ann-model-in-classification

February 20, 2024

0.1 Artificial Neural Networks

0.1.1 Importing the libraries

[1]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
```

```
C:\Users\Soubhik\anaconda3\Anaconda\lib\site-
packages\pandas\core\computation\expressions.py:21: UserWarning: Pandas requires
version '2.8.0' or newer of 'numexpr' (version '2.7.3' currently installed).
  from pandas.core.computation.check import NUMEXPR_INSTALLED
C:\Users\Soubhik\anaconda3\Anaconda\lib\site-
packages\pandas\core\arrays\masked.py:62: UserWarning: Pandas requires version
'1.3.4' or newer of 'bottleneck' (version '1.3.2' currently installed).
  from pandas.core import (
```

[2]:
```
tf.__version__
```

[2]:
```
'2.13.0'
```

0.2 Part 1 DATA PREPROCESSING

0.2.1 Importing the dataset

[3]:
```
dataset = pd.read_csv('Churn_Modelling.csv')
X = dataset.iloc[:, 3:-1].values
y = dataset.iloc[:, -1].values
```

[4]:
```
X
```

[4]:
```
array([[619, 'France', 'Female', …, 1, 1, 101348.88],
       [608, 'Spain', 'Female', …, 0, 1, 112542.58],
       [502, 'France', 'Female', …, 1, 0, 113931.57],
       …,
       [709, 'France', 'Female', …, 0, 1, 42085.58],
       [772, 'Germany', 'Male', …, 1, 0, 92888.52],
       [792, 'France', 'Female', …, 1, 0, 38190.78]], dtype=object)
```

[5]:
```
y
```

[5]:
```
array([1, 0, 1, …, 1, 1, 0], dtype=int64)
```

0.2.2 Encoding categorical data

1. Label Encoding of the Gender column

[6]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
X[:,2] = le.fit_transform(X[:,2])
```

[7]:
```
print(X)
```

```
[[619 'France' 0 … 1 1 101348.88]
 [608 'Spain' 0 … 0 1 112542.58]
 [502 'France' 0 … 1 0 113931.57]
 …
 [709 'France' 0 … 0 1 42085.58]
 [772 'Germany' 1 … 1 0 92888.52]
 [792 'France' 0 … 1 0 38190.78]]
```

2. One Hot Encoding of the Geography column

[8]:
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])],
 ↪remainder='passthrough')
X = np.array(ct.fit_transform(X))
```

[9]:
```
print(X)
```

```
[[1.0 0.0 0.0 … 1 1 101348.88]
 [0.0 0.0 1.0 … 0 1 112542.58]
 [1.0 0.0 0.0 … 1 0 113931.57]
 …
 [1.0 0.0 0.0 … 0 1 42085.58]
 [0.0 1.0 0.0 … 1 0 92888.52]
 [1.0 0.0 0.0 … 1 0 38190.78]]
```

0.2.3 Splitting the dataset into Training set and Testing dataset

[10]:
```
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 = 0)
```

[11]:
```
X_train = np.array(X_train, dtype=np.float32)
y_train = np.array(y_train, dtype=np.float32)
```

0.2.4 Feature Scaling

[12]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train[:, 3:] = sc.fit_transform(X_train[:, 3:])
X_test[:, 3:] = sc.transform(X_test[:, 3:])
```

0.3 Part 2 Building The ANN

1. Initialising The ANN

[13]:
```
ann = tf.keras.models.Sequential()
```

2. Adding Input Layer or First Hidden Layer

[14]:
```
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

3. Adding Second Hidden Layer

[15]:
```
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

4. Adding Output Layer

[16]:
```
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

0.4 Part 3 Training an ANN

0.4.1 Compiling an ANN

[17]:
```
ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =␣
 ↪['accuracy'])
```

0.4.2 Training an ANN

[18]:
```
ann.fit(X_train, y_train, batch_size = 32, epochs = 100)
```

```
Epoch 1/100
250/250 [==============================] - 4s 5ms/step - loss: 0.5688 -
accuracy: 0.7570
Epoch 2/100
250/250 [==============================] - 1s 5ms/step - loss: 0.4423 -
accuracy: 0.8039
Epoch 3/100
250/250 [==============================] - 1s 5ms/step - loss: 0.4262 -
accuracy: 0.8201
Epoch 4/100
250/250 [==============================] - 1s 5ms/step - loss: 0.4184 -
accuracy: 0.8210
Epoch 5/100
```

