Università Commerciale Luigi Bocconi

Master of Science in Business Analytics and Data Science

20596 Machine Learning cl.23

Individual Project

Prediciting Rental Prices

Professor:
Daniele Durante

Author
Ioannis Thomopoulos

Academic Year 2024–2025

1 Introduction

This project aims to predict apartment prices in Milan, Italy using Random Forest (RF) models on cleaned and enriched real-estate data which has been extracted from *Immobiliare.it*. Multiple variants of an RF model were explored: basic RF, tuned RF, interaction-augmented RF. Ultimately, the simplest RF (with a few engineered features) delivered the lowest validation MAE, demonstrating that targeted feature engineering can outperform increasingly complex models.

2 Theoretical Framework

1. **Decision Trees:** The input space is divided using axis-aligned splits to reduce the mean squared error (MSE) within each resulting region. At each split, the algorithm selects the feature f and threshold t that yield the lowest MSE:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

2. **Bagging (Bootstrap Aggregation):** Each of the B trees is trained on a bootstrap sample (63% unique observations). Predictions are averaged across trees, reducing variance:

$$\hat{y} = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$$

3. **Feature Randomness:** At each split, only a random subset of m features ($\sqrt{p}$ by default for regression) is considered. This further de-correlates trees, boosting variance reduction without increasing bias.

4. **Bias-Variance Trade-off:** Bias is low (each tree is deep enough to capture non-linearities). Variance is controlled by ensembling and feature sub-sampling. Generalization improves as the model averages out high-variance idiosyncrasies of single trees.

5. **Key hyperparameters:**
   - *n_estimators* (B): number of trees, more trees $\rightarrow$ lower variance, higher compute.
   - max_depth: tree depth limit directly limits complexity to avoid overfitting.
   - min_samples_split / min_samples_leaf: control minimum samples per node/leaf causing the pre-pruning of small and noisy splits.
   - max_features: controls the number of features considered at each split, using a subset reduces correlation between trees and improves generalization.

6. **Prediction variance formula (approximate):**

$$\text{Var}(\hat{y}) = \rho\,\sigma^2 + \frac{1-\rho}{B}\,\sigma^2,$$

   where ρ is the average pairwise correlation between tree predictions, σ^2 is the variance of an individual tree, and B is the number of trees. As $B \to \infty$, the variance approaches $\rho\,\sigma^2$, reflecting the irreducible component due to correlated predictions.

   This expression is an approximation that relies on several simplifying assumptions:
   - All trees have equal variance (σ^2).
   - All pairs of trees have the same correlation (ρ).
   - The ensemble prediction is the average of individual tree predictions.
   - Higher-order dependencies between trees are ignored.

   Reducing ρ (by injecting randomness into feature selection) is the key to lowering the limiting variance.

3 Data Preparation

1. **Raw Data:** Two CSV files (`train.csv`, `test.csv`) containing property listings with various attributes (e.g., contract type, availability, description, zone, square meters, bathrooms, etc.).

2. **Cleaning Functions:**

   - *Contract Type:* Match keywords to one of seven categories (e.g., 4+4, 3+2, . . . , rent).
   - *Availability:* Parse phrases like "available from . . . " to extract the month name or assign "Other".
   - *Description:* Use regex to extract bedroom and bathroom counts; scan for keywords indicating kitchen type; flag listings that mention "disabled".
   - *Other Features:* Fix malformed separators in strings.
   - *Condition:* Map known condition terms (e.g., excellent, good condition, new) to themselves; assign "undefined" as the default.
   - *Floor:* Assign "undefined" to missing values; otherwise, keep as string.
   - *Energy Class:* Map letters A-G to "Class A"-"Class G" and assign "undefined" if missing.
   - *Elevator:* Map "yes" to 1 and all other values to 0.
   - *Condominium Fees:* Fill missing values with the median (only one instance of NaN in data).

3. **Geocoding and Distances:**

   - Query the Nominatim API for each unique zone to obtain latitude and longitude, falling back to a simplified zone label if needed.
   - Manually assign coordinates for the zone "monte rosa - lotto" as the API could not provide its coordinates.
   - Compute the distance from each zone's centroid to the city center, which for simplicity has been set to the "Duomo" (45.4641° N, 9.1919° E).

4. **Feature-Flag Expansion:** A fixed list of 18 amenities (e.g., electric gate, optic fiber, etc.) is scanned within the cleaned feature string to create corresponding boolean columns.

5. **Column Cleanup and Renaming:** Drop all raw and intermediate columns. Rename cleaned columns using TitleCase (e.g., `SquaredMeters`, `Bedrooms`, `DistanceToDuomo`). The cleaning process is applied independently to training and test sets to avoid in-place mutation.

