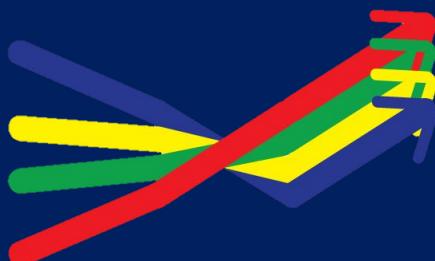
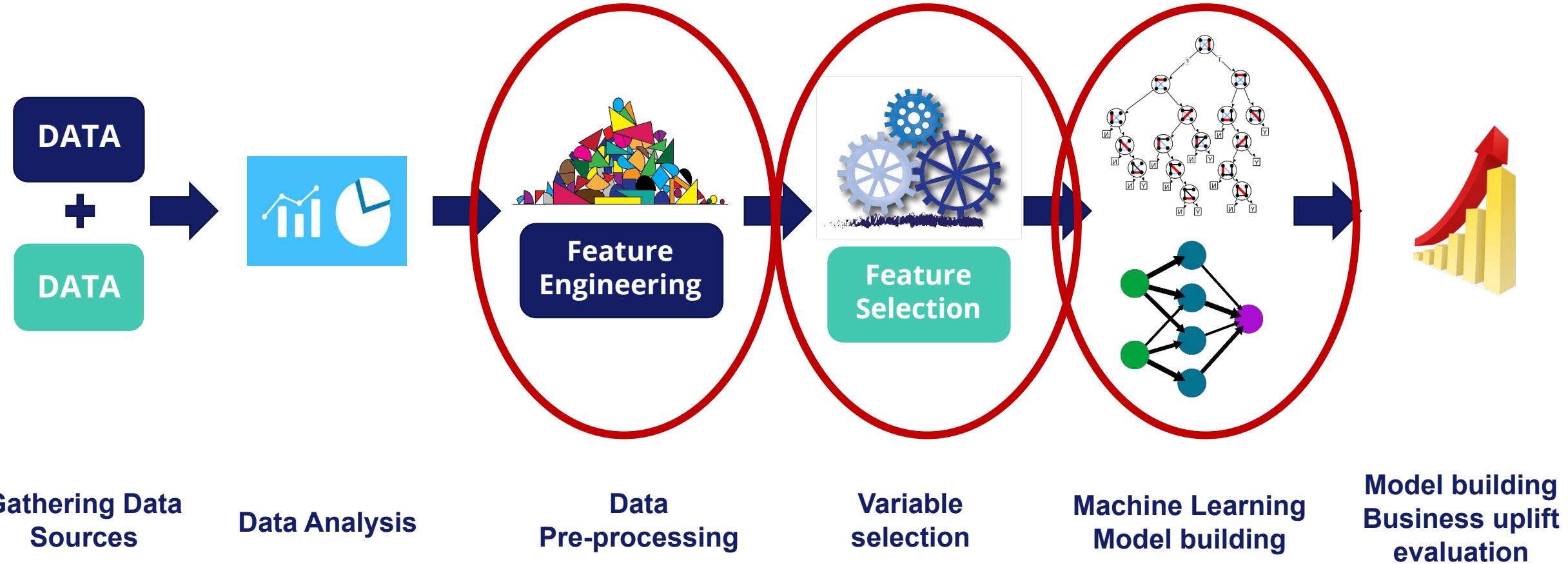


Writing Production Code for Machine Learning Deployment

Overview



Machine Learning Pipeline: Production



Towards deployment code

A screenshot of a code editor showing two tabs: "main.py" and "index.html".

The "main.py" tab contains Python code for a Flask application:

```
File Edit Selection Find View Goto Tools Project Preferences Help Laravel
< > main.py x
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4 ...
5 ...
6 ===== Application Routes =====
7 | Following are the application routes which
8 | control all of the routing inside of the web
9 | application.
10 ...
11 ...
12 ...
13 ...
14 ...
15 @app.route("/")
16 def index():
17     return render_template("index.html")
18 ...
19 @app.route("/portfolio")
20 def about():
21     return render_template("portfolio.html")
22 ...
23 @app.route("/projects")
24 def about():
25     return render_template("projects.html")
26 ...
27 @app.route("/contact")
28 def about():
29     return render_template("contact.html")
30 ...
31 @app.route("/about")
```

The "index.html" tab contains HTML and Jinja2 template code:

```
<% extends "layout.html" %>
<% block page_title %>
Home
<% endblock %>
<% block header %>
<header class="jumbotron-fluid text-center">
<h1> A Simple Quote <br>
<small> A Person</small>
</h1>
</header>
<div class="wrapper">
<ol class="breadcrumb">
<li class="breadcrumb-item">Home</li>
</ol>
</div>
<% endblock %>
<% block main_content %>
<main class="col-md-8">
<article class="card-plain">
<header class="card-header">
<p>
<a href="#" class="label label-primary">Category</a>
</p>
<h3 class="">Post Title</h3>
...
<% endblock %>
```

Code to:

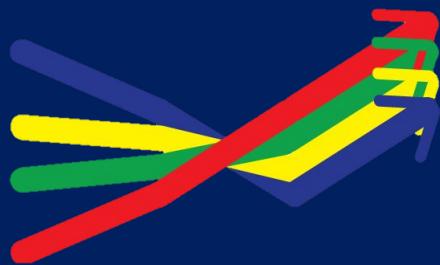
- Create and Transform features
- Incorporate the feature selection
- Build Machine Learning Models
- Score new data

How to write deployment code for ML

- Procedural Programming
- Custom Pipeline Code
- Third Party Pipeline Code



Procedural Programming



Procedural Programming

In **procedural programming**, procedures, also known as routines, subroutines or functions, are carried out as a series of computational steps.

