Machine Learning Workflow

Plan

1. **Model Selection Tips** 💡
2. **Pipelines** 🔥
    A. Preprocessing Pipes
        - Pipelines → → →
        - Column Transformers ⑂
        - Custom Transformers →
        - Feature Unions ||
    B. Full Pipes (Preprocessing + Models)
3. **Surprise** 🤗

1. Model Selection

Let's take a step back: which models have we seen so far?

1️⃣ Regression models are parametric

- $\hat{y}$
- An arbitrarily large number n of datapoints can be modeled with few β parameters

Note: Neural Networks are also parametrics models (See `Deep Learning`)

✅ Fast to train, even on large datasets with Stochastic Gradient Descent

❗ Requires prior assumptions f about the structure of the data; may not find complex patterns, unless given complex features

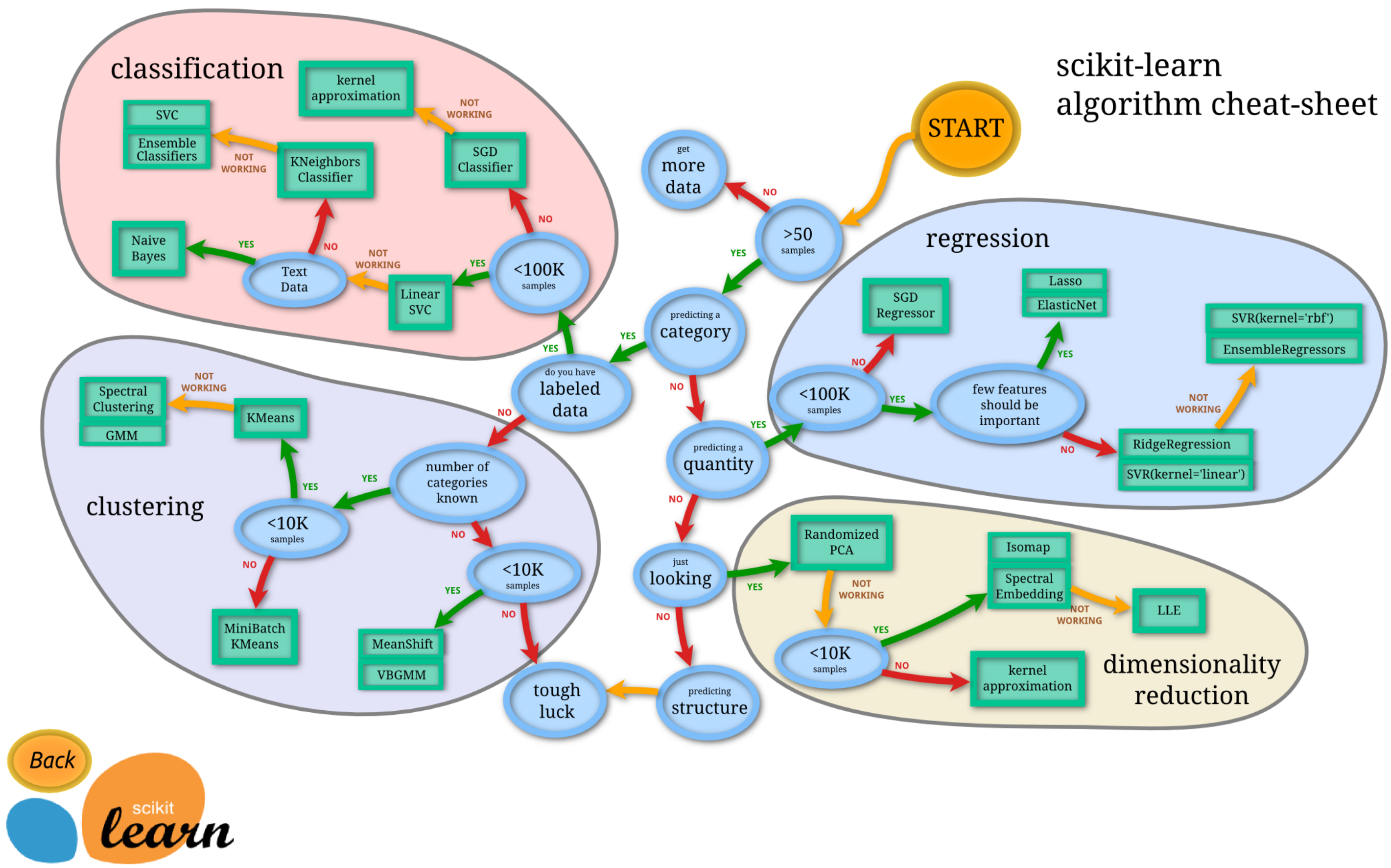
2️⃣ KNN, kernel-SVM are non-parametric

- No prior assumptions about the data structure are needed
- Possibly many parameters to learn (not known beforehand)
  - e.g. KNN `.fit` stores the *whole dataset*
  - e.g. rbf-SVM `.fit` must compute a Kernel between *each pair* of datapoints

Note: `Trees` are also non-parametric models (See `Ensemble Methods`)

✅ Can find complex features for you!

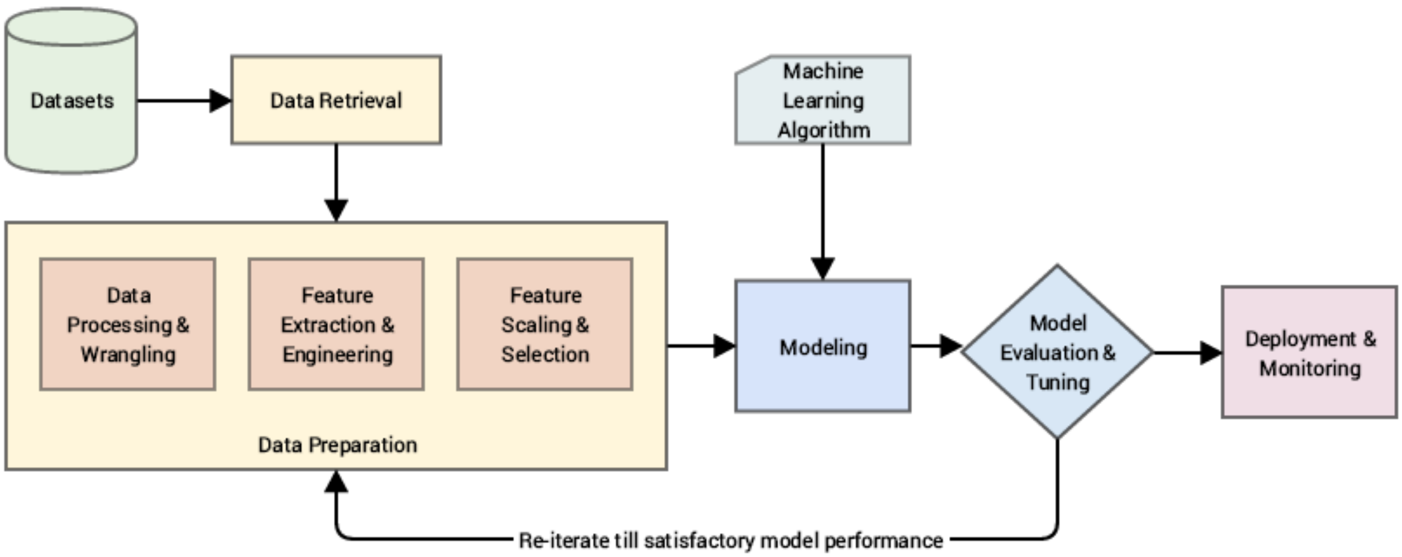
❗ Harder to train on large datasets and prone to overfitting



