**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.



|  |  |
| --- | --- |
| 1  2  3  4  5 | ...  # Load dataset  url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"  names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']  dataset = read\_csv(url, names=names) |

The dataset should load without incident.

If you do have network problems, you can download the [iris.csv](https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv) file into your working directory and load it using the same method, changing URL to the local file name.

**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.



|  |  |
| --- | --- |
| 1  2  3 | ...  # shape  print(dataset.shape) |

You should see 150 instances and 5 attributes:



|  |  |
| --- | --- |
| 1 | (150, 5) |

**3.2 Peek at the Data**

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



|  |  |
| --- | --- |
| 1  2  3 | ...  # head  print(dataset.head(20)) |

You should see the first 20 rows of the data:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | 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.

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



|  |  |
| --- | --- |
| 1  2  3 | ...  # descriptions  print(dataset.describe()) |

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



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



|  |  |
| --- | --- |
| 1  2  3 | ...  # 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  2  3  4 | class  Iris-setosa        50  Iris-versicolor    50  Iris-virginica     50 |

**3.5 Complete Example**

For reference, we can tie all of the previous elements together into a single script.

The complete example is listed below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # summarize the data  from pandas import read\_csv  # Load dataset  url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"  names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']  dataset = read\_csv(url, names=names)  # shape  print(dataset.shape)  # head  print(dataset.head(20))  # descriptions  print(dataset.describe())  # class distribution  print(dataset.groupby('class').size()) |

**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**

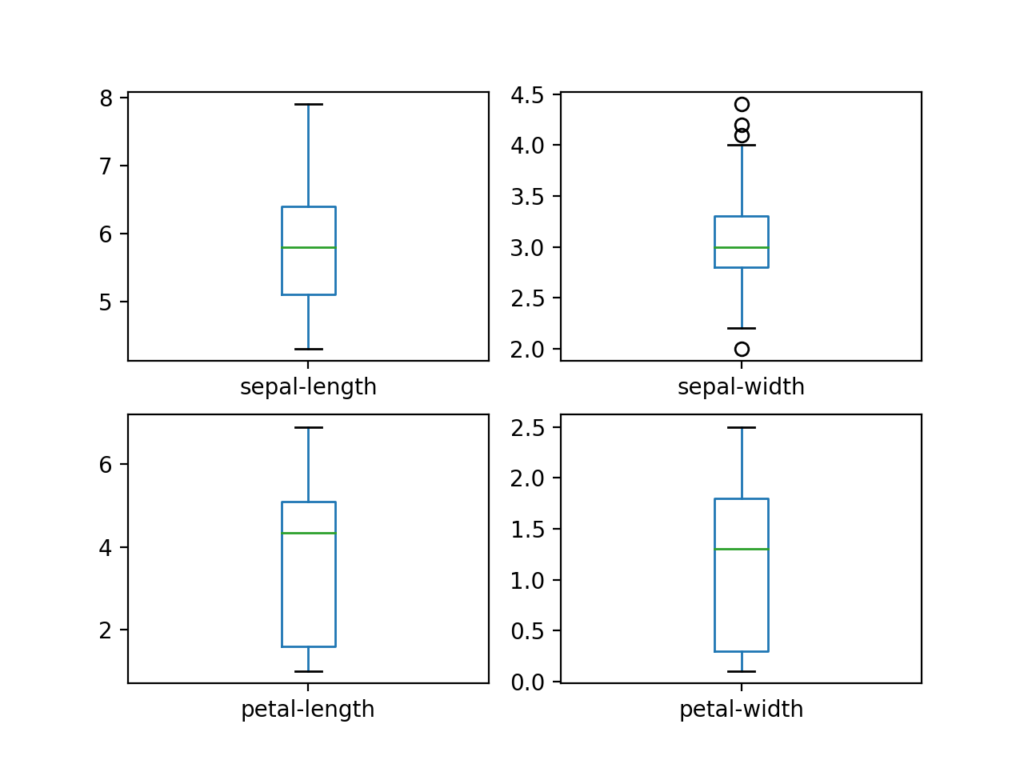
We start with some univariate plots, that is, plots of each individual variable.

