Welcome to Supervised Learning

Part 2: How to prepare your data for supervised machine learning

Instructor: Andras Zsom

[https://github.com/azsom/Supervised-Learning (https://github.com/azsom/Supervised-Learning)](https://github.com/azsom/Supervised-Learning)

The topic of the course series: supervised Machine Learning (ML)

- how to build an ML pipeline from beginning to deployment
- we assume you already performed data cleaning
- this is the first course out of 6 courses
  - Part 1: Introduction to machine learning and the bias-variance tradeoff
  - **Part 2: How to prepare your data for supervised machine learning**
  - Part 3: Evaluation metrics in supervised machine learning
  - Part 4: SVMs, Random Forests, XGBoost
  - Part 5: Missing data in supervised ML
  - Part 6: Interpretability
- you can complete the courses in sequence or complete individual courses based on your interest

Structured data

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**

We focus on the feature matrix (X) in this course.

Learning objectives of this course

By the end of the course, you will be able to

- describe why data splitting is necessary in machine learning
- summarize the properties of IID data
- list examples of non-IID datasets
- apply IID splitting techniques
- apply non-IID splitting techniques
- identify when a custom splitting strategy is necessary
- describe the two motivating concepts behind preprocessing
- apply various preprocessors to categorical and continuous features
- perform preprocessing with a sklearn pipeline and ColumnTransformer

Module 1: Split IID data

Learning objectives of this module:

- describe why data splitting is necessary in machine learning
- summarize the properties of IID data
- apply IID splitting techniques

Why do we split the data?

- we want to find the best hyper-parameters of our ML algorithms
  - fit models to training data
  - evaluate each model on validation set
  - we find hyper-parameter values that optimize the validation score
- we want to know how the model will perform on previously unseen data - the generalization error
  - apply our final model on the test set

We need to split the data into three parts!

Ask yourself these questions!

- What is the intended use of the model? What is it supposed to do/predict?
- What data/info do you have available at the time of prediction?
- Your split must mimic the intended use of the model only then will you accurately estimate how well the model will perform on previously unseen points (generalization error).
- two examples:
  - if you want to predict the outcome of a new patient's visit to the ER:
    - your test score must be based on patients not included in training and validation
    - your validation score must be based on patients not included in training

        - points of one patient should not be distributed over multiple sets because your generalization error will be off
    - a youtube video was released 4 weeks ago and you want to predict if it will be featured a week from now, your training data should only contain info that will be available when you make predictions (stuff you know 4 weeks after release)
        - split data based on youtube vid ID
        - use info that's available 4 weeks after release
        - your classification label will be whether it was featured or not 5 weeks after release

How should we split the data into train/validation/test?

- data is **Independent and Identically Distributed** (iid)
    - all samples stem from the same generative process and the generative process is assumed to have no memory of past generated samples
    - identify cats and dogs on images
    - predict the house price
    - predict if someone's salary is above or below 50k
- examples of not iid data:
    - data generated by time-dependent processes
    - data has group structure (samples collected from e.g., different subjects, experiments, measurement devices)

Splitting strategies for iid data: basic approach

- 60% train, 20% validation, 20% test for small datasets
- 98% train, 1% validation, 1% test for large datasets
    - if you have 1 million points, you still have 10000 points in validation and test which is plenty to assess model performance

Let's work with the adult data!

https://archive.ics.uci.edu/ml/datasets/adult (https://archive.ics.uci.edu/ml/datasets/adult)

In [ ]:
```python
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 th
X = df.loc[:, df.columns != 'gross-income'] # all other columns are fea
print(y)
print(X.head())

