Etivity_2_CarlosSiqueiraDoAmaral_20151586

October 3, 2021

1 Artificial Intelligence - MSc

1.1 ET5003 - MACHINE LEARNING APPLICATIONS

1.1.1 Instructor: Enrique Naredo

1.1.2 ET5003_Etivity-2

[ ]:
```
# @title Current Date
Today = "2021-10-03"  # @param {type:"date"}
```

[ ]:
```
# @markdown ---
# @markdown ### Enter your details here:
Student_ID = "20151586"  # @param {type:"string"}
Student_full_name = "Carlos Siqueira do Amaral"  # @param {type:"string"}
# @markdown ---
```

[ ]:
```
# @title Notebook information
Notebook_type = "Assignment"  # @param ["Example", "Lab", "Practice",␣
↪"Etivity", "Assignment", "Exam"]
Version = "Final"  # @param ["Draft", "Final"] {type:"raw"}
Submission = True  # @param {type:"boolean"}
```

2 INTRODUCTION

Piecewise regression, extract from Wikipedia:

Segmented regression, also known as piecewise regression or broken-stick regression, is a method in regression analysis in which the independent variable is partitioned into intervals and a separate line segment is fit to each interval.

- Segmented regression analysis can also be performed on multivariate data by partitioning the various independent variables.
- Segmented regression is useful when the independent variables, clustered into different groups, exhibit different relationships between the variables in these regions.
- The boundaries between the segments are breakpoints.

- Segmented linear regression is segmented regression whereby the relations in the intervals are obtained by linear regression.

The goal is to use advanced Machine Learning methods to predict House price.

2.1 Imports

[ ]:
```
import os
import pandas as pd
import numpy as np
import pymc3 as pm
import arviz as az
import theano as tt

# to plot
import matplotlib.colors
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm


# Sklearn
import sklearn.datasets as dt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KernelDensity
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncoder
from sklearn.pipeline import Pipeline

%matplotlib inline
%load_ext lab_black

print(f"pymc3 version: {pm.__version__}")
print(f"arviz version: {az.__version__}")
print(f"theano version: {tt.__version__}")
!python --version
```

```
WARNING (theano.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
```

```
pymc3 version: 3.11.4
arviz version: 0.11.2
theano version: 1.1.2
Python 3.7.11
```

[ ]:
```
# Define the seed so that results can be reproduced
seed = 11
rand_state = 1
```

```
# Define the color maps for plots
# color_map = plt.cm.get_cmap('RdYlBu')
# color_map_discrete = matplotlib.colors.LinearSegmentedColormap.from_list("",␣
↪["red","cyan","magenta","blue"])
```

3 DATASET

Extract from this paper:

- House prices are a significant impression of the economy, and its value ranges are of great concerns for the clients and property dealers.
- Housing price escalate every year that eventually reinforced the need of strategy or technique that could predict house prices in future.
- There are certain factors that influence house prices including physical conditions, locations, number of bedrooms and others.

1. Download the dataset.
2. Upload the dataset into your folder.

The challenge is to predict the final price of each house.

4 Data Preparation

4.1 Loading datasets

[ ]:
```
fpath = os.path.join(os.getcwd(), "house_prices")

train_fname = "house_train.csv"
test_fname = "house_test.csv"
cost_fname = "true_price.csv"

train_set = pd.read_csv(os.path.join(fpath, train_fname)).drop(columns="ad_id")
X_test = pd.read_csv(os.path.join(fpath, test_fname)).drop(columns="ad_id")
y_test = pd.read_csv(os.path.join(fpath, cost_fname)).drop(columns="Id")

print("Loaded training data of shape", train_set.shape)
print("Loaded test data of shape", X_test.shape)
print("Loaded cost data of shape", y_test.shape)
```

```
Loaded training data of shape (2982, 16)
Loaded test data of shape (500, 15)
Loaded cost data of shape (500, 1)
```

There are some entries with null values for `price` in the training set, let's remove them

[ ]:
```
print("Training set rows with no house price:", train_set["price"].isna().sum())

train_set = train_set[train_set["price"].notna()]
```

```
Training set rows with no house price: 90
```

[ ]:
```
train_set.sample(5)
```

[ ]:
```
             area  bathrooms  beds ber_classification  county  \
1321    Dublin 8        1.0   1.0                NaN  Dublin
727      Finglas        2.0   4.0                 C3  Dublin
2194   Rathcoole        2.0   2.0                NaN  Dublin
2900      Santry        1.0   3.0                  F  Dublin
352    Donaghmede       1.0   3.0                 E1  Dublin

                                        description_block environment  \
1321   The property comprises a second floor one bedr...        prod
727    LEONARD WILSON KEENAN ESTATE &amp; LETTING AGE...        prod
2194   RAY COOKE AUCTIONEERS are delighted to present...        prod
2900   Dublin Homes are delighted to present this 3 b...        prod
352    DNG are delighted to present 119 Grange Abbey ...        prod

                                                 facility  \
1321                           Wired for Cable Television
727                                                   NaN
2194                                                  NaN
2900   Gas Fired Central Heating,Wired for Cable Tele...
352                                               Parking

                                                 features   latitude  longitude  \
1321   Superb Condition \nSpacious Second Floor Apart...  53.345125  -6.268618
727                                                 None  53.381444  -6.321115
2194                                                None  53.282229  -6.471410
2900   Kitchen Extension\nPVC double glazed windows \...  53.389088  -6.260865
352    Quiet &amp; Convenient location\nAttic convers...  53.397080  -6.154744

      no_of_units     price property_category   property_type  surface
1321          NaN  295000.0              sale       apartment     50.0
727           NaN  289950.0              sale  end-of-terrace      NaN
2194          NaN  235000.0              sale       apartment    113.8
2900          NaN  370000.0              sale   semi-detached      NaN
352           NaN  325000.0              sale        terraced    101.0
```

4.2 Data Exploration

Some features that could influence house price are more or less intuitive, for example: - area (which could also be encoded with latitude/longitude) - bathrooms - beds - surface

But there are other features which need further analysis to decide whether they should be included or not. - ber_classification - property_type - property_category What are some features that could possibly be useful?