4 Modeling Variants and Selection

Four main Random Forest models were built and compared by validation MAE on a 20% hold-out set or 5-fold cross validation. The first model started simple and progressively added complexity, then found that a lightly augmented baseline performed best.

4.1 Baseline RF

This "Base" model was made to have a baseline to compare future, more complex, models to. This version has no parameter tuning or interaction terms, simply a basic RF regression with the following set-up:

```
rf = RandomForestRegressor(
    n_estimators=200,
    max_depth=None,
    random_state=42,
    n_jobs=-1   )
```

The "Base" model had a Validation MAE: 357.36, this score would end up being very good and most improvements on this were very minimal

4.2 Hyperparameter Tuning (RandomizedSearchCV)

In an attempt to improve on the "Base" model, hyperparameter tuning was performed using grid-search with the following parameters:

```
param_dist = {
    'n_estimators': [100,200,300,500,1000],
    'max_depth': [10,20,30,None],
    'min_samples_split': [2,5,10],
    'min_samples_leaf': [1,2,4],
    'max_features': ['sqrt','log2', 0.5, None]      }
```

This only led to a very modest MAE improvement, but came with higher complexity and compute time. This complexity and compute time is only relevant for the grid-search as in practice a new model would be run using the best parameters from the grid-search. The best parameters were:

```
Best params: {
    'n_estimators': 500,
    'min_samples_split': 2,
    'min_samples_leaf': 1,
    'max_features': 0.5,
    'max_depth': 30     }
```

4.3 RF with Interaction Terms

To further improve the "Base" model, a better understanding of the impact that each attribute has is required. This can be obtained by getting the feature importance of each attribute:

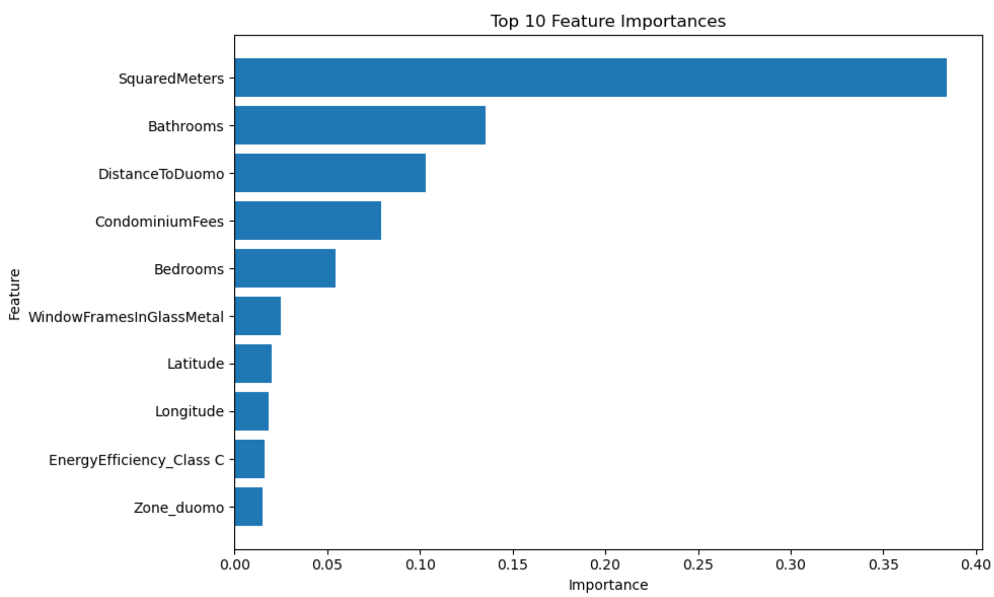

Figure 1: Feature Importance of RF Model

It can clearly be seen that `SquaredMeters, Bathrooms, DistanceToDuomo, CondominiumFees, Bedrooms` are the most important features, with `SquaredMeters` being by far the most important. Using this, interaction terms can be created to try and capture the effect that one feature may have on another (e.g., proximity to Duomo might matter more for larger apartments). The interaction terms created were:

```
data_int['sqm_bathrooms']        = data_int['SquaredMeters'] * data_int['Bathrooms']
data_int['sqm_bedrooms']         = data_int['SquaredMeters'] * data_int['Bedrooms']
data_int['bedrooms_bathrooms']   = data_int['Bedrooms'] * data_int['Bathrooms']
data_int['distance_sqm']         = data_int['DistanceToDuomo'] * data_int['SquaredMeters']
data_int['fees_sqm']             = data_int['CondominiumFees'] * data_int['SquaredMeters']
data_int['bathrooms_distance']   = data_int['Bathrooms'] * data_int['DistanceToDuomo']
data_int['bedrooms_fees']        = data_int['Bedrooms'] * data_int['CondominiumFees']
```

Using these interaction terms an RF model was run, bringing the MAE to 349.29.

