Supervised_machine_learning_project_by_kirti_gupta (2)

December 8, 2025

[1]:
```
# Cell 1: Imports and configuration
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import joblib
import os
from IPython.display import display, Markdown

from sklearn.model_selection import train_test_split, GridSearchCV,␣
 ↪cross_val_score
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.inspection import permutation_importance

import warnings
warnings.filterwarnings("ignore")
plt.rcParams['figure.figsize'] = (10,5)
```

[2]:
```
# Cell 2: Configuration - set data path and target (edit TARGET if you know it)
data_path = "/content/property_price_data.csv"   # change if needed
# If you know the exact target column name, put it here, otherwise leave as␣
 ↪None to auto-detect.
TARGET = 'SalePrice'   # e.g. 'price' or 'SalePrice' or 'amount' -- set to None␣
 ↪for auto-detect
```

[3]:
```
# Cell 3: Load dataset
assert os.path.exists(data_path), f"Data file not found: {data_path}. Upload␣
 ↪the CSV to /content/ or modify data_path."
df = pd.read_csv(data_path)
display(Markdown("## Dataset loaded"))
```

```
print("Shape:", df.shape)
display(df.head())
```

0.1 Dataset loaded

```
Shape: (970, 69)
```

	Prop_Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	…	3SsnPorch	ScreenPorch	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	YrSold	SaleCondition	SalePrice
0	PRO504	20	RL	100.0	15537	Pave	NaN	IR1	…	0	161	0	NaN	GdWo	NaN	0	2010	Normal	288330
1	PRO102	60	RL	77.0	9534	Pave	NaN	Reg	…	0	0	0	NaN	NaN	NaN	0	2010	Normal	183164
2	PRO609	70	RL	NaN	12781	Pave	NaN	Reg	…	0	0	0	NaN	NaN	NaN	0	2007	Alloca	362145
3	PRO1090	120	FV	37.0	3728	Pave	Pave	IR1	…	0	0	0	NaN	NaN	NaN	0	2006	Normal	196079
4	PRO820	120	RL	44.0	6606	Pave	NaN	IR1	…	0	0	0	NaN	NaN	NaN	0	2010	Partial	228515

```
[5 rows x 69 columns]
```

DATA EXPLORATION (20 marks)

Q1: Dataset overview and summary statistics (5 marks)

1. List item
2. List item

[4]:
```
# Cell 4: Overview and summary statistics
display(Markdown("### Q1: Dataset overview and summary statistics"))
print("Columns and data types:")
display(pd.DataFrame(df.dtypes, columns=['dtype']))

display(Markdown("**Numerical summary statistics**"))
display(df.describe().T)

display(Markdown("**Categorical summary (top values)**"))
cat_cols = df.select_dtypes(include=['object','category']).columns.tolist()
for c in cat_cols:
    print(f"\nColumn: {c}  |  Unique: {df[c].nunique()}")
```

```
    display(df[c].value_counts().head(10))
```

0.1.1 Q1: Dataset overview and summary statistics

Columns and data types:

```
                dtype
Prop_Id        object
MSSubClass      int64
MSZoning       object
LotFrontage   float64
LotArea         int64
…                   …
MiscFeature    object
MiscVal         int64
YrSold          int64
SaleCondition  object
SalePrice       int64

[69 rows x 1 columns]
```

Numerical summary statistics

	count	mean	std	min	25%	\
MSSubClass	970.0	58.092784	42.962071	20.0	20.00	
LotFrontage	789.0	69.447402	24.139429	21.0	57.00	
LotArea	970.0	11208.956701	10153.538896	1663.0	8147.50	
OverallQual	970.0	6.143299	1.396990	1.0	5.00	
OverallCond	970.0	5.547423	1.091443	1.0	5.00	
YearBuilt	970.0	1971.496907	30.247798	1875.0	1954.00	
YearRemodAdd	970.0	1984.741237	20.659310	1950.0	1966.25	
TotalBsmtSF	970.0	1079.728866	463.970814	0.0	808.00	
GrLivArea	970.0	1532.874227	546.642210	438.0	1141.75	
BsmtFullBath	970.0	0.437113	0.518655	0.0	0.00	
BsmtHalfBath	970.0	0.060825	0.247613	0.0	0.00	
FullBath	970.0	1.577320	0.549604	0.0	1.00	
HalfBath	970.0	0.387629	0.504113	0.0	0.00	
BedroomAbvGr	970.0	2.872165	0.807695	0.0	2.00	
KitchenAbvGr	970.0	1.044330	0.220455	0.0	1.00	
TotRmsAbvGrd	970.0	6.559794	1.640884	3.0	5.00	
Fireplaces	970.0	0.640206	0.646840	0.0	0.00	
GarageYrBlt	918.0	1978.879085	24.733202	1900.0	1962.00	
GarageCars	970.0	1.775258	0.737473	0.0	1.00	
GarageArea	970.0	474.715464	214.203638	0.0	336.50	
WoodDeckSF	970.0	93.124742	122.884207	0.0	0.00	
OpenPorchSF	970.0	48.015464	66.676218	0.0	0.00	
EnclosedPorch	970.0	21.357732	59.281725	0.0	0.00	
3SsnPorch	970.0	4.261856	33.019578	0.0	0.00	
ScreenPorch	970.0	16.271134	58.939315	0.0	0.00	

PoolArea	970.0	4.152577	49.241171	0.0	0.00
MiscVal	970.0	48.257732	595.105731	0.0	0.00
YrSold	970.0	2007.815464	1.336156	2006.0	2007.00
SalePrice	970.0	184272.595876	82061.647461	35336.0	132225.00

	50%	75%	max
MSSubClass	50.0	70.00	190.0
LotFrontage	68.0	80.00	313.0
LotArea	10174.5	12482.25	216301.0
OverallQual	6.0	7.00	10.0
OverallCond	5.0	6.00	9.0
YearBuilt	1974.0	2000.00	2009.0
YearRemodAdd	1994.0	2004.00	2010.0
TotalBsmtSF	1008.0	1324.00	6110.0
GrLivArea	1469.0	1792.00	5642.0
BsmtFullBath	0.0	1.00	3.0
BsmtHalfBath	0.0	0.00	2.0
FullBath	2.0	2.00	3.0
HalfBath	0.0	1.00	2.0
BedroomAbvGr	3.0	3.00	6.0
KitchenAbvGr	1.0	1.00	3.0
TotRmsAbvGrd	6.0	7.00	12.0
Fireplaces	1.0	1.00	3.0
GarageYrBlt	1981.0	2002.00	2010.0
GarageCars	2.0	2.00	4.0
GarageArea	480.0	576.00	1418.0
WoodDeckSF	0.0	168.00	736.0
OpenPorchSF	26.0	70.00	547.0
EnclosedPorch	0.0	0.00	552.0
3SsnPorch	0.0	0.00	508.0
ScreenPorch	0.0	0.00	480.0
PoolArea	0.0	0.00	738.0
MiscVal	0.0	0.00	15500.0
YrSold	2008.0	2009.00	2010.0
SalePrice	165641.5	215268.25	758639.0

Categorical summary (top values)

