Preprocessing

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Patrick Laub

Deep learning project life cycle

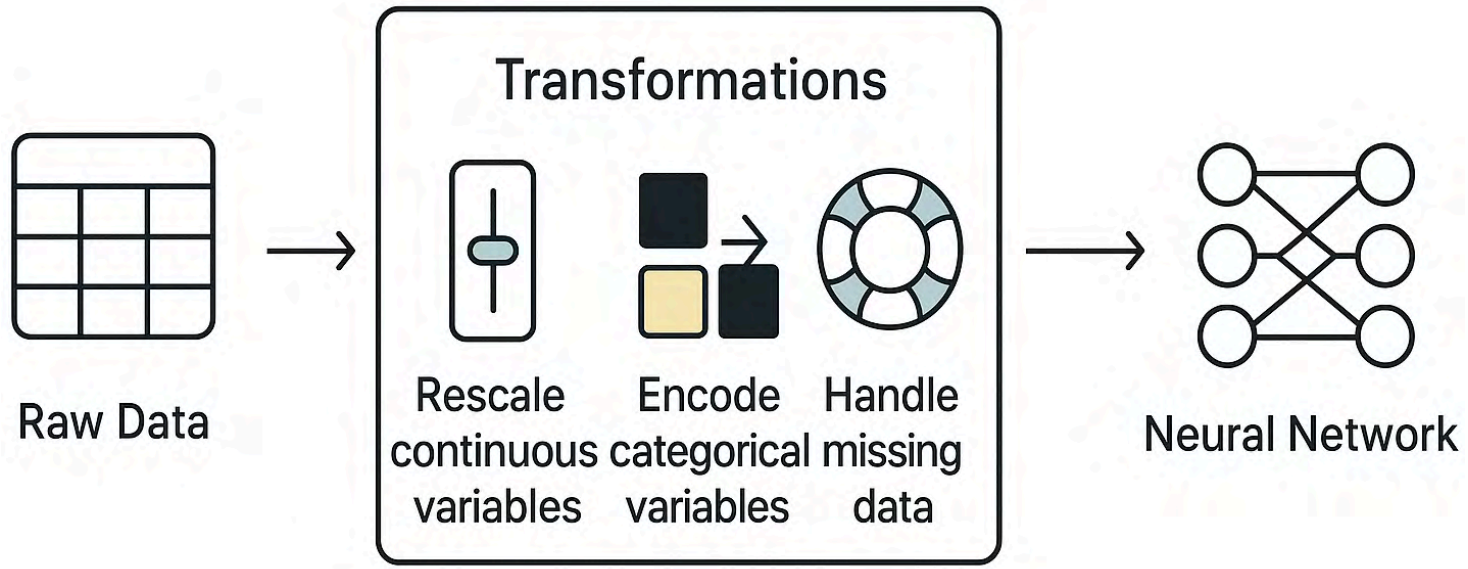
1. Define objectives
2. *Collect & label data*
3. Clean & preprocess data (opt. *feature engineering*)
4. Design network architecture
5. Train and validate model
6. Hyperparameter tuning
7. Test and evaluate performance
8. *Deploy to production*
9. *Monitor and iterate*

(We won't cover the italicised items in this course, though they are very important.)

UNSW SYDNEY

Overview

Every dataset needs to be transformed in various ways before it can be fed into a machine learning model.



Preprocessing refers to the transformations of the raw data before input to the network

Some steps are specific to neural networks (e.g. standardising continuous variables), others are necessary for every model (e.g. encoding categorical variables, handling missing data).

Lecture Outline

- **Example 1: Continuous Variables**
- Example 2: Continuous and Categorical Variables
- Example 3: Continuous and Categorical Variables and Missing Values

UNSW SYDNEY

California housing dataset

```
features, target = sklearn.datasets.fetch_california_housing(
    as_frame=True, return_X_y=True)
features
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population
0	8.3252	41.0	6.984127	1.023810	322.0
1	8.3014	21.0	6.238137	0.971880	2401.0
...	...	...	...	...	...
20638	1.8672	18.0	5.329513	1.171920	741.0
20639	2.3886	16.0	5.254717	1.162264	1387.0

20640 rows × 8 columns

UNSW SYDNEY

All the features were continuous variables

Can look at the data types of the features.