4.4 RF with Metro Proximity

Seeing the effect of `distanceToDuomo` in the feature importance, clearly access to the city center is important. Another way to get to the city center is with the Metro, to investigate this a csv with the coordinates of Metro stops in Milan was downloaded and the `distanceToMetro` variable was created. An RF model was run with `distanceToMetro`, producing the best results yet with an MAE = 347.08.

Incorporating the interaction terms from before, another RF model was created using the new `distanceToMetro`. This model produced an MAE = 352.49, showing that interaction terms actually decrease the accuracy of the model when incorporating `distanceToMetro`. This is likely occuring because the `distanceToMetro` is very important in the prediction process and takes away from the effect that one feature has on another.

4.5 Final Model

After all experiments, the best variation of the RF model was that with `distanceToMetro` and no interaction terms. This version includes the `Latitude`, `longitude`, and `distanceToDuomo` which were created at the very start. This RF model produces an MAE score of 347.08 as seen previously. To ensure that this is the best score that can be achieved using the RF and the feature engineering that has been done, one last parameter search was performed resulting in the following final parameters:

```
Best_params: {
    n_estimators=1000,
    max_depth=30,
    min_samples_split=2,
    min_samples_leaf=1,
    max_features=0.5   }
```

Running the final RF Model using this, a score of 327.96 was achieved.

5 Possible Extensions

Going forward, if this prediction was to be improved further, more complex models would need to be implemented. Boosting Models as well as unsupervised machine learning methods likely can score better, at the cost of interpretability. As a proof-of-concept, a Histogram Gradient Boosting Model was run to check the results. Using the same data that the final RF model used, the Boosting Model achieved a score of 304. Additionally, stacking models can also produce an improved score. As a further proof-of-concept, a Histogram Gradient boosting model, Extreme Gradient boosting model, and a CatBoost model were stacked together with a LassoCV meta-learner to achieve a final, slightly improved, score of 302. These models, although score higher, are much more complex and hard to understand which is why they were avoided in the first place.
Final Score on Data Science Challenges Platform:
Random Forest Model: 327.96
Gradient Boosting Model: 304.41
Stacked LassoCV Model: 302.07