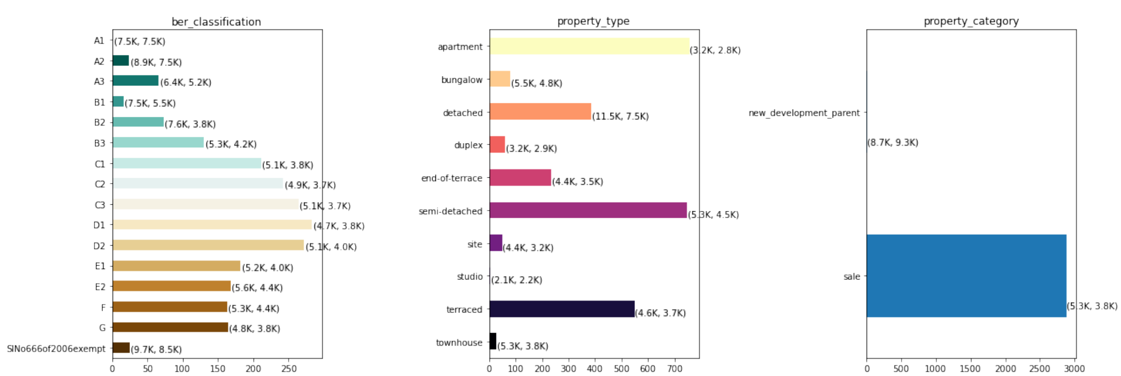
There's also the case of `facility` which could be interesting to analyse but has many values and could add noise.

[ ]:
```python
def add_labels_barh(x, y, axis, **kwargs):
    """Adds label values to a horizontal barplot

    Parameters
    ----------
    x : pd.Series
        Heights of the bars
    y : pd.Series
        The value labels of each bar
    axis : matplotlib.pyplot.axis
        The axes to plot

    Returns
    -------
    None
    """
    for y_pos, x_pos in enumerate(x):
        axis.text(x_pos + 2, y_pos - 0.2, y[y_pos], **kwargs)


def format_prices(x):
    """Helper function to format the mean and median house prices to text.
    To be used as part of a pd.Series.apply method.

    Parameters
    ----------
    x : iterable
        Pair containing mean and median house prices, respectively

    Returns
    -------
        A string with the formatted prices, in hundred thousands

    """
    mean, median = x
    return f"({mean/100_000:.1f}K, {median/100_000:,.1f}K)"


def agg_count_plot(data, feature, target, ax, **plot_kwargs):
    """Aggregates data on a given feature, and creates a horizontal bar plot
    labelled with the mean and median of the target feature in hundreds of␣
 ↪thousands
```

```
    Parameters
    ----------
    data : pd.DataFrame
        Data to aggregate.
    feature : str
        Name of Data column to analyse.
    target : str
        Name of data column with target variable
    ax : matplotlib.pyplot.axis
        The axes to plot

    Returns
    -------
    None
    """
    agg_df = (
        data[[feature, target]]
        .groupby(feature)
        .agg(
            count=(feature, "count"),
            mean_price=(target, "mean"),
            median_price=(target, "median"),
        )
    )

    agg_df["count"].sort_index(ascending=False).plot(kind="barh", ax=ax,␣
↪**plot_kwargs)
    add_labels_barh(
        agg_df["count"][::-1],
        agg_df[[f"mean_{target}", f"median_{target}"]].apply(format_prices,␣
↪axis=1)[
            ::-1
        ],
        ax,
    )
    ax.set_ylabel("")
```

[ ]:
```
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Plot for ber classification
unique_ber = train_set["ber_classification"].nunique()
ber_cmap = cm.get_cmap("BrBG", unique_ber)
ber_colors = [ber_cmap(i) for i in range(unique_ber)]
agg_count_plot(
    train_set,
    "ber_classification",
```

```
    "price",
    ax=axes[0],
    color=ber_colors,
    title="ber_classification",
)

unique_types = train_set["property_type"].nunique()
type_cmap = cm.get_cmap("magma", unique_types)
type_colors = [type_cmap(i) for i in range(unique_types)]
agg_count_plot(
    train_set,
    "property_type",
    "price",
    ax=axes[1],
    title="property_type",
    color=type_colors,
)

agg_count_plot(
    train_set, "property_category", "price", ax=axes[2],␣
 ↪title="property_category"
)

fig.tight_layout()
plt.show()
```



Some notes:

`ber_classification` and `property_type` do seem to have some impact on the mean and median house prices at different levels, while the class imbalance of `property_category` types makes it hard to make any conclusions. The `ber_classification` price varies a bit, while `property_type` has bigger rance between mean and median values.