2. Pipelines

📚 **sklearn - Pipeline and composite estimators** (https://scikit-learn.org/stable/modules/compose.html)

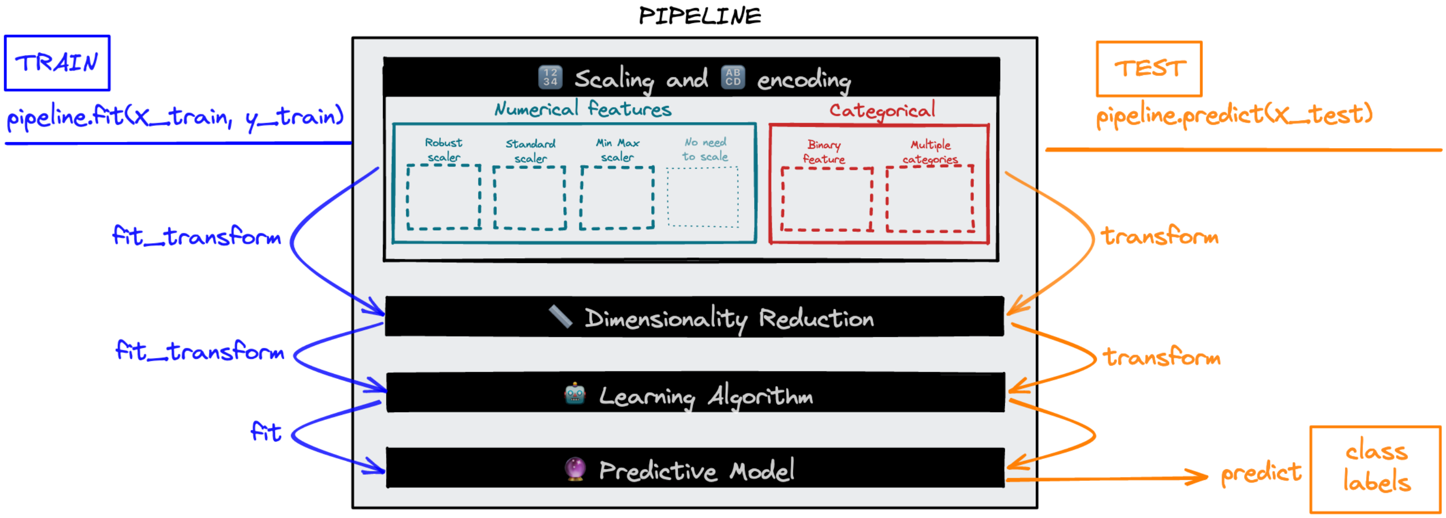
📚 **sklearn.pipeline** (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.pipeline)



A ***Pipeline*** is a chain of operations in a Machine Learning project (preprocessing, training, predicting, etc.)

Pipelines are powerful because they:

- 🏄 make your workflow much easier to read and understand
- 💪 enforce the implementation and order of steps in your project
- ⚙️ make your work reproducible and deployable



2.1 Preprocessing Pipelines

🎯 We are going to predict the **charges** of a health insurance contract based on various features using the following dataset.

💾 Download the dataset [here (https://wagon-public-datasets.s3.amazonaws.com/data_workflow.csv)](https://wagon-public-datasets.s3.amazonaws.com/data_workflow.csv)

In [ ]:
```
data.head(5)
```

Out[ ]:

	age	bmi	children	smoker	region	charges
0	19.0	27.900	0	True	southwest	16884.92400
1	18.0	33.770	1	False	southeast	1725.55230
2	NaN	33.000	3	False	southeast	4449.46200
3	33.0	22.705	0	False	northwest	21984.47061
4	32.0	28.880	0	False	northwest	3866.85520

In [ ]:
```
data.shape
```

Out[ ]:
```
(1338, 6)
```

In [ ]:
```
# Defining the features and the target

X = data.drop(columns='charges')
y = data['charges']

# Train-Test split

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[ ]:
```
((1070, 5), (268, 5), (1070,), (268,))
```

✏️ **Today's challenges**:

1. *Impute* missing values
2. Preprocessing:
    - *Scale* numerical features
    - *Encode* categorical features
3. *Fine-tune* your ML model ***and*** the preprocessing steps...

... 🔥 in one cell ! 🔥

a) Pipeline → → →

A Pipeline essentially **chains** multiple steps **in sequence** (e.g. *imputing* then *scaling*)

📚 [**sklearn.pipeline.Pipeline** (https://scikit-learn.org/0.16/modules/generated/sklearn.pipeline.Pipeline.html)](https://scikit-learn.org/0.16/modules/generated/sklearn.pipeline.Pipeline.html)

```
from sklearn.pipeline import Pipeline
```

In [ ]:
```
# Preprocess "age"
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Build the pipeline with the different steps
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('standard_scaler', StandardScaler())
])

pipeline.fit(X_train[['age']])
pipeline.transform(X_train[['age']])
```

Out[ ]:
```
array([[ 1.03287039],
       [-1.45497346],
       [ 1.1750329 ],
       ...,
       [ 0.25097661],
       [-0.17551091],
       [-1.2417297 ]])
```

In [ ]:
```
# Show the different steps of the pipeline
pipeline
```

Out[ ]:
```
▸ Pipeline
  ▸ SimpleImputer
  ▸ StandardScaler
```

b) Column Transformer ⑂

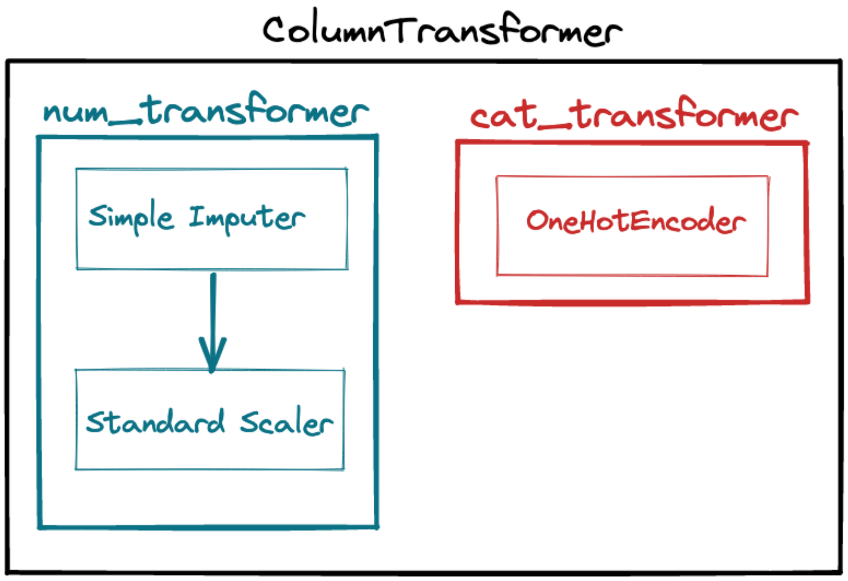
Column Transformers allow you to apply specific changes to specific columns **in parallel**

📚 [**sklearn.compose.ColumnTransformer** (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html)](https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html)