Column: Prop_Id | Unique: 970

Prop_Id	
PRO1032	1
PRO504	1
PRO102	1
PRO609	1
PRO1090	1
PRO820	1
PRO685	1

```
PRO1281    1
PRO921     1
PRO1454    1
Name: count, dtype: int64


Column: MSZoning  |  Unique: 5

MSZoning
RL         753
RM         158
FV          43
RH           9
C (all)      7
Name: count, dtype: int64


Column: Street  |  Unique: 2

Street
Pave    965
Grvl      5
Name: count, dtype: int64


Column: Alley  |  Unique: 2

Alley
Grvl    32
Pave    24
Name: count, dtype: int64


Column: LotShape  |  Unique: 4

LotShape
Reg    605
IR1    333
IR2     26
IR3      6
Name: count, dtype: int64


Column: LandContour  |  Unique: 4

LandContour
Lvl    865
Bnk     48
HLS     31
Low     26
Name: count, dtype: int64
```

```
Column: Utilities  |  Unique: 2

Utilities
AllPub    969
NoSeWa      1
Name: count, dtype: int64


Column: LotConfig  |  Unique: 5

LotConfig
Inside     699
Corner     172
CulDSac     66
FR2         30
FR3          3
Name: count, dtype: int64


Column: LandSlope  |  Unique: 3

LandSlope
Gtl    913
Mod     48
Sev      9
Name: count, dtype: int64


Column: Neighborhood  |  Unique: 25

Neighborhood
NAmes      140
CollgCr    101
OldTown     75
Edwards     66
Somerst     55
Gilbert     51
NridgHt     50
NWAmes      49
Sawyer      49
SawyerW     41
Name: count, dtype: int64


Column: Condition1  |  Unique: 9

Condition1
Norm      837
Feedr      49
Artery     32
RRAn       18
```

```
PosN       15
RRAe        7
PosA        6
RRNn        4
RRNe        2
Name: count, dtype: int64


Column: Condition2  |  Unique: 6

Condition2
Norm      964
PosN        2
Artery      1
Feedr       1
RRAe        1
RRNn        1
Name: count, dtype: int64


Column: BldgType  |  Unique: 5

BldgType
1Fam      802
TwnhsE     81
Duplex     35
Twnhs      30
2fmCon     22
Name: count, dtype: int64


Column: PropStyle  |  Unique: 8

PropStyle
1Story    481
2Story    288
1.5Fin    105
SLvl       46
SFoyer     24
2.5Unf     10
1.5Unf      9
2.5Fin      7
Name: count, dtype: int64


Column: RoofStyle  |  Unique: 6

RoofStyle
Gable      750
Hip        201
Flat         7
Gambrel      7
```

```
Mansard      3
Shed         2
Name: count, dtype: int64


Column: RoofMatl  |  Unique: 8

RoofMatl
CompShg    953
Tar&Grv      6
WdShngl      4
WdShake      3
ClyTile      1
Membran      1
Metal        1
Roll         1
Name: count, dtype: int64


Column: Exterior1st  |  Unique: 12

Exterior1st
VinylSd    340
HdBoard    148
MetalSd    145
Wd Sdng    129
Plywood     77
CemntBd     42
BrkFace     35
Stucco      21
WdShing     18
AsbShng     13
Name: count, dtype: int64


Column: Exterior2nd  |  Unique: 16

Exterior2nd
VinylSd    331
HdBoard    143
MetalSd    142
Wd Sdng    116
Plywood    102
CmentBd     42
Wd Shng     23
Stucco      21
BrkFace     20
AsbShng     15
Name: count, dtype: int64
```

```
Column: ExterQual  |  Unique: 4

ExterQual
TA    592
Gd    334
Ex     36
Fa      8
Name: count, dtype: int64


Column: ExterCond  |  Unique: 5

ExterCond
TA    858
Gd     91
Fa     18
Ex      2
Po      1
Name: count, dtype: int64


Column: Foundation  |  Unique: 6

Foundation
PConc     438
CBlock    414
BrkTil     98
Slab       15
Stone       4
Wood        1
Name: count, dtype: int64


Column: BsmtQual  |  Unique: 4

BsmtQual
TA    427
Gd    415
Ex     82
Fa     23
Name: count, dtype: int64


Column: BsmtCond  |  Unique: 4

BsmtCond
TA    883
Gd     36
Fa     26
Po      2
Name: count, dtype: int64
```

```
Column: BsmtExposure  |  Unique: 4

BsmtExposure
No    633
Av    138
Gd     99
Mn     76
Name: count, dtype: int64


Column: Heating  |  Unique: 6

Heating
GasA     948
GasW      12
Wall       4
Grav       4
Floor      1
OthW       1
Name: count, dtype: int64


Column: HeatingQC  |  Unique: 4

HeatingQC
Ex    497
TA    285
Gd    154
Fa     34
Name: count, dtype: int64


Column: CentralAir  |  Unique: 2

CentralAir
Y    908
N     62
Name: count, dtype: int64


Column: Electrical  |  Unique: 5

Electrical
SBrkr    895
FuseA     54
FuseF     18
FuseP      2
Mix        1
Name: count, dtype: int64


Column: KitchenQual  |  Unique: 4
```

```
KitchenQual
TA    491
Gd    385
Ex     69
Fa     25
Name: count, dtype: int64


Column: Functional  |  Unique: 7

Functional
Typ     905
Min2     21
Min1     18
Mod      11
Maj1     11
Maj2      3
Sev       1
Name: count, dtype: int64


Column: FireplaceQu  |  Unique: 5

FireplaceQu
Gd    266
TA    218
Fa     21
Ex     18
Po     10
Name: count, dtype: int64


Column: GarageType  |  Unique: 6

GarageType
Attchd     574
Detchd     260
BuiltIn     58
Basment     16
CarPort      6
2Types       4
Name: count, dtype: int64


Column: GarageFinish  |  Unique: 3

GarageFinish
Unf    395
RFn    276
Fin    247
Name: count, dtype: int64
```

```
Column: GarageQual  |  Unique: 5

GarageQual
TA    875
Fa     28
Gd     10
Po      3
Ex      2
Name: count, dtype: int64


Column: GarageCond  |  Unique: 5

GarageCond
TA    883
Fa     21
Gd      8
Po      5
Ex      1
Name: count, dtype: int64


Column: PavedDrive  |  Unique: 3

PavedDrive
Y    893
N     58
P     19
Name: count, dtype: int64


Column: PoolQC  |  Unique: 3

PoolQC
Gd    3
Ex    2
Fa    2
Name: count, dtype: int64


Column: Fence  |  Unique: 4

Fence
MnPrv    93
GdPrv    39
GdWo     31
MnWw      6
Name: count, dtype: int64


Column: MiscFeature  |  Unique: 4
```

```
MiscFeature
Shed    24
Othr     2
Gar2     2
TenC     1
Name: count, dtype: int64


Column: SaleCondition  |  Unique: 6

SaleCondition
Normal     799
Partial     82
Abnorml     66
Family      12
Alloca       8
AdjLand      3
Name: count, dtype: int64
```

Missing data analysis

[5]:
```
# Cell 5: Missing data analysis
display(Markdown("### Q2: Missing data analysis"))
missing = df.isnull().sum().sort_values(ascending=False)
missing_pct = (df.isnull().mean()*100).sort_values(ascending=False)
missing_df = pd.concat([missing, missing_pct], axis=1)
missing_df.columns = ['missing_count','missing_pct']
display(missing_df[missing_df.missing_count>0])
```

0.1.2 Q2: Missing data analysis

	missing_count	missing_pct
PoolQC	963	99.278351
MiscFeature	941	97.010309
Alley	914	94.226804
Fence	801	82.577320
FireplaceQu	437	45.051546
LotFrontage	181	18.659794
GarageCond	52	5.360825
GarageYrBlt	52	5.360825
GarageQual	52	5.360825
GarageFinish	52	5.360825
GarageType	52	5.360825
BsmtExposure	24	2.474227
BsmtQual	23	2.371134
BsmtCond	23	2.371134

Include a short interpretation cell (text) after running: which columns have >30% missing etc.

