AA1 PROJECT

Santiago Salom Torrens

2024

Index

About the project

This project is carried out in GCED's AA1 subject. The goal of this project is to apply the techniques seen during the course to a real-world dataset. All aspects of the modeling methodology studied during the course from preprocessing to generating predictive models will be applied and covered.

This project has been done in python programming language, via Jupyter Notebook. All code and plots included in this report have been extracted from there.

Introduction and main goals

The dataset where I will be working is named "Miami housing" and its author is Sebastian Fischer. This dataset contains information about 13932 houses sold in Miami, United States.

Link: https://www.openml.org/search?type=data&status=active&id=44983

What we want to predict with this dataset is the Sale Price of a house. To do this, the dataset contains 16 predictor variables (not all of them will be used as it will be seen later) which are the following:

- PARCELNO: Unique identifier for each house. Approximately 1% appears multiple times (the same house may have been sold on more than one occasion).
- LND_SQFOOT: Total land surface of the property in square feet.
- TOTLVGAREA: Total constructed area of the house in square feet.
- SPECFEATVAL: Value of special attributes (for example pool) in dollars.
- RAIL_DIST: Distance to the nearest train tracks (is an indicator of noise) in feet.
- OCEAN_DIST: Distance to the ocean in feet.
- WATER_DIST: Distance to the nearest body of water (eg lakes) in feet.
- CNTR_DIST: Distance in feet to the financial center of Miami.
- SUBCNTR_DI: Distance in feet to the nearest subcenter.
- HWY_DIST: Distance to the nearest freeway (is an indicator of noise) in feet.
- age: Age of the house.
- avno60plus: Binary variable that takes the value 1 if an acceptable level of aircraft noise is exceeded and 0 otherwise.
- structure_quality: Variable that takes values between 1 and 5 depending on the state of the house.
- month_sold: Variable that indicates the month of sale of the house, being January 1, February 2...
- LATITUDE: Variable indicating the latitude coordinate.
- LONGITUDE: Variable that indicates the longitude coordinate.

The main goal of this project is to train different models in order to predict the Sale Price of a house, based on the different predictor variables. The objective is to find the model that has

the best prediction capacity on unseen data, as we want to be able to produce forecasts and accurate predictions.

My personal goals are to better understand the whole process of predicting data, from preprocessing to modeling, until a solid model is obtained.

Preprocessing

Datasets found in the internet can not be directly used to fit models as certain standards are required to ensure a decent performance.
In this part of the report I'll be explaining how I have transformed the original dataset into a useful and solid dataframe capable of being trained and of making good predictions.

Before going into more detail, several libraries have been imported in order to be able to use some predefined functions and to get access to some other helpful tools. The libraries are located in the work annex at the end of this report.

A seed has been used to ensure code is reproducible (random.seed(7), np.random.seed(42)).

Feature extraction

A useful dataset needs to contain all those variables helpful to make predictions, while those that aren't essential need to be removed so that no unnecessary complexity is added.
In this case, I've dropped PARCELNO, as it's an identifier and it isn't useful to predict sale price, and LATITUDE and LONGITUDE, as their influence is really hard to appreciate, specially when all houses are located within the same city.

Now the dataset looks something like this (not all variables can be displayed):

	SALE_PRC	LND_SQFOOT	TOT_LVG_AREA	SPEC_FEAT_VAL	RAIL_DIST	OCEAN_DIST	WATER_DIST	CNTR_DIST	SUBCNTR_DI	HWY_DIST	
count	1.393e+04	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	1393
mean	3.999e+05	8620.880	2058.045	9562.493	8348.100	31690.548	11959.837	68489.879	41114.596	7723.320	3
std	3.172e+05	6070.089	813.539	13890.968	6178.029	17595.078	11932.985	32008.471	22161.821	6068.939	2
min	7.200e+04	1248.000	854.000	0.000	10.000	236.000	0.000	3825.000	1462.000	90.000	
25%	2.350e+05	5400.000	1470.000	810.000	3299.000	18079.250	2675.750	42822.500	23995.750	2997.750	1
50%	3.100e+05	7500.000	1877.500	2765.500	7105.500	28541.500	6922.000	65852.000	41109.500	6159.000	2
75%	4.280e+05	9126.250	2471.000	12352.250	12102.000	44309.750	19199.750	89357.750	53949.000	10853.500	4
max	2.650e+06	57064.000	6287.000	175020.000	29621.000	75744.000	50399.000	159976.000	110553.000	48167.000	9

Dealing with missing values

All models that will be trained and later used to make predictions work with datasets that don't contain missing values. Missing values are cells in the dataset that aren't defined, or that have abnormal numbers to represent an unknown value.