```

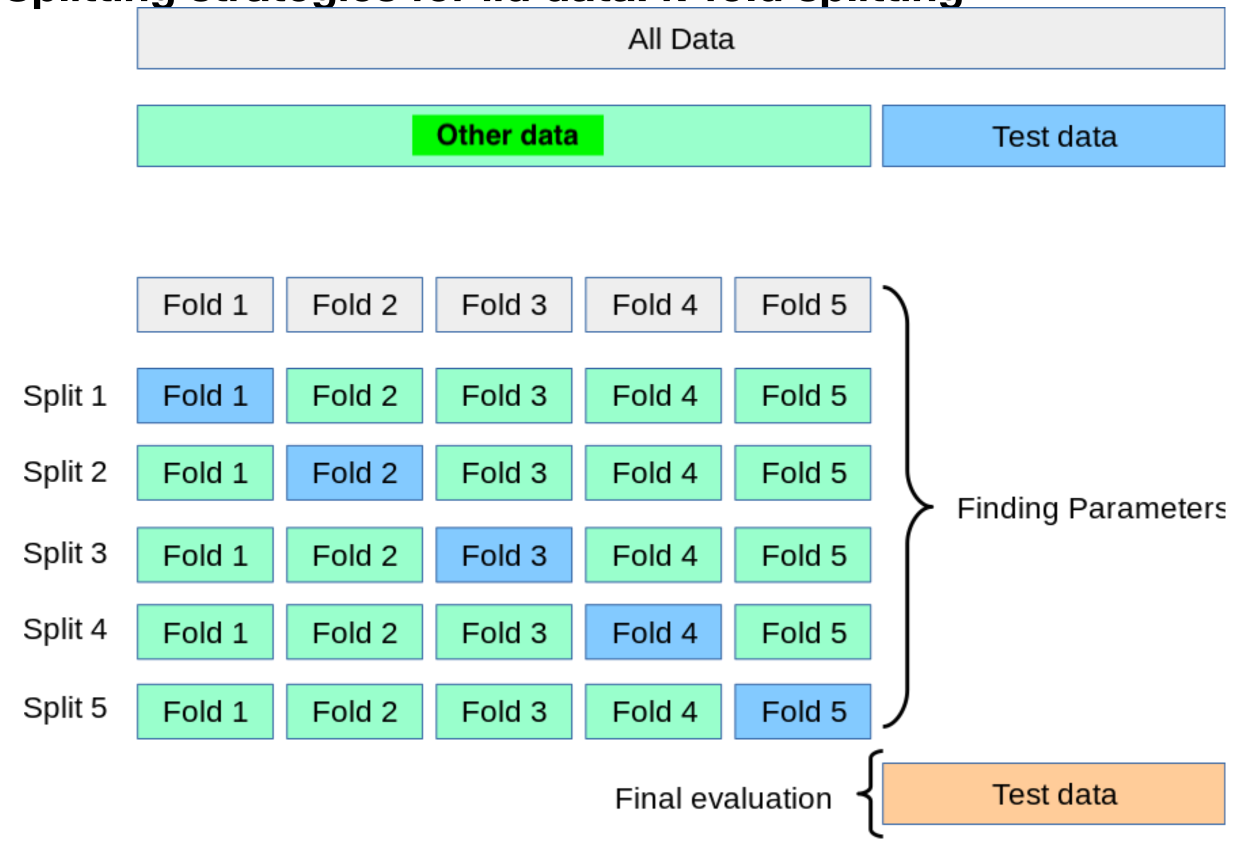
In [ ]:
```python
help(train_test_split)
```

In [ ]:
```python
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 =
print('training set:',X_train.shape, y_train.shape) # 60% of points are
print(X_other.shape, y_other.shape) # 40% of points are in other

# 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_s
print('validation set:',X_val.shape, y_val.shape) # 20% of points are i
print('test set:',X_test.shape, y_test.shape) # 20% of points are in te
```

Randomness due to splitting

- the model performance, validation and test scores will change depending on which points are in train, val, test
  - inherent randomness or uncertainty of the ML pipeline
- change the random state a couple of times and repeat the whole ML pipeline to assess how much the random splitting affects your test score
  - you would expect a similar uncertainty when the model is deployed