```
from sklearn.compose import ColumnTransformer
```

💻 Let's perform the following operations **in parallel**:

- 🔢 *Impute* then *scale* numerical values
- 🔠 *Encode* categorical values



☝️ Notice how a `Pipeline` object can be passed into a `ColumnTransformer` !

In [ ]:
```python
from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

from sklearn.preprocessing import OneHotEncoder


# Impute then scale numerical values:
num_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy="mean")),
    ('standard_scaler', StandardScaler())
])

# Encode categorical values
cat_transformer = OneHotEncoder(handle_unknown='ignore')

# Parallelize "num_transformer" and "cat_transfomer"
preprocessor = ColumnTransformer([
    ('num_transformer', num_transformer, ['age', 'bmi']),
    ('cat_transformer', cat_transformer, ['smoker', 'region'])
])
```

In [ ]:
```python
# Visualizing Pipelines in HTML
from sklearn import set_config; set_config(display='diagram')
preprocessor
```

Out[ ]:

```
▸ ColumnTransformer
▸ num_transformer      ▸ cat_transformer
  ▸ SimpleImputer        ▸ OneHotEncoder
  ▸ StandardScaler
```

In [ ]:
```python
X_train_transformed = preprocessor.fit_transform(X_train)

print("Original training set")
display(X_train.head(3))

print("Preprocessed training set")
display(pd.DataFrame(X_train_transformed).head(3))
```

Original training set

	age	bmi	children	smoker	region
162	54.0	39.60	1	False	southwest
410	19.0	17.48	0	False	northwest
639	56.0	33.66	4	False	southeast

Preprocessed training set

	0	1	2	3	4	5	6	7
0	1.032979	1.456688	1.0	0.0	0.0	0.0	0.0	1.0
1	-1.454870	-2.170790	1.0	0.0	0.0	1.0	0.0	0.0
2	1.175141	0.482582	1.0	0.0	0.0	0.0	1.0	0.0

😱 Where are the columns' names?

😉 Don't worry and stay tuned to `scikit-learn updates` !

- [scikit-learn.org/stable/whats_new.html (https://scikit-learn.org/stable/whats_new.html)](https://scikit-learn.org/stable/whats_new.html)

🚀 `get_feature_names_out()` 🚀

- New in `scikit-learn` 1.0.2 *(September 2021)*
  - ✅ This new method helps retrieve the names of the features which went through some transformations like StandardScaler or OheHotEncoder
  - ❌ Not all the transformers in Scikit-Learn have this new method
- New in `scikit-learn` 1.1.3: *(October 2022)*
  - ✅ *ALL* the transformers have this method!

In [ ]:
```
# Get your features' names
preprocessor.get_feature_names_out()
```

Out[ ]:
```
array(['num_transformer__age', 'num_transformer__bmi',
       'cat_transformer__smoker_False', 'cat_transformer__smoker_Tru
e',
       'cat_transformer__region_northeast',
       'cat_transformer__region_northwest',
       'cat_transformer__region_southeast',
       'cat_transformer__region_southwest'], dtype=object)
```

In [ ]:
```
pd.DataFrame(
    X_train_transformed,
    columns=preprocessor.get_feature_names_out()
).head()
```

Out[ ]:

	num_transformer__age	num_transformer__bmi	cat_transformer__smoker_False	cat_transform
0	1.032979	1.456688	1.0	
1	-1.454870	-2.170790	1.0	
2	1.175141	0.482582	1.0	
3	-0.815138	0.157880	0.0	
4	-0.601894	-0.148783	0.0	

🤔 What happened to the `children` column? What if we want to keep it untouched?

👉 **`remainder=passthrough`**

In [ ]:
```
preprocessor = ColumnTransformer([
    ('num_transformer', num_transformer, ['age','bmi']),
    ('cat_transformer', cat_transformer, ['region','smoker'])],
    remainder='passthrough'
)

preprocessor
```

Out[ ]:
```
▸ ColumnTransformer
▸ num_transformer    ▸ cat_transformer    ▸ remainder
▸ SimpleImputer      ▸ OneHotEncoder      ▸ passthrough
▸ StandardScaler
```

In [ ]:
```
pd.DataFrame(preprocessor.fit_transform(X_train),
             columns=preprocessor.get_feature_names_out()).head(3)
```

Out[ ]:

	num_transformer__age	num_transformer__bmi	cat_transformer__region_northeast	cat_transf
0	1.032979	1.456688	0.0	
1	-1.454870	-2.170790	0.0	
2	1.175141	0.482582	0.0	

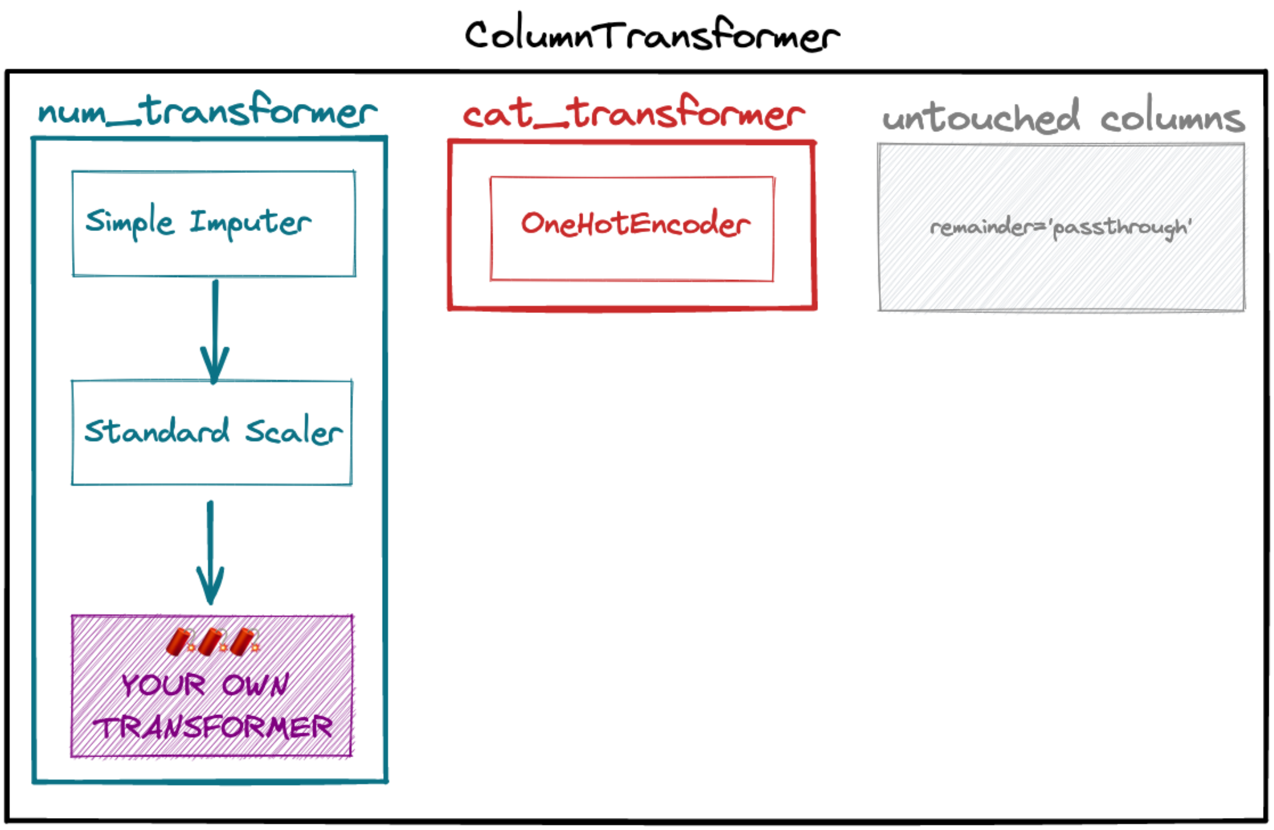
c) Custom: Function Transformer →

📚 [**sklearn.preprocessing.FunctionTransformer** (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.FunctionTransformer.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.FunctionTransformer.html)

