Mudcard

- **can we work on time series data for the project?**
  - yes, in fact you are encouraged to work with non-iid datasets
  - that's how you learn the most in this class
- **Can the autocorrelation plot help you pick an optimal lag period to use?**
  - it might give you an idea but you really need to measure the generalization error as a function of the number of lag-shifted features to come up with an answer
- **"What is the proper lag-step and number of features for different scenarios?**
  - as I said in class, there is no magic number that works for all cases
  - measure how the generalization error changes as a function of the number of lag-shifted features
  - that plot will give you an answer, but that answer will be specific to your project
- **I'm still unsure about how to prevent info leak in time series, doesn't all future data contain info from the past?**
  - the point is to not use future data to predict the past
  - this might seem obvious but it is one of those mistakes that are easy to make because your code will not produce an error message
- **Why does introducing lag features make the feature matrix iid?**
  - it doesn't
  - maybe you have an earlier version of the lecture notes, I made some changes on tuesday morning
  - check out the current version
- **I am still very unclear about what the "lag" is or how the "shifting" results in iid data.**
  - shifting does not result in iid data
  - maybe you also have access to the earlier version of the lecture notes
- **Why the autocorrelation plot decreases so smoothly in this pattern?**
- **can you explain the autocorrelation figure? why with the increasing lag, the correlation coefficienct convergent to 0**
  - check the code in the previous lecture
  - the autocorrelation coefficient has two terms: the Pearson correlation coefficient multiplied by the fraction of the dataset used
  - that second term approaches 0 as the leg increases
  - it is 1 for 0 leg, and 0 for the maximum lag you can calculate based on your dataset
- **Why do we have to make time lags for the weather data to predict future outcomes? Why cant we just take every other day and put in training set, and the remaining go in testing - or something similar with a not 50/50 split?**
  - because one time series measurement does not give you a feature matrix and a target variable
  - you need to transform it
  - you can't do a simple train_test_split because time series datasets are not iid

- **For time series, is the number of features inversely correlated with the number of observations for each one?**
  - Linearly correlated
  - if you have more observations, the maxmimum number of lag-shifted features you can create goes up
- **Why do we use autocorrelation and autoregression?**
  - to study and work with time series data
- **If your target variable is very far off in the future. How do you test for the accuracy of your prediction when you only have data form the past?**
  - you usually predict near future stuff
  - predictive power far in the future is usually pretty bad otherwise ML would be much widely used
- **For the wrist-band hospital model, would more of the same kind of data allow us to build a better model?**
  - that's the hope, yes
  - it is not guaranteed though
  - the conclusion of the paper was that there might be something promising here but we will need more data to verify
- **What is the best way to balance group splitting and stratified splitting? For example, if a class shows up in only one group, we may train models that do not ever see that class in training, but we also dont want to split the group. Is there a compromise between these two factors?**
  - good question!
  - I think it depends on how many grops you have
  - if you have many groups, this is not a problem because you will have multiple groups in each set
  - if you only have a limited number of groups, it might happen that your validation and/or test set will only contain that group and it will be difficult to assess the generalization error
- **So when making features for time-series data, these are all just lags?**
  - that's the basic idea, yes
  - you can add moving averages instead of the feature values
  - but generally lag shifting is what you do with time series data in ML
- **Do we just have one column of data (but being shifted differently into different columns) for autoregression?**
  - yep
  - you might have multiple time series observations, e.g., you might want to predict the S&P 500 price using the Tesla, Meta, and Amazon prices. You would need to shift all three price feeds to generate features
- **What does the term "data-leakage" mean exactly?**
  - it means that you use information during model development that will not be available to you when the model is deployed