```
250/250 [==============================] - 1s 5ms/step - loss: 0.4127 -
accuracy: 0.8244
Epoch 6/100
250/250 [==============================] - 1s 5ms/step - loss: 0.4080 -
accuracy: 0.8274
Epoch 7/100
250/250 [==============================] - 1s 4ms/step - loss: 0.4033 -
accuracy: 0.8301
Epoch 8/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3991 -
accuracy: 0.8309
Epoch 9/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3956 -
accuracy: 0.8331
Epoch 10/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3916 -
accuracy: 0.8344
Epoch 11/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3878 -
accuracy: 0.8356
Epoch 12/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3835 -
accuracy: 0.8404
Epoch 13/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3806 -
accuracy: 0.8426
Epoch 14/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3767 -
accuracy: 0.8426
Epoch 15/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3734 -
accuracy: 0.8454
Epoch 16/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3708 -
accuracy: 0.8474
Epoch 17/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3683 -
accuracy: 0.8490
Epoch 18/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3659 -
accuracy: 0.8478
Epoch 19/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3631 -
accuracy: 0.8497
Epoch 20/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3614 -
accuracy: 0.8504
Epoch 21/100
```

```
250/250 [==============================] - 1s 5ms/step - loss: 0.3594 -
accuracy: 0.8520
Epoch 22/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3577 -
accuracy: 0.8537
Epoch 23/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3562 -
accuracy: 0.8551
Epoch 24/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3548 -
accuracy: 0.8535
Epoch 25/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3539 -
accuracy: 0.8546
Epoch 26/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3523 -
accuracy: 0.8565
Epoch 27/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3510 -
accuracy: 0.8566
Epoch 28/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3499 -
accuracy: 0.8571
Epoch 29/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3486 -
accuracy: 0.8569
Epoch 30/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3473 -
accuracy: 0.8577
Epoch 31/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3467 -
accuracy: 0.8569
Epoch 32/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3459 -
accuracy: 0.8580
Epoch 33/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3450 -
accuracy: 0.8589
Epoch 34/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3439 -
accuracy: 0.8590
Epoch 35/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3430 -
accuracy: 0.8590
Epoch 36/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3427 -
accuracy: 0.8569
Epoch 37/100
```

```
250/250 [==============================] - 1s 5ms/step - loss: 0.3415 -
accuracy: 0.8602
Epoch 38/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3405 -
accuracy: 0.8596
Epoch 39/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3400 -
accuracy: 0.8597
Epoch 40/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3396 -
accuracy: 0.8608
Epoch 41/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3394 -
accuracy: 0.8581
Epoch 42/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3388 -
accuracy: 0.8599
Epoch 43/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3385 -
accuracy: 0.8597
Epoch 44/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3380 -
accuracy: 0.8629
Epoch 45/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3375 -
accuracy: 0.8619
Epoch 46/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3376 -
accuracy: 0.8611
Epoch 47/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3370 -
accuracy: 0.8619
Epoch 48/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3370 -
accuracy: 0.8627
Epoch 49/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3366 -
accuracy: 0.8620
Epoch 50/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3365 -
accuracy: 0.8630
Epoch 51/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3359 -
accuracy: 0.8615
Epoch 52/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3358 -
accuracy: 0.8624
Epoch 53/100
```

```
250/250 [==============================] - 1s 5ms/step - loss: 0.3358 -
accuracy: 0.8616
Epoch 54/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3357 -
accuracy: 0.8626
Epoch 55/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3354 -
accuracy: 0.8627
Epoch 56/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3350 -
accuracy: 0.8640
Epoch 57/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3358 -
accuracy: 0.8629
Epoch 58/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3347 -
accuracy: 0.8648
Epoch 59/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3350 -
accuracy: 0.8620
Epoch 60/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3350 -
accuracy: 0.8636
Epoch 61/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3344 -
accuracy: 0.8624
Epoch 62/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3351 -
accuracy: 0.8646
Epoch 63/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3344 -
accuracy: 0.8625
Epoch 64/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3346 -
accuracy: 0.8635
Epoch 65/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3339 -
accuracy: 0.8633
Epoch 66/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3344 -
accuracy: 0.8611
Epoch 67/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3339 -
accuracy: 0.8636
Epoch 68/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3340 -
accuracy: 0.8625
Epoch 69/100
```

