Machine Learning Classification - Scikit Learn

There are different types of classifiers, a classifier is an algorithm that maps the input data to a specific category. Now, let us take a look at the different types of classifiers:

1. Perceptron
2. Naive Bayes
3. Decision Tree
4. Logistic Regression
5. K-Nearest Neighbor
6. Artificial Neural Networks/Deep Learning
7. Support Vector Machine

Then there are the ensemble methods: Random Forest, Bagging, AdaBoost, etc.

As we have seen before, linear models give us the same output for a given data over and over again.

Whereas, machine learning models, irrespective of classification or regression give us different results. This is because they work on random simulation when it comes to supervised learning.

In the same way Artificial Neural Networks use random weights. Whatever method you use; these machine learning models have to reach a level of accuracy of prediction with the given data input.

A machine learning project may not be linear, but it has a number of well-known steps:

1. Define Problem.
2. Prepare Data.
3. Evaluate Algorithms.
4. Improve Results.
5. Present Results.

The best way to really come to terms with a new platform or tool is to work through a machine learning project end-to-end and cover the key steps. Namely, from loading data, summarizing data, evaluating algorithms and making some predictions.

If you can do that, you have a template that you can use on dataset after dataset. You can fill in the gaps such as further data preparation and improving result tasks later, once you have more confidence.

1.Hello World of Machine Learning

The best small project to start with on a new tool is the classification of iris flowers (e.g. the iris dataset). This is a good project because it is so well understood.

- Attributes are numeric so you have to figure out how to load and handle data.
- It is a classification problem, allowing you to practice with perhaps an easier type of supervised learning algorithm.

- It is a multi-class classification problem (multi-nominal) that may require some specialized handling.
- It only has 4 attributes and 150 rows, meaning it is small and easily fits into memory (and a screen or A4 page).
- All of the numeric attributes are in the same units and the same scale, not requiring any special scaling or transforms to get started.

Let's get started with your hello world machine learning project in Python.

Machine Learning in Python: Step-By-Step

In this section, we are going to work through a small machine learning project end-to-end.

2. Load The Data

We are going to use the iris flowers dataset. This dataset is famous because it is used as the "hello world" dataset in machine learning and statistics by pretty much everyone.

The dataset contains 150 observations of iris flowers. There are four columns of measurements of the flowers in centimeters. The fifth column is the species of the flower observed. All observed flowers belong to one of three species.

You can learn more about this dataset on Wikipedia.
In this step we are going to load the iris data from CSV file URL.

2.1 Import libraries

First, let's import all of the modules, functions and objects we are going to use in this lesson.

```
import pandas
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
from sklearn import model_selection
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```

Everything should load without error. If you have an error, stop, check if you need to install some packages. You need a working SciPy environment before continuing.

2.2 Load Dataset

We can load the data directly from the UCI Machine Learning repository.

We are using pandas to load the data. We will also use pandas next to explore the data both with descriptive statistics and data visualization.

Note that we are specifying the names of each column when loading the data. This will help later when we explore the data.

```
dataset = pandas.read_csv("iris.csv")
```

The dataset should load without incident.

You can download the iris.csv file into your project

.

3. Summarize the Dataset

Now it is time to take a look at the data.

In this step we are going to take a look at the data a few different ways:

1. Dimensions of the dataset.
2. Peek at the data itself.
3. Statistical summary of all attributes.
4. Breakdown of the data by the class variable.

Don't worry, each look at the data is one command. These are useful commands that you can use again and again on future projects.

3.1 Dimensions of Dataset

We can get a quick idea of how many instances (rows) and how many attributes (columns) the data contains with the shape property.

```
print(dataset.shape)
```

You should see 150 instances and 5 attributes:

```
(150, 5)
```

3.2 Peek at the Data

It is also always a good idea to actually eyeball your data.

You should see the first 20 rows of the data:

```
print(dataset.head(20))
```

Should print

	sepal-length	sepal-width	petal-length	petal-width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
11	4.8	3.4	1.6	0.2	Iris-setosa
12	4.8	3.0	1.4	0.1	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
17	5.1	3.5	1.4	0.3	Iris-setosa
18	5.7	3.8	1.7	0.3	Iris-setosa
19	5.1	3.8	1.5	0.3	Iris-setosa

3.3 Statistical Summary

Now we can take a look at a summary of each attribute.

```
# descriptions
print(dataset.describe())
```

This includes the count, mean, the min and max values as well as some percentiles.

We can see that all of the numerical values have the same scale (centimeters) and similar ranges between 0 and 8 centimeters.