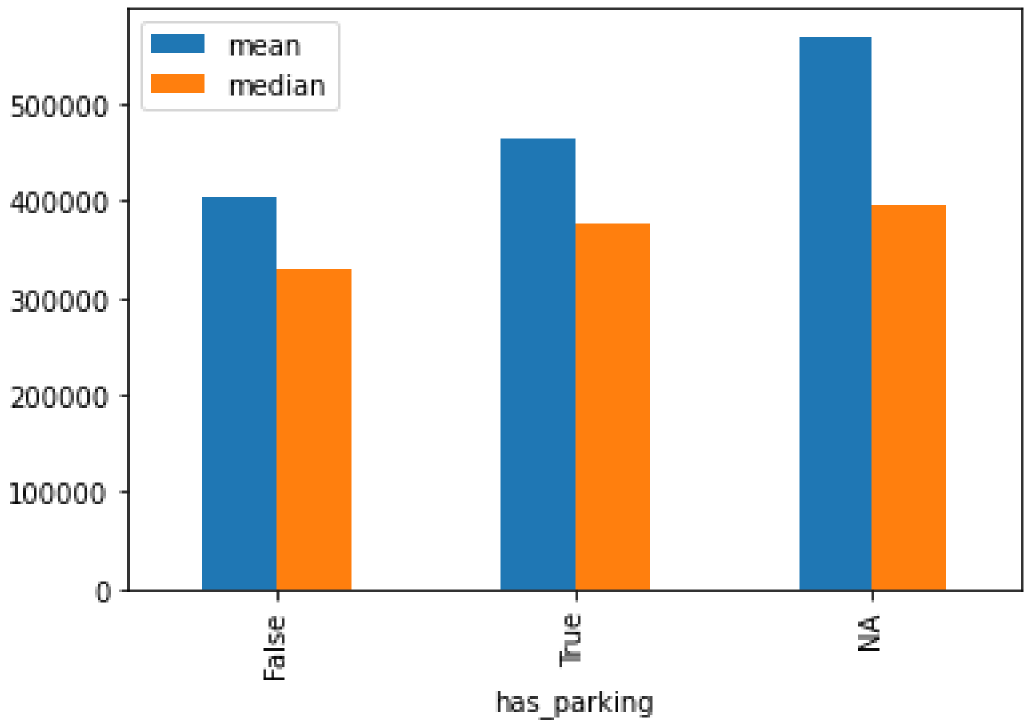
I'll encode `ber_classification` and `property_type` and use these as features.

[ ]:
```
train_set["county"].value_counts()
```

[ ]: Dublin   2892
Name: county, dtype: int64

As there are only entries for Dublin, the feature county is not useful.

[ ]:
```
train_set["has_parking"] = train_set["facility"].str.contains("Parking").
↪fillna("NA")
train_set.groupby("has_parking")["price"].agg(["mean", "median"]).
↪plot(kind="bar")
plt.show()

train_set["has_parking"].value_counts()
```



[ ]: NA   1942
True   839
False   111
Name: has_parking, dtype: int64

Looks like there's also a slight difference in the prices of houses that have parking and those that don't, but there are many missing values for this feature, so I won't use it either.

4.2.1 Latitude / Longitude

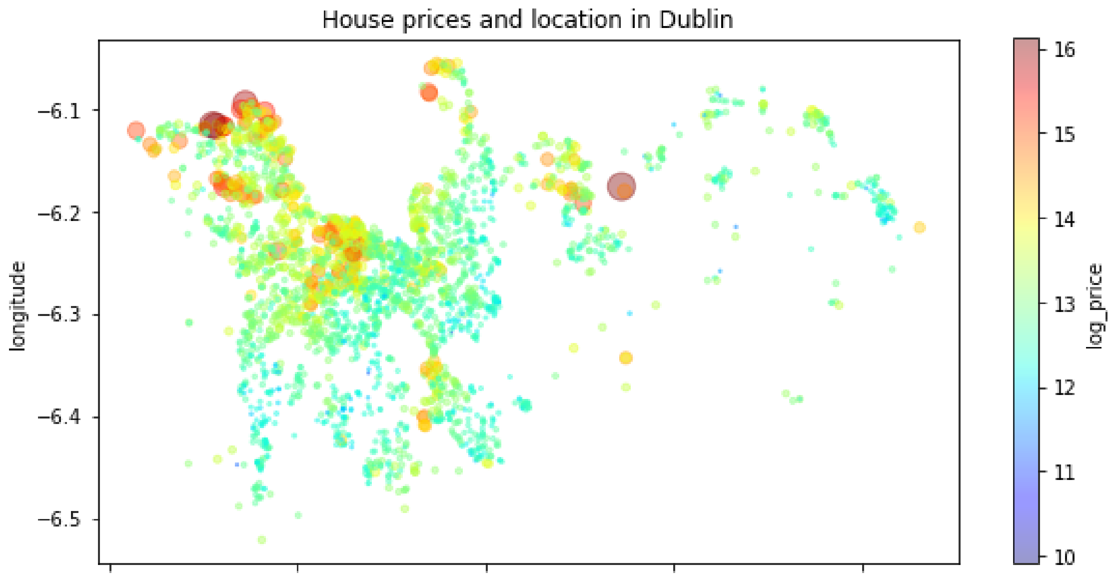
An inspection of these variables showed two entries that appear to have been incorrectly input as they point to places outside of Ireland

[ ]:
```
train_set[train_set["latitude"] < 53]
```

[ ]:
```
           area  bathrooms  beds ber_classification  county  \
767  Clondalkin        1.0   3.0                NaN  Dublin
861  Glenageary        2.0   4.0                  F  Dublin

                                     description_block environment facility  \
767  RAY COOKE AUCTIONEERS take great pleasure in i...        prod      NaN
861  LEONARD WILSON KEENAN ESTATE &amp; LETTING AGE...        prod      NaN

    features   latitude  longitude  no_of_units     price property_category  \
767     None  52.501856  -1.744995          NaN  199000.0              sale
861     None  51.458439  -2.496219          NaN  795000.0              sale

     property_type  surface has_parking
767  semi-detached     79.0          NA
861  semi-detached      NaN          NA
```

[ ]:
```python
# Remove wrong entries
train_set = train_set[train_set["latitude"] > 53]
```

[ ]:
```python
train_set.loc[:, "log_price"] = np.log(train_set.loc[:, "price"],)
train_set.plot.scatter(
    "latitude",
    "longitude",
    s=train_set["price"] / 50_000,
    c="log_price",
    cmap="jet",
    figsize=(10, 5),
    alpha=0.4,
)
plt.xlabel("latitude")
plt.title("House prices and location in Dublin")
plt.show()
```



Instead of encoding `area` for feature, I'll use Latitude and longitude instead, as this can be a way to differentiate on house prices.

4.3 Feature engineering

Encoding categorical variables and scaling.

