Milan Rent Prices Forecasting

May 2025 – Letizia Dimonopoli 3132775

Abstract

In this paper, I investigate different machine learning approaches to forecast rent prices in Milan. My analysis begins with a detailed data cleaning, preparation, and feature engineering based on geographic and text data about the properties. I analyze different models to get the best performance, from Gradient Boosting to Random Forest to CatBoost. The model that achieved the smallest prediction error is CatBoost.

1 Steps and codes for the final submitted predictions

First, I import the datasets. Before starting with any model, I need to prepare and clean the data.

1.1 Data Cleaning

In the instructions, it is indicated that the variable `w` in the train dataset is always 1, thus it can be discarded.

```
train.drop("w", axis = 1, inplace = True)
```

First, I want to visualize the count (and percentage) of the missing variables in the dataset to be able to start the data cleaning. I want to explore them before possibly removing them or imputing them. There are some variables that have quite a few missing variables. In particular, these variables are:

- `availability`, for which there isn't much I can do as I can't know when a house becomes available;
- `energy_efficiency_class`, for which I will impute the mean value per zone;
- `condominium_fees`, for which I will also impute the mean value per zone;
- `conditions`, which might be a little harder to deal with: the options are only "good condition", "new" or "excellent". Since we don't know the reason why one or the other was picked and it can be very subjective, we will impute "good condition" to all the missing ones, which is general enough;
- `other_features`, which I can't deduce from any of the other columns, thus I will simply put "unknown";
- `floor`, which again is hard to deduce based on the other observations we have, so I am going to impute the mode of the floor column.

I am going to deal with the missing values of `other_features` and `floor` first, as they are the most straightforward as there is nothing I can do to deduce them. For the column `floor`, I am also converting any non-numerical value into a numerical value for simplicity.

```
train["other_features"] = train["other_features"].apply(lambda x: ["unknown"]
    if pd.isna(x) else x)
test['other_features'] = test["other_features"].apply(lambda x: ["unknown"] if
    pd.isna(x) else x)
train["floor"] = train["floor"].fillna(int(train["floor"].mode()[0]))
test["floor"] = test["floor"].fillna(int(test["floor"].mode()[0]))

floor_mapping = {"ground floor": 0, "mezzanine": 0.5, "semi-basement": -1}

def convert_floor(value):
    value = str(value).lower().strip()
    if value in floor_mapping:
        return int(floor_mapping[value])
    match = re.search(r"\d+", value)
```

```
    if match:
        return int(match.group())
    return int(value)

train["floor"] = train["floor"].apply(convert_floor)
test["floor"] = test["floor"].apply(convert_floor)
```

For my analysis, and also for imputing some of the missing values mentioned above, I really believe that the zone the apartment is located in is one of the variables that affects its price the most. Therefore, I transform the column "zone" into a list. I then transform each zone name into its corresponding "via", "piazza", "corso", etc (I used the help of ChatGPT but since it still made some mistakes, I had to manually inspect the result afterwards). I assume all were places in Milan, e.g. Abbiategrasso is via Abbiategrasso and not the town. I decide to use the geopy library for scaling. Basically what I would like to do is to transform each zone into a readable area (e.g. "Farini" will become "via carlo farini, Milano, Italia") to feed it to the geocoder, which will output latitude and longitude coordinates. In this way, I can then use the coordinates to compute a distance from Duomo (coordinates: (45.4642, 9.1900), which I am considering as the center of Milan). The distance is stored in a variable that I call `distance_km`. Finally, I classify the distance of every zone. I will call this the `proximity_to_center`, where 5 represents the furthest area (>15km) and 1 represents the closest area (in this case the area of the Duomo itself, <=1km). In further analyses, I will use more the variable `distance_km` compared to `proximity_to_center`, as it is more accurate and yields better results.

The code for this is quite long, so please refer to the Appendix for the Python implementation.

