Avoiding Data Leakage

Emmanuel Amador Maldonado

The main objective is to see the difference in the model performance between using data preparation in the whole dataset before split it into test and train sets, and using data preparation after splitting the dataset into train and test sets.

In [ ]:
```
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split, RepeatedStratifiedKFold, cros
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline
```

Evaluating the model using Train-Test Evaluation

In this section, we'll evaluate a logistic regression model using train and test sets on a synthetic binary classification dataset where the inpyt variables have been normalized.

Using the **nake_classification** function in *sklearn.datasets*, we'll create a dataset with a binary target variable

- n_samples: Total number of rows
- n_features: Number of independent variables (n_features = n_informative + n_redundant + n_repeated)
  - n_informative: Number of features that it gives real information related to the target variable
  - n_redundant: Number of features that doesn't give real information related to the target variable
  - n_repreated: Number of features repeated
- random_state: Number of the seed. To make the results replicables
- n_classes: Number of labels of the classification problem (Number of labels for the target variable) -> default: 2

Returns X, y -> independent variables, and the target variable, respectively

In [ ]:
```
X, y = make_classification(n_samples = 1000, n_features = 20, n_informative = 15, n
```

In [ ]:
```
print(X.shape, y.shape)
```

```
(1000, 20) (1000,)
```

In [ ]:
```
pd.DataFrame(X).head(15)
```

Out[ ]:

	0	1	2	3	4	5	6	7
0	0.292995	-4.212231	-1.288332	-2.178498	-0.645277	2.580977	0.284224	-7.182793
1	-0.068399	5.518841	11.238977	-5.039700	-2.086784	2.149685	0.559734	15.113777
2	0.731616	-0.684686	-0.981742	-2.552465	-5.270308	-1.561498	-1.169269	-2.104087
3	2.309107	-0.320548	-6.591664	1.070525	-4.418769	1.134274	2.340813	-5.983425
4	-0.488406	-3.213065	1.100805	-1.356223	5.325086	0.729179	-0.257040	-1.035284
5	-0.156687	-2.491359	-0.319048	-0.180767	-5.161745	0.536021	-1.435684	0.708005
6	3.095598	5.155741	4.846930	1.677064	-5.461116	2.922476	-4.679053	2.916699
7	1.482795	1.159066	0.805299	-3.453292	-9.464460	-3.631272	-0.010151	3.080085
8	-4.548827	1.388299	1.854430	1.201001	-5.795249	-0.654437	0.537701	1.920046
9	-0.641345	-1.285319	0.959277	-1.510647	7.076031	-1.491739	-0.409155	-3.735169
10	2.517704	-0.816360	-9.601974	3.063419	-0.206767	3.427853	4.472281	-11.700568
11	1.578265	6.335031	13.688271	-1.759311	5.184626	3.591914	-0.080637	-0.336176
12	-2.145484	-1.672974	-0.183422	-2.307678	11.477898	-1.239618	0.852235	-6.647052
13	3.641875	3.194960	-10.560793	2.471885	-1.691838	-1.197527	2.901170	-2.877372
14	0.461111	-3.719436	-8.380107	-2.745835	-8.501479	-1.603518	0.815392	-5.369283

In [ ]:
```
pd.DataFrame(y).head(15)
```

```
Out[ ]:      0
        0    1
        1    1
        2    1
        3    0
        4    0
        5    0
        6    1
        7    1
        8    0
        9    1
       10    0
       11    1
       12    1
       13    0
       14    1
```

Train-test evaluation using data preparation to the whole dataset and then split it into train and test datasets

1. Data Preparation
2. Split dataset into train and test sets
3. Model evaluation

We can normalize the independent variables using the MinMaxScaler function from *sklearn.preprocessing*

- feature_range: (min_val, max_val) The range of values each feature will have after the transformation
- copy: True/False. If True, performs inplace row normalization

The transformation is given by:

- X_std = (X - X.min(axis=0))/(X.max(axis=0) - X.min(axis = 0))
- X_scaled = X_std*(max - min) + min

Where min, max = feature_range

In [ ]:

```
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

pd.DataFrame(X).head(15)
```

Out[ ]:

	0	1	2	3	4	5	6	7	
0	0.478319	0.186936	0.424031	0.429320	0.585333	0.674249	0.529972	0.314806	0.39374
1	0.451084	0.734584	0.771313	0.240540	0.546150	0.642636	0.548818	0.891427	0.31969
2	0.511375	0.385460	0.432530	0.404646	0.459618	0.370612	0.430548	0.446149	0.44360
3	0.630259	0.405953	0.277012	0.643687	0.482764	0.568208	0.670650	0.345824	0.55899
4	0.419431	0.243167	0.490262	0.483573	0.747615	0.538515	0.492948	0.473789	0.54635
5	0.444430	0.283784	0.450901	0.561128	0.462569	0.524357	0.412324	0.518873	0.35746
6	0.689532	0.714149	0.594112	0.683706	0.454431	0.699281	0.190466	0.575993	0.41997
7	0.567986	0.489223	0.482070	0.345210	0.345615	0.218900	0.509836	0.580219	0.61934
8	0.113426	0.502124	0.511154	0.652296	0.445349	0.437098	0.547311	0.550218	0.48070
9	0.407905	0.351658	0.486339	0.473384	0.795209	0.375725	0.482542	0.403967	0.52307
10	0.645980	0.378050	0.193560	0.775176	0.597252	0.736324	0.816450	0.197970	0.53843
11	0.575181	0.780518	0.839212	0.456977	0.743798	0.748350	0.505014	0.491869	0.62512
12	0.294549	0.329841	0.454661	0.420796	0.914858	0.394205	0.568826	0.328661	0.43577
13	0.730701	0.603800	0.166979	0.736147	0.556886	0.397290	0.708981	0.426150	0.61381
14	0.490989	0.214670	0.227432	0.391887	0.371790	0.367532	0.566306	0.361706	0.51993

Then we split the data into train and test sets using **train_test_split** from *sklearn.model_selection*, where its parameters are:

- test_size: It should be between 0 and 1, where it represents the percentage of the full dataset that will be the test set
- train_size: It should be between 0 and 1, if nothing is selected, then the value is automatically set to the complement of the test size
- random_state: Seed to allow replicate the results

Returns

X_train, X_test, y_train, y_test arrays

In [ ]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_
```

In [ ]:

```
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(670, 20) (330, 20) (670,) (330,)
```

Then we create our logistic regression using **LogisticRegression** from *sklearn.linear_model*

In [ ]:
```
model = LogisticRegression()

model.fit(X_train, y_train)
```

Out[ ]:
```
▾ LogisticRegression
LogisticRegression()
```

When fitting the model, we create the specific formula for the problem and then we can make a prediction using the test set. We can compare the prediction to the expected values and calculate a classification accuracy score using **accuracy_score** from *sklearn.metrics*

- normalize: True/False. If False, return the number of correctly classified samples. Otherwise, return the fraction of correctly classificed samples

In [ ]:
```
y_predicted = model.predict(X_test)
accuracy_data_leakaged = accuracy_score(y_test, y_predicted)
print(f"Accuracy of the model with data leakaged: {round(accuracy_data_leakaged*100
```

```
Accuracy of the model with data leakaged: 84.848 %
```

We already know that there was data leakage, and this estimate of model accuracy is wrong

Train-test evaluation using data preparation to the whole dataset and then split it into train and test datasets

1. Split dataset into train and test sets
2. Data Preparation in the train and test sets individually
3. Model evaluation

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_
```

In [ ]:
```
scaler = MinMaxScaler()

X_train = scaler.fit_transform(X_train) # Notice that we only fit the scale on the

X_test = scaler.transform(X_test)
```

In [ ]:
```
model = LogisticRegression()

model.fit(X_train, y_train)

y_predicted = model.predict(X_test)

accuracy_without_data_leakaged = accuracy_score(y_test, y_predicted)
```

```
print(f"Accuracy of the model without data leakaged: {round(accuracy_without_data_l
```

```
Accuracy of the model without data leakaged: 85.455 %
```

In this case, we can see that the estimate for the model is about 85.455%, which is more accurate than the estiamate with data leakage with just 84.848%. We would expect this to be an optimistic estimate with data leakage (better performance), although in this case, we can see that data leakage resulted in slightly worse performance. This might be because of the difficulty of the prediction task

Evaluating the model using Cross-Validation Evaluation

We'll use the same dataset as in the previous section

Cross-Validation Evaluation using data preparation to the whole dataset first

In [ ]:
```
scaler = MinMaxScaler()

X = scaler.fit_transform(X)
```

The k-fold cross-validation procedure must first be defined. We'll use repeated stratified 10-fold cross-validation which is a best practice for classification.

