Tikrit University
Computer Science Dept.

Master Degree
Lecture -4-

Associate Professor

Dr. Eng. Zaidoon.T.AL-Qaysi

College of Computer Science and Mathematics

(2023-2024)

Real-world data often has missing values. Data can have missing values for many reasons, often specific to the problem domain, and might include reasons such as corrupt measurements or data unavailability. They may occur for a number of reasons, such as malfunctioning measurement equipment, changes in experimental design during data collection, and collation of several similar but not identical datasets. Handling missing data is important as many machine learning algorithms do not support data with missing values. Most data has missing values, and the likelihood of having missing values increases with the size of the dataset.

```
# load and summarize the dataset
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# summarize the dataset
print(dataset.describe())
```

```
                0           1           2  ...           6           7           8
count  768.000000  768.000000  768.000000  ...  768.000000  768.000000  768.000000
mean     3.845052  120.894531   69.105469  ...    0.471876   33.240885    0.348958
std      3.369578   31.972618   19.355807  ...    0.331329   11.760232    0.476951
min      0.000000    0.000000    0.000000  ...    0.078000   21.000000    0.000000
25%      1.000000   99.000000   62.000000  ...    0.243750   24.000000    0.000000
50%      3.000000  117.000000   72.000000  ...    0.372500   29.000000    0.000000
75%      6.000000  140.250000   80.000000  ...    0.626250   41.000000    1.000000
max     17.000000  199.000000  122.000000  ...    2.420000   81.000000    1.000000
```

Example 1: Calculating summary statistics for each variable.

Specifically, the following columns have an invalid zero minimum value: ˆ

1: Plasma glucose concentration ˆ

2: Diastolic blood pressure ˆ

3: Triceps skinfold thickness ˆ

4: 2-Hour serum insulin ˆ

5: Body mass index

```
# example of summarizing the number of missing values for each variable
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# count the number of missing values for each column
num_missing = (dataset[[1,2,3,4,5]] == 0).sum()
# report the results
print(num_missing)
```

```
1      5
2     35
3    227
4    374
5     11
```

Example 2: Example output from reporting the number of missing values in each column.

This is useful. We can see that there are columns that have a minimum value of zero (0). On some columns, a value of zero does not make sense and indicates an invalid or missing value. We can get a count of the number of missing values on each of these columns. We can do this by marking all of the values in the subset of the DataFrame we are interested in that have zero values as True. We can then count the number of true values in each column.

We can see that columns 1, 2 and 5 have just a few zero values, whereas columns 3 and 4 show a lot more, nearly half of the rows. This highlights that different missing value strategies may be needed for different columns, e.g. to ensure that there are still a sufficient number of records left to train a predictive model.

```
# example of marking missing values with nan values
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# count the number of nan values in each column
print(dataset.isnull().sum())
```

```
0      0
1      5
2     35
3    227
4    374
5     11
6      0
7      0
8      0
dtype: int64
```

Example 3: Marking missing values in the dataset.

Running the example, we can clearly see NaN values in the columns 2, 3, 4 and 5. There are only 5 missing values in column 1, so it is not surprising we did not see an example in the first 20 rows. It is clear from the raw data that marking the missing values had the intended effect.

```
# example of review data with missing values marked with a nan
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# summarize the first 20 rows of data
print(dataset.head(20))
```

```
    0      1     2     3      4     5      6   7  8
0   6  148.0  72.0  35.0    NaN  33.6  0.627  50  1
1   1   85.0  66.0  29.0    NaN  26.6  0.351  31  0
2   8  183.0  64.0   NaN    NaN  23.3  0.672  32  1
3   1   89.0  66.0  23.0   94.0  28.1  0.167  21  0
4   0  137.0  40.0  35.0  168.0  43.1  2.288  33  1
5   5  116.0  74.0   NaN    NaN  25.6  0.201  30  0
6   3   78.0  50.0  32.0   88.0  31.0  0.248  26  1
7  10  115.0   NaN   NaN    NaN  35.3  0.134  29  0
8   2  197.0  70.0  45.0  543.0  30.5  0.158  53  1
9   8  125.0  96.0   NaN    NaN   NaN  0.232  54  1
10  4  110.0  92.0   NaN    NaN  37.6  0.191  30  0
11 10  168.0  74.0   NaN    NaN  38.0  0.537  34  1
12 10  139.0  80.0   NaN    NaN  27.1  1.441  57  0
13  1  189.0  60.0  23.0  846.0  30.1  0.398  59  1
14  5  166.0  72.0  19.0  175.0  25.8  0.587  51  1
15  7  100.0   NaN   NaN    NaN  30.0  0.484  32  1
16  0  118.0  84.0  47.0  230.0  45.8  0.551  31  1
17  7  107.0  74.0   NaN    NaN  29.6  0.254  31  1
18  1  103.0  30.0  38.0   83.0  43.3  0.183  33  0
19  1  115.0  70.0  30.0   96.0  34.6  0.529  32  1
```