Images (a) and (b) show a brief overview of what the output looks like: it isn't perfect, but it is good enough for my analysis.

10 Smallest Proximity Zones:

	zone_rewritten	proximity_to_center
20	via brera, Milano, Italia	1
23	via francesco sforza, Milano, Italia	1
46	piazza missori, Milano, Italia	1
51	piazza della scala, Milano, Italia	1
61	via torino, Milano, Italia	1
129	piazza del duomo, Milano, Italia	1
201	corso di porta ticinese, Milano, Italia	1
390	via della spiga, Milano, Italia	1
820	piazza san babila, Milano, Italia	1
1799	piazza san carlo, Milano, Italia	1

(a) Smallest Proximity Zones

10 Largest Proximity Zones:

	zone_rewritten	proximity_to_center
9	via lanza, Milano, Italia	5
24	via pasteur, Milano, Italia	5
48	via monterosa, Milano, Italia	5
119	via molise, Milano, Italia	5
454	via emilia, Milano, Italia	5
55	via filippo turati, Milano, Italia	4
63	via luigi pirandello, Milano, Italia	4
86	via tripoli, Milano, Italia	4
224	via amendola, Milano, Italia	4
479	via ghisolfa, Milano, Italia	4

(b) Largest Proximity Zones

Let's now complete the cleaning. The column `condominium_fees` has 197 missing values. I decide to impute them, and I choose to fill missing values based on the mean fee of each zone of Milan. This is because I believe that zones have pretty much similar buildings, thus their respective fees (also after a careful manual analysis) should be more or less similar. In fact, after performing the imputation, I observe that the highest condominium fees are in piazza Tre Torri (CityLife area), which a mean of 481€, while the lowest ones are in via Ortica (close to Parco Forlanini), with a mean of around 58€. This is consistent with my overall expectations.

```
zone_fee_means = train.groupby("zone_rewritten")["condominium_fees"].mean()
def impute_fee(row):
    if pd.isna(row["condominium_fees"]):
        return zone_fee_means.get(row["zone_rewritten"], None)
    else:
        return row["condominium_fees"]
train["condominium_fees"] = train.apply(impute_fee, axis=1)

#TEST
zone_fee_means_test = test.groupby("zone_rewritten")["condominium_fees"].mean()
def impute_fee(row):
    if pd.isna(row["condominium_fees"]):
        return zone_fee_means_test.get(row["zone_rewritten"], None)
    else:
        return row["condominium_fees"]
test["condominium_fees"] = test.apply(impute_fee, axis=1)
```

I do the same with the 380 missing values of the column `energy_efficiency_class`. This is because I believe that similar areas have buildings that have been built in approximately the same decade, and they are supposed to

have the same energy class (even though they might have been renovated, but there's nothing I can do to learn that). The code here is slightly different as the variables are categorical and not numerical.

```
train["energy_efficiency_class"] = train["energy_efficiency_class"].astype(str)
    .str.lower().str.strip()
valid_classes = {"a", "b", "c", "d", "e", "f", "g"}
train.loc[~train["energy_efficiency_class"].isin(valid_classes), "
    energy_efficiency_class"] = np.nan

zone_energy = train.dropna(subset=["energy_efficiency_class"]).groupby("
    zone_rewritten")["energy_efficiency_class"].agg(lambda x: x.mode()[0])

def impute_energy_class(row):
    val = row["energy_efficiency_class"]
    if pd.isna(val):
        val = zone_energy.get(row["zone_rewritten"], "g")
    return str(val).lower().strip()

energy_map = {"a":1, "b": 2, "c": 3, "d": 4, "e": 5, "f": 6, "g": 7}

train["energy_efficiency_class"] = train.apply(impute_energy_class, axis=1)
train["energy_efficiency_class"] = train["energy_efficiency_class"].map(
    energy_map)

#TEST
zone_energy_test = test.dropna(subset=["energy_efficiency_class"]).groupby("
    zone_rewritten")["energy_efficiency_class"].agg(lambda x: x.mode()[0])

def impute_energy_class(row):
    if pd.isna(row["energy_efficiency_class"]):
        return zone_energy_test.get(row["zone_rewritten"], "g")
    else:
        return row["energy_efficiency_class"]

test["energy_efficiency_class"] = test.apply(impute_energy_class, axis=1)
test["energy_efficiency_class"] = test["energy_efficiency_class"].str.lower().
    map(energy_map)
```