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



|  |  |
| --- | --- |
| 1  2  3  4 | ...  # box and whisker plots  dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)  pyplot.show() |

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



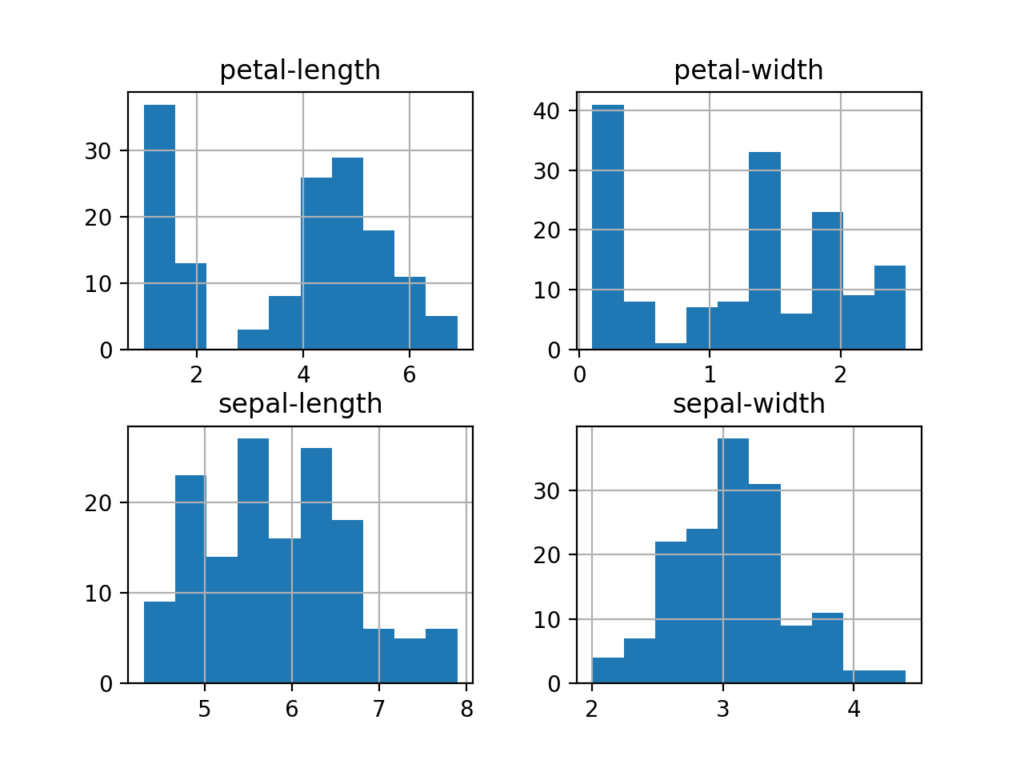
Box and Whisker Plots for Each Input Variable for the Iris Flowers Dataset

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



|  |  |
| --- | --- |
| 1  2  3  4 | ...  # histograms  dataset.hist()  pyplot.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**

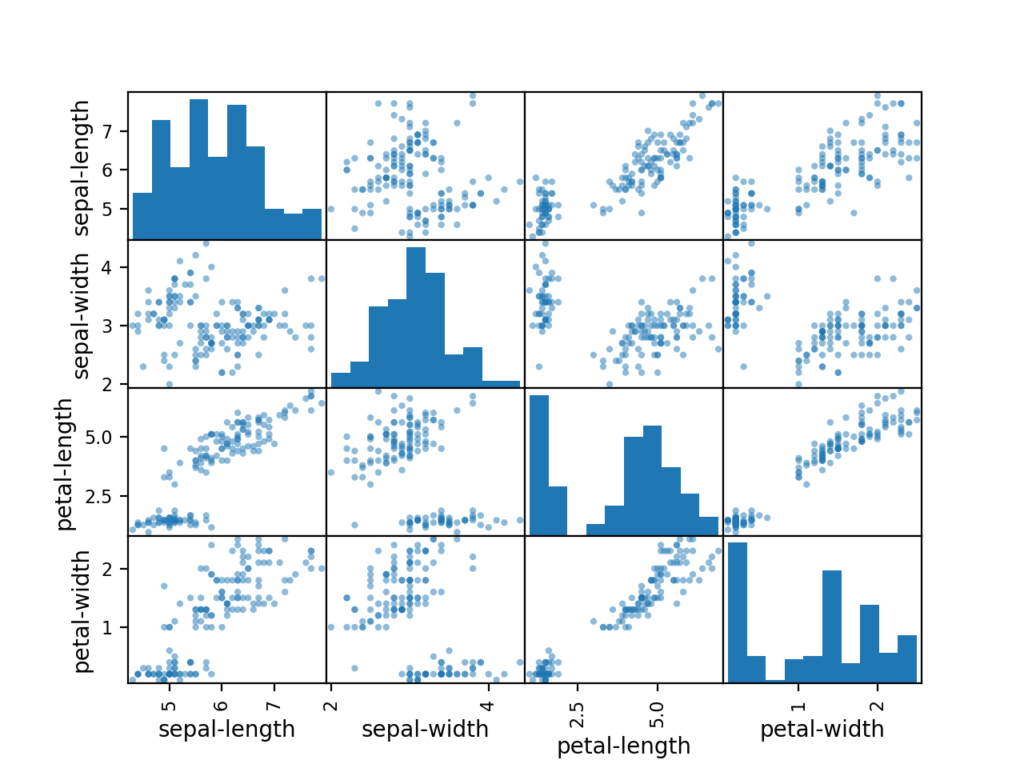
Now we can look at the interactions between the variables.

