Project Report

Project Statement and Objectives

The goal of this project is to predict house prices based on data obtained in a real dataset. The app allows users to explore dataset information through an EDA page with statistics and plots, train regression machine learning models interactively by choosing which features of the dataset and what regression model to use between four available and make predictions using such trained models.

Data Analysis Approach and Findings

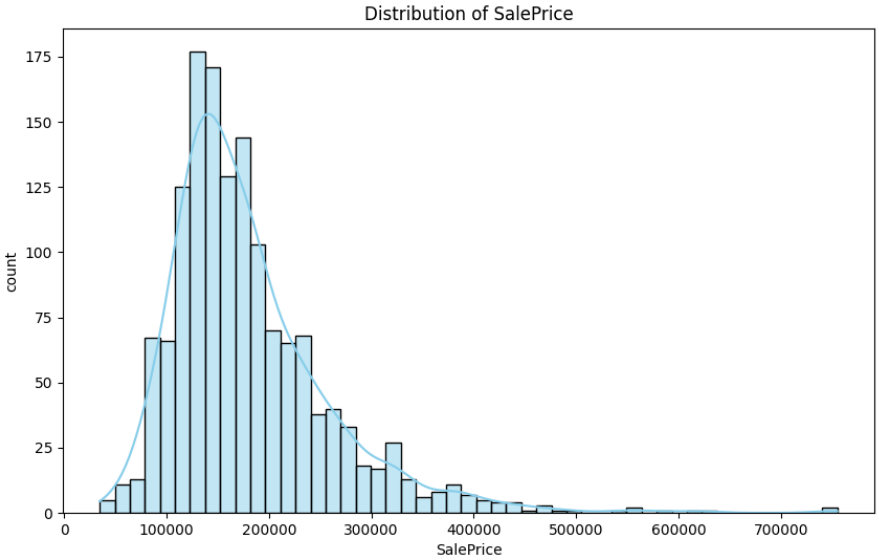
Data are taken in the Kaggle Ames House Pricing Dataset, reachable by the following link:

- https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques

The aim of this EDA section Is to extract useful information from the dataset.

Generalities

First, the used dataset is composed of 79 features (plus an ID and the target variable), representing different aspects of a house, such as square feet dimensions, presence or absence of amenities (pool, garage, masonry and so on) with the related quality and condition, neighborhood, years of construction and/or renovation and overall quality indicators. The target variable (that is, the variable we are trying to predict with trained models) is SalePrice, indicating the price of the house. The following plot shows its distribution:



As expected, the distribution has a Gaussian trend with a mean value between 100000 and 200000, showing how most houses have not very high prices.

Missing Values

The dataset has some features possibly containing NA values, particularly those regarding the presence of different kinds of structures. The following table shows all such features with the percentage of NA values compared to the total rows:

Features	NA Count	NA Percentage (%)
Pool Quality	1453	99,5
Other Structures	1406	96,3
Alley	1369	93,8
Fence	1179	80,8
Masonry Type	872	59,7
Fireplace Quality	690	47,2
Lot Frontage	259	17,7
Garage Type	81	5,5
Garage Construction Year	81	5,5
Finished Garage	81	5,5
Garage Quality	81	5,5
Garage Condition	81	5,5
Basement Exposure	38	2,6
Second Basement Type	38	2,6
Basement Quality	37	2,5
Basement Condition	37	2,5
First Basement Type	37	2,5
Masonry Area	8	0,5
Electrical	1	0,07

From the documentation we can understand how most of these NA values refer to the absence of the features and not the lack of information about it: this also shows how most houses have not expensive structures like pools (0,5%), alleys (6,2%), fences (19,2%) or similar (3,7%); on the other hand, we can see that most houses have a garage (94,5%) and a basement (97,5%).

Also, most of these features are categorical ones, with the only exceptions of Lot Frontage, Masonry Area and Garage Construction Year.