    - as a result, your model gives a great test score (generalization error), but it will perform poorly when it is deployed
- **When should you use auto correlation? It intuitively feel like past values might negatively influence the prediction. Unless something is very predictable like temperature or weekly purchases, I feel like taking in account of past prices could be bad. It feels weird like cofounding variables.**
    - using the past data to predict the future is perfectly fine
    - that's mostly what people use to predict any sort of price movements
    - what other info would you use as features?
- **For predicting time series data, can we use an ARIMA model?**
    - no, stick to autoregressive features and apply the ML models we will cover in a few weeks during class
- **What would be a more intuitive way to understand these segmented bar charts for GroupShuffleSplit, GroupKFold, etc.? Currently not sure how the colored bars link to the concept**
    - read the manuals on the sklearn website
    - they provide longer explanations and more examples
- **Is it worthwhile to see how a target variable correlates to a lag of different features you have?**
    - if those other features are time series of other observations, yes
    - you only lag-shift time series observations
- **"I only have a vague understanding on the underlying purpose of using lags on time-series data, could you give some further explanation?**
    - check the sklearn website for more explanations and examples here
- **For the varieties of auto-regression that would show up in technical interviews, how deep should we know about it? Should we know how to actually imply them or to be able to explain the mechanism is fine?**
    - the more you know, the better
- **I am generally confused about the topic of autoregression and why one reorients features. Is it solely due based on time, or is there other considerations?**
    - yes, you would apply autoregression on time series data only
- **It would be helpful to fully walk through a time series analysis. What model would we use to predict the target variable? How do we measure the accuracy of our model as we try to predict more distant target variables?**
    - once the features are generated via autoregression, the rest of the steps are the same as for an iid or any other dataset
    - we will cover all of these topics in the coming weeks
- **I think I became more confused with the concept of K-fold and n_splits. Is there a really helpful crash course video that you would recommend in addition to watching the lecture over again?**
    - check out the sklearn website for more examples and more explanation
- **Could you explain more about the GroupKFold?**
    - I won't have time but check out the sklearn website for more info

- **And also for the part of time series, will we learn more about it? (like ARMA, SARIMA in this class)**
  - no, we won't cover more in this class, I just gave you the big picture overview and idea
  - but i do recommend you read more about those techniques
- **I am unclear how you calculate the confidence interval in the autocorrelation plot.**
  - pandas calculates it for you
  - check the pandas manual for an explanation
- **If there are non-time series features combined with time-series features, how do we train the model? Do we train the time-series "lag features" first, then we modify the model based on non-time series features? Training all the features at the same time does not seem feasible to me.**
  - it is feasable, you simply need to merge the time series features with the non-time series features
- **What are the advantages of group-based split instead of a Group Shuffle Split? When exactly we should choose one instead of the other?**
- **Would you be able to clarify we should use¬† GroupKFold vs GroupShuffleFold?**
  - both are fine
  - I'm giving you multiple tools to solve the same problem
  - I have a small personal preference for GroupKFold but that's subjective
- **Im still unclear what the main advantage/use of doing autocorrelation and lag would be? Is it exclusive for timeseries data?**
  - yes it is
- **Will we have much time series data?**
  - you can use a time series prediction for your project
  - and it is almost certain that you will come across time series data in your career at some point
- **What do you do with uncorrelated time series data?**
  - good question
  - I have not seen uncorrelated time series data
  - if the autocorrelation plot is flat with a correlation coefficient 0, it means you should not create autoregressive features, you need to collect some other features for prediction
- **What does it mean for data to be iid, or not iid, and why do other correlations not count as violating iid data?**
  - iid vs non-iid is about how the data is generated, it is not about correlations between the features and the target variable
- **Very muddy about the time series scenarios and why we even need to use machine learning for those types of problems. You're just calculating everything out no?**
  - what do you mean by 'calculating everything out'?
- **What is the purpose of the Pearson coefficient ?**
  - check out here