For the variable `availability`, 14% of the data has missing values. It is way too much for it to be removed completely, so I assume these apartments are available right away: I can not infer a precise date of availability from the data.
The last variable I need to deal with is `conditions`. As mentioned before, conditions of a house for rent are very subjective, and the available options for this dataset are only "new", "excellent" and "good condition". I assume that all the missing ones are in good conditions.

```
train["availability"].fillna("available", inplace=True)
train["conditions"].fillna("good condition", inplace=True)
test["availability"].fillna("available", inplace=True)
test["conditions"].fillna("good condition", inplace=True)

condition_mapping = {"new": 1, "excellent": 2, "good condition": 3}
train["conditions"] = train["conditions"].map(condition_mapping)
test["conditions"] = test["conditions"].map(condition_mapping)
```

Finally, after having explored the data types of the observations for each column, I see that the elevator column has strings ("yes" and "no"). I want to make it binary, thus using an OrdinalEncoder which will put 0 for every "no" and 1 for every "yes".

```
ordinal_map = {"no": 0, "yes": 1}
ordinal_encoder = OrdinalEncoder(categories=[list(ordinal_map.keys())])
train["elevator"] = ordinal_encoder.fit_transform(train[["elevator"]]).astype(
    int)

```

```
test["elevator"] = ordinal_encoder.fit_transform(test[["elevator"]]).astype(int
    )
```

1.2 Data Preparation

Let's proceed to the preparation of the data. After having observed a bit of the train dataset, I see that the columns `description` and `other_features` actually have a lot of different material inside them. Thus, I use some NLP techniques to extract what I am interested in. For example, I don't really care about the type of kitchen (e.g. "kitchenette", "open kitchen", etc) but I believe it is important to have a kitchen compared to not having it. I would also like to have a few more columns of dummy variables: one indicating the number of rooms (excluding bathroom(s) and kitchen), one for the number of bedrooms, and one for the number of bathrooms.

```
def parse_description(desc):
    #is there a kitchen?
    kitchen_present = int("kitchen" in desc.lower())
    #total rooms
    room_match = re.match(r"(\d+)", desc)
    total_rooms = int(room_match.group(1)) if room_match else None
    #nr bedrooms
    bedroom_match = re.search(r"(\d+)\s+bedroom", desc)
    bedrooms = int(bedroom_match.group(1)) if bedroom_match else 0
    #nr bathrooms
    bathroom_match = re.search(r"(\d+)\s+bathroom", desc)
    bathrooms = int(bathroom_match.group(1)) if bathroom_match else 0
    #other rooms
    other_rooms = total_rooms if total_rooms is not None else bedrooms
    return pd.Series({
        "kitchen_present": kitchen_present,
        "nr_rooms": other_rooms,
        "nr_bedrooms": bedrooms,
        "nr_bathrooms": bathrooms})

parsed_features = train["description"].apply(parse_description)
train = pd.concat([train, parsed_features], axis=1)

parsed_features_test = test["description"].apply(parse_description)
test = pd.concat([test, parsed_features_test], axis=1)
```

The column `other_features` is trickier, as I am not interested in all the features that are mentioned. In particular, I don't believe that the following would contribute in a significantly higher rent price: video entryphone, optic fiber, security door, centralized TV system, concierge (unless it is a property of luxury, but I can't infer this from the data), internal/external exposure, window frame materials, alarm system, closet, electric gate. All of them together could make the apartment more prestigious and might be the cause of a significant raise in the price. However, not all apartments have the same set of features, and considering them one by one wouldn't be that important in the price difference. Therefore I do not want to create dummies for these features.
Nonetheless, after a careful analysis, I see other features I am more interested in, which are the following:

- balcony
- terrace
- private garden
- shared garden
- furnished / partially furnished

I will classify the first four in the same column to not have too many dummies (even though I am aware that a garden and a balcony are not exactly the same). Then, I will create another column of dummies for "furnished" and "partially furnished" (1) or "unfurnished" (0).

```
features_lower = train["other_features"]
features_lower_test = test["other_features"]

```

```
#outdoors
outdoor_keywords = ["balcony", "terrace", "private garden", "shared garden"]
train["has_outdoor_space"] = features_lower.apply(lambda x: int(any(feature in
    x for feature in outdoor_keywords)))

test["has_outdoor_space"] = features_lower_test.apply(lambda x: int(any(feature
     in x for feature in outdoor_keywords)))

#furnished?
def get_furnishing_status(text):
    if "partially furnished" in text:
        return 1
    elif "furnished" in text:
        return 1
    else:
        return 0

train["is_furnished"] = features_lower.apply(get_furnishing_status) #contains
    also partially furnished
test["is_furnished"] = features_lower_test.apply(get_furnishing_status) #
    contains also partially furnished
```

No missing data is left now and everything is prepared for my analyses. I also did some checks for the outliers with the help of boxplots but they were not significant at this point of the analysis. I don't want to remove anything that I might need later on.
Finally, I also feature engineered the two following variables:

- `zone_avg_price`: for which I compute the mean price per zone, as it is explained by the name;
- `zone_cluster`: for which I divide the zones into 10 clusters, to capture even better spatial similarity.

Below is the code for both.

```
zone_avg_price = train.groupby("zone_rewritten")["y"].mean().to_dict()
train["zone_avg_price"] = train["zone_rewritten"].map(zone_avg_price)
test["zone_avg_price"] = test["zone_rewritten"].map(zone_avg_price)

kmeans = KMeans(n_clusters=10, random_state=42)
train["zone_cluster"] = kmeans.fit_predict(train[["lat", "lon"]])
test["zone_cluster"] = kmeans.predict(test[["lat", "lon"]])
```

The data cleaning and preparation is now complete.

2 Description of the models

In this section, I explore the models that I used, in particular with a detailed description of the one of my submission, CatBoost, as we didn't study it in this class.

2.1 Trial models

First, as recommended in the exercise sessions did in class, I decide to see what the mean of the prices of rent is and use that as a baseline score for my analyses. I should never do worse than that. My score is obviously very high, reaching 880. Then, I decide to proceed with a Linear Regression model, with an improved score of 565. In this case, I didn't do anything special, but I decided to drop some columns which I believed weren't necessary for the analysis. For instance, I decided not to consider in my regression the following columns:

- `contract_type`, as most of the values were very similar and didn't give me any insight. Most of them are just "rent" and some are "4+4" or "students (6 - 36 months)" or "transitory". I know it could influence a bit but it would be a bit hard to classify them;
- `availability`, which was also the variable with the most missing values, reaching 645. Most of them are marked as "available", and I don't believe that the date when it becomes available would have a

huge influence on the price, especially because we don't have the exact date when this dataset has been compiled;

- `description`, which I'm dropping because I analyzed it previously and extracted the number of bedrooms and the number of bathrooms, which I am going to use instead;
- `other_features`, for the same reason as above. I previously extracted the features I believed were important for my analysis, so I don't need the whole column anymore;
- `zone_rewritten`, as I have feature engineered other features that I can use now, such as `distance_km`, `lon`, and `lat`.