6 Project Code

```
'''
External Data is used in this code, Internet connection is required.

Zone Coordinates were taken from: https://nominatim.openstreetmap.org/
Metro Coordinates were taken from: https://dati.comune.milano.it/dataset/b7344a8f-0ef5-
424b-a902-f7f06e32dd67/resource/0f4d4d05-b379-45a4-9a10-
412a34708484/download/tpl_metrofermate.csv

All code should run as is, assuming all libraries are installed.
Given data path should be set to the location of the data files on your machine.
Grid-Search and Parameter Tuning might take a while to run depending on your machine.
'''

import pandas as pd
import numpy as np
import re
import requests
import time
import math
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.metrics import mean_absolute_error
from tqdm import tqdm
import warnings
import matplotlib.pyplot as plt
from joblib import Parallel, delayed
from itertools import product
from sklearn.model_selection import cross_val_score, KFold

warnings.filterwarnings("ignore")

TRAIN_PATH = 'original_data/train.csv'
TEST_PATH = 'original_data/test.csv'
METRO_PATH = 'metro_stops_milan.csv_'

train_data = pd.read_csv(TRAIN_PATH)
test_data  = pd.read_csv(TEST_PATH)

#Manually getting Duomo Coordinates
duomo_lat, duomo_lon = 45.4641, 9.1919

def clean_contract_type(value):
    if isinstance(value, str):
        for k in ['4+4','3+2','6+6','transitory','students','open','rent']:
            if k in value:
                return k
    return 'rent'
```

```
def parse_availability(value):
    if isinstance(value, str) and value.startswith('available from'):
        date = pd.to_datetime(value.replace('available from','').strip(),
                              dayfirst=True, errors='coerce')
        if pd.notnull(date):
            return date.strftime('%B')
    return 'Other'

def parse_description(desc):
    bedrooms = bathrooms = disabled = 0
    kitchen = 'Unknown'
    if isinstance(desc, str):
        m = re.search(r'(\d+)\s+bedrooms?', desc, re.I)
        if m:
            bedrooms = int(m.group(1))
        m = re.search(r'(\d+)\s+bathrooms?', desc, re.I)
        if m:
            bathrooms = int(m.group(1))
        for kt in ['kitchenette','open kitchen','kitchen diner',
                   'kitchen nook','semi-habitable kitchen']:
            if kt in desc.lower():
                kitchen = kt
                break
        if 'disabled' in desc.lower():
            disabled = 1
    return pd.Series([bedrooms, bathrooms, kitchen, disabled],
                     index=['cleaned_bedrooms','cleaned_bathrooms',
                            'cleaned_kitchen_type','cleaned_disabled_friendly'])

def clean_other_features(value):
    if isinstance(value, str):
        return value.replace('pvcdouble exposure','pvc | double exposure')
    return ''

def clean_condition(value):
    if isinstance(value, str):
        v = value.strip().lower()
        if v in ['excellent','good condition','new']:
            return v
    return 'undefined'

def clean_floor(value):
    return 'undefined' if pd.isna(value) else str(value).strip()

efficiency_map = {c: f'Class {c.upper()}' for c in list('abcdefg')}
def clean_efficiency_class(x):
```

```
    if pd.isna(x):
        return 'undefined'
    return efficiency_map.get(str(x).lower().strip(), 'undefined')

def get_coordinates(zone):
    url = 'https://nominatim.openstreetmap.org/search'
    params = {'q': f"{zone}, Milan, Italy",'format':'json','limit':1}
    headers = {'User-Agent':'OpenAI GPT-4'}     #Need to define who is using the API, apparently mimicing and llm is effective for this (I havent really used APIs that much so I am not sure if this is the best way to do it, but it works)
    r = requests.get(url, params=params, headers=headers)
    if r.status_code==200 and r.json():
        d = r.json()[0]
        return float(d['lat']), float(d['lon'])
    return None, None

def haversine(lat1, lon1, lat2, lon2):
    R=6371
    ϕ1,ϕ2 = math.radians(lat1), math.radians(lat2)
    dϕ = math.radians(lat2-lat1)
    dλ = math.radians(lon2-lon1)
    a = math.sin(dϕ/2)**2 + math.cos(ϕ1)*math.cos(ϕ2)*math.sin(dλ/2)**2
    return R * 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))

def haversine_array(lat, lon, lats, lons):
    R = 6371  # km
    ϕ1 = np.radians(lat)
    ϕ2 = np.radians(lats)
    dϕ = np.radians(lats - lat)
    dλ = np.radians(lons - lon)
    a  = np.sin(dϕ/2.0)**2 + np.cos(ϕ1)*np.cos(ϕ2)*np.sin(dλ/2.0)**2
    c  = 2 * np.arctan2(np.sqrt(a), np.sqrt(1-a))
    return R * c

def fallback_zone(z):
    return z.split('-')[0].split('/')[0].strip()

features = [
    'electric gate','optic fiber','security door','internal exposure',
    'external exposure','closet','balcony','full day concierge',