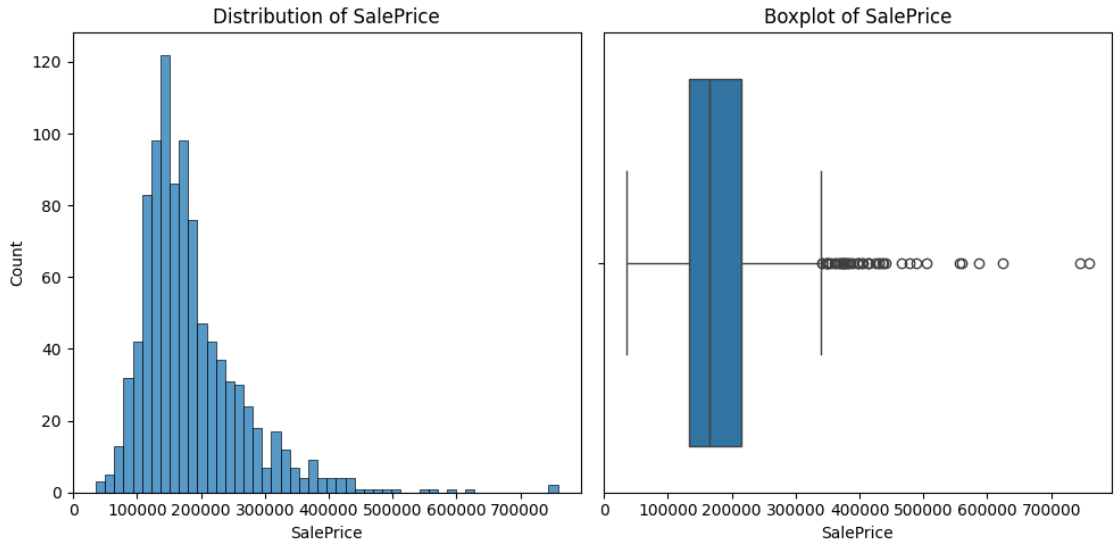
Target variable analysis

[6]:
```
# Cell 6: Target detection and analysis
display(Markdown("### Q3: Target variable analysis"))

# Auto-detect possible targets
if TARGET is None:
    possible_targets = [c for c in df.columns if any(k in c.lower() for k in␣
 ↪['price','sale','amount','target','value','rent'])]
    if len(possible_targets)==0:
        # pick numeric column with highest variance as fallback
        numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()
        if len(numeric_cols)>0:
            TARGET = max(numeric_cols, key=lambda c: df[c].var())
            print("Fallback target chosen (highest variance numeric):", TARGET)
        else:
            raise ValueError("No numeric columns to use as target. Please set␣
 ↪TARGET variable manually.")
    else:
        TARGET = possible_targets[0]
        print("Auto-detected target column:", TARGET)
else:
    print("User-specified target:", TARGET)

# Summary / distribution
display(df[TARGET].describe().to_frame(name='stats'))
plt.subplot(1,2,1)
sns.histplot(df[TARGET].dropna(), bins=50, kde=False)
plt.title(f"Distribution of {TARGET}")
plt.subplot(1,2,2)
sns.boxplot(x=df[TARGET])
plt.title(f"Boxplot of {TARGET}")
plt.tight_layout()
plt.show()

print("Skewness:", df[TARGET].skew(), "Kurtosis:", df[TARGET].kurt())
```

0.1.3 Q3: Target variable analysis

```
User-specified target: SalePrice
```

	stats
count	970.000000
mean	184272.595876
std	82061.647461
min	35336.000000
25%	132225.000000
50%	165641.500000
75%	215268.250000
max	758639.000000




```
Skewness: 1.9756251962264588 Kurtosis: 7.118816958890717
```

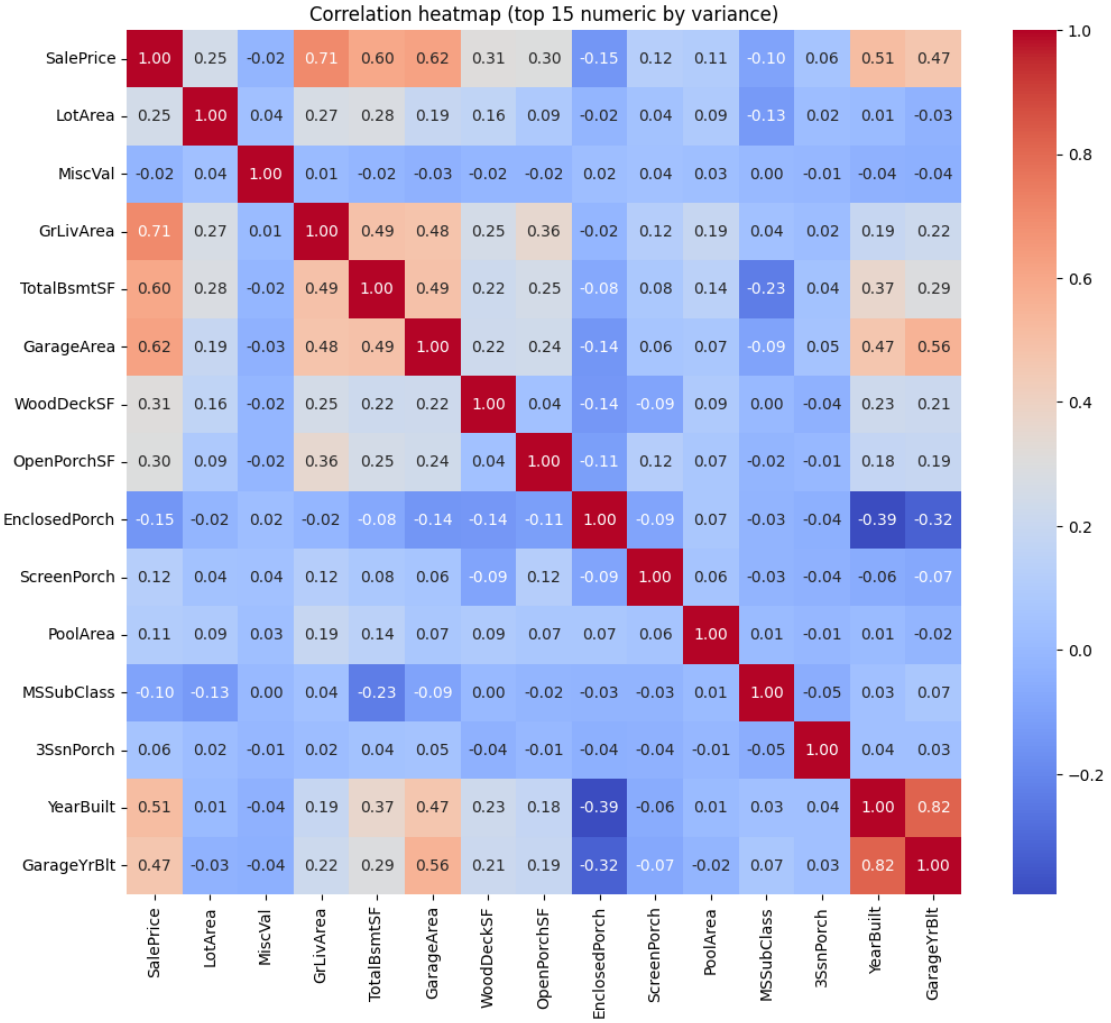
Feature correlation analysis

[7]:
```
# Cell 7: Correlation analysis among numeric features & with target
display(Markdown("### Q4: Feature correlation analysis"))

num_df = df.select_dtypes(include=[np.number]).copy()
corr = num_df.corr()
# show correlations with target
if TARGET in corr.columns:
    corr_with_target = corr[TARGET].drop(labels=[TARGET]).abs().
 ↪sort_values(ascending=False)
    display(Markdown("**Top features correlated with target (absolute␣
 ↪correlation)**"))
    display(corr_with_target.head(15).to_frame(name='abs_corr'))
else:
    print("Target not in numeric columns for correlation analysis.")

# Display heatmap for top numeric features (limit to top 15 by variance)
top_vars = num_df.var().sort_values(ascending=False).head(15).index.tolist()
plt.figure(figsize=(12,10))
sns.heatmap(df[top_vars].corr(), annot=True, fmt=".2f", cmap='coolwarm')
plt.title("Correlation heatmap (top 15 numeric by variance)")
plt.show()
```

0.1.4 Q4: Feature correlation analysis

Top features correlated with target (absolute correlation)

	abs_corr
OverallQual	0.787500
GrLivArea	0.705701
GarageCars	0.642999
GarageArea	0.619476
TotalBsmtSF	0.597116
FullBath	0.574070
TotRmsAbvGrd	0.520046
YearBuilt	0.511482
YearRemodAdd	0.486579
Fireplaces	0.484090
GarageYrBlt	0.466701
LotFrontage	0.387281
WoodDeckSF	0.309042
OpenPorchSF	0.299719
HalfBath	0.271580



DATA PREPROCESSING Missing value treatment

We handle missing values carefully:

Numeric: impute with median (robust), optionally KNN for sensitive columns.