Splitting strategies for iid data: k-fold splitting



In [ ]:
```python
from sklearn.model_selection import KFold
help(KFold)
```

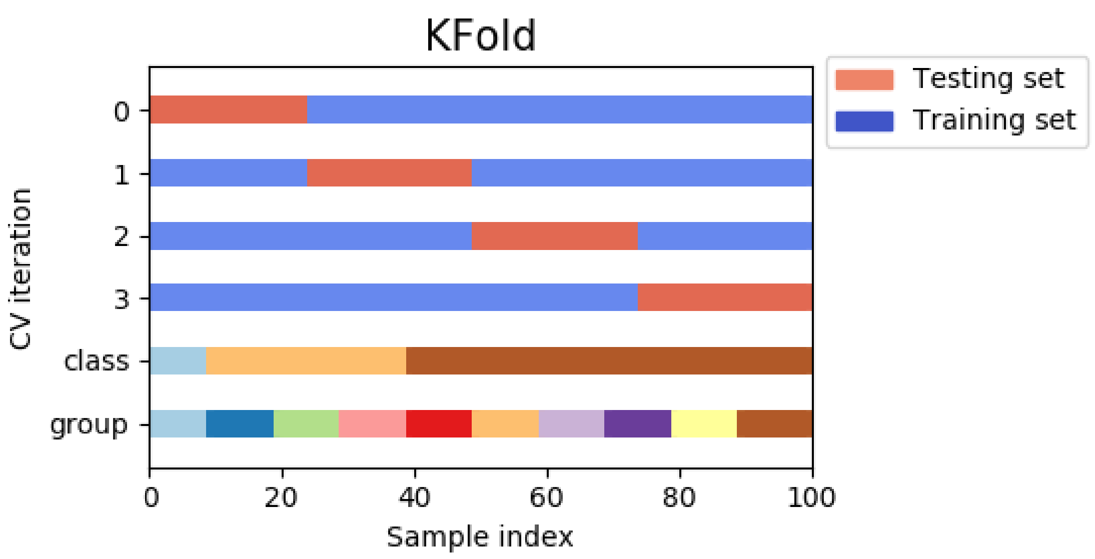
In [ ]:
```python
random_state = 42

# first split to separate out the test set
X_other, X_test, y_other, y_test = train_test_split(X,y,test_size = 0.2
print(X_other.shape,y_other.shape)
print('test set:',X_test.shape,y_test.shape)

# do KFold split on other
kf = KFold(n_splits=5,shuffle=True,random_state=random_state)
for train_index, val_index in kf.split(X_other,y_other):
    X_train = X_other.iloc[train_index]
    y_train = y_other.iloc[train_index]
    X_val = X_other.iloc[val_index]
    y_val = y_other.iloc[val_index]
    print('   training set:',X_train.shape, y_train.shape)
    print('   validation set:',X_val.shape, y_val.shape)
    # the validation set contains different points in each iteration
    print(X_val[['age','workclass','education']].head())

```

How many splits should I create?

- tough question, 3-5 is most common
- if you do n splits, n models will be trained, so the larger the n, the most computationally intensive it will be to train the models
- KFold is usually better suited for small datasets
- KFold is good to estimate uncertainty due to random splitting of train and val, but it is not perfect
  - the test set remains the same

Why shuffling iid data is important?

- by default, data is not shuffled by Kfold which can introduce errors!