- **Sorry that this is a basic question: I'm a bit confused -- in a groupKfolds split, does each group get put in a fold by itself? How does that differentiate conceptually from GroupShuffleSplit?**
  - yes, in groupkfold, each group gets put in a fold by itself, so each group appears only once in the validation set
  - in groupshufflesplit, the groups in the validation are determined randomly, so the same group can be put in the validation set multiple times
- **I want to more practice during the class!**
  - me too but there is no time
  - you need to practice outside of class

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 c
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,rand

# 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.

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(sklearn.base._OneToOneFeatureMixin, _BaseEncoder)
 |  OrdinalEncoder(*, categories='auto', dtype=<class 'numpy.float64'>, handle
_unknown='error', unknown_value=None, encoded_missing_value=nan)
 |
 |  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 Non
e.
 |
 |      .. 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 o
f
 |      the categories in `fit`. If set to np.nan, the `dtype` parameter must
 |      be a float dtype.
 |
 |      .. versionadded:: 0.24
 |
 |  encoded_missing_value : int or np.nan, default=np.nan
 |      Encoded value of missing categories. If set to `np.nan`, then the `dty
pe`
 |      parameter must be a float dtype.
 |
```

```
|      .. versionadded:: 1.1
|
|  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``.
|
|  n_features_in_ : int
|      Number of features seen during :term:`fit`.
|
|      .. versionadded:: 1.0
|
|  feature_names_in_ : ndarray of shape (`n_features_in_`,)
|      Names of features seen during :term:`fit`. Defined only when `X`
|      has feature names that are all strings.
|
|      .. versionadded:: 1.0
|
|  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)
|
|  By default, :class:`OrdinalEncoder` is lenient towards missing values by
|  propagating them.
|
|  >>> import numpy as np
|  >>> X = [['Male', 1], ['Female', 3], ['Female', np.nan]]
|  >>> enc.fit_transform(X)
|  array([[ 1.,  0.],
|         [ 0.,  1.],
|         [ 0., nan]])
|
|  You can use the parameter `encoded_missing_value` to encode missing value
```

```
s.
 |
 |  >>> enc.set_params(encoded_missing_value=-1).fit_transform(X)
 |  array([[ 1.,  0.],
 |         [ 0.,  1.],
 |         [ 0., -1.]])
 |
 |  Method resolution order:
 |      OrdinalEncoder
 |      sklearn.base._OneToOneFeatureMixin
 |      _BaseEncoder
 |      sklearn.base.TransformerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *, categories='auto', dtype=<class 'numpy.float64'>, handle
_unknown='error', unknown_value=None, encoded_missing_value=nan)
 |      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 : object
 |          Fitted encoder.
 |
 |  inverse_transform(self, X)
 |      Convert the data back to the original representation.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_encoded_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._OneToOneFeatureMixin:
 |
 |  get_feature_names_out(self, input_features=None)
 |      Get output feature names for transformation.
 |
 |      Parameters
 |      ----------
 |      input_features : array-like of str or None, default=None
 |          Input features.
 |
 |          - If `input_features` is `None`, then `feature_names_in_` is
 |            used as feature names in. If `feature_names_in_` is not defined,
 |            then the following input feature names are generated:
 |            `["x0", "x1", ..., "x(n_features_in_ - 1)"]`.
 |          - If `input_features` is an array-like, then `input_features` must
 |            match `feature_names_in_` if `feature_names_in_` is defined.
 |
 |      Returns
 |      -------
 |      feature_names_out : ndarray of str objects
 |          Same as input features.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base._OneToOneFeatureMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  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.
|
|  ----------------------------------------------------------------------
|  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','Bache
```

