Mudcard

- **Is there ever a reason to use stratification on a feature variable? Eg. If we have an indicator variable signifying whether market conditions are currently 'extreme'**
    - usually you stratify on the target variable and not on a feature
    - btw, we covered how to stratify on the classification target variable
    - sometimes it is necessary to stratify on the regression target variable but sklearn has no function for it as far as I know
        - if you ever encounter this problem, you'll need to write code for it
- **For KFold splitting, how do we interpret the results from the k models? Do we somehow calculate an average of these models?**
    - yes, if you have k folds, you'll calculate k training and validation scores for each model and hyperparameter combo
    - you can calculate their means and stds to decide which model and hyperparameter combo is the best
- **what is the disadvantage of the stratified method? Why we only applied it with imbalanced data?**
    - there is no disadvantage as far as I know
    - you can in principle apply stratified splitting in any classification problem
    - but you MUST apply it if your problem is imbalanced
- **why the cats and dogs on images are iid? Could you elaborate more about it ?**
    - iid properties:
        - independence: observing the current datapoint doesn't influence or provide insight about the next datapoint or any other datapoint in your sample
            - the next cat or dog picture you download from the net is independent from all other pics you download
        - identically distributed: all points are sampled from the same probability density distribution
            - coin toss, or the IQ of people which follows a normal distribution
            - no trends, the distribution doesn't change while you sample or collect the data
    - time series: the next datapoint is influenced by the previous data point - autocorrelation
    - group structure: each group could have a different probability density distribution
- **I am not quite 100% understanding how is the K-fold works. I only see the validation and other data in the picture, but there is no training data. So how to split the other data in the K-fold approach?**
    - in each split, the green folds are used for training and the blue fold is validation
- **i think the difference between validation and testing is still vague. I guess I want to better understand (through an example) how validation is used in tuning.**
    - I know, this material is difficult to teach and learn if you are new to ML
    - I promise that everything will fall into place by early to mid November once you see the final product, an ML pipeline

- **is it in the EDA process that we can decide which features are time-series structured and which features are group structured?**
    - usually you only need to decide whether the target variable is time-series or group-structured
- **If there are extremely rare cases, 2 out of 100000, then how should I put these cases into train/validation/test sets?**
    - if your whole dataset contains only two samples of one class, I'd argue you don't have enough data to do proper ML
    - you need to collect more data
- **After using KFolds to split your train and validation sets, how should you proceed? Should you create a list for each to store the different splits and iterate through those in the next steps of the pipeline?**
    - we iterated through the various splits in a for loop and we will just keep adding code to that for loop to cover the next steps
- **What's the difference between ¬†train_test_split(X,y,train_size = 0.6,stratify=... and StratifiedShuffleSplit()?**
    - train_test_split splits the data once
        - the split is stratified if you use the stratify argument
    - stratified shuffle split creates `n_splits` number of splits that are stratified
- **are there best practices for picking random seeds/states?**
    - nope :)
    - you can literally use whatever number you want
- **Why did you split the data in a different order when showing the code for how to complete Quiz 1?**
    - the basic split can be performed in three different ways and all three ways are equally good
    - split X to X_other, X_train, and then split X_other to X_val and X_test
    - split X to X_other, X_val, and then split X_other to X_train and X_test
    - split X to X_other, X_test, and then split X_other to X_train and X_val
- **The concept of stratifying data set!**
- **Could you please explain more about stratify=y in part two today?**
    - read more here
- **is there any special case when not using shuffle is more meaningful?**
    - not that I aware of
    - you can use shuffle in any splitting strategy
    - if your dataset is not shuffled, you MUST use shuffle
- **For train_test_split, we had to specifiy which variable to stratify on. Does StratifiedKFold just assume that the second variable you pass is the one upon which to stratify?**
    - yes, that's exactly right
    - https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
- **I am still confused about what does the Class and Group in the image. Since it is an iid datasets, why is there Class and Group?**

- class is there because we illustrate the stratification and splits on a classification example
  - you are right, group is not necessary when discussing an iid dataset
  - we cover groups today
- **Do you have any additional materials we might be able to review to complement this course?
  - the sklearn website is a great resource
  - check out the model selection section
- **What are the pros and cons of using kfold vs regular splitting for iid data?**
  - a good way to approach this is to consider the sources of uncertainty on the generalization error
  - source 1: the points in the test set
    - if you have different points in the test set, the generalization error will be different for the same model
  - source 2: the models are different
    - if the points in the test set are the same, different models (and sometimes the same model trained with a different random seed) will give a different generalization error
  - if you perform the basic split a couple of times with a couple of different seed, both sources of uncertainty will be measured in the generalization error
  - if you do a kfold split, the points in the test set will be the same so you'll only measure source 2
- **I've heard leave-one-out cross validation can lead to poor estimates of generalization error. Do you agree or is it generally better to use as many folds as your computational resources allow.**
  - leave-one-out is pretty rarely used in general
  - you'd only use that on small datasets
  - but then you should consider not using ML anyway

Data preprocessing

By the end of this lecture, you will be able to

- apply one-hot encoding or ordinal encoding to categorical variables
- apply scaling and normalization to continuous variables

The supervised ML pipeline

The goal: Use the training data (X and y) to develop a model which can accurately predict the target variable (y_new') for previously unseen data (X_new).

1. Exploratory Data Analysis (EDA): you need to understand your data and verify that it doesn't contain errors

- do as much EDA as you can!

2. Split the data into different sets: most often the sets are train, validation, and test (or holdout)

- practitioners often make errors in this step!
- you can split the data randomly, based on groups, based on time, or any other non-standard way if necessary to answer your ML question

3. Preprocess the data: ML models only work if X and Y are numbers! Some ML models additionally require each feature to have 0 mean and 1 standard deviation (standardized features)

- often the original features you get contain strings (for example a gender feature would contain 'male', 'female', 'non-binary', 'unknown') which needs to transformed into numbers
- often the features are not standardized (e.g., age is between 0 and 100) but it needs to be standardized

4. Choose an evaluation metric: depends on the priorities of the stakeholders

- often requires quite a bit of thinking and ethical considerations

5. Choose one or more ML techniques: it is highly recommended that you try multiple models

- start with simple models like linear or logistic regression
- try also more complex models like nearest neighbors, support vector machines, random forest, etc.

6. Tune the hyperparameters of your ML models (aka cross-validation)

- ML techniques have hyperparameters that you need to optimize to achieve best performance
- for each ML model, decide which parameters to tune and what values to try
- loop through each parameter combination
    - train one model for each parameter combination
    - evaluate how well the model performs on the validation set
- take the parameter combo that gives the best validation score
- evaluate that model on the test set to report how well the model is expected to perform on previously unseen data

7. Interpret your model: black boxes are often not useful

- check if your model uses features that make sense (excellent tool for debugging)
- often model predictions are not enough, you need to be able to explain how the model arrived to a particular prediction (e.g., in health care)

Problem description, why preprocessing is necessary

Data format suitable for ML: 2D numerical values.