At the beginning, in my Linear Regression, I also didn't consider the features `energy_efficiency_class`, `floor`, and `conditions`, because some of the analyses I performed suggested they weren't that important. However, I then decided to keep them in my following models.
Subsequently, I tried Gradient Boosting, because the data had nonlinear interactions that couldn't be captured by Linear Regression. This can be noticed, for instance, by the two variables `square_meters` and `zone_avg_price` which interact. I achieved a score of around 444, but I still believed my score was too high and that it could be improved. Thus, I decided to try some different models, from Random Forest (to ameliorate the variance reduction of bagging without increasing a lot the bias), to different variations of Gradient Boosting. CatBoost outperformed all the models.

2.2 CatBoost

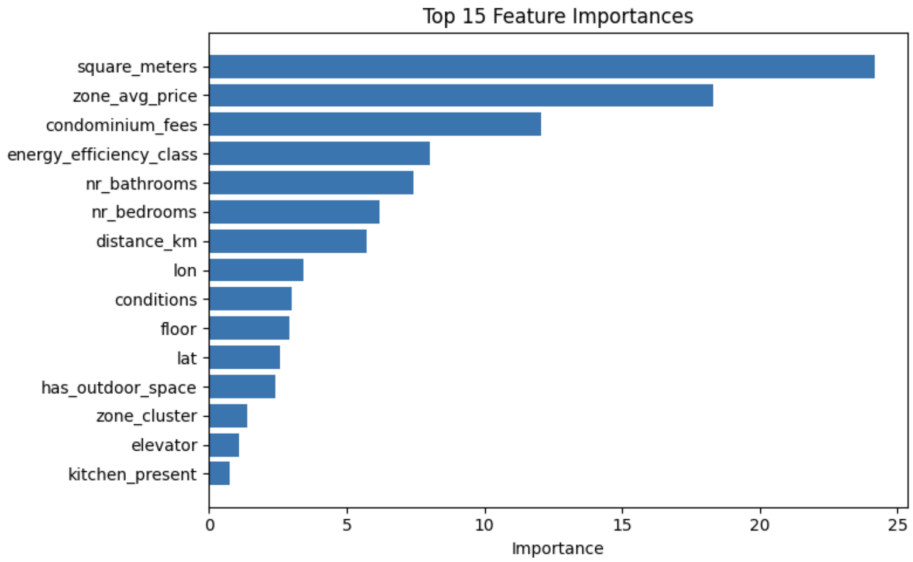
The following are the variables that I am considering for my CatBoost model:
`square_meters`, `conditions`, `floor`, `elevator`, `energy_efficiency_class`, `condominium_fees`, `lat`, `lon`, `distance_km`, `kitchen_present`, `nr_bedrooms`, `nr_bathrooms`, `has_outdoor_space`, `is_furnished`, `zone_avg_price`, `zone_cluster`.
I use the MinMaxScaler to scale all features between 0 and 1. Then, I train the CatBoost regressor on the scaled features (the lines of code stacking the features with hstack is quite useless, but I tried to perform PCA before, which didn't help, so the piece of code is there as I needed it to stack the embedded descriptions). I use the MAE because it is robust to outliers. Since we didn't see CatBoost in this class, below is a description of how it works.

```
drop_cols = train.drop["y", "contract_type", "availability", "description",
             "other_features", "zone_rewritten", "nr_rooms", "
                proximity_to_center"]
X_base = train.drop(columns=drop_cols)
y = train["y"]

X_base["zone_avg_price"] = train["zone_avg_price"]
X_base["zone_cluster"] = train["zone_cluster"]
test_base = test[X_base.columns]
test_base["zone_avg_price"] = test["zone_avg_price"]
test_base["zone_cluster"] = test["zone_cluster"]

scaler = MinMaxScaler()
X_base_scaled = scaler.fit_transform(X_base)
X_test_base_scaled = scaler.transform(X_test_base)

X_train_combined = np.hstack([X_base_scaled])
X_test_combined = np.hstack([X_test_base_scaled])

cat_model = CatBoostRegressor(
    iterations=1000,
    depth=8,
    learning_rate=0.05,
    loss_function="MAE",
    random_seed=42,
    early_stopping_rounds=50,
    verbose=100
)
cat_model.fit(X_train_combined, y)
test_predictions = cat_model.predict(X_test_combined)
```