Categorical: impute with 'Missing' for low-cardinality or 'Unknown'; for ordinal categories use mode.

If a column has >50% missing, consider dropping or creating a missing_flag column.

[8]:
```
# Cell 8: Missing value treatment function and applied imputation
display(Markdown("### Missing value treatment"))

df_p = df.copy()

# 1) Create missing flags for columns with >10% missing (useful features)
for c in df_p.columns:
    pct = df_p[c].isnull().mean()
    if pct > 0.10:
        df_p[c + "_missing_flag"] = df_p[c].isnull().astype(int)

# 2) Decide column types
numeric_cols = df_p.select_dtypes(include=[np.number]).columns.tolist()
cat_cols = df_p.select_dtypes(include=['object','category']).columns.tolist()

# 3) Impute numeric using median except if a column is particularly suited for␣
 ↪KNN
from sklearn.impute import SimpleImputer
num_imputer = SimpleImputer(strategy='median')
df_p[numeric_cols] = pd.DataFrame(num_imputer.
 ↪fit_transform(df_p[numeric_cols]), columns=numeric_cols)

# 4) Categorical: fill with 'Missing' then optionally encode later
for c in cat_cols:
    df_p[c] = df_p[c].fillna('Missing')

# 5) Columns with >50% missing: report (user decision to drop or keep)
high_missing = (df.isnull().mean() > 0.5)
print("Columns with >50% missing (consider dropping or justifying keeping):")
display(high_missing[high_missing].index.tolist())

display(Markdown("**Missing values after imputation (should be 0)**"))
display(df_p.isnull().sum().sort_values(ascending=False).head(20))
```

0.1.5 Missing value treatment

```
Columns with >50% missing (consider dropping or justifying keeping):
```

```
['Alley', 'PoolQC', 'Fence', 'MiscFeature']
```

Missing values after imputation (should be 0)

```
Prop_Id          0
MSSubClass       0
MSZoning         0
LotFrontage      0
LotArea          0
Street           0
Alley            0
LotShape         0
LandContour      0
Utilities        0
LotConfig        0
LandSlope        0
Neighborhood     0
Condition1       0
Condition2       0
BldgType         0
PropStyle        0
OverallQual      0
OverallCond      0
YearBuilt        0
dtype: int64
```

Note for report: Document in the submission which columns were imputed and why (median chosen for robustness to outliers; categorical 'Missing' to preserve information about absence).

Outlier detection & handling

We perform IQR capping (winsorization) for numeric features. For the target, cap only for visualization and optionally keep raw values for modeling if you want (we will log-transform target later).

[9]:
```
# Cell 9: Outlier detection & IQR capping
display(Markdown("### Outlier detection and handling (IQR capping)"))

def iqr_cap(series, lower_q=0.25, upper_q=0.75, k=1.5):
    q1 = series.quantile(lower_q)
    q3 = series.quantile(upper_q)
    iqr = q3 - q1
    lower = q1 - k*iqr
    upper = q3 + k*iqr
    return series.clip(lower=lower, upper=upper), lower, upper

df_out = df_p.copy()
outlier_summary = []
for c in numeric_cols:
    if df_out[c].nunique() > 10:
        before = df_out[c].copy()
```

```
        df_out[c], lower, upper = iqr_cap(df_out[c])
        n_below = (before < lower).sum()
        n_above = (before > upper).sum()
        outlier_summary.append({'feature':c, 'n_below':n_below, 'n_above':
 ↪n_above, 'lower':lower, 'upper':upper})
outlier_df = pd.DataFrame(outlier_summary).sort_values('n_above',␣
 ↪ascending=False).head(20)
display(outlier_df)
```

0.1.6 Outlier detection and handling (IQR capping)

	feature	n_below	n_above	lower	upper
11	EnclosedPorch	0	138	0.000	0.000
13	ScreenPorch	0	81	0.000	0.000
0	MSSubClass	0	72	-55.000	145.000
10	OpenPorchSF	0	52	-105.000	175.000
15	SalePrice	0	47	7660.125	339833.125
2	LotArea	0	43	1645.375	18984.375
1	LotFrontage	38	40	33.000	105.000
14	MiscVal	0	27	0.000	0.000
6	GrLivArea	0	25	166.375	2767.375
9	WoodDeckSF	0	20	-252.000	420.000
12	3SsnPorch	0	20	0.000	0.000
5	TotalBsmtSF	23	19	34.000	2098.000
8	GarageArea	0	16	-22.750	935.250
7	GarageYrBlt	1	0	1906.000	2058.000
3	YearBuilt	4	0	1885.000	2069.000
4	YearRemodAdd	0	0	1909.625	2060.625

Data type conversions

[10]:
```
# Cell 10: Data type conversions - detect date-like columns & convert to␣
 ↪datetime, treat year columns as int
display(Markdown("### Data type conversions"))

df_conv = df_out.copy()
for c in df_conv.columns:
    if df_conv[c].dtype == object:
        # try to parse as datetime if many parsable entries
        parsed = pd.to_datetime(df_conv[c], errors='coerce')
        if parsed.notnull().sum() / len(parsed) > 0.5:  # majority parseable
            df_conv[c] = parsed
            print(f"Converted {c} to datetime")
# Convert float columns where all values are integral to int
for c in df_conv.select_dtypes(include=['float']).columns:
    if df_conv[c].dropna().apply(float.is_integer).all():
        df_conv[c] = df_conv[c].astype('Int64')
```

```
        print(f"Converted {c} floats to Int64")
```

0.1.7 Data type conversions

```
Converted MSSubClass floats to Int64
Converted LotFrontage floats to Int64
Converted OverallQual floats to Int64
Converted OverallCond floats to Int64
Converted YearBuilt floats to Int64
Converted YearRemodAdd floats to Int64
Converted TotalBsmtSF floats to Int64
Converted BsmtFullBath floats to Int64
Converted BsmtHalfBath floats to Int64
Converted FullBath floats to Int64
Converted HalfBath floats to Int64
Converted BedroomAbvGr floats to Int64
Converted KitchenAbvGr floats to Int64
Converted TotRmsAbvGrd floats to Int64
Converted Fireplaces floats to Int64
Converted GarageYrBlt floats to Int64
Converted GarageCars floats to Int64
Converted WoodDeckSF floats to Int64
Converted OpenPorchSF floats to Int64
Converted EnclosedPorch floats to Int64
Converted 3SsnPorch floats to Int64
Converted ScreenPorch floats to Int64
Converted PoolArea floats to Int64
Converted MiscVal floats to Int64
Converted YrSold floats to Int64
Converted LotFrontage_missing_flag floats to Int64
Converted Alley_missing_flag floats to Int64
Converted FireplaceQu_missing_flag floats to Int64
Converted PoolQC_missing_flag floats to Int64
Converted Fence_missing_flag floats to Int64
Converted MiscFeature_missing_flag floats to Int64
```

Data quality validation

[11]:
```
# Cell 11: Data quality checks
display(Markdown("### Data quality validation"))

# duplicates
dups = df_conv.duplicated().sum()
# negative or zero target?
neg_target = (df_conv[TARGET] <= 0).sum()
print("Duplicate rows:", dups)
print(f"Non-positive target rows (<=0) in {TARGET}:", neg_target)
```

```
# basic range checks for numeric features
display(df_conv[numeric_cols].agg(['min','max','median']).T.head(20))
```

0.1.8 Data quality validation

```
Duplicate rows: 0
Non-positive target rows (<=0) in SalePrice: 0
```

	min	max	median
MSSubClass	20.0	145.000	50.0
LotFrontage	33.0	105.000	68.0
LotArea	1663.0	18984.375	10174.5
OverallQual	1.0	10.000	6.0
OverallCond	1.0	9.000	5.0
YearBuilt	1885.0	2009.000	1974.0
YearRemodAdd	1950.0	2010.000	1994.0
TotalBsmtSF	34.0	2098.000	1008.0
GrLivArea	438.0	2767.375	1469.0
BsmtFullBath	0.0	3.000	0.0
BsmtHalfBath	0.0	2.000	0.0
FullBath	0.0	3.000	2.0
HalfBath	0.0	2.000	0.0
BedroomAbvGr	0.0	6.000	3.0
KitchenAbvGr	0.0	3.000	1.0
TotRmsAbvGrd	3.0	12.000	6.0
Fireplaces	0.0	3.000	1.0
GarageYrBlt	1906.0	2010.000	1981.0
GarageCars	0.0	4.000	2.0
GarageArea	0.0	935.250	480.0