Example 4: Summarizing the first few rows of the dataset.

```
# example where missing values cause errors
from numpy import nan
from pandas import read_csv
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# split dataset into inputs and outputs
values = dataset.values
X = values[:,0:8]
y = values[:,8]
# define the model
model = LinearDiscriminantAnalysis()
# define the model evaluation procedure
cv = KFold(n_splits=3, shuffle=True, random_state=1)
# evaluate the model
result = cross_val_score(model, X, y, cv=cv, scoring='accuracy')
# report the mean performance
print('Accuracy: %.3f' % result.mean())
```

```
ValueError: Input contains NaN, infinity or a value too large for dtype('float64').
```

Example 5: an error caused by the presence of missing values.

In the **Example 5**, we will try to evaluate the Linear Discriminant Analysis (LDA) algorithm on the dataset with missing values. This is an algorithm that does not work when there are missing values in the dataset.

```
# example of removing rows that contain missing values
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# summarize the shape of the raw data
print(dataset.shape)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# drop rows with missing values
dataset.dropna(inplace=True)
# summarize the shape of the data with missing rows removed
print(dataset.shape)
```

```
(768, 9)
(392, 9)
```

Example 5: Output from removing rows that contain missing values

```
# evaluate model on data after rows with missing data are removed
from numpy import nan
from pandas import read_csv
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# drop rows with missing values
dataset.dropna(inplace=True)
# split dataset into inputs and outputs
values = dataset.values
X = values[:,0:8]
y = values[:,8]
# define the model
model = LinearDiscriminantAnalysis()
# define the model evaluation procedure
cv = KFold(n_splits=3, shuffle=True, random_state=1)
# evaluate the model
result = cross_val_score(model, X, y, cv=cv, scoring='accuracy')
# report the mean performance
print('Accuracy: %.3f' % result.mean())
```

```
Accuracy: 0.781
```

Example 6: Evaluating a model after rows with missing values are removed

- Datasets may have missing values, and this can cause problems for many machine learning algorithms. As such, it is good practice to identify and replace missing values for each column in your input data prior to modeling your prediction task. This is called missing data imputation, or imputing for short. A popular approach for data imputation is to calculate a statistical value for each column (such as a mean) and replace all missing values for that column with the statistic. It is a popular approach because the statistic is easy to calculate using the training dataset and because it often results in good performance.

- Most machine learning algorithms require numeric input values, and a value to be present for each row and column in a dataset. As such, missing values can cause problems for machine learning algorithms. Because of this, it is common to identify missing values in a dataset and replace them with a numeric value. This is called data imputing, or missing data imputation. A simple and popular approach to data imputation involves using statistical methods to estimate a value for a column from those values that are present, then replace all missing values in the column with the calculated statistic. It is simple because statistics are fast to calculate and it is popular because it often proves very effective. Common statistics calculated include: ˆ