Exceptions to the NA values rule are Lot Frontage and Electrical: the first one is a feature every house will surely have, so NA values mean lack of information, whereas the second one present NA value in just one sample, so we can omit it.

Next analysis will be carried out after having filled all NA values with the following rules:

- For categorical features in which NA means absence of feature, it will be replaced with a ‘None’ (string) value.

- For numeric features in which NA means absence of feature, it will be replaced with a ‘0’ value.
- In Lot Frontage feature NA values will be replaced with the median Lot Frontage value between all the non-NA rows (this is done because almost a fifth of the Lot Frontage rows have NA, and dropping all those would come with too high a cost).
- Electrical feature has just one sample in which NA is found, so we can drop it without introducing bias.

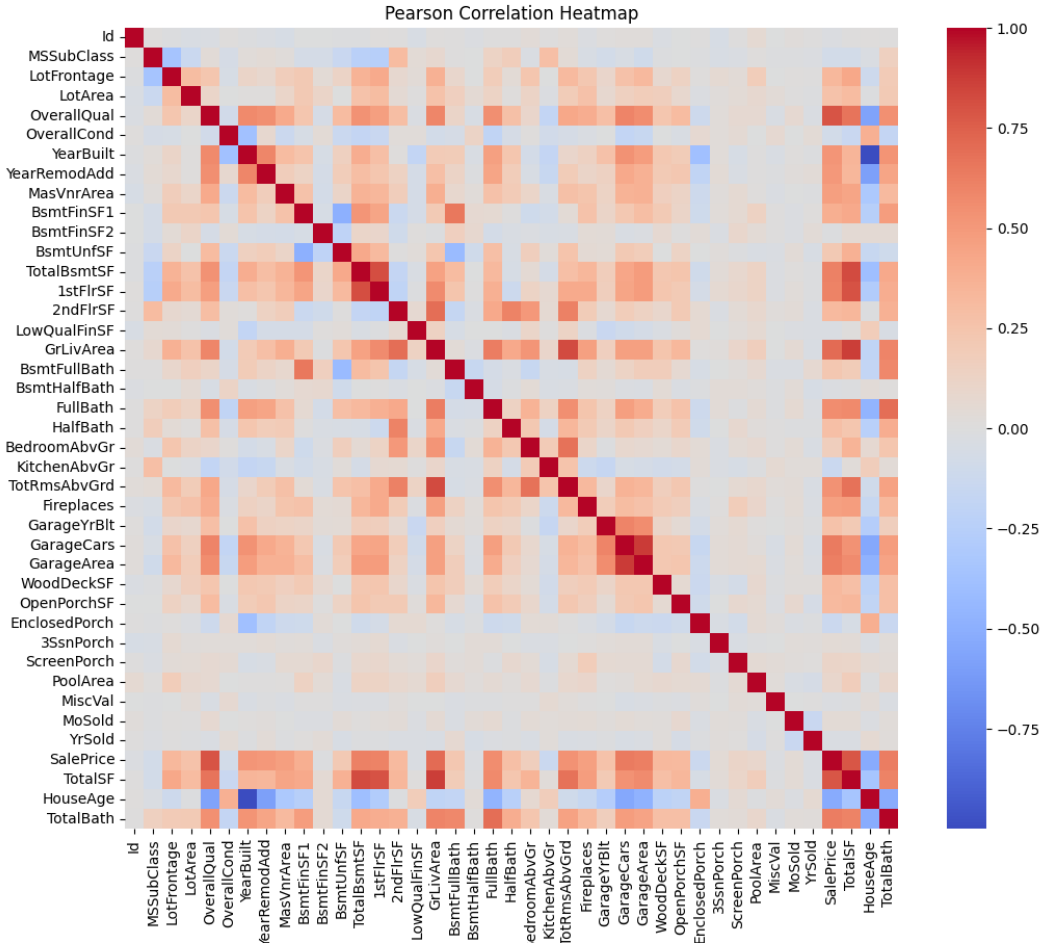
Derived Features

It might be useful to create some derived features to enrich the comprehension of the dataset trends. Some interesting ones may be the following:

- House Total Square Feet: the house’s comprehensive dimension in square feet, calculated as the sum of Total Basement (first and second) Square Feet, First Floor Square Feet and Second Floor Square Feet.
- House Age from the moment it was sold, calculated as the difference between the year the house was sold and year it was built.
- Total Number of Bath, calculated as the sum of all full baths and of half of all half baths.