[ ]:
```python
drop_cols = [
    "area",
    "county",
    "description_block",
    "environment",
    "facility",
    "features",
    "has_parking",
    "no_of_units",
    "property_category",
    "price",
]
train_set = train_set.drop(columns=drop_cols)
train_set.head()
```

[ ]:
```
    bathrooms  beds ber_classification   latitude  longitude property_type  \
15        3.0   5.0                 A3  53.400454  -6.445730      detached
26        4.0   4.0                 A3  53.316410  -6.385214  semi-detached
27        3.0   5.0                 A3  53.401414  -6.446634      detached
35        5.0   5.0                 A2  53.375377  -6.056749      detached
38        2.0   2.0                 A3  53.372130  -6.338466     apartment

    surface  log_price
```

```
15     321.0  13.748302
26     144.0  13.091904
27     321.0  13.748302
35     312.0  14.204169
38      83.0  12.923912
```

Train / test split

```
[ ]: num_cols = ["bathrooms", "beds", "latitude", "longitude", "surface"]
one_hot_cols = ["property_type"]
ordinal_cols = ["ber_classification"]
column_order = num_cols + ordinal_cols + one_hot_cols

train_set["surface"] = train_set["surface"].fillna(
    train_set["surface"].median()
)  # As suggested by Nigel
train_set = train_set[column_order + ["log_price"]].dropna(axis=0)

X_train, y_train = (
    train_set.drop(columns=["log_price"]),
    train_set["log_price"].copy().values,
)

X_test = X_test[column_order].drop(columns=drop_cols, errors="ignore")
y_test = np.log(y_test.copy())

print("Split training data:")
print("\tX_train shape:", X_train.shape)
print("\tX_test shape:", X_test.shape)
```

```
Split training data:
        X_train shape: (2283, 7)
        X_test shape: (500, 7)
```

4.3.1 Feature scaling

```
[ ]: scaler = StandardScaler()
y_scaler = StandardScaler()
ordinal_enc = OrdinalEncoder(handle_unknown="use_encoded_value",␣
 ↪unknown_value=-1)
one_hot_enc = OneHotEncoder(sparse=False)

X_train_scaled = X_train.reset_index(drop=True).copy()
X_test_scaled = X_test.copy()

# Fit and transform training set
X_train_scaled[num_cols] = scaler.fit_transform(X_train_scaled[num_cols])
```

```
X_train_scaled[ordinal_cols] = ordinal_enc.
 ↪fit_transform(X_train_scaled[ordinal_cols])
X_one_hot = one_hot_enc.fit_transform(X_train_scaled[one_hot_cols])
X_one_hot = pd.DataFrame(X_one_hot, columns=one_hot_enc.categories_)
X_train_scaled = pd.concat(
    [X_train_scaled.drop(columns=one_hot_cols), X_one_hot], axis=1,␣
 ↪ignore_index=True
).to_numpy(dtype=np.float32)

# Transform test set
X_test_scaled[num_cols] = scaler.transform(X_test_scaled[num_cols])
X_test_scaled[ordinal_cols] = ordinal_enc.transform(X_test[ordinal_cols])
X_one_hot = one_hot_enc.transform(X_test_scaled[one_hot_cols])
X_one_hot = pd.DataFrame(X_one_hot, columns=one_hot_enc.categories_)
X_test_scaled = pd.concat(
    [X_test_scaled.drop(columns=one_hot_cols), X_one_hot], axis=1,␣
 ↪ignore_index=True
).to_numpy(dtype=np.float32)

# # Fit and transform train and test targets
y_train_scaled = y_scaler.fit_transform(y_train.reshape(-1, 1))
y_test_scaled = y_scaler.transform(y_test.values)

assert len(X_train_scaled) == len(y_train_scaled)
assert len(X_test_scaled) == 500
assert len(y_test_scaled) == len(X_test_scaled)
```

5 PIECEWISE REGRESSION

5.1 Full Model

[ ]:
```
# select some features columns just for the baseline model
# assume not all of the features are informative or useful
# in this exercise you could try all of them if possible

print("Using all features")
print()

X_train_subset = X_train_scaled
print("X_train_subset shape", X_train_subset.shape)
print("y_train_scaled shape", y_train_scaled.shape)

X_test_subset = X_test_scaled
print("X_test_subset shape", X_test_subset.shape)
print("y_test_scaled shape", y_test_scaled.shape)
```

```
Using all features
```

```
X_train_subset shape (2283, 16)
y_train_scaled shape (2283, 1)
X_test_subset shape (500, 16)
y_test_scaled shape (500, 1)
```

5.1.1 Note on fitting times

I found that the shape of the observations plays a huge factor in how long it takes to fit.

Initially I passed in `y_train_scaled` of shape `(1401, 1)` and it took over 5 minutes to fit. After reshaping to `(1401,)` fitting time was drastically reduced to a few seconds!