```python
from sklearn.preprocessing import FunctionTransformer
```

⚒️ Function Transformers enable you to encapsulate a *Python* function within a `scikit` Transformer (→) Object

💪 They can be used with either Pipelines (→ → →) or ColumnTransformers (⑂)



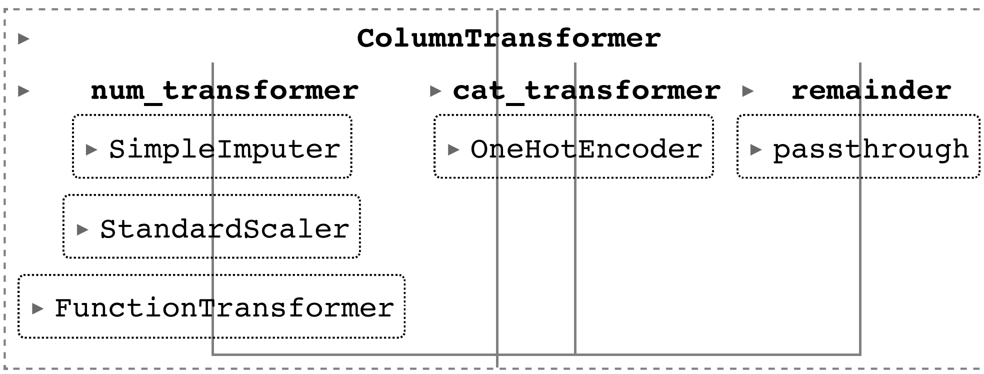
👆 If you want to use your own transformer in a Pipeline or a ColumnTransformer *(not one already available in Sklearn)*, you must encapsulate your function within a **FunctionTransformer**.

In [ ]:
```python
from sklearn.preprocessing import FunctionTransformer
```

In [ ]:
```
# Create a transformer that compresses data to 2 digits (for instanc
e!)
# rounder = FunctionTransformer(np.round)

# We can use a lambda function for more customizable functions
rounder = FunctionTransformer(lambda array: np.round(array, decimals=
2))
```

In [ ]:
```
# Add it at the end of our numerical transformer
num_transformer = Pipeline([
    ('imputer', SimpleImputer()),
    ('scaler', StandardScaler()),
    ('rounder', rounder)])

# Encode categorical values
cat_transformer = OneHotEncoder(drop='if_binary',
                                handle_unknown='ignore')

preprocessor = ColumnTransformer([
    ('num_transformer', num_transformer, ['bmi', 'age']),
    ('cat_transformer', cat_transformer, ['region', 'smoker'])],
    remainder='passthrough')
preprocessor
```

Out[ ]:



In [ ]:
```
pd.DataFrame(preprocessor.fit_transform(X_train)).head(3)
```

Out[ ]:

	0	1	2	3	4	5	6	7
0	1.46	1.03	0.0	0.0	0.0	1.0	0.0	1.0
1	-2.17	-1.45	0.0	1.0	0.0	0.0	0.0	0.0
2	0.48	1.18	0.0	0.0	1.0	0.0	0.0	4.0

❗ **`FunctionTransformer`** only works for <u>**stateless**</u> transformations ❗

🧑‍🏫 **stateless transformations** are transformations which cannot *store* information during `.fit(X_train)` that would be used for the `.transform(X_test)` .

👍 Since a stateless transformation doesn't learn anything, fitting it is impossible, it does nothing other than transform!

✅ `FunctionTransformer` is compatible with **stateless** transformations.

Examples of transformations which don't "learn" anything:

- $$X \rightarrow log(X)$$
- $$(X_1, X_2) \rightarrow X_1 + 5X_2$$

🧑‍🏫 **stateful transformations** are transformations which *store* information during `.fit(X_train`). This information is re-used for `.transform(X_test)` .

Examples of transformations which "learn" something:

$X_{train} \rightarrow StandardScaler(X_{train})$
learns
μ_{train}
and
σ_{train}

$X_{train} \rightarrow MinMaxScaler(X_{train})$
learns
$X_{train}^{(min)}$
and
$X_{train}^{(max)}$

❌ `FunctionTransformer` is not compatible with **stateful** transformations

🌶️ We will have to code our own `Class` to use FunctionTransformer with stateful transformations!

🕵️ **Transformers under the hood**

In [ ]:
```python
from sklearn.base import TransformerMixin, BaseEstimator

class MyCustomTranformer(TransformerMixin, BaseEstimator):
    # BaseEstimator generates the get_params() and set_params() methods that all Pipelines require
    # TransformerMixin creates the fit_transform() method from fit() and transform()

    def __init__(self):
        pass

    def fit(self, X, y=None):
        # Here you store what needs to be stored/learned during .fit(X_train) as instance attributes
        # Return "self" to allow chaining .fit().transform()
        pass

    def transform(self, X, y=None):
        # Return the result as a DataFrame for an integration into the ColumnTransformer
        pass
```

