in binary, multinomial, or One-vs-Rest (https://en.wikipedia.org/wiki/Multiclass_classification) classification problems.

The first step is to load the necessary packages into Python as seen in Figure 8.1:

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: import os
   ...: from sklearn.model_selection import train_test_split
   ...: from sklearn import metrics
```

Fig 8.1 (click to enlarge)

(click the titles to learn more)

Data Preparation

For all models built in the lesson, we will use glass dataset as an example. You can download the glass.data from th is website (https://archive.ics.uci.edu/ml/datasets/glass+identification) and the description is also provided there.

Since the data file doesn't't come with column names, we need to assign column names to the Dataframe as seen in Figure 8.2:

```
In [2]: os.chdir("E:\\GoogleDrive\\PSU\\DAAN862\\Course contents\\Lesson 8")

In [3]: glass = pd.read_csv('glass.data', header = None)

In [4]: glass.columns = ['Id', 'RI', 'Na', 'Mg', 'Al', 'Si', 'K',
   ...:                  'Ca', 'Ba', 'Fe', 'Type']
```

Fig 8.2 (click to enlarge)

Let's perform some data exploratory analysis (Figure 8.3):

```
In [5]: glass = glass.drop('Id', axis=1) # remove ID column

In [6]: glass.head()
Out[6]:
        RI     Na    Mg    Al     Si     K    Ca   Ba   Fe  Type
0  1.52101  13.64  4.49  1.10  71.78  0.06  8.75  0.0  0.0     1
1  1.51761  13.89  3.60  1.36  72.73  0.48  7.83  0.0  0.0     1
2  1.51618  13.53  3.55  1.54  72.99  0.39  7.78  0.0  0.0     1
3  1.51766  13.21  3.69  1.29  72.61  0.57  8.22  0.0  0.0     1
4  1.51742  13.27  3.62  1.24  73.08  0.55  8.07  0.0  0.0     1

In [7]: glass.describe()
Out[7]:
               RI          Na     ...              Fe        Type
count  214.000000  214.000000     ...      214.000000  214.000000
mean     1.518365   13.407850     ...        0.057009    2.780374
std      0.003037    0.816604     ...        0.097439    2.103739
min      1.511150   10.730000     ...        0.000000    1.000000
25%      1.516523   12.907500     ...        0.000000    1.000000
50%      1.517680   13.300000     ...        0.000000    2.000000
75%      1.519157   13.825000     ...        0.100000    3.000000
max      1.533930   17.380000     ...        0.510000    7.000000

[8 rows x 10 columns]

In [8]: glass.Type.value_counts()
Out[8]:
2    76
1    70
7    29
3    17
5    13
6     9
Name: Type, dtype: int64

In [9]: glass.isnull().sum().sum()
Out[9]: 0
```

Fig 8.3 (click to enlarge)

See Figure 8.4 for to see how we calculate the correlation matrix:

```
In [10]: glass.corr()
Out[10]:
            RI        Na        Mg     ...           Ba        Fe      Type
RI    1.000000 -0.191885 -0.122274     ...    -0.000386  0.143010 -0.164237
Na   -0.191885  1.000000 -0.273732     ...     0.326603 -0.241346  0.502898
Mg   -0.122274 -0.273732  1.000000     ...    -0.492262  0.083060 -0.744993
Al   -0.407326  0.156794 -0.481799     ...     0.479404 -0.074402  0.598829
Si   -0.542052 -0.069809 -0.165927     ...    -0.102151 -0.094201  0.151565
K    -0.289833 -0.266087  0.005396     ...    -0.042618 -0.007719 -0.010054
Ca    0.810403 -0.275442 -0.443750     ...    -0.112841  0.124968  0.000952
Ba   -0.000386  0.326603 -0.492262     ...     1.000000 -0.058692  0.575161
Fe    0.143010 -0.241346  0.083060     ...    -0.058692  1.000000 -0.188278
Type -0.164237  0.502898 -0.744993     ...     0.575161 -0.188278  1.000000

[10 rows x 10 columns]
```

Fig 8.4 (click to enlarge)

If you prefer to use a plot to show the correlations, you can use the codes seen in Figures 8.5 A & B:

```
In [11]: plt.matshow(glass.corr())
    ...: plt.title('Correlation Matrix', position = (0.5, 1.1))
    ...: plt.colorbar()
    ...: plt.xticks(range(11), list(glass.columns))
    ...: plt.yticks(range(11), list(glass.columns))
    ...: plt.ylabel('True label')
    ...: plt.xlabel('Predicted lable')
Out[11]: Text(0.5,0,'Predicted lable')
```

Fig 8.5 (A) (click to enlarge)

Fig 8.5 (B)

Next, we split the data into training and test sets (as seen in Figure 8.6):

```
In [10]: X = glass.iloc[:, 0 : 9]

In [11]: y = glass.Type

In [12]: X_train, X_test, y_train, y_test = train_test_split(
    ...:         X, y, test_size=0.3, random_state=34)
```

Fig 8.6 (click to enlarge)

Model Generation

Since scikit-learn is a big package, we only import the model we want to use (as seen in Figure 8.7)