[ ]:
```
def define_lin_reg(
    predictors,
    observed,
    model_name,
    n_iterations=30_000,
    n_samples=5_000,
    alpha=("Normal", 0, 10),
    beta=("Normal", 0, 10),
    sigma=("HalfCauchy", 5),
    plot_loss=False,
):
    """Defines and trains a Bayesian linear regression model:
            mu ~ alpha + beta * predictors
    With likelihood
            likelihood ~ N(mu, sigma)
    Where alpha, beta and sigma are pymc distributions defined by the user.

    Parameters
    ----------
    predictors : np.ndarray
        Numpy array with model features.
    observed : np.ndarray
        Numpy array with observed values of the target feature. Preferably as a␣
↪1-D array
        to speed up fitting time.
    model_name : str
        Identifier for the model being defined
    n_iterations : int
        The number of iterations for fitting.
    n_samples : int
        The number of samples to draw for the posterior.
    alpha : tuple(string, int, [int, ])
        Prior distribution of alpha. The first argument should be a string with
        a pymc3 model followed by 1 or more integer arguments for the␣
↪parameters of that distribution.
    beta : tuple(string, int, [int, ])
```

```
        Prior distribution of beta. The first argument should be a string with
        a pymc3 model followed by 1 or more integer arguments for the␣
↪parameters of that distribution.
    sigma : tuple(string, int, [int, ])
        Prior distribution of sigma. The first argument should be a string with
        a pymc3 model followed by 1 or more integer arguments for the␣
↪parameters of that distribution.

    Returns
    -------
    posterior : pymc3.backends.base.MultiTrace
        Posterior distribution estimated by pymc model.
    """

    with pm.Model() as model:
        alpha = getattr(pm, alpha[0])("alpha", *alpha[1:])
        beta = getattr(pm, beta[0])("beta", *beta[1:], shape=predictors.
↪shape[1])

        mu = alpha + pm.math.dot(beta, predictors.T)

        sigma = getattr(pm, sigma[0])("sigma", *sigma[1:])
        likelihood = pm.Normal("likelihood", mu=mu, sigma=sigma,␣
↪observed=observed)
        approximation = pm.fit(n_iterations, method="advi",␣
↪random_seed=rand_state)
        posterior = approximation.sample(n_samples)
    if plot_loss:
        plt.figure(figsize=(10, 6))
        plt.plot(approximation.hist, alpha=0.7)
        plt.title("Full model loss")
        plt.xlabel("n_iterations")
        plt.ylabel("log(loss)")
        plt.yscale("log")
        plt.grid(True, which="both", axis="y", linestyle="--")
        plt.show()
    return posterior


def mean_absolute_error(y_true, y_pred):
    return np.mean(abs(y_true - y_pred))


def mape(y_true, y_pred):
    return np.mean(abs(y_true - y_pred) / y_true)
```

```python
def predict(posterior, X, y_scaler):
    """Calculates the predictions for a given X based on a learned posterior

    Parameters
    ----------
    posterior : pymc3.backends.base.MultiTrace.
        Posterior distribution estimated by pymc model.
    X : np.ndarray
        Input features of data to estimate.
    y_scaler : sklearn.preprocessing._data.StandardScaler
        Scaler used to transform the predictor variable.
    Returns
    -------
    np.ndarray
        The model predictions.
    """
    log_likelihood = np.mean(posterior["alpha"]) + np.dot(
        np.mean(posterior["beta"], axis=0), X.T
    )
    y_pred = np.exp(y_scaler.inverse_transform(log_likelihood.reshape(-1, 1)))
    return y_pred


def evaluate(posterior, X, y, y_scaler, model_name, dataset_name):
    """Generates predictions for a dataset and evaluates MAE and MAPE.

    Parameters
    ----------
    posterior : pymc3.backends.base.MultiTrace.
        Posterior distribution estimated by pymc model.
    X : np.ndarray
        Input features of data to estimate.
    y : np.ndarray
        Observed values of target feature.
    y_scaler : sklearn.preprocessing._data.StandardScaler
        Scaler used to transform the predictor variable.
    model_name : str
        Identifier for the model being used.
    model_name : str
        Identifier for the dataset being used.
    Returns
    -------
    None
    """
    y_pred = predict(posterior, X, y_scaler)
    mae, mape_ = mean_absolute_error(y, y_pred), mape(y, y_pred)
```

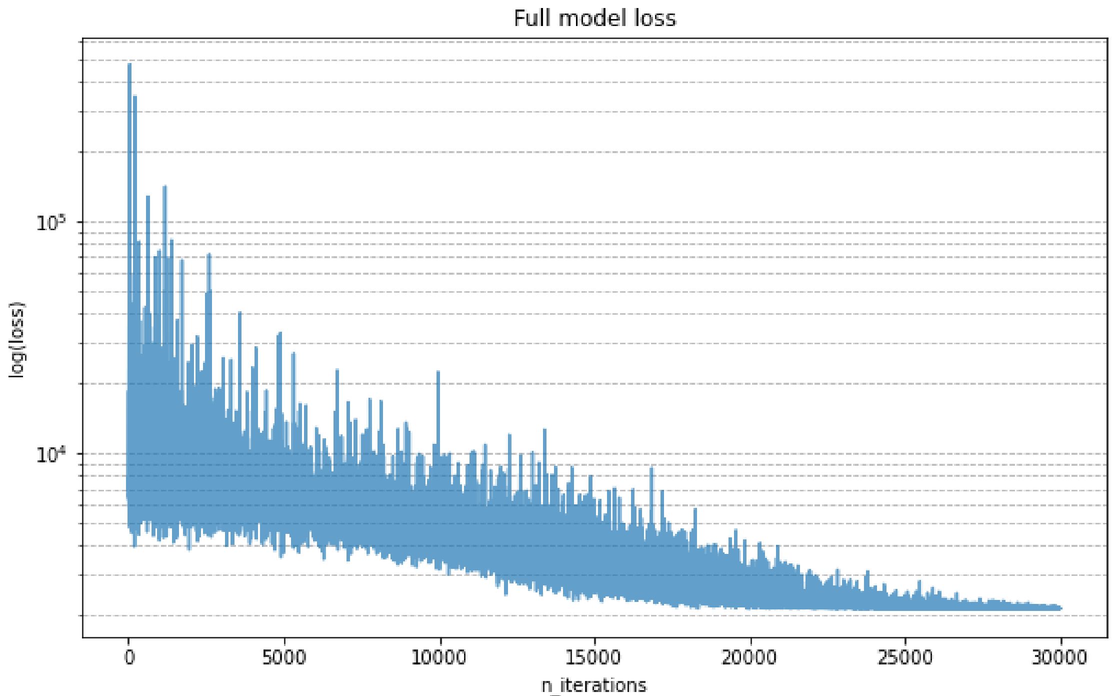
```
    print("\tMAE = ", mae)
    print("\tMAPE = ", mape_)
    return y_pred
```