Correlations

The following matrix shows the Pearson linear correlations between dataset numeric features (derived features are shown too):



First, we can notice that the derived features (as expected) have a high linear correlation with the target variable; in particular, House Total Square Feet has the highest correlation of all features, and house age has negative correlation. Between non-derived features, the highest correlation features are, between numeric ones, the following:

- Overall Quality
- Living Area
- Garage Area
- Total Basement Area
- Year Built
- Number of Full Baths

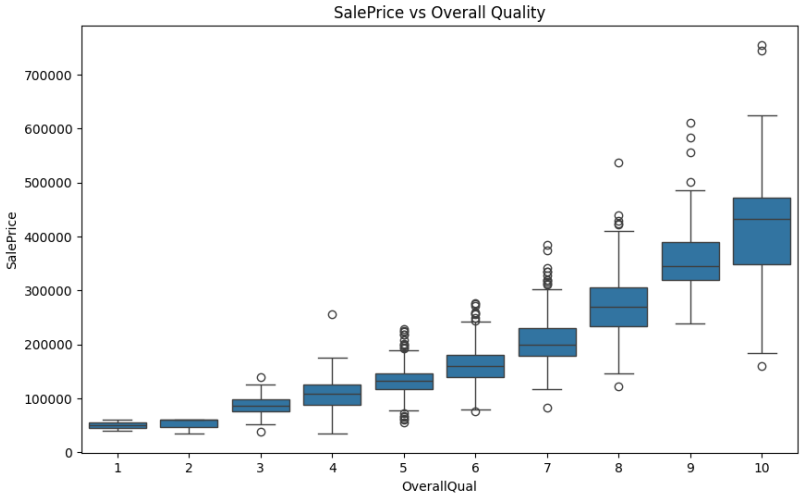
Among non-derived categorical features, highest correlations are found, predictably, with:

- Neighborhood
- Exterior Quality
- Kitchen Quality
- Basement Quality
- Sale Condition
- House Style

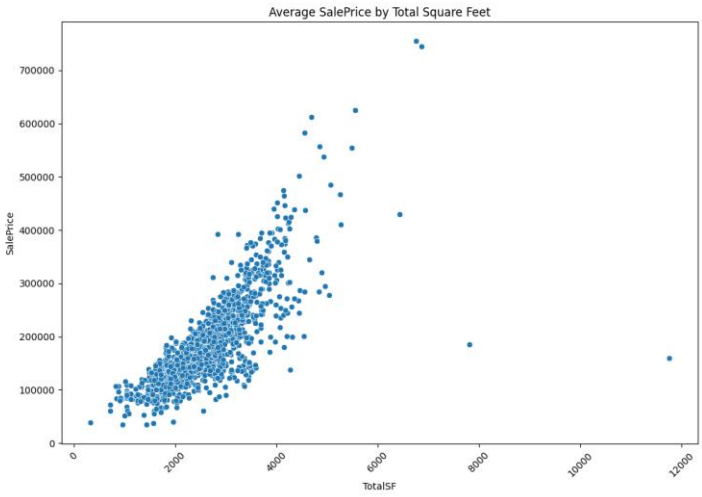
Plots

Here are shown plots of some of the above features to further show the correlations with the target variable (other plots can be found in the application's EDA section):