```
In [13]: from sklearn import linear_model
```

Fig 8.7 (click to enlarge)

The machine learning algorithms in sklearn are objects. You need to create the classifier first (without providing the training data at this time). If you want to use default parameters for the logistic regression function, you can create a classifier as seen in Figure 8.8:

```
In [14]: lr = linear_model.LogisticRegression()
```

Fig 8.8 (click to enlarge)

If you don’t want to use the default value, you can specify the parameters in parenthesis. The main parameters of logistic regression are listed below in Table 8.2.1:

Table 8.2.1 Main parameter for LostisticRegression class

Parameters	Description
penalty	str, 'l1' or 'l2', default: 'l2' Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties.
tol	float, default: 1e-4 Tolerance for stopping criteria.
c	Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.
multi_class	str, {'ovr', 'multinomial'}, default: 'ovr' Multiclass option can be either 'ovr' or 'multinomial'. If the option chosen is 'ovr', then a binary problem is fit for each label.

All model building and evaluations are defined as methods to the objects. The main methods for the class are listed in Table 8.2.2. These are standard methods for all models in sklearn*. Once you learned the standard process, you can easily apply to other algorithms.

Methods	Description
decision_function(X)	Predict confidence scores for samples.
densify()	Convert coefficient matrix to dense array format.
fit(X, y[, sample_weight])	Fit the model according to the given training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class labels for samples in X.
predict_log_proba(X)	Log of probability estimates.
predict_proba(X)	Probability estimates.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.

Methods	Description
set_params(params)**	Set the parameters of this estimator.
sparsify()	Convert coefficient matrix to sparse format.

*Since these methods are similar for all classification, regression, and clustering models, we are not going to list them again for the rest of the algorithms.

For all models in sklearn package, we typically use the *fit* method to build the model on training sets, then use the *predict* method to predict the class values for test sets (see Figure 8.9):

```
In [15]: lr.fit(X_train, y_train)
Out[15]:
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

Fig 8.9 (click to enlarge)

For multi-class classification, the logistic function will build several models. The slopes and the intercepts are listed below in Figure 8.10:

```
In [16]: lr.coef_
Out[16]:
array([[-0.03325984, -0.77740596,  1.40781807, -1.7451754 ,  0.10053265,
        -0.34840158,  0.0447264 ,  0.14748637, -0.33073295],
       [ 0.04928365, -0.48762456,  0.43623008,  0.88941578,  0.00728638,
         0.11324937,  0.29550609, -0.53547888,  0.75435463],
       [ 0.00267277,  0.51928299,  0.85747563, -0.22991605, -0.17826714,
        -0.16269428,  0.13161504, -0.35530477,  0.1122821 ],
       [ 0.006872  , -0.64841573, -0.49545987,  0.9843578 ,  0.03024727,
         1.19392478,  0.24644452, -0.36235022, -0.58781227],
       [-0.05763563,  0.82493412, -0.75033562, -0.62436306, -0.13677465,
        -0.97544592, -0.13931342, -1.5665454 , -0.18995061],
       [ 0.01155696,  0.8744333 , -0.91989761,  1.46270511, -0.16328497,
        -0.83151195, -0.35415806,  1.01071993, -0.10809724]])

In [17]: lr.intercept_
Out[17]:
array([-0.02345114,  0.02975752,  0.00620605,  0.0073286 , -0.03489365,
        0.00559849])
```

Fig 8.10 (click to enlarge)

For multi-class classification, the logistic function will use one-vs-rest by default, and it will build 6 models instead of one, that's why the coefficients are a 6x9 array instead of a 1*9 array.

Model Evaluation

Now, we can predict the labels for training and test sets and evaluate the model performance with accuracy and confusion matrix as seen in Figures 8.11 A, B, & C:

```
In [18]: lr_train_pred = lr.predict(X_train)

In [19]: lr_test_pred = lr.predict(X_test)
```

Fig 8.11 (A) (click to enlarge)

```
In [20]: metrics.accuracy_score(y_train, lr_train_pred)
Out[20]: 0.6510067114093959

In [21]: metrics.accuracy_score(y_test, lr_test_pred)
Out[21]: 0.5230769230769231
```

Fig 8.11 (B) (click to enlarge)

```
In [22]: train_cm = metrics.confusion_matrix(y_train, lr_train_pred)

In [23]: train_cm
Out[23]:
array([[44,  7,  0,  0,  0,  0],
       [19, 29,  0,  1,  1,  0],
       [ 9,  3,  0,  0,  0,  0],
       [ 0,  5,  0,  1,  0,  1],
       [ 0,  2,  0,  0,  2,  3],
       [ 0,  1,  0,  0,  0, 21]], dtype=int64)

In [24]: test_cm = metrics.confusion_matrix(y_test, lr_test_pred)

In [25]: test_cm
Out[25]:
array([[14,  5,  0,  0,  0,  0],
       [11, 14,  0,  0,  1,  0],
       [ 4,  1,  0,  0,  0,  0],
       [ 0,  4,  0,  2,  0,  0],
       [ 0,  2,  0,  0,  0,  0],
       [ 1,  1,  0,  1,  0,  4]], dtype=int64)
```

Fig 8.11 (C) (click to enlarge)

Figure 8.12 A & B shows how to plot the confusion matrix:

Fig 8.12

(A)

(click to

enlarge)

Fig. 8.12 (B)

Transcript

In lesson 8, I will introduce four classification models with sklearn package. Sklearn provides a standard way to build and evaluate models. Once you learn one, it is easy to transfer to other models.

