The predictive modeling pipeline

Objectives

- build intuitions about an unknown dataset
- identify and differentiate numerical and categorical features
- create an advanced predictive pipeline with scikit-learn

(1) Tabular data exploration

First look at our dataset

Necessary steps before any machine learning happens:

- load the data
- look at the variables in the dataset: numerical vs categorical variables --> they need different processing
- visualize the distribution of the variables to gain insights into the dataset

Loading the adult census dataset

- see openml website: https://www.openml.org/search?type=data&sort=runs&id=1590&status=active

```
In [1]: import pandas as pd
```

```
In [2]: adult_census = pd.read_csv("../datasets/adult-census.csv")
```

Goal: we would like to predict whether a person earns more than 50k a year from data such as

- age
- employment
- education
- family information

The variables in the dataset

- `pandas` dataframe = data composed of 2 dimensions. "tabular data"
    - one row = one "sample" // "record", "instance", "observation"
    - one column = attribute of the observation // "variable", "attribute", "covariate"

```
In [3]: adult_census.head()
```

Out[3]:

	age	workclass	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	cla
0	25	Private	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=5
1	38	Private	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=5
2	28	Local-gov	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>5
3	44	Private	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>5
4	18	?	Some-college	10	Never-married	?	Own-child	White	Female	0	0	30	United-States	<=5

- column `class` is the *target variable* -- variable we want to predict
- binary prediction: `class` has two possible values
- we will use the remaining columns as input variables for the models

Let's see the distribution over the two classes with the `value_counts()` method.

```
In [4]:  target_column = "class"
         adult_census[target_column].value_counts()

Out[4]:  class
          <=50K    37155
          >50K     11687
         Name: count, dtype: int64
```

Class imbalance *in the outcome variable*

- needs special techniques for predictive modeling (intuition: model ignores small groups because they are less important in the objective function)
- example: medical setting with rare diseases

Column types

- numerical -- continuous values
- categorical -- finite number of values

<mark>NOTE copy-paste the code below → inform helper</mark>

```
In [5]:  numerical_columns = [
             "age",
             "education-num",
             "capital-gain",
             "capital-loss",
             "hours-per-week",
         ]
         categorical_columns = [
             "workclass",
             "education",
             "marital-status",
             "occupation",
             "relationship",
             "race",
             "sex",
             "native-country",
         ]
```

```
In [6]:  all_columns = numerical_columns + categorical_columns + [target_column]
```

```
In [7]:  # let's just make sure we only have the cells we are interested in
         adult_census = adult_census[all_columns]
```

```
In [8]:  adult_census.shape
```

```
Out[8]:  (48842, 14)
```

```
In [9]:  # Check the number of observations and number of columns
         # print(
         #     f"The dataset contains {adult_census.shape[0]} samples and "
         #     f"{adult_census.shape[1]} columns"
         # )
```

<mark>NOTE Explain what</mark> `df.shape` <mark>does</mark>

```
In [10]:  ?adult_census.shape
```

```
In [11]:  # since 1 column is the dataset, we can count the number of features as
          # print(f"The dataset contains {adult_census.shape[1] - 1} features.")
```
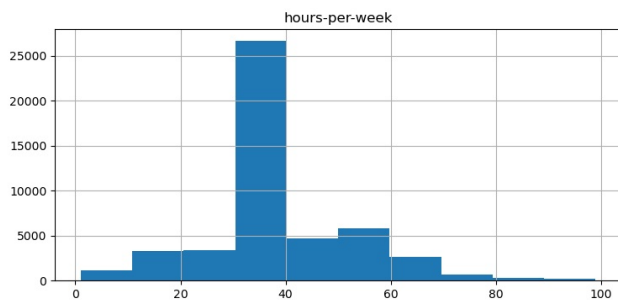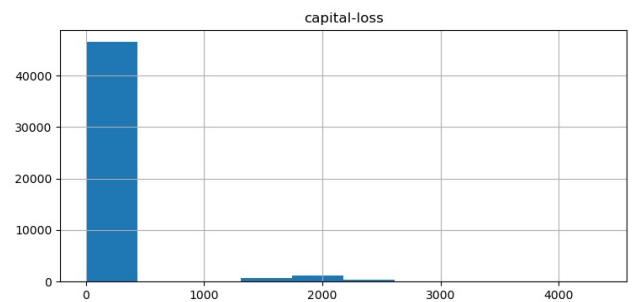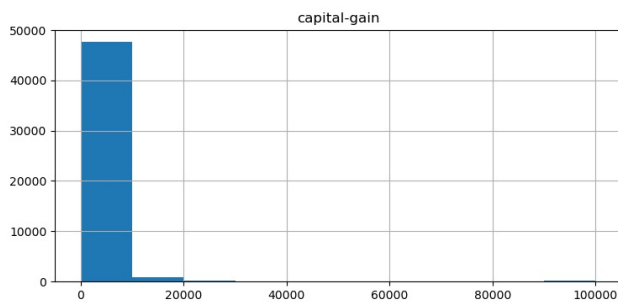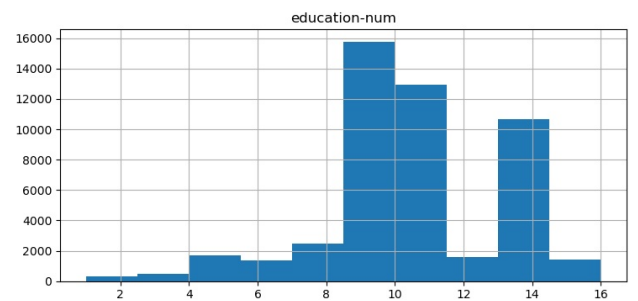
Visually inspecting the data

