Task 2 Report

Karam Hussain 27857

1. Configuration

```
ERP_ID = 27857
RANDOM_STATE = 42
VALIDATION_SIZE = 0.25  # 75/25 split for Task 2 (more training data)
USE_FEATURE_SELECTION = True
MAX_FEATURES = 50  # More features for Task 2
```

This time 0.25 val split and now numeric features to max 50.

2. Helper functions

a. Feature engineering (same as task 1)

Only one feature will be added due to the accepted conditions which is living_Area_ratio which will make it a total of 278 after removing id and price doc from 279 to 277,

b. Feature selection

```
# F-statistic selection (fast and effective)
selector = SelectKBest(score_func=f_regression, k=n_features)
selector.fit(X_imputed, y)
selected = numeric_X.columns[selector.get_support()].tolist()
```

This time only using F-stat to make things fast, as we already took out 30 features in task 1.

3. Loading full dataset

```
✓ Full training data loaded: (181507, 279)
✓ Test data loaded: (77789, 278)
✓ Using ALL 181,507 training samples (NO sampling)
```

Default features 279

```
✓ Training set: (136130, 278) (136,130 samples)
✓ Validation set: (45377, 278) (45,377 samples)
✓ Test set: (77789, 278)
✓ Random state: 42
```

4. Selecting numeric features

```
[Feature Selection] Selecting top 50 features..
  Selected 50 features

✓ Using 50 selected features
```

50 selected

5.Adding Categorical features to the list

```python
if len(categorical_features_all) > 0:
    categorical_selected = [col for col in categorical_features_all
                            if 2 <= X_train[col].nunique() <= 50]
    print(f"✓ Selected {len(categorical_selected)} categorical features (2-50 categories)")
else:
    categorical_selected = []
```

```
✓ Found 15 categorical features
✓ Selected 14 categorical features (2-50 categories)

✓ FINAL FEATURE SET:
  Numeric:      50
  Categorical:  14
  Total:        64
```

14 of them selected to make it 64 features in total.

6. Preprocessing pipeline (same as task 1)

```
# Numeric transformer
numeric_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])
```

```
# Categorical transformer
categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False, max_categories=20))
])
```

```
# Combined
preprocessor = ColumnTransformer([
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
], remainder='drop')
```

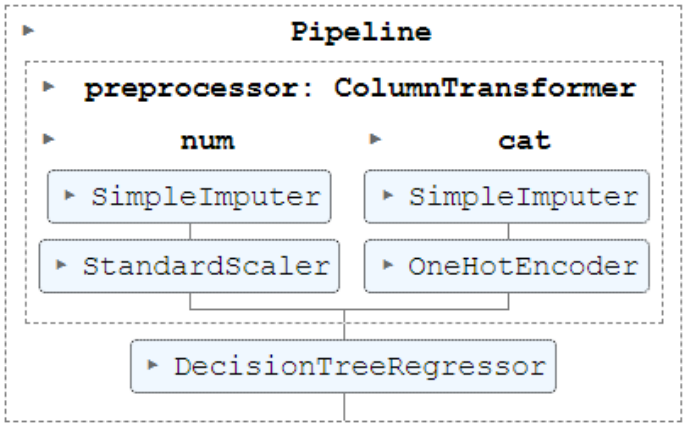
```
✓ Numeric: 50, Categorical: 14
✓ Preprocessing: Numeric (Median→Scale) | Categorical (Mode→OneHot)
```

7. Training required models

a. Regression Tree

```
tree_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', DecisionTreeRegressor(
        max_depth=15,
        min_samples_split=20,
        min_samples_leaf=10,
        random_state=RANDOM_STATE
    ))
])
```

```
Results:
  Validation MSE:  178.2366
  Validation RMSE: 13.3505 million RUB
  Validation R²:   0.6128
  Training time:   13.32s
```