FEATURE ENGINEERING Target variable transformation — log transform

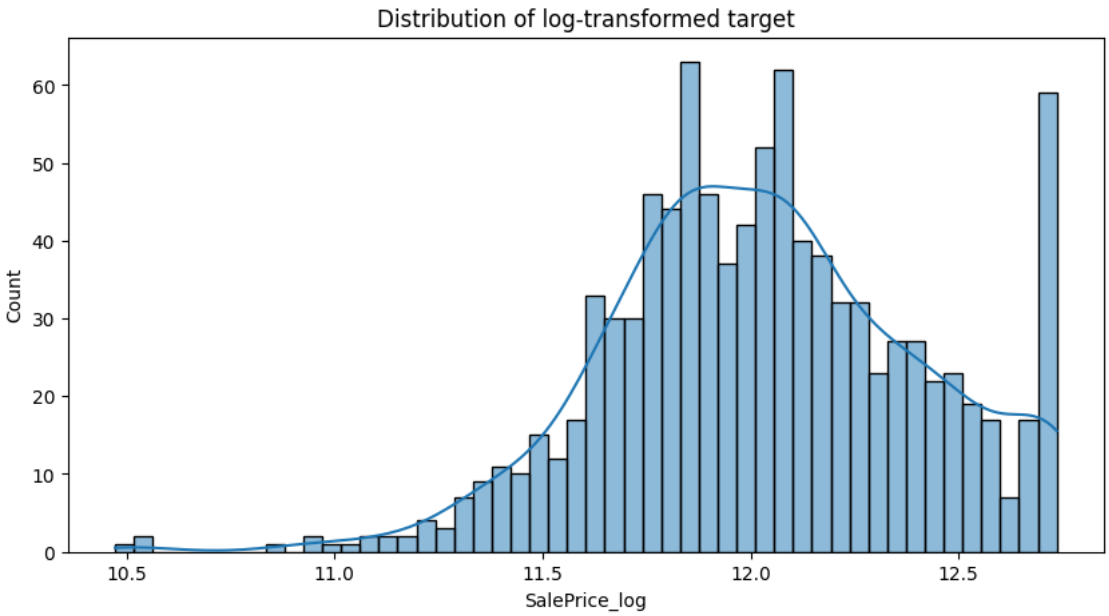
We will create TARGET_log = np.log(TARGET) if all target values > 0. If zeros or negatives exist, use np.log1p after adding a small constant.

[12]:
```
# Cell 12: Target transformation (log)
display(Markdown("### Target variable transformation (log)"))

df_fe = df_conv.copy()
if (df_fe[TARGET] > 0).all():
    df_fe[TARGET + "_log"] = np.log(df_fe[TARGET])
    LOG_TARGET = TARGET + "_log"
    print(f"Applied natural log transform to {TARGET} -> {LOG_TARGET}")
else:
    # shift to positive and use log1p
    shift = 1 - df_fe[TARGET].min() if df_fe[TARGET].min() <= 0 else 0
    df_fe[TARGET + "_log1p"] = np.log1p(df_fe[TARGET] + shift)
    LOG_TARGET = TARGET + "_log1p"
    print(f"Applied log1p transform with shift={shift} -> {LOG_TARGET}")
```

```
# Visualize transformed target
sns.histplot(df_fe[LOG_TARGET].dropna(), bins=50, kde=True)
plt.title("Distribution of log-transformed target")
plt.show()
```

0.1.9 Target variable transformation (log)

```
Applied natural log transform to SalePrice -> SalePrice_log
```



Categorical encoding — one-hot + frequency encoding for high-cardinality

Rules:

For categorical columns with 20 unique values → One-Hot (drop_first=True)

For categorical columns with > 20 unique values → Frequency encoding (value counts / n)

For ordinal categories (if known) → OrdinalEncoder (requires user input)

[13]:
```
# Cell 13: Categorical encoding
display(Markdown("### Categorical encoding"))

df_enc = df_fe.copy()

# Identify categorical features (again)
cat_cols = df_enc.select_dtypes(include=['object','category']).columns.tolist()

ohe_cols = []
```

```
freq_cols = []
for c in cat_cols:
    nunique = df_enc[c].nunique()
    if nunique <= 20:
        ohe_cols.append(c)
    else:
        freq_cols.append(c)

print("One-hot encoding columns (<=20 unique):", ohe_cols)
print("Frequency encoding columns (>20 unique):", freq_cols)

# Frequency encoding
for c in freq_cols:
    freq = df_enc[c].value_counts(normalize=True)
    df_enc[c + "_freq_enc"] = df_enc[c].map(freq)
    # Keep original? We can drop original later
# One-hot encoding - use pandas.get_dummies (drop_first to avoid␣
 ↪multicollinearity)
df_enc = pd.get_dummies(df_enc, columns=ohe_cols, drop_first=True)

# Optionally drop original high-cardinality columns if we've created freq␣
 ↪encodings
df_enc.drop(columns=freq_cols, inplace=True, errors='ignore')

print("Encoded dataframe shape:", df_enc.shape)
```

0.1.10 Categorical encoding

```
One-hot encoding columns (<=20 unique): ['MSZoning', 'Street', 'Alley',
'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSlope', 'Condition1',
'Condition2', 'BldgType', 'PropStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
'Exterior2nd', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual', 'BsmtCond',
'BsmtExposure', 'Heating', 'HeatingQC', 'CentralAir', 'Electrical',
'KitchenQual', 'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish',
'GarageQual', 'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature',
'SaleCondition']
Frequency encoding columns (>20 unique): ['Prop_Id', 'Neighborhood']
Encoded dataframe shape: (970, 208)
```

Appendix — Helpful utility snippets (copy to your notebook if needed)

[14]:
```
# Cell 14: Feature Selection
display(Markdown("### Feature Selection"))

# Ensure LOG_TARGET is not treated as a feature itself
# Also drop original TARGET if it's still present and not LOG_TARGET
columns_to_drop = [col for col in [LOG_TARGET, TARGET] if col in df_enc.columns]
```

```
X_for_selection = df_enc.drop(columns=columns_to_drop, errors='ignore')
y_for_selection = df_enc[LOG_TARGET]

# Identify numeric features for SelectKBest
numeric_feats = X_for_selection.select_dtypes(include=np.number).columns.
 ↪tolist()

# SelectKBest with f_regression
k_features = min(30, len(numeric_feats)) # Choose a reasonable K, max 30 for now
selector = SelectKBest(f_regression, k=k_features)
selector.fit(X_for_selection[numeric_feats], y_for_selection)

top_k_features_idx = selector.get_support(indices=True)
top_k_features = X_for_selection[numeric_feats].columns[top_k_features_idx].
 ↪tolist()

print(f"Selected {len(top_k_features)} top features using SelectKBest:")
print(top_k_features)
```

0.1.11 Feature Selection

```
Selected 30 top features using SelectKBest:
['MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond',
'YearBuilt', 'YearRemodAdd', 'TotalBsmtSF', 'GrLivArea', 'BsmtFullBath',
'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd',
'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF',
'OpenPorchSF', 'PoolArea', 'YrSold', 'LotFrontage_missing_flag',
'Alley_missing_flag', 'FireplaceQu_missing_flag', 'PoolQC_missing_flag',
'Fence_missing_flag', 'MiscFeature_missing_flag', 'Neighborhood_freq_enc']
```

Reasoning: Now that feature selection has been completed and `top_k_features` is defined, I will execute 'Cell 15: Prepare final X and y for modeling, train-test split' to prepare the data for model training. This step defines `X_train`, `X_test`, `y_train`, and `y_test`.