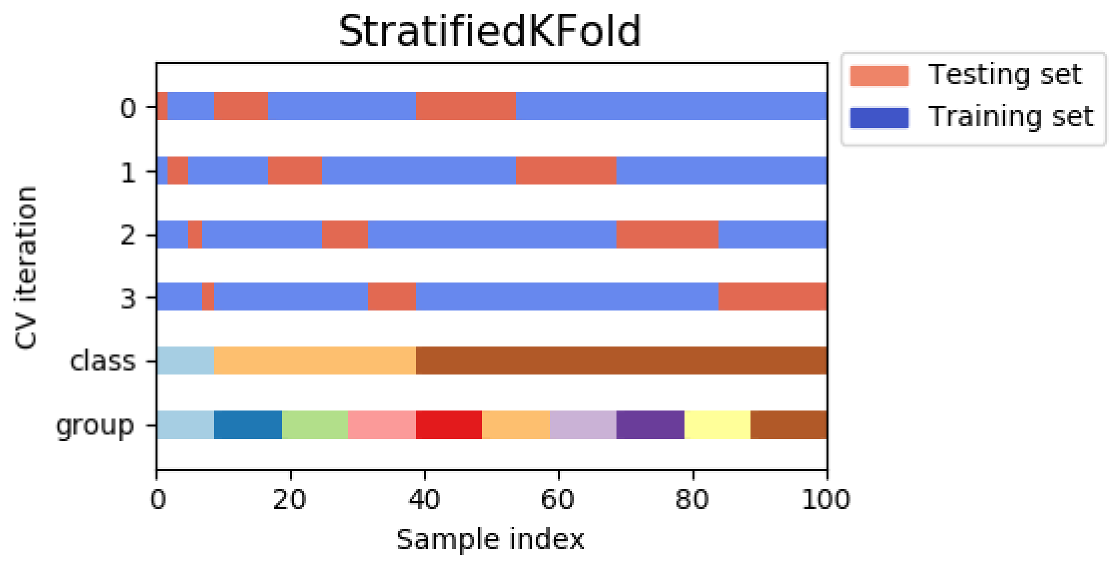
Imbalanced data

- imbalanced data: only a small fraction of the points are in one of the classes, usually ~5% or less but there is no hard limit here
- examples:
  - people visit a bank's website. do they sign up for a new credit card?
    - most customers just browse and leave the page
    - usually 1% or less of the customers get a credit card (class 1), the rest leaves the page without signing up (class 0).
  - fraud detection
    - only a tiny fraction of credit card payments are fraudulent
  - rare disease diagnosis
- the issue with imbalanced data:
  - if you apply train_test_split or KFold, you might not have class 1 points in one of your sets by chance
  - this is what we need to fix

Solution: stratified splits

In [ ]:
```python
random_state = 42

X_train, X_other, y_train, y_other = train_test_split(X,y,train_size =
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_s

print('**balance without stratification:**')
# a variation on the order of 1% which would be too much for imbalanced
print(y_train.value_counts(normalize=True))
print(y_val.value_counts(normalize=True))
print(y_test.value_counts(normalize=True))

X_train, X_other, y_train, y_other = train_test_split(X,y,train_size =
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_s
print('**balance with stratification:**')
# very little variation (in the 4th decimal point only) which is import
print(y_train.value_counts(normalize=True))
print(y_val.value_counts(normalize=True))
print(y_test.value_counts(normalize=True))
```

Stratified folds



In [ ]:
```python
from sklearn.model_selection import StratifiedKFold
help(StratifiedKFold)
```

In [ ]:
```python
# what we did before: variance in balance on the order of 1%
random_state = 42

X_other, X_test, y_other, y_test = train_test_split(X,y,test_size = 0.2
print('test balance:',y_test.value_counts(normalize=True))

# do KFold split on other
kf = KFold(n_splits=5,shuffle=True,random_state=random_state)
for train_index, val_index in kf.split(X_other,y_other):
    X_train = X_other.iloc[train_index]
    y_train = y_other.iloc[train_index]
    X_val = X_other.iloc[val_index]
    y_val = y_other.iloc[val_index]
    print('train balance:')
    print(y_train.value_counts(normalize=True))
    print('val balance:')
    print(y_val.value_counts(normalize=True))
```