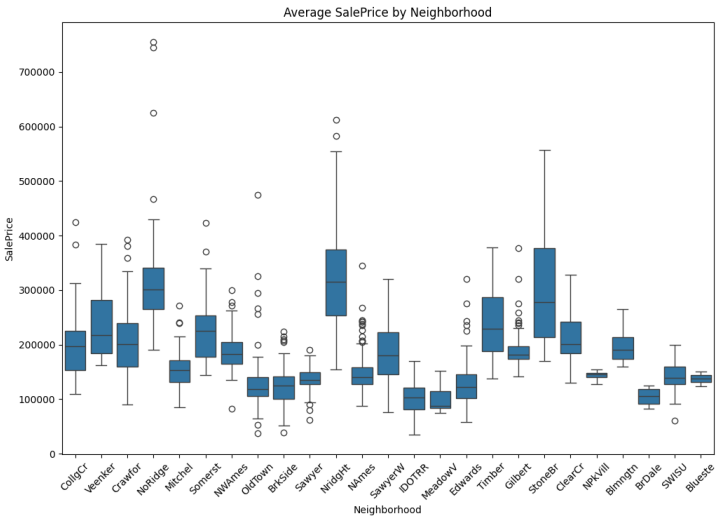
Overall Quality



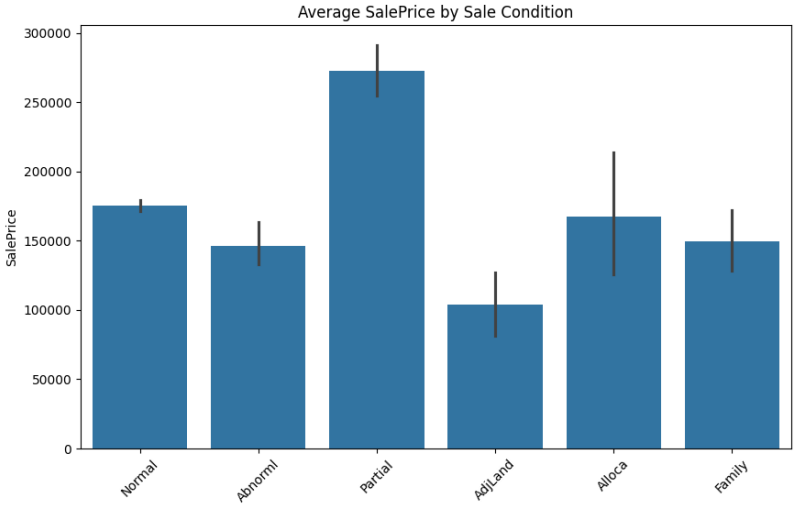
Total Square Feet



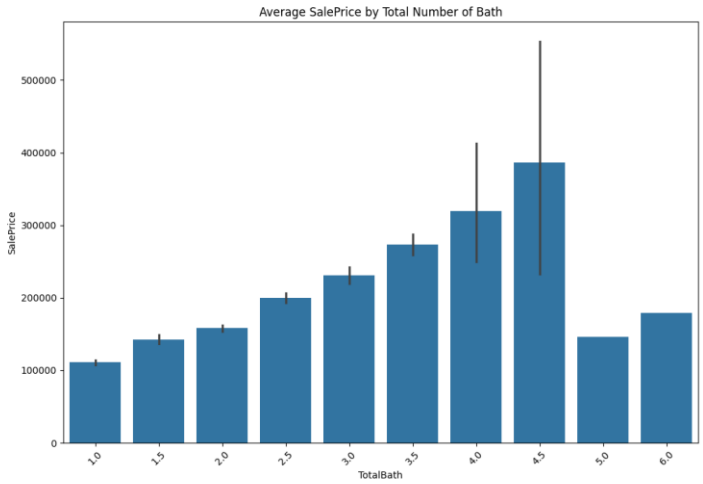
Neighborhood



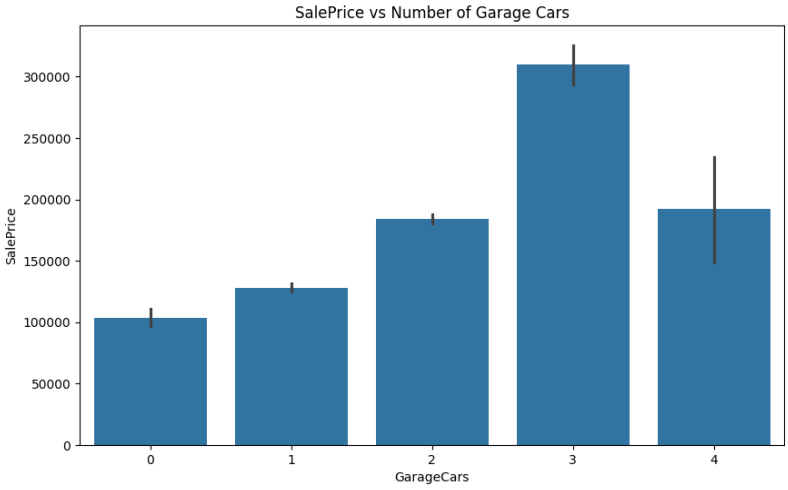
Sale Condition



Total Number of Baths



Number of Garage Cars



Insights

Looking at the above plots and the previously explained statistics, we can make some considerations:

- Structures like pool, fence or similar (although rare) tend to increase the price more than basic structures like garage or basement.
- Quality is a bit more important than size: although dimensions are, of course, a relevant parameter to increase the price, the quality of amenities and the general quality has a slightly stronger impact on it.

- Neighborhood is a strong indicator (but not absolute) of whether a house will have a high price or not (as expected).
- There are some special sale conditions that pose themselves as exceptions; for example, big houses sold still unfinished have a low price compared to their sheer dimensions. Houses sold under normal conditions tend to have higher prices.
- Other outliers are big yet poor-quality houses.

Machine Learning Methodology and Results

Methodology

The aim of the project being to predict house prices based on data, the problem is one of regression. The application gives users the possibility to choose features to train a ML model on; this is done because the dataset possesses a lot of features, and not all users may have available all of them: this gives the possibility to perform predictions interactively.

Users can choose from four different regression models to do the training:

- Linear Regressor
- Gradient Descent Regressor
- Random Forest Regressor
- Gradient Boosting Regressor