```
1 features.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   MedInc      20640 non-null  float64
 1   HouseAge    20640 non-null  float64
 2   AveRooms    20640 non-null  float64
 3   AveBedrms   20640 non-null  float64
 4   Population  20640 non-null  float64
 5   AveOccup    20640 non-null  float64
 6   Latitude    20640 non-null  float64
 7   Longitude   20640 non-null  float64
dtypes: float64(8)
memory usage: 1.3 MB
```

Note that this isn't foolproof. If you have 'postcode' as a feature, it may be read in as a number when it should be stored as a string and treated as a categorical variable.

UNSW SYDNEY

Splitting into train, validation and test sets

```
X_main, X_test_raw, y_main, y_test = train_test_split(
    features, target, test_size=0.2, random_state=1
)
X_train_raw, X_val_raw, y_train, y_val = train_test_split(
    X_main, y_main, test_size=0.25, random_state=1
)
```

Note that we really cannot allow there to be duplicate rows in the training, validation, or test sets. If there are, we will have data leakage between the sets, which will lead to overfitting and poor generalisation.

Checking for leaks

We can look for rows that appear in multiple of the datasets.

```
train_val_overlap = pd.merge(X_train_raw, X_val_raw, how='inner')
train_test_overlap = pd.merge(X_train_raw, X_test_raw, how='inner')
val_test_overlap = pd.merge(X_val_raw, X_test_raw, how='inner')

print(f"Number of duplicated rows between train and val: {len(train_val_overlap)}")
print(f"Number of duplicated rows between train and test: {len(train_test_overlap)}")
print(f"Number of duplicated rows between val and test: {len(val_test_overlap)}")
```

```
Number of duplicated rows between train and val: 0
Number of duplicated rows between train and test: 0
Number of duplicated rows between val and test: 0
```

An earlier check is to check that there are no duplicate rows in the dataset.

```
features.duplicated().sum()
```

```
np.int64(0)
```

UNSW SYDNEY

We rescaled every column

```
1 scaler = StandardScaler()
2 scaler.fit(X_train_raw)
3
4 X_train = scaler.transform(X_train_raw)
5 X_val = scaler.transform(X_val_raw)
6 X_test = scaler.transform(X_test_raw)
7
8 X_train
```

```
array([[ 0.15, -0.76,  0.26, ... , -0.01, -0.47,  0.69],
       [-1.79, -1.48,  0.48, ... , -0.08, -0.44,  1.33],
       [-0.97, -0.13, -0.67, ... ,  0.05, -0.86,  0.65],
       ... ,
       [ 0.11, -0.84, -0.54, ... , -0.03, -0.81,  0.6 ],
       [-0.82,  0.99, -0.03, ... , -0.03,  0.54, -0.11],
       [ 0.9 , -1.56,  0.66, ... ,  0.07, -0.75,  0.97]])
```

That is because the neural network expects continuous inputs to be normalised to help with the training procedure.

> **Note**
>
> Notice that the output of `X_train` here looks a bit different to `X_train_raw` earlier? We'll touch on this in a couple of slides.

UNSW SYDNEY

Scikit-learn preprocessing methods

- `fit`: learn the parameters of the transformation
- `transform`: apply the transformation
- `fit_transform`: learn the parameters and apply the transformation

`fit` | `fit_transform`

```
1  scaler = StandardScaler()
2  scaler.fit(X_train_raw)
3  X_train = scaler.transform(X_train_raw)
4  X_val = scaler.transform(X_val_raw)
5  X_test = scaler.transform(X_test_raw)
6
7  print(X_train.mean(axis=0))
8  print(X_train.std(axis=0))
9  print(X_val.mean(axis=0))
10 print(X_val.std(axis=0))
```

```
[ 1.11e-16 -1.08e-16 -2.81e-16  3.75e-16 -5.02e-18
-4.36e-17  1.15e-15
  1.75e-15]
[1. 1. 1. 1. 1. 1. 1. 1.]
[ 0.01  0.01  0.   -0.01 -0.    0.01  0.   -0.01]
[1.01 1.   0.81 0.81 0.98 0.84 0.99 1.  ]
```

UNSW SYDNEY

Dataframes & arrays