X	feature_1	feature_2	...	feature_j	...	feature_m	y
data_point_1	x_11	x_12	...	x_1j	...	x_1m	**y_1**
data_point_2	x_21	x_22	...	x_2j	...	x_2m	**y_2**
...	...	...	...	...	...	...	**...**
data_point_i	x_i1	x_i2	...	x_ij	...	x_im	**y_i**
...	...	...	...	...	...	...	**...**
data_point_n	x_n1	x_n2	...	x_nj	...	x_nm	**y_n**

Data almost never comes in a format that's directly usable in ML.

- let's check the adult data

In [1]:

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

df = pd.read_csv('data/adult_data.csv')

# let's separate the feature matrix X, and target variable y
y = df['gross-income'] # remember, we want to predict who earns more than 50k or
X = df.loc[:, df.columns != 'gross-income'] # all other columns are features

random_state = 42

# first split to separate out the training set
X_train, X_other, y_train, y_other = train_test_split(X,y,train_size = 0.6,rando

# second split to separate out the validation and test sets
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_size = 0.5

print('training set')
print(X_train.head()) # lots of strings!
print(y_train.head()) # even our labels are strings and not numbers!
```

```
training set
       age    workclass  fnlwgt      education  education-num  \
25823   31      Private   87418      Assoc-voc             11
10274   41      Private  121718   Some-college             10
27652   61      Private   79827        HS-grad              9
13941   33    State-gov  156015      Bachelors             13
31384   38      Private  167882   Some-college             10

            marital-status         occupation     relationship    race  \
25823   Married-civ-spouse    Exec-managerial          Husband   White
10274   Married-civ-spouse       Craft-repair          Husband   White
27652   Married-civ-spouse    Exec-managerial          Husband   White
13941   Married-civ-spouse    Exec-managerial          Husband   White
31384              Widowed      Other-service   Other-relative   Black

           sex  capital-gain  capital-loss  hours-per-week  native-country
25823     Male             0             0              40   United-States
10274     Male             0             0              40           Italy
27652     Male             0             0              50   United-States
13941     Male             0             0              40   United-States
31384   Female             0             0              45           Haiti
```

```
25823     <=50K
10274     <=50K
27652     <=50K
13941      >50K
31384     <=50K
Name: gross-income, dtype: object
```

scikit-learn transformers to the rescue!

Preprocessing is done with various transformers. All transformes have three methods:

- **fit** method: estimates parameters necessary to do the transformation,
- **transform** method: transforms the data based on the estimated parameters,
- **fit_transform** method: both steps are performed at once, this can be faster than doing the steps separately.

Transformers we cover today

- **OneHotEncoder** - converts categorical features into dummy arrays
- **OrdinalEncoder** - converts categorical features into an integer array
- **MinMaxScaler** - scales continuous variables to be between 0 and 1
- **StandardScaler** - standardizes continuous features by removing the mean and scaling to unit variance

By the end of this lecture, you will be able to

- **apply one-hot encoding or ordinal encoding to categorical variables**
- apply scaling and normalization to continuous variables

Ordered categorical data: OrdinalEncoder

- use it on categorical features if the categories can be ranked or ordered
  - educational level in the adult dataset
  - reaction to medication is described by words like 'severe', 'no response', 'excellent'
  - any time you know that the categories can be clearly ranked

In [2]:

```
from sklearn.preprocessing import OrdinalEncoder
help(OrdinalEncoder)
```

```
Help on class OrdinalEncoder in module sklearn.preprocessing._encoders:

class OrdinalEncoder(_BaseEncoder)
 |  OrdinalEncoder(*, categories='auto', dtype=<class 'numpy.float64'>, handle_u
nknown='error', unknown_value=None)
 |
 |  Encode categorical features as an integer array.
 |
 |  The input to this transformer should be an array-like of integers or
 |  strings, denoting the values taken on by categorical (discrete) features.
 |  The features are converted to ordinal integers. This results in
 |  a single column of integers (0 to n_categories - 1) per feature.
 |
```

```
 |  Read more in the :ref:`User Guide <preprocessing_categorical_features>`.
 |  
 |  .. versionadded:: 0.20
 |  
 |  Parameters
 |  ----------
 |  categories : 'auto' or a list of array-like, default='auto'
 |      Categories (unique values) per feature:
 |  
 |      - 'auto' : Determine categories automatically from the training data.
 |      - list : ``categories[i]`` holds the categories expected in the ith
 |        column. The passed categories should not mix strings and numeric
 |        values, and should be sorted in case of numeric values.
 |  
 |      The used categories can be found in the ``categories_`` attribute.
 |  
 |  dtype : number type, default np.float64
 |      Desired dtype of output.
 |  
 |  handle_unknown : {'error', 'use_encoded_value'}, default='error'
 |      When set to 'error' an error will be raised in case an unknown
 |      categorical feature is present during transform. When set to
 |      'use_encoded_value', the encoded value of unknown categories will be
 |      set to the value given for the parameter `unknown_value`. In
 |      :meth:`inverse_transform`, an unknown category will be denoted as None.
 |  
 |      .. versionadded:: 0.24
 |  
 |  unknown_value : int or np.nan, default=None
 |      When the parameter handle_unknown is set to 'use_encoded_value', this
 |      parameter is required and will set the encoded value of unknown
 |      categories. It has to be distinct from the values used to encode any of
 |      the categories in `fit`. If set to np.nan, the `dtype` parameter must
 |      be a float dtype.
 |  
 |      .. versionadded:: 0.24
 |  
 |  Attributes
 |  ----------
 |  categories_ : list of arrays
 |      The categories of each feature determined during ``fit`` (in order of
 |      the features in X and corresponding with the output of ``transform``).
 |      This does not include categories that weren't seen during ``fit``.
 |  
 |  See Also
 |  --------
 |  OneHotEncoder : Performs a one-hot encoding of categorical features.
 |  LabelEncoder : Encodes target labels with values between 0 and
 |      ``n_classes-1``.
 |  
 |  Examples
 |  --------
 |  Given a dataset with two features, we let the encoder find the unique
 |  values per feature and transform the data to an ordinal encoding.
 |  
 |  >>> from sklearn.preprocessing import OrdinalEncoder
 |  >>> enc = OrdinalEncoder()
 |  >>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
 |  >>> enc.fit(X)
 |  OrdinalEncoder()
 |  >>> enc.categories_
```

```
 |  [array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
 |  >>> enc.transform([['Female', 3], ['Male', 1]])
 |  array([[0., 2.],
 |         [1., 0.]])
 |
 |  >>> enc.inverse_transform([[1, 0], [0, 1]])
 |  array([['Male', 1],
 |         ['Female', 2]], dtype=object)
 |
 |  Method resolution order:
 |      OrdinalEncoder
 |      _BaseEncoder
 |      sklearn.base.TransformerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *, categories='auto', dtype=<class 'numpy.float64'>, handle_u
nknown='error', unknown_value=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y=None)
 |      Fit the OrdinalEncoder to X.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          The data to determine the categories of each feature.
 |
 |      y : None
 |          Ignored. This parameter exists only for compatibility with
 |          :class:`~sklearn.pipeline.Pipeline`.
 |
 |      Returns
 |      -------
 |      self
 |
 |  inverse_transform(self, X)
 |      Convert the data back to the original representation.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The transformed data.
 |
 |      Returns
 |      -------
 |      X_tr : ndarray of shape (n_samples, n_features)
 |          Inverse transformed array.
 |
 |  transform(self, X)
 |      Transform X to ordinal codes.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          The data to encode.
 |
 |      Returns
 |      -------
```