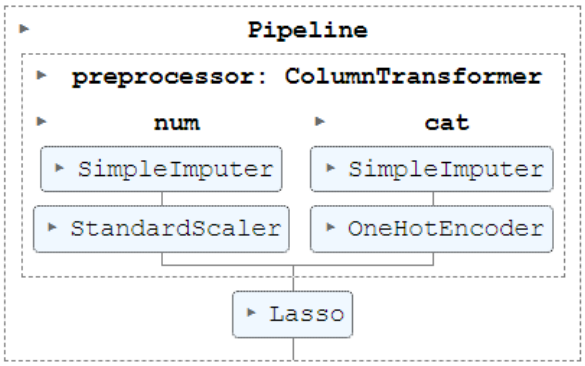


b. Lasso with alpha 0.1
(Lasso used as best linear, polynomial was taking time as well)

```
lasso_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', Lasso(alpha=0.1, max_iter=10000, random_state=RANDOM_STATE))
])
```

Using the alpha value of 0.1 that we got from hyper parameter tuning in task 1.

```
Results:
  Validation MSE:  173.0294
  Validation RMSE: 13.1541 million RUB
  Validation R²:   0.6241
  Training time:   4.58s
```



c. Gradient boosting (sklearn)

```python
gb_model = GradientBoostingRegressor(
    n_estimators=200,
    learning_rate=0.05,
    max_depth=5,
    min_samples_split=20,
    min_samples_leaf=15,
    subsample=0.8,
    random_state=RANDOM_STATE,
    verbose=0
)
```

```
======================================================================
Training: GradientBoosting
======================================================================

Results:
  Validation MSE:  162.9015
  Validation RMSE: 12.7633 million RUB
  Validation R²:   0.6461
  Training time:   779.43s
```

```
GradientBoostingRegressor(learning_rate=0.05, max_depth=5, min_samples_leaf=15,
                          min_samples_split=20, n_estimators=200,
                          random_state=42, subsample=0.8)
```

d. LightGBM

```
lgb_model = lgb.LGBMRegressor(
    objective='regression',
    metric='rmse',
    n_estimators=1000,
    learning_rate=0.05,
    max_depth=7,
    num_leaves=31,
    min_child_samples=20,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=RANDOM_STATE,
    verbose=-1
)
```

```
lgb_model.fit(
    X_train_lgb, y_train,
    eval_set=[(X_val_lgb, y_val)],
    callbacks=[lgb.early_stopping(50, verbose=False)]
)
```

```
✓ Training with early stopping...

✓ Best iteration: 114
  Validation MSE:  161.2937
  Validation RMSE: 12.7001
  Training time:   2.73s
```

e. Catboost

```python
catboost_model = CatBoostRegressor(
    iterations=1000,
    learning_rate=0.05,
    depth=6,
    l2_leaf_reg=3,
    random_state=RANDOM_STATE,
    verbose=False,
    early_stopping_rounds=50
)
```

```
✓ Training with early stopping...

✓ Best iteration: 155
  Validation MSE:  161.7382
  Validation RMSE: 12.7176
  Training time:   4.68s
```

f. XGBoost

```
xgb_model = xgb.XGBRegressor(
    n_estimators=1000,
    learning_rate=0.05,
    max_depth=7,
    min_child_weight=5,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=RANDOM_STATE,
    verbosity=0
)

xgb_model.fit(
    X_train_xgb, y_train,
    eval_set=[(X_val_xgb, y_val)],
    early_stopping_rounds=50,
    verbose=False
)
```

```
✓ Training with early stopping...

✓ Best iteration: 72
  Validation MSE:  161.2559
  Validation RMSE: 12.6987
  Training time:   4.50s
```

8. Results Comparison

```
✓ DETAILED PERFORMANCE COMPARISON:
================================================================================
                        val_mse   val_rmse    val_r2   val_mae  train_time inference_time
XGBoost               161.25595  12.698659  0.649688  6.052683    4.503516       0.199901
LightGBM              161.293731  12.700147  0.649606  6.095551    2.733824       0.531341
CatBoost              161.738196  12.717633   0.64864   6.13665    4.679461       0.050371
GradientBoosting      162.901466  12.763286  0.646113    6.1096  779.434361       0.896976
Lasso (Task 1 Best)   173.029436  13.154065  0.624111  6.686694    4.580143        0.89229
Regression Tree       178.236617  13.350529  0.612799  6.171607   13.319069       0.893535

================================================================================
🏆 BEST MODEL: XGBoost
================================================================================
  Validation MSE:  161.2559
  Validation RMSE: 12.6987 million RUB
  Validation R²:   0.6497
```