```
250/250 [==============================] - 1s 5ms/step - loss: 0.3339 -
accuracy: 0.8635
Epoch 70/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3341 -
accuracy: 0.8624
Epoch 71/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3334 -
accuracy: 0.8636
Epoch 72/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3341 -
accuracy: 0.8625
Epoch 73/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3336 -
accuracy: 0.8652
Epoch 74/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3332 -
accuracy: 0.8626
Epoch 75/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3332 -
accuracy: 0.8640
Epoch 76/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3334 -
accuracy: 0.8618
Epoch 77/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3334 -
accuracy: 0.8620
Epoch 78/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3334 -
accuracy: 0.8622
Epoch 79/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3332 -
accuracy: 0.8631
Epoch 80/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3331 -
accuracy: 0.8630
Epoch 81/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3328 -
accuracy: 0.8639
Epoch 82/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3332 -
accuracy: 0.8629
Epoch 83/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3335 -
accuracy: 0.8634
Epoch 84/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3322 -
accuracy: 0.8648
Epoch 85/100
```

```
250/250 [==============================] - 1s 5ms/step - loss: 0.3334 -
accuracy: 0.8625
Epoch 86/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3320 -
accuracy: 0.8651
Epoch 87/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3326 -
accuracy: 0.8651
Epoch 88/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3323 -
accuracy: 0.8630
Epoch 89/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3325 -
accuracy: 0.8625
Epoch 90/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3321 -
accuracy: 0.8644
Epoch 91/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3320 -
accuracy: 0.8648
Epoch 92/100
250/250 [==============================] - 1s 4ms/step - loss: 0.3323 -
accuracy: 0.8635
Epoch 93/100
250/250 [==============================] - 1s 6ms/step - loss: 0.3313 -
accuracy: 0.8641
Epoch 94/100
250/250 [==============================] - 2s 6ms/step - loss: 0.3323 -
accuracy: 0.8658
Epoch 95/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3316 -
accuracy: 0.8633
Epoch 96/100
250/250 [==============================] - 1s 6ms/step - loss: 0.3317 -
accuracy: 0.8648
Epoch 97/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3323 -
accuracy: 0.8622
Epoch 98/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3317 -
accuracy: 0.8626
Epoch 99/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3313 -
accuracy: 0.8643
Epoch 100/100
250/250 [==============================] - 1s 5ms/step - loss: 0.3317 -
accuracy: 0.8640
```

[18]:
```
<keras.src.callbacks.History at 0x2a2483da850>
```

0.5 Part 4 Predicting the output

Use our model to predict the following output with following information if he will leave the bank or not: 1. Geography : France 2. Credit Score : 600 3. Gender : Male 4. Age : 40 years old 5. Tenure : 3 years 6. Balance : 600007.$NumberofProducts$: 28.$Doesthiscustomerhasacreditcard?Yes$9.$Isthiscustomeranactivemember?Yes$10.$EstimatedSalary$: 50000

Should we say Goodbye to this customer ?

Make sure to use proper Encoding techniques and the scale as used to train the ANN model

[19]:
```
X = [1., 0., 0., 600, 1, 40, 3, 60000, 2, 1, 1, 50000]
X = np.array(X).reshape(1, -1)
X[:, 3:] = sc.transform(X[:, 3:])
print(ann.predict(X))
```

```
1/1 [==============================] - 1s 514ms/step
[[0.04895078]]
```

NOTE :

1. Predict() method expects a 2D array as a format of all inputs. And putting it in double brackets makes it a 2D array.
   2. The country “France” is not taken input as a string but, it is taken input as a [1, 0, 0] as an encoded vector.

0.6 Making Confusion Matrix

1. Predicting Test Results

[21]:
```
y_pred = ann.predict(X_test.astype(float))
y_pred = (y_pred > 0.5)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.
 ↪reshape(len(y_test),1)),1))
```

```
63/63 [==============================] - 0s 5ms/step
[[0 0]
 [0 1]
 [0 0]
 …
 [0 0]
 [0 0]
 [0 0]]
```

2. Making Confusion Matrix

[22]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
accuracy_score(y_test, y_pred)
```

```
[[1511   84]
 [ 192  213]]
```