```
 |      X_out : ndarray of shape (n_samples, n_features)
 |          Transformed input.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.TransformerMixin:
 |
 |  fit_transform(self, X, y=None, **fit_params)
 |      Fit to data, then transform it.
 |
 |      Fits transformer to `X` and `y` with optional parameters `fit_params`
 |      and returns a transformed version of `X`.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Input samples.
 |
 |      y :  array-like of shape (n_samples,) or (n_samples, n_outputs),
default=None
 |          Target values (None for unsupervised transformations).
 |
 |      **fit_params : dict
 |          Additional fit parameters.
 |
 |      Returns
 |      -------
 |      X_new : ndarray array of shape (n_samples, n_features_new)
 |          Transformed array.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.TransformerMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      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, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that it's
 |      possible to update each component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : estimator instance
 |          Estimator instance.
```

In [3]:

```
# toy example
import pandas as pd

train_edu = {'educational level':['Bachelors','Masters','Bachelors','Doctorate',
test_edu = {'educational level':['HS-grad','Masters','Masters','College','Bachel

Xtoy_train = pd.DataFrame(train_edu)
Xtoy_test = pd.DataFrame(test_edu)

# initialize the encoder
cats = [['HS-grad','College','Bachelors','Masters','Doctorate']]

enc = OrdinalEncoder(categories = cats) # The ordered list of
# categories need to be provided. By default, the categories are alphabetically

# fit the training data
enc.fit(Xtoy_train)
# print the categories - not really important because we manually gave the order
print(enc.categories_)
# transform X_train. We could have used enc.fit_transform(X_train) to combine fi
X_train_oe = enc.transform(Xtoy_train)
print(X_train_oe)
# transform X_test
X_test_oe = enc.transform(Xtoy_test) # OrdinalEncoder always throws an error mes
                                  # it encounters an unknown category in test
print(X_test_oe)
```

```
[array(['HS-grad', 'College', 'Bachelors', 'Masters', 'Doctorate'],
      dtype=object)]
[[2.]
 [3.]
 [2.]
 [4.]
 [0.]
 [3.]]
[[0.]
 [3.]
 [3.]
 [1.]
 [2.]]
```

In [4]:
```python
# apply OE to the adult dataset
# initialize the encoder
ordinal_ftrs = ['education'] # if you have more than one ordinal feature, add th
ordinal_cats = [[' Preschool',' 1st-4th',' 5th-6th',' 7th-8th',' 9th',' 10th','
                ' Some-college',' Assoc-voc',' Assoc-acdm',' Bachelors',' Master
# ordinal_cats must contain one list per ordinal feature! each list contains the
# of the corresponding feature

enc = OrdinalEncoder(categories = ordinal_cats)   # By default, the categories a
                                                   # which is NOT what you want

# fit the training data
enc.fit(X_train[ordinal_ftrs])  # the encoder expects a 2D array, that's why the

# transform X_train. We could use enc.fit_transform(X_train) to combine fit and
ordinal_train = enc.transform(X_train[ordinal_ftrs])
print('transformed train features:')
print(ordinal_train)
# transform X_val
ordinal_val = enc.transform(X_val[ordinal_ftrs])
print('transformed validation features:')
print(ordinal_val)
# transform X_test
ordinal_test = enc.transform(X_test[ordinal_ftrs])
print('transformed test features:')
print(ordinal_test)
```

```
transformed train features:
[[10.]
 [ 9.]
 [ 8.]
 ...
 [ 6.]
 [ 8.]
 [12.]]
transformed validation features:
[[14.]
 [13.]
 [ 9.]
 ...
 [12.]
 [ 8.]
 [ 8.]]
transformed test features:
[[12.]
 [ 9.]
 [12.]
 ...
 [ 9.]
 [ 9.]
 [11.]]
```

Unordered categorical data: one-hot encoder

- some categories cannot be ordered. e.g., workclass, relationship status

In [5]:
```python
from sklearn.preprocessing import OneHotEncoder
help(OneHotEncoder)
```

```
Help on class OneHotEncoder in module sklearn.preprocessing._encoders:

class OneHotEncoder(_BaseEncoder)
 |  OneHotEncoder(*, categories='auto', drop=None, sparse=True, dtype=<class 'nu
mpy.float64'>, handle_unknown='error')
 |
 |  Encode categorical features as a one-hot numeric array.
 |
 |  The input to this transformer should be an array-like of integers or
 |  strings, denoting the values taken on by categorical (discrete) features.
 |  The features are encoded using a one-hot (aka 'one-of-K' or 'dummy')
 |  encoding scheme. This creates a binary column for each category and
 |  returns a sparse matrix or dense array (depending on the ``sparse``
 |  parameter)
 |
 |  By default, the encoder derives the categories based on the unique values
 |  in each feature. Alternatively, you can also specify the `categories`
 |  manually.
 |
 |  This encoding is needed for feeding categorical data to many scikit-learn
 |  estimators, notably linear models and SVMs with the standard kernels.
 |
 |  Note: a one-hot encoding of y labels should use a LabelBinarizer
 |  instead.
 |
 |  Read more in the :ref:`User Guide <preprocessing_categorical_features>`.
 |
 |  Parameters
 |  ----------
 |  categories : 'auto' or a list of array-like, default='auto'
 |      Categories (unique values) per feature:
 |
 |      - 'auto' : Determine categories automatically from the training data.
 |      - list : ``categories[i]`` holds the categories expected in the ith
 |        column. The passed categories should not mix strings and numeric
 |        values within a single feature, and should be sorted in case of
 |        numeric values.
 |
 |      The used categories can be found in the ``categories_`` attribute.
 |
 |      .. versionadded:: 0.20
 |
 |  drop : {'first', 'if_binary'} or a array-like of shape (n_features,),
default=None
 |      Specifies a methodology to use to drop one of the categories per
 |      feature. This is useful in situations where perfectly collinear
 |      features cause problems, such as when feeding the resulting data
 |      into a neural network or an unregularized regression.
 |
 |      However, dropping one category breaks the symmetry of the original
 |      representation and can therefore induce a bias in downstream models,
 |      for instance for penalized linear classification or regression models.
 |
 |      - None : retain all features (the default).
 |      - 'first' : drop the first category in each feature. If only one
 |        category is present, the feature will be dropped entirely.
 |      - 'if_binary' : drop the first category in each feature with two
 |        categories. Features with 1 or more than 2 categories are
 |        left intact.
 |      - array : ``drop[i]`` is the category in feature ``X[:, i]`` that
```