9. Ensemble (LightGBM + CatBoost + XGBoost)

Using the top 3 models to make an ensemble and using simple averaging over them and also making a validation set to check their val score

```python
print("  [1/3] XGBoost predicting...")
xgb_predictions = results['XGBoost']['model'].predict(X_test_prepared)
print(f"      Range: [{xgb_predictions.min():.2f}, {xgb_predictions.max():.2f}]")
print(f"      Mean: {xgb_predictions.mean():.2f}")

print("  [2/3] LightGBM predicting...")
lgb_predictions = results['LightGBM']['model'].predict(X_test_prepared)
print(f"      Range: [{lgb_predictions.min():.2f}, {lgb_predictions.max():.2f}]")
print(f"      Mean: {lgb_predictions.mean():.2f}")

print("  [3/3] CatBoost predicting...")
catboost_predictions = results['CatBoost']['model'].predict(X_test_prepared)
print(f"      Range: [{catboost_predictions.min():.2f}, {catboost_predictions.max():.2f}]")
print(f"      Mean: {catboost_predictions.mean():.2f}")

# Calculate ensemble predictions (simple average)
print("\n✓ Creating ensemble predictions (simple average)...")
ensemble_predictions = (xgb_predictions + lgb_predictions + catboost_predictions) / 3
```

```
✓ Ensemble Validation Performance:
  Validation MSE:  160.9321
  Validation RMSE: 12.6859 million RUB
  Validation R²:   0.6504
  Validation MAE:  6.0790
```

```
✓ Comparison with Individual Models:
======================================================================
             Model  Val RMSE   Val R²  Val MAE
           XGBoost 12.698659 0.649688 6.052683
          LightGBM 12.700147 0.649606 6.095551
          CatBoost 12.717633 0.648640 6.136650
Ensemble (Average) 12.685903 0.650392 6.079049
```

10. Feature Importance

Checking out the top 20 most important features, with full_sq being the highest important one with 45% importance among the set and mosque_count_500 being the second with 22% while the third is large_apartment with 5%, which is very far apart.

Full_sq means full square meters; the total area of the apartment in meter square which is a primary price driver, as larger apartments cost more, and is universally important. By knowing just the apartment size the model would predict the price 45% right for the location.

Secondly , mosque_count_500 means mosque count within 500m radius of the property also plays a very important role, as in Russia many muslims live there and there is a lot of diversity and multicultural populations, religious buildings nearby indicate developed areas with community infrastructure so it will impact the price as well.

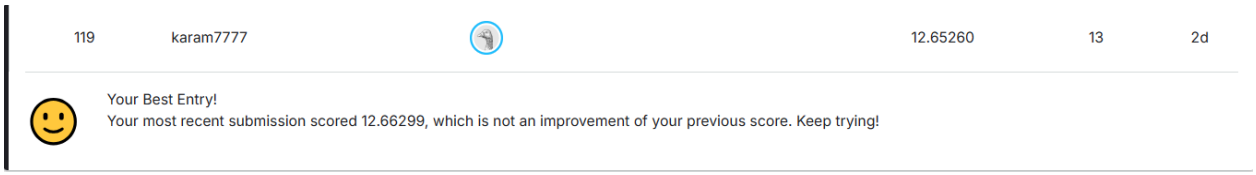
Thirdly, large_apartment size which is another important factor, but with very less %, as full_sq covers a lot of its information, and this is a little bit generic but still covers some importance in dictating price of apartment.