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



|  |  |
| --- | --- |
| 1  2  3  4 | ...  # scatter plot matrix  scatter\_matrix(dataset)  pyplot.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

**4.3 Complete Example**

For reference, we can tie all of the previous elements together into a single script.

The complete example is listed below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # visualize the data  from pandas import read\_csv  from pandas.plotting import scatter\_matrix  from matplotlib import pyplot  # Load dataset  url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"  names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']  dataset = read\_csv(url, names=names)  # box and whisker plots  dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)  pyplot.show()  # histograms  dataset.hist()  pyplot.show()  # scatter plot matrix  scatter\_matrix(dataset)  pyplot.show() |

**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 multiple 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 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, evaluate and select among our models, and 20% that we will hold back as a validation dataset.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ...  # Split-out validation dataset  array = dataset.values  X = array[:,0:4]  y = array[:,4]  X\_train, X\_validation, Y\_train, Y\_validation = train\_test\_split(X, y, test\_size=0.20, random\_state=1) |

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:

* [How to Index, Slice and Reshape NumPy Arrays for Machine Learning in Python](https://machinelearningmastery.com/index-slice-reshape-numpy-arrays-machine-learning-python/)

**5.2 Test Harness**

We will use stratified 10-fold cross validation to estimate model 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.

Stratified means that each fold or split of the dataset will aim to have the same distribution of example by class as exist in the whole training dataset.

For more on the k-fold cross-validation technique, see the tutorial:

* [A Gentle Introduction to k-fold Cross-Validation](https://machinelearningmastery.com/k-fold-cross-validation/)

We set the random seed via the *random\_state* argument to a fixed number to ensure that each algorithm is evaluated on the same splits of the training dataset.

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

* [Introduction to Random Number Generators for Machine Learning in Python](https://machinelearningmastery.com/introduction-to-random-number-generators-for-machine-learning/)

We are using the metric of ‘*accuracy*‘ to evaluate models.

This is a ratio of the number of correctly predicted instances 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 test 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.

Let’s build and evaluate our models:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | ...  # 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  results = []  names = []  for name, model in models:  kfold = StratifiedKFold(n\_splits=10, random\_state=1, shuffle=True)  cv\_results = cross\_val\_score(model, X\_train, Y\_train, cv=kfold, scoring='accuracy')  results.append(cv\_results)  names.append(name)  print('%s: %f (%f)' % (name, cv\_results.mean(), cv\_results.std())) |

**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  2  3  4  5  6 | LR: 0.960897 (0.052113)  LDA: 0.973974 (0.040110)  KNN: 0.957191 (0.043263)  CART: 0.957191 (0.043263)  NB: 0.948858 (0.056322)  SVM: 0.983974 (0.032083) |

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

**What scores did you get?**  
Post your results in the comments below.

In this case, we can see that it looks like Support Vector Machines (SVM) has the largest estimated accuracy score at about 0.98 or 98%.