```
|       should be dropped.
|
|     .. versionadded:: 0.21
|        The parameter `drop` was added in 0.21.
|
|     .. versionchanged:: 0.23
|        The option `drop='if_binary'` was added in 0.23.
|
| sparse : bool, default=True
|     Will return sparse matrix if set True else will return an array.
|
| dtype : number type, default=float
|     Desired dtype of output.
|
| handle_unknown : {'error', 'ignore'}, default='error'
|     Whether to raise an error or ignore if an unknown categorical feature
|     is present during transform (default is to raise). When this parameter
|     is set to 'ignore' and an unknown category is encountered during
|     transform, the resulting one-hot encoded columns for this feature
|     will be all zeros. In the inverse transform, an unknown category
|     will be denoted as None.
|
| Attributes
| ----------
| categories_ : list of arrays
|     The categories of each feature determined during fitting
|     (in order of the features in X and corresponding with the output
|     of ``transform``). This includes the category specified in ``drop``
|     (if any).
|
| drop_idx_ : array of shape (n_features,)
|     - ``drop_idx_[i]`` is the index in ``categories_[i]`` of the category
|       to be dropped for each feature.
|     - ``drop_idx_[i] = None`` if no category is to be dropped from the
|       feature with index ``i``, e.g. when `drop='if_binary'` and the
|       feature isn't binary.
|     - ``drop_idx_ = None`` if all the transformed features will be
|       retained.
|
|     .. versionchanged:: 0.23
|        Added the possibility to contain `None` values.
|
| See Also
| --------
| OrdinalEncoder : Performs an ordinal (integer)
|   encoding of the categorical features.
| sklearn.feature_extraction.DictVectorizer : Performs a one-hot encoding of
|   dictionary items (also handles string-valued features).
| sklearn.feature_extraction.FeatureHasher : Performs an approximate one-hot
|   encoding of dictionary items or strings.
| LabelBinarizer : Binarizes labels in a one-vs-all
|   fashion.
| MultiLabelBinarizer : Transforms between iterable of
|   iterables and a multilabel format, e.g. a (samples x classes) binary
|   matrix indicating the presence of a class label.
|
| Examples
| --------
| Given a dataset with two features, we let the encoder find the unique
| values per feature and transform the data to a binary one-hot encoding.
|
```

```
 |  >>> from sklearn.preprocessing import OneHotEncoder
 |  
 |  One can discard categories not seen during `fit`:
 |  
 |  >>> enc = OneHotEncoder(handle_unknown='ignore')
 |  >>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
 |  >>> enc.fit(X)
 |  OneHotEncoder(handle_unknown='ignore')
 |  >>> enc.categories_
 |  [array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
 |  >>> enc.transform([['Female', 1], ['Male', 4]]).toarray()
 |  array([[1., 0., 1., 0., 0.],
 |         [0., 1., 0., 0., 0.]])
 |  >>> enc.inverse_transform([[0, 1, 1, 0, 0], [0, 0, 0, 1, 0]])
 |  array([['Male', 1],
 |         [None, 2]], dtype=object)
 |  >>> enc.get_feature_names(['gender', 'group'])
 |  array(['gender_Female', 'gender_Male', 'group_1', 'group_2', 'group_3'],
 |    dtype=object)
 |  
 |  One can always drop the first column for each feature:
 |  
 |  >>> drop_enc = OneHotEncoder(drop='first').fit(X)
 |  >>> drop_enc.categories_
 |  [array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
 |  >>> drop_enc.transform([['Female', 1], ['Male', 2]]).toarray()
 |  array([[0., 0., 0.],
 |         [1., 1., 0.]])
 |  
 |  Or drop a column for feature only having 2 categories:
 |  
 |  >>> drop_binary_enc = OneHotEncoder(drop='if_binary').fit(X)
 |  >>> drop_binary_enc.transform([['Female', 1], ['Male', 2]]).toarray()
 |  array([[0., 1., 0., 0.],
 |         [1., 0., 1., 0.]])
 |  
 |  Method resolution order:
 |      OneHotEncoder
 |      _BaseEncoder
 |      sklearn.base.TransformerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __init__(self, *, categories='auto', drop=None, sparse=True, dtype=<class 'n
umpy.float64'>, handle_unknown='error')
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  fit(self, X, y=None)
 |      Fit OneHotEncoder to X.
 |      
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          The data to determine the categories of each feature.
 |      
 |      y : None
 |          Ignored. This parameter exists only for compatibility with
 |          :class:`~sklearn.pipeline.Pipeline`.
 |  
```

```
 |      Returns
 |      -------
 |      self
 |
 |  fit_transform(self, X, y=None)
 |      Fit OneHotEncoder to X, then transform X.
 |
 |      Equivalent to fit(X).transform(X) but more convenient.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          The data to encode.
 |
 |      y : None
 |          Ignored. This parameter exists only for compatibility with
 |          :class:`~sklearn.pipeline.Pipeline`.
 |
 |      Returns
 |      -------
 |      X_out : {ndarray, sparse matrix} of shape                 (n_samples, n_
encoded_features)
 |          Transformed input. If `sparse=True`, a sparse matrix will be
 |          returned.
 |
 |  get_feature_names(self, input_features=None)
 |      Return feature names for output features.
 |
 |      Parameters
 |      ----------
 |      input_features : list of str of shape (n_features,)
 |          String names for input features if available. By default,
 |          "x0", "x1", ... "xn_features" is used.
 |
 |      Returns
 |      -------
 |      output_feature_names : ndarray of shape (n_output_features,)
 |          Array of feature names.
 |
 |  inverse_transform(self, X)
 |      Convert the data back to the original representation.
 |
 |      In case unknown categories are encountered (all zeros in the
 |      one-hot encoding), ``None`` is used to represent this category.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape                 (n_samples, n_e
ncoded_features)
 |          The transformed data.
 |
 |      Returns
 |      -------
 |      X_tr : ndarray of shape (n_samples, n_features)
 |          Inverse transformed array.
 |
 |  transform(self, X)
 |      Transform X using one-hot encoding.
 |
 |      Parameters
 |      ----------
```

```
 |      X : array-like of shape (n_samples, n_features)
 |          The data to encode.
 |
 |      Returns
 |      -------
 |      X_out : {ndarray, sparse matrix} of shape                 (n_samples, n_
encoded_features)
 |          Transformed input. If `sparse=True`, a sparse matrix will be
 |          returned.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.TransformerMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      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, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that it's
 |      possible to update each component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : estimator instance
 |          Estimator instance.
```

In [22]:

```
# toy example
train = {'gender':['Male','Female','Unknown','Male','Female','Female'],\
         'browser':['Safari','Safari','Internet Explorer','Chrome','Chrome','Int
test = {'gender':['Female','Male','Unknown','Female'],'browser':['Chrome','Firef

Xtoy_train = pd.DataFrame(train)
Xtoy_test = pd.DataFrame(test)

ftrs = ['gender','browser']