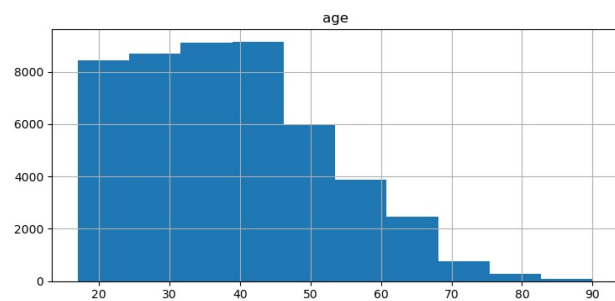Good idea to look at the data before building a predictive model

- maybe the task does not need ML (example??)
- check the information necessary for the task is in the data
- find data peculiarities: missing data, capped values, ...

<mark>NOTE First run without the</mark> `_` `=` <mark>to show its impact. "garbage variable" we don't need</mark>

```
In [12]:  _ = adult_census.hist(figsize=(20, 14))
```

What have we done?

- `hist` creates a histogram. It is built-in to the dataframe object and makes one histogram per numerical variable. Convenient for quickly exploring data
- the x axis shows the values of the variable and the y axis shows the number of samples in each bin of x

Call-out What do we see in these histograms?

- "age": few data beyond 70. reason: data filtered `hours-per-week` > 0 (data description)
- "education-num": peak around 10-13, but unclear where exactly
- "hours-per-week": peak at 40 -- standard number of hours worked at the time when data were collected
- "capital-gain", "capital-loss": most values close to 0.

For categorical variables, look at the distribution as follows

```
In [13]: adult_census["sex"].value_counts()
```

```
Out[13]: sex
         Male      32650
         Female    16192
         Name: count, dtype: int64
```

(Explain the output: `name`, `dtype`.)

Class imbalance in the features

- many more men than women (-- perhaps b/c hours-per-week > 0?)
- this can lead to disproportionate prediction errors for under-represented groups -- fairness problems in ML when systems deployed naively
- fairlearn.org: learn more about makeing ML fair across social groups. https://fairlearn.org/

```
In [14]: adult_census["education"].value_counts()
```

education
          HS-grad   15784
          Some-college   10878
          Bachelors   8025
          Masters   2657
          Assoc-voc   2061
          11th   1812
          Assoc-acdm   1601
          10th   1389
          7th-8th   955
          Prof-school   834
          9th   756
          12th   657
          Doctorate   594
          5th-6th   509
          1st-4th   247
          Preschool   83
          Name: count, dtype: int64

What is the relation between `education` and `education-num` ?

```python
pd.crosstab(
    index=adult_census["education"], columns=adult_census["education-num"]
)
```

education-num	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
education																
10th	0	0	0	0	0	1389	0	0	0	0	0	0	0	0	0	0
11th	0	0	0	0	0	0	1812	0	0	0	0	0	0	0	0	0
12th	0	0	0	0	0	0	0	657	0	0	0	0	0	0	0	0
1st-4th	0	247	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5th-6th	0	0	509	0	0	0	0	0	0	0	0	0	0	0	0	0
7th-8th	0	0	0	955	0	0	0	0	0	0	0	0	0	0	0	0
9th	0	0	0	0	756	0	0	0	0	0	0	0	0	0	0	0
Assoc-acdm	0	0	0	0	0	0	0	0	0	0	1601	0	0	0	0	0
Assoc-voc	0	0	0	0	0	0	0	0	0	2061	0	0	0	0	0	0
Bachelors	0	0	0	0	0	0	0	0	0	0	0	8025	0	0	0	0
Doctorate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	594
HS-grad	0	0	0	0	0	0	0	0	15784	0	0	0	0	0	0	0
Masters	0	0	0	0	0	0	0	0	0	0	0	0	2657	0	0	0
Preschool	83	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Prof-school	0	0	0	0	0	0	0	0	0	0	0	0	0	834	0	0
Some-college	0	0	0	0	0	0	0	0	0	10878	0	0	0	0	0	0

What do we see?

- entries in `education` and `education-num` correspond exactly to each other
  - they give us the same information
- --> we can remove `education-num` without losing information

Let's do this for future reference

NOTE: do not drop `"education-num "` already; it's used below for the pairplot

We can also inspect the data with a pairplot, and show how each variable differs according to the target variable.