In [ ]:
```python
# stratified K Fold: variation in balance is very small (4th decimal pc
random_state = 42

# stratified train-test split
X_other, X_test, y_other, y_test = train_test_split(X,y,test_size = 0.2
print('test balance:',y_test.value_counts(normalize=True))

# do StratifiedKFold split on other
kf = StratifiedKFold(n_splits=5,shuffle=True,random_state=random_state)
for train_index, val_index in kf.split(X_other,y_other):
    X_train = X_other.iloc[train_index]
    y_train = y_other.iloc[train_index]
    X_val = X_other.iloc[val_index]
    y_val = y_other.iloc[val_index]
    print('train balance:')
    print(y_train.value_counts(normalize=True))
    print('val balance:')
    print(y_val.value_counts(normalize=True))
```

Module 2: Split non-IID data

Learning objectives of this module:

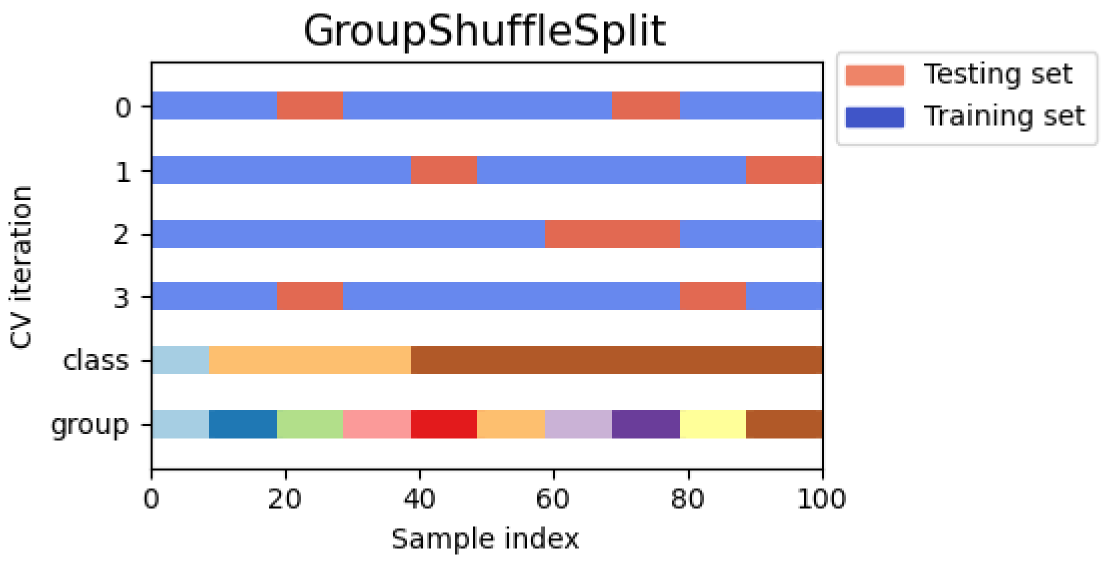
- list examples of non-IID datasets
- apply non-IID splitting techniques
- identify when a custom splitting strategy is necessary

Examples of non-iid data

- if there is any sort of time or group structure in your data, it is likely non-iid
  - group structure:
    - each point is someone's visit to the ER and some people visited the ER multiple times