[ ]:
```
full_posterior = define_lin_reg(
    X_train_subset,
    y_train_scaled.ravel(),
    "full_model",
    alpha=("Normal", 0, 10),
    beta=("Normal", 0, 10),
    sigma=("HalfCauchy", 5),
    plot_loss=True,
)

print("Full model results on the training set:")
y_pred_train = evaluate(
    full_posterior, X_train_subset, y_train_scaled, y_scaler, "full_model",␣
 ↪"train"
)

print("Full model results on the test set:")
y_pred_val = evaluate(
    full_posterior, X_test_subset, y_test_scaled, y_scaler, "full_model", "test"
)
```

```
C:\Users\gamin\anaconda3\envs\et5003\lib\site-
packages\theano\gpuarray\dnn.py:193: UserWarning: Your cuDNN version is more
recent than Theano. If you encounter problems, try updating Theano or
downgrading cuDNN to a version >= v5 and <= v7.
  "Your cuDNN version is more recent than "
```

```
<IPython.core.display.HTML object>
```

```
Finished [100%]: Average Loss = 2,145.7
```



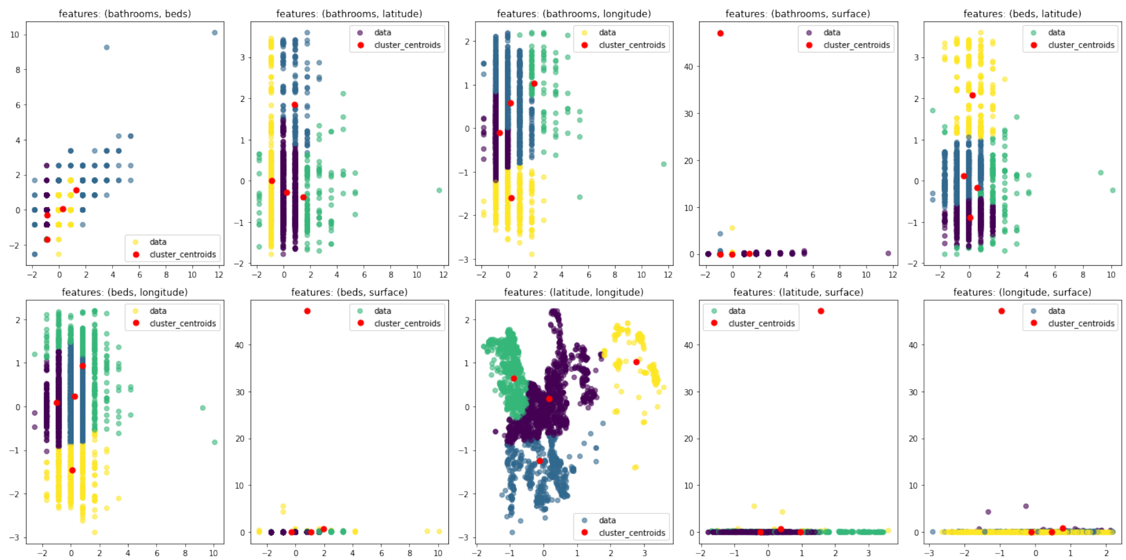

```
Full model results on the training set:
        MAE =  491615.8181058971
        MAPE =  726651.99367199
Full model results on the test set:
        MAE =  484063.295847468
        MAPE =  854733.4213530538
```

5.2 Clustering

Let's try all possible combinations of feature clustering to try to identify the 'pieces' of the data.

[ ]:
```
# training gaussian mixture model
from sklearn.mixture import GaussianMixture
from itertools import combinations

feat_names = train_set.columns.tolist()
# Ideally, create a cluster for each feature
n_clusters = 4
gmm = GaussianMixture(n_components=n_clusters)

feature_combinations = combinations([0, 1, 2, 3, 4], 2)
fig, axes = plt.subplots(2, 5, figsize=(20, 10))

for ax, feat_idx in zip(axes.flatten(), feature_combinations):
    X_clustering = np.r_[X_train_subset[:, feat_idx]]
```

```
    X_cluster_labels = gmm.fit_predict(X_clustering)

    ax.scatter(
        X_clustering[:, 0],
        X_clustering[:, 1],
        alpha=0.6,
        label="data",
        c=X_cluster_labels,
    )
    ax.scatter(
        gmm.means_[:, 0],
        gmm.means_[:, 1],
        s=50,
        c="r",
        marker="o",
        label="cluster_centroids",
    )
    name_1, name_2 = feat_idx
    ax.set_title(f"features: ({feat_names[name_1]}, {feat_names[name_2]})")
    ax.legend()
fig.tight_layout()
plt.show()
```



As suggested in the lecture, we'll use latitude and longitude (features 2, 3) to cluster the data points.

[ ]:
```
gmm = GaussianMixture(n_components=n_clusters, random_state=rand_state)
feat_idx = [2, 3]

train_cluster_labels = gmm.fit_predict(X_train_subset[:, feat_idx])
```

```
test_cluster_labels = gmm.predict(X_test_subset[:, feat_idx])
```

[ ]:
```
# Training clusters
X_train_clusters = [
    X_train_subset[train_cluster_labels == idx] for idx in range(n_clusters)
]

y_scalers = [StandardScaler() for _ in range(n_clusters)]
y_train_clusters = [
    y_scaler.fit_transform(y_train[train_cluster_labels == idx].reshape(-1, 1))
    for idx, y_scaler in enumerate(y_scalers)
]

print("Training cluster shapes:")
print(
    "\n".join(
        f"\tX_{idx}={X.shape}, y_{idx}={y.shape}"
        for idx, (X, y) in enumerate(zip(X_train_clusters, y_train_clusters))
    ),
)
```

```
Training cluster shapes:
        X_0=(151, 16), y_0=(151, 1)
        X_1=(482, 16), y_1=(482, 1)
        X_2=(534, 16), y_2=(534, 1)
        X_3=(1116, 16), y_3=(1116, 1)
```