```python
my_transformer = MyCustomTranformer()
my_transformer.fit(X_train)
my_transformer.transform(X_train)
my_transformer.transform(X_test)
```

💻 *More in today's challenges* 💻

d) FeatureUnion | |

`FeatureUnion` applies a list of transformer objects **in parallel** to the input data, then **concatenates** the results. This is useful to combine several feature extraction mechanisms into a single transformer

📚 [**sklearn.pipeline.FeatureUnion** (https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html)](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html)

👌 Useful to **create entirely new features**!

Example: let's build and add a new feature called `bmi_age_ratio`

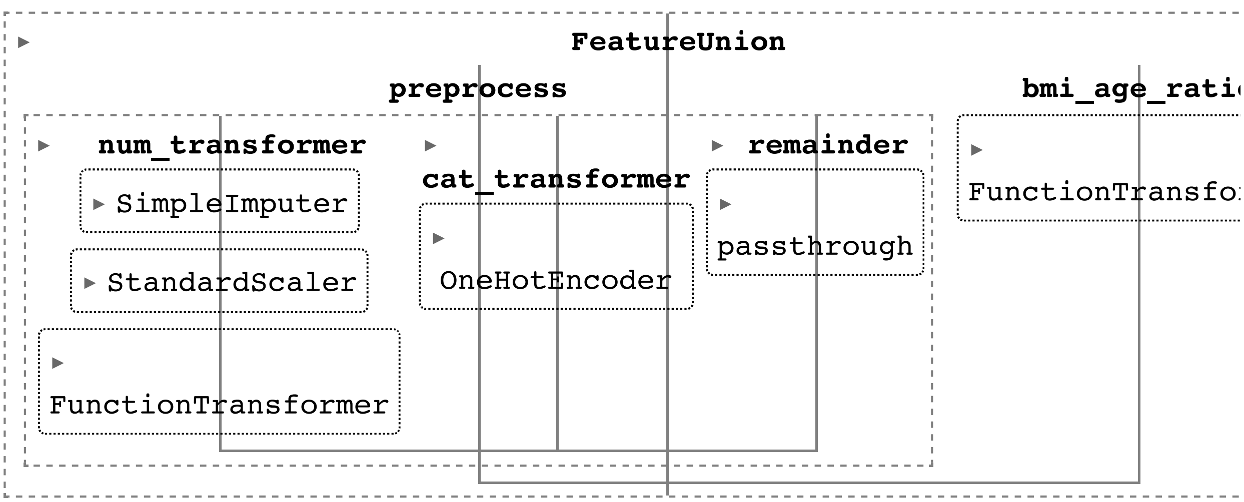
In [ ]:
```
X_train.head(3)
```

Out[ ]:

	age	bmi	children	smoker	region
162	54.0	39.60	1	False	southwest
410	19.0	17.48	0	False	northwest
639	56.0	33.66	4	False	southeast

In [ ]:
```
from sklearn.pipeline import FeatureUnion

# Create a custom transformer that multiplies/divides two columns
# Notice that we are creating this new feature completely randomly just as an example
bmi_age_ratio_constructor = FunctionTransformer(lambda df: pd.DataFrame(df["bmi"] / df["age"]))

union = FeatureUnion([
    ('preprocess', preprocessor), # columns 0-7
    ('bmi_age_ratio', bmi_age_ratio_constructor) # new column 8
])

union
```

Out[ ]:



In [ ]:
```
pd.DataFrame(union.fit_transform(X_train)).head(1)
```

Out[ ]:

	0	1	2	3	4	5	6	7	8
0	1.46	1.03	0.0	0.0	0.0	1.0	0.0	1.0	0.733333

Building your preprocessor with `make_***` shortcuts ⚡

```
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion
from sklearn.compose import ColumnTransformer
```

There are equivalent transformers using the syntax **make_*** 👇

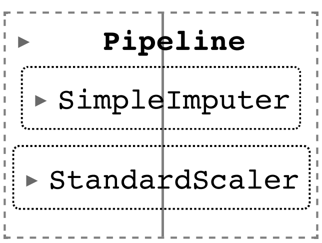
In [ ]:
```
from sklearn.pipeline import make_pipeline
from sklearn.pipeline import make_union
from sklearn.compose import make_column_transformer
```

```
Pipeline([
    ('my_name_for_the_imputer', SimpleImputer()),
    ('my_name_for_the_scaler', StandardScaler())
])
```

⇔

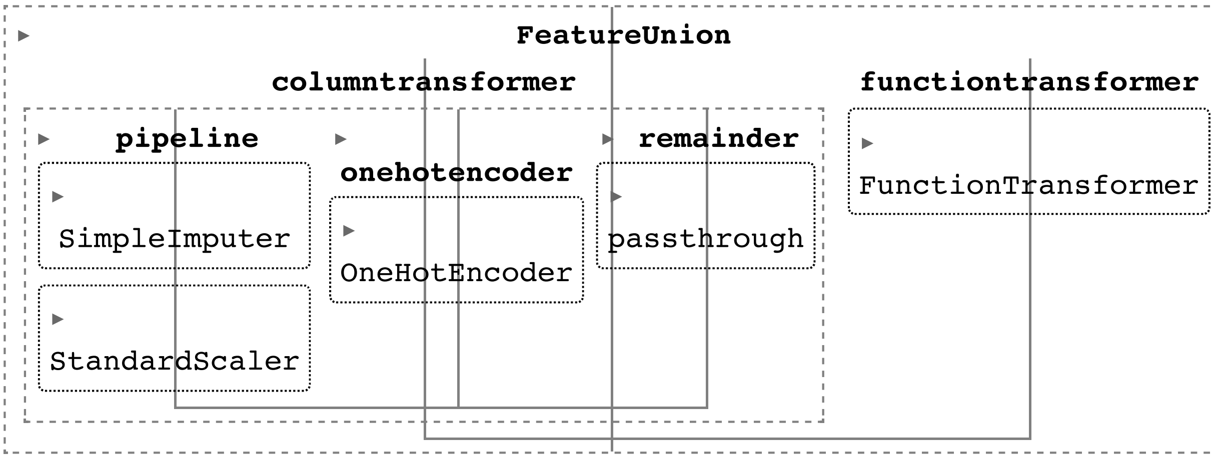
In [ ]:
```
make_pipeline(SimpleImputer(), StandardScaler())
```

Out[ ]:



In [ ]:
```python
num_transformer = make_pipeline(SimpleImputer(), StandardScaler())
cat_transformer = OneHotEncoder()

preproc_basic = make_column_transformer(
    (num_transformer, ['age', 'bmi']),
    (cat_transformer, ['smoker', 'region']),
    remainder='passthrough'
)

preproc_full = make_union(preproc_basic, bmi_age_ratio_constructor)

preproc_full
```

Out[ ]:



🍒 **`make_column_selector` selects features automatically based on `dtype`**