CatBoost, from Category Boosting, is a type of Gradient Boosting. It is an Ensemble Method. Specifically, it is a high-performing Gradient Boosting library. It stems from decision trees algorithms. CatBoost builds trees one after the other, learning from the errors in the previous trees. Basically, it's like playing a game (for example where somebody asks you a certain number n of questions) over and over again. Every time you play the game, you would ask better questions to get closer and closer to the exact answer because you learned from each of the previous rounds.
CatBoost is known for its ability to capture categorical data, however, I don't really needed this because I had already one-hot encoded most of my features. Nonetheless, CatBoost is also very robust to overfitting, and since the dataset wasn't extremely large, I believe this helped quite a lot. Finally, CatBoost does also very well with imbalanced data, and I had features like `conditions` (prevalence of 1s, count: 3191), `elevator` (prevalence of 1s, count: 3198), `nr_bathrooms` (prevalence of 1 bathroom, count: 3373) and `kitchen_present` (prevalence of 1s, count: 3916), etc, that were quite imbalanced.

3 Insights of the analysis

There are a few relevant patterns that arose from my analysis. As you can see in the barplot below, I analyzed Feature Importance after training my CatBoost model. The variable that has the most influence is `square_meters`.



This is very expected, as bigger apartments tend to cost more. The second variable is `zone_avg_price`, which captures the prices per zone. This is also anticipated as different locations offer distinct types of infrastructure and amenities, thus differently influencing the rental market. `Condominium_fees` and `energy_efficiency_class` are also quite important, which means they are features that are considered and valued by the rental market.
Geographical features are not at the top but they have their relevance. We can observe that rent prices in Milan seem to be very dependent on the location where the property is located. This was my initial assumption when I decided to feature engineer variables such as `distance_km` or `proximity_to_center`. Properties that are close to the city center (in Milan, the Duomo) or to trending areas (such as CityLife) appear to be more expensive. Even clustering with K-Means helped in discovering these hidden spatial characteristics, even though it is less significant than other features.

4 Conclusion

In conclusion, rent prices in Milan are driven by different factors, combining apartment characteristics and spatial features. Among all of them, space and average price per area remain the strongest ones.

