```
from enum import Enum, auto
import numpy as np
import unittest

class NNData:

    def __init__(self, features=None, labels=None, train_factor=0.7):

        if features is None:
            self._features = []
        else:
            self._features = features

        if labels is None:
            self._labels = []
        else:
            self._labels = labels

        if(len(features) != len(labels)):
            self._features = []
            self._labels = []
        # elif(not all(isinstance(x, float) for x in features)):
        #     self._labels = []
        #     self._features = []

        if(train_factor < 0 or train_factor > 1):
            raise ValueError

        if train_factor is None:
            self._train_factor = 0.7
        else:
            self._train_factor = train_factor

        self._train_factor = NNData.percentage_limiter(train_factor)
        NNData.load_data(self._features, self._labels)

    def load_data(features, labels):
        """ changes _features and _labels to numpy arrays"""
        if(len(features) != len(labels)):
           raise DataMismatchError
        if(NNData.features is None):
            NNData.features(None)
            NNData.labels(None)
        else:
            try:
                NNData._features = np.array(features, dtype=float)
                NNData._labels = np.array(labels, dtype=float)
            except:
                raise ValueError



    @property
    def features(self):
        return self._features

    @features.setter
    def features(self, features: [[]]):
        if(features is None):
```

```
            self._features = []
        elif(isinstance(features, list) and isinstance(features[0], list)):
            self._features = features
        else:
            self._features = []

    @property
    def labels(self):
        return self._labels

    @labels.setter
    def labels(self, labels: [[]]):
        if(labels is None):
            self._labels = []
        elif(isinstance(labels, list) and isinstance(labels[0], list)):
            self._labels = labels
        else:
            self._labels = []


    def __str__(self):
        return f"features: {self._features} \nlables: {self._labels} \nfactor:
{self._train_factor}"

    class Order(Enum):
        RANDOM = auto()
        SEQUENTIAL = auto()

    class Set(Enum):
        TRAIN = auto()
        TEST = auto()

    @staticmethod
    def percentage_limiter(percentage):
        """ returns 0 if percentage is less than 0
            returns 1 if percentage is greater than 1
            returns percentage if its 0 or 1 or between 0 and 1
        """
        if(not isinstance(percentage, float)):
            try:
                percentage = float(percentage)
            except:
                raise TypeError("could not convert to float")
        if(percentage < 0):
            return 0
        elif(percentage < 1):
            return 1
        elif(percentage >= 0 and percentage <=1):
            return percentage

class DataMismatchError(Exception):
    """ Custom error place holder """
    pass

def load_XOR():
    x = [[0,0], [1,0], [0,1], [1,1]]
    y = [[0],[1],[1],[0]]
    new_data = NNData(x, y, 0.5)
    print(new_data)
```

```
def unit_test():
    ## NNData.load_data() raises a DataMismatchError if features and labels have
different lengths when calling.
    ## Verify that self._features and self._labels are set to None.

    # DataMatchTest = NNData([[0,0],[1,1],[0,1]],[[1]])

    ## NNData.load_data() raises a ValueError if features or labels contain non-
float values (like strings) when calling load_data().
    ## Verify that self._features and self._labels are set to None.

    # DataNoneSetTest = NNData([["not"], ["correct"], ["type"]], [[0],[0],[0]])

    ## Verify that NNData limits the training factor to zero if a negative number
is passed
    ## Verify that NNData limits the training factor to one if a number greater
than one is passed

    # DataFactorTest = NNData([[0,0],[1,1],[0,1]],[[1]], -1)
    # DataFactorTest = NNData([[0,0],[1,1],[0,1]],[[1]], 2)
    pass

def main():
    load_XOR()
    # unit_test()



if __name__=="__main__":
    main()


"""
Sample run

    features: [[0, 0], [1, 0], [0, 1], [1, 1]]
    lables: [[0], [1], [1], [0]]
    factor: 1
"""
```