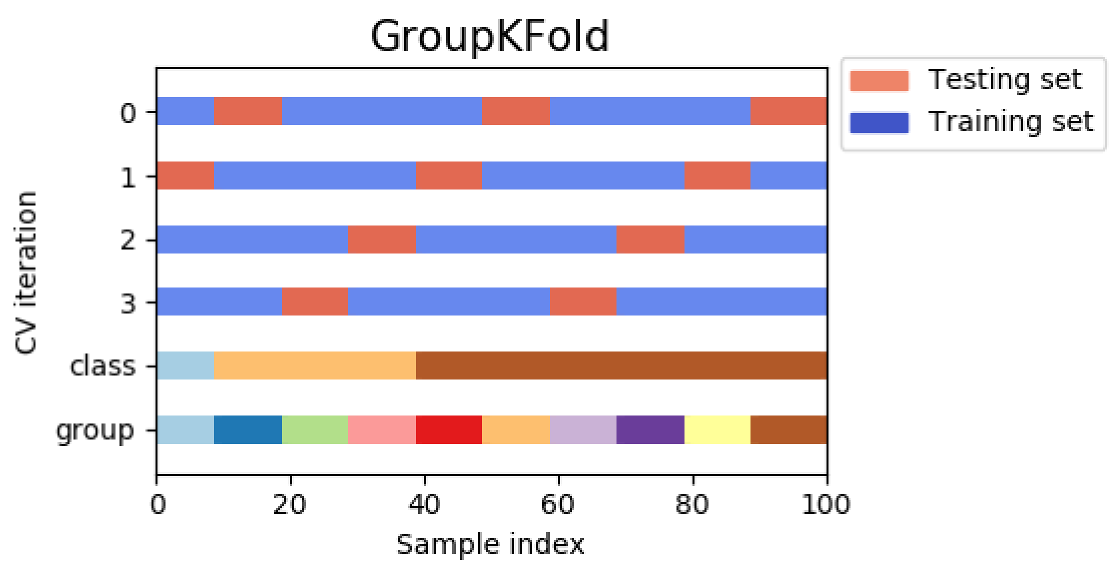
        - each point is stats of a youtube video and the stats are collected weekly, one of the stats is whether it is featured
        - each point is a customer's visit to CVS and customers tend to return regularly
    - time structure
        - each point is the stocks price at a given time
        - eahc point is a person's health or activity status

Group-based split: GroupShuffleSplit



In [ ]:
```python
import numpy as np
from sklearn.model_selection import GroupShuffleSplit
X = np.ones(shape=(8, 2))
y = np.ones(shape=(8, 1))
groups = np.array([1, 1, 2, 2, 2, 3, 3, 3])

gss = GroupShuffleSplit(n_splits=10, train_size=.8, random_state=42)

for train_idx, test_idx in gss.split(X, y, groups):
    print("TRAIN:", train_idx, "TEST:", test_idx)