<mark>NOTE copy-paste the code below</mark>

<mark>Note to helper</mark>

> Copy the code for creating the plots into the collaborative document. Since our course is on ML and not on plotting, we don't want to spend time on explaining this code.

```python
import seaborn as sns

# We will plot a subset of the data to keep the plot readable and make the
# plotting faster
n_samples_to_plot = 5000
```

```
columns = ["age", "education-num", "hours-per-week"]
_ = sns.pairplot(
    data=adult_census[:n_samples_to_plot],
    vars=columns,
    hue=target_column,
    plot_kws={"alpha": 0.2},
    height=3,
    diag_kind="hist",
    diag_kws={"bins": 30},
)
```
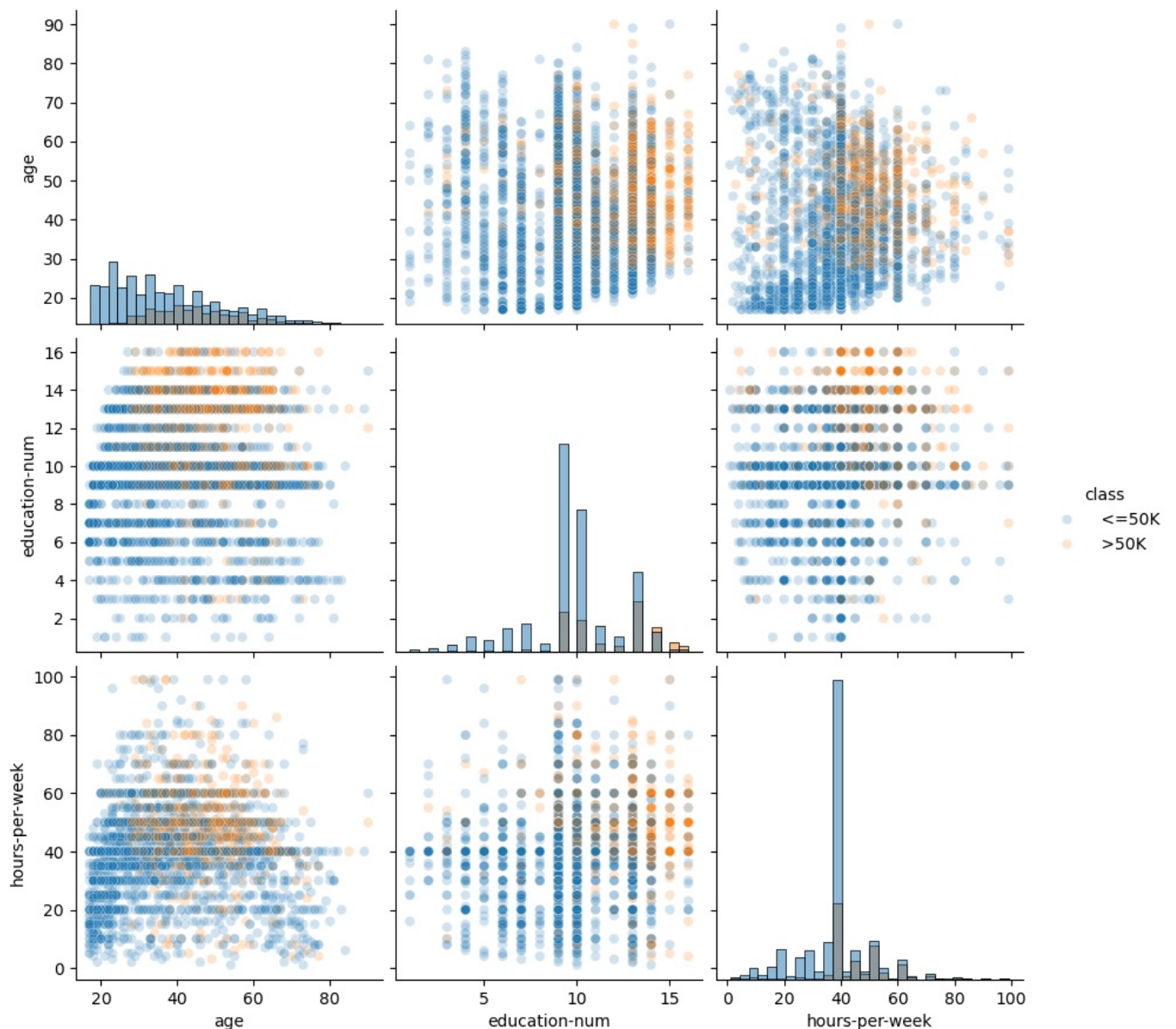
- the colors: blue -- below 50k, red above 50k
- in the diagonal: histograms separately by class in the target
- in the off-diagonal: scatter plots between variables. what do they show? -- correlations between the variables, and how they relate to the outcome/target variable.