```python
from sklearn.compose import make_column_selector

num_col = make_column_selector(dtype_include=['float64'])
cat_col = make_column_selector(dtype_include=['object','bool'])
```

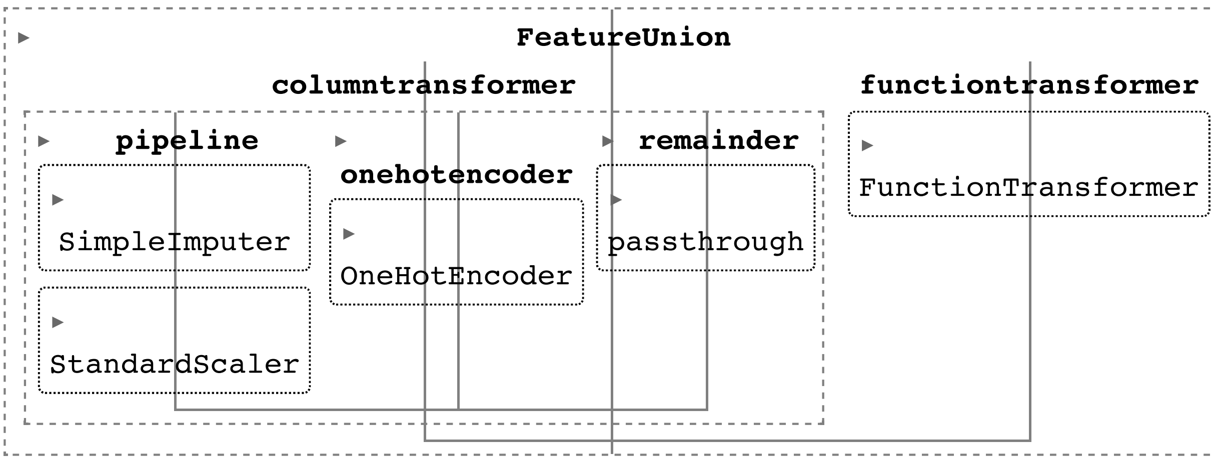
In [ ]:
```python
X_train.dtypes
```

Out[ ]:
```
age          float64
bmi          float64
children       int64
smoker          bool
region        object
dtype: object
```

🎉 Complete preprocessing pipeline 🎉

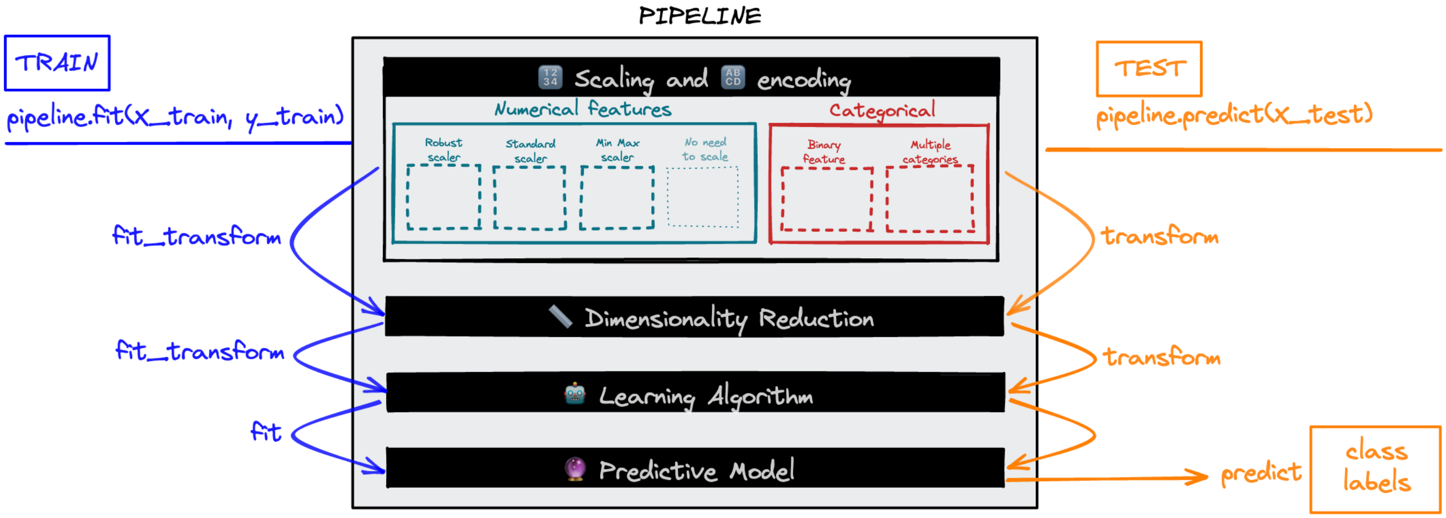
In [ ]:
```python
from sklearn.compose import make_column_selector

num_transformer = make_pipeline(SimpleImputer(), StandardScaler())
num_col = make_column_selector(dtype_include=['float64'])

cat_transformer = OneHotEncoder()
cat_col = make_column_selector(dtype_include=['object','bool'])

preproc_basic = make_column_transformer(
    (num_transformer, num_col),
    (cat_transformer, cat_col),
    remainder='passthrough'
)

preproc_full = make_union(preproc_basic, bmi_age_ratio_constructor)

preproc_full
```

Out[ ]:



2.2 Including models in Pipelines

- Model objects can be plugged into Pipelines
- Pipelines inherit the methods of the **last** object in the sequence
  - Transformers: `fit` and `transform`
  - Models: `fit` , `score` , `predict` , etc.

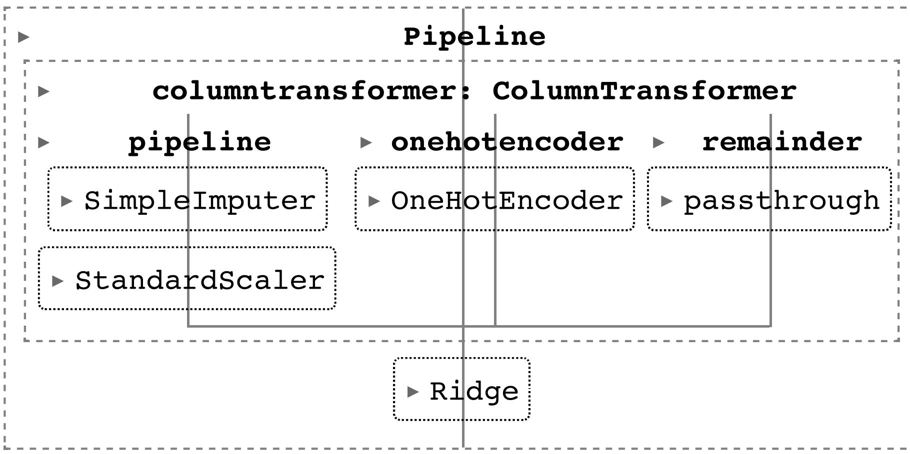


- When executing the pipeline.fit method, the transformer's `.fit` and `.transform` methods will be called sequentially, and the model will be trained.
  - At this stage, all transformers' variables are saved into the memory of the pipeline
- When executing the pipeline.predict method, only the transformer's `.transform` method will be called, using the variables learned during the original `fit`

a) Full pipeline

In [ ]:
```python
from sklearn.linear_model import Ridge

# Preprocessor
num_transformer = make_pipeline(SimpleImputer(), StandardScaler())
cat_transformer = OneHotEncoder()

preproc = make_column_transformer(
    (num_transformer, make_column_selector(dtype_include=['float6
4'])),
    (cat_transformer, make_column_selector(dtype_include=['object','bo
ol'])),
    remainder='passthrough'
)

# Add estimator
pipeline = make_pipeline(preproc, Ridge())
pipeline
```

Out[ ]:



In [ ]:
```python
# Train Pipeline
pipeline.fit(X_train,y_train)

# Make predictions
pipeline.predict(X_test.iloc[0:1])

# Score model
pipeline.score(X_test,y_test)
```

Out[ ]: 0.7473478157212925

b) Cross-validate a Pipeline

In [ ]:
```
from sklearn.model_selection import cross_val_score