```

Group-based split: GroupKFold

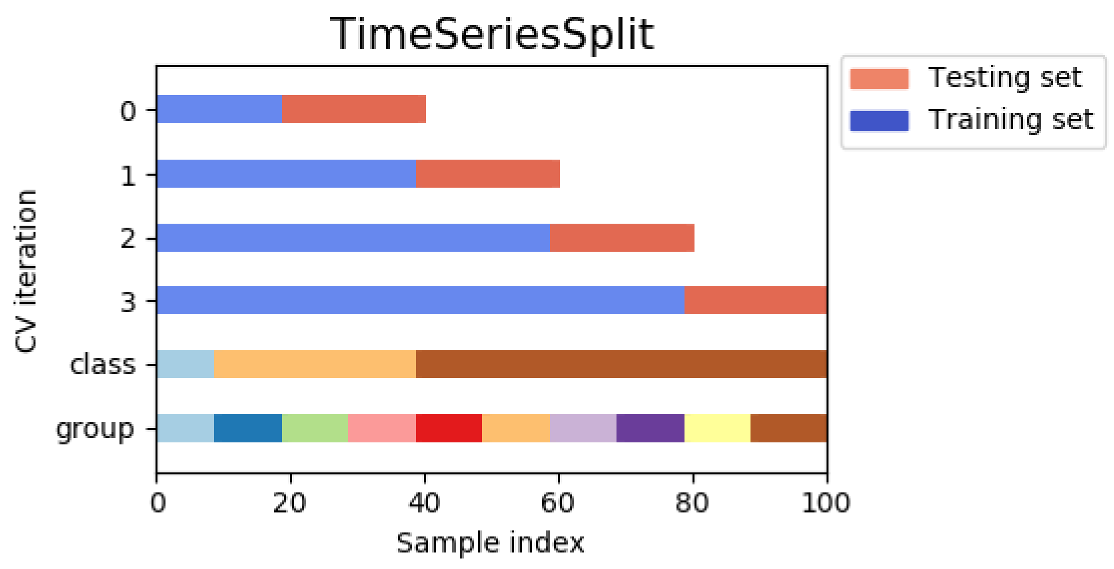


In [ ]:
```python
from sklearn.model_selection import GroupKFold

group_kfold = GroupKFold(n_splits=3)

for train_index, test_index in group_kfold.split(X, y, groups):
    print("TRAIN:", train_index, "TEST:", test_index)

```

In [ ]:
```python
help(GroupKFold)
```

Data leakage in time series data is similar!

- do NOT use information in validation or test which will not be available once your model is deployed
  - don't use future information!



In [ ]:
```python
import numpy as np
from sklearn.model_selection import TimeSeriesSplit
X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
y = np.array([1, 2, 3, 4, 5, 6])
tscv = TimeSeriesSplit()
for train_index, test_index in tscv.split(X):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

```

Module 3: Preprocess continuous and categorical features

Learning objectives of this module:

- describe the two motivating concepts behind preprocessing
- apply various preprocessors to categorical and continuous features
- perform preprocessing with a sklearn pipeline and ColumnTransformer

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

- ML works with numerical data but some columns can be text (e.g., home country, educational level, gender, race)
  - some ML algorithms accept (and prefer) a non-numerical feature matrix (like [CatBoost (https://catboost.ai/)](https://catboost.ai/)) but that's not standard
  - sklearn throws an error message if the feature matrix contains non-numerical elements
- the order of magnitude of numerical features can vary greatly which is not good for most ML algorithms (e.g., salary in USD, age in years, time spent on the site in sec)
  - many ML algorithms are distance-based and they perform better and converge faster if the features are standardized (features have a mean of 0 and the same standard deviation, usually 1)
    - Lasso and Ridge regression because of the penalty term, K Nearest Neighbors, SVM, linear models if you want to use the coefficients to measure feature importance (more on this in part 6), neural networks
  - tree-based methods don't require standardization
  - check out part 1 to learn more about linear and logistic regression, Lasso and Ridge
  - check out part 4 to learn more about SVMs, tree-based methods, and K Nearest Neighbors

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

- **OrdinalEncoder** - converts categorical features into an integer array
- **OneHotEncoder** - converts categorical features into dummy arrays
- **StandardScaler** - standardizes continuous features by removing the mean and scaling to unit variance

Ordered categorical data: OrdinalEncoder

Let's assume we have a categorical feature and training and test sets

The cateogies can be ordered or ranked

E.g., educational level in the adult dataset

In [ ]:
```
import pandas as pd

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

X_train = pd.DataFrame(train_edu)
X_test = pd.DataFrame(test_edu)
```

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

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

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

# fit the training data
enc.fit(X_train)
# print the categories - not really important because we manually gave
print(enc.categories_)
# transform X_train. We could have used enc.fit_transform(X_train) to c
X_train_oe = enc.transform(X_train)
print(X_train_oe)
# transform X_test
X_test_oe = enc.transform(X_test) # OrdinalEncoder always throws an err
                                  # it encounters an unknown category i
print(X_test_oe)
```

Unordered categorical data: one-hot encoder

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

first feature: gender (male, female, unknown)

second feature: browser used

these categories cannot be ordered

In [ ]:
```
train = {'gender':['Male','Female','Unknown','Male','Female','Female'],
         'browser':['Safari','Safari','Internet Explorer','Chrome','Chr
test = {'gender':['Female','Male','Unknown','Female'],'browser':['Chrom