Creating decision rules by hand

From the previous plot (bottom left): create some rule by hand with the `hours-per-week` and `age` features

In [17]:
```
_ = sns.scatterplot(
    x="age",
    y="hours-per-week",
    data=adult_census[:n_samples_to_plot],
    hue=target_column,
    alpha=0.5,
)
```
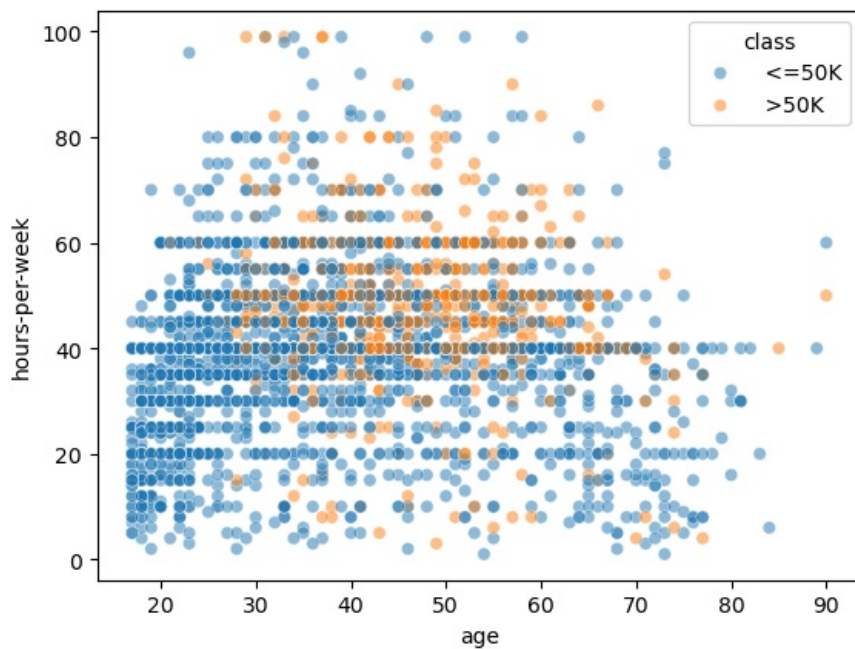
- young people, below 27, almost all have low income
- above 27, there seem to be 2 groups
    - those with low hours per week, say below 40, have low income
    - those with hours 40 or more, the prediction is much less clear
- we will see later that some methods -- decision tree -- work similar to what we did here, but they select the thresholds automatically (and in an optimal way)
- moreover, ML is handy precisely in cases where it's not obvious to the human eye
    - like in the last group above
    - or when there are many features

In sum

- ML automatically creates the rules from the existing data to make predictions on new unseen data.

Wrap-up: important observations

- if target is imbalanced, special techniques are necessary for training and evaluating the model
- redundant (highly correlated) features can be a problem for some ML algorithms

Exercise M1.01: exploring a dataset

Questions: see the exercise document, header "Data exploration"

Solutions

```
In [18]: penguins = pd.read_csv("../datasets/penguins_classification.csv")
```

```
In [19]: penguins.head()
```

Out[19]:

	Culmen Length (mm)	Culmen Depth (mm)	Species
0	39.1	18.7	Adelie
1	39.5	17.4	Adelie
2	40.3	18.0	Adelie
3	36.7	19.3	Adelie
4	39.3	20.6	Adelie

```
In [20]: nrows, ncols = penguins.shape
         print(f"The data have {nrows} rows and {ncols} cols")
```

The data have 342 rows and 3 cols

(1) How many features are numerical? How many features are categorical?

features

- culmen length and depth are numerical
- (species is the outcome and is categorical -- to the extent this is considered a feature)

(2) What are the different penguins species available in the dataset and how many samples of each species are there?

```
In [21]: target_column = "Species"
         penguins[target_column].value_counts()
```

```
Out[21]: Species
         Adelie       151
         Gentoo       123
         Chinstrap     68
         Name: count, dtype: int64
```

(3) Plot histograms for the numerical features

```
In [22]: _ = penguins.hist()
```
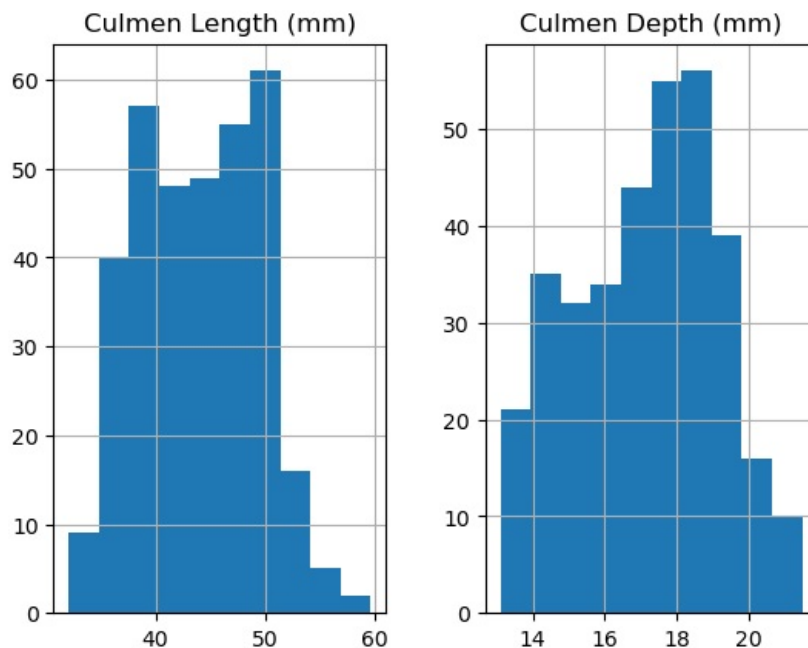