1. The column mean value. ˆ
2. The column median value. ˆ
3. The column mode value. ˆ
4. A constant value.

- The horse colic dataset describes medical characteristics of horses with colic and whether they lived or died. There are 300 rows and 26 input variables with one output variable. It is a binary classification prediction task that involves predicting 1 if the horse lived and 2 if the horse died.

```
# summarize the horse colic dataset
from pandas import read_csv
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# summarize the first few rows
print(dataframe.head())
# summarize the number of rows with missing values for each column
for i in range(dataframe.shape[1]):
  # count number of rows with missing values
  n_miss = dataframe[[i]].isnull().sum()
  perc = n_miss / dataframe.shape[0] * 100
  print('> %d, Missing: %d (%.1f%%)' % (i, n_miss, perc))
```

```
    0   1        2     3      4     5    6   ...   21   22  23     24  25  26  27
0  2.0   1   530101  38.5   66.0  28.0  3.0  ...  NaN  2.0   2  11300   0   0   2
1  1.0   1   534817  39.2   88.0  20.0  NaN  ...  2.0  3.0   2   2208   0   0   2
2  2.0   1   530334  38.3   40.0  24.0  1.0  ...  NaN  1.0   2      0   0   0   1
3  1.0   9  5290409  39.1  164.0  84.0  4.0  ...  5.3  2.0   1   2208   0   0   1
4  2.0   1   530255  37.3  104.0  35.0  NaN  ...  NaN  2.0   2   4300   0   0   2
```

Example 7: Summarizing the number of missing values for each column

- The scikit-learn machine learning library provides the SimpleImputer class that supports statistical imputation. The SimpleImputer is a data transform that is first configured based on the type of statistic to calculate for each column, e.g. mean.

- We can demonstrate its usage on the horse colic dataset and confirm it works by summarizing the total number of missing values in the dataset before and after the transform. The complete example is listed below.

```
# statistical imputation transform for the horse colic dataset
from numpy import isnan
from pandas import read_csv
from sklearn.impute import SimpleImputer
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# summarize total missing
print('Missing: %d' % sum(isnan(X).flatten()))
# define imputer
imputer = SimpleImputer(strategy='mean')
# fit on the dataset
imputer.fit(X)
# transform the dataset
Xtrans = imputer.transform(X)
# summarize total missing
print('Missing: %d' % sum(isnan(Xtrans).flatten()))
```

```
Missing: 1605
Missing: 0
```

Example 8: Imputing missing values in the dataset

- We can evaluate the mean-imputed dataset and random forest modeling pipeline for the horse colic dataset with repeated 10-fold cross-validation. The complete example is listed below.

```
# evaluate mean imputation and random forest for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# define modeling pipeline
model = RandomForestClassifier()
imputer = SimpleImputer(strategy='mean')
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
# define model evaluation
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Mean Accuracy: 0.866 (0.061)
```

Example 9: Evaluating a model on a dataset with statistical imputation.

```
# compare statistical imputation strategies for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# evaluate each strategy on the dataset
results = list()
strategies = ['mean', 'median', 'most_frequent', 'constant']
for s in strategies:
  # create the modeling pipeline
  pipeline = Pipeline(steps=[('i', SimpleImputer(strategy=s)), ('m',
      RandomForestClassifier())])
  # evaluate the model
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store results
  results.append(scores)
  print('>%s %.3f (%.3f)' % (s, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=strategies, showmeans=True)
pyplot.show()
```

```
>mean 0.867 (0.056)
>median 0.868 (0.050)
>most_frequent 0.867 (0.060)
>constant 0.878 (0.046)
```

Example 10: Comparing model performance with different statistical imputation strategies.

- In **Figure 1** we can see that a box and whisker plot is created for each set of results, allowing the distribution of results to be compared. We can see that the distribution of accuracy scores for the constant strategy may be better than the other strategies.