```
1       sepal-length  sepal-width  petal-length  petal-width
2 count    150.000000   150.000000   150.000000   150.000000
3 mean       5.843333     3.054000     3.758667     1.198667
4 std        0.828066     0.433594     1.764420     0.763161
5 min        4.300000     2.000000     1.000000     0.100000
6 25%        5.100000     2.800000     1.600000     0.300000
7 50%        5.800000     3.000000     4.350000     1.300000
8 75%        6.400000     3.300000     5.100000     1.800000
9 max        7.900000     4.400000     6.900000     2.500000
```

3.4 Class Distribution

Let's now take a look at the number of instances (rows) that belong to each class. We can view this as an absolute count.

```
# class distribution
print(dataset.groupby('class').size())
```

We can see that each class has the same number of instances (50 or 33% of the dataset).

```
1 class
2 Iris-setosa       50
3 Iris-versicolor    50
4 Iris-virginica     50
```

4. Data Visualization

We now have a basic idea about the data. We need to extend that with some visualizations.

We are going to look at two types of plots:

1. Univariate plots to better understand each attribute.
2. Multivariate plots to better understand the relationships between attributes.

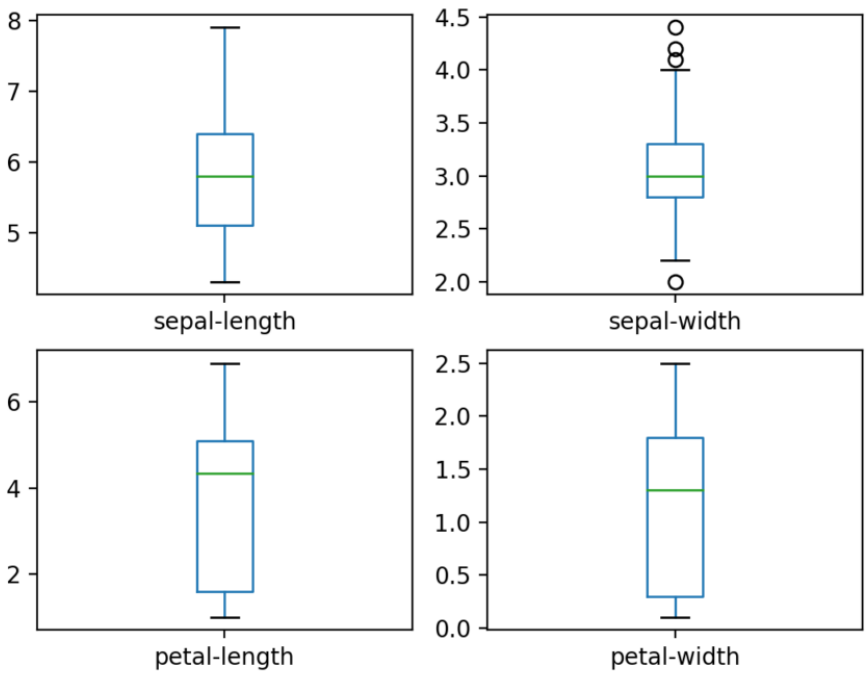
4.1 Univariate Plots

We start with some univariate plots, that is, plots of each individual variable.

Given that the input variables are numeric, we can create box and whisker plots of each.

```
# box and whisker plots
dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
plt.show()
```

This gives us a much clearer idea of the distribution of the input attributes:

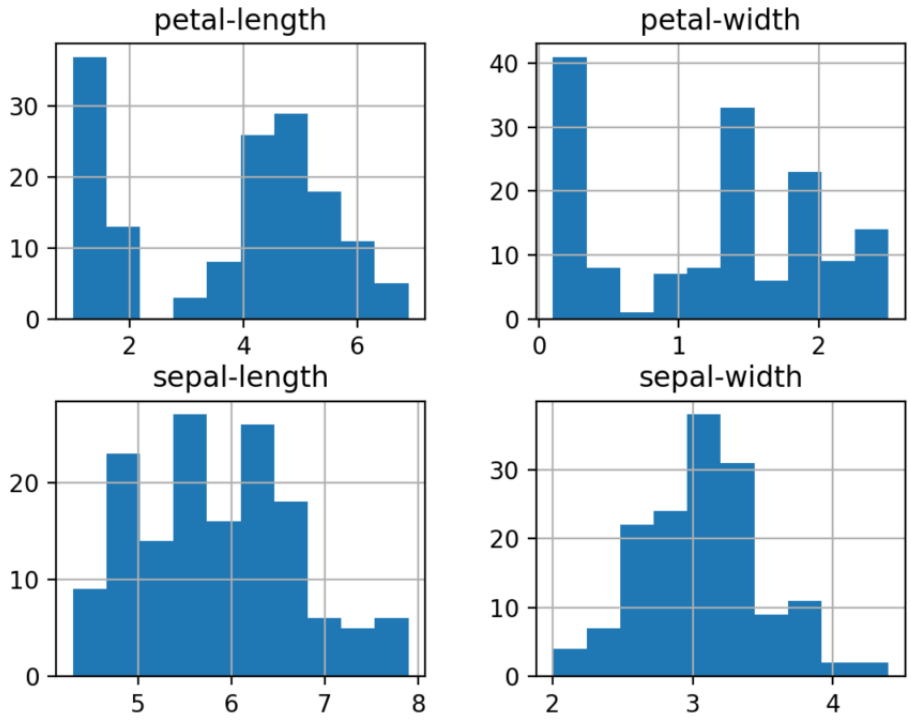


Box and Whisker Plots for Each Input Variable for the Iris Flowers Dataset

We can also create a histogram of each input variable to get an idea of the distribution.