For us, it refers to writing the series of feature creation, feature transformation, model training and data scoring steps, as functions, that then we can call and run one after the other.

Procedural Programming: the functions

```
1 # ===== EXAMPLE OF PROCEDURAL PROGRAMMING SCRIPT ====
2
3 import pandas as pd
4 import numpy as np
5
6 # to divide train and test set
7 from sklearn.model_selection import train_test_split
8
9 # feature scaling
10 from sklearn.preprocessing import StandardScaler
11
12 # to build the models
13 from sklearn.linear_model import LinearRegression, Lasso
14
15 # to evaluate the models
16 from sklearn.metrics import mean_squared_error
17
18
19 # Functional programming, getting all parameters from yaml file
20
21 def load_data(df_path):
22     return pd.read_csv(df_path)
23
24
25 def divide_train_test(df, target):
26     X_train, X_test, y_train, y_test = train_test_split(df, df[target],
27                                                       test_size=0.2,
28                                                       random_state=0)
29
30     return X_train, X_test, y_train, y_test
31
32 def remove_numerical_na(df, var, mean_val):
33     return df[var].fillna(mean_val)
34
35
36 def remove_categorical_na(df, var):
37     return df[var].fillna('Missing')
38
39
40 def cap_outliers(df, var, cap, bigger_than=False):
41     if bigger_than:
42         capped_var = np.where(df[var]>cap, cap, df[var])
43     else:
44         capped_var = np.where(df[var]<cap, cap, df[var])
45
46     return capped_var
```

```
32 https://www.udemy.com/.../outlier.ipynb 01_functional_programming_hardcoding.yaml.py
33 def remove_numerical_na(df, var, mean_val):
34     return df[var].fillna(mean_val)
35
36 def remove_categorical_na(df, var):
37     return df[var].fillna('Missing')
38
39
40 def cap_outliers(df, var, cap, bigger_than=False):
41     if bigger_than:
42         capped_var = np.where(df[var]>cap, cap, df[var])
43     else:
44         capped_var = np.where(df[var]<cap, cap, df[var])
45
46     return capped_var
47
48
49 def transform_skewed_variables(df, var):
50     return np.log(df[var])
51
52
53 def remove_rare_labels(df, var, frequent_labels):
54     return np.where(df[var].isin(frequent_labels, df[var], 'Rare'))
55
56
57 def train_scaler(df, output_path):
58     scaler = StandardScaler()
59     scaler.fit(df)
60     joblib.save(scaler, output_path)
61
62
63 def scale_features(df, scaler):
64     scaler = load(scaler) # with joblib probably
65     return scaler.transform(df)
66
67
68 def train_model(df, target, features, scaler, output_path):
69     lin_model = Lasso(random_state=2909)
70     lin_model.fit(scaler.transform(df[features]), target)
71     joblib.save(lin_model, output_path)
72
73
74 def predict(df, model, features, scaler):
75     return model.predict_proba(scaler.transform(df[features]))
```

The functions or procedures to create and transform features, and to train and save the models and make the predictions

Procedural Programming: train script

```
#===== training pipeline =====

df = load(yaml_path_to_file)
train, test, y_train, y_test = divide_train_test(df, yaml_target_name)

# remove NA numerical
train[var1] = remove_numerical_na(train, var1, mean_val1_in_yaml)
train[var2] = remove_numerical_na(train, var2, mean_val2_in_yaml)

train[var3] = remove_categorical_na(train[var3])
train[var4] = remove_categorical_na(train[var4])

train[var5] = cap_outliers(train, var5, cap_value_in_yaml, bigger_than=False)
train[var6] = cap_outliers(train, var6, cap_value_in_yaml, bigger_than=False)

train[var7] = transform_skewed_variables(train, var7)

train[var8] = remove_rare_labels(train, var8, frequent_labels_in_yaml)

scaler = train_scaler(train, output_path_in_yaml)

lin_model = train_model(train, y_train, feature_list_in_yaml, scaler, output_path_in_yaml)

#== END
```

Calls the previous functions in order, to train and save the models

Procedural Programming: score script

```
# ===== scoring pipeline =====

data = 'load it from somewhere'

# remove NA numerical
data[var1] = remove_numerical_na(data, var1, mean_val1_in_yaml)
data[var2] = remove_numerical_na(data, var2, mean_val2_in_yaml)

data[var3] = remove_categorical_na(data[var3])
data[var4] = remove_categorical_na(data[var4])

data[var5] = cap_outliers(data, var5, cap_value_in_yaml, bigger_than=False)
data[var6] = cap_outliers(data, var6, cap_value_in_yaml, bigger_than=False)

data[var7] = transform_skewed_variables(data, var7)

data[var8] = remove_rare_labels(data, var8, frequent_labels_in_yaml)

scaler = joblib.load((output_path_in_yaml_to_scaler)
lin_model = joblib.load(output_path_in_yaml_to_model)

score = predict(data, lin_model, feature_list_in_yaml, scaler)

# ===== END
```

Calls the previous functions in order, to score new data

Procedural Programming: yaml file

```
### example of yaml ####

#paths
path_to_dataset = "path_to_my_dataset"
output_scaler_path = 'path_to_store_scaler'
output_model_path = 'path_to_store_model'                                Hard coded variables  
to engineer, and  
values to use to  
transform features.

# preproc
var1_mean_val = 1
var2_mean_val = 2                                                       Hardcoded paths to  
retrieve and store data

var4_cap_value = 1000
var5_cap_value = 5000

var8_frequent_labels = ['frequent1', 'frequent2', 'frequent3']          By changing these  
values, we can  
re-adjust our models

# features
features = ['var1', 'var2', 'var3', 'var4', 'etc']

===== END ======
```

