TITANIC3 - train & test split

Required imports

```
import pandas as pd
from sklearn.model_selection import train_test_split
```

Read input file

```
data = pd.read_csv("/content/titanic_data.csv")
```

```
data.head(10)
```

	Unnamed: 0	pclass	survived	name	sex	age	sibsp	parch	ticket	far
0	0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.337
1	1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.550
2	2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.550
3	3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.550
4	4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.550
5	5	1	1	Anderson, Mr. Harry	male	48.0000	0	0	19952	26.550
6	6	1	1	Andrews, Miss. Kornelia Theodosia	female	63.0000	1	0	13502	77.958
7	7	1	0	Andrews, Mr. Thomas Jr	male	39.0000	0	0	112050	0.000
8	8	1	1	Appleton, Mrs. Edward Dale (Charlotte Lamson)	female	53.0000	2	0	11769	51.479
9	9	1	0	Artagaveytia, Mr. Ramon	male	71.0000	0	0	PC 17609	49.504

Next steps: View recommended plots

train_test_split function from scikit-learn

This function is used to split dataset into training and testing sets for further evaluation of how well model performs on new data. This helps to avoid overfitting of model and leads to better results on real-life data. There are many parameters to adjust the split to specific of our problem, such as test and train_size (where we can determine how much data we want to learn our model on and how much data will be used for evaluation) or stratify (passing this parameter we are making sure that proportions of classes in splits will be the same as in original dataset - useful for imbalanced data). Parameter random_state allows us to have control over randomization, we can have reproducible results, which is important for verification of results. This function returns list containing split datasets, the length of this array is always twice longer than input array since we are splitting into test and train set.

```
help(train_test_split)
```

```
            the value is automatically set to the complement of the test size.

        random_state : int, RandomState instance or None, default=None
            Controls the shuffling applied to the data before applying the split.
            Pass an int for reproducible output across multiple function calls.
            See :term:`Glossary <random_state>`.

        shuffle : bool, default=True
            Whether or not to shuffle the data before splitting. If shuffle=False
            then stratify must be None.

        stratify : array-like, default=None
            If not None, data is split in a stratified fashion, using this as
            the class labels.
            Read more in the :ref:`User Guide <stratification>`.

        Returns
        -------
        splitting : list, length=2 * len(arrays)
            List containing train-test split of inputs.

            .. versionadded:: 0.16
                If the input is sparse, the output will be a
                ``scipy.sparse.csr_matrix``. Else, output type is the same as the
                input type.

        Examples
        --------
        >>> import numpy as np
        >>> from sklearn.model_selection import train_test_split
        >>> X, y = np.arange(10).reshape((5, 2)), range(5)
        >>> X
        array([[0, 1],
               [2, 3],
               [4, 5],
               [6, 7],
               [8, 9]])
        >>> list(y)
        [0, 1, 2, 3, 4]

        >>> X_train, X_test, y_train, y_test = train_test_split(
        ...     X, y, test_size=0.33, random_state=42)
        ...
        >>> X_train
        array([[4, 5],
               [0, 1],
               [6, 7]])
        >>> y_train
        [2, 0, 3]
        >>> X_test
        array([[2, 3],
               [8, 9]])
        >>> y_test
        [1, 4]

        >>> train_test_split(y, shuffle=False)
        [[0, 1, 2], [3, 4]]
```

```
col_name = ['cabin', 'CabinReduced', 'sex']
```

```
X_train, X_test, y_train, y_test = train_test_split(data[col_name], data['survived'], random_state = 0, test_size = 0.2)
```

```
print(f"X_train: {X_train.shape}, X_test: {X_test.shape}, y_train: {y_train.shape}, y_test: {y_test.shape}")
```

```
X_train: (1047, 3), X_test: (262, 3), y_train: (1047,), y_test: (262,)
```

```
print(f"X_train: {X_train.columns.values}, X_test: {X_test.columns.values}, y_train: {y_train.name}, y_test: {y_test.name}")
```

```
X_train: ['cabin' 'CabinReduced' 'sex'], X_test: ['cabin' 'CabinReduced' 'sex'], y_train: survived, y_test: survived
```

```
percentage = X_test.shape[0] / (X_train.shape[0] + X_test.shape[0])
print(f"Proportion test set / whole dataset: {percentage}")
```

```
    Proportion test set / whole dataset: 0.2001527883880825
```