```
1 X_test_raw.head(3)
```

	MedInc	HouseAge	Ave
4712	3.2500	39.0	4.50
2151	1.9784	37.0	4.98
15927	4.0132	46.0	4.48

```
1 X_test
```

```
array([[-0.33,  0.83, -0.34, ..., -0.11, -0.73,  0.6 ],
       [-1.  ,  0.67, -0.16, ..., -0.04,  0.54, -0.1 ],
       [ 0.07,  1.38, -0.35, ...,  0.06,  0.98, -1.42],
       ...,
       [ 0.62, -0.21, -0.16, ...,  0.05,  0.92, -1.4 ],
       [ 0.53,  1.07, -0.03, ..., -0.  , -0.72,  0.73],
       [-0.62,  1.86,  0.11, ..., -0.02, -0.77,  1.09]])
```

> **Note**
>
> By default, when you pass `sklearn` a DataFrame it returns a `numpy` array.

UNSW SYDNEY

Keep as a DataFrame

From scikit-learn 1.2:

```
from sklearn import set_config
set_config(transform_output="pandas")

imp = SimpleImputer()

X_train = imp.fit_transform(X_train_raw)
X_val = imp.transform(X_val_raw)
X_test = imp.transform(X_test_raw)
```

```
X_test
```

	MedInc	HouseAge
4712	3.2500	39.0
2151	1.9784	37.0
…	…	…
6823	4.8750	42.0
11878	2.7054	52.0

4128 rows × 8 columns

From SimpleImputer's docs:

> Replace missing values using a descriptive statistic (e.g. mean, median, or most frequent) along each column, or using a constant value… default='mean'

UNSW SYDNEY

Lecture Outline

- Example 1: Continuous Variables
- **Example 2: Continuous and Categorical Variables**
- Example 3: Continuous and Categorical Variables and Missing Values

UNSW SYDNEY

French motor dataset

```python
 # Download the dataset if we don't have it already.
if not Path("french-motor.csv").exists():
    freq = sklearn.datasets.fetch_openml(data_id=41214, as_frame=True).frame
    freq.to_csv("french-motor.csv", index=False)
else:
    freq = pd.read_csv("french-motor.csv")
freq.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 678013 entries, 0 to 678012
Data columns (total 12 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   IDpol        678013 non-null  float64
 1   ClaimNb      678013 non-null  float64
 2   Exposure     678013 non-null  float64
 3   Area         678013 non-null  object
 4   VehPower     678013 non-null  float64
 5   VehAge       678013 non-null  float64
 6   DrivAge      678013 non-null  float64
 7   BonusMalus   678013 non-null  float64
 8   VehBrand     678013 non-null  object
 9   VehGas       678013 non-null  object
 10  Density      678013 non-null  float64
 11  Region       678013 non-null  object
dtypes: float64(8), object(4)
```

UNSW SYDNEY

The data

```
1 freq
```

	IDpol	ClaimNb	Exposure	Area	VehPower	VehA
0	1.0	1.0	0.10000	D	5.0	0.0
1	3.0	1.0	0.77000	D	5.0	0.0
2	5.0	1.0	0.75000	B	6.0	2.0
...	...	...	...	...	...	...
678010	6114328.0	0.0	0.00274	D	6.0	2.0
678011	6114329.0	0.0	0.00274	B	4.0	0.0
678012	6114330.0	0.0	0.00274	B	7.0	6.0

678013 rows × 12 columns

UNSW SYDNEY

Data dictionary

- `IDpol`: policy number (unique identifier)
- `ClaimNb`: number of claims on the given policy
- `Exposure`: total exposure in yearly units
- `Area`: area code (categorical, ordinal)
- `VehPower`: power of the car (categorical, ordinal)
- `VehAge`: age of the car in years
- `DrivAge`: age of the (most common) driver in years
- `BonusMalus`: bonus-malus level between 50 and 230 (with reference level 100)
- `VehBrand`: car brand (categorical, nominal)
- `VehGas`: diesel or regular fuel car (binary)
- `Density`: of inhabitants per km^2 in the city of the living place of the driver
- `Region`: regions in France (prior to 2016)

We have $\{(\mathbf{x}_i, y_i)\}_{i=1,\dots,n}$ for $\mathbf{x}_i \in \mathbb{R}^p$ and $y_i \in \mathbb{N}_0$. Assume the distribution

$$Y_i \sim \mathsf{Poisson}(\lambda(\mathbf{x}_i))$$

We have $\mathbb{E}Y_i = \lambda(\mathbf{x}_i)$. The NN takes $\mathbf{x}_i$ & predicts $\mathbb{E}Y_i$.

