```python
from .consts import Brand

from typing import Optional

import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.compose import TransformedTargetRegressor as TransformedTargetRegressor_
from category_encoders import TargetEncoder
from xgboost import XGBRegressor


class CategoricalEncoder(TransformerMixin, BaseEstimator):
    def fit(
        self, X: pd.DataFrame, y: Optional[pd.Series] = None
    ) -> "CategoricalEncoder":
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()

        for feature in X.select_dtypes(["category"]):
            X[feature] = X[feature].cat.codes

        return X


class BrandModelExtractor(TransformerMixin, BaseEstimator):
    def __init__(
        self,
        name_col: str = "Name",
        brand_col: str = "Brand",
        model_col: str = "Model",
        brand_enum=Brand,
    ):
        self.name_col = name_col
        self.brand_col = brand_col
        self.model_col = model_col
        self.brand_enum = brand_enum

    def fit(self, X: pd.DataFrame, y: Optional[pd.Series]) -> "BrandModelExtractor":
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()

        names = X[self.name_col].str.title()
        brands = pd.Series([None] * len(names), index=names.index, dtype=object)
        models = pd.Series([None] * len(names), index=names.index, dtype=object)

        for brand in Brand:
            condition = names.str.startswith(brand.value, na=False) & brands.isna()

            if condition.any():
                brands.loc[condition] = brand.name
                matched_names = names.loc[condition]
                residuals = matched_names.str[len(brand.value) :].str.strip()
                models.loc[condition] = residuals.where(residuals != "", None)

        brands = brands.rename("Brand").astype("category")
        models = models.rename("Model").astype("category")

        X[self.brand_col] = brands
        X[self.model_col] = models

        return X


class YearToAgeTransformer(TransformerMixin, BaseEstimator):
    def __init__(
        self, year_col: str = "Year", age_col: str = "Age", current_year: int = 2020
    ):
        self.year_col = year_col
        self.age_col = age_col
        self.current_year = current_year

    def fit(
        self, X: pd.DataFrame, y: Optional[pd.Series] = None
    ) -> "YearToAgeTransformer":
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()

        X[self.age_col] = self.current_year - X[self.year_col]
        X.drop(columns=[self.year_col], inplace=True)

        return X


class KilometersDrivenClipper(TransformerMixin, BaseEstimator):
    def __init__(
        self,
        kilometers_driven_col: str = "Kilometers_Driven",
        clipping_quantile: float = 0.995,
    ):
        self.kilometers_driven_col = kilometers_driven_col
        self.clipping_quantile = clipping_quantile

    def fit(
        self, X: pd.DataFrame, y: Optional[pd.Series] = None
    ) -> "KilometersDrivenClipper":
        return self

    def transform(self, X: pd.DataFrame):
        X = X.copy()

        X[self.kilometers_driven_col] = X[self.kilometers_driven_col].clip(
            upper=int(X[self.kilometers_driven_col].quantile(self.clipping_quantile))
        )

        return X


class FuelTypeGrouper(TransformerMixin, BaseEstimator):
    def __init__(
        self,
        fuel_type_col: str = "Fuel_Type",
        target_fuel_types: list[str] = ["CNG", "LPG", "Electric"],
    ):
        self.fuel_type_col = fuel_type_col
        self.target_fuel_types = target_fuel_types
```

```
    def fit(self, X: pd.DataFrame, y: Optional[pd.Series]) -> "FuelTypeGrouper":
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()

        X[self.fuel_type_col] = (
            X[self.fuel_type_col]
            .astype("object")
            .replace(dict((fuel_type, "Other") for fuel_type in self.target_fuel_types))
            .astype("category")
        )

        return X


class MileageClipper(TransformerMixin, BaseEstimator):
    def __init__(
        self,
        mileage_col: str = "Mileage",
        clipping_quantile: float = 0.995,
    ):
        self.mileage_col = mileage_col
        self.clipping_quantile = clipping_quantile

    def fit(self, X: pd.DataFrame, y: Optional[pd.Series] = None) -> "MileageClipper":
        return self

    def transform(self, X: pd.DataFrame):
        X = X.copy()

        X[self.mileage_col] = X[self.mileage_col].clip(
            upper=int(X[self.mileage_col].quantile(self.clipping_quantile))
        )

        return X


class PowerImputer(TransformerMixin, BaseEstimator):
    def __init__(
        self, engine_col: str = "Engine", power_col="Power", clip_negative: bool = True