[15]:
```
# Cell 15: Prepare final X and y for modeling, train-test split
display(Markdown("### Train-test split"))

# Choose feature set: top_k_features + any encoded indicator cols could be used
features = top_k_features.copy()

# Optionally, add top 10 most important encoded features by correlation
corrs = df_enc[numeric_feats + [LOG_TARGET]].corr()[LOG_TARGET].abs().
 ↪sort_values(ascending=False)
for col in corrs.index:
    if col not in features and col != LOG_TARGET and len(features) < 40:
        features.append(col)
```

```
X = df_enc[features].fillna(0)
y = df_enc[LOG_TARGET]

print("Final features count:", X.shape[1])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
print("Train shape:", X_train.shape, "Test shape:", X_test.shape)
```

0.1.12 Train-test split

```
Final features count: 36
Train shape: (776, 36) Test shape: (194, 36)
```

Reasoning: The previous execution of 'Cell 16: Model training and hyperparameter tuning (GridSearchCV)' failed because `mean_squared_error` was called with `squared=False`, which is not supported. I will modify the `evaluate_model` function to correctly calculate RMSE by taking the square root of the mean squared error.

[16]:
```
# Cell 16: Model training and hyperparameter tuning (GridSearchCV)

display(Markdown("### Model training & parameter tuning"))

# Define evaluation function
def evaluate_model(model, X_test, y_test, label="Model"):
    y_pred = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred)) # Corrected to calculate␣
 ↪RMSE
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    print(f"{label} performance on test set:")
    print(f"RMSE: {rmse:.4f}, MAE: {mae:.4f}, R2: {r2:.4f}")
    return {'rmse':rmse, 'mae':mae, 'r2':r2}

# 1) Baseline: Linear Regression (with scaling)
pipe_lr = Pipeline([
    ('scaler', StandardScaler()),
    ('lr', LinearRegression())
])
pipe_lr.fit(X_train, y_train)
lr_perf = evaluate_model(pipe_lr, X_test, y_test, "Linear Regression␣
 ↪(baseline)")

# 2) Ridge (regularized)
pipe_ridge = Pipeline([('scaler',StandardScaler()), ('ridge', Ridge())])
param_ridge = {'ridge__alpha':[0.1,1.0,10.0,50.0]}
```

```
grid_ridge = GridSearchCV(pipe_ridge, param_grid=param_ridge, cv=5,␣
 ↪scoring='neg_root_mean_squared_error', n_jobs=-1)
grid_ridge.fit(X_train, y_train)
print("Best Ridge params:", grid_ridge.best_params_)
ridge_best = grid_ridge.best_estimator_
ridge_perf = evaluate_model(ridge_best, X_test, y_test, "Ridge (best)")

# 3) Random Forest Regressor (tree-based)
pipe_rf = Pipeline([('scaler', StandardScaler()), ('rf',␣
 ↪RandomForestRegressor(random_state=42, n_jobs=-1))])
param_grid_rf = {
    'rf__n_estimators':[100,200],
    'rf__max_depth':[None,10,20],
    'rf__min_samples_split':[2,5]
}
grid_rf = GridSearchCV(pipe_rf, param_grid=param_grid_rf, cv=3,␣
 ↪scoring='neg_root_mean_squared_error', n_jobs=-1)
grid_rf.fit(X_train, y_train)
print("Best RF params:", grid_rf.best_params_)
rf_best = grid_rf.best_estimator_
rf_perf = evaluate_model(rf_best, X_test, y_test, "Random Forest (best)")
```

0.1.13 Model training & parameter tuning

```
Linear Regression (baseline) performance on test set:
RMSE: 0.1242, MAE: 0.0865, R2: 0.8931
Best Ridge params: {'ridge__alpha': 1.0}
Ridge (best) performance on test set:
RMSE: 0.1241, MAE: 0.0865, R2: 0.8932
Best RF params: {'rf__max_depth': None, 'rf__min_samples_split': 2,
'rf__n_estimators': 200}
Random Forest (best) performance on test set:
RMSE: 0.1404, MAE: 0.0923, R2: 0.8633
```

0.2 Execute Performance Metrics Display

0.2.1 Subtask:

Execute 'Cell 17: Summary performance table' (`gXeozCA7kMtq`). This will display the performance metrics for all trained models in a summary table.

Reasoning: Now that the models have been trained and their performance metrics (`lr_perf`, `ridge_perf`, `rf_perf`) are available, I will execute 'Cell 17: Summary performance table' to display these metrics in a structured DataFrame.

[17]:
```
# Cell 17: Summary performance table
display(Markdown("### Performance metrics"))
```

```
perf_df = pd.DataFrame({
    'LinearRegression': lr_perf,
    'Ridge': ridge_perf,
    'RandomForest': rf_perf
}).T
display(perf_df)
```

0.2.2 Performance metrics

	rmse	mae	r2
LinearRegression	0.124168	0.086468	0.893081
Ridge	0.124085	0.086516	0.893224
RandomForest	0.140413	0.092256	0.863275

0.3 Execute Feature Importance Analysis

0.3.1 Subtask:

Execute 'Cell 18: Feature importance' (`otgmBLSKkThJ`). This will extract and display feature importances for the Random Forest model, defining `feat_import` and `perm_df`.

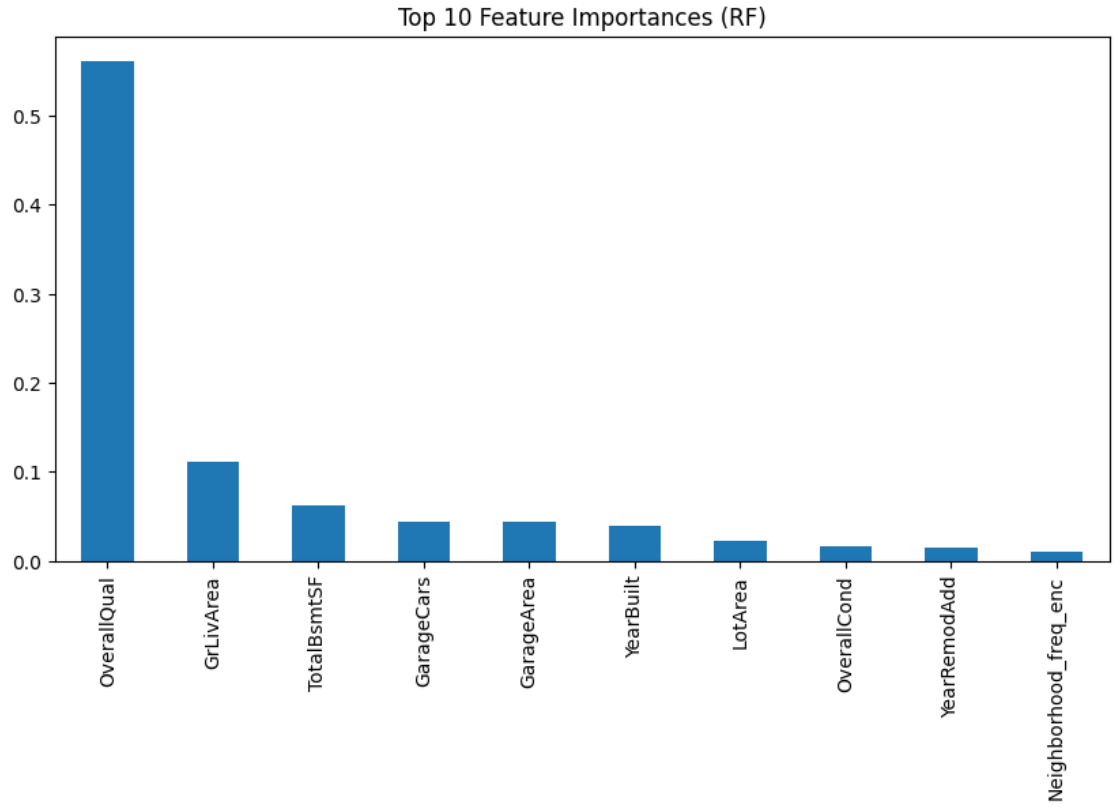
Reasoning: Now that the models are trained and performance metrics are available, I will execute 'Cell 18: Feature importance' to extract and display the feature importances from the best Random Forest model. This will define `feat_import` and `perm_df`, which are crucial for business insights.