```
freq = freq.drop("IDpol", axis=1)
```

Source: Nell et al. (2020), Case Study: French Motor Third-Party Liability Claims, SSRN.

Split the data

```
X_train_raw, X_test_raw, y_train, y_test = train_test_split(
  freq.drop("ClaimNb", axis=1), freq["ClaimNb"], random_state=2023)
X_train_raw = X_train_raw.reset_index(drop=True)
X_test_raw = X_test_raw.reset_index(drop=True)
X_train_raw
```

	Exposure	Area	VehPower	VehAge	DrivAge	BonusM
0	0.01	A	4.0	0.0	70.0	50.0
1	1.00	C	6.0	11.0	36.0	50.0
2	0.08	D	7.0	9.0	35.0	50.0
...	...	...	...	...	...	...
508506	0.22	E	7.0	21.0	32.0	90.0
508507	1.00	C	7.0	15.0	51.0	50.0
508508	0.42	D	6.0	1.0	37.0	54.0

508509 rows × 10 columns

UNSW SYDNEY

What values do we see in the data?

```
X_train_raw["Area"].value_counts()
X_train_raw["VehBrand"].value_counts()
X_train_raw["VehGas"].value_counts()
X_train_raw["Region"].value_counts()
```

```
Area
C    144156
D    113532
E    102791
A     78017
B     56571
F     13442
Name: count, dtype: int64
```

```
VehGas
Regular    259422
Diesel     249087
Name: count, dtype: int64
```

```
VehBrand
B12    124490
B1     122200
B2     120155
        ...
B11     10218
B13      9160
B14      2998
Name: count, Length: 11, dtype: int64
```

```
Region
R24    120597
R82     63555
R93     59627
        ...
R21      2283
R42      1634
R43      1024
Name: count, Length: 22, dtype: int64
```

UNSW SYDNEY

Nominal categorical variables

These are categorical variables which you cannot order in a 'default' sensible way.

```
from sklearn.preprocessing import OneHotEncoder
gas = X_train_raw[["VehGas"]]
```

The original data:

```
gas.head()
```

	VehGas
0	Regular
1	Regular
2	Diesel
3	Regular
4	Diesel

One-hot encoding

```
enc = OneHotEncoder(sparse_output=False)

X_train = enc.fit_transform(gas)
X_train.head()
```

	VehGas_Diesel	VehGas_Regular
0	0.0	1.0
1	0.0	1.0
2	1.0	0.0
3	0.0	1.0
4	1.0	0.0

Dummy encoding

```
enc = OneHotEncoder(
    drop="first",
    sparse_output=False)
X_train = enc.fit_transform(gas
X_train.head()
```

	VehGas_Regular
0	1.0
1	1.0
2	0.0
3	1.0
4	0.0

⚠ Warning

Collinearity is not a big deal for neural networks. However if you want to reuse the same `X_train` data for a (generalised) linear regression, then you probably want to dummy-encode here to avoid needing two different preprocessing regimes for the two competing models.

UNSW SYDNEY

Ordinal categorical variables

These are categorical variables which have a natural order to them.