In this video, I will introduce logistic regression. First, let's import the necessary packages: pandas, numpy, matplotlib, os, and import train_test_split and metrics from sklearn.

For lesson 8, we will use glass as the example. Here we change working directory and load the data into Python. The data doesn't have a column name. So we need to assign the column names to it.

ID is irrelevant to data mining task, we need to drop it.

We can explore the data, this is the head, statistic analysis, use value_Counts to check the categories and distribution. If there are missing values. Correlation of variables, You can try to plot the correlation matrix using matshow. Adding a title, color bar, change x, y ticks and x, y labels.

Next, we need to split the data into train and test, this has been introduced in lesson 7. Here we use 30 percents as a test set.

LogestiRegression model is under sklean linear_model, we can import linear_model from sklean.

Line 50 is used to initiate the model, if you want to change argument, you need to do it here.

Now, if you type “lr.” and press “Tab”, all functions associated with lr will be listed here. There are very similar to all models. Fit is used to build the model with supplied data. Line 51 is used X_Train and y_train to build the model.

For the linear model, you can use coef_ and intercept_ to check the coefficients and intercept.

No you can evaluate your model on both train and test set.

Predict function will predict the class for given data. Here we predict the class label for both train and test data, you can evaluate the accuracy by using metrics.accuray_score by providing the true label and predicted label.

Line 60-63 are confusion matrices for both train and test data.

You can also use matshow to plot confusion matrix.

Lesson 8.2: Naïve Bayes

The second model we introduce is Naïve Bayes, and we still use the glass data. Once you import the model, the rest is similar to how to build and evaluate the Logistic Regression Model:

(click the titles to learn more)

Model Generation

GaussianNB is one of the Naïve Bayes models in scikit=learn. You can crate a GaussianNB object, then fit it with train datasets (see Figure 8.13).

```
In [32]: from sklearn.naive_bayes import GaussianNB

In [33]: NB = GaussianNB()

In [34]: NB.fit(X_train, y_train)
Out[34]: GaussianNB(priors=None)
```

Fig 8.13 (click to enlarge)

For the Naïve Bayes model, it only has one parameter:

- Priors: Priori probabilities of the classes.

Since this information of the classes is unknown, we don't specify this parameter in the model.

Model Evaluation

The model evaluation process is similar to what you did in lesson 8.1. You need to predict the target labels for both train and test data, and then calculate their accuracy and confusion matrix (see Figures 8.14 A, B, & C):

Fig 8.14 (A) (click to enlarge)

```
In [37]: metrics.accuracy_score(y_train, NB_train_pred)
Out[37]: 0.5906040268456376

In [38]: metrics.accuracy_score(y_test, NB_test_pred)
Out[38]: 0.38461538461538464
```

Fig 8.14 (B) (click to enlarge)

```
In [39]: metrics.confusion_matrix(y_train, NB_train_pred)
Out[39]:
array([[41,  2,  7,  1,  0,  0],
       [32,  9,  1,  7,  1,  0],
       [ 5,  0,  6,  0,  1,  0],
       [ 0,  0,  0,  7,  0,  0],
       [ 0,  0,  0,  0,  7,  0],
       [ 0,  0,  0,  4,  0, 18]], dtype=int64)

In [40]: metrics.confusion_matrix(y_test, NB_test_pred)
Out[40]:
array([[13,  2,  3,  0,  1,  0],
       [20,  2,  1,  1,  2,  0],
       [ 4,  0,  1,  0,  0,  0],
       [ 0,  2,  0,  4,  0,  0],
       [ 0,  0,  0,  0,  2,  0],
       [ 1,  0,  0,  3,  0,  3]], dtype=int64)
```

Fig 8.14 (C) (click to enlarge)

The Bayesian model can also provide the probability to each class instead of final labels. Figure 8.15 shows how to compute the probabilities for the top 5 rows of test data by using the *predict_proba* method:

```
In [41]: np.set_printoptions(precision=2)  # display 2 decimal places

In [42]: NB.predict_proba(X_test[:5])
Out[42]:
array([[7.08e-001, 1.71e-002, 2.71e-001, 3.99e-003, 0.00e+000, 1.63e-009],
       [8.71e-001, 1.19e-001, 1.09e-002, 0.00e+000, 0.00e+000, 2.56e-021],
       [3.86e-077, 3.73e-008, 4.42e-132, 2.41e-002, 0.00e+000, 9.76e-001],
       [2.63e-001, 1.82e-002, 7.18e-001, 0.00e+000, 0.00e+000, 3.53e-026],
       [7.86e-001, 4.08e-003, 2.09e-001, 6.13e-004, 0.00e+000, 3.13e-009]])
```

Fig 8.15 (click to enlarge)

Additional Practice

I will gradually introduce advanced model comparison and parameter tuning in each lesson. Here, we are going to use both linear regression and Naïve Bayes models to demonstrate the relationship between test accuracy and training sample size.