After users send selected features with related values, data are split into a training set (80) and a test set (20), to later evaluate the performance of the model.

Successively, before training, preprocessing is applied to the user selected features to perform better training. All the preprocessing is applied through a “Pipeline” object: this way, every time training or prediction is performed, the actions are automatically executed on all the selected features, without having to do it manually.

The applied preprocessing to NA values is the same seen in EDA section (although not used in predictions, due to users having to insert all values before proceeding) and is implemented with the “SimpleImputer” object. For numeric features, in linear and gradient descent models, a scaling is applied with the “StandardScaling” object (tree-based models do not need to scale features), whereas for categorical features, in all models, feature encoding is done with the “OneHotEncoder” object. All these actions are then effectively applied to the related columns with the “ColumnTransformer” object and, finally, all the transformations are put into another “Pipeline”, ready to perform now training or prediction on given data.

After training the model is evaluated using two metrics:

- RMSE (Root-Mean-Squared-Error)
- R^2 (Coefficient)

Results

Here are shown evaluation results of models trained with the four different regressor on the complete set of features and on a set of 22 high correlated features:

- Linear Regressor with all features:
  RMSE: 65277.26
  R^2: 0.44
  The low results are due to the presence of strong non-linear relationships and of strong correlation with categorical features.
- Linear Regressor with h-c features:
  RMSE: 32987.23
  R^2: 0.85
  Selecting just high correlated features (especially features with high linear correlation) leads to much better results.