```
1 from sklearn.preprocessing import OrdinalEncoder
2 enc = OrdinalEncoder()
3 enc.fit(X_train_raw[["Area"]])
4 enc.categories_
```

```
[array(['A', 'B', 'C', 'D', 'E', 'F'], dtype=object)]
```

```
1 for i, area in enumerate(enc.categories_[0]):
2     print(f"The Area value {area} gets turned into {i}.")
```

```
The Area value A gets turned into 0.
The Area value B gets turned into 1.
The Area value C gets turned into 2.
The Area value D gets turned into 3.
The Area value E gets turned into 4.
The Area value F gets turned into 5.
```

In other words,

```
1 print(" < ".join(enc.categories_[0]) + ", and ")
2 print(" = ".join(f"{r} - {l}" for l, r in zip(enc.categories_[0][:-1], enc.categories_[0
```

```
A < B < C < D < E < F, and
B - A = C - B = D - C = E - D = F - E
```

UNSW SYDNEY

Ordinal encoded values

```
1 X_train = enc.transform(X_train_raw[["Area"]])
2 X_test = enc.transform(X_test_raw[["Area"]])
```

```
1 X_train_raw[["Area"]].head()
```

	Area
0	A
1	C
2	D
3	E
4	B

```
1 X_train.head()
```

	Area
0	0.0
1	2.0
2	3.0
3	4.0
4	1.0

We could train on this `X_train`, or on the previous `X_train`, but each only contains one feature from the dataset. How do we add the continuous variables back in? Use a sklearn *column transformer* for that.

UNSW SYDNEY

Transform each column in one hit

```
from sklearn.compose import make_column_transformer

ct = make_column_transformer(
  (OneHotEncoder(sparse_output=False, drop="first"), ["VehGas", "VehBrand", "Region"]),
  (OrdinalEncoder(), ["Area"]),
  remainder=StandardScaler()
)

X_train = ct.fit_transform(X_train_raw)
```

```
X_train_raw
```

	Exposure	Area	VehP
0	0.01	A	4.0
...	...	...	...
508508	0.42	D	6.0

508509 rows × 10 columns

```
X_train
```

	onehotencoder__VehGa
0	1.0
...	...
508508	0.0

508509 rows × 39 columns

UNSW SYDNEY

Transform each column in one hit II

```
from sklearn.compose import make_column_transformer

ct = make_column_transformer(
  (OneHotEncoder(sparse_output=False, drop="first"), ["VehGas", "VehBrand", "Region"]),
  (OrdinalEncoder(), ["Area"]),
  ("drop", ["VehBrand", "Region"]),
  remainder=StandardScaler(),
  verbose_feature_names_out=False
)
X_train = ct.fit_transform(X_train_raw)
```

```
X_train_raw
```

	Exposure	Area	VehP
0	0.01	A	4.0
...	...	...	...
508508	0.42	D	6.0

508509 rows × 10 columns

```
X_train
```

	VehGas_Regular	VehE
0	1.0	0.0
...	...	...
508508	0.0	0.0

508509 rows × 39 columns

UNSW SYDNEY

Aside: The order of the variables matters

```
X_train_raw["Area_In_Words"] = X_train_raw["Area"].map({
    "A": "Very low", "B": "Low", "C": "Medium",
    "D": "High", "E": "Very high", "F": "Extremely high"
})
```

```
enc_wrong = OrdinalEncoder()
enc_wrong.fit(X_train_raw[["Area_In_Words"]])
enc_wrong.categories_
```

```
[array(['Extremely high', 'High', 'Low', 'Medium', 'Very high', 'Very low'],
       dtype=object)]
```

```
for i, area in enumerate(enc_wrong.categories_[0]):
    print(f"The Area value {area} gets turned into {i}.")
```

```
The Area value Extremely high gets turned into 0.
The Area value High gets turned into 1.
The Area value Low gets turned into 2.
The Area value Medium gets turned into 3.
The Area value Very high gets turned into 4.
The Area value Very low gets turned into 5.
```

In other words,

```
print(" < ".join(enc_wrong.categories_[0]))
```

```
Extremely high < High < Low < Medium < Very high < Very low
```

UNSW SYDNEY

Aside: Manually set the order

```
ordered = [["Very low", "Low", "Medium", "High", "Very high", "Extremely high"]]
enc_right = OrdinalEncoder(categories=ordered)
enc_right.fit(X_train_raw[["Area_In_Words"]])
enc_right.categories_
```

```
[array(['Very low', 'Low', 'Medium', 'High', 'Very high', 'Extremely high'],
       dtype=object)]
```

```
for i, area in enumerate(enc_right.categories_[0]):
    print(f"The Area value {area} gets turned into {i}.")
```