# initialize the encoder
enc = OneHotEncoder(sparse=False,handle_unknown='ignore') # by default, OneHotEn
# fit the training data
enc.fit(Xtoy_train)
print('categories:',enc.categories_)
print('feature names:',enc.get_feature_names(ftrs))
# transform X_train
X_train_ohe = enc.transform(Xtoy_train)
#print(X_train_ohe)
# do all of this in one step
X_train_ohe = enc.fit_transform(Xtoy_train)
print('X_train transformed')
print(X_train_ohe)

# transform X_test
X_test_ohe = enc.transform(Xtoy_test)
print('X_test transformed')
print(X_test_ohe)
```

```
categories: [array(['Female', 'Male', 'Unknown'], dtype=object), array(['Chrom
e', 'Internet Explorer', 'Safari'], dtype=object)]
feature names: ['gender_Female' 'gender_Male' 'gender_Unknown' 'browser_Chrome'
 'browser_Internet Explorer' 'browser_Safari']
X_train transformed
[[0. 1. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 1. 0.]
 [0. 1. 0. 1. 0. 0.]
 [1. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 1. 0.]]
X_test transformed
[[1. 0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 1. 0.]
 [1. 0. 0. 0. 0. 1.]]
```

In [23]:

```
# apply OHE to the adult dataset

# let's collect all categorical features first
onehot_ftrs = ['workclass','marital-status','occupation','relationship','race','
# initialize the encoder
enc = OneHotEncoder(sparse=False,handle_unknown='ignore') # by default, OneHotEn
# fit the training data
enc.fit(X_train[onehot_ftrs])
print('feature names:',enc.get_feature_names(onehot_ftrs))
print(len(enc.get_feature_names(onehot_ftrs)))
```

```
feature names: ['workclass_ ?' 'workclass_ Federal-gov' 'workclass_ Local-gov'
 'workclass_ Never-worked' 'workclass_ Private' 'workclass_ Self-emp-inc'
```

```
 'workclass_ Self-emp-not-inc' 'workclass_ State-gov'
 'workclass_ Without-pay' 'marital-status_ Divorced'
 'marital-status_ Married-AF-spouse' 'marital-status_ Married-civ-spouse'
 'marital-status_ Married-spouse-absent' 'marital-status_ Never-married'
 'marital-status_ Separated' 'marital-status_ Widowed' 'occupation_ ?'
 'occupation_ Adm-clerical' 'occupation_ Armed-Forces'
 'occupation_ Craft-repair' 'occupation_ Exec-managerial'
 'occupation_ Farming-fishing' 'occupation_ Handlers-cleaners'
 'occupation_ Machine-op-inspct' 'occupation_ Other-service'
 'occupation_ Priv-house-serv' 'occupation_ Prof-specialty'
 'occupation_ Protective-serv' 'occupation_ Sales'
 'occupation_ Tech-support' 'occupation_ Transport-moving'
 'relationship_ Husband' 'relationship_ Not-in-family'
 'relationship_ Other-relative' 'relationship_ Own-child'
 'relationship_ Unmarried' 'relationship_ Wife' 'race_ Amer-Indian-Eskimo'
 'race_ Asian-Pac-Islander' 'race_ Black' 'race_ Other' 'race_ White'
 'sex_ Female' 'sex_ Male' 'native-country_ ?' 'native-country_ Cambodia'
 'native-country_ Canada' 'native-country_ China'
 'native-country_ Columbia' 'native-country_ Cuba'
 'native-country_ Dominican-Republic' 'native-country_ Ecuador'
 'native-country_ El-Salvador' 'native-country_ England'
 'native-country_ France' 'native-country_ Germany'
 'native-country_ Greece' 'native-country_ Guatemala'
 'native-country_ Haiti' 'native-country_ Holand-Netherlands'
 'native-country_ Honduras' 'native-country_ Hong'
 'native-country_ Hungary' 'native-country_ India' 'native-country_ Iran'
 'native-country_ Ireland' 'native-country_ Italy'
 'native-country_ Jamaica' 'native-country_ Japan' 'native-country_ Laos'
 'native-country_ Mexico' 'native-country_ Nicaragua'
 'native-country_ Outlying-US(Guam-USVI-etc)' 'native-country_ Peru'
 'native-country_ Philippines' 'native-country_ Poland'
 'native-country_ Portugal' 'native-country_ Puerto-Rico'
 'native-country_ Scotland' 'native-country_ South'
 'native-country_ Taiwan' 'native-country_ Thailand'
 'native-country_ Trinadad&Tobago' 'native-country_ United-States'
 'native-country_ Vietnam' 'native-country_ Yugoslavia']
86
```

In [8]:

```python
# transform X_train
onehot_train = enc.transform(X_train[onehot_ftrs])
print('transformed train features:')
print(onehot_train)
# transform X_val
onehot_val = enc.transform(X_val[onehot_ftrs])
print('transformed val features:')
print(onehot_val)
# transform X_test
onehot_test = enc.transform(X_test[onehot_ftrs])
print('transformed test features:')
print(onehot_test)
```

```
transformed train features:
[[0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]
 ...
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]]
transformed val features:
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]]
transformed test features:
[[0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]]
```

Quiz 1

Which of the following categorical features should be preprocessed with the Ordinal Encoder?

- marital status (Married, Divorced, Never-married, Separated, Widowed)
- exterior quality of a house (Excellent, Good, Average/Typical, Fair, Poor)
- native country (USA, Hungary, China, India, Germany)

By the end of this lecture, you will be able to

- apply one-hot encoding or ordinal encoding to categorical variables
- **apply scaling and normalization to continuous variables**

Continuous features: MinMaxScaler

- If the continuous feature values are reasonably bounded, MinMaxScaler is a good way to scale the features.
- Age is expected to be within the range of 0 and 100.
- Number of hours worked per week is in the range of 0 to 80.
- If unsure, plot the histogram of the feature to verify or just go with the standard scaler!

In [9]:

```python
from sklearn.preprocessing import MinMaxScaler
help(MinMaxScaler)
```

```
Help on class MinMaxScaler in module sklearn.preprocessing._data:

class MinMaxScaler(sklearn.base.TransformerMixin, sklearn.base.BaseEstimator)
 |  MinMaxScaler(feature_range=(0, 1), *, copy=True, clip=False)
 |
 |  Transform features by scaling each feature to a given range.
 |
 |  This estimator scales and translates each feature individually such
 |  that it is in the given range on the training set, e.g. between
 |  zero and one.
 |
 |  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.
|
| This transformation is often used as an alternative to zero mean,
| unit variance scaling.
|
| Read more in the :ref:`User Guide <preprocessing_scaler>`.
|
| Parameters
| ----------
| 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`.
|
|     .. versionadded:: 0.24
|
| Attributes
| ----------
| min_ : ndarray of shape (n_features,)
|     Per feature adjustment for minimum. Equivalent to
|     ``min - X.min(axis=0) * self.scale_``
|
| scale_ : ndarray of shape (n_features,)
|     Per feature relative scaling of the data. Equivalent to
|     ``(max - min) / (X.max(axis=0) - X.min(axis=0))``
|
|     .. versionadded:: 0.17
|        *scale_* attribute.
|
| data_min_ : ndarray of shape (n_features,)
|     Per feature minimum seen in the data
|
|     .. versionadded:: 0.17
|        *data_min_*
|
| data_max_ : ndarray of shape (n_features,)
|     Per feature maximum seen in the data
|
|     .. versionadded:: 0.17
|        *data_max_*
|
| data_range_ : ndarray of shape (n_features,)
|     Per feature range ``(data_max_ - data_min_)`` seen in the data
|
|     .. versionadded:: 0.17
|        *data_range_*
|
| n_samples_seen_ : int
|     The number of samples processed by the estimator.
|     It will be reset on new calls to fit, but increments across
|     ``partial_fit`` calls.
|
```