# Cross-validate Pipeline
cross_val_score(pipeline, X_train, y_train, cv=5, scoring='r2').mean()
```

Out[ ]:
```
0.7434317676218065
```

c) Grid Search a Pipeline

- *Grid Searching* allows you to check which combination of preprocessing/modeling **hyperparameters** works best.
- It is possible to *Grid Search* the hyperparameters of **any component of the Pipeline**
  - Typical Sklearn syntax: `step_name__transformer_name__hyperparameter_name`
  - To check which hyperparameters of the pipeline can be optimized: `pipeline.get_params()`

In [ ]:
```
# Which parameters of the pipeline are GridSearch-able?
pipeline.get_params()
```

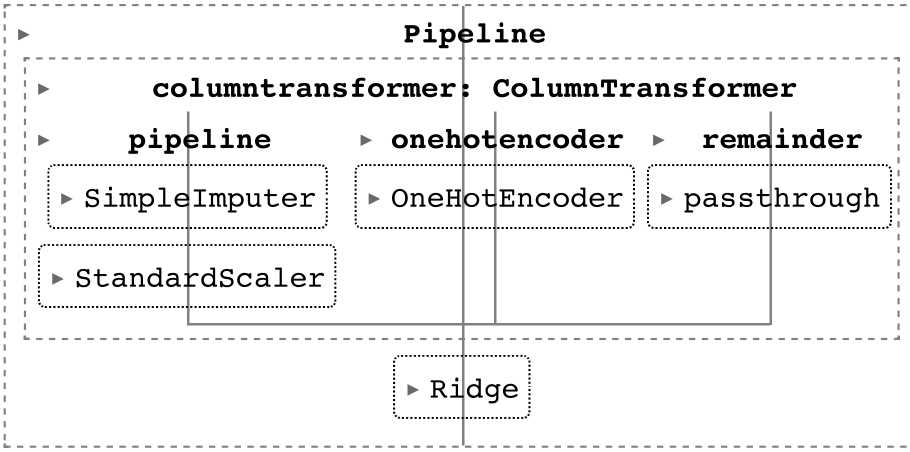
In [ ]:
```
from sklearn.model_selection import GridSearchCV

grid_search = GridSearchCV(
    pipeline,
    param_grid={
        # Access any component of the Pipeline
        # and any available hyperparamater you want to optimize
        'columntransformer__pipeline__simpleimputer__strategy': ['mea
n', 'median'],
        'ridge__alpha': [0.1, 0.5, 1, 5, 10]
    },
    cv=5,
    scoring="r2")

grid_search.fit(X_train, y_train)

grid_search.best_params_
```

Out[ ]:
```
{'columntransformer__pipeline__simpleimputer__strategy': 'mean',
 'ridge__alpha': 1}
```

💾 Let's save the pipelined model with the best hyperparameters.

In [ ]:
```
pipeline_tuned = grid_search.best_estimator_
pipeline_tuned
```

Out[ ]:



🔮 We can use this "best" model for predictions without re-training it!

In [ ]:
```
pipeline_tuned.predict(X_test[0:1])
```

Out[ ]:
```
array([10216.56989159])
```

d) Caching to avoid repeated computations

😠 Are your preprocessing steps too long to run?

🪄 You can use caching techniques!

```python
from tempfile import mkdtemp
from shutil import rmtree

# Create a temp folder
cachedir = mkdtemp()

# Instantiate the Pipeline with the cache parameter
pipeline = Pipeline(steps, memory=cachedir)

# Clear the cache directory after the cross-validation
rmtree(cachedir)
```

With the parameter `memory=cachedir` , `preproc` parameters can be cached into memory.

- Avoid recalculating all of the parameters during CrossValidation or GridSearchCV on `estimator` hyperparams only
- Helpful only when the transformer's `.fit` time is long and the dataset is very large

e) Debug your pipe

In [ ]:
```python
# Access the components of a Pipeline with `named_steps`
pipeline_tuned.named_steps.keys()
```

Out[ ]:
```
dict_keys(['columntransformer', 'ridge'])
```

In [ ]:
```python
# Check intermediate steps
print("Before preprocessing, X_train.shape = ")
print(X_train.shape)
print("After preprocessing, X_train_preprocessed.shape = ")
pipeline_tuned.named_steps["columntransformer"].fit_transform(X_trai
n).shape
```

```
Before preprocessing, X_train.shape =
(1070, 5)
After preprocessing, X_train_preprocessed.shape =
```

Out[ ]:
```
(1070, 9)
```

f) Exporting models/Pipelines

💾 You can export your final model/pipeline as a `pickle` file

👉The file can then be loaded back into a notebook or deployed on a server (see `ML Ops` module).

In [ ]:
```
import pickle

# Export Pipeline as pickle file
with open("pipeline.pkl", "wb") as file:
    pickle.dump(pipeline_tuned, file)

# Load Pipeline from pickle file
my_pipeline = pickle.load(open("pipeline.pkl","rb"))

my_pipeline.score(X_test, y_test)
```

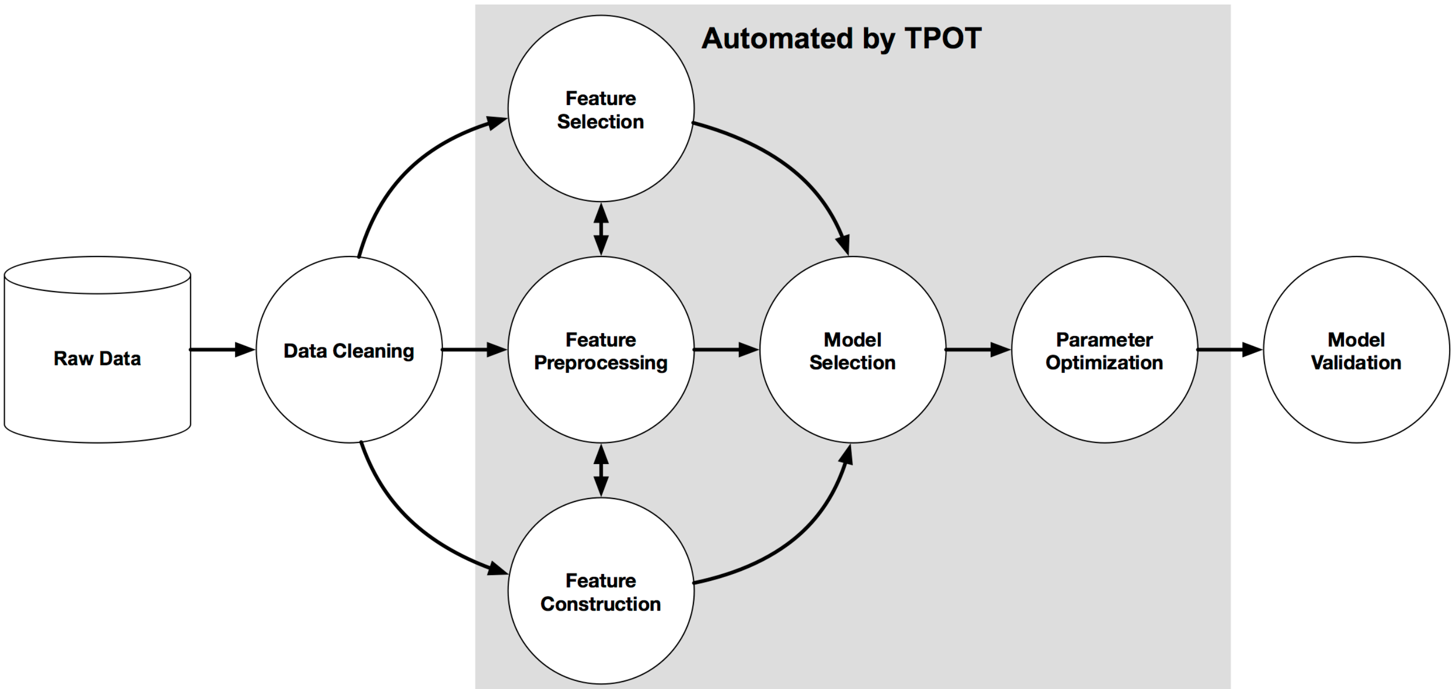
Out[ ]:
```
0.7473478157212925
```

3. Surprise 🎉

AutoML

TPOT

The Tree-based Pipeline Optimization Tool (TPOT) is an automated Machine Learning tool that optimizes Machine Learning Pipelines



📚 More details available in the [**TPOT documentation** (http://epistasislab.github.io/tpot/)](http://epistasislab.github.io/tpot/)

Installation

pip install TPOT

In [ ]:
```
import os
from tpot import TPOTRegressor