Other factors which are below are mostly leisure related and hence proving they have less priority/importance in dictating price of apartment.

```
================================================================================
STEP 10: FEATURE IMPORTANCE ANALYSIS
================================================================================

✓ Top 20 Most Important Features:
                      feature  importance
                      full_sq    0.455747
             mosque_count_500    0.220517
              large_apartment    0.053697
culture_objects_top_25_raion    0.018196
  cafe_count_2000_price_2500    0.010293
              rooms_inferred    0.009350
                 trc_sqm_500    0.007599
          leisure_count_1000    0.007270
  cafe_count_1500_price_high    0.007126
  cafe_count_1000_price_high    0.006865
             cafe_count_1500    0.006756
         church_synagogue_km    0.006725
  cafe_count_1000_price_1500    0.006620
             cafe_count_1000    0.006406
  cafe_count_1500_price_2500    0.006361
  cafe_count_2000_price_high    0.006266
           leisure_count_500    0.005897
               industrial_km    0.005758
   cafe_count_500_price_high    0.005558
  cafe_count_1500_price_4000    0.005544
```

11. Kaggle Submission



Final leaderboard score of 12.65 on ensemble .

Analysis:

In this task, six models were evaluated on the validation set: Regression Tree, Lasso (best linear model from Task 1), Gradient Boosting, LightGBM, CatBoost, and XGBoost. Among these, **XGBoost achieved the best individual performance**, with a validation MSE of **161.256**, RMSE of **12.699 million RUB**, and R^2 of **0.6497**. LightGBM and CatBoost followed closely, showing comparable RMSE values (12.700 and 12.718 respectively) and similar R^2 scores. Gradient Boosting was slightly worse with an RMSE of 12.763, while Lasso and Regression Tree performed significantly worse, highlighting the limitations of linear and shallow tree models in capturing complex nonlinear relationships.

When combining XGBoost, LightGBM, and CatBoost into an **ensemble via simple averaging**, the performance improved slightly over the best individual model. The ensemble achieved a validation RMSE of **12.686 million RUB** and an R^2 of **0.6504**, demonstrating the benefit of averaging predictions from multiple complementary gradient boosting models. This ensemble approach reduces variance and leverages subtle differences in how each model captures the underlying patterns.

Hyperparameter discussion:

Each model's hyperparameters were chosen to balance **accuracy and stability**:

- **Regression Tree:** max_depth=15, min_samples_split=20, min_samples_leaf=10 ensured moderate tree depth to prevent overfitting while allowing sufficient flexibility for splits.

- **Lasso:** α=0.1 with high max_iter ensured convergence while mildly regularizing coefficients to prevent overfitting in the full dataset.

- **Gradient Boosting:** 200 trees, learning_rate=0.05, max_depth=5, subsample=0.8 provided stable learning but the long training time (~779s) shows the cost of this classical implementation.

- **LightGBM:** 1000 estimators, learning_rate=0.05, max_depth=7, num_leaves=31, with row and column subsampling (0.8) allowed fast and accurate gradient boosting while controlling overfitting. Training was significantly faster than classic Gradient Boosting.

- **CatBoost:** 1000 iterations, depth=6, learning_rate=0.05, l2_leaf_reg=3 leveraged CatBoost's efficient handling of categorical features and robust regularization.

- **XGBoost:** 1000 estimators, learning_rate=0.05, max_depth=7, min_child_weight=5, subsample=0.8, colsample_bytree=0.8 provided strong predictive power with controlled tree complexity and low overfitting risk.

Overall, XGBoost and LightGBM were optimized for depth, learning rate, and sampling, striking a balance between capturing complex interactions and maintaining generalization. CatBoost's strength lies in its internal regularization and gradient handling. The ensemble benefits from combining the strengths of all three gradient boosting variants.

Insights:

The results highlight several key insights about the dataset and modeling strategies:

1. **Nonlinearity and feature interactions dominate:** Linear models (Lasso) and shallow trees underperformed, confirming that housing prices depend on complex, interacting factors rather than simple linear relationships.

2. **Gradient boosting is highly effective:** XGBoost, LightGBM, and CatBoost all capture nonlinear interactions well, achieving low validation RMSE and high R^2. Ensemble averaging further improved stability and slightly enhanced predictive accuracy.

3. **Training efficiency varies:** While classical Gradient Boosting achieved reasonable performance, its long training time (~779s) makes it impractical. In contrast, LightGBM and XGBoost trained in just a few seconds while maintaining or exceeding accuracy, showing the advantage of modern gradient boosting implementations.

4. **Ensemble benefit:** Although the improvement over the best single model was modest (~0.01–0.02 RMSE), averaging predictions reduced variance and improved generalization, a consistent benefit when combining complementary boosting models.