- Repeated means that the whole cross-validation procedure is repeated multiple times, three in this case.
- Stratified means that each group of rows will have the relative composition of examples from each class as the whole dataset.

We will use k = 10 or 10-fold cross-validation. This can be achieved using the RepeatedStratifiedKFold which can be configured to three repeats and 10 folds, and then using the cross_val_score() function to perform the procedure, passing in the defined model, cross-validation object, and metric to calculate, in this case, the accuracy

The parameters for this **RepeatedStratifiedKFold** from *sklearn.model_selection* are:

- n_splits: Number of folds, must be at least 2 (default: 5)
- n_repeats: Number of times cross-validator needs to be repeated (default: 10)
- random_state: Integer. Seed to replicate the results

The parameters for this **cross_val_score** from *sklearn.model_selection* are:

- estimator: Estimator object (function, it can be LogisticRegression, LinearRegression, etc) implementing "fit"
- X: independent variables

- y: target variable
- n_jobs: Int (default None). Number of jobs to run in parallel. Training the estimator and computing the score are parallelized over the cross-validation splits. None means 1. -1 means using all processors
- scoring: A str or a socrer callable object/function with signature scorer(estimator, X, y), which should return only a single value
- cv: Int, cross-validation generator or an interable. Determines the cross-validation splitting strategy. Possible inputs are:
  - None, to use the default 5-fold cross-validation
  - int, to specify the number of folds in a (Stratified)KFold
  - CV splitter,

In [ ]:
```
model = LogisticRegression()
cross_val = RepeatedStratifiedKFold(n_splits = 10, n_repeats = 3, random_state = 1)
scores = cross_val_score(model, X, y, scoring="accuracy", cv = cross_val, n_jobs =
```

In [ ]:
```
print(scores)
```

```
[0.86 0.91 0.88 0.81 0.83 0.84 0.81 0.84 0.88 0.84 0.84 0.86 0.85 0.83
 0.89 0.87 0.79 0.97 0.84 0.84 0.81 0.88 0.8  0.85 0.89 0.88 0.87 0.83
 0.83 0.87]
```

In [ ]:
```
print(f"Mean of the accuracy getting data leakage:{round(100*scores.mean(),3)} %, s
```

```
Mean of the accuracy getting data leakage:85.3 %, std = 3.607 %
```

Cross-Validation Evaluation with Correct Data Preparation

Data preparation without data leakage when using cross-validation is slightly more challenging, it requires that the data preparation method is prepared on the training set and applied to the train and test sets within the cross-validation procedure. We can achieve this by defining a modeling pipeline that defiens a sequence of data preparation steps to performs and endning in the model to fit and evaluate.

The evaluation procedure can be achieved if we create a pipeline class. This class takes a list of steps that define the pipeline. Each step in the list is a tuple with two elements.

- The first element is the name of the step (a string)
- The second element is the configured object of the step, such as a transform or a model. The model is only supported as the final step, although we can have as many transforms as we like in the sequence

In [ ]:
```
steps = [] #define the pipeline
steps.append(("scaler", MinMaxScaler()))
steps.append(("model", LogisticRegression()))

pipeline = Pipeline(steps = steps)
```

In [ ]:
```
cross_val = RepeatedStratifiedKFold(n_splits = 10, n_repeats=3, random_state=1)
scores = cross_val_score(pipeline, X, y, scoring = "accuracy", cv = cross_val, n_jo
print(scores)
```

```
[0.86 0.91 0.87 0.81 0.83 0.84 0.81 0.84 0.88 0.84 0.84 0.86 0.85 0.83
 0.89 0.88 0.8  0.97 0.84 0.84 0.81 0.88 0.81 0.85 0.89 0.88 0.87 0.84
 0.84 0.87]
```

In [ ]:
```
print(f"Mean of the accuracy getting data leakage:{round(100*scores.mean(),3)} %, s
```

```
Mean of the accuracy getting data leakage:85.433 %, std = 3.471 %
```

In this case, we can see that the model has an estimated accuracy of about 85.433% compared to the approach with the data leakage that achieved an accuracy of about 85.3%. As with the train-test example, removing data leakage has resulted in a slight improvement in performance when our intuition might suggest a drop given that data leakage often results in an optimistic estimate of model performance. Nevertheless, the examples demonstrate that data leakage may impact the estimate of model performance and how to correct data leakage by correctly performing data preparation after the data is split