Procedural Programming: Overview

Advantages

- Straightforward from jupyter notebook
- No software development skills required
- Easy to manually check if it reproduces the original model

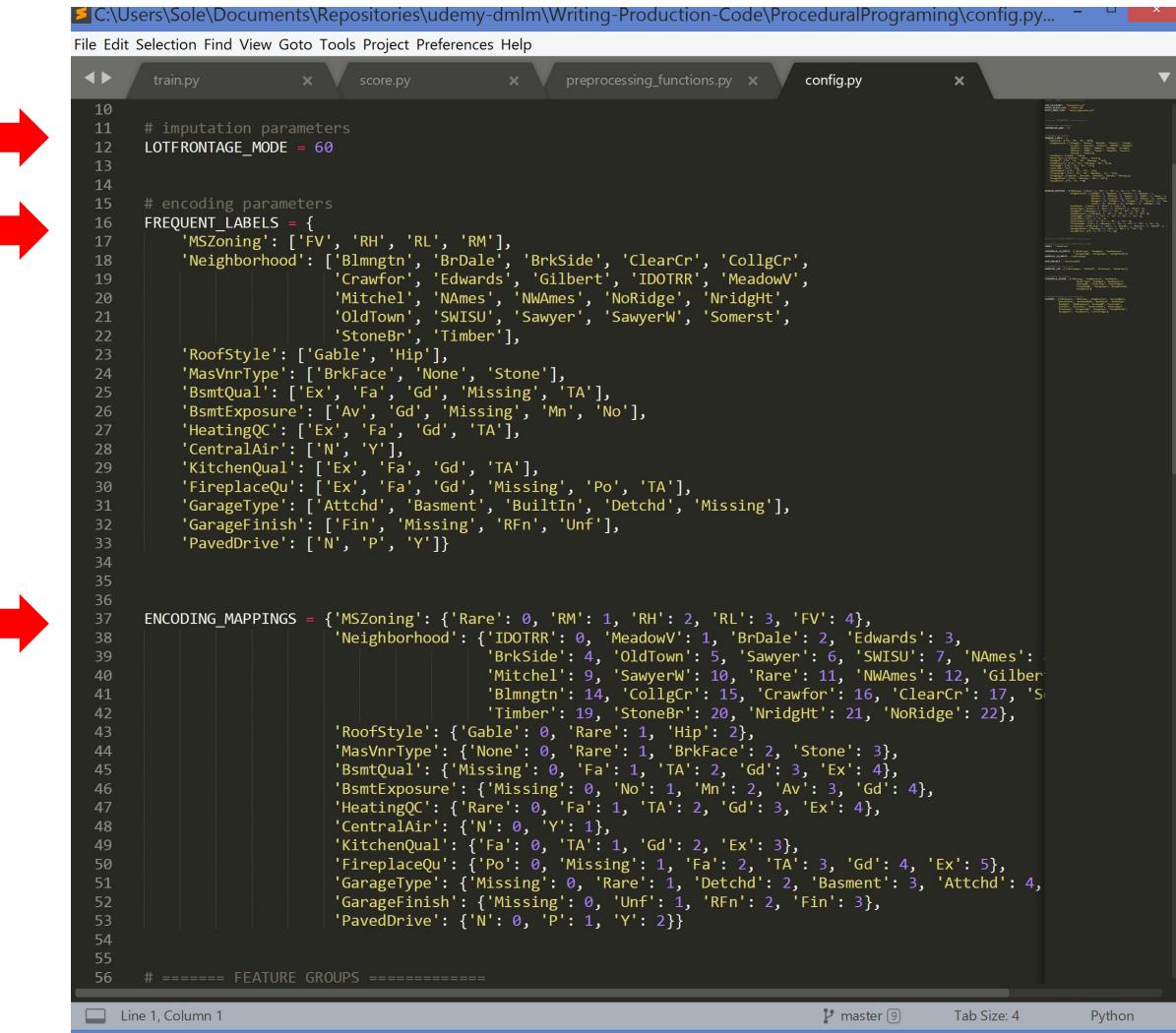
Disadvantages

- Can get buggy
- Difficult to test
- Difficult to build software on top of it
- Need to save a lot of intermediate files to store the transformation parameters