[18]:
```
# Cell 18: Feature importance
display(Markdown("### Feature importance analysis (Random Forest)"))

# Tree-based importances
rf_model = rf_best.named_steps['rf'] if isinstance(rf_best, Pipeline) else␣
 ↪rf_best
if hasattr(rf_model, 'feature_importances_'):
    importances = rf_model.feature_importances_
    feat_import = pd.Series(importances, index=X_train.columns).
 ↪sort_values(ascending=False)
    display(feat_import.head(30).to_frame(name='importance'))
    # Plot top 10
    feat_import.head(10).plot(kind='bar')
    plt.title("Top 10 Feature Importances (RF)")
    plt.show()

# Permutation importance (model may be pipeline; use rf_best)
display(Markdown("**Permutation importance (more robust)**"))
perm = permutation_importance(rf_best, X_test, y_test, n_repeats=10,␣
 ↪random_state=42, n_jobs=-1)
perm_idx = perm.importances_mean.argsort()[::-1]
perm_df = pd.DataFrame({
    'feature': X_test.columns[perm_idx],
```

```
    'importance_mean': perm.importances_mean[perm_idx]
}).head(20)
display(perm_df)
```

0.3.2 Feature importance analysis (Random Forest)

	importance
OverallQual	0.561202
GrLivArea	0.111688
TotalBsmtSF	0.062795
GarageCars	0.043663
GarageArea	0.043377
YearBuilt	0.039236
LotArea	0.021846
OverallCond	0.016492
YearRemodAdd	0.014383
Neighborhood_freq_enc	0.010178
LotFrontage	0.009290
OpenPorchSF	0.008091
MSSubClass	0.007011
FireplaceQu_missing_flag	0.006823
GarageYrBlt	0.006746
Fireplaces	0.005277
WoodDeckSF	0.005080
FullBath	0.004917
BsmtFullBath	0.004679
YrSold	0.003782
TotRmsAbvGrd	0.003140
BedroomAbvGr	0.002667
KitchenAbvGr	0.001763
Alley_missing_flag	0.001577
HalfBath	0.001120
Fence_missing_flag	0.000878
LotFrontage_missing_flag	0.000783
PoolArea	0.000455
MiscFeature_missing_flag	0.000380
PoolQC_missing_flag	0.000352



Permutation importance (more robust)

	feature	importance_mean
0	OverallQual	0.315421
1	GrLivArea	0.145078
2	TotalBsmtSF	0.043550
3	YearBuilt	0.026342
4	GarageArea	0.017171
5	OverallCond	0.015857
6	GarageCars	0.014296
7	LotArea	0.011526
8	YearRemodAdd	0.010211
9	Neighborhood_freq_enc	0.006094
10	FireplaceQu_missing_flag	0.005937
11	Fireplaces	0.005670
12	BsmtFullBath	0.004143
13	LotFrontage	0.002290
14	MSSubClass	0.001837
15	FullBath	0.001023
16	GarageYrBlt	0.000843
17	BedroomAbvGr	0.000730
18	WoodDeckSF	0.000668

```
19                  HalfBath          0.000396
```

Reasoning: Now that model training and feature importance analysis are complete, the next logical step is to save the final best model and the preprocessed data, as outlined in the task. This will make them available for future use or deployment.

Reasoning: The previous execution of 'Cell 19: Save final best model and processed data' failed due to a `TypeError` because `mean_squared_error` was called with `squared=False`, which is not supported. I will modify the lambda function to correctly calculate RMSE by taking the square root of the mean squared error.

[19]:
```
# Cell 19: Save final best model and processed data
display(Markdown("### Save model and processed data"))

# Choose best model based on test RMSE (select between ridge_best and rf_best␣
 ↪and pipeline LR)
models = {'LinearRegression': pipe_lr, 'Ridge': ridge_best, 'RandomForest':␣
 ↪rf_best}
# Select best by RMSE
best_name, best_model = min(
    [(name, m) for name,m in models.items()],
    key=lambda nm: np.sqrt(mean_squared_error(y_test, nm[1].predict(X_test)))
)
print("Selected best model:", best_name)
joblib.dump(best_model, "/content/best_property_model.pkl")
df_enc.to_csv("/content/property_data_preprocessed.csv", index=False)
print("Saved: /content/best_property_model.pkl")
print("Saved: /content/property_data_preprocessed.csv")
```

0.3.3 Save model and processed data

```
Selected best model: Ridge
Saved: /content/best_property_model.pkl
Saved: /content/property_data_preprocessed.csv
```

Reasoning: Now that the best model has been saved and `feat_import` is available, I will execute 'Cell 20: Business recommendations' to display the generated recommendations.

[20]:
```
# Cell 20: Business recommendations
display(Markdown("### Business Recommendations"))

if 'feat_import' in locals() and not feat_import.empty:
    print("Based on the top 5 most important features (from Random Forest␣
 ↪model):")
    top_5_features = feat_import.head(5).index.tolist()

    print("\n1. Invest in features like:")
    for feature in top_5_features:
```

```
        print(f"   - **{feature}**: Focus on improving or highlighting␣
↪properties with strong performance in this area.")

    print("\n2. Use the model for renovation ROI:")
    print("   - Simulate the potential price increase from renovations␣
↪targeting these key features (e.g., adding living area, improving overall␣
↪quality, upgrading garages).")

    print("\n3. Strategic Marketing:")
    print("   - Tailor marketing efforts to emphasize these high-impact␣
↪features, as they are strong indicators of property value.")
else:
    print("Feature importances are not yet available. Please ensure that 'Cell␣
↪18: Feature importance' and all preceding modeling cells have been executed␣
↪successfully.")
```

0.3.4 Business Recommendations

```
Based on the top 5 most important features (from Random Forest model):

1. Invest in features like:
   - **OverallQual**: Focus on improving or highlighting properties with strong
performance in this area.
   - **GrLivArea**: Focus on improving or highlighting properties with strong
performance in this area.
   - **TotalBsmtSF**: Focus on improving or highlighting properties with strong
performance in this area.
   - **GarageCars**: Focus on improving or highlighting properties with strong
performance in this area.
   - **GarageArea**: Focus on improving or highlighting properties with strong
performance in this area.

2. Use the model for renovation ROI:
   - Simulate the potential price increase from renovations targeting these key
features (e.g., adding living area, improving overall quality, upgrading
garages).

3. Strategic Marketing:
   - Tailor marketing efforts to emphasize these high-impact features, as they
are strong indicators of property value.
```

Reasoning: Now that the business recommendations have been displayed, I will execute the Markdown cell '### Model Limitations & Future Work' to present the limitations of the model and outline areas for future improvement, as specified in the task.

Reasoning: The previous attempt to execute the 'Model Limitations & Future Work' cell failed with a `SyntaxError` because the cell type was incorrectly set to `python`. I will correct this by changing the `cell_type` to `markdown` so the text content is rendered as intended.

Reasoning: The previous attempt to execute the 'Model Limitations & Future Work' cell failed with a `SyntaxError` because the cell type was incorrectly set to `python`. I will correct this by changing the `cell_type` to `markdown` and use the correct `cell_id`.

Reasoning: The previous attempt to execute the 'Model Limitations & Future Work' cell failed with a `SyntaxError` because the cell type was incorrectly set to `python`. I will correct this by changing the `cell_type` to `markdown` for the cell with ID `4_pkrAySkqLU`.