X_train_preproc = preproc_basic.fit_transform(X_train)
X_test_preproc = preproc_basic.transform(X_test)
```

In [ ]:
```python
# Instantiate TPOTClassifier
tpot = TPOTRegressor(generations=4, population_size=20, verbosity=2, s
coring='r2', n_jobs=-1, cv=2)

# Process autoML with TPOT
tpot.fit(X_train_preproc, y_train)

# Print score
print(tpot.score(X_test_preproc, y_test))
```

```
Generation 1 - Current best internal CV score: 0.8517440046999218

Generation 2 - Current best internal CV score: 0.853008927910814

Generation 3 - Current best internal CV score: 0.853008927910814

Generation 4 - Current best internal CV score: 0.8558530771102855

Best pipeline: RidgeCV(GradientBoostingRegressor(input_matrix, alpha
=0.9, learning_rate=0.01, loss=ls, max_depth=3, max_features=0.60000
00000000001, min_samples_leaf=14, min_samples_split=13, n_estimators
=100, subsample=0.55))
0.872811877434264
```

```
/Users/davywai/.pyenv/versions/3.8.12/envs/lewagon-data/lib/python3.
8/site-packages/sklearn/metrics/_scorer.py:765: FutureWarning: sklea
rn.metrics.SCORERS is deprecated and will be removed in v1.3. Please
use sklearn.metrics.get_scorer_names to get a list of available scor
ers and sklearn.metrics.get_metric to get scorer.
  warnings.warn(
```

In [ ]:
```
# Export TPOT Pipeline to a Python file
tpot.export(os.path.join(os.getcwd(),'tpot_iris_pipeline.py'))

! cat 'tpot_iris_pipeline.py'
```

```
import numpy as np
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline, make_union
from tpot.builtins import StackingEstimator

# NOTE: Make sure that the outcome column is labeled 'target' in the
data file
tpot_data = pd.read_csv('PATH/TO/DATA/FILE', sep='COLUMN_SEPARATOR',
dtype=np.float64)
features = tpot_data.drop('target', axis=1)
training_features, testing_features, training_target, testing_target
= \
            train_test_split(features, tpot_data['target'], random_s
tate=None)

# Average CV score on the training set was: 0.8558530771102855
exported_pipeline = make_pipeline(
    StackingEstimator(estimator=GradientBoostingRegressor(alpha=0.9,
learning_rate=0.01, loss="ls", max_depth=3, max_features=0.600000000
0000001, min_samples_leaf=14, min_samples_split=13, n_estimators=10
0, subsample=0.55)),
    RidgeCV()
)

exported_pipeline.fit(training_features, training_target)
results = exported_pipeline.predict(testing_features)
```

Summary

- `Pipeline` $\rightarrow$ list of sequential steps

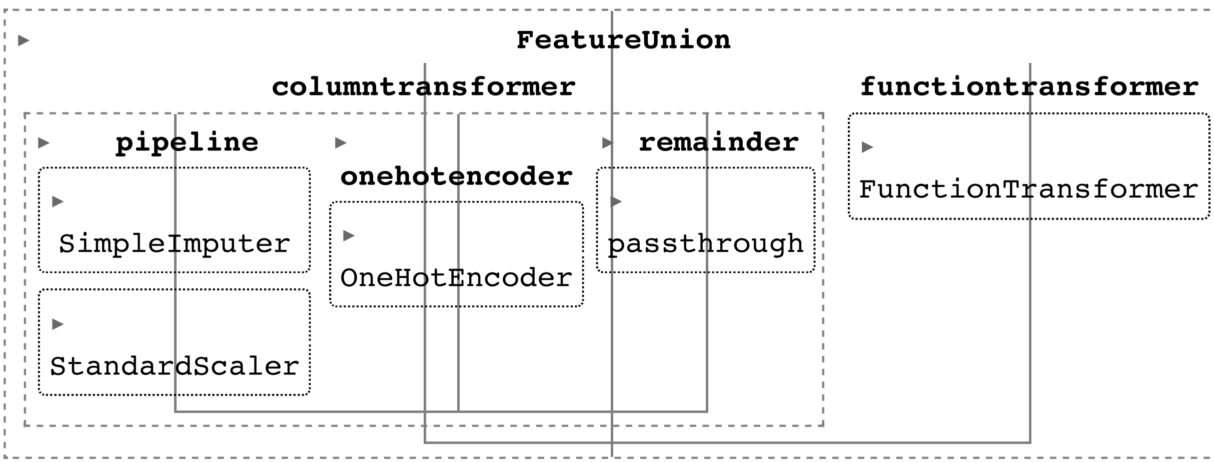
- `ColumnTransformer` → list of parallel steps
    - `remainder="passthrough"` : used to save untransformed columns

- `FunctionTransformer` → encapsulates a function as a Scikit-Learn transformer that you can plug into a Pipeline or a ColumnTransformer

- `FeatureUnion` → applies transformations in parallel and concatenates the results, quite useful for feature creation

In [ ]:
```
preproc_full
```

Out[ ]:



- You can chain a `preprocessor pipeline` with a Scikit Learn model

- A full pipeline can go through `cross_validate` , `GridSearchCV` , `RandomizedSearchCV`

In [ ]:
```
pipeline_tuned
```

Out[ ]:
```
▸ Pipeline
  ▸ columntransformer: ColumnTransformer
    ▸ pipeline          ▸ onehotencoder     ▸ remainder
      ▸ SimpleImputer     ▸ OneHotEncoder     ▸ passthrough
      ▸ StandardScaler
  ▸ Ridge
```

Your turn! 🚀