[ ]:
```
# Test clusters
X_test_clusters = [
    X_test_subset[test_cluster_labels == idx] for idx in range(n_clusters)
]
y_test_clusters = [
    y_test_scaled[test_cluster_labels == idx].ravel() for idx in␣
 ↪range(n_clusters)
]

print("Test cluster shapes:")
print(
    "\n".join(
        f"\tX_{idx}={X.shape}, y_{idx}={y.shape}"
        for idx, (X, y) in enumerate(zip(X_test_clusters, y_test_clusters))
    ),
)
```

```
Test cluster shapes:
        X_0=(42, 16), y_0=(42,)
        X_1=(84, 16), y_1=(84,)
```

```
X_2=(115, 16), y_2=(115,)
X_3=(259, 16), y_3=(259,)
```

5.3 Piecewise Models

[ ]:
```
posterior_0 = define_lin_reg(
    X_train_clusters[0], y_train_clusters[0].ravel(), "piece_0"
)
posterior_1 = define_lin_reg(
    X_train_clusters[1], y_train_clusters[1].ravel(), "piece_1"
)
posterior_2 = define_lin_reg(
    X_train_clusters[2], y_train_clusters[2].ravel(), "piece_2"
)
posterior_3 = define_lin_reg(
    X_train_clusters[3], y_train_clusters[3].ravel(), "piece_3"
)

posteriors = [
    posterior_0,
    posterior_1,
    posterior_2,
    posterior_3,
]
```

```
<IPython.core.display.HTML object>


Finished [100%]: Average Loss = 206.94

<IPython.core.display.HTML object>


Finished [100%]: Average Loss = 469.98

<IPython.core.display.HTML object>


Finished [100%]: Average Loss = 445.05

<IPython.core.display.HTML object>


Finished [100%]: Average Loss = 1,078.5
```

[ ]:
```
print("Piecewise evaluation on training set:\n")

for idx, (posterior, X, y, y_scaler) in enumerate(
```

```
    zip(posteriors, X_train_clusters, y_train_clusters, y_scalers)
):
    print(f"Cluster {idx}, size: {len(y)}")
    evaluate(posterior, X, y, y_scaler, f"piece_{idx}", "train")
all_posteriors = np.r_[posteriors]
```

```
Piecewise evaluation on training set:

Cluster 0, size: 151
        MAE =  357295.429869079
        MAPE =  1548304.608764297
Cluster 1, size: 482
        MAE =  363103.90473494935
        MAPE =  -371356.48718546785
Cluster 2, size: 534
        MAE =  710433.0234449274
        MAPE =  6248013.94423051
Cluster 3, size: 1116
        MAE =  490407.06197411206
        MAPE =  -382901.77960450464
```

[ ]:

```
print("Piecewise evaluation on test set:\n")

y_preds = []
for idx, (posterior, X, y, y_scaler) in enumerate(
    zip(posteriors, X_test_clusters, y_test_clusters, y_scalers)
):
    print(f"Cluster {idx}, size: {len(y)}")
    y_preds.append(evaluate(posterior, X, y, y_scaler, f"piece_{idx}", "test"))
```

```
Piecewise evaluation on test set:

Cluster 0, size: 42
        MAE =  458874.9960082172
        MAPE =  93139.65913675146
Cluster 1, size: 84
        MAE =  342785.91194444406
        MAPE =  243947.7770812006
Cluster 2, size: 115
        MAE =  702508.7415607062
        MAPE =  828845.890877508
Cluster 3, size: 259
        MAE =  510796.6075683432
        MAPE =  971644.1155840937
```

```
[ ]: abs_errors = np.hstack(
         [abs((y_test - y_pred).ravel()) for y_test, y_pred in zip(y_test_clusters,␣
      ↪y_preds)]
     )
     print("Overall MAE", np.mean(abs_errors))
```

```
Overall MAE 524915.0441742124
```

