Lab #3: Preprocessing data - Appendix

The main aim of the appendix is to get familiar with the tasks of preprocessing data using Scikit-learn – a Python-based Machine Learning framework.

==========

TASK 1: HANDLE MISSING DATA USING SIMPLEIMPUTER

==========

Syntax:

```
class sklearn.impute.SimpleImputer(*, missing_values=nan, strategy='mean', fill_value=None, verbose=0, copy=True, add_indicator=False)
```

where,

- **missing_values:** int, float, str, np.nan or None, default=np.nan. The placeholder for the missing values. All occurrences of missing_values will be imputed.
- **Strategy:** str, default=’mean’, the imputation strategy.
  - “**mean**”: replace missing values using the mean along each column. Can only be used with numeric data.
  - “**median**”: replace missing values using the median along each column. Can only be used with numeric data.
  - “**most_frequent**”: replace missing using the most frequent value along each column. Can be used with strings or numeric data. If there is more than one such value, only the smallest is returned.
  - “**constant**”: replace missing values with fill_value. Can be used with strings or numeric data.
- **verbose:** int, default=0. Controls the verbosity of the imputer.
- **copy:** bool, default=True.
  - If True, a copy of X will be created.
  - If False, imputation will be done in-place whenever possible.
- **add_indicator:** bool, default=False.

- If True, a **MissingIndicator** transform will stack onto output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation.
- If a feature has no missing values at fit/train time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

Usage:

- **Step 1**: init SimpleImputer:
  - sklearn.impute.SimpleImputer(missing_values= {np.nan} strategy={"mean",“median”, “most_frequent”, ..})
- **Step 2**: fit the imputer on X:
  - imputer.fit(X[:, 1:3])
- **Step 3**: Impute all missing values in X:
  - imputer.transform(X[:, 1:3]) - Replace missing value from numerical Col 1 and Col 2

==========

TASK 2: SPLIT DATASET

==========

This task aim at spliting dataset into training set and test set by using train_test_split of sklearn-Model Selection.

Syntax:

sklearn.model_selection.train_test_split(**arrays, test_size=None, train_size=None, random_state=None, shuffle=True, stratify=None*)

where,

- ***arrays:** sequence of indexables with same length / shape[0]. Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.
- **test_size:** float or int, default=None.
  - If **float**, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split.
  - If **int**, represents the absolute number of test samples.

- If **None**, the value is set to the complement of the train size.
- If **train_size** is also None, it will be set to 0.25.

- **train_size:** float or int, default=None
  - If **float**, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split.
  - If **int**, represents the absolute number of train samples.
  - If None, 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.
- **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.

Usage:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 1)
```

Where,

- X_train: training set (features)
- X_test: test set (features)
- y_train: training set (output)
- y_test: test set (output)
- test_size: the size of tesing set
- random_state: the seed for random state so that we can have the same training & test sets anytime.

==========

TASK 3: FEATURE SCALING

==========

Feature scaling: scale all the features in the same scale to prevent 1 feature dominates the others & then neglected by ML Model.

There are 2 main Feature Scaling Technique: Standardisation & Normalisation.

- **Standardisation**: This makes the dataset, center at 0 i.e mean at 0, and changes the standard deviation value to 1 (μ: mean, σ: standard deviation)

$$x' = \frac{x - \mu(x)}{\sigma(x)}$$

  - *Usage*: apply all the situations

- **Normalisation**: This makes the dataset in range [0, 1]

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

  - *Usage*: apply when the all the features in the data set have the **normal distribution**

Notice that, Feature scaling must be done after splitting training and test sets => it does not include the mean & median of both training set and test sets.

Usage:

- **Standardisation Feature Scaling:**

  - Using StandardScaler from sklearn.preprocessing

    from sklearn.preprocessing import **StandardScaler**
    sc = **StandardScaler**()

  - For X_train: apply **StandardScaler** by using **fit_transform**

    X_train[:,3:] = sc.fit_transform(X_train[:,3:])

  - For X_test: apply **StandardScaler** only use transform, because we want to apply the SAME scale as X_train

    X_test[:,3:] = sc.transform(X_test[:,3:])

- **MinMaxScaler:** For each value in a feature, **MinMaxScaler** subtracts the minimum value in the feature and then divides by the range. The range is the difference between the original maximum and original minimum.

  - Syntax:

```
klearn.preprocessing.MinMaxScaler(feature_range=(0, 1), *, copy=True, clip=False)
```

    - **feature_range:** tuple (min, max), default=(0, 1)
      - Desired range of transformed data.
    - **copy:** bool, default=True
      - Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).
    - **clip:** bool, default=False
      - Set to True to clip transformed values of held-out data to provided `feature range`.

  - Example:

```
from sklearn import preprocessing
mm_scaler = preprocessing.MinMaxScaler()
X_train_minmax = mm_scaler.fit_transform(X_train)
mm_scaler.transform(X_test)
```

==========

TASK 4: DISCRETIZATION

==========

Discretization (otherwise known as quantization or binning) provides a way to partition continuous features into discrete values. Certain datasets with continuous features may benefit from discretization, because discretization can transform the dataset of continuous attributes to one with only nominal attributes.

- **K-bins discretization:**

sklearn.preprocessing.KBinsDiscretizer(*n_bins=5, *, encode='onehot', strategy='quantile', dtype=None*)

- **n_bins:** int or array-like of shape (n_features,), default=5. The number of bins to produce. Raises ValueError if `n_bins < 2`.
- **encode:** {'onehot', 'onehot-dense', 'ordinal'}, default='onehot'. Method used to encode the transformed result.
    - '**onehot'**: Encode the transformed result with one-hot encoding and return a sparse matrix. Ignored features are always stacked to the right.
    - '**onehot-dense**': Encode the transformed result with one-hot encoding and return a dense array. Ignored features are always stacked to the right.
    - '**ordinal'**: Return the bin identifier encoded as an integer value.

- **strategy**{'uniform', 'quantile', 'kmeans'}, default='quantile'. Strategy used to define the widths of the bins.
    - '**uniform'**: All bins in each feature have identical widths.
    - '**quantile':** All bins in each feature have the same number of points.
    - **'kmeans'**: Values in each bin have the same nearest center of a 1D k-means cluster.

- **dtype:**{np.float32, np.float64}, default=None. The desired data-type for the output. If None, output dtype is consistent with input dtype. Only np.float32 and np.float64 are supported.

==========

TASK 4: CATEGORICAL TO NUMERIC

==========

In general, machine learning is always good at dealing with numeric values. In order to build machine learning models by using text data, there is a need to convert categorical data into numeric form. Pandas provides two approaches for such a task.

For a given dataset having categorical attributes as follows:

	Person	Education	Salary(annum)
0	amit	Under-Graduate	$90k
1	vishal	Diploma	$80k
2	john	Under-Graduate	$90k
3	marry	Diploma	$60k
4	sherin	Under-Graduate	$90k
5	komal	Under-Graduate	$100k
6	jay	Under-Graduate	$60k
7	shree	Under-Graduate	$100k
8	kishore	Diploma	$90k
9	geetha	Diploma	$70k
10	savitha	Under-Graduate	$50k
11	vinith	Under-Graduate	$90k

s

1) Using **replace** method:

Syntax: Replace values given in *to_replace* with *value*.

```
replace(to_replace=None, value=None, inplace=False, limit=None,
regex=False, method='pad')
```

where,

- to_replace: Values that will be replaced
- inplace: If True, in place replacement
- limit: Maximum size gap to forward or backward fill
- regrex: Whether to interpret *to_replace* and/or *value* as regular expressions
- method: The method to use when for replacement, when to_replace is a scalar, list or tuple and value is None, {'pad', 'ffill', 'bfill', None}

Usage:

```
#import pandas
import pandas as pd

# read csv file
df = pd.read_csv('data.csv')
```

```
# replacing values
df['Education'].replace(['Under-Graduate', 'Diploma '],
                        [0, 1], inplace=True)
```

Results:

	Person	Education	Salary(annum)
0	amit	0	$90k
1	vishal	1	$80k
2	john	0	$90k
3	marry	1	$60k
4	sherin	0	$90k
5	komal	0	$100k
6	jay	0	$60k
7	shree	0	$100k
8	kishore	1	$90k
9	geetha	1	$70k
10	savitha	0	$50k
11	vinith	0	$90k

2) Using **get_dummies** method:

Replacing the values is not the most efficient way to convert them. Pandas provide a method called **get_dummies** which will return the dummy variable columns.

Syntax:

pandas.get_dummies(data, prefix=None, prefix_sep='_', dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=None)

pandas.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=False, copy=True)

Usage:

– **Step 1**: create dummy columns

```
#import pandas
```

```
import pandas as pd

# read csv
df = pd.read_csv('salary.csv')

# get the dummies and store it in a variable
dummies = pd.get_dummies(df.Education)
```

	Diploma	Master's	Under-Graduate
0	0	0	1
1	1	0	0
2	0	1	0
3	0	1	0
4	0	0	1
5	1	0	0
6	0	0	1
7	0	0	1
8	0	0	1
9	0	0	1
10	1	0	0
11	1	0	0
12	0	0	1
13	0	0	1

– **Step 2**: concatenate colums. The next step is to concatenate the dummies columns into the data frame. In Pandas, there is a concat() method, which you can call to join two data frames

```
# Concatenate the dummies to original dataframe
merged = pd.concat([df, dummies], axis='columns')
```

	Person	Education	Salary(annum)	Diploma	Master's	Under-Graduate
0	amit	Under-Graduate	$90k	0	0	1
1	vishal	Diploma	$80k	1	0	0
2	Sindhu	Master's	$160k	0	1	0
3	Sanju	Master's	$200k	0	1	0
4	john	Under-Graduate	$90k	0	0	1
5	marry	Diploma	$60k	1	0	0
6	sherin	Under-Graduate	$90k	0	0	1
7	komal	Under-Graduate	$100k	0	0	1
8	jay	Under-Graduate	$60k	0	0	1
9	shree	Under-Graduate	$100k	0	0	1
10	kishore	Diploma	$90k	1	0	0
11	geetha	Diploma	$70k	1	0	0
12	savitha	Under-Graduate	$50k	0	0	1
13	vinith	Under-Graduate	$90k	0	0	1

- **Step 3**: drop columns

```
# drop the values
merged.drop(['Education', 'Under-Graduate'], axis='columns')
```

	Person	Salary(annum)	Diploma	Master's
0	amit	$90k	0	0
1	vishal	$80k	1	0
2	Sindhu	$160k	0	1
3	Sanju	$200k	0	1
4	john	$90k	0	0
5	marry	$60k	1	0
6	sherin	$90k	0	0
7	komal	$100k	0	0
8	jay	$60k	0	0
9	shree	$100k	0	0
10	kishore	$90k	1	0
11	geetha	$70k	1	0
12	savitha	$50k	0	0
13	vinith	$90k	0	0

==========

PRACTICE

==========

For given 2 datasets (preprocessing_weather.csv, preprocessing_salary_dummy.csv),

- Identify issues with these datasets
- Then, apply preprocessing techniques to the datasets