[22]:
```
0.862
```

0.7 Appendix

[23]:
```
help(ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])],␣
 ↪remainder='passthrough'))
```

```
Help on ColumnTransformer in module sklearn.compose._column_transformer object:

class ColumnTransformer(sklearn.base.TransformerMixin,
sklearn.utils.metaestimators._BaseComposition)
 |  ColumnTransformer(transformers, *, remainder='drop', sparse_threshold=0.3,
n_jobs=None, transformer_weights=None, verbose=False)
 |
 |  Applies transformers to columns of an array or pandas DataFrame.
 |
 |  This estimator allows different columns or column subsets of the input
 |  to be transformed separately and the features generated by each transformer
 |  will be concatenated to form a single feature space.
 |  This is useful for heterogeneous or columnar data, to combine several
 |  feature extraction mechanisms or transformations into a single transformer.
 |
 |  Read more in the :ref:`User Guide <column_transformer>`.
 |
 |  .. versionadded:: 0.20
 |
 |  Parameters
 |  ----------
 |  transformers : list of tuples
 |      List of (name, transformer, columns) tuples specifying the
 |      transformer objects to be applied to subsets of the data.
 |
 |      name : str
 |          Like in Pipeline and FeatureUnion, this allows the transformer and
 |          its parameters to be set using ``set_params`` and searched in grid
 |          search.
 |      transformer : {'drop', 'passthrough'} or estimator
 |          Estimator must support :term:`fit` and :term:`transform`.
 |          Special-cased strings 'drop' and 'passthrough' are accepted as
```

```
 |          well, to indicate to drop the columns or to pass them through
 |          untransformed, respectively.
 |      columns :  str, array-like of str, int, array-like of int,
array-like of bool, slice or callable
 |          Indexes the data on its second axis. Integers are interpreted as
 |          positional columns, while strings can reference DataFrame columns
 |          by name.  A scalar string or int should be used where
 |          ``transformer`` expects X to be a 1d array-like (vector),
 |          otherwise a 2d array will be passed to the transformer.
 |          A callable is passed the input data `X` and can return any of the
 |          above. To select multiple columns by name or dtype, you can use
 |          :obj:`make_column_selector`.
 |
 |  remainder : {'drop', 'passthrough'} or estimator, default='drop'
 |      By default, only the specified columns in `transformers` are
 |      transformed and combined in the output, and the non-specified
 |      columns are dropped. (default of ``'drop'``).
 |      By specifying ``remainder='passthrough'``, all remaining columns that
 |      were not specified in `transformers` will be automatically passed
 |      through. This subset of columns is concatenated with the output of
 |      the transformers.
 |      By setting ``remainder`` to be an estimator, the remaining
 |      non-specified columns will use the ``remainder`` estimator. The
 |      estimator must support :term:`fit` and :term:`transform`.
 |      Note that using this feature requires that the DataFrame columns
 |      input at :term:`fit` and :term:`transform` have identical order.
 |
 |  sparse_threshold : float, default=0.3
 |      If the output of the different transformers contains sparse matrices,
 |      these will be stacked as a sparse matrix if the overall density is
 |      lower than this value. Use ``sparse_threshold=0`` to always return
 |      dense.  When the transformed output consists of all dense data, the
 |      stacked result will be dense, and this keyword will be ignored.
 |
 |  n_jobs : int, default=None
 |      Number of jobs to run in parallel.
 |      ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
 |      ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
 |      for more details.
 |
 |  transformer_weights : dict, default=None
 |      Multiplicative weights for features per transformer. The output of the
 |      transformer is multiplied by these weights. Keys are transformer names,
 |      values the weights.
 |
 |  verbose : bool, default=False
 |      If True, the time elapsed while fitting each transformer will be
 |      printed as it is completed.
```

```
|
|  Attributes
|  ----------
|  transformers_ : list
|      The collection of fitted transformers as tuples of
|      (name, fitted_transformer, column). `fitted_transformer` can be an
|      estimator, 'drop', or 'passthrough'. In case there were no columns
|      selected, this will be the unfitted transformer.
|      If there are remaining columns, the final element is a tuple of the
|      form:
|      ('remainder', transformer, remaining_columns) corresponding to the
|      ``remainder`` parameter. If there are remaining columns, then
|      ``len(transformers_)==len(transformers)+1``, otherwise
|      ``len(transformers_)==len(transformers)``.
|
|  named_transformers_ : :class:`~sklearn.utils.Bunch`
|      Read-only attribute to access any transformer by given name.
|      Keys are transformer names and values are the fitted transformer
|      objects.
|
|  sparse_output_ : bool
|      Boolean flag indicating whether the output of ``transform`` is a