The test used are 0.1, 0.15, 0.2, ..., 0.9. Here we perform tran_test_split 30 times, and use the average accuracy as the final accuracy for each trainning size. Because of the random split, the data cannot estimate the model accuracy accurately.

For each training data. And the accuracies for each train size are saved in lr_scores and nb_scores. (See Figure 8.16)

```
In [56]: lr = linear_model.LogisticRegression()

In [57]: nb = GaussianNB()

In [58]: lr_scores = []

In [59]: nb_scores = []

In [60]: test_sizes = np.linspace(0.1, 0.6, 11)

In [61]: test_sizes
Out[61]: array([0.1 , 0.15, 0.2 , 0.25, 0.3 , 0.35, 0.4 , 0.45, 0.5 , 0.55, 0.6 ])

In [62]: for test_size in test_sizes:
    ...:     nb_accuracy = []
    ...:     lr_accuracy = []
    ...:     for i in range(30):
    ...:         X_trains, X_tests, y_trains, y_tests = train_test_split(
    ...:                 X, y, test_size = test_size)
    ...:         nb.fit(X_trains, y_trains)
    ...:         nb_accuracy.append(nb.score(X_test, y_test))
    ...:         lr.fit(X_trains, y_trains)
    ...:         lr_accuracy.append(lr.score(X_test, y_test))
    ...:     nb_scores.append(np.mean(nb_accuracy))
    ...:     lr_scores.append(np.mean(lr_accuracy))
    ...:
```

Fig 8.16 (click to enlarge)

In order to show the relationship, we use line plots for both models as seen in Figures 8.17 A & B. We can see that the accuracy for the logistic regression increases with the trainning data size, while Naive Bayes model is less affected by the train size.

```
In [65]: plt.figure()
    ...: plt.plot(1-test_sizes, nb_scores, label='Naive Bayes')
    ...: plt.plot(1-test_sizes, lr_scores, linestyle='--',
label='LogisticRegression')
    ...: plt.xlabel("Train size (Percentage)")
    ...: plt.ylabel("Test set accuracy")
    ...: plt.legend()
Out[65]: <matplotlib.legend.Legend at 0x2080b50fa20>
```

Fig. 8.17 (A) (click to enlarge)

Fig. 8.17 (B)

Transcript

We can import GaussianNB form skleanr.naive_bayes

Initiate a GaussianNB model called NB

Now you can fit the model with X_train and y_train.

The evaluation is exactly the same with logistic regression models.

Here is the accuracy and confusion matrix for both train and test sets.

For this model, it can also give the probability of the class label by using the predict_proba function.

Here we show the first 5 rows of test data.

I will gradually introduce more advanced practice.

This practice is to show the relation between test accuracy and the training sample size.

First, you need initiate two models

I prefer to use a List to record the results. It is just a personal preference, you can either use dataframe or numpy arrays. Here we initiate two empty listed called lr_scores and nb_scores.

I use numpy linspace to create linear spacing array start with 0.1, end with 0.6 and the spacing is 0.05.

For each test size, I will build 30 models to estimate average accuracy.

Nb_accuracy and lr_accuracy are used to record the accuracy for the 30 round.

For each round, we split the data into train, test sets, and build the model with x_train and y_train, then append the accuracy for the test data into nb_accuracy and lr_accuracy. After running 30 times, the mean of the accuracy is appended in nb_score and lr_score.

This is final results. Let's use a line plot to show the relation.

As you can see that for logistic regression models, the accuracy increases with the increase of training data, while the Naïve Bayes model seems less affected by the training size.

Lesson 8.3: Decision Tree

There are various decision tree algorithms such as ID3, C4.5, C5.0, and CART. *scikit-learn* provides an optimized version of the CART model.

(click the titles to learn more)

Model Generation

Import the Decision tree model and create the classifier as seen in Figure 8.18:

```
In [56]: from sklearn import tree

In [57]: DT = tree.DecisionTreeClassifier(max_depth = 10, min_samples_split = 5)

In [58]: DT.fit(X_train, y_train)
Out[58]:
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=10,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=5,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

Fig 8.18 (click to enlarge)

tree.DecisionTreeClassifer is the CART classification tree model. Here we use the parameters: *max_depth = 10 and min_samples_split = 5.*

The main parameters for decision tree model are found in table 8.3.1:

Table 8.3.1: Main Parameters for Decision Tree Model

Parameters	Description
criterion	string, optional (default=”gini”) The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain.
splitter	string, optional (default=”best”) The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split.
max depth	int or None, optional (default=None) The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

Parameters	Description
min samples split	int, float, optional (default=2) The minimum number of samples required to split an internal node: If int, then consider min_samples_split as the minimum number. If float, then min_samples_split is a percentage and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
min sample leaf	int, float, optional (default=1) The minimum number of samples required to be at a leaf node: If int, then consider min_samples_leaf as the minimum number. If float, then min_samples_leaf is a percentage and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

For more information about the classification tree, please refer to this website (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier) .

Model Evaluation

Next, we predict the glass type and evaluate the performance of the test set. This time we use a classification report which shows the model performance on each type and the overall performance (see Figure 8.19):

```
In [59]: DT_pred = DT.predict(X_test)