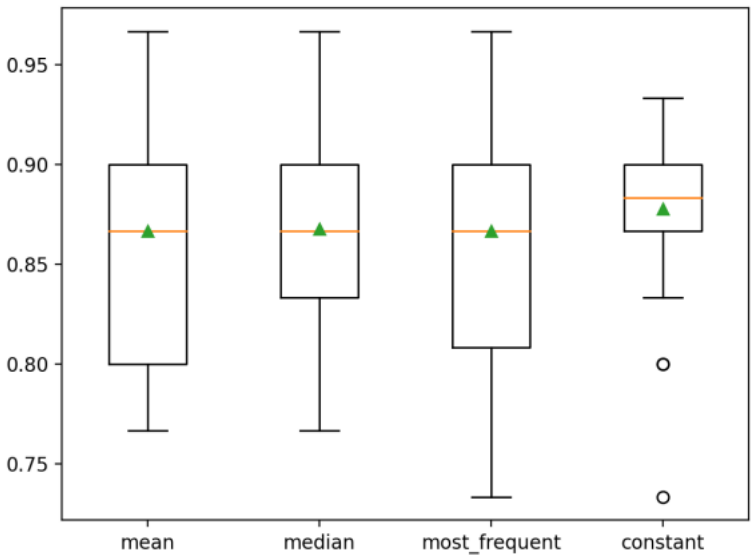

Figure 1: Box and Whisker Plot of Statistical Imputation Strategies

- A popular approach to missing data imputation is to use a model to predict the missing values. This requires a model to be created for each input variable that has missing values. Although any one among a range of different models can be used to predict the missing values, the k-nearest neighbor (KNN) algorithm has proven to be generally effective, often referred to as nearest neighbor imputation. The KNNImputer is a data transform that is first configured based on the method used to estimate the missing values. The default distance measure is a Euclidean distance measure that is NaN aware, e.g. will not include NaN values when calculating the distance between members of the training dataset. This is set via the metric argument. The number of neighbors is set to five by default and can be configured by the n neighbors argument. Finally, the distance measure can be weighed proportional to the distance between instances (rows), although this is set to a uniform weighting by default, controlled via the weights argument.

```
# knn imputation transform for the horse colic dataset
from numpy import isnan
from pandas import read_csv
from sklearn.impute import KNNImputer
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# summarize total missing
print('Missing: %d' % sum(isnan(X).flatten()))
# define imputer
imputer = KNNImputer()
# fit on the dataset
imputer.fit(X)
# transform the dataset
Xtrans = imputer.transform(X)
# summarize total missing
print('Missing: %d' % sum(isnan(Xtrans).flatten()))
```

```
Missing: 1605
Missing: 0
```

Example 11: Using the KNNImputer to impute missing values.

```
# evaluate knn imputation and random forest for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# define modeling pipeline
model = RandomForestClassifier()
imputer = KNNImputer()
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
# define model evaluation
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Example 11: Evaluating a model on a dataset transformed with the KNNImputer.

```
# compare knn imputation strategies for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# evaluate each strategy on the dataset
results = list()
strategies = [str(i) for i in [1,3,5,7,9,15,18,21]]
for s in strategies:
  # create the modeling pipeline
  pipeline = Pipeline(steps=[('i', KNNImputer(n_neighbors=int(s))), ('m',
      RandomForestClassifier())])
  # evaluate the model
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store results
  results.append(scores)
  print('>%s %.3f (%.3f)' % (s, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=strategies, showmeans=True)
pyplot.show()
```

```
>1 0.861 (0.055)
>3 0.860 (0.058)
>5 0.869 (0.051)
>7 0.864 (0.056)
>9 0.866 (0.052)
>15 0.869 (0.058)
>18 0.861 (0.055)
>21 0.857 (0.056)
```

Example 12: Comparing the number of neighbors used in the KNNImputer

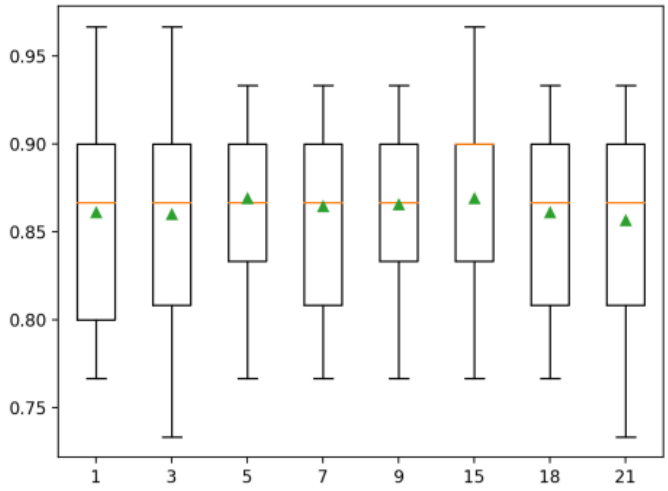

Figure 2: Box and Whisker Plot of Imputation Number of Neighbors

- A sophisticated approach involves defining a model to predict each missing feature as a function of all other features and to repeat this process of estimating feature values multiple times. The repetition allows the refined estimated values for other features to be used as input in subsequent iterations of predicting missing values. This is generally referred to as iterative imputation. Iterative imputation refers to a process where each feature is modeled as a function of the other features, e.g. a regression problem where missing values are predicted. Each feature is imputed sequentially, one after the other, allowing prior imputed values to be used as part of a model in predicting subsequent features. It is iterative because this process is repeated multiple times, allowing ever improved estimates of missing values to be calculated as missing values across all features are estimated. This approach may be generally referred to as fully conditional specification (FCS) or multivariate imputation by chained equations (MICE).