Custom Machine Learning Pipeline



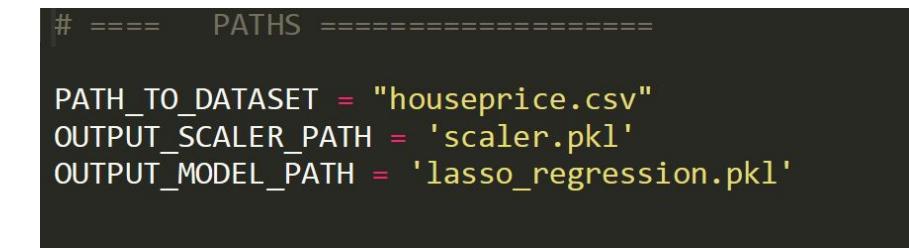
Procedural Programming: hard-coded values



```
C:\Users\Sole\Documents\Repositories\udemy-dmlm\Writing-Production-Code\ProceduralProgramming\config.py...
File Edit Selection Find View Goto Tools Project Preferences Help
train.py score.py preprocessing_functions.py config.py

10
11     # imputation parameters
12     LOTFRONTAGE_MODE = 60
13
14
15     # encoding parameters
16     FREQUENT_LABELS = {
17         'MSZoning': ['FV', 'RH', 'RL', 'RM'],
18         'Neighborhood': ['Blmgtn', 'BrDale', 'BrkSide', 'ClearCr', 'CollgCr',
19             'Crawfor', 'Edwards', 'Gilbert', 'IDOTRR', 'MeadowV',
20             'Mitchel', 'NAmes', 'NWAmes', 'NoRidge', 'Nridgt',
21             'OldTown', 'SWISU', 'Sawyer', 'SawyerW', 'Somerst',
22             'StoneBr', 'Timber'],
23         'RoofStyle': ['Gable', 'Hip'],
24         'MasVnrType': ['BrkFace', 'None', 'Stone'],
25         'BsmtQual': ['Ex', 'Fa', 'Gd', 'Missing', 'TA'],
26         'BsmtExposure': ['Av', 'Gd', 'Missing', 'Mn', 'No'],
27         'HeatingQC': ['Ex', 'Fa', 'Gd', 'TA'],
28         'CentralAir': ['N', 'Y'],
29         'KitchenQual': ['Ex', 'Fa', 'Gd', 'TA'],
30         'FireplaceQu': ['Ex', 'Fa', 'Gd', 'Missing', 'Po', 'TA'],
31         'GarageType': ['Attchd', 'Basment', 'BuiltIn', 'Detchd', 'Missing'],
32         'GarageFinish': ['Fin', 'Missing', 'RFn', 'Unf'],
33         'PavedDrive': ['N', 'P', 'Y']}
34
35
36 ENCODING_MAPPINGS = {'MSZoning': {'Rare': 0, 'RM': 1, 'RH': 2, 'RL': 3, 'FV': 4},
37     'Neighborhood': {'IDOTRR': 0, 'MeadowV': 1, 'BrDale': 2, 'Edwards': 3,
38         'BrkSide': 4, 'OldTown': 5, 'Sawyer': 6, 'SWISU': 7, 'NAmes': 8,
39         'Mitchel': 9, 'SawyerW': 10, 'Rare': 11, 'NWAmes': 12, 'Gilbert': 13,
40         'Blmgtn': 14, 'CollgCr': 15, 'Crawfor': 16, 'ClearCr': 17, 'Somerst': 18,
41         'StoneBr': 19, 'Nridgt': 20, 'NoRidge': 21, 'NoRidge': 22},
42     'RoofStyle': {'Gable': 0, 'Rare': 1, 'Hip': 2},
43     'MasVnrType': {'None': 0, 'Rare': 1, 'BrkFace': 2, 'Stone': 3},
44     'BsmtQual': {'Missing': 0, 'Fa': 1, 'TA': 2, 'Gd': 3, 'Ex': 4},
45     'BsmtExposure': {'Missing': 0, 'No': 1, 'Mn': 2, 'Av': 3, 'Gd': 4},
46     'HeatingQC': {'Rare': 0, 'Fa': 1, 'TA': 2, 'Gd': 3, 'Ex': 4},
47     'CentralAir': {'N': 0, 'Y': 1},
48     'KitchenQual': {'Fa': 0, 'TA': 1, 'Gd': 2, 'Ex': 3},
49     'FireplaceQu': {'Po': 0, 'Missing': 1, 'Fa': 2, 'TA': 3, 'Gd': 4, 'Ex': 5},
50     'GarageType': {'Missing': 0, 'Rare': 1, 'Detchd': 2, 'Basment': 3, 'Attchd': 4,
51     'GarageFinish': {'Missing': 0, 'Unf': 1, 'RFn': 2, 'Fin': 3},
52     'PavedDrive': {'N': 0, 'P': 1, 'Y': 2}}
53
54     # ===== FEATURE GROUPS =====
55
56 # ======
```

- Straightforward
- Hard-code parameters
- Save multiple objects or data structures



```
# ===== PATHS =====
PATH_TO_DATASET = "houseprice.csv"
OUTPUT_SCALER_PATH = 'scaler.pkl'
OUTPUT_MODEL_PATH = 'lasso_regression.pkl'
```

Object Oriented Programming - OOP

In **Object-oriented programming (OOP)** we write code in the form of “objects”.

This “objects” can store **data**, and can also store instructions or **procedures** to modify that data.

- Data ⇒ attributes
- Instructions or procedures ⇒ methods

Custom ML Pipeline: OOP

In **Object-oriented programming (OOP)** the “objects” can learn and store this parameters

- Parameters get automatically refreshed every time model is re-trained
- No need of manual hard-coding
- Methods:
 - Fit: to learn parameters
 - Saves the parameter in object attribute
 - Transform: to transform data with the learnt parameters
 - Calls to the object attribute to recall the parameters
- Attributes: store the learn parameters

Custom ML Pipeline: Pipeline

A **pipeline** is a set of data processing steps connected in series, where typically, the output of one element is the input of the next one.

The elements of a pipeline can be executed in parallel or in time-sliced fashion. This is useful when we require use of big data, or high computing power, e.g., for neural networks.

Custom ML Pipeline: Summary

A custom Machine Learning pipeline is therefore a sequence of steps, aimed at loading and transforming the data, to get it ready for training or scoring, where:

- We write the processing steps as objects (OOP)
- We write the sequence, i.e., the pipeline as objects (OOP)

We save one object, the pipeline, as a Python pickle, with all the information needed to transform the raw inputs and get the predictions

Custom Pipeline: Overview

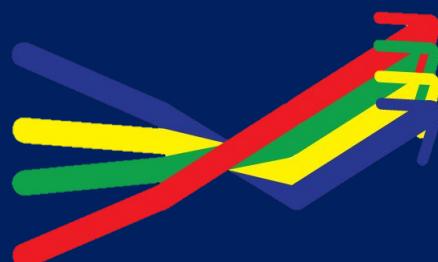
Advantages

- Can be tested, versioned, tracked and controlled
- Can build future models on top
- Good software developer practice
- Built to satisfy business needs

Disadvantages

- Requires team of software developers to build and maintain
- Overhead for DS to familiarise with code for debugging or adding on future models
- Preprocessor not reusable, need to re-write Preprocessor class for each new ML model
- Need to write new pipeline for each new ML model
- Lacks versatility, may constrain DS to what is available with the implemented pipeline

Third Party Machine Learning Pipeline: Leveraging the power of Scikit-Learn



Scikit-Learn and sklearn pipeline

Scikit-Learn is a Python library that provides a solid implementation of a range of machine learning algorithms.

Scikit-Learn provides efficient versions of a large number of common algorithms.

Scikit-Learn is characterised by a clean, uniform, and streamlined API.

Scikit-Learn is written so that most of its algorithms follow the same functionality

Once you understand the basic use and syntax of Scikit-Learn for one type of model, switching to a new model or algorithm is very straightforward

Scikit-Learn provides useful and complete online documentation that allows you to understand both what the algorithm is about and how to use it from scikit-learn

Scikit-Learn is so well established in the community, that new packages are typically designed following scikit-learn functionality to be quickly adopted by end users, e.g, Keras, MLXtend.

Scikit-Learn and sklearn pipeline

Scikit-Learn Objects

- **Transformers** - class that have fit and transform method, it transforms data
 - Scalers
 - Feature selectors
 - One hot encoders
- **Predictor** - class that has fit and predict methods, it fits and predicts.
 - Any ML algorithm like lasso, decision trees, svm, etc
- **Pipeline** - class that allows you to list and run transformers and predictors in sequence
 - All steps should be transformers except the last one
 - Last step should be a predictor

Scikit-Learn and sklearn pipeline

[Here](#) is a good example of Pipeline usage. Pipeline gives you a single interface for all 3 steps of transformation and resulting estimator. It encapsulates transformers and predictors inside, and now you can do something like:

```
1 vect = CountVectorizer()
2 tfidf = TfidfTransformer()
3 clf = SGDClassifier()
4
5 vX = vect.fit_transform(Xtrain)
6 tfidfX = tfidf.fit_transform(vX)
7 predicted = clf.fit_predict(tfidfX)
8
9 # Now evaluate all steps on test set
10 vX = vect.transform(Xtest)
11 tfidfX = tfidf.transform(vX)
12 predicted = clf.predict(tfidfX)
```

With just:

```
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', SGDClassifier()),
])
predicted = pipeline.fit(Xtrain).predict(Xtrain)
# Now evaluate all steps on test set
predicted = pipeline.predict(Xtest)
```

[Taken from stackoverflow](#)

Scikit-Learn and sklearn pipeline

Feature Creation and Feature Engineering steps as Scikit-Learn Objects

- **Transformers** - class that have fit and transform method, it transforms data
- Use of scikit-learn base transformers
 - Inherit class and adjust the fit and transform methods

Scikit-Learn and sklearn pipeline

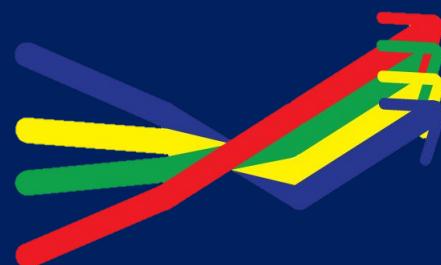
Advantages

- Can be tested, versioned, tracked and controlled
- Can build future models on top
- Good software developer practice
- Leverages the power of acknowledged API
- Data scientists familiar with Pipeline use, reduced over-head
- Engineering steps can be packaged and re-used in future ML models

Disadvantages

- Requires team of software developers to build and maintain
- Overhead for software developers to familiarise with code for sklearn API ⇒ difficulties debugging

Dive into Scikit-learn API



Scikit-Learn Objects

- **Estimators**
- **Transformers**
- **Pipeline**

Scikit-Learn Estimators

Estimator - A class with fit() and predict() methods.

It fits and predicts.

Any ML algorithm like Lasso, Decision trees, SVMs, are coded as estimators within Scikit-Learn.

```
class Estimator(object):  
  
    def fit(self, X, y=None):  
        """  
        Fits the estimator to data.  
        """  
        return self  
  
    def predict(self, X):  
        """  
        Compute the predictions  
        """  
        return predictions
```

Scikit-Learn Transformers

Transformers - class that have fit() and transform() methods.

It transforms data.

- Scalers
- Feature selectors
- One hot encoders

➤ This is the core of Feature Engine

```
class Transformer(object):  
  
    def fit(self, X, y=None):  
        """  
        Learn the parameters to  
        engineer the features  
        """  
  
    def transform(X):  
        """  
        Transforms the input data  
        """  
        return X_transformed
```

Scikit-Learn Pipeline

Pipeline - class that allows to run transformers and estimators in sequence.

- Most steps are Transformers
- Last step can be an Estimator

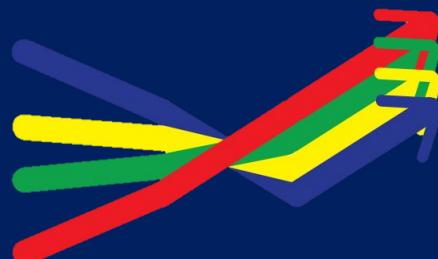
```
class Pipeline(Transformer):  
  
    @property  
    def name_steps(self):  
        """Sequence of transformers  
        """  
        return self.steps  
  
    @property  
    def _final_estimator(self):  
        """  
        Estimator  
        """  
        return self.steps[-1]
```

Scikit-Learn API documentation

<https://scikit-learn.org/stable/modules/classes.html>

- [base.BaseEstimator](#)
- [base.TransformerMixin](#)

Packaging the transformers to re-use across projects



Scikit-Learn API documentation

```
from preprocessors import CategoricalImputer
```

Scikit-Learn Transformers



- Missing Data Imputation
 - SimpleImputer
 - IterativeImputer
- Categorical Variable Encoding
 - OneHotEncoder
 - OrdinalEncoder
- Discretisation
 - KBinsDiscretizer
- Variable Transformation
 - PowerTransformer
 - FunctionTransformer

Feature Engine Transformers



- Missing Data Imputation
 - [MeanMedianImputer](#)
 - [RandomSampleImputer](#)
 - [EndTailImputer](#)
 - [AddNaNBinaryImputer](#)
 - [CategoricalVariableImputer](#)
 - [FrequentCategoryImputer](#)
- Categorical Variable Encoding
 - [CountFrequencyCategoricalEncoder](#)
 - [OrdinalCategoricalEncoder](#)
 - [MeanCategoricalEncoder](#)
 - [WoERatioCategoricalEncoder](#)
 - [OneHotCategoricalEncoder](#)
 - [RareLabelCategoricalEncoder](#)
- Outlier Removal
 - [Windsorizer](#)
 - [ArbitraryOutlierCapper](#)
- Discretisation
 - [EqualFrequencyDiscretiser](#)
 - [EqualWidthDiscretiser](#)
 - [DecisionTreeDiscretiser](#)
- Variable Transformation
 - [LogTransformer](#)
 - [ReciprocalTransformer](#)
 - [ExponentialTransformer](#)
 - [BoxCoxTransformer](#)

Category Encoders

The screenshot shows a web browser displaying the [Category Encoders](https://contrib.scikit-learn.org/categorical-encoding/) documentation. The page has a dark blue header with the title "Category Encoders" and a "latest" link. Below the header is a search bar labeled "Search docs". The main content area lists various encoder classes: Backward Difference Coding, BaseN, Binary, CatBoost Encoder, Hashing, Helmert Coding, James-Stein Encoder, Leave One Out, M-estimate, One Hot, Ordinal, Polynomial Coding, Sum Coding, Target Encoder, and Weight of Evidence. To the right of the sidebar, there is a "Category Encoders" section with a brief description of the library's purpose and a bulleted list of its features. Below this is a "Usage" section with installation instructions for pip and conda, followed by code examples for importing the library and using its encoders.

Category Encoders

A set of scikit-learn-style transformers for encoding categorical variables into numeric with different techniques. While ordinal, one-hot, and hashing encoders have similar equivalents in the existing scikit-learn version, the transformers in this library all share a few useful properties:

- First-class support for pandas dataframes as an input (and optionally as output)
- Can explicitly configure which columns in the data are encoded by name or index, or infer non-numeric columns regardless of input type
- Can drop any columns with very low variance based on training set optionally
- Portability: train a transformer on data, pickle it, reuse it later and get the same thing out.
- Full compatibility with sklearn pipelines, input an array-like dataset like any other transformer

Usage

install as:

```
pip install category_encoders
```

or

```
conda install -c conda-forge category_encoders
```

To use:

```
import category_encoders as ce

encoder = ce.BackwardDifferenceEncoder(cols=[...])
encoder = ce.BaseNEncoder(cols=[...])
encoder = ce.BinaryEncoder(cols=[...])
encoder = ce.CatBoostEncoder(cols=[...])
encoder = ce.HashingEncoder(cols=[...])
encoder = ce.HelmertEncoder(cols=[...])
encoder = ce.JamesSteinEncoder(cols=[...])
encoder = ce.LeaveOneOutEncoder(cols=[...])
encoder = ce.MEstimateEncoder(cols=[...])
encoder = ce.OneHotEncoder(cols=[...])
encoder = ce.OrdinalEncoder(cols=[...])
encoder = ce.SumEncoder(cols=[...])
encoder = ce.PolynomialEncoder(cols=[...])
encoder = ce.TargetEncoder(cols=[...])
encoder = ce.WOEEncoder(cols=[...])

encoder.fit(X, y)
X_cleaned = encoder.transform(X_dirty)
```

Pipeline with Feature-engine

The screenshot shows the homepage of the feature-engine documentation. It features a dark blue header with the title "feature-engine" and "latest". Below the header is a search bar with the placeholder "Search docs". A "TABLE OF CONTENTS" section follows, listing various topics under "Quick Start" and "Missing data imputation: imputers". The "Missing data imputation: imputers" section is expanded, showing sub-topics like "Categorical variable encoding: encoders", "Variable transformation: variable transformers", "Variable discretisation: discretisers", "Outlier capping: cappers", and "Changelog".

```
from math import sqrt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline as pipe
from sklearn.preprocessing import MinMaxScaler

from feature_engine import categorical_encoders as ce
from feature_engine import discretisers as dsc
from feature_engine import missing_data_imputers as mdi

# Load dataset
data = pd.read_csv('houseprice.csv')

# drop some variables
data.drop(labels=['YearBuilt', 'YearRemodAdd', 'GarageYrBlt', 'Id'], axis=1, inplace=True)

# categorical encoders work only with object type variables
data[discrete] = data[discrete].astype('O')

# separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data.drop(labels=['SalePrice'], axis=1),
                                                    data.SalePrice,
                                                    test_size=0.1,
                                                    random_state=0)

# set up the pipeline
price_pipe = pipe([
    # add a binary variable to indicate missing information for the 2 variables below
    ('continuous_var_imputer', mdi.AddNaNBinaryImputer(variables = ['LotFrontage'])),

    # replace NA by the median in the 2 variables below, they are numerical
    ('continuous_var_median_imputer', mdi.MeanMedianImputer(imputation_method='median', variables = ['LotFrontage', 'MasVnrArea'])),

    # replace NA by adding the label "Missing" in categorical variables
    ('categorical_imputer', mdi.CategoricalVariableImputer(variables = categorical)),

    # discretise numerical variables using trees
    ('numerical_tree_discretiser', dsc.DecisionTreeDiscretiser(cv = 3, scoring='neg_mean_squared_error')),

    # remove rare labels in categorical and discrete variables
    ('rare_label_encoder', ce.RareLabelCategoricalEncoder(tol = 0.03, n_categories=1, variables = categorical)),

    # encode categorical and discrete variables using the target mean
    ('categorical_encoder', ce.MeanCategoricalEncoder(variables = categorical+discrete)),

    # scale features
    ('scaler', MinMaxScaler()),

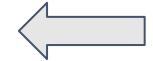
    # Lasso
    ('lasso', Lasso(random_state=2909, alpha=0.005))
])

# train feature engineering transformers and Lasso
price_pipe.fit(X_train, np.log(y_train))

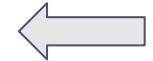
# predict
pred_train = price_pipe.predict(X_train)
pred_test = price_pipe.predict(X_test)
```



Import predefined transformers



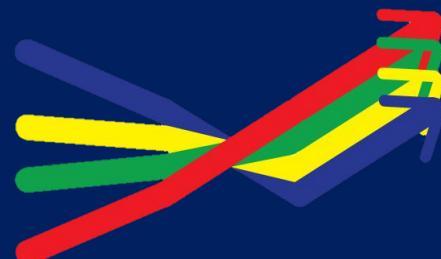
Accommodate transformers in the pipeline



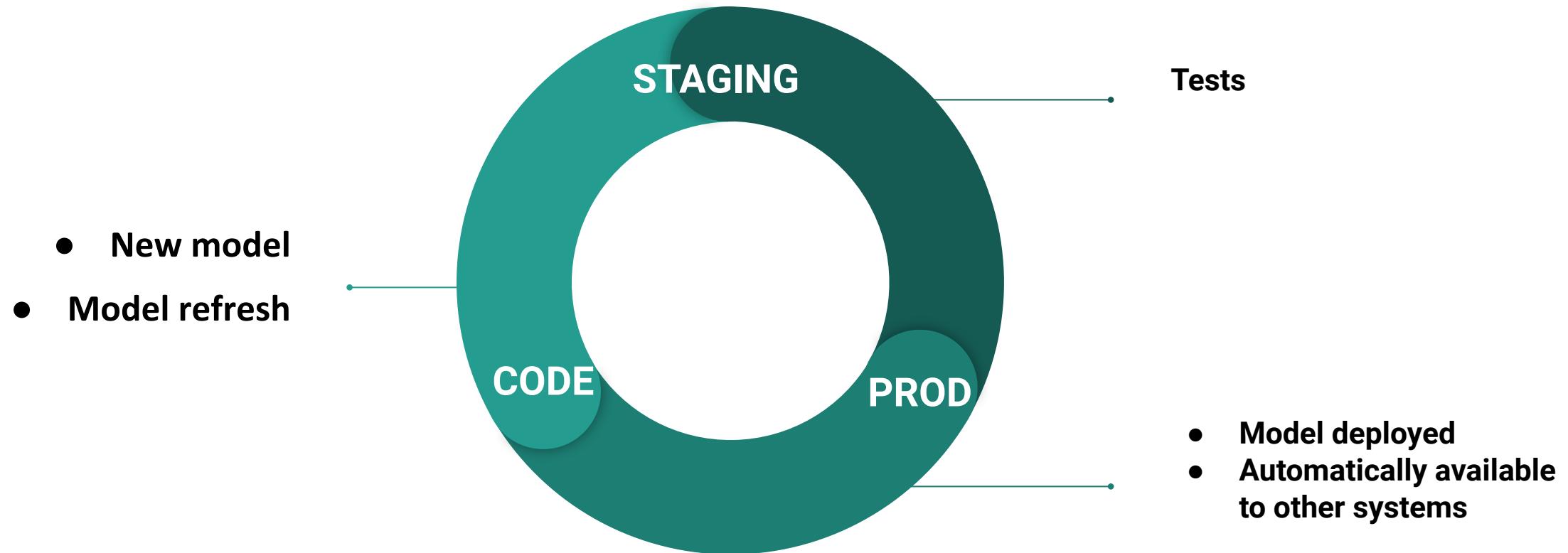
Fit the pipeline and make predictions



Should feature selection be part of the Machine Learning automated pipeline?



Feature selection in CI/CD



Feature selection in CI/CD

Advantages

- Reduced overhead in the implementation of the new model
- The new model is almost immediately available to the business systems

Disadvantages

- Lack of data versatility
- No additional data can be fed through the pipeline, as the entire processes are based on the first dataset on which it was built

Feature selection in CI/CD

Including a feature selection algorithm as part of the pipeline,

- Ensures that from all the available features only the most useful ones are selected to train the model
- Potentially avoids overfitting
- Enhances model interpretability

However,

- We would need to deploy code to engineer all available features in the dataset, regardless of whether they will be finally used by the model
- Error handling and unit testing for all the code to engineering features