```
 |  Examples
 |  --------
 |  >>> from sklearn.preprocessing import MinMaxScaler
 |  >>> data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
 |  >>> scaler = MinMaxScaler()
 |  >>> print(scaler.fit(data))
 |  MinMaxScaler()
 |  >>> print(scaler.data_max_)
 |  [ 1. 18.]
 |  >>> print(scaler.transform(data))
 |  [[0.   0.  ]
 |   [0.25 0.25]
 |   [0.5  0.5 ]
 |   [1.   1.  ]]
 |  >>> print(scaler.transform([[2, 2]]))
 |  [[1.5 0. ]]
 |
 |  See Also
 |  --------
 |  minmax_scale : Equivalent function without the estimator API.
 |
 |  Notes
 |  -----
 |  NaNs are treated as missing values: disregarded in fit, and maintained in
 |  transform.
 |
 |  For a comparison of the different scalers, transformers, and normalizers,
 |  see :ref:`examples/preprocessing/plot_all_scaling.py
 |  <sphx_glr_auto_examples_preprocessing_plot_all_scaling.py>`.
 |
 |  Method resolution order:
 |      MinMaxScaler
 |      sklearn.base.TransformerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, feature_range=(0, 1), *, copy=True, clip=False)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y=None)
 |      Compute the minimum and maximum to be used for later scaling.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          The data used to compute the per-feature minimum and maximum
 |          used for later scaling along the features axis.
 |
 |      y : None
 |          Ignored.
 |
 |      Returns
 |      -------
 |      self : object
 |          Fitted scaler.
 |
 |  inverse_transform(self, X)
 |      Undo the scaling of X according to feature_range.
 |
```

```
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Input data that will be transformed. It cannot be sparse.
 |
 |      Returns
 |      -------
 |      Xt : ndarray of shape (n_samples, n_features)
 |          Transformed data.
 |
 |  partial_fit(self, X, y=None)
 |      Online computation of min and max on X for later scaling.
 |
 |      All of X is processed as a single batch. This is intended for cases
 |      when :meth:`fit` is not feasible due to very large number of
 |      `n_samples` or because X is read from a continuous stream.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          The data used to compute the mean and standard deviation
 |          used for later scaling along the features axis.
 |
 |      y : None
 |          Ignored.
 |
 |      Returns
 |      -------
 |      self : object
 |          Fitted scaler.
 |
 |  transform(self, X)
 |      Scale features of X according to feature_range.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Input data that will be transformed.
 |
 |      Returns
 |      -------
 |      Xt : ndarray of shape (n_samples, n_features)
 |          Transformed data.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.TransformerMixin:
 |
 |  fit_transform(self, X, y=None, **fit_params)
 |      Fit to data, then transform it.
 |
 |      Fits transformer to `X` and `y` with optional parameters `fit_params`
 |      and returns a transformed version of `X`.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Input samples.
 |
 |      y :  array-like of shape (n_samples,) or (n_samples, n_outputs),
default=None
 |          Target values (None for unsupervised transformations).
```

```
 |  
 |      **fit_params : dict
 |          Additional fit parameters.
 |  
 |      Returns
 |      -------
 |      X_new : ndarray array of shape (n_samples, n_features_new)
 |          Transformed array.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.TransformerMixin:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |  
 |  __getstate__(self)
 |  
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |  
 |  __setstate__(self, state)
 |  
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |  
 |      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, **params)
 |      Set the parameters of this estimator.
 |  
 |      The method works on simple estimators as well as on nested objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that it's
 |      possible to update each component of a nested object.
 |  
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |  
 |      Returns
 |      -------
 |      self : estimator instance
 |          Estimator instance.
```

In [10]:

```
# toy data
# let's assume we have two continuous features:
train = {'age':[32,65,13,68,42,75,32],'number of hours worked':[0,40,10,60,40,20
test = {'age':[83,26,10,60],'number of hours worked':[0,40,0,60]}

# (value - min) / (max - min), if value is 32, min is 13 and max is 75, then we

Xtoy_train = pd.DataFrame(train)
Xtoy_test = pd.DataFrame(test)

scaler = MinMaxScaler()
scaler.fit(Xtoy_train)
print(scaler.transform(Xtoy_train))
print(scaler.transform(Xtoy_test)) # note how scaled X_test contains values larg
```

```
[[0.30645161 0.        ]
 [0.83870968 0.66666667]
 [0.         0.16666667]
 [0.88709677 1.        ]
 [0.46774194 0.66666667]
 [1.         0.33333333]
 [0.30645161 0.66666667]]
[[ 1.12903226  0.        ]
 [ 0.20967742  0.66666667]
 [-0.0483871   0.        ]
 [ 0.75806452  1.        ]]
```

In [11]:

```
# adult data

minmax_ftrs = ['age','hours-per-week']

scaler = MinMaxScaler()
scaler.fit(X_train[minmax_ftrs])
print(scaler.transform(X_train[minmax_ftrs]))
print(scaler.transform(X_val[minmax_ftrs]))
print(scaler.transform(X_test[minmax_ftrs]))
```

```
[[0.19178082 0.39795918]
 [0.32876712 0.39795918]
 [0.60273973 0.5       ]
 ...
 [0.01369863 0.19387755]
 [0.45205479 0.84693878]
 [0.23287671 0.60204082]]
[[0.35616438 0.5       ]
 [0.68493151 0.39795918]
 [0.09589041 0.39795918]
 ...
 [0.09589041 0.19387755]
 [0.02739726 0.44897959]
 [0.38356164 0.39795918]]
[[0.06849315 0.39795918]
 [0.23287671 0.39795918]
 [0.43835616 0.5       ]
 ...
 [0.20547945 0.39795918]
 [0.21917808 0.37755102]
 [0.08219178 0.35714286]]
```

Continuous features: StandardScaler

- If the continuous feature values follow a tailed distribution, StandardScaler is better to use!
- Salaries are a good example. Most people earn less than 100k but there are a small number of super-rich people.

In [12]:

```
from sklearn.preprocessing import StandardScaler
help(StandardScaler)
```

```
Help on class StandardScaler in module sklearn.preprocessing._data:

class StandardScaler(sklearn.base.TransformerMixin, sklearn.base.BaseEstimator)
 |  StandardScaler(*, copy=True, with_mean=True, with_std=True)
 |
 |  Standardize features by removing the mean and scaling to unit variance
 |
 |  The standard score of a sample `x` is calculated as:
 |
 |      z = (x - u) / s
 |
 |  where `u` is the mean of the training samples or zero if `with_mean=False`,
 |  and `s` is the standard deviation of the training samples or one if
 |  `with_std=False`.
 |
 |  Centering and scaling happen independently on each feature by computing
 |  the relevant statistics on the samples in the training set. Mean and
 |  standard deviation are then stored to be used on later data using
 |  :meth:`transform`.
 |
 |  Standardization of a dataset is a common requirement for many
 |  machine learning estimators: they might behave badly if the
 |  individual features do not more or less look like standard normally
 |  distributed data (e.g. Gaussian with 0 mean and unit variance).
 |
 |  For instance many elements used in the objective function of
 |  a learning algorithm (such as the RBF kernel of Support Vector
 |  Machines or the L1 and L2 regularizers of linear models) assume that
 |  all features are centered around 0 and have variance in the same
 |  order. If a feature has a variance that is orders of magnitude larger
 |  that others, it might dominate the objective function and make the
 |  estimator unable to learn from other features correctly as expected.
 |
 |  This scaler can also be applied to sparse CSR or CSC matrices by passing
 |  `with_mean=False` to avoid breaking the sparsity structure of the data.
 |
 |  Read more in the :ref:`User Guide <preprocessing_scaler>`.
 |
 |  Parameters
 |  ----------
 |  copy : bool, default=True
 |      If False, try to avoid a copy and do inplace scaling instead.
 |      This is not guaranteed to always work inplace; e.g. if the data is
 |      not a NumPy array or scipy.sparse CSR matrix, a copy may still be
 |      returned.
 |
 |  with_mean : bool, default=True
 |      If True, center the data before scaling.
 |      This does not work (and will raise an exception) when attempted on
```

```
 |          sparse matrices, because centering them entails building a dense
 |          matrix which in common use cases is likely to be too large to fit in
 |          memory.
 |
 |      with_std : bool, default=True
 |          If True, scale the data to unit variance (or equivalently,
 |          unit standard deviation).
 |
 |      Attributes
 |      ----------
 |      scale_ : ndarray of shape (n_features,) or None
 |          Per feature relative scaling of the data to achieve zero mean and unit
 |          variance. Generally this is calculated using `np.sqrt(var_)`. If a
 |          variance is zero, we can't achieve unit variance, and the data is left
 |          as-is, giving a scaling factor of 1. `scale_` is equal to `None`
 |          when `with_std=False`.
 |
 |          .. versionadded:: 0.17
 |             *scale_*
 |
 |      mean_ : ndarray of shape (n_features,) or None
 |          The mean value for each feature in the training set.
 |          Equal to ``None`` when ``with_mean=False``.
 |
 |      var_ : ndarray of shape (n_features,) or None
 |          The variance for each feature in the training set. Used to compute
 |          `scale_`. Equal to ``None`` when ``with_std=False``.
 |
 |      n_samples_seen_ : int or ndarray of shape (n_features,)
 |          The number of samples processed by the estimator for each feature.
 |          If there are no missing samples, the ``n_samples_seen`` will be an
 |          integer, otherwise it will be an array of dtype int. If
 |          `sample_weights` are used it will be a float (if no missing data)
 |          or an array of dtype float that sums the weights seen so far.
 |          Will be reset on new calls to fit, but increments across
 |          ``partial_fit`` calls.
 |
 |      Examples
 |      --------
 |      >>> from sklearn.preprocessing import StandardScaler
 |      >>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
 |      >>> scaler = StandardScaler()
 |      >>> print(scaler.fit(data))
 |      StandardScaler()
 |      >>> print(scaler.mean_)
 |      [0.5 0.5]
 |      >>> print(scaler.transform(data))
 |      [[-1. -1.]
 |       [-1. -1.]
 |       [ 1.  1.]
 |       [ 1.  1.]]
 |      >>> print(scaler.transform([[2, 2]]))
 |      [[3. 3.]]
 |
 |      See Also
 |      --------
 |      scale : Equivalent function without the estimator API.
 |
 |      :class:`~sklearn.decomposition.PCA` : Further removes the linear
 |          correlation across features with 'whiten=True'.
 |
```

```
 |  Notes
 |  -----
 |  NaNs are treated as missing values: disregarded in fit, and maintained in
 |  transform.
 |
 |  We use a biased estimator for the standard deviation, equivalent to
 |  `numpy.std(x, ddof=0)`. Note that the choice of `ddof` is unlikely to
 |  affect model performance.
 |
 |  For a comparison of the different scalers, transformers, and normalizers,
 |  see :ref:`examples/preprocessing/plot_all_scaling.py
 |  <sphx_glr_auto_examples_preprocessing_plot_all_scaling.py>`.
 |
 |  Method resolution order:
 |      StandardScaler
 |      sklearn.base.TransformerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *, copy=True, with_mean=True, with_std=True)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y=None, sample_weight=None)
 |      Compute the mean and std to be used for later scaling.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The data used to compute the mean and standard deviation
 |          used for later scaling along the features axis.
 |
 |      y : None
 |          Ignored.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Individual weights for each sample.
 |
 |          .. versionadded:: 0.24
 |             parameter *sample_weight* support to StandardScaler.
 |
 |      Returns
 |      -------
 |      self : object
 |          Fitted scaler.
 |
 |  inverse_transform(self, X, copy=None)
 |      Scale back the data to the original representation
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The data used to scale along the features axis.
 |      copy : bool, default=None
 |          Copy the input X or not.
 |
 |      Returns
 |      -------
 |      X_tr : {ndarray, sparse matrix} of shape (n_samples, n_features)
 |          Transformed array.
```

```
 |  
 |  partial_fit(self, X, y=None, sample_weight=None)
 |      Online computation of mean and std on X for later scaling.
 |      
 |      All of X is processed as a single batch. This is intended for cases
 |      when :meth:`fit` is not feasible due to very large number of
 |      `n_samples` or because X is read from a continuous stream.
 |      
 |      The algorithm for incremental mean and std is given in Equation 1.5a,b
 |      in Chan, Tony F., Gene H. Golub, and Randall J. LeVeque. "Algorithms
 |      for computing the sample variance: Analysis and recommendations."
 |      The American Statistician 37.3 (1983): 242-247:
 |      
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The data used to compute the mean and standard deviation
 |          used for later scaling along the features axis.
 |      
 |      y : None
 |          Ignored.
 |      
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Individual weights for each sample.
 |      
 |          .. versionadded:: 0.24
 |             parameter *sample_weight* support to StandardScaler.
 |      
 |      Returns
 |      -------
 |      self : object
 |          Fitted scaler.
 |  
 |  transform(self, X, copy=None)
 |      Perform standardization by centering and scaling
 |      
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix of shape (n_samples, n_features)
 |          The data used to scale along the features axis.
 |      copy : bool, default=None
 |          Copy the input X or not.
 |      
 |      Returns
 |      -------
 |      X_tr : {ndarray, sparse matrix} of shape (n_samples, n_features)
 |          Transformed array.
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.TransformerMixin:
 |  
 |  fit_transform(self, X, y=None, **fit_params)
 |      Fit to data, then transform it.
 |      
 |      Fits transformer to `X` and `y` with optional parameters `fit_params`
 |      and returns a transformed version of `X`.
 |      
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Input samples.
```