```
# iterative imputation transform for the horse colic dataset
from numpy import isnan
from pandas import read_csv
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# summarize total missing
print('Missing: %d' % sum(isnan(X).flatten()))
# define imputer
imputer = IterativeImputer()
# fit on the dataset
imputer.fit(X)
# transform the dataset
Xtrans = imputer.transform(X)
# summarize total missing
print('Missing: %d' % sum(isnan(Xtrans).flatten()))
```

```
Missing: 1605
Missing: 0
```

Example 13: using the IterativeImputer to impute missing values.

```
# evaluate iterative imputation and random forest for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# define modeling pipeline
model = RandomForestClassifier()
imputer = IterativeImputer()
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
# define model evaluation
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Mean Accuracy: 0.870 (0.049)
```

Example 14: Evaluating a model on a dataset transformed with the IterativeImputer.

- By default, imputation is performed in ascending order from the feature with the least missing values to the feature with the most. This makes sense as we want to have more complete data when it comes time to estimating missing values for columns where the majority of values are missing. Nevertheless, we can experiment with different imputation order strategies, such as descending, right-to-left (Arabic), left-to-right (Roman), and random. The example below evaluates and compares each available imputation order configuration.

```
# compare iterative imputation strategies for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# evaluate each strategy on the dataset
results = list()
strategies = ['ascending', 'descending', 'roman', 'arabic', 'random']
for s in strategies:
  # create the modeling pipeline
  pipeline = Pipeline(steps=[('i', IterativeImputer(imputation_order=s)), ('m',
      RandomForestClassifier())])
  # evaluate the model
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store results
  results.append(scores)
  print('>%s %.3f (%.3f)' % (s, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=strategies, showmeans=True)
pyplot.show()
```

```
>ascending 0.871 (0.048)
>descending 0.868 (0.050)
>roman 0.880 (0.056)
>arabic 0.872 (0.058)
>random 0.868 (0.051)
```

Example 15: Comparing model performance with different data order in the IterativeImputer.

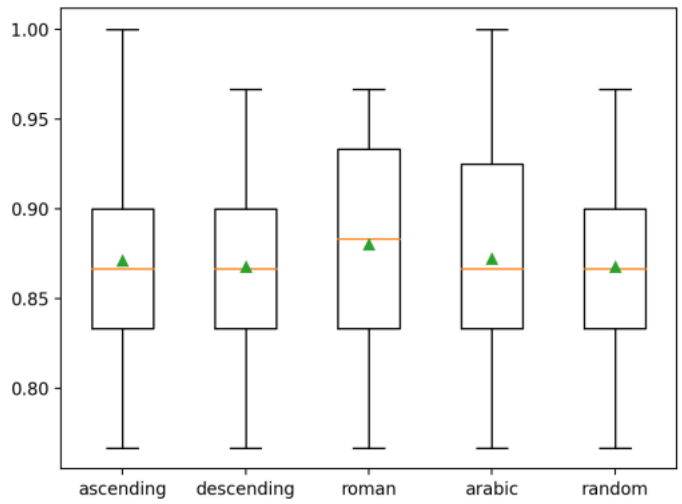

Figure 3: Box and Whisker Plot of Imputation Order Strategies