In [60]: metrics.accuracy_score(DT_pred, y_test)
Out[60]: 0.6615384615384615

In [61]: print(metrics.classification_report(y_test, DT_pred))
             precision    recall  f1-score   support

          1       0.65      0.58      0.61        19
          2       0.70      0.73      0.72        26
          3       0.38      0.60      0.46         5
          5       1.00      0.67      0.80         6
          6       0.40      1.00      0.57         2
          7       1.00      0.57      0.73         7

avg / total       0.71      0.66      0.67        65
```

Fig 8.19 (click to enlarge)

Decision tree model also provides an attribute to return the feature importance. Since the method returns an array, we create a DataFrame to show both the variable names and importance factors as seen in Figure 8.20:

```
In [62]: DT.feature_importances_
Out[62]: array([0.09, 0.03, 0.16, 0.2 , 0.11, 0.16, 0.01, 0.21, 0.04])

In [63]: pd.DataFrame({'variable':glass.columns[:9],
    ...:                'importance':DT.feature_importances_})
Out[63]:
  variable  importance
0       RI    0.088592
1       Na    0.029991
2       Mg    0.157680
3       Al    0.198486
4       Si    0.114687
5        K    0.158124
6       Ca    0.007584
7       Ba    0.206695
8       Fe    0.038161
```

Fig 8.20 (click to enlarge)

Visualization of Decision Tree Model

You can export the tree model in Graphviz format by using the *tree.export_graphviz* function. In order to visualize the tree, please install python-graphviz.

Please click *Start -> Anaconda 3 -> Anaconda Navigator*. (Click the arrow to view how it looks):

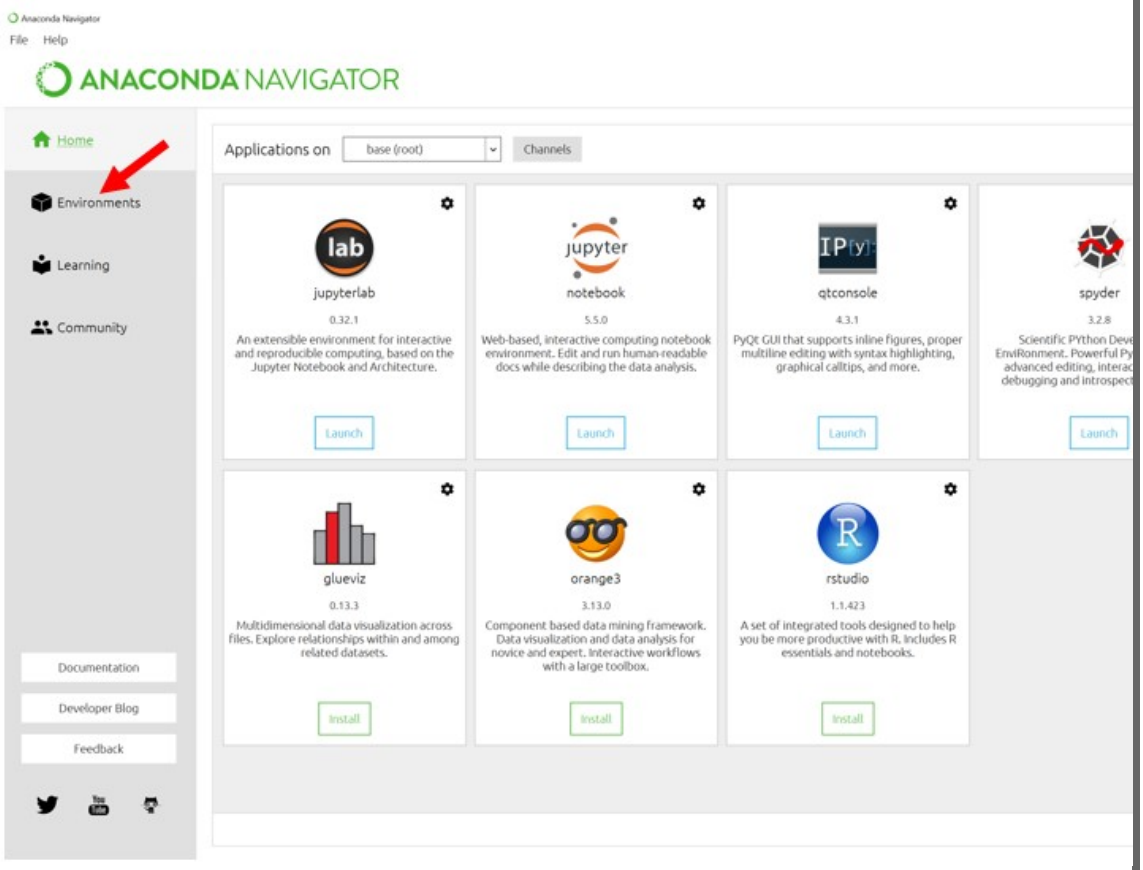

Fig 8.21 (click to enlarge)

Next, go to Environments: (click the arrow for a screenshot)

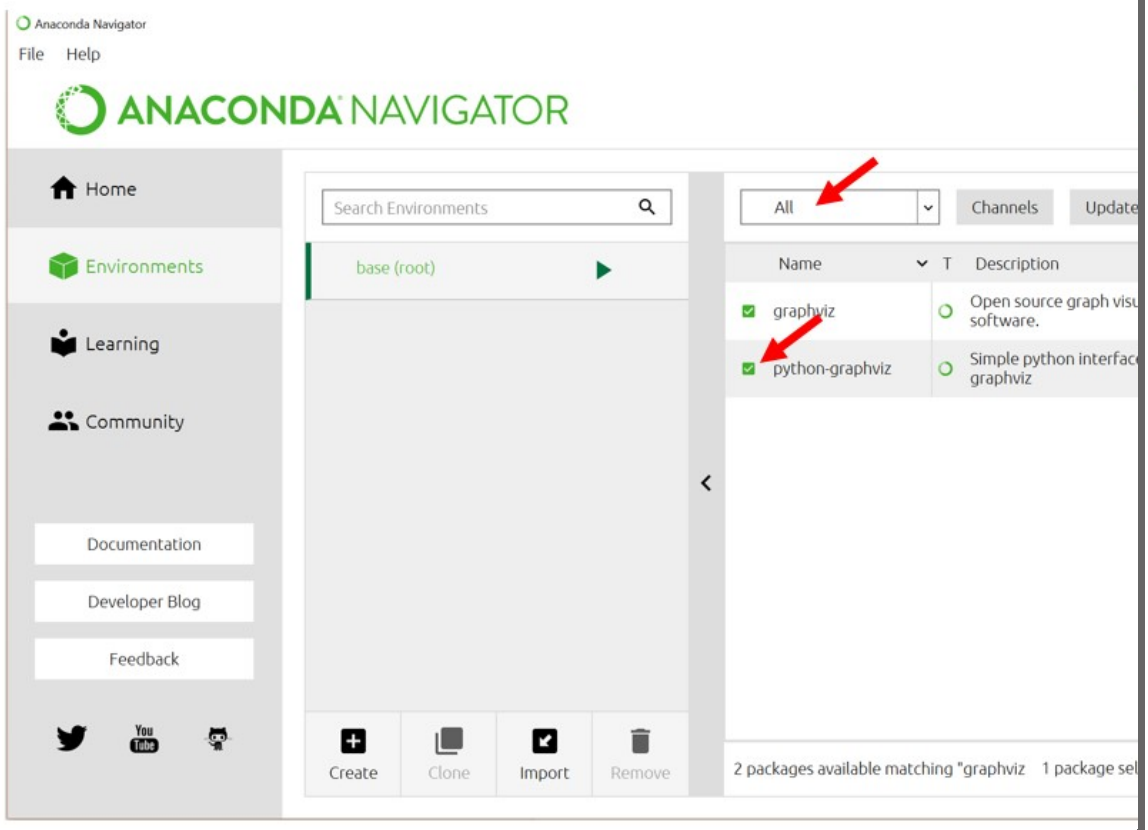

Fig 8.22 (click to enlarge)

Type ‘graphviz’ to search all packages whose names contain graphviz, then select python-graphviz and click apply to install it. Once you have installed the package, please restart the Spyder.

The *export_graphviz* function in Tree model can create a dot file of tree structures. After you import Source function in graphviz package, you can plot the dot file created as seen in Figure 8.23:

```
In [64]: from graphviz import Source

In [65]: dot_data = tree.export_graphviz(DT, out_file=None,
    ...:                                 feature_names=X_train.columns)

In [66]: Source(dot_data)
Out[66]:
```

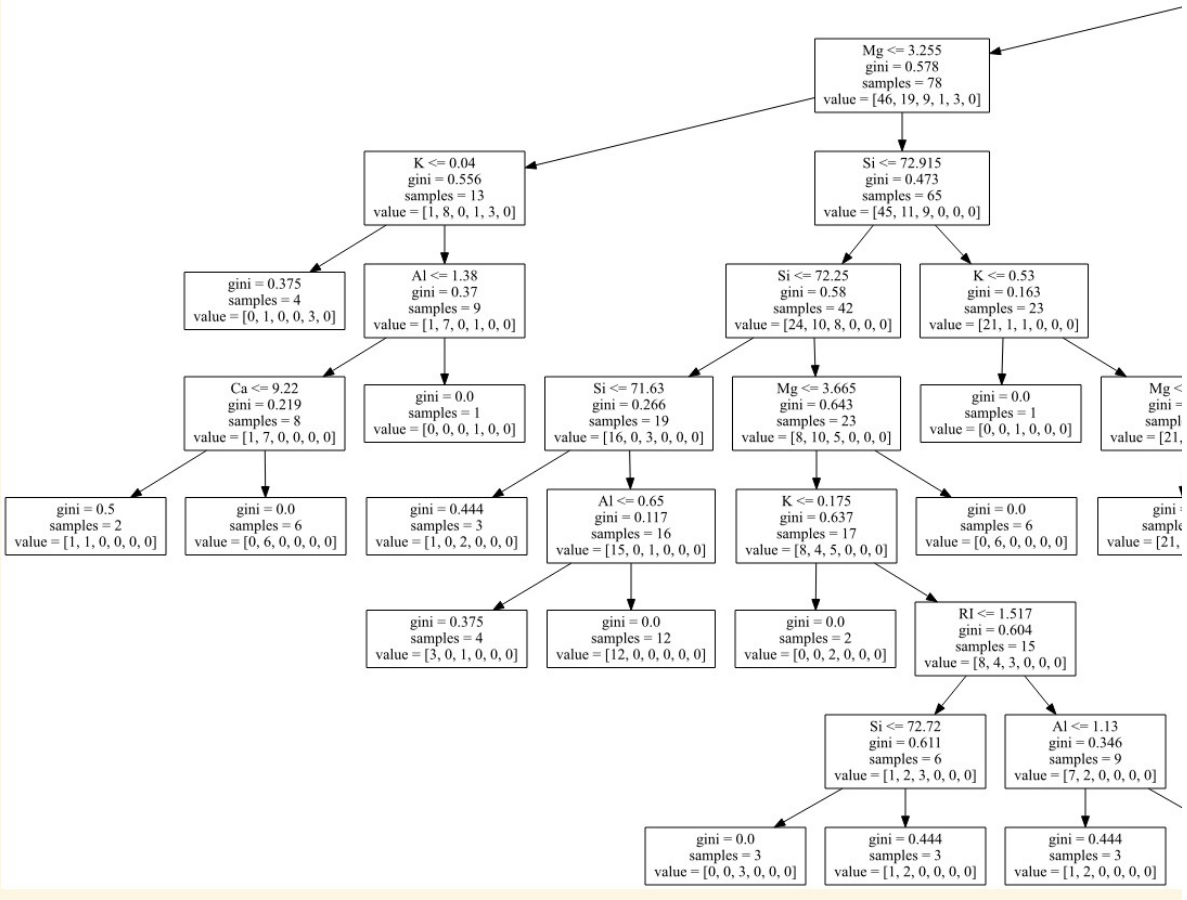

Fig 8.23 (click to enlarge)

The tree is too big to be shown in the Python console. You can export it into a pdf file (see Figures 8.24 A & B):