Dataset was divided into four sets - two training sets (one has three columns with independent variables and the other one has only one column with dependent variable - 'survived') and two testing sets (the same as for training sets). Proportion of split is around 1/5 which I was expecting after passing *test_size = 0.2* as parameter of function. Result is not exactly 20% since number of rows was not divisible by 5.

Label distribution in subsets

```
for col in X_train.columns.values:
  unique_test = [x for x in X_test[col].unique() if x not in X_train[col].unique()]
  print(f"There are {len(unique_test)} labels that appear in {col} column in train set and not in test set")
  if len(unique_test):
    print(unique_test)
```

```
    There are 24 labels that appear in cabin column in train set and not in test set
    [nan, 'E12', 'C104', 'A31', 'D11', 'D48', 'D10 D12', 'B38', 'D45', 'C50', 'C31', 'B82 B84', 'A32', 'C53', 'B10', 'C70', 'A23', 'C106',
    There are 1 labels that appear in CabinReduced column in train set and not in test set
    [nan]
    There are 0 labels that appear in sex column in train set and not in test set
```

```
X_test.CabinReduced.unique()
```

```
    array([nan, 'G', 'E', 'C', 'B', 'A', 'F', 'D'], dtype=object)
```

```
X_test.cabin.unique()
```

```
    array([nan, 'G6', 'E12', 'C104', 'B57 B59 B63 B66', 'A31', 'B96 B98',
           'B18', 'F33', 'C124', 'D11', 'D48', 'D10 D12', 'D30', 'B38', 'D45',
           'D15', 'C50', 'C31', 'E24', 'B82 B84', 'C85', 'C22 C26', 'A32',
           'C53', 'B78', 'B10', 'C70', 'A23', 'B41', 'C106', 'D37', 'C46',
           'B20', 'E58', 'D', 'B11', 'C101', 'B49', 'E25', 'F E69', 'B80',
           'C80', 'C52', 'C2', 'E39 E41', 'F4', 'D22', 'B51 B53 B55'],
          dtype=object)
```

For '**sex**' column there is no label that appears only in training set. For '**CabinReduced**" column our function returned nan as such value but as we can see nan appears also in testing set. It is because nan is not equal to nan. For '**cabin**' column there are 24 such labels but one is nan so we can reduce it to 23, which is still problematic, because after we train our model on train data and want to evaluate its performance on test dataset there will be 23 new labels our model has not seen before.

Variable encoding

```
mapping = {}

for var in col_name:
  mapping_var = {}
  for i, label in enumerate(data[var].unique()):
      mapping_var[label] = i
  mapping[var] = mapping_var
```

```
mapping.keys()
```

```
    dict_keys(['cabin', 'CabinReduced', 'sex'])
```

```
for var in col_name:
  X_train[f"{var}_mapped"] = X_train[var].map(mapping[var])
  X_test[f"{var}_mapped"] = X_test[var].map(mapping[var])
```

```
X_train.head(10)
```

	cabin	CabinReduced	sex	cabin_mapped	CabinReduced_mapped	sex_mapped
1118	NaN	NaN	male	6	5	1
44	E40	E	female	28	2	0
1072	NaN	NaN	male	6	5	1
1130	NaN	NaN	female	6	5	0
574	NaN	NaN	male	6	5	1
1217	F G73	F	male	181	7	1
500	NaN	NaN	male	6	5	1
958	NaN	NaN	female	6	5	0
269	A19	A	male	145	4	1
322	C32	C	female	164	1	0