    'half-day concierge','centralized tv system','furnished',
    'shared garden','window frames in double glass / pvc',
    'window frames in triple glass / metal','window frames in glass / metal',
    'attic','tavern','video entryphone'
]

def data_cleaner(data):
```

```
    df = data.copy()

    #Simple Pandas cleaning for the 'basic' columns
    df['cleaned_contract_type']     = df['contract_type'].apply(clean_contract_type)
    df['cleaned_availability']      = df['availability'].apply(parse_availability)
    df[['cleaned_bedrooms','cleaned_bathrooms',
        'cleaned_kitchen_type','cleaned_disabled_friendly']] = \
         df['description'].apply(parse_description)
    df['cleaned_floor']             = df['floor'].apply(clean_floor)
    df['cleaned_conditions']        = df['conditions'].apply(clean_condition)
    df['cleaned_elevator']          = df['elevator'].map(lambda x: 1 if x=='yes' else 0)
    df['cleaned_energy_efficiency'] =
df['energy_efficiency_class'].apply(clean_efficiency_class)
    df['cleaned_condominium_fees']  =
df['condominium_fees'].fillna(df['condominium_fees'].median())
    df['other_features_cleaned']    = df['other_features'].apply(clean_other_features)

    #Creating the Binary flags for 'other_features'
    for feat in features:
        col = 'cleaned_' + feat.replace(' ', '_')\
                          .replace('/', '_')\
                          .replace('-', '_')\
                          .lower()
        df[col] = df['other_features_cleaned'].apply(lambda x: int(feat in x))

    zone_coords = {}
    for z in df['zone'].unique():
        lat, lon = get_coordinates(z)
        if lat is None:
            lat, lon = get_coordinates(fallback_zone(z))
        zone_coords[z] = (lat or 0, lon or 0)
        time.sleep(1)   #Avoids the API rate limit

    #API cannot find this zone, so it's hardcoded
    zone_coords['monte rosa - lotto'] = (45.4785, 9.1343)

    df['cleaned_latitude'] = df['zone'].map(lambda z: zone_coords[z][0])
    df['cleaned_longitude'] = df['zone'].map(lambda z: zone_coords[z][1])
    df['cleaned_distance_to_duomo'] = df['zone'].map(
        lambda z: haversine(zone_coords[z][0],
                      zone_coords[z][1],
                      duomo_lat, duomo_lon)
    )

    df = df.drop(columns=[
        'contract_type','availability','description',
        'other_features','other_features_cleaned',
```

```
    'conditions','floor','elevator',
    'energy_efficiency_class','condominium_fees'
  ])

  renamed = {
    c: c.replace('cleaned_','').replace('_',' ').title().replace(' ','')
    for c in df.columns if c.startswith('cleaned_')
  }
  prepped = df.rename(columns=renamed)\
        .rename(columns={'square_meters':'SquaredMeters','zone':'Zone'})

  #Rename 'Price' in Train data
  if 'w' in prepped.columns:
    prepped = prepped.drop(columns=['w']).rename(columns={'y':'Price'})
  return prepped

train_data_cleaned = data_cleaner(train_data)
test_data_cleaned  = data_cleaner(test_data)

'''
BASE RANDOM FOREST MODEL
'''

df = train_data_cleaned.copy()
X = pd.get_dummies(df.drop(columns='Price'))
y = df['Price']
X_train, X_val, y_train, y_val = train_test_split(X, y,
                                                  test_size=0.2,
                                                  random_state=42)

rf = RandomForestRegressor(n_estimators=200,
                           max_depth=None,
                           random_state=42,
                           n_jobs=-1)


rf.fit(X_train, y_train)
val_preds = rf.predict(X_val)
print(f"Validation MAE: {mean_absolute_error(y_val, val_preds):.2f}")

'''
BASE RANDOM FOREST MODEL WITH GRID-SEARCH
'''

df = train_data_cleaned.copy().drop(columns=['Zone'])
cat_cols = df.select_dtypes(include='object').columns
data_enc = pd.get_dummies(df, columns=cat_cols)
```

```
X_full = data_enc.drop(columns='Price')
y_full = data_enc['Price']
X_tr, X_va, y_tr, y_va = train_test_split(X_full, y_full,
                                          test_size=0.2,
                                          random_state=42)

param_dist = {
    'n_estimators': [100,200,300,500,1000],
    'max_depth': [10,20,30, None],
    'min_samples_split': [2,5,10],
    'min_samples_leaf': [1,2,4],
    'max_features': ['sqrt','log2', 0.5, None]
}

class TqdmRandomizedSearchCV(RandomizedSearchCV):
    def fit(self, X, y=None, **fit_params):
        total = self.n_iter * self.cv
        with tqdm(total=total, desc="RandomizedSearchCV Progress") as pbar:
            old = self._run_search
            def new_run(evaluate):
                def wrapper(params):
                    out = evaluate(params)
                    pbar.update(self.cv)
                    return out
                return old(wrapper)