```
In [67]: Source(dot_data).render('tree')
```

Fig 8.24A (click to enlarge)

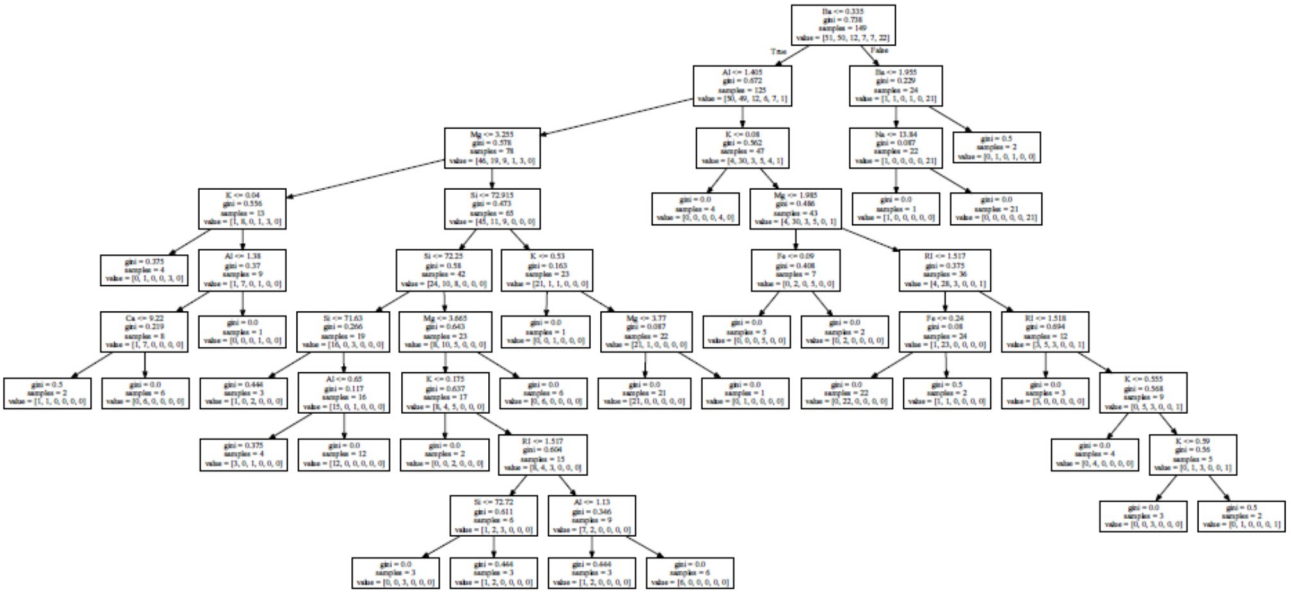

Fig 8.24B (click to enlarge)

If you would like to modify the tree plot, please refer to the parameters of the *export_graphviz* *(http://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html#sklearn.tree.export_graphviz)* function.

Transcript

First, we import all tree model from sklearn package

Here, we initiate the decision tree classifier with max_dept of 10 and min_sample_split of 5.

You can fit the tree model with train data. And here I only evaluated the test sets. Here we also tried classification report, which shows precision, recall, f1-score and support for each category.

Decision tree model provides which features are more important to the task. You can use feature_importance_ to get this information.

In order to view the tree, you need to import source from graphviz.

Tree.export_graphvix will create a dot-formated file and you can use source function to view it. If you want to export it, you can use render function which creates a pdf of tree.

Lesson 8.4: Nerual Network

(click the titles to learn more)

Data Preprocessing

For Neural network models, you need to preprocess the data. Since neural networks prefer the data with the range [0, 1], first, we are going to rescale the data with MinMaxScaler, as seen in Figure 8.25:

```
In [69]: from sklearn.preprocessing import MinMaxScaler

In [70]: scaler = MinMaxScaler()

In [71]: X_train_scaled = scaler.fit_transform(X_train)

In [72]: X_test_scaled = scaler.transform(X_test)
```

Fig 8.25 (click to enlarge)

The way to process the data is similar to build models. First, we create a MinMaxScalter class. You can fit and transform training data at the same time with *fit_transform* function. With information (minimums and maximums) learned from training data, you can transform/rescale the test data. Figure 8.26 shows the first 3 rows of training and test datasets:

```
In [73]: X_train_scaled[:3]
Out[73]:
array([[0.23, 0.4 , 0.78, 0.38, 0.51, 0.11, 0.22, 0.  , 0.27],
       [0.46, 0.41, 0.83, 0.19, 0.39, 0.03, 0.37, 0.  , 0.19],
       [0.21, 0.34, 0.8 , 0.38, 0.59, 0.11, 0.21, 0.  , 0.  ]])

In [74]: X_test_scaled[:3]
Out[74]:
array([[0.53, 0.41, 0.74, 0.29, 0.46, 0.1 , 0.29, 0.  , 0.  ],
       [0.28, 0.22, 0.72, 0.27, 0.67, 0.1 , 0.3 , 0.  , 0.65],
       [0.18, 0.62, 0.  , 0.66, 0.7 , 0.  , 0.25, 0.18, 0.  ]])
```

Fig 8.26 (click to enlarge)

Model Generation

To import the Decision tree model, create the classifier and fit the model with training dataset as seen in Figure 8.27:

```
In [75]: from sklearn.neural_network import MLPClassifier

In [76]: NN = MLPClassifier(solver = 'lbfgs', alpha = 1e-5,
    ...:                    hidden_layer_sizes = (10, 4), random_state = 1)

In [77]: NN.fit(X_train_scaled, y_train)
Out[77]:
MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(10, 4), learning_rate='constant',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=1, shuffle=True,
       solver='lbfgs', tol=0.0001, validation_fraction=0.1, verbose=False,
       warm_start=False)
```

Fig 8.27 (click to enlarge)

Table 8.4.1 shows the mian parameters for MLPClassifer:

Table 8.4.1: Main Parameters for MLP Classfier

Parameters	Description
hidden layer sizes	tuple, length = n_layers - 2, default (100,) The ith element represents the number of neurons in the ith hidden layer.
activation	{'identity', 'logistic', 'tanh', 'relu'}, default 'relu'.
learning rate	{'constant', 'invscaling', 'adaptive'}, default 'constant.

Parameters	Description
learning rate init	double, optional, default 0.001 The initial learning rate used. It controls the step-size in updating the weights.
max iter	int, optional, default 200 Maximum number of iterations.
tol	float, optional, default 1e-4 Tolerance for the optimization.

For all parameters of MLPCalssfieer, please visit this site (http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier) .

Model Evaluation

Next, we train the model and evaluate the performance of the test set as seen in Figure 8.28:

```
In [78]: NN_pred = NN.predict(X_test_scaled)

In [79]: metrics.accuracy_score(NN_pred, y_test)
Out[79]: 0.5846153846153846

In [80]: print(metrics.classification_report(y_test, NN_pred))
             precision    recall  f1-score   support

          1       0.64      0.74      0.68        19
          2       0.63      0.65      0.64        26
          3       0.00      0.00      0.00         5
          5       0.00      0.00      0.00         6
          6       1.00      1.00      1.00         2
          7       0.36      0.71      0.48         7

avg / total       0.51      0.58      0.54        65
```

Fig 8.28 (click to enlarge)

Transcript

For a neural network, we need to rescale the variables to [0, 1]. Sklearn provides various data processing tools. Here we use minmaxscaler.

Similar to creating classification models. The processors are an object too. You need to initial minmaxscaler object first. The use it to fit and transform the train data at the same time. And then use transform function to rescale the data based on the information learned in train data.

You can take a look at the head of the rescaled data.

MLPClassifer the neural_network classification model, and you can import it first. And initiate the MLPClassifer object with arguments solve = lbfgs, alpha = 1e-5, hidden layer size (10, 4)which mean the first layer has 10 nodes and the second layer has 4 nodes. Set a random state to reproduce the result. Typically, these parameters should be optimized in order to get the highest accuracy. This will be introduced in the next lesson.

Now, you can fit the data and evaluate the models. This accuracy and classification report for the test set.

Lesson 8 References (6 of 6)

Lesson 8 References

- http://scikit-learn.org/stable/supervised_learning.html#supervised-learning (http://scikit-learn.org/stable/supervised_le

arning.html#supervised-learning)

- Python Data Science Handbook, Jake VanderPlas https://jakevdp.github.io/PythonDataScienceHandbook/index.html (https://jakevdp.github.io/PythonDataScienceHandbook/index.html)