    ):
        self.engine_col = engine_col
        self.power_col = power_col
        self.clip_negative = clip_negative

    def fit(self, X: pd.DataFrame, y=Optional[pd.Series]) -> "PowerImputer":
        df = X[[self.engine_col, self.power_col]].dropna()
        engines = df[self.engine_col].astype(float)
        powers = df[self.power_col].astype(float)

        # Linear regression
        cov = ((engines - engines.mean()) * (powers - powers.mean())).sum()
        var = ((engines - engines.mean()) ** 2).sum()
        self.slope = cov / var
        self.intercept = powers.mean() - self.slope * engines.mean()

        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()

        mask = X[self.power_col].isna()
        has_engine = X[self.engine_col].notna() & mask
        X.loc[has_engine, self.power_col] = (
            self.slope * X.loc[has_engine, self.engine_col] + self.intercept
        )

        if self.clip_negative:
            X[self.power_col] = X[self.power_col].clip(lower=0)

        return X


class SeatsBinner(TransformerMixin, BaseEstimator):
    def __init__(self, seats_col: str = "Seats"):
        self.seats_col = seats_col

    def fit(self, X: pd.DataFrame, y: Optional[pd.Series]) -> "SeatsBinner":
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()
        seats = X[self.seats_col]

        X[self.seats_col] = seats.replace(
            {
                2: 4,
                4: 4,
                # Small
                5: 5,
                # Standard
                6: 7,
                7: 7,
                8: 7,
                # Large
                9: 9,
                10: 9,
                # Van
            }
        )

        return X


class NewPriceTransformer(TransformerMixin, BaseEstimator):
    def __init__(self, new_price_col: str = "New_Price", func=np.log1p):
        self.new_price_col = new_price_col
        self.missing_col = "Missing_" + new_price_col
        self.func = func

    def fit(self, X: pd.DataFrame, y: Optional[pd.Series]) -> "NewPriceTransformer":
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()

        X[self.missing_col] = X[self.new_price_col].isna().astype(int)
        X[self.new_price_col] = X[self.new_price_col].apply(self.func)

        return X
```

```python
class BrandPowerInteraction(TransformerMixin, BaseEstimator):
    def __init__(
        self,
        brand_col: str = "Brand",
        power_col: str = "Power",
        brand_power_col: str = "Brand-Pwer",
    ):
        self.brand_col = brand_col
        self.power_col = power_col
        self.brand_power_col = brand_power_col

    def fit(self, X: pd.DataFrame, y: Optional[pd.Series]) -> "BrandPowerInteraction":
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()
        X[self.brand_power_col] = X[self.brand_col] * X[self.power_col]
        return X


class ModelPowerInteraction(BaseEstimator, TransformerMixin):
    def __init__(
        self,
        model_col="Model",
        power_col="Power",
        model_power_col="Model-Power",
    ):
        self.model_col = model_col
        self.power_col = power_col
        self.model_power_col = model_power_col

    def fit(self, X: pd.DataFrame, y: Optional[pd.Series]) -> "ModelPowerInteraction":
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()

        X[self.model_power_col] = X[self.model_col] * X[self.power_col]

        return X


class TransformedPriceRegressor(TransformedTargetRegressor_):
    def __init__(
        self,
        regressor,
        func=np.log1p,
        inverse_func=np.expm1,
    ):
        super().__init__(regressor=regressor, func=func, inverse_func=inverse_func)


def build_feature_engineering_pipeline(regressor) -> Pipeline:
    return Pipeline(
        [
            ("extract_brand_model", BrandModelExtractor()),
            (
                "target_encode",
                TargetEncoder(cols=["Brand", "Model", "Name", "Location"]),
            ),
            ("transform", YearToAgeTransformer()),
            ("Kilometers_Driven_clip_outliers", KilometersDrivenClipper()),
            ("group_infrequent_fuel_type", FuelTypeGrouper()),
            ("mileage_clip_outliers", MileageClipper()),
            ("imput_power", PowerImputer()),
            ("bin_seats", SeatsBinner()),
            ("transform_new_price", NewPriceTransformer()),
            ("brand_power_interaction", BrandPowerInteraction()),
            ("model_power_interaction", ModelPowerInteraction()),
            ("category_encode", CategoricalEncoder()),
            ("model", TransformedPriceRegressor(regressor)),
        ]
    )
```