(4) Plot features distribution for each class (Hint: use `seaborn.pairplot`).

```
In [23]: # We will plot a subset of the data to keep the plot readable and make the
         # plotting faster
         _ = sns.pairplot(penguins, hue="Species")
```
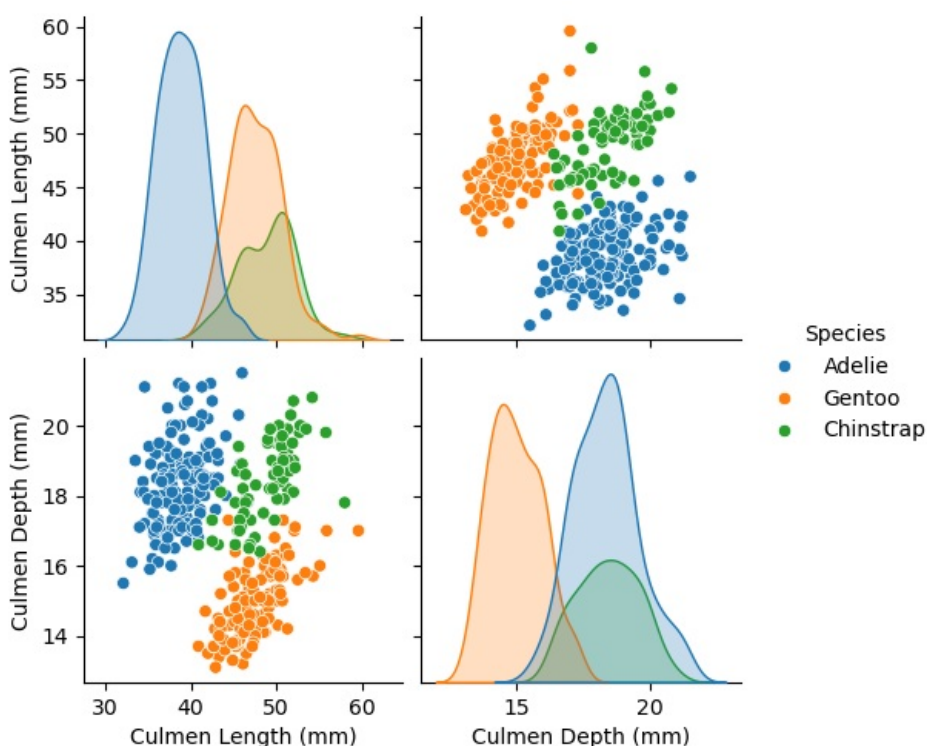

(5) Looking at the distributions you got, how hard do you think it will be to classify the penguins only using "culmen depth" and

"culmen length"?

It looks like classification should not be too hard -- the groups seem well separated from each other based on the numerical features

(2) Handling categorical data, encoding

Encoding of categorical variables

<mark>Question to audience</mark> : what are examples of categorical data?

- gender
- place of work
- ...

```
In [24]: # adult_census = pd.read_csv("../datasets/adult-census.csv")
```

```
In [25]: # drop duplicated column
         # adult_census = adult_census.drop(columns="education-num")
         adult_census = adult_census.drop(columns="education-num")
```

```
In [26]: target_name = "class"
         target = adult_census[target_name]
         data = adult_census.drop(columns=[target_name])
```

Identify categorical variables

they are different from the numerical variables

- they are often encoded as strings
- they have a finite set of different values

Explain what the command below does

- `value_counts` : counts occurences of each group
- `sort_index()` : sorts by index, here the index is native-country (this index is created by the preceding function)

```
In [27]: data["native-country"].value_counts().sort_index()
```

```
Out[27]: native-country
         ?                              857
         Cambodia                        28
         Canada                         182
         China                          122
         Columbia                        85
         Cuba                           138
         Dominican-Republic             103
         Ecuador                         45
         El-Salvador                    155
         England                        127
         France                          38
         Germany                        206
         Greece                          49
         Guatemala                       88
         Haiti                           75
         Holand-Netherlands               1
         Honduras                        20
         Hong                            30
         Hungary                         19
         India                          151
         Iran                            59
         Ireland                         37
         Italy                          105
         Jamaica                        106
         Japan                           92
         Laos                            23
         Mexico                         951
         Nicaragua                       49
         Outlying-US(Guam-USVI-etc)      23
         Peru                            46
         Philippines                    295
         Poland                          87
         Portugal                        67
         Puerto-Rico                    184
         Scotland                        21
         South                          115
         Taiwan                          65
         Thailand                        30
         Trinadad&Tobago                 27
         United-States                43832
         Vietnam                         86
         Yugoslavia                      23
         Name: count, dtype: int64
```