5 Appendix

5.1 Code to retrieve a measure of the distance to Duomo

```
zones_list = train["zone"].tolist()

def rewrite_zones(zones_list):
    rewritten = []

    for elem in zones_list:
        parts = elem.split(" - ")
        resolved_part = None
        ambiguous_names = {"abbiategrasso": "piazza abbiategrasso", "affori": "
            via affori","amendola": "via amendola","arco della pace": "arco
            della pace", "arena": "via arena","argonne": "viale argonne","
            ascanio sforza": "alzaia naviglio pavese","baggio": "via baggio", "
            bande nere": "via bande nere","barona": "via barona","bicocca": "
            viale bicocca","bignami": "viale fulvio testi", "bisceglie": "via
            bisceglie","bocconi": "via bocconi","bologna": "via emilia","
            borgogna": "via borgogna", "bovisa": "via bovisa","brenta": "viale
            brenta","brera": "via brera","bruzzano": "piazza bruzzano","buenos
            aires": "corso buenos aires", "buonarroti": "via buonarroti","ca'
            granda": "viale ca' granda", "cadore": "via cadore","cadorna": "
            piazzale cadorna","cantalupa": "via del mare","carrobbio": "via
            torino","via cesare correnti": "via leoncavallo", "cascina merlata"
            : "via pier paolo pasolini","casoretto": "via casoretto","castello"
            : "piazza castello","cenisio": "via cenisio", "centrale": "piazza
            duca d'aosta", "cermenate": "via cermenate", "certosa": "viale
            certosa","chiesa rossa": "via della chiesa rossa","cimiano": "via
            padova","citt  studi": "viale romagna", "city life": "piazza tre
            torri","comasina": "via comasina", "corsica": "viale corsica","
            corvetto": "piazzale corvetto","crescenzago": "via padova", "
            crocetta": "largo della crocetta","cuoco": "via cuoco", "darsena":
            "viale gorizia","de angeli": "piazza de angeli","dergano": "via
            dergano","dezza": "via dezza", "duomo": "piazza del duomo","faenza"
            : "via faenza", "famagosta": "via famagosta","farini": "via carlo
            farini","fatima": "via diomede","frua": "via frua", "gallaratese":
            "via gallarate","gambara": "piazzale gambara","garibaldi": "corso
            garibaldi", "ghisolfa": "via ghisolfa", "giambellino": "via
            giambellino","gorla": "via gorla","gratosoglio": "via dei missaglia
            ","greco": "via prospero finzi", "guastalla": "via francesco sforza
            ","indipendenza": "corso indipendenza","inganni": "via inganni", "
            insubria": "piazza insubria","isola": "via borsieri","istria": "
            piazzale istria", "lambrate": "piazza gobetti","lanza": "via lanza"
            ,"lodi": "corso lodi", "lorenteggio": "via lorenteggio","lotto": "
            piazzale lotto","mac mahon": "via mac mahon", "maggiolina": "via
            melchiorre gioia","manzoni": "via alessandro manzoni", "martini": "
            viale martini","mecenate": "via mecenate", "meda": "via meda","
            medaglie d'oro": "piazza medaglie d'oro", "melchiorre gioia": "via
            melchiorre gioia","missori": "piazza missori","molise": "via molise
            ","montenero": "viale monte nero", "morgagni": "via morgagni","
            moscova": "via della moscova", "musocco": "via musocco","navigli":
            "alzaia naviglio grande", "niguarda": "piazza belloveso","ortica":
            "via ortica","pagano": "via mario pagano", "palestro": "via
            palestro","paolo sarpi": "via paolo sarpi", "parco trotter": "via
            giuseppe giacosa", "parco vittoria": "viale certosa","pasteur": "
            via pasteur","pezzotti": "via pezzotti", "piave": "viale piave","
            plebisciti": "corso plebisciti","ponale": "via ponale","ponte
            lambro": "via ponte lambro", "porta nuova": "piazza gae aulenti","
            porta romana": "corso di porta romana","porta venezia": "corso
            buenos aires", "porta vittoria": "viale monte nero","portello": "
            piazzale portello","prato centenaro": "viale ca' granda", "precotto
            ": "via precotto","primaticcio": "via primaticcio","qt8": "piazza
            santa maria nascente", "quadrilatero della moda": "via della spiga"
```

```
            ,"quadronno": "via quadronno", "quartiere adriano": "via adriano","
            quartiere feltre": "via feltre","quartiere forlanini": "via
            forlanini", "quarto cagnino": "via quarto cagnino","quarto oggiaro"
            : "via quarto oggiaro", "repubblica": "piazza della repubblica","