```
# histograms
dataset.hist()
plt.show()
```

It looks like perhaps two of the input variables have a Gaussian distribution. This is useful to note as we can use algorithms that can exploit this assumption.



Histogram Plots for Each Input Variable for the Iris Flowers Dataset

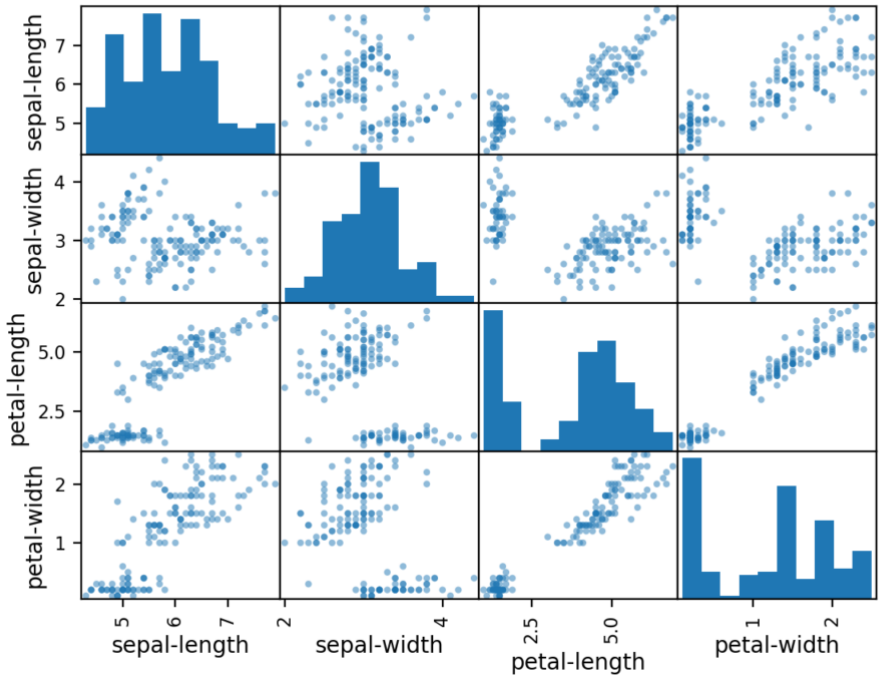
4.2 Multivariate Plots

Now we can look at the interactions between the variables.

First, let's look at scatterplots of all pairs of attributes. This can be helpful to spot structured relationships between input variables.

```
# scatter plot matrix
scatter_matrix(dataset)
plt.show()
```

Note the diagonal grouping of some pairs of attributes. This suggests a high correlation and a predictable relationship.



Scatter Matrix Plot for Each Input Variable for the Iris Flowers Dataset

5. Evaluate Some Algorithms

Now it is time to create some models of the data and estimate their accuracy on unseen data.

Here is what we are going to cover in this step:

1. Separate out a validation dataset.
2. Set-up the test harness to use 10-fold cross validation.
3. Build 5 different models to predict species from flower measurements
4. Select the best model.

5.1 Create a Validation Dataset

We need to know that the model we created is any good.

Later, we will use statistical methods to estimate the accuracy of the models that we create on unseen data. We also want a more concrete estimate of the accuracy of the best model on unseen data by evaluating it on actual unseen data.

That is, we are going to hold back some data that the algorithms will not get to see and we will use this data to get a second and independent idea of how accurate the best model might actually be.

We will split the loaded dataset into two, 80% of which we will use to train our models and 20% that we will hold back as a validation dataset.

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:4]
Y = array[:,4]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = model_selection.train_test_split(X, Y,
test_size=validation_size, random_state=seed)
```

You now have training data in the *X_train* and *Y_train* for preparing models and a *X_validation* and *Y_validation* sets that we can use later.
Notice that we used a python slice to select the columns in the NumPy array. If this is new to you, you might want to check-out this post:

5.2 Test Harness

We will use 10-fold cross validation to estimate accuracy.