we can recognize categorical columns with the data type

```
In [28]: data.dtypes
```

```
Out[28]: age              int64
         capital-gain     int64
         capital-loss     int64
         hours-per-week   int64
         workclass        object
         education        object
         marital-status   object
         occupation       object
         relationship     object
         race             object
         sex              object
         native-country   object
         dtype: object
```

the variable `"native-country"` is data type `object` -- this means it contains string values

Instead of manually selecting the columns, we can use the scikit-learn helper function `make_column_selector`. It allows us to select columns based on the data type

```
In [29]: from sklearn.compose import make_column_selector as selector

         categorical_columns_selector = selector(dtype_include=object)#create the selector and pass argument about which
         categorical_columns = categorical_columns_selector(data)
         categorical_columns # returns a list of column names that satisfy the specification in the selector
```

```
Out[29]: ['workclass',
          'education',
          'marital-status',
          'occupation',
          'relationship',
          'race',
          'sex',
          'native-country']
```

Now we can select the columns in the original data set

```
In [30]: data_categorical = data[categorical_columns]
         data_categorical.head()
```

Out[30]:

	workclass	education	marital-status	occupation	relationship	race	sex	native-country
0	Private	11th	Never-married	Machine-op-inspct	Own-child	Black	Male	United-States
1	Private	HS-grad	Married-civ-spouse	Farming-fishing	Husband	White	Male	United-States
2	Local-gov	Assoc-acdm	Married-civ-spouse	Protective-serv	Husband	White	Male	United-States
3	Private	Some-college	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	United-States
4	?	Some-college	Never-married	?	Own-child	White	Female	United-States

```
In [31]: print(f"The dataset is composed of {data_categorical.shape[1]} features")
```

The dataset is composed of 8 features

Now that we have the columns ready, we can look at different strategies to encode categorical data into numerical data, which are suited for machine learning

Strategies to encode categories

Encoding ordinal categories

Most intuitive: just assign a different number to each category. We can do this with the `OrdinalEncoder`.

```
In [32]: from sklearn.preprocessing import OrdinalEncoder

         education_column = data_categorical[["education"]]

         encoder = OrdinalEncoder().set_output(transform="pandas") # we set the output to pandas dataframe
         education_encoded = encoder.fit_transform(education_column)
         # fit_transform first fits (=detects unique values and assigns them a number), and then transforms the input
         # we could also only fit, and then transform a new dataset

         education_encoded
```

Out[32]:

	education
0	1.0
1	11.0
2	7.0
3	15.0
4	15.0
...	...
48837	7.0
48838	11.0
48839	11.0
48840	11.0
48841	11.0

48842 rows × 1 columns

```
In [33]: # ?OrdinalEncoder.set_output
```

We can see which values of the original columns receive which number -- starting at 0 and counting through. For example, the first row was "11th", and now is 1.0. Thus, "11th" is mapped to 1.0.

```
In [34]: encoder.categories_
```

```
Out[34]: [array([' 10th', ' 11th', ' 12th', ' 1st-4th', ' 5th-6th', ' 7th-8th',
                 ' 9th', ' Assoc-acdm', ' Assoc-voc', ' Bachelors', ' Doctorate',
                 ' HS-grad', ' Masters', ' Preschool', ' Prof-school',
                 ' Some-college'], dtype=object)]
```

```
In [35]: # apply the encoding to all categorical features
         data_encoded = encoder.fit_transform(data_categorical)
         data_encoded[:5] # show the first five rows
```

	workclass	education	marital-status	occupation	relationship	race	sex	native-country
0	4.0	1.0	4.0	7.0	3.0	2.0	1.0	39.0
1	4.0	11.0	2.0	5.0	0.0	4.0	1.0	39.0
2	2.0	7.0	2.0	11.0	0.0	4.0	1.0	39.0
3	4.0	15.0	2.0	7.0	0.0	2.0	1.0	39.0
4	0.0	15.0	4.0	0.0	3.0	4.0	0.0	39.0

In [36]:
```python
print(f"The dataset encoded contains {data_encoded.shape[1]} features")
```
The dataset encoded contains 8 features

Observations

- each feature was encoded independently
- the number of features remains the same

leave the below out -- see exercise instead One needs to be careful when applying this encoding strategy

- the numbering implies that the values are ordered, for instance 0 < 1 < 2 < ...
- however, depending on the model we use, this can be misleading.
- first, the order may not be what we want. for instance, suppose we have a `"size"` variable with categories "S", "M", "L", "XL". The encoder would assign the values `2, 1, 0, 3` to these groups, following alphabetical order.
- for this case, one can specify the ordering explicitly. see here: https://scikit-learn.org/stable/modules/preprocessing.html#encoding-categorical-features
- second, sometimes having a number odes not even make sense, like in variable `"native-country"`.