```
The Area value Very low gets turned into 0.
The Area value Low gets turned into 1.
The Area value Medium gets turned into 2.
The Area value High gets turned into 3.
The Area value Very high gets turned into 4.
The Area value Extremely high gets turned into 5.
```

In other words,

```
print(" < ".join(enc_right.categories_[0]))
```

```
Very low < Low < Medium < High < Very high < Extremely high
```

```
X_train_raw = X_train_raw.drop("Area_In_Words", axis=1)
```

UNSW SYDNEY

Lecture Outline

- Example 1: Continuous Variables
- Example 2: Continuous and Categorical Variables
- **Example 3: Continuous and Categorical Variables and Missing Values**

UNSW SYDNEY

Stroke prediction dataset

Dataset source: Kaggle Stroke Prediction Dataset.

```
RAW_DATA_DIR = Path("stroke/raw")
data = pd.read_csv(RAW_DATA_DIR / "stroke.csv")
data.head()
```

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residenc
0	9046	Male	67.0	0	1	Yes	Private	Urban
1	51676	Female	61.0	0	0	Yes	Self-employed	Rural
2	31112	Male	80.0	0	1	Yes	Private	Rural
3	60182	Female	49.0	0	0	Yes	Private	Urban
4	1665	Female	79.0	1	0	Yes	Self-employed	Rural

Data description

1. `id`: unique identifier
2. `gender`: “Male”, “Female” or “Other”
3. `age`: age of the patient
4. `hypertension`: 0 or 1 if the patient has hypertension
5. `heart_disease`: 0 or 1 if the patient has any heart disease
6. `ever_married`: “No” or “Yes”
7. `work_type`: “children”, “Govt_job”, “Never_worked”, “Private” or “Self-employed”
8. `Residence_type`: “Rural” or “Urban”
9. `avg_glucose_level`: average glucose level in blood
10. `bmi`: body mass index
11. `smoking_status`: “formerly smoked”, “never smoked”, “smokes” or “Unknown”
12. `stroke`: 0 or 1 if the patient had a stroke

Source: Kaggle, Stroke Prediction Dataset.

Look for missing values and split the data

First, look for missing values.

```
number_missing = data.isna().sum()
number_missing[number_missing > 0]
```

```
bmi    201
dtype: int64
```

> **Tip**
>
> Always take a look at the missing data in, say, Excel. Sometimes pandas reads in a category like "None" and interprets that as a missing value, when it's really a valid category. This happened to me just the other day.

```
features = data.drop(["id", "stroke"], axis=1)
target = data["stroke"]

X_main_raw, X_test_raw, y_main, y_test = train_test_split(
    features, target, test_size=0.2, random_state=7)
X_train_raw, X_val_raw, y_train, y_val = train_test_split(
    X_main_raw, y_main, test_size=0.25, random_state=12)

X_train_raw.shape, X_val_raw.shape, X_test_raw.shape
```

```
((3066, 10), (1022, 10), (1022, 10))
```

UNSW SYDNEY

What values do we see in the data?

```
1 X_train_raw["gender"].value_counts()
```

```
gender
Female    1802
Male      1264
Name: count, dtype: int64
```

```
1 X_train_raw["ever_married"].value_counts
```

```
ever_married
Yes    2007
No     1059
Name: count, dtype: int64
```

```
1 X_train_raw["Residence_type"].value_coun
```

```
Residence_type
Urban    1536
Rural    1530
Name: count, dtype: int64
```

```
1 X_train_raw["work_type"].value_counts()
```

```
work_type
Private          1754
Self-employed     490
children          419
Govt_job          390
Never_worked       13
Name: count, dtype: int64
```

```
1 X_train_raw["smoking_status"].value_coun
```

```
smoking_status
never smoked       1130
Unknown             944
formerly smoked     522
smokes              470
Name: count, dtype: int64
```

Tip

Look for sparse categories. One common trick is to lump together all the small categories (e.g. < 5% or 10% each) into a new combined 'OTHER' or 'RARE' category.

UNSW SYDNEY

Make a plan for preprocessing each column

1. Take categorical columns ↪ one-hot vectors
2. binary columns ↪ do nothing (already 0 & 1)
3. continuous columns ↪ impute NaNs & standardise.

Make the column transformer