If we call the function value_counts() for each variable, neither missing values nor abnormal numbers are observed, so it looks like we've been lucky.

However, as one of the goals is to learn, I'll add some missing values randomly, just to have to deal with them. I'll introduce missing values at random in columns OCEAN_DIST, CNTR_DIST and STRUCTURE_QUALITY with a missing proportion of 1%:

```
df.STRUCTURE_QUALITY.isna().value_counts()
df.OCEAN_DIST.isna().value_counts()
df.CNTR_DIST.isna().value_counts()
```

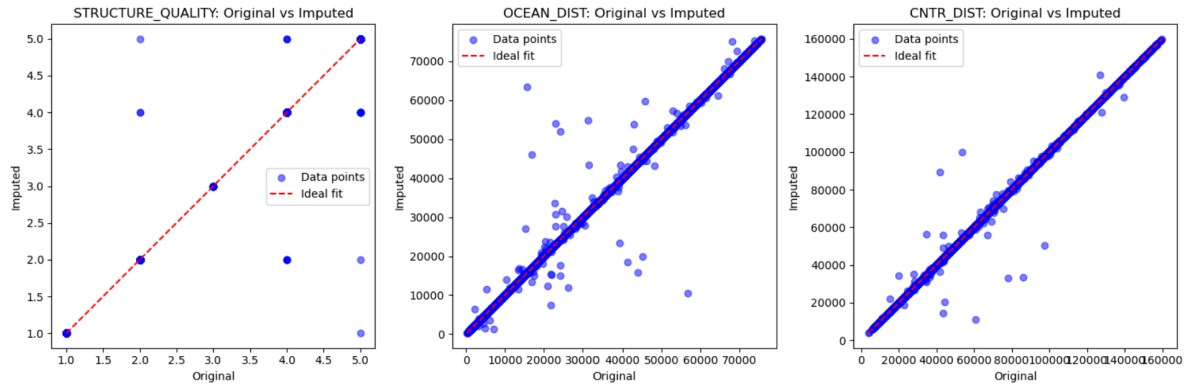
```
STRUCTURE_QUALITY
False    13793
True       139
Name: count, dtype: int64

OCEAN_DIST
False    13794
True       138
Name: count, dtype: int64

CNTR_DIST
False    13795
True       137
Name: count, dtype: int64
```

As it can be seen, these three variables now have missing values scattered throughout the different rows.

Directly eliminating those rows that contain the NAs isn't a good option as a lot of information would be lost. Instead I'll impute them (imputting a missing value means to infer it from the known part of the data). In the case of the numerical variables (OCEAN_DIST and CNTR_DIST), KNN Regressor will be applied, while KNN Classifier will be used for the categorical variable. KNN stands for K Nearest Neighbours, and is a Regressor (or Classifier) used to predict or classify a point based on proximity.
Here's an scatter plot of the old against new columns to visualize the impact of the imputations:

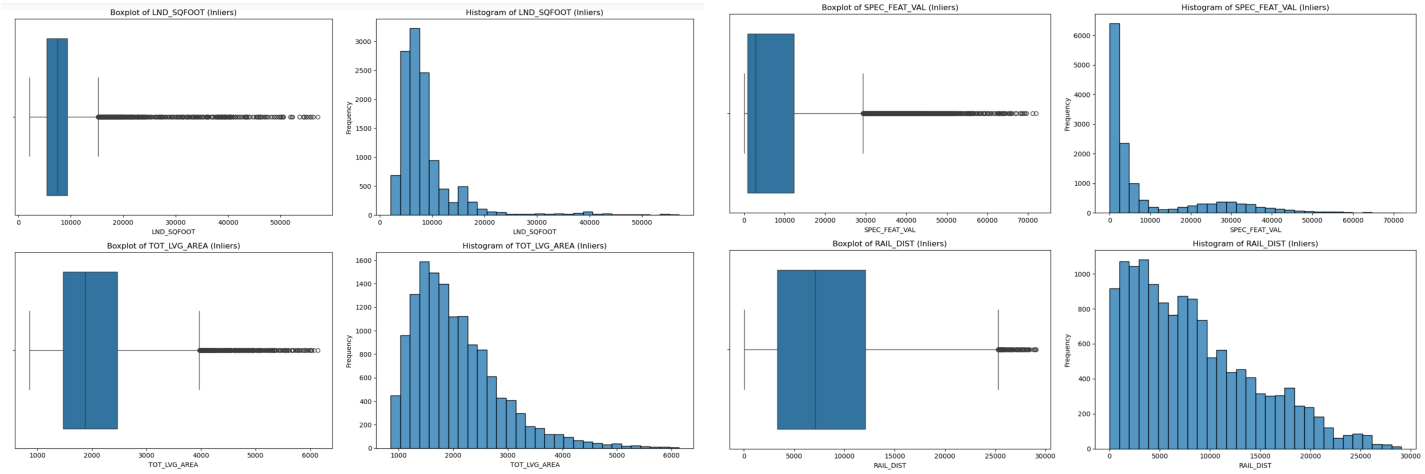


As it can be seen, in STRUCTURE_QUALITY, there are some points that differ from the ideal fit, but there's nothing to worry about, as there are almost 14000 observations and it's reasonable to think that some of them have been misclassified.
On the other hand, the numerical variables show really good results as a big percentage of the points are correctly placed.

Outliers

When doing some modeling, it's important to have a dataset that doesn't contain any observation known as an outlier. This observations present some extreme values, and can be a potential danger for the quality of the model, as they can have a big influence on

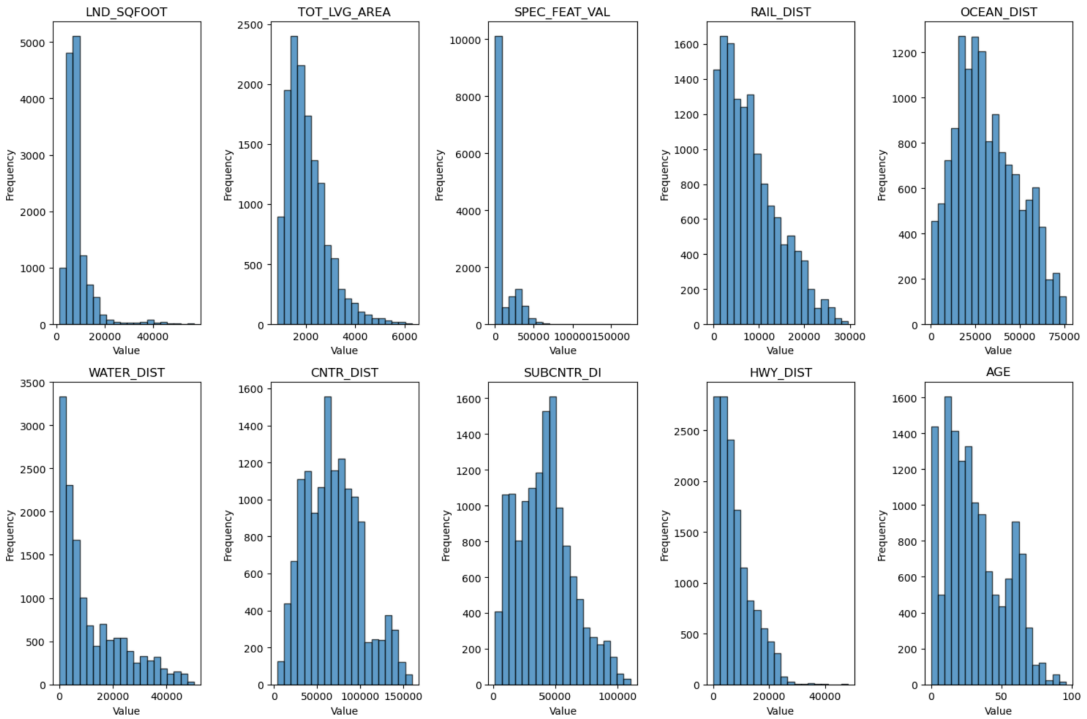
metrics like the mean or standard deviation, and can lead to further issues like overfitting the data.
To check for outliers, the histograms of the main variables have been plotted and examined. Here are some of them:



