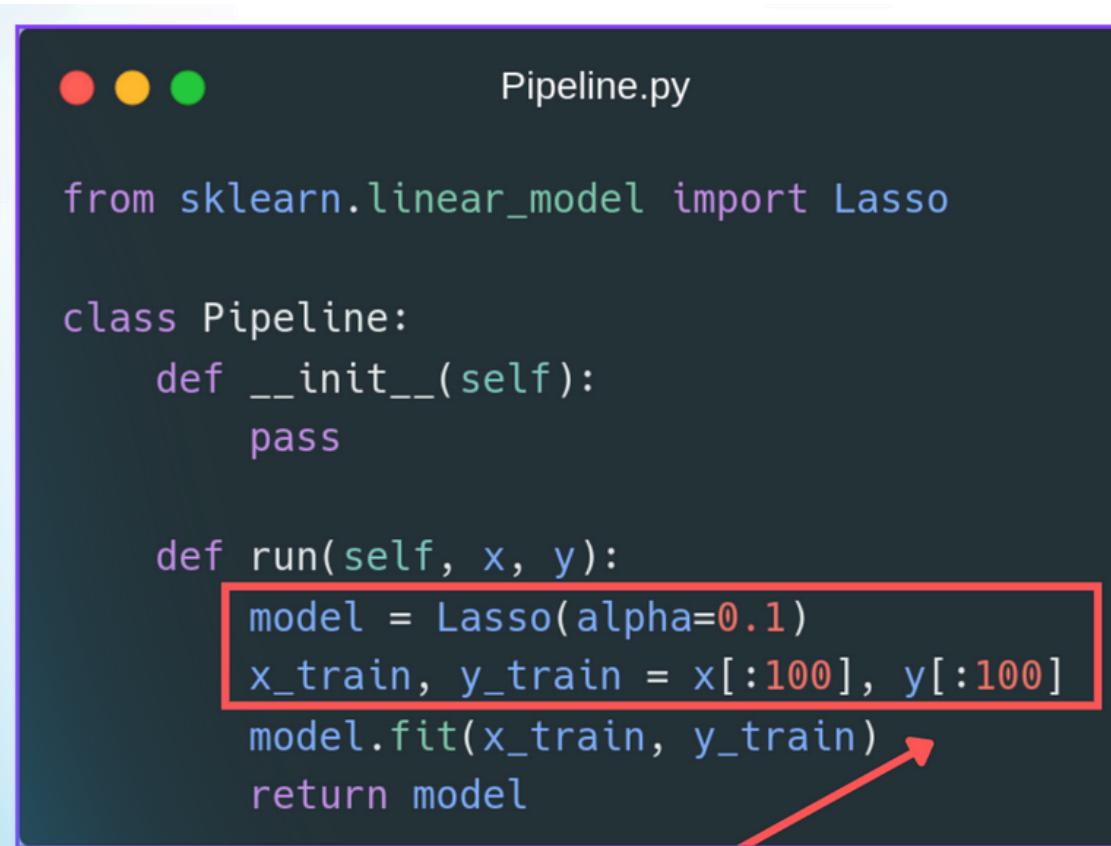


TOP 3 BAD CODING PRACTICES DATA SCIENTISTS should avoid



Pipeline.py

```
from sklearn.linear_model import Lasso

class Pipeline:
    def __init__(self):
        pass

    def run(self, x, y):
        model = Lasso(alpha=0.1)
        x_train, y_train = x[:100], y[:100]
        model.fit(x_train, y_train)
        return model
```

A screenshot of a code editor showing a Python file named 'Pipeline.py'. The code defines a class 'Pipeline' with a constructor and a method 'run'. Inside 'run', a Lasso model is created with 'alpha=0.1', and training data is selected from the first 100 elements of x and y. A red box highlights the line 'model = Lasso(alpha=0.1)'. A red arrow points from the text 'Hardcoded parameters inside the method' at the bottom left to this highlighted line.

**Hardcoded parameters
inside the method**



TIMUR BIKMUKHAMEDOV



1. Hardcoding ML pipeline and model configuration parameters



Pipeline.py

```
from sklearn.linear_model import Lasso

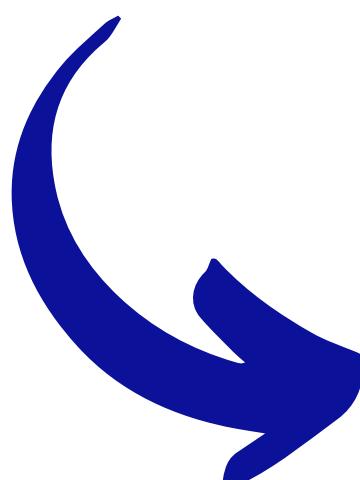
class Pipeline:
    def __init__(self):
        pass

    def run(self, x, y):
        model = Lasso(alpha=0.1)
        x_train, y_train = x[:100], y[:100]
        model.fit(x_train, y_train)
        return model
```

Hardcoded parameters
inside the method

1.8

Why is it bad?





Pipeline.py

```
from sklearn.linear_model import Lasso

class Pipeline:
    def __init__(self):
        pass

    def run(self, x, y):
        model = Lasso(alpha=0.1)
        x_train, y_train = x[:100], y[:100]
        model.fit(x_train, y_train)
        return model
```

Hardcoding leads to:

- Difficulty in making changes if the codebase is big
- Difficulty in adapting parameters for similar modeled objects without lengthy "if-else" statements.
- Need for re-deployment if the parameters need to be changed.

Instead ...





SAVE/LIKE FOR LATER

1. Create a config file (.yaml should do)

2. Initialize config as a class attribute

```
● ● ● Pipeline.py

from sklearn.linear_model import Lasso

class Pipeline:
    def __init__(self, config):
        self.config = config

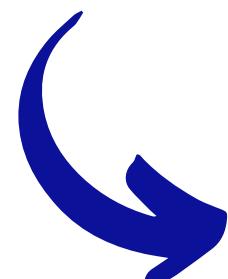
    def run(self, x, y):
        model = Lasso(alpha=self.config['alpha'])
        x_train = x[:self.config['train_size']]
        y_train = y[:self.config['train_size']]
        model.fit(x_train, y_train)
        return model
```

3. Get values from the attribute

Config allows:

- Easy finding and changing parameters in one place
- Avoiding re-deployment of an ML application, parameters can be changed on the fly
- Flexible selection of these parameters in the code, e.g. hyperparameters tuning.

Next!





2. Ignoring Modularization

Pipeline.py

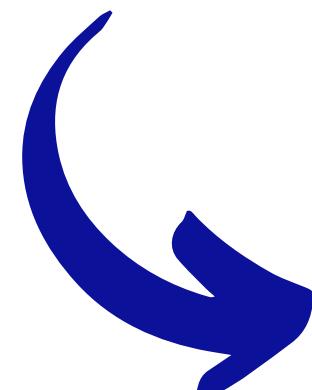
```
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

class Pipeline:
    def __init__(self, config):
        self.config = config

    def run(self, df):
        x_train, x_test, y_train, y_test = train_test_split(
            df.drop('target', axis=1),
            df['target'],
            test_size=self.config['test_size'])
        scaler_x = StandardScaler()
        scaler_y = StandardScaler()
        x_train_sc = scaler_x.fit_transform(x_train)
        y_train_sc = scaler_y.fit_transform(y_train)
        model = Lasso(alpha=self.config['alpha'])
        model.fit(x_train_sc, y_train_sc)
        return model
```

1 **Splitting, scaling, and fitting
are done in one function**

Why is it bad?





```
 Pipeline.py

from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

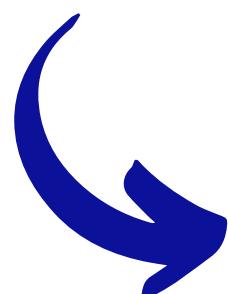
class Pipeline:
    def __init__(self, config):
        self.config = config

    def run(self, df):
        x_train, x_test, y_train, y_test = train_test_split(
            df.drop('target', axis=1),
            df['target'],
            test_size=self.config['test_size'])
        scaler_x = StandardScaler()
        scaler_y = StandardScaler()
        x_train_sc = scaler_x.fit_transform(x_train)
        y_train_sc = scaler_y.fit_transform(y_train)
        model = Lasso(alpha=self.config['alpha'])
        model.fit(x_train_sc, y_train_sc)
        return model
```

Ignoring modularization leads to:

- Poor code readability and difficulty in following data transformation
- Difficult maintenance and testing
- Code repetition and limited reusability

Instead ...





Split the code into functions (class methods)

```
● ● ● Pipeline.py

from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

class Pipeline:
    def __init__(self, config):
        self.config = config

    def get_train_test_data(self, df):
        x_train, x_test, y_train, y_test = train_test_split(
            df.drop('target', axis=1),
            df['target'],
            test_size=self.config['test_size'])
        return x_train, x_test, y_train, y_test

    @staticmethod
    def scale_data(x, y):
        scaler_x = StandardScaler()
        scaler_y = StandardScaler()
        x_sc = scaler_x.fit_transform(x)
        y_sc = scaler_y.fit_transform(y)
        return x_sc, y_sc

    def fit_model(self, x, y):
        model = Lasso(alpha=self.config['alpha'])
        model.fit(x, y)
        return model

    def run(self, df):
        x_train, x_test, y_train, y_test = self.get_train_test_data(df)
        x_train_sc, y_train_sc = self.scale_data(x_train, y_train)
        model = self.fit_model(x_train_sc, y_train_sc)
        return model
```

Ideally, one function should perform one task

Good practice for “run” method is only to call other methods





3. Avoiding type annotations and documenting the code



Pipeline.py

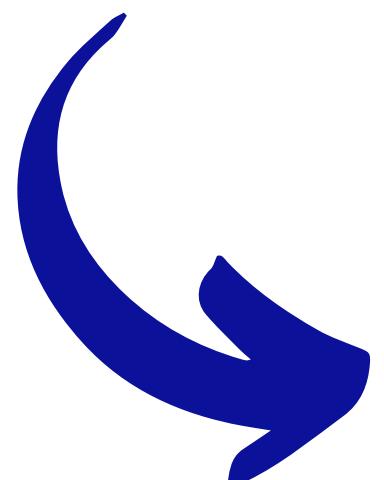
```
from sklearn.preprocessing import StandardScaler

def scale_data(x, y):
    scaler_x = StandardScaler()
    scaler_y = StandardScaler()
    x_sc = scaler_x.fit_transform(x)
    y_sc = scaler_y.fit_transform(y)
    return x_sc, y_sc
```

No annotation
and docstring

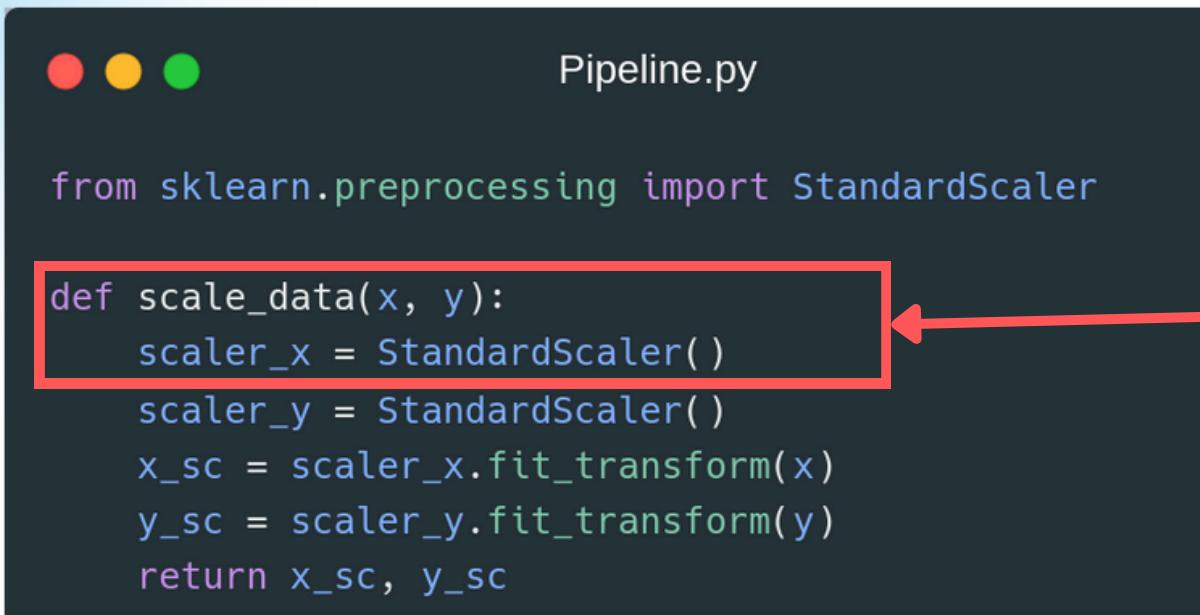
1.3

Why is it bad?





SAVE/LIKE FOR LATER



```
from sklearn.preprocessing import StandardScaler

def scale_data(x, y):
    scaler_x = StandardScaler()
    scaler_y = StandardScaler()
    x_sc = scaler_x.fit_transform(x)
    y_sc = scaler_y.fit_transform(y)
    return x_sc, y_sc
```

No annotation
and docstring

This leads to:

- Poor code readability, especially for other developers
- Inability to check type-related issues for linters
- Poor maintainability, especially in big codebases
- Inability to produce high-quality code documentation

1.8

Instead ...





SAVE/LIKE FOR LATER

Add typing annotations and docstrings

```
Pipeline.py

def scale_data(x: pd.DataFrame, y: pd.Series) -> Tuple[np.ndarray, np.ndarray]:
    """
    Scales the input features and target column using StandardScaler.

    Parameters:
    x: The input features as a pandas DataFrame.
    y: The target column as a pandas Series.

    Returns:
    x_sc, y_sc: a tuple containing the scaled features and scaled target as numpy arrays.
    """
    scaler_x = StandardScaler()
    scaler_y = StandardScaler()
    x_sc = scaler_x.fit_transform(x)
    y_sc = scaler_y.fit_transform(y)
    return x_sc, y_sc
```

Even in this simple case, we can see that we change the data type from a dataframe to NumPy arrays

Let's see tips altogether!





Bad practices code

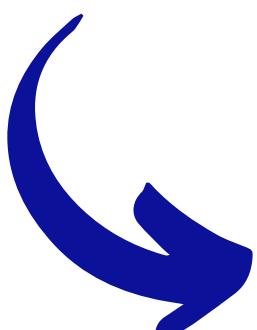


Pipeline.py

```
class Pipeline:  
    def __init__(self):  
        pass  
  
    def run(self, df):  
        x_train, x_test, y_train, y_test = train_test_split(  
            df.drop('target', axis=1),  
            df['target'],  
            test_size=0.2)  
        scaler_x = StandardScaler()  
        scaler_y = StandardScaler()  
        x_train_sc = scaler_x.fit_transform(x_train)  
        y_train_sc = scaler_y.fit_transform(y_train)  
        model = Lasso(alpha=0.1)  
        model.fit(x_train_sc, y_train_sc)  
        return model
```

1. • **Hardcoded**
- **Monolithic**
- **Hard to test**
- **Hard to understand if the code growths**

Instead ...



Best practices code

```

● ● ● Pipeline.py

class Pipeline:
    def __init__(self, config: Dict[Any, Any]):
        """
        Initializes the Pipeline with the given configuration file.
        """
        self.config = config

    def get_train_test_data(self, df: pd.DataFrame) -> Tuple[pd.DataFrame, pd.DataFrame, pd.Series, pd.Series]:
        """
        Splits the DataFrame into training and testing sets.

        Parameters:
        df: The input DataFrame containing features and target.

        Returns:
        x_train, x_test, y_train, y_test: The training and testing sets for features and target.
        """
        x_train, x_test, y_train, y_test = train_test_split(
            df.drop('target', axis=1),
            df['target'],
            test_size=self.config['test_size'])
        return x_train, x_test, y_train, y_test

    @staticmethod
    def scale_data(x: pd.DataFrame, y: pd.Series) -> Tuple[np.ndarray, np.ndarray]:
        """
        Scales the features and target using StandardScaler.

        Parameters:
        x (pd.DataFrame): The input features as a pandas DataFrame.
        y (pd.Series): The target column as a pandas Series.

        Returns:
        x_sc, y_sc: The scaled features and target as numpy arrays.
        """
        scaler_x = StandardScaler()
        scaler_y = StandardScaler()
        x_sc = scaler_x.fit_transform(x)
        y_sc = scaler_y.fit_transform(y.values.reshape(-1, 1)).flatten()
        return x_sc, y_sc

    def fit_model(self, x: np.ndarray, y: np.ndarray) -> BaseEstimator:
        """
        Fits a Lasso model to the scaled training data.

        Parameters:
        x (np.ndarray): The scaled training features.
        y (np.ndarray): The scaled training target.

        Returns:
        model: The trained Lasso model.
        """
        model = Lasso(alpha=self.config['alpha'])
        model.fit(x, y)
        return model

    def run(self, df: pd.DataFrame) -> BaseEstimator:
        """
        Runs the complete pipeline: data splitting, scaling, and model fitting.

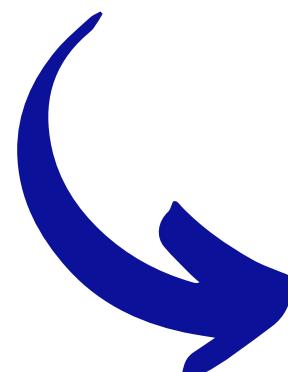
        Parameters:
        df (pd.DataFrame): The input DataFrame containing features and target.

        Returns:
        model: The trained Lasso model.
        """
        x_train, x_test, y_train, y_test = self.get_train_test_data(df)
        x_train_sc, y_train_sc = self.scale_data(x_train, y_train)
        model = self.fit_model(x_train_sc, y_train_sc)
        return model

```

- Configurable
- Less error-prone
- Easy to read
- Easy to maintain
- Easy to test

You may think it is
too much...





SAVE/LIKE FOR LATER

Short answer - No!

Long answer:

- This code is scalable to more complex pipelines and multiple ML model cases
- In a growing codebase, it will be more maintainable
- It is easy to collaborate with colleagues and future yourself

Finally...



Want to see more tips like this?



Like post

Follow

Follow me

so I know that the content resonates!



TIMUR BIKMUKHAMEDOV