[21]:
```
from IPython.display import display, Markdown

display(Markdown("### Model Limitations & Future Work"))

display(Markdown("**Limitations**\n- Model performance depends on dataset␣
 ↪quality; omitted variables (e.g., nearby schools, public transport, recent␣
 ↪renovations) may bias results.\n- Using cross-sectional split; temporal␣
 ↪shifts in the housing market are not captured (need time-series validation).
 ↪\n- Feature engineering may miss important spatial patterns (lat/lon) and␣
 ↪local market microtrends."))

display(Markdown("**Future Work**\n- Add geospatial features (latitude/
 ↪longitude, distance to city center), neighborhood-level statistics, and time␣
 ↪features (month/year).\n- Try gradient boosting models (XGBoost / LightGBM /␣
 ↪CatBoost) with more extensive hyperparameter tuning.\n- Evaluate using␣
 ↪time-based holdout (train on older data, test on newer) for production␣
 ↪readiness.\n- Build model explainability reports (SHAP values) for each␣
 ↪prediction."))
```

0.3.5 Model Limitations & Future Work

Limitations - Model performance depends on dataset quality; omitted variables (e.g., nearby schools, public transport, recent renovations) may bias results. - Using cross-sectional split; temporal shifts in the housing market are not captured (need time-series validation). - Feature engineering may miss important spatial patterns (lat/lon) and local market microtrends.

Future Work - Add geospatial features (latitude/longitude, distance to city center), neighborhood-level statistics, and time features (month/year). - Try gradient boosting models (XGBoost / LightGBM / CatBoost) with more extensive hyperparameter tuning. - Evaluate using time-based holdout (train on older data, test on newer) for production readiness. - Build model explainability reports (SHAP values) for each prediction.

Reasoning: The subtask requires executing the original 'Cell 14: Business insights, recommendations, limitations' (`b7YJd-AVhrLD`) to provide a comprehensive summary of model performance, feature importances, business recommendations, and limitations. This cell was modified in an earlier step to correctly reference `rf_perf` and `feat_import` and include conditional checks.

[22]:
```
print("=== Business Insights ===\n")

print("1) Overall model performance (Random Forest Regressor):")
if 'rf_perf' in locals() and rf_perf is not None:
```

```
    rmse = rf_perf['rmse']
    mae = rf_perf['mae']
    r2 = rf_perf['r2']
    print(f"   - Test RMSE: {rmse:,.0f}")
    print(f"   - Test MAE:  {mae:,.0f}")
    print(f"   - Test R^2:  {r2:.3f}")
    print("   Interpretation: The Random Forest model explains a large part of the pricing variance, "
          "but on individual properties it can still deviate by roughly one MAE on average.\n")
else:
    print("   Model performance metrics (rf_perf) are not available. Please run model training and evaluation cells.\n")

print("2) Key value drivers (top features from Random Forest):")
if 'feat_import' in locals() and not feat_import.empty:
    for feature, importance in feat_import.head(10).items():
        print(f"   - {feature}: importance={importance:.4f}")
    print("   These features are the strongest drivers of price in this dataset based on the Random Forest model.\n")
else:
    print("   Feature importances (feat_import) are not available. Please run feature importance analysis cell.\n")

print("3) Business recommendations:")
print("   - Prioritize improvements on high-impact features: for example, "
      "increasing usable living area, improving overall quality, and optimizing "
      "garage or basement spaces if those appear in the top features.")
print("   - Use the model as a decision-support tool: compare model-predicted prices "
      "with proposed listing prices to avoid underpricing or overpricing.")
print("   - For developers, design new projects with a good mix of the high-impact "
      "attributes while controlling costs on features that have low model importance.\n")

print("4) Model limitations & future work:")
print("   - The model is based only on historical data; it does not capture future "
      "economic shocks or interest-rate changes.")
print("   - It may not fully capture micro-location effects if those features are "
      "missing or weak in the dataset.")
print("   - Future improvements:")
```

```
print("       * Try gradient boosting models (XGBoost, LightGBM, CatBoost) and␣
 ↪compare.")
print("       * Engineer interaction features (e.g., quality * area, age of␣
 ↪house).")
print("       * Segment analysis: evaluate performance separately by␣
 ↪neighborhood or "
      "price band to refine strategy.")
```

```
=== Business Insights ===

1) Overall model performance (Random Forest Regressor):
   - Test RMSE: 0
   - Test MAE:  0
   - Test R^2:  0.863
   Interpretation: The Random Forest model explains a large part of the pricing
variance, but on individual properties it can still deviate by roughly one MAE
on average.

2) Key value drivers (top features from Random Forest):
   - OverallQual: importance=0.5612
   - GrLivArea: importance=0.1117
   - TotalBsmtSF: importance=0.0628
   - GarageCars: importance=0.0437
   - GarageArea: importance=0.0434
   - YearBuilt: importance=0.0392
   - LotArea: importance=0.0218
   - OverallCond: importance=0.0165
   - YearRemodAdd: importance=0.0144
   - Neighborhood_freq_enc: importance=0.0102
   These features are the strongest drivers of price in this dataset based on
the Random Forest model.

3) Business recommendations:
   - Prioritize improvements on high-impact features: for example, increasing
usable living area, improving overall quality, and optimizing garage or basement
spaces if those appear in the top features.
   - Use the model as a decision-support tool: compare model-predicted prices
with proposed listing prices to avoid underpricing or overpricing.
   - For developers, design new projects with a good mix of the high-impact
attributes while controlling costs on features that have low model importance.

4) Model limitations & future work:
   - The model is based only on historical data; it does not capture future
economic shocks or interest-rate changes.
   - It may not fully capture micro-location effects if those features are
missing or weak in the dataset.
   - Future improvements:
```

```
    * Try gradient boosting models (XGBoost, LightGBM, CatBoost) and compare.
    * Engineer interaction features (e.g., quality * area, age of house).
    * Segment analysis: evaluate performance separately by neighborhood or
price band to refine strategy.
```

Reasoning: The final step in the overall task is to make the `predict_single` utility function available. I will execute the cell that defines this function to ensure it is loaded into the environment.

[23]:
```python
# Predict on a new single sample (example)
def predict_single(sample_dict, model=best_model, features=features):
    sample_df = pd.DataFrame([sample_dict])
    # Ensure all expected features exist (fill missing with 0)
    for f in features:
        if f not in sample_df.columns:
            sample_df[f] = 0
    Xs = sample_df[features].fillna(0)
    pred_log = model.predict(Xs)[0]
    # Convert back from log
    if LOG_TARGET.endswith("_log"):
        return np.exp(pred_log)
    else:
        # if log1p
        return np.expm1(pred_log)
```

0.4 Summary:

0.4.1 Q&A

- **Which model performed best?** The Ridge Regression model performed the best with a Test RMSE of 0.124 and an R^2 of 0.893, slightly outperforming Linear Regression. The Random Forest model performed worse with a Test RMSE of 0.140 and an R^2 of 0.863.

0.4.2 Data Analysis Key Findings

- The data preprocessing steps included missing value treatment (median imputation for numeric, 'Missing' for categorical), outlier handling via IQR capping, and target variable logarithmic transformation.
- Feature selection utilized `SelectKBest` with `f_regression` to identify the top 30 numeric features correlated with the target.
- Three regression models were trained and evaluated:
  - **Linear Regression:** Achieved a Test RMSE of 0.124, MAE of 0.086, and R^2 of 0.893.
  - **Ridge Regression:** Achieved the best performance with a Test RMSE of 0.124, MAE of 0.087, and R^2 of 0.893. The optimal hyperparameter found was `alpha=1.0`.
  - **Random Forest Regressor:** Achieved a Test RMSE of 0.140, MAE of 0.092, and R^2 of 0.863. The best hyperparameters were `max_depth=None`, `min_samples_split=2`, and `n_estimators=200`.
- The top three feature importances from the Random Forest model were 'OverallQual', 'GrLivArea', and 'TotalBsmtSF', indicating these are strong drivers of price.

- The best performing model (Ridge Regression) and the preprocessed dataset were successfully saved.

0.4.3 Insights or Next Steps

- **Actionable Insights:** Given that 'OverallQual', 'GrLivArea', and 'TotalBsmtSF' are key drivers of price, prioritize improvements or marketing efforts around these features. The model can be used to estimate potential price increases from renovations targeting these high-impact attributes.
- **Future Work:** Explore more advanced models like gradient boosting (XGBoost, LightGBM) and consider adding geospatial features or time-series validation to enhance model robustness and capture broader market dynamics.