```
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 orde
print(enc.categories_)
# transform X_train. We could have used enc.fit_transform(X_train) to combine f
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 me
                                     # 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]:

```
# apply OE to the adult dataset
# initialize the encoder
ordinal_ftrs = ['education'] # if you have more than one ordinal feature, add t
ordinal_cats = [[' Preschool',' 1st-4th',' 5th-6th',' 7th-8th',' 9th',' 10th','
                ' Some-college',' Assoc-voc',' Assoc-acdm',' Bachelors',' Maste
# ordinal_cats must contain one list per ordinal feature! each list contains th
# of the corresponding feature

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

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

# 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]:
```
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
'numpy.float64'>, handle_unknown='error', min_frequency=None, max_categories=N
one)
 |
 |  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 an 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 an unregularized linear regression model.
 |
 |      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.
 |
 |      .. versionchanged:: 1.1
 |          Support for dropping infrequent categories.
 |
 |  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', 'infrequent_if_exist'},
default='error'
 |      Specifies the way unknown categories are handled during :meth:`transfo
rm`.
 |
 |      - 'error' : Raise an error if an unknown category is present during tr
ansform.
 |      - 'ignore' : When 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.
 |      - 'infrequent_if_exist' : When an unknown category is encountered
 |        during transform, the resulting one-hot encoded columns for this
 |        feature will map to the infrequent category if it exists. The
 |        infrequent category will be mapped to the last position in the
 |        encoding. During inverse transform, an unknown category will be
 |        mapped to the category denoted `'infrequent'` if it exists. If the
 |        `'infrequent'` category does not exist, then :meth:`transform` and
 |        :meth:`inverse_transform` will handle an unknown category as with
 |        `handle_unknown='ignore'`. Infrequent categories exist based on
 |        `min_frequency` and `max_categories`. Read more in the
 |        :ref:`User Guide <one_hot_encoder_infrequent_categories>`.
 |
 |      .. versionchanged:: 1.1
 |          `'infrequent_if_exist'` was added to automatically handle unknown
 |          categories and infrequent categories.
 |
 |  min_frequency : int or float, default=None
 |      Specifies the minimum frequency below which a category will be
 |      considered infrequent.
 |
 |      - If `int`, categories with a smaller cardinality will be considered
 |        infrequent.
 |
 |      - If `float`, categories with a smaller cardinality than
 |        `min_frequency * n_samples`  will be considered infrequent.
```

```
 |
 |      .. versionadded:: 1.1
 |          Read more in the :ref:`User Guide <one_hot_encoder_infrequent_cate
gories>`.
 |
 |  max_categories : int, default=None
 |      Specifies an upper limit to the number of output features for each inp
ut
 |      feature when considering infrequent categories. If there are infrequen
t
 |      categories, `max_categories` includes the category representing the
 |      infrequent categories along with the frequent categories. If `None`,
 |      there is no limit to the number of output features.
 |
 |      .. versionadded:: 1.1
 |          Read more in the :ref:`User Guide <one_hot_encoder_infrequent_cate
gories>`.
 |
 |  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.
 |
 |      If infrequent categories are enabled by setting `min_frequency` or
 |      `max_categories` to a non-default value and `drop_idx[i]` corresponds
 |      to a infrequent category, then the entire infrequent category is
 |      dropped.
 |
 |      .. versionchanged:: 0.23
 |         Added the possibility to contain `None` values.
 |
 |  infrequent_categories_ : list of ndarray
 |      Defined only if infrequent categories are enabled by setting
 |      `min_frequency` or `max_categories` to a non-default value.
 |      `infrequent_categories_[i]` are the infrequent categories for feature
 |      `i`. If the feature `i` has no infrequent categories
 |      `infrequent_categories_[i]` is None.
 |
 |      .. versionadded:: 1.1
 |
 |  n_features_in_ : int
 |      Number of features seen during :term:`fit`.
 |
 |      .. versionadded:: 1.0
 |
```