5.4 Simulations & PPC

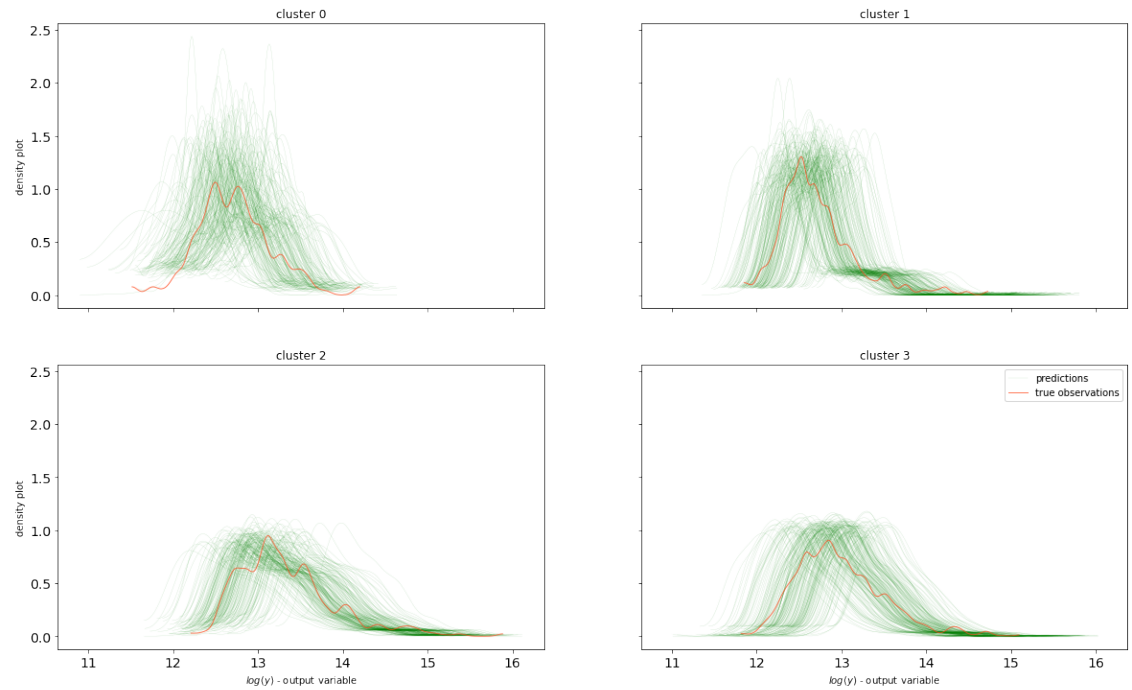
First let's define some helper functions

```
[ ]: # Posterior predictive checks (PPCs)
     def ppc(posterior, X, n_samples=200):
         """Create n_samples predictions for X given a learned posterior.

         Parameters
         ----------
         posterior : pymc3.backends.base.MultiTrace.
             Posterior distribution estimated by pymc model.
         X : np.ndarray
             Features to use for the predictions
         n_samples : int, Optional
             Number of draws to take from the posterior
         Notes
         -----
         The posterior predictive check is done by sampling parameters from the␣
      ↪posterior
         and generating a vector of outcomes bases on these parameters. The result␣
      ↪is a
         matrix of shape SxN where S is the number of samples to draw from the␣
      ↪posterior
         and N is the number of data points in the target y.
         """

         sample_idx = np.random.randint(
             0, len(posterior), size=n_samples
         )  # Indexes to sample from the posterior

         alpha_idx = posterior["alpha"][sample_idx].reshape(-1, 1)  # Shape␣
      ↪(n_samples, 1)
         beta_idx = posterior["beta"][sample_idx, :]  # Shape (n_samples,␣
      ↪n_features)
         sigma_idx = posterior["sigma"][sample_idx].reshape(-1, 1)  # Shape␣
      ↪(n_samples, 1)

         # we generate data from linear model
```

```
    y_pred = (
        alpha_idx
        + np.dot(beta_idx, X.T)
        + np.random.randn(*sigma_idx.shape) * sigma_idx
    )
    assert y_pred.shape == (
        n_samples,
        len(X),
    )  # Final shape should be (n_samples, len(X))
    return y_pred


def plot_ppc(y_true, y_pred, ax, remove_legend, linewidth=0.2, alpha=0.3):
    """Generates a Posterior Predictive check plot, comparing predictions␣
 ↪sampled
    from the learned posterior against the observed values.

    Parameters
    ----------
    y_true : np.ndarray
        The observed values.
    y_pred : np.ndarray
        Values predicted from the posterior
    ax : matplotlib.pyplot.axis
        The axes to plot
    remove_legend : bool
        Removes the axis legend if set to True
    linewidth : float, Optional
        Linewidth value for PPC plot
    alpha : float, Optional
        Alpha value for PPC plot
    """
    plot_kwargs = dict(linewidth=linewidth, alpha=alpha)

    # Plot the predictions
    for row in range(len(y_pred)):
        az.plot_dist(y_pred[row], color="green", ax=ax, plot_kwargs=plot_kwargs)
    az.plot_dist(
        y_pred, color="green", ax=ax, label="predictions",␣
 ↪plot_kwargs=plot_kwargs,
    )

    # Plot the true data
    plot_kwargs.update({"linewidth": 0.9, "alpha": 0.8})
    az.plot_dist(
        y_true,
        color="#ff491c",
```

```
        ax=ax,
        label="true observations",
        plot_kwargs=plot_kwargs,
    )

    if remove_legend:
        ax.get_legend().remove()
```

[ ]:
```
fig, axes = plt.subplots(2, 2, figsize=(20, 12), sharex=True, sharey=True)
piecewise_ppc = []
y_preds = []

for idx, ax in enumerate(axes.flatten()):
    idx_ppc = y_scalers[idx].inverse_transform(
        ppc(posteriors[idx], X_train_clusters[idx],)
    )
    piecewise_ppc.append(idx_ppc)

    # Plotting
    remove_legend = True if idx != n_clusters - 1 else False
    plot_ppc(
        y_scalers[idx].inverse_transform(y_train_clusters[idx]),
        idx_ppc,
        ax,
        remove_legend,
    )
    ax.set_title(f"cluster {idx}")
    if idx % 2 == 0:
        ax.set_ylabel("density plot")
    if idx > 1:
        ax.set_xlabel("$log({y})$ - output variable")
plt.show()
```



6 SUMMARY

Piecewise regression is a technique that allow us to fit portions of the dataset closely when used correctly, however it must be used carefully. If the data is split into too many parts, the training sets for each piece become sparse and the overall fit gets worse.

In this notebook we compared results of a full linear regression model against piecewise regression for the Dublin House prices dataset. Overall I didn't spend too much time with feature engineering and data exploration. Instead I focused on trying various approaches for modelling to better understand the behaviour of Bayesian models. This approach helped me understand the theoretical approach better, however my model performance isn't as strong as that of my peers.

My approach to the problem was:

- Explore the data to identify features, outliers and problematic data points
- Identify numerical columns to scale with StandardScaler
- Identify categorical columns to process with OrdinalScaler (`ber_rating`)
- Identify categorical columns to process with OneHotEncoder (`property_type`)
- Transform train and test sets
- Fit and evaluate full model
- Explore GMM clustering
- Prepare train and test sets with clustering
- Fit and evaluate piecewise models
- Examine PPC plots

The average performance results I'm obtaining are reflected in the PPC plots, the posterior samples are not converging well to the shape of the true observations, they present larger variance and are not exactly centering around the observed mean. To improve this model I would spend more time on the feature engineering and data preparation. One approach that comes to mind is to create an estimator for `surface`, rather than imputing the median values.

[ ]:
```
%%capture
!wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
from colab_pdf import colab_pdf
colab_pdf('Etivity_2_CarlosSiqueiraDoAmaral_20151586.ipynb')
```