|      sparse matrix or a dense numpy array, which depends on the output
|      of the individual transformers and the `sparse_threshold` keyword.
|
|  Notes
|  -----
|  The order of the columns in the transformed feature matrix follows the
|  order of how the columns are specified in the `transformers` list.
|  Columns of the original feature matrix that are not specified are
|  dropped from the resulting transformed feature matrix, unless specified
|  in the `passthrough` keyword. Those columns specified with `passthrough`
|  are added at the right to the output of the transformers.
|
|  See Also
|  --------
|  make_column_transformer : Convenience function for
|      combining the outputs of multiple transformer objects applied to
|      column subsets of the original feature space.
|  make_column_selector : Convenience function for selecting
|      columns based on datatype or the columns name with a regex pattern.
|
|  Examples
|  --------
|  >>> import numpy as np
|  >>> from sklearn.compose import ColumnTransformer
|  >>> from sklearn.preprocessing import Normalizer
|  >>> ct = ColumnTransformer(
```

```
 |  ...     [("norm1", Normalizer(norm='l1'), [0, 1]),
 |  ...      ("norm2", Normalizer(norm='l1'), slice(2, 4))])
 |  >>> X = np.array([[0., 1., 2., 2.],
 |  ...               [1., 1., 0., 1.]])
 |  >>> # Normalizer scales each row of X to unit norm. A separate scaling
 |  >>> # is applied for the two first and two last elements of each
 |  >>> # row independently.
 |  >>> ct.fit_transform(X)
 |  array([[0. , 1. , 0.5, 0.5],
 |         [0.5, 0.5, 0. , 1. ]])
 |
 |  Method resolution order:
 |      ColumnTransformer
 |      sklearn.base.TransformerMixin
 |      sklearn.utils.metaestimators._BaseComposition
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, transformers, *, remainder='drop', sparse_threshold=0.3,
n_jobs=None, transformer_weights=None, verbose=False)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y=None)
 |      Fit all transformers using X.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, dataframe} of shape (n_samples, n_features)
 |          Input data, of which specified subsets are used to fit the
 |          transformers.
 |
 |      y : array-like of shape (n_samples,...), default=None
 |          Targets for supervised learning.
 |
 |      Returns
 |      -------
 |      self : ColumnTransformer
 |          This estimator
 |
 |  fit_transform(self, X, y=None)
 |      Fit all transformers, transform the data and concatenate results.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, dataframe} of shape (n_samples, n_features)
 |          Input data, of which specified subsets are used to fit the
```

```
 |          transformers.
 |
 |      y : array-like of shape (n_samples,), default=None
 |          Targets for supervised learning.
 |
 |      Returns
 |      -------
 |      X_t : {array-like, sparse matrix} of                 shape (n_samples,
sum_n_components)
 |          hstack of results of transformers. sum_n_components is the
 |          sum of n_components (output dimension) over transformers. If
 |          any result is a sparse matrix, everything will be converted to
 |          sparse matrices.
 |
 |  get_feature_names(self)
 |      Get feature names from all transformers.
 |
 |      Returns
 |      -------
 |      feature_names : list of strings
 |          Names of the features produced by transform.
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Returns the parameters given in the constructor as well as the
 |      estimators contained within the `transformers` of the
 |      `ColumnTransformer`.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : dict
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **kwargs)
 |      Set the parameters of this estimator.
 |
 |      Valid parameter keys can be listed with ``get_params()``. Note that you
 |      can directly set the parameters of the estimators contained in
 |      `transformers` of `ColumnTransformer`.
 |
 |      Returns
```

```
 |      -------
 |      self
 |
 |  transform(self, X)
 |      Transform X separately by each transformer, concatenate results.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, dataframe} of shape (n_samples, n_features)
 |          The data to be transformed by subset.
 |
 |      Returns
 |      -------
 |      X_t : {array-like, sparse matrix} of                 shape (n_samples,
sum_n_components)
 |          hstack of results of transformers. sum_n_components is the
 |          sum of n_components (output dimension) over transformers. If
 |          any result is a sparse matrix, everything will be converted to
 |          sparse matrices.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  named_transformers_
 |      Access the fitted transformer by name.
 |
 |      Read-only attribute to access any transformer by given name.
 |      Keys are transformer names and values are the fitted transformer
 |      objects.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.TransformerMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from
sklearn.utils.metaestimators._BaseComposition:
 |
```

```
 |  __annotations__ = {'steps': typing.List[typing.Any]}
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
```

[ ]:

```

```