            self._run_search = new_run
            return super().fit(X, y, **fit_params)

search = TqdmRandomizedSearchCV(
    estimator=RandomForestRegressor(random_state=42, n_jobs=6),
    param_distributions=param_dist,
    n_iter=100,
    scoring='neg_mean_absolute_error',
    cv=5,
    random_state=42,
    n_jobs=6
)
search.fit(X_tr, y_tr)

best_rf = search.best_estimator_
val_preds = best_rf.predict(X_va)
print(f"Tuned Validation MAE: {mean_absolute_error(y_va, val_preds):.2f}")
print("Best params:", search.best_params_)

'''
CREATING TXT FILE FOR SUBMISSION (CAN IGNORE SECTION)
```

```
'''

train_df = train_data_cleaned.copy()
test_df  = test_data_cleaned.copy()

target = 'Price'
exclude_cols = []
features = [c for c in train_df.columns if c != target and c not in exclude_cols]

X_trn = train_df[features]
y_trn = train_df[target]
X_tst = test_df[features]

combined = pd.concat([X_trn, X_tst], keys=['train','test'])
combined = combined.drop(columns=exclude_cols, errors='ignore')
cat_cols = combined.select_dtypes(include='object').columns
combined_enc = pd.get_dummies(combined, columns=cat_cols)

X_trn_enc = combined_enc.xs('train')
X_tst_enc = combined_enc.xs('test').reindex(
    columns=X_trn_enc.columns, fill_value=0
)

final_rf = RandomForestRegressor(
    n_estimators=1000,
    min_samples_split=6,
    min_samples_leaf=1,
    max_features=0.5,
    max_depth=30,
    random_state=42,
    n_jobs=6
)
final_rf.fit(X_trn_enc, y_trn)

test_preds = final_rf.predict(X_tst_enc)
with open("test_predictions_1.txt", "w") as f:
    for v in test_preds:
        f.write(f"{v:.2f}\n")

'''
FEATURE IMPORTANCE
'''

importances = final_rf.feature_importances_
features    = X_trn_enc.columns

importance_df = pd.DataFrame({
```

```
    'Feature': features,
    'Importance': importances
}).sort_values('Importance', ascending=False)

top10 = importance_df.head(10)
fig, ax = plt.subplots(figsize=(10, 6))

ax.barh(top10['Feature'][::-1], top10['Importance'][::-1])
ax.set_xlabel('Importance')
ax.set_ylabel('Feature')
ax.set_title('Top 10 Feature Importances')
plt.tight_layout()
plt.show()

'''
RANDOM FOREST USING INTERACTION TERMS
'''

data_int = train_data_cleaned.copy()

data_int['sqm_bathrooms']       = data_int['SquaredMeters'] * data_int['Bathrooms']
data_int['sqm_bedrooms']        = data_int['SquaredMeters'] * data_int['Bedrooms']
data_int['bedrooms_bathrooms']   = data_int['Bedrooms'] * data_int['Bathrooms']
data_int['distance_sqm']        = data_int['DistanceToDuomo'] * data_int['SquaredMeters']
data_int['fees_sqm']            = data_int['CondominiumFees'] * data_int['SquaredMeters']
data_int['bathrooms_distance']   = data_int['Bathrooms'] * data_int['DistanceToDuomo']
data_int['bedrooms_fees']       = data_int['Bedrooms'] * data_int['CondominiumFees']

test_int = test_data_cleaned.copy()

test_int['sqm_bathrooms']      = test_int['SquaredMeters'] * test_int['Bathrooms']
test_int['sqm_bedrooms']       = test_int['SquaredMeters'] * test_int['Bedrooms']
test_int['bedrooms_bathrooms'] = test_int['Bedrooms']      * test_int['Bathrooms']
test_int['distance_sqm']       = test_int['DistanceToDuomo'] * test_int['SquaredMeters']
test_int['fees_sqm']           = test_int['CondominiumFees'] * test_int['SquaredMeters']
test_int['bathrooms_distance'] = test_int['Bathrooms'] * test_int['DistanceToDuomo']
test_int['bedrooms_fees']      = test_int['Bedrooms'] * test_int['CondominiumFees']

y = data_int['Price']
X = data_int.drop(columns='Price')

X = pd.get_dummies(X, drop_first=True)  #One-hot encoding
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

rf_int = RandomForestRegressor(
    n_estimators=500,
    max_depth=30,
```

```
    min_samples_split=2,
    min_samples_leaf=1,
    max_features=0.5,
    random_state=42,
    n_jobs=-1
)
rf_int.fit(X_train, y_train)
preds = rf_int.predict(X_val)
print(f"Validation MAE with interactions: {mean_absolute_error(y_val, preds):.2f}")

'''
RANDOM FOREST WITH METRO STOPS
'''

metro_df = pd.read_csv(METRO_PATH, sep=';', quotechar='"')

train_df = train_data_cleaned.copy()
test_df  = test_data_cleaned.copy()

metro_df = metro_df.rename(columns={
    'nome':      'Station',
    'LAT_Y_4326':'MetroLat',
    'LONG_X_4326':'MetroLon'
})[['Station','MetroLat','MetroLon']]

station_lats = metro_df['MetroLat'].values
station_lons = metro_df['MetroLon'].values
station_names = metro_df['Station'].values

def find_nearest_station(row):
    dists = haversine_array(
        row['Latitude'],
        row['Longitude'],
        station_lats,
        station_lons
    )
    idx = np.argmin(dists)
    return pd.Series({
        'NearestMetro':     station_names[idx],
        'DistanceToMetro':  dists[idx]
    })

train_data_metro = train_df.copy()