```python
from sklearn.pipeline import make_pipeline

nom_vars =  ["gender", "ever_married", "Residence_type",
    "work_type", "smoking_status"]

ct = make_column_transformer(
  (OneHotEncoder(sparse_output=False, handle_unknown="ignore"), nom_vars),
  ("passthrough", ["hypertension", "heart_disease"]),
  remainder=make_pipeline(SimpleImputer(), StandardScaler()),
  verbose_feature_names_out=False
)

X_train = ct.fit_transform(X_train_raw)
X_val = ct.transform(X_val_raw)
X_test = ct.transform(X_test_raw)

for name, X in zip(("train", "val", "test"), (X_train, X_val, X_test)):
    num_na = pd.DataFrame(X).isna().sum().sum()
    print(f"The {name} set has shape {X.shape} & with {num_na} NAs.")
```

```
The train set has shape (3066, 20) & with 0 NAs.
The val set has shape (1022, 20) & with 0 NAs.
The test set has shape (1022, 20) & with 0 NAs.
```

UNSW SYDNEY

Handling unseen categories

```
1 X_train_raw["gender"].value_counts()
```

```
gender
Female    1802
Male      1264
Name: count, dtype: int64
```

```
1 X_val_raw["gender"].value_counts()
```

```
gender
Female    615
Male      406
Other       1
Name: count, dtype: int64
```

```
1 ind = np.argmax(X_val_raw["gender"] == "
2 X_val_raw.iloc[ind-1:ind+3][["gender"]]
```

	gender
4970	Male
3116	Other
4140	Male
2505	Female

```
1 gender_cols = pd.DataFrame(X_val)[["gend
2 gender_cols.iloc[ind-1:ind+3]
```

	gender_Female	gender_
4970	0.0	1.0
3116	0.0	0.0
4140	0.0	1.0
2505	1.0	0.0

UNSW SYDNEY

Handling unseen categories II

This was caused by fitting the encoder on the `X_train` which, by bad luck, didn't have the "Other" category. Why not just read the entire dataset before splitting to collect all the values the categorical variables could take?

It's a very mild form of data leakage, though it may be justifiable in some cases. If we made the datasets, we'd have a better idea of whether it makes sense on a case-by-case basis. We struggle here as we (in this course) are just analysing datasets which we didn't collect.

Save the preprocessed data to files

```
PROCESSED_DATA_DIR = Path("stroke/processed")
PROCESSED_DATA_DIR.mkdir(parents=True, exist_ok=True)

X_train.to_csv(PROCESSED_DATA_DIR / "x_train.csv", index=False)
X_val.to_csv(PROCESSED_DATA_DIR / "x_val.csv", index=False)
X_test.to_csv(PROCESSED_DATA_DIR / "x_test.csv", index=False)
y_train.to_csv(PROCESSED_DATA_DIR / "y_train.csv", index=False)
y_val.to_csv(PROCESSED_DATA_DIR / "y_val.csv", index=False)
y_test.to_csv(PROCESSED_DATA_DIR / "y_test.csv", index=False)
```

UNSW SYDNEY

Lastly, keep the preprocessor around

Later want to make a prediction on some new data point. It has to go through the **exact same** preprocessing steps.

```
import joblib
joblib.dump(ct, PROCESSED_DATA_DIR / "preprocessor.pkl")
preprocessor = joblib.load(PROCESSED_DATA_DIR / "preprocessor.pkl")
```

```
X_test_raw.head(1)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_ty
2804	Female	69.0	0	0	Yes	Govt_job	Rural

```
new_data = X_test_raw.head(1).copy()
new_data["ever_married"] = "No" # Let's pretend this is the new data point
```

```
ct.transform(new_data)
```

	gender_Female	gender_Male	ever_married_No	ever_married_Yes	Residence_type
2804	1.0	0.0	1.0	0.0	1.0

```
preprocessor.transform(new_data)
```

	gender_Female	gender_Male	ever_married_No	ever_married_Yes	Residence_type
2804	1.0	0.0	1.0	0.0	1.0

UNSW SYDNEY

Glossary

- column transformer
- nominal variables
- ordinal variables
- one-hot encoding
- missing values
- imputation
- standardisation