- Gradient Descent Regressor with all features:
  RMSE: 30878.29
  R^2: 0.87
- Gradient Descent Regressor with h-c features:
  RMSE: 32702.34
  R^2: 0.86
  Being a regularized linear model, it is a lot more stable than Linear Regressor, with good performance in both trainings.

- Random Forest Regressor with all features:
  RMSE: 28855.86
  R^2: 0.89
- Random Forest Regressor with h-c features:
  RMSE: 30695.02
  R^2: 0.87
  Tree-based model that benefits from feature diversity and is good with both trainings.

- Gradient Boosting Regressor with all features:
  RMSE: 28204.30
  R^2: 0.89
- Gradient Boosting Regressor with h-c features:
  RMSE: 26757.47
  R^2: 0.90
  Tree-based model that is overall the best (although not very different in results from RF).

Technical Implementation Details

Project Structure

```
HousePricePrediction/

├── app.py                      # Main Flask application
├── dataset_manager.py          # Dataset handling, preprocessing, and EDA
├── training_manager.py         # Model training and management
├── eda.py                      # File for EDA operations
├── models/                     # Storage for trained models and metadata
│   ├── models.json             # JSON file tracking trained models
│   └── *.pkl                   # Serialized trained model files
├── static/                     # Static assets for the frontend
│   ├── css/                    # CSS stylesheets
│   │   └── *.css               # Custom styling (Bootstrap-inspired)
│   ├── js/                     # Frontend JS scripts for dynamic behavior
│   │   └── *.js                # JS files
│   └── eda/                    # Generated plots from EDA
│       └── *.png               # Plot images
├── templates/                  # HTML templates for Flask routes
│   └── *.html                  # Application's pages
├── data/                       # Folder to store the dataset
│   └── dataset.csv             # Dataset file
├── Dockerfile                  # Configuration for building the container
├── docker-compose.yml          # Docker Compose file
├── requirements.txt            # Python dependencies
├── README.md                   # Project information
├── .gitignore                  # Gitignore file
└── report/                     # Folder for documentation
    └── report.pdf              # Final report
```

Implementation

Client-side, the application is composed of 5 html pages, each provided with a CSS stylesheet to create a Bootstrap style; three of the pages have also a related JavaScript file to collect input data and send them to the server through AJAX messages.

Server-side is located all the ML and EDA code. The main structure is realized using Flask as server. EDA is done with libraries such as Pandas, NumPy, Matplotlib and Seaborn. ML uses Scikit-learn library (see ML Methodology for precise objects used).

Dataset is stored in a CSV file and imported when the application starts running. Models, once trained, are stored as '.pkl' files, serialized with the Joblib library and loaded with the same library when a prediction is to be made. Json library is used to save models' metadata in the 'models.json' file.

The environment is managed by a Dockerfile that install all requirement (from the '.txt' file); also, a docker-compose file is present, and it search for the first free port starting from the port 5000 (until port 5100) to run the application on.

To build and run the only requirements is to have Docker and docker compose (latest version), then run "docker compose up --build" to start the application and "ctrl + c" to stop it.

Challenges Encountered and Solutions

- Applying multiple preprocessing actions to data before training and predicting maintaining consistency -> Resolved using Pipelines to automate the preprocessing.
- Saving trained model to use them later for predictions -> Resolved using Joblib to serialize the model in '.pkl' files and load it on need; also, saving model metadata in a Json file.
- Handling docker exposed port in case a fixed port is occupied -> Resolved using dynamic port selection in docker-compose file.
- Applying preprocessing to dataset rows with NA values -> Resolved studying the nature of NA values for each feature using documentation and filling them consequently.

Possible Future Improvements

- Add new regression models to further extend user possibilities.
- Add dataset selection on training page to allow users to train models based on different datasets.
- Add CSV file input to perform multiple predictions at once.
- Add user authentication to save different models for different users.