train_data_metro[['NearestMetro','DistanceToMetro']] = train_data_metro.apply(
    find_nearest_station, axis=1
)
```

```
test_data_metro = test_df.copy()
test_data_metro[['NearestMetro','DistanceToMetro']] = test_data_metro.apply(
    find_nearest_station, axis=1
)

data = train_data_metro.copy()

target = 'Price'
features = [col for col in data.columns if col != target]

X = data[features]
y = data[target]

combined = pd.concat([X, data[features]], keys=['train', 'test'])

categorical_cols = combined.select_dtypes(include='object').columns
combined_encoded = pd.get_dummies(combined, columns=categorical_cols)

X_encoded = combined_encoded.xs('train')
X_test_encoded = combined_encoded.xs('test')

X_train, X_val, y_train, y_val = train_test_split(X_encoded, y, test_size=0.2,
random_state=42)
rf = RandomForestRegressor(n_estimators=500, max_depth=30, min_samples_split=2,
min_samples_leaf=1, max_features=0.5, random_state=42, n_jobs=-1)
rf.fit(X_train, y_train)

val_preds = rf.predict(X_val)
mae = mean_absolute_error(y_val, val_preds)
print(f"Validation MAE: {mae:.2f}")

'''
RANDOM FOREST WITH METRO STOP + INTERACTION TERMS
'''

metro_int_train = pd.concat([data_int, train_data_metro], axis=1)
metro_int_test = pd.concat([test_int, test_data_metro], axis=1)

data = metro_int_train.copy()

target = 'Price'
features = [col for col in data.columns if col != target]

X = data[features]
y = data[target]

combined = pd.concat([X, data[features]], keys=['train', 'test'])
```

```
categorical_cols = combined.select_dtypes(include='object').columns
combined_encoded = pd.get_dummies(combined, columns=categorical_cols)

X_encoded = combined_encoded.xs('train')
X_test_encoded = combined_encoded.xs('test')

X_train, X_val, y_train, y_val = train_test_split(X_encoded, y, test_size=0.2,
random_state=42)

rf = RandomForestRegressor(n_estimators=500, max_depth=30, min_samples_split=2,
min_samples_leaf=1, max_features=0.5, random_state=42, n_jobs=-1)
rf.fit(X_train, y_train)

val_preds = rf.predict(X_val)
mae = mean_absolute_error(y_val, val_preds)
print(f"Validation MAE: {mae:.2f}")

'''
FINAL RANDOM FOREST MODEL
'''

data = train_data_metro.copy()

target = 'Price'
features = [col for col in data.columns if col != target]

X = data[features]
y = data[target]

combined = pd.concat([X, data[features]], keys=['train', 'test'])

categorical_cols = combined.select_dtypes(include='object').columns
combined_encoded = pd.get_dummies(combined, columns=categorical_cols)

X_encoded = combined_encoded.xs('train')
X_test_encoded = combined_encoded.xs('test')

X_train, X_val, y_train, y_val = train_test_split(X_encoded, y, test_size=0.2,
random_state=42)

rf = RandomForestRegressor(n_estimators=1000, max_depth=30, min_samples_split=2,
min_samples_leaf=1, max_features=0.3, random_state=42, n_jobs=-1)
rf.fit(X_train, y_train)

val_preds = rf.predict(X_val)
mae = mean_absolute_error(y_val, val_preds)
```

```
print(f"Validation MAE: {mae:.2f}")

model_data = train_data_metro.copy()

X_train = model_data.drop(columns='Price')
y_train = model_data['Price']

X_test = test_data_metro.copy()

combined = pd.concat([X_train, X_test], keys=['train', 'test'])

categorical_cols = combined.select_dtypes(include='object').columns
combined_encoded = pd.get_dummies(combined, columns=categorical_cols)

X_train_encoded = combined_encoded.xs('train')
X_test_encoded  = combined_encoded.xs('test')

X_test_encoded = X_test_encoded.reindex(
   columns=X_train_encoded.columns,
   fill_value=0
)
final_rf.fit(X_train_encoded, y_train)
test_preds = final_rf.predict(X_test_encoded)

with open("Final_Rent_Predictions.txt", "w") as f:
   for val in test_preds:
      f.write(f"{val:.2f}\n")

'''
HISTOGRAM GRADIENT BOOSTING MODEL
'''

import pandas as pd
import numpy as np
from joblib import Parallel, delayed
from tqdm import tqdm
from itertools import product

from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score, KFold

X = train_data_metro.drop(columns='Price')
y = train_data_metro['Price']
```

```
cat_cols = X.select_dtypes(include='object').columns.tolist()

preprocessor = ColumnTransformer(
    [('ohe', OneHotEncoder(handle_unknown='ignore', sparse_output=False), cat_cols)],
    remainder='passthrough'
)

#This was adjusted to find the best parameters for the HistGradientBoostingRegressor, only
the final combination is used in this model
param_grid = {
    'learning_rate':      [0.05],
    'max_iter':          [1500],
    'max_leaf_nodes':    [50],
    'max_depth':         [None],
    'min_samples_leaf':  [5]
}

param_names = list(param_grid.keys())
param_list = [dict(zip(param_names, combo)) for combo in product(*param_grid.values())]

cv = KFold(n_splits=3, shuffle=True, random_state=42)

def evaluate(params):