```
|  feature_names_in_ : ndarray of shape (`n_features_in_`,)
|      Names of features seen during :term:`fit`. Defined only when `X`
|      has feature names that are all strings.
|
|      .. versionadded:: 1.0
|
|  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_out(['gender', 'group'])
|  array(['gender_Female', 'gender_Male', 'group_1', 'group_2', 'group_3'],
...)
|
|  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.]])
 |
 |  Infrequent categories are enabled by setting `max_categories` or `min_freq
uency`.
 |
 |  >>> import numpy as np
 |  >>> X = np.array([["a"] * 5 + ["b"] * 20 + ["c"] * 10 + ["d"] * 3], dtype=
object).T
 |  >>> ohe = OneHotEncoder(max_categories=3, sparse=False).fit(X)
 |  >>> ohe.infrequent_categories_
 |  [array(['a', 'd'], dtype=object)]
 |  >>> ohe.transform([["a"], ["b"]])
 |  array([[0., 0., 1.],
 |         [1., 0., 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
'numpy.float64'>, handle_unknown='error', min_frequency=None, max_categories=N
one)
 |      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
 |          Fitted encoder.
 |
 |  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)
 |      DEPRECATED: get_feature_names is deprecated in 1.0 and will be removed
in 1.2. Please use get_feature_names_out instead.
 |
 |      Return feature names for output features.
 |
 |          For a given input feature, if there is an infrequent category, the
most
 |          'infrequent_sklearn' will be used as a feature name.
 |
 |          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.
 |
 |  get_feature_names_out(self, input_features=None)
 |      Get output feature names for transformation.
 |
 |      Parameters
 |      ----------
 |      input_features : array-like of str or None, default=None
 |          Input features.
 |
 |          - If `input_features` is `None`, then `feature_names_in_` is
 |            used as feature names in. If `feature_names_in_` is not defined,
 |            then the following input feature names are generated:
 |            `["x0", "x1", ..., "x(n_features_in_ - 1)"]`.
 |          - If `input_features` is an array-like, then `input_features` must
 |            match `feature_names_in_` if `feature_names_in_` is defined.
 |
 |      Returns
 |      -------
 |      feature_names_out : ndarray of str objects
 |          Transformed feature names.
 |
 |  inverse_transform(self, X)
 |      Convert the data back to the original representation.
 |
 |      When unknown categories are encountered (all zeros in the
```

```
 |      one-hot encoding), ``None`` is used to represent this category. If the
 |      feature with the unknown category has a dropped category, the dropped
 |      category will be its inverse.
 |
 |      For a given input feature, if there is an infrequent category,
 |      'infrequent_sklearn' will be used to represent the infrequent categor
y.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape                (n_samples, n
_encoded_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.
 |
 |      If there are infrequent categories for a feature, the infrequent
 |      categories will be grouped into a single category.
 |
 |      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.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  infrequent_categories_
 |      Infrequent categories for each feature.
 |
 |  ----------------------------------------------------------------------
 |  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 [6]:

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

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

ftrs = ['gender','browser']

# initialize the encoder
enc = OneHotEncoder(sparse=False) # by default, OneHotEncoder returns a sparse
# 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)
```

```python
# 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_Chrom
e'
 '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.]]
```

```
/Users/azsom/opt/anaconda3/envs/data1030/lib/python3.10/site-packages/sklearn/
utils/deprecation.py:87: FutureWarning: Function get_feature_names is deprecat
ed; get_feature_names is deprecated in 1.0 and will be removed in 1.2. Please
use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [6], in <cell line: 26>()
     23 print(X_train_ohe)
     25 # transform X_test
---> 26 X_test_ohe = enc.transform(Xtoy_test)
     27 print('X_test transformed')
     28 print(X_test_ohe)

File ~/opt/anaconda3/envs/data1030/lib/python3.10/site-packages/sklearn/prepro
cessing/_encoders.py:882, in OneHotEncoder.transform(self, X)
    877 # validation of X happens in _check_X called by _transform
    878 warn_on_unknown = self.drop is not None and self.handle_unknown in {
    879     "ignore",
    880     "infrequent_if_exist",
    881 }
--> 882 X_int, X_mask = self._transform(
    883     X,
    884     handle_unknown=self.handle_unknown,
    885     force_all_finite="allow-nan",
    886     warn_on_unknown=warn_on_unknown,
    887 )
    888 self._map_infrequent_categories(X_int, X_mask)
    890 n_samples, n_features = X_int.shape