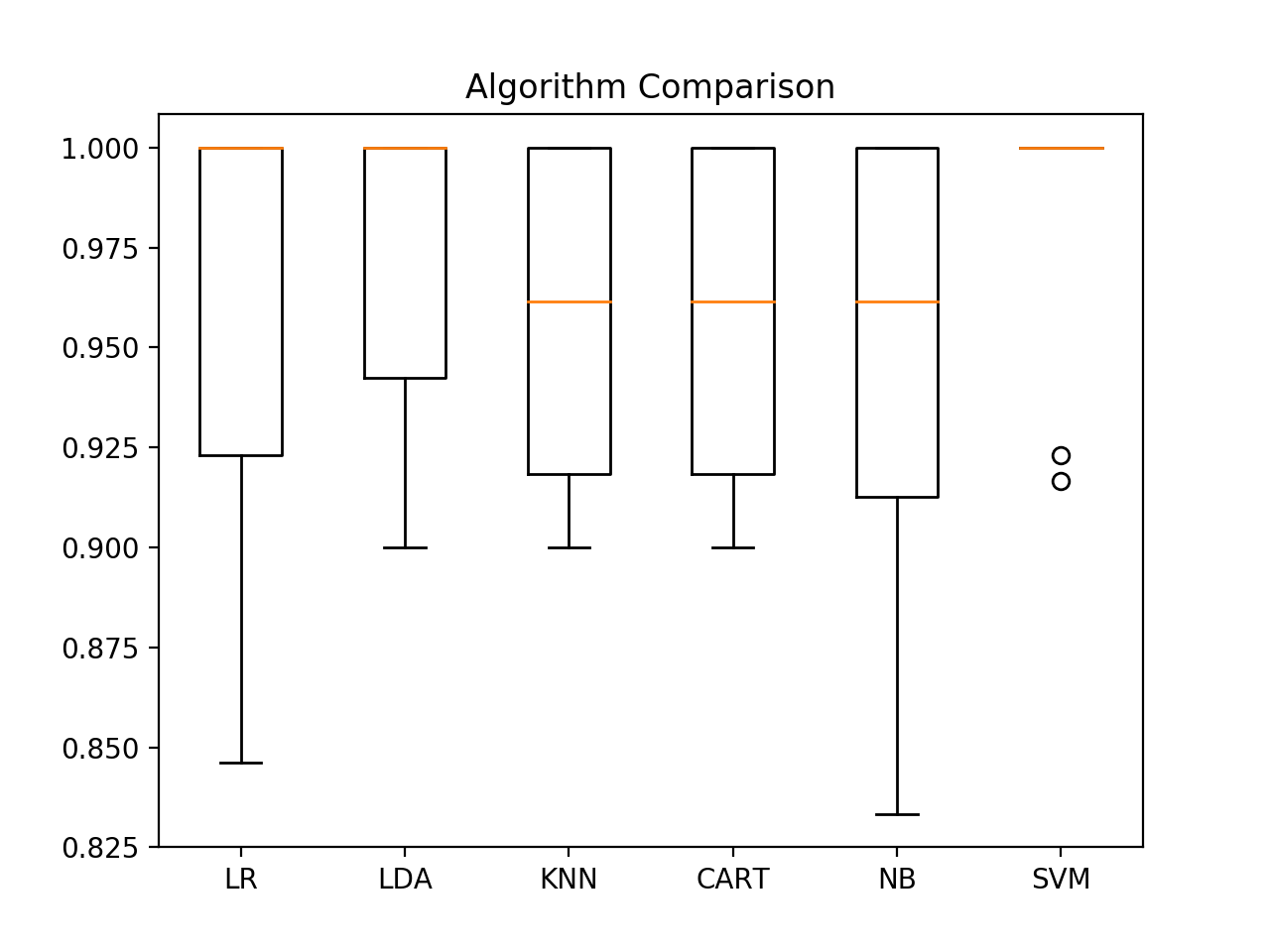
We can also create a plot of the model evaluation results and compare the spread and the mean accuracy of each model. There is a population of accuracy measures for each algorithm because each algorithm was evaluated 10 times (via 10 fold-cross validation).

A useful way to compare the samples of results for each algorithm is to create a box and whisker plot for each distribution and compare the distributions.



|  |  |
| --- | --- |
| 1  2  3  4  5 | ...  # Compare Algorithms  pyplot.boxplot(results, labels=names)  pyplot.title('Algorithm Comparison')  pyplot.show() |

We can see that the box and whisker plots are squashed at the top of the range, with many evaluations achieving 100% accuracy, and some pushing down into the high 80% accuracies.



Box and Whisker Plot Comparing Machine Learning Algorithms on the Iris Flowers Dataset