(difficult to explain, leave out) moreover, it implies that changing from one neighboring category to the next is always the same "amount of change". this may make sense in some cases, but for instance for the country of origin

Exercise: ordinal encoding

header "ordinal encoding" in exercise document

Answers:

- A1: Only education (in fact, the encoder was already present in the data set as education-num), as this is the only one that can be expressed as an incremental feature
- A2: Examples could be:
  - Alphabetized: US grading system: A, B, C, D, F
  - Not alphabetized: clothing sizes: XS, S, M, L, XL, XXL
- A3: Would not be in correct order (it's alphabetized).
- AA4: top of documention will tell you to use the `categories` argument with a list in the correct order

  ```python
  ordered_size_list = ['XS', 'S', 'M', 'L', 'XL', 'XXL']
  encoder_with_order = OrdinalEncoder(categories=ordered_size_list)
  ```

- A5: US grading scheme is alphabetical to begin with (A,B,C,D,F)

Encoding nominal categories -- without assuming any order

This is an alternative encoder. It prevents the model from making false assumptions about the ordering of the categories.

-> `OneHotEncoder`

Encoding this way will create one new column for each category of a variable, containing `0`s and `1`s.

In [37]:
```python
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False).set_output(transform="pandas")
# NOTE: we use sparse_output=False for illustration. In practice, it's better to use sparse_output=True, which
education_encoded = encoder.fit_transform(education_column)
education_encoded
```

	education_10th	education_11th	education_12th	education_1st-4th	education_5th-6th	education_7th-8th	education_9th	education_Assoc-acdm	education_Assoc-voc	education_Bachelors	e
0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
...	...	...	...	...	...	...	...	...	...	...	
48837	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	
48838	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
48839	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
48840	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
48841	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

48842 rows × 16 columns
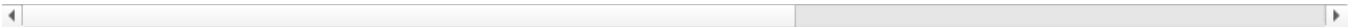
Comment

- each category has become a column. It includes 1 when the observation falls into that category
- cold also use `education_encoded.describe()` to show values are always either 0/1

We can apply this encoding to the full dataset.

```python
print(f"The dataset is composed of {data_categorical.shape[1]} features")
data_categorical.head()
```

The dataset is composed of 8 features

	workclass	education	marital-status	occupation	relationship	race	sex	native-country
0	Private	11th	Never-married	Machine-op-inspct	Own-child	Black	Male	United-States
1	Private	HS-grad	Married-civ-spouse	Farming-fishing	Husband	White	Male	United-States
2	Local-gov	Assoc-acdm	Married-civ-spouse	Protective-serv	Husband	White	Male	United-States
3	Private	Some-college	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	United-States
4	?	Some-college	Never-married	?	Own-child	White	Female	United-States

```python
data_encoded = encoder.fit_transform(data_categorical)
```

```python
data_encoded[:5]
```

	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	education_10th	...
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	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...
2	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
3	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...
4	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

5 rows × 102 columns

```python
print(f"The encoded dataset contains {data_encoded.shape[1]} features")
```

The encoded dataset contains 102 features

Notes

- the number of features is now much larger -- that is because some columns in the original data have many possible categories

Look at how the "workclass" variable of the 3 first records has been encoded and compare this to the original string representation.

```python
data_categorical["workclass"].head()
```

```
0       Private
1       Private
2      Local-gov
3       Private
4          ?
Name: workclass, dtype: object
```

Choosing an encoding strategy <mark>How to best teach this?</mark>

- it depends on the model we used
  - for linear models, one typically uses `OneHotEncoder`
  - for tree-based models, one typically uses `OrdinalEncoder`

`OrdinalEncoder` outputs ordinal categories, so there is an order in the resulting categories (0 < 1 < 2). Depending on the model, applying a Ordinal when categories are not ordered impacts linear models, but not tree-based models.

Using `OneHotEncoder` for linear models can cause computational inefficiency in tree-based models. Using `OrdinalEncoder` for a linear model is ok under some exceptions

1. Original data have an ordering
2. The encoded categories follow the same ordering as the original categories

In [ ]:

In [ ]:

Evaluate our predictive pipeline

We can now integrate the encoded data into the machine learning model.

But first: to do some more wrangling with the `native-country` column

In [43]:
```
data["native-country"].value_counts()
```

Out[43]:
```
native-country
United-States              43832
Mexico                       951
?                            857
Philippines                  295
Germany                      206
Puerto-Rico                  184
Canada                       182
El-Salvador                  155
India                        151
Cuba                         138
England                      127
China                        122
South                        115
Jamaica                      106
Italy                        105
Dominican-Republic           103
Japan                         92
Guatemala                     88
Poland                        87
Vietnam                       86
Columbia                      85
Haiti                         75
Portugal                      67
Taiwan                        65
Iran                          59
Nicaragua                     49
Greece                        49
Peru                          46
Ecuador                       45
France                        38
Ireland                       37
Thailand                      30
Hong                          30
Cambodia                      28
Trinadad&Tobago               27
Laos                          23
Outlying-US(Guam-USVI-etc)    23
Yugoslavia                    23
Scotland                      21
Honduras                      20
Hungary                       19
Holand-Netherlands             1
Name: count, dtype: int64
```

Problem: "Holand-Netherlands" only occurs once.