    model = HistGradientBoostingRegressor(random_state=42, **params,
early_stopping=True)
    pipe  = make_pipeline(preprocessor, model)
    # neg MAE across 3 folds
    neg_maes = cross_val_score(
        pipe, X, y,
        scoring='neg_mean_absolute_error',
        cv=cv,
        n_jobs=1
    )
    return -np.mean(neg_maes), params

results = Parallel(n_jobs=14)(
    delayed(evaluate)(p)
    for p in tqdm(param_list, desc="HGB Grid Search (3-fold CV)", unit="model")
)

best_mae, best_params = min(results, key=lambda x: x[0])
print(f"\nBest 3-fold CV MAE: {best_mae:.2f}")
print("Best params:", best_params)

final_model = make_pipeline(
    preprocessor,
    HistGradientBoostingRegressor(random_state=42, **best_params)
```

```
)
final_model.fit(X, y)

X_test = test_data_metro.drop(columns=['Price'], errors='ignore')
test_preds = final_model.predict(X_test)

with open("Final_Boosting_model_predictions.txt", "w") as f:
    for val in test_preds:
        f.write(f"{val:.2f}\n")

'''
STACKING HGBOOST, XGBOOST, & CATBOOST MODELS
'''

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import HistGradientBoostingRegressor, StackingRegressor
from xgboost import XGBRegressor
from catboost import CatBoostRegressor
from sklearn.linear_model import LassoCV
from sklearn.model_selection import RandomizedSearchCV, KFold
from sklearn.metrics import make_scorer, mean_absolute_error
from tqdm.auto import tqdm

X = train_data_metro.drop(columns='Price')
y = train_data_metro['Price'].values
X_test = test_data_metro.drop(columns='Price', errors='ignore')

combined = pd.concat([X, X_test], keys=['train', 'test'])
combined_encoded = pd.get_dummies(
    combined,
    columns=combined.select_dtypes(include='object').columns
)
X_enc_df      = combined_encoded.xs('train')
X_test_enc_df = combined_encoded.xs('test').reindex(columns=X_enc_df.columns,
fill_value=0)

X_enc      = X_enc_df.values
X_test_enc = X_test_enc_df.values
y_enc      = y

#set up HGB using parameters found previously
hgb = HistGradientBoostingRegressor(
    random_state=42,
    learning_rate=0.05,
    max_iter=1500,
    max_leaf_nodes=50,
```

```
    max_depth=None,
    min_samples_leaf=5
)

#set up random search
class TqdmRandomizedSearchCV(RandomizedSearchCV):
    def fit(self, X, y=None, **fit_params):
        total_iters = self.n_iter * (self.cv or 1)
        with tqdm(total=total_iters, desc=self.__class__.__name__, unit="fit") as pb:
            orig_run_search = self._run_search
            def wrapped_run_search(evaluate_candidates):
                def wrapped_evaluate(candidates):
                    results = evaluate_candidates(candidates)
                    pb.update(len(candidates) * (self.cv or 1))
                    return results
                return orig_run_search(wrapped_evaluate)
            self._run_search = wrapped_run_search
            return super().fit(X, y, **fit_params)


#tuning XGBoost model
xgb = XGBRegressor(objective='reg:squarederror', random_state=42, n_jobs=1)
param_dist_xgb = {
    'n_estimators':     [200, 400, 600, 800, 1000],
    'learning_rate':    [0.01, 0.03, 0.05, 0.1],
    'max_depth':        [4, 6, 8, 10, 12],
    'subsample':        [0.6, 0.7, 0.8, 0.9, 1.0],
    'colsample_bytree': [0.6, 0.7, 0.8, 0.9, 1.0],
    'min_child_weight': [1, 3, 5, 7, 10]
}
search_xgb = TqdmRandomizedSearchCV(
    estimator=xgb,
    param_distributions=param_dist_xgb,
    n_iter=30,
    scoring='neg_mean_absolute_error',
    cv=5,
    n_jobs=-1,
    random_state=42,
    verbose=0
)
search_xgb.fit(X_enc, y_enc)
best_xgb = search_xgb.best_estimator_

#tuning CatBoost model
cat = CatBoostRegressor(random_seed=42, verbose=False)
param_dist_cat = {
    'iterations':      [500, 1000, 1500],
```

```
    'learning_rate':   [0.01, 0.03, 0.05, 0.1],
    'depth':           [4, 6, 8, 10],
    'l2_leaf_reg':     [1, 3, 5, 7, 10]
}
search_cat = TqdmRandomizedSearchCV(
    estimator=cat,
    param_distributions=param_dist_cat,
    n_iter=20,
    scoring='neg_mean_absolute_error',
    cv=5,
    n_jobs=-1,
    random_state=42,
    verbose=0
)
search_cat.fit(X_enc, y_enc)
best_cat = search_cat.best_estimator_


#set up lassoCV for stacking
meta = LassoCV(
    alphas=[0.001, 0.01, 0.1, 1.0, 10.0],
    cv=5,
    n_jobs=-1,
    max_iter=5000
)


#set up stack
stack = StackingRegressor(
    estimators=[
        ('hgb',  hgb),
        ('xgb',  best_xgb),
        ('cat',  best_cat),
    ],
    final_estimator=meta,
    passthrough=True,
    cv=KFold(n_splits=5, shuffle=True, random_state=42),
    n_jobs=-1
)

#train stack on 3 models
stack.fit(X_enc, y_enc)

preds = stack.predict(X_test_enc)
with open("stacked_hgb_xgb_cat_lasso.txt", "w") as f:
    for p in preds:
        f.write(f"{p:.2f}\n")
```