File ~/opt/anaconda3/envs/data1030/lib/python3.10/site-packages/sklearn/prepro
cessing/_encoders.py:160, in _BaseEncoder._transform(self, X, handle_unknown,
 force_all_finite, warn_on_unknown)
    155 if handle_unknown == "error":
    156     msg = (
    157         "Found unknown categories {0} in column {1}"
    158         " during transform".format(diff, i)
    159     )
--> 160     raise ValueError(msg)
    161 else:
    162     if warn_on_unknown:

ValueError: Found unknown categories ['Firefox'] in column 1 during transform
```

In [ ]:
```
# 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, OneHotE
# fit the training data
enc.fit(X_train[onehot_ftrs])
print('feature names:',enc.get_feature_names(onehot_ftrs))
```

In [ ]:
```
# 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)
```

Quiz 1

Please explain how you would encode the race feature below and what would be the output of the encoder. Do not write code. The goal of this quiz is to test your conceptual understanding so write text and the output array.

race = [' Amer-Indian-Eskimo', 'White', 'Black', 'Asian-Pac-Islander', Black, White, 'White']

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 [ ]:
```
from sklearn.preprocessing import MinMaxScaler
help(MinMaxScaler)
```

In [ ]:
```
# toy data
# let's assume we have two continuous features:
train = {'age':[32,65,13,68,42,75,32],'number of hours worked':[0,40,10,60,40,2
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 lar
```

In [ ]:
```
# 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]))
```

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 [ ]:
```
from sklearn.preprocessing import StandardScaler
help(StandardScaler)
```

In [ ]:
```
# 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))
```

In [ ]:
```
# 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]))
```

Quiz 2

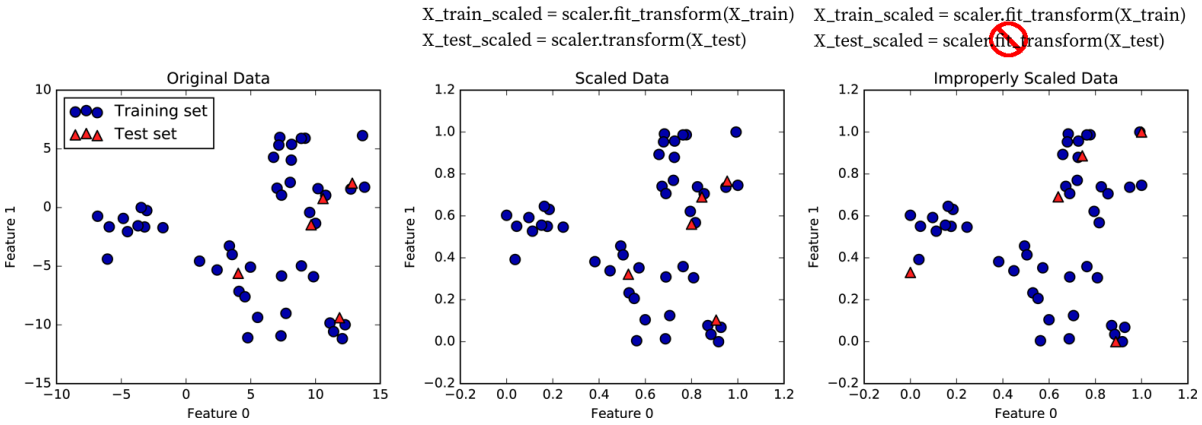
Which of these features could be safely preprocessed by the minmax scaler?

- 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 [ ]:
```
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
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
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,rand

# 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.
```

In [ ]:
```
# 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',' Maste
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_
        ('minmax', MinMaxScaler(), minmax_ftrs),
        ('std', StandardScaler(), std_ftrs)])

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

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)
```

Mudcard

In [ ]:
```

```