X_train = pd.DataFrame(train)
X_test = pd.DataFrame(test)
```

In [ ]:
```
# How do we convert this to numerical features?
from sklearn.preprocessing import OneHotEncoder

help(OneHotEncoder)
```

In [ ]:
```
# initialize the encoder
enc = OneHotEncoder(sparse=False) # by default, OneHotEncoder returns a
# fit the training data
enc.fit(X_train)
print('categories:',enc.categories_)
print('feature names:',enc.get_feature_names())
# transform X_train
X_train_ohe = enc.transform(X_train)
#print(X_train_ohe)
# do all of this in one step
X_train_ohe = enc.fit_transform(X_train)
#print(X_train_ohe)

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

Continuous features: StandardScaler

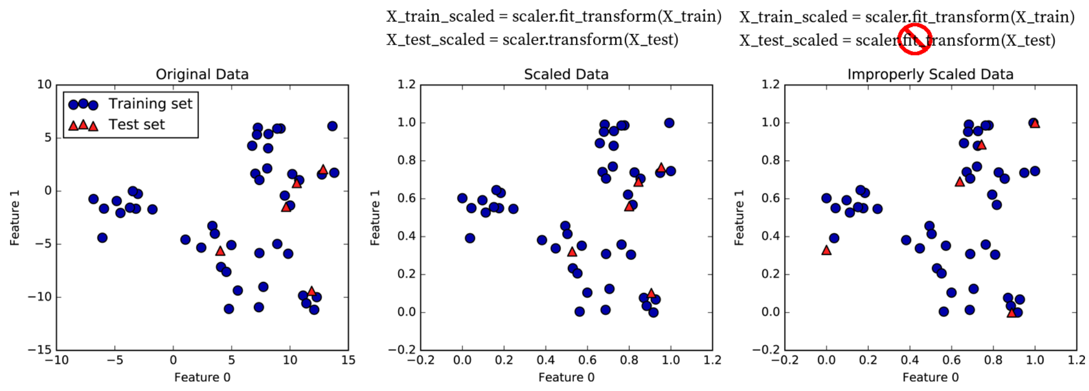
In [ ]:
```
train = {'salary':[50_000,75_000,40_000,1_000_000,30_000,250_000,35_000
test = {'salary':[25_000,55_000,1_500_000,60_000]}

X_train = pd.DataFrame(train)
X_test = pd.DataFrame(test)
```

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

In [ ]:
```python
scaler = StandardScaler()
print(scaler.fit_transform(X_train))
print(scaler.transform(X_test))
```

How and when to do preprocessing in the ML pipeline?

- **SPLIT YOUR DATA FIRST!**
- **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 from train and val, and val must be separated from 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

- Preprocessing and model training (not the splitting) 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 (https://scikit-learn.org/stable/auto_examples/compose/plot_column_transformer_mixed_types.html)](https://scikit-learn.org/stable/auto_examples/compose/plot_column_transformer_mixed_types.html)

In [ ]:

```python
import pandas as pd
import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder, Ordina
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 th
X = df.loc[:, df.columns != 'gross-income'] # all other columns are fea

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 =

# 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_s

```

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','
                ' Some-college',' Assoc-voc',' Assoc-acdm',' Bachelors'
onehot_ftrs = ['workclass','marital-status','occupation','relationship'
std_ftrs = ['capital-gain','capital-loss','age','hours-per-week']

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

# for now we only preprocess, later on we will add other steps here
# note the final scaler which is a standard scaler
# the ordinal and one hot encoded features do not have a mean of 0 and
# the final scaler standardizes those features
clf = Pipeline(steps=[('preprocessor', preprocessor),('final scaler',St

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(np.mean(X_train_prep,axis=0))
print(np.std(X_train_prep,axis=0))
print(np.mean(X_val_prep,axis=0))
print(np.std(X_val_prep,axis=0))
print(np.mean(X_test_prep,axis=0))
print(np.std(X_test_prep,axis=0))

```