Feature selection in CI/CD

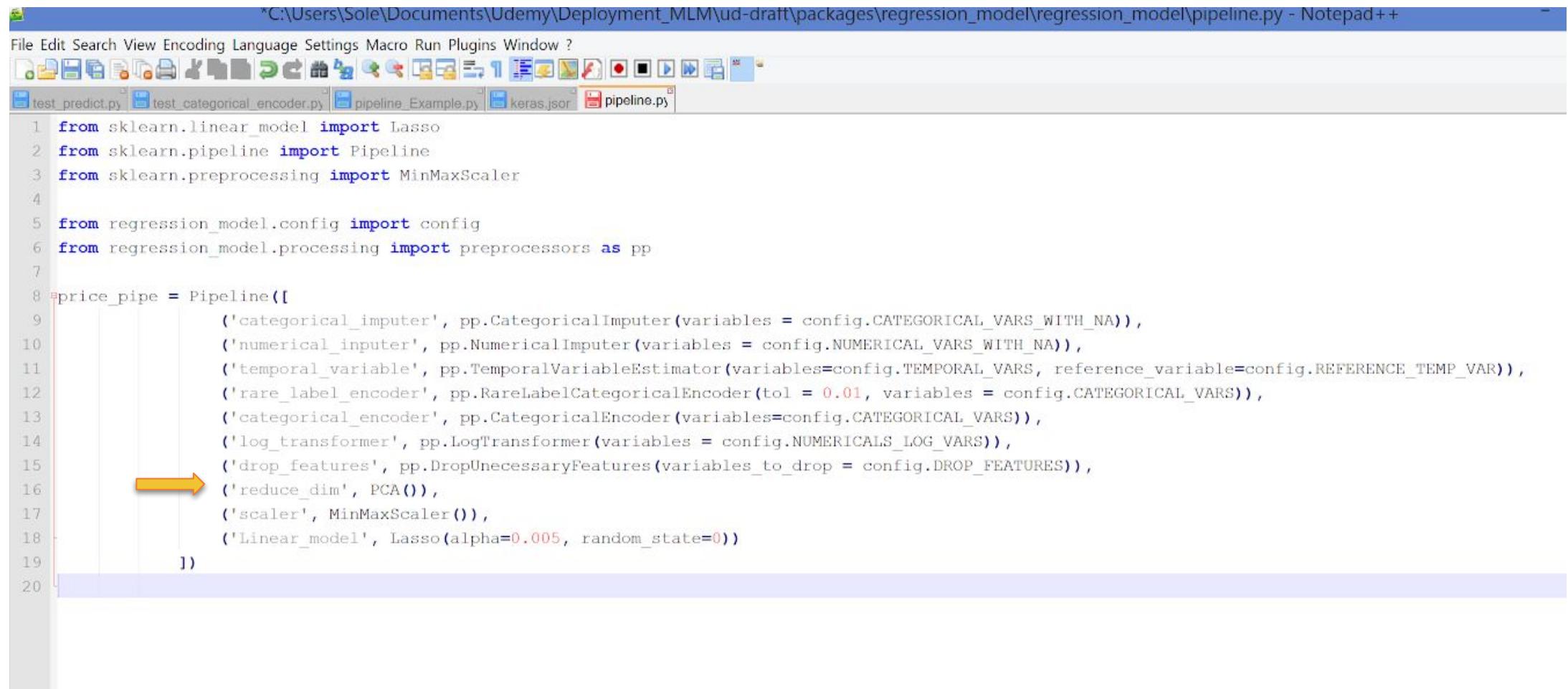
Suitable:

- Model build and refresh on same data
- Model build and refresh on smaller datasets

Not suitable,

- If model is built using datasets with a high feature space
- If model is constantly enriched with new data sources

Feature selection in the scikit-learn pipe



The screenshot shows a Notepad++ window displaying a Python script named `pipeline.py`. The script defines a pipeline for regression modeling. A yellow arrow points to the closing brace of the pipeline definition at line 19.

```
*C:\Users\Sole\Documents\Udemy\Deployment_MLM\ud-draft\packages\regression_model\regression_model\pipeline.py - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
test_predict.py test_categorical_encoder.py pipeline.Example.py keras.json pipeline.py
1 from sklearn.linear_model import Lasso
2 from sklearn.pipeline import Pipeline
3 from sklearn.preprocessing import MinMaxScaler
4
5 from regression_model.config import config
6 from regression_model.processing import preprocessors as pp
7
8 price_pipe = Pipeline([
9     ('categorical_imputer', pp.CategoricalImputer(variables = config.CATEGORICAL_VARS_WITH_NA)),
10    ('numerical_imputer', pp.NumericalImputer(variables = config.NUMERICAL_VARS_WITH_NA)),
11    ('temporal_variable', pp.TemporalVariableEstimator(variables=config.TEMPORAL_VARS, reference_variable=config.REFERENCE_TEMP_VAR)),
12    ('rare_label_encoder', pp.RareLabelCategoricalEncoder(tol = 0.01, variables = config.CATEGORICAL_VARS)),
13    ('categorical_encoder', pp.CategoricalEncoder(variables=config.CATEGORICAL_VARS)),
14    ('log_transformer', pp.LogTransformer(variables = config.NUMERICALS_LOG_VARS)),
15    ('drop_features', pp.DropUnnecessaryFeatures(variables_to_drop = config.DROP_FEATURES)),
16    ('reduce_dim', PCA()),
17    ('scaler', MinMaxScaler()),
18    ('Linear_model', Lasso(alpha=0.005, random_state=0))
19 ])
20
```

Feature selection transformer skeleton

```
cat_producer.py  test_categorical_encoder.py  pipeline_Example.py  notes.ipynb  -r requirements.txt
import numpy as np
import pandas as pd

from sklearn.base import BaseEstimator, TransformerMixin

from regression_model.preprocessing import errors


# categorical missing value imputer
class MySpecificSelector(BaseEstimator, TransformerMixin):

    def __init__(self, some_param=None):
        self.some_param = some_param

    def fit(self, X, y=None):
        '''Code to select features. Any of your liking'''
        self.selected_features = output of the above code, the selected fetures

        return self

    def transform(self, X):
        X = X.copy()
        X = X[self.selected_features]
        return X
```