```
 |
 |      y :  array-like of shape (n_samples,) or (n_samples, n_outputs),
default=None
 |          Target values (None for unsupervised transformations).
 |
 |      **fit_params : dict
 |          Additional fit parameters.
 |
 |      Returns
 |      -------
 |      X_new : ndarray array of shape (n_samples, n_features_new)
 |          Transformed array.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.TransformerMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      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, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that it's
 |      possible to update each component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
```

```
 |      self : estimator instance
 |          Estimator instance.
```

In [13]:

```python
# toy data
train = {'salary':[50_000,75_000,40_000,1_000_000,30_000,250_000,35_000,45_000]}
test = {'salary':[25_000,55_000,1_500_000,60_000]}

Xtoy_train = pd.DataFrame(train)
Xtoy_test = pd.DataFrame(test)

scaler = StandardScaler()
print(scaler.fit_transform(Xtoy_train))
print(scaler.transform(Xtoy_test))
```

```
[[-0.44873188]
 [-0.36895732]
 [-0.4806417 ]
 [ 2.58270127]
 [-0.51255153]
 [ 0.18946457]
 [-0.49659661]
 [-0.46468679]]
[[-0.52850644]
 [-0.43277697]
 [ 4.1781924 ]
 [-0.41682206]]
```

In [14]:

```python
# adult data

std_ftrs = ['capital-gain','capital-loss']
scaler = StandardScaler()
print(scaler.fit_transform(X_train[std_ftrs]))
print(scaler.transform(X_val[std_ftrs]))
print(scaler.transform(X_test[std_ftrs]))
```

```
[[-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 ...
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]]
[[-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 ...
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]]
[[-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 ...
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]
 [-0.14633293 -0.22318878]]
```

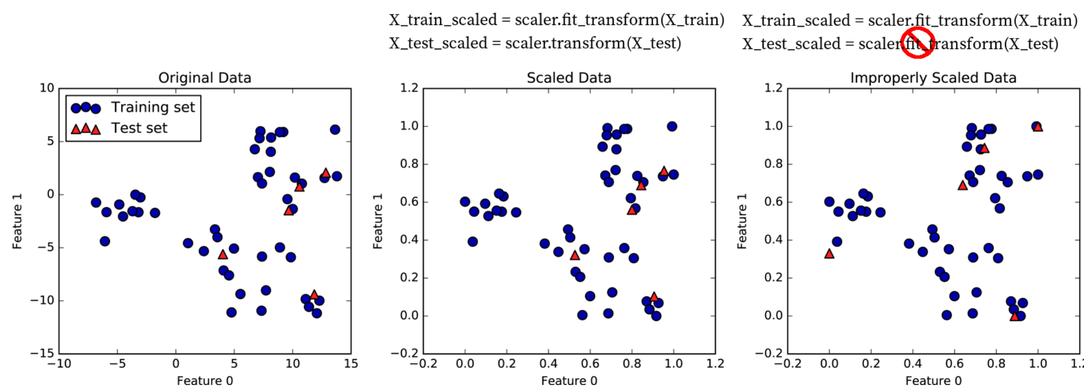
Lecture 7, Quiz 1 on Canvas

Which of these features could be safely preprocessed by the minmax scaler?

- number of minutes spent on the website in a day
- number of days a year spent abroad in a year
- USD donated to charity

How and when to do preprocessing in the ML pipeline?

- **APPLY TRANSFORMER.FIT ONLY ON YOUR TRAINING DATA!** Then transform the validation and test sets.
- One of the most common mistake practitioners make is leaking statistics!
    - fit_transform is applied to the whole dataset, then the data is split into train/validation/test
        - this is wrong because the test set statistics impacts how the training and validation sets are transformed
        - but the test set must be separated by train and val, and val must be separated by train
    - or fit_transform is applied to the train, then fit_transform is applied to the validation set, and fit_transform is applied to the test set
        - this is wrong because the relative position of the points change



Scikit-learn's pipelines

- The steps in the ML pipleine can be chained together into a scikit-learn pipeline which consists of transformers and one final estimator which is usually your classifier or regression model.
- It neatly combines the preprocessing steps and it helps to avoid leaking statistics.

https://scikit-learn.org/stable/auto_examples/compose/plot_column_transformer_mixed_types.html

In [15]:

```
import pandas as pd
import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncoder,
```

```python
from sklearn.model_selection import train_test_split

np.random.seed(0)

df = pd.read_csv('data/adult_data.csv')

# let's separate the feature matrix X, and target variable y
y = df['gross-income'] # remember, we want to predict who earns more than 50k or
X = df.loc[:, df.columns != 'gross-income'] # all other columns are features

random_state = 42

# first split to separate out the training set
X_train, X_other, y_train, y_other = train_test_split(X,y,train_size = 0.6,rando

# second split to separate out the validation and test sets
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_size = 0.5
```

In [16]:

```python
# collect which encoder to use on each feature
# needs to be done manually
ordinal_ftrs = ['education']
ordinal_cats = [[' Preschool',' 1st-4th',' 5th-6th',' 7th-8th',' 9th',' 10th','
                ' Some-college',' Assoc-voc',' Assoc-acdm',' Bachelors',' Master
onehot_ftrs = ['workclass','marital-status','occupation','relationship','race','
minmax_ftrs = ['age','hours-per-week']
std_ftrs = ['capital-gain','capital-loss']

# collect all the encoders
preprocessor = ColumnTransformer(
    transformers=[
        ('ord', OrdinalEncoder(categories = ordinal_cats), ordinal_ftrs),
        ('onehot', OneHotEncoder(sparse=False,handle_unknown='ignore'), onehot_f
        ('minmax', MinMaxScaler(), minmax_ftrs),
        ('std', StandardScaler(), std_ftrs)])

clf = Pipeline(steps=[('preprocessor', preprocessor)]) # for now we only preproc
                                                       # later on we will add ot

X_train_prep = clf.fit_transform(X_train)
X_val_prep = clf.transform(X_val)
X_test_prep = clf.transform(X_test)

print(X_train.shape)
print(X_train_prep.shape)
print(X_train_prep)
```

```
(19536, 14)
(19536, 91)
[[10.          0.          0.         ...  0.39795918 -0.14633293
  -0.22318878]
 [ 9.          0.          0.         ...  0.39795918 -0.14633293
  -0.22318878]
 [ 8.          0.          0.         ...  0.5        -0.14633293
  -0.22318878]
 ...
 [ 6.          0.          0.         ...  0.19387755 -0.14633293
  -0.22318878]
 [ 8.          0.          0.         ...  0.84693878 -0.14633293
  -0.22318878]
```

```
 [12.          0.          0.         ...  0.60204082 -0.14633293
  -0.22318878]]
```

```
In [ ]:
```