Next steps: View recommended plots

```
X_test.head(10)
```

	cabin	CabinReduced	sex	cabin_mapped	CabinReduced_mapped	sex_mapped
1139	NaN	NaN	male	6	5	1
533	NaN	NaN	female	6	5	0
459	NaN	NaN	male	6	5	1
1150	NaN	NaN	male	6	5	1
393	NaN	NaN	male	6	5	1
1189	G6	G	female	185	8	0
5	E12	E	male	2	2	1
231	C104	C	male	130	1	1
330	NaN	NaN	male	6	5	1
887	NaN	NaN	male	6	5	1

Next steps: View recommended plots

```
col_name_mapped = [f"{col}_mapped" for col in col_name]


for col in col_name_mapped:
  print(f"In training set there are: {X_train[col].isna().sum()} missing values in {col} column and in testing set: {X_test[col].isna().sum()

     In training set there are: 0 missing values in cabin_mapped column and in testing set: 0
     In training set there are: 0 missing values in CabinReduced_mapped column and in testing set: 0
     In training set there are: 0 missing values in sex_mapped column and in testing set: 0
```

```
X_train[X_train['cabin'].isna()][['cabin', 'cabin_mapped']]
```

	cabin	cabin_mapped
1118	NaN	6
1072	NaN	6
1130	NaN	6
574	NaN	6
500	NaN	6
...	...	...
763	NaN	6
835	NaN	6
1216	NaN	6
559	NaN	6
684	NaN	6

803 rows × 2 columns

```
X_train[X_train['CabinReduced'].isna()][['CabinReduced', 'CabinReduced_mapped']]
```

	CabinReduced	CabinReduced_mapped
1118	NaN	5
1072	NaN	5
1130	NaN	5
574	NaN	5
500	NaN	5
...	...	...
763	NaN	5
835	NaN	5
1216	NaN	5
559	NaN	5
684	NaN	5

803 rows × 2 columns

There are no missing values in mapped columns because all nans were replaced by number during mapping. NaNs would appear if mapping was created only based on training or test dataset. I don't think that would be the best solution since we are losing information about labels that appear only in one set (all of them would have one value - nan). Also, we couldn't replace those nans with 0 since 0 already has some corresponding value. We would have to use some number that doesn't appear in our dataset yet.

How I would deal with this problem?

I would at first reduce 'cabin' variable (like we did to get 'CabinReduced' or more granular - level and side of ship) and drop it (no need for two columns that represent the same information). As seen above, this way there are no labels that appear only in one dataset since none of them appear very few times. We could also use parameter stratify in train_test_split function to ensure the same distribution of labels. Then we could encode this feature - it will not reduce cardinality but will make values easier for some models to process.

```
print(f"Number of labels before reducing and mapping: {X_train['cabin'].nunique(dropna = False)}")
```

```
Number of labels before reducing and mapping: 164
```

```
print(f"Number of labels before reducing and after mapping: {X_train['cabin_mapped'].nunique()}")
```

```
Number of labels before reducing and after mapping: 164
```

```
print(f"Number of labels after reducing and before mapping: {X_train['CabinReduced'].nunique(dropna = False)}")
```

```
Number of labels after reducing and before mapping: 9
```

```
print(f"Number of labels after reducing and before mapping: {X_train['CabinReduced_mapped'].nunique(dropna = False)}")
```

```
Number of labels after reducing and before mapping: 9
```

Reducing has impact on number of labels - there are fewer of them, while after mapping number of labels stays the same as it only encodes them in different way. Both of those actions may have impact on model performance.

Cardinality reduction can simplify our model, reduce risk of overfitting and generally make model more interpretable. However, some part of information is lost during this process, therefore it has to be well thought out and preceded by thorough EDA.

Variable encoding generally has positive impact on model performance because it ensures that data is represented in most appropriate way for the model (following all assumptions and requirements). However, assigning number to categorical value implies that there is some ordinal relationship between them, which is not always true and might lead to poor performance of model.