- if in test but not training data, then the model will not know what to do with it
- more generally, we can deal with small categories in different manners

We only discuss one option here: we set the parameter `handle_unkown="ignore"`. if the transformation encounters an unknown category, it assigns a 0 to all the one-hot encoded columns.

(like the case above when the "Holand-Netherlands" record is in the test, but not the training data.)

Now we can create the pipeline

```
In [44]: from sklearn.pipeline import make_pipeline
         from sklearn.linear_model import LogisticRegression

         model = make_pipeline(
             OneHotEncoder(handle_unknown="ignore"), LogisticRegression(max_iter=500)
         ) # need more iterations to fully converge
```

```
In [45]: from sklearn.model_selection import cross_validate

         cv_results = cross_validate(model, data_categorical, target)
         cv_results
         # what is it doing under the hood?
         # https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html
         # implicitly, 5-fold cross-validation (meaning 20% of the dataset are held out for prediction, one record held (
```

```
Out[45]: {'fit_time': array([0.36075687, 0.33713675, 0.30161715, 0.3332839 , 0.3268528 ]),
          'score_time': array([0.0138762 , 0.01405883, 0.01498389, 0.01467896, 0.01378369]),
          'test_score': array([0.83222438, 0.83560242, 0.82882883, 0.83312858, 0.83466421])}
```

```
In [46]: scores = cv_results["test_score"]
         print(f"The accuracy is: {scores.mean():.3f} ± {scores.std():.3f}")
```

The accuracy is: 0.833 ± 0.002

We see that this prediction is slightly better than the prediction with the numerical variables we used before.

<mark>Transfer the questions from here to the exercise document and decide how they should be solved</mark>

Exercise M1.04. Categorical data

For the questions, see the exercise document header in exercise document: The impact of using integer encoding for with logistic regression (groups of 2, 15min) [Flavio]"

```
In [47]: # Load dataset -- we've done that already before
         # adult_census = pd.read_csv("../datasets/adult-census.csv")

         # target_name = "clCategoricalass"
         # target = adult_census[target_name]
         # data = adult_census.drop(columns=[target_name, "education-num"])
```

(0) select all columns containing strings

```
In [48]: # keep only columns containing strings (with the `object` dtype)
         # from sklearn.compose import make_column_selector as selector

         categorical_columns_selector = selector(dtype_include=object)
         categorical_columns = categorical_columns_selector(data)
         data_categorical = data[categorical_columns]
```

```
In [ ]:
```

(1) Define scikit-learn pipeline composed of an `OrdinalEncoder` and a `LogisticRegression` classifier.

```
In [49]: # from sklearn.pipeline import make_pipeline
         # from sklearn.preprocessing import OrdinalEncoder
         # from sklearn.linear_model import LogisticRegression

         # Write your code here.
```

```
In [50]: # import numpy as np
         model = make_pipeline(
             OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=-1),
             LogisticRegression(max_iter=500) # need more iterations to fully converge
         )
         # we need to choose values that do not already exist. -1 is an easy solution, since the encoder only uses positi
         # but it does not seem to make any difference to the out-of-sample prediction
```

(2) Evalute the model with cross-validation

```
In [51]: cv_results = cross_validate(model, data_categorical, target) # to see the trace, add error_score="raise"
         scores = cv_results["test_score"]
         print(
             "The mean cross-validation accuracy is: "
             f"{scores.mean():.3f} ± {scores.std():.3f}"
         )
```

The mean cross-validation accuracy is: 0.755 ± 0.002

(3) Compare the generalization performance to a new model where we use `OneHotEncoder` **instead of the** `OrdinalEncoder` **. Compare the score of both models and conclude on the impact of choosing a specific encoding strategy when using a linear model.**

```
In [52]: model = make_pipeline(
             OneHotEncoder(handle_unknown="ignore"), LogisticRegression(max_iter=500) # need more iterations to fully co
         )

         cv_results = cross_validate(model, data_categorical, target)
         scores = cv_results["test_score"]
         print(
             "The mean cross-validation accuracy is: "
             f"{scores.mean():.3f} ± {scores.std():.3f}"
         )
```

The mean cross-validation accuracy is: 0.833 ± 0.002

What do we learn?

- using the `OrdinalEncoder` gives lower test performance: 0.75 instead of 0.83
- but this is still fairly good?
-

Exercise (if time permits) Quiz: categorical and numerical variables

```
In [ ]:
```

```
In [ ]:
```