            ripamonti": "via ripamonti", "rogoredo": "via rogoredo","rovereto":
             "via rovereto","rubattino": "via rubattino", "san babila": "piazza
             san babila","san siro": "piazzale dello sport","sant'ambrogio": "
            piazza sant'ambrogio", "santa giulia": "via cassinari","scala": "
            piazza della scala", "segnano": "viale fulvio testi", "sempione": "
            corso sempione","soderini": "via soderini","solari": "via solari",
            "sulmona": "via sulmona","susa": "piazzale susa","ticinese": "corso
             di porta ticinese","tre castelli": "via tre castelli", "trenno": "
            via trenno","tricolore": "piazza del tricolore", "tripoli": "via
            tripoli","turati": "via filippo turati","turro": "via turro", "
            udine": "piazzale udine","vercelli": "corso vercelli","vigentino":
            "via vigentino","villa san giovanni": "viale monza", "vincenzo
            monti": "via vincenzo monti","wagner": "piazza wagner", "washington
            ": "via luigi pirandello","zara": "viale zara","monte rosa": "via
            monterosa", "ponte nuovo": "via ponte nuovo","san carlo": "piazza
            san carlo","san paolo": "via san paolo", "san vittore": "via san
            vittore","abbiategrasso": "piazza abbiategrasso"}
        for part in parts:
            key = part.strip().lower()
            resolved_part = ambiguous_names.get(key, part.strip())
            break
        formatted = ", ".join([resolved_part.strip(), "Milano", "Italia"])
        rewritten.append(formatted)
    return rewritten

new_zone_names = rewrite_zones(zones_list)
train["zone_rewritten"] = rewrite_zones(train["zone"].tolist())

test["zone_rewritten"] = rewrite_zones(test["zone"].tolist())

from geopy.geocoders import Nominatim
from geopy.distance import geodesic
import time
duomo_coords = (45.4642, 9.1900) #duomo coordinates
geolocator = Nominatim(user_agent="my_milan_script")

#THIS IS TO PREVENT IT RUNNING OVER AND OVER BECAUSE IT TAKES A WHILE
#I CAN SEND THE zone_coordinates.csv FILE IF NECESSARY
try:
    coords_df = pd.read_csv("zone_coordinates.csv")
    coords_dict = dict(zip(coords_df["zone"], zip(coords_df["lat"], coords_df["
        lon"], coords_df["distance_km"])))
except FileNotFoundError:
    coords_dict = {}

latitude_list = []
longitude_list = []
distance_km_list = []
proximity_to_center_list = []

for zone in train["zone_rewritten"]:
    if zone in coords_dict:
        lat, lon, dist = coords_dict[zone]
        latitude_list.append(lat)
        longitude_list.append(lon)
        distance_km_list.append(dist)
    else:
        location = geolocator.geocode(zone, timeout=5)
        time.sleep(1)
        if location:
            coords = (location.latitude, location.longitude)
```

```
            distance_km = geodesic(coords, duomo_coords).km
            latitude_list.append(coords[0])
            longitude_list.append(coords[1])
            distance_km_list.append(distance_km)
            coords_dict[zone] = (coords[0], coords[1], distance_km)
        else:
            coords_dict[zone] = (None, None, None)
            latitude_list.append(None)
            longitude_list.append(None)
            distance_km_list.append(None)
            print(f"NOT FOUND: {zone}")

train["lat"] = latitude_list
train["lon"] = longitude_list
train["distance_km"] = distance_km_list

def classify_proximity(distance):
    if distance is None:
        return None
    elif distance <= 1:
        return 1
    elif distance <= 5:
        return 2
    elif distance <= 10:
        return 3
    elif distance <= 15:
        return 4
    else:
        return 5

train["proximity_to_center"] = train["distance_km"].apply(classify_proximity)

#CODE USED TO SAVE IT FOR NEXT TIMES
#pd.DataFrame([{"zone": k, "lat": v[0], "lon": v[1], "distance_km": v[2]} for k
    , v in coords_dict.items()]).to_csv("zone_coordinates.csv", index=False)

#TEST SET
latitude_list_test = []
longitude_list_test = []
distance_km_list_test = []
proximity_to_center_list_test = []

for zone_test in test["zone_rewritten"]:
    if zone_test in coords_dict:
        lat, lon, dist = coords_dict[zone_test]
        latitude_list_test.append(lat)
        longitude_list_test.append(lon)
        distance_km_list_test.append(dist)

test["distance_km"] = distance_km_list_test
test["proximity_to_center"] = test["distance_km"].apply(classify_proximity)
test["lat"] = latitude_list_test
test["lon"] = longitude_list_test

train.drop(columns=["zone"], inplace=True)
test.drop(columns=["zone"], inplace=True)
```