This will split our dataset into 10 parts, train on 9 and test on 1 and repeat for all combinations of train-test splits.

```
# Test options and evaluation metric
seed = 7
scoring = 'accuracy'
```

The specific random seed does not matter, learn more about pseudorandom number generators here:

- Introduction to Random Number Generators for Machine Learning in Python
  We are using the metric of '*accuracy*' to evaluate models. This is a ratio of the number of correctly predicted instances in divided by the total number of instances in the dataset multiplied by 100 to give a percentage (e.g. 95% accurate). We will be using the *scoring* variable when we run build and evaluate each model next.

5.3 Build Models

We don't know which algorithms would be good on this problem or what configurations to use. We get an idea from the plots that some of the classes are partially linearly separable in some dimensions, so we are expecting generally good results.

Let's evaluate 6 different algorithms:

- Logistic Regression (LR)
- Linear Discriminant Analysis (LDA)
- K-Nearest Neighbors (KNN).
- Classification and Regression Trees (CART).
- Gaussian Naive Bayes (NB).
- Support Vector Machines (SVM).

This is a good mixture of simple linear (LR and LDA), nonlinear (KNN, CART, NB and SVM) algorithms. We reset the random number seed before each run to ensure that the evaluation of each algorithm is performed using exactly the same data splits. It ensures the results are directly comparable.

Let's build and evaluate our models:

```
# Spot Check Algorithms
models = []
models.append(('LR', LogisticRegression(solver='liblinear', multi_class='ovr')))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto')))
# evaluate each model in turn
for name, model in models:
  kfold = model_selection.KFold(n_splits=10, random_state=seed)
  cv_results = model_selection.cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
  print(name, cv_results.mean())
```

5.4 Select Best Model

We now have 6 models and accuracy estimations for each. We need to compare the models to each other and select the most accurate.

Running the example above, we get the following raw results:

```
1 LR: 0.966667 (0.040825)
2 LDA: 0.975000 (0.038188)
3 KNN: 0.983333 (0.033333)
4 CART: 0.975000 (0.038188)
5 NB: 0.975000 (0.053359)
6 SVM: 0.991667 (0.025000)
```

6. Make Predictions

The KNN algorithm is very simple and was an accurate model based on our tests. Now we want to get an idea of the accuracy of the model on our validation set.

This will give us an independent final check on the accuracy of the best model. It is valuable to keep a validation set just in case you made a slip during training, such as overfitting to the training set or a data leak. Both will result in an overly optimistic result.

We can run the KNN model directly on the validation set and summarize the results as a final accuracy score, a confusion matrix and a classification report.

```
# Make predictions on validation dataset
knn = KNeighborsClassifier()
knn.fit(X_train, Y_train)
predictions = knn.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

We can see that the accuracy is 0.9 or 90%. The confusion matrix provides an indication of the three errors made. Finally, the classification report provides a breakdown of each class by precision, recall, f1-score and support showing excellent results (granted the validation dataset was small).

```
1   0.9
2  [[ 7  0  0]
3   [ 0 11  1]
4   [ 0  2  9]]
5               precision    recall  f1-score   support
```

```
 6
 7      Iris-setosa       1.00      1.00      1.00         7
 8  Iris-versicolor       0.85      0.92      0.88        12
 9   Iris-virginica       0.90      0.82      0.86        11
10
11       micro avg       0.90      0.90      0.90        30
12       macro avg       0.92      0.91      0.91        30
13    weighted avg       0.90      0.90      0.90        30
```

Now we can use our model to predict flowers

```
predicted_flower = knn.predict([[1.3,1.9,1.5,5]])
print("Predicted: ", predicted_flower)
```

will print

['iris-vesicolor']

Done

Check out this code @ https://justpaste.it/7dsep

https://towardsdatascience.com/how-to-build-a-gender-classifier-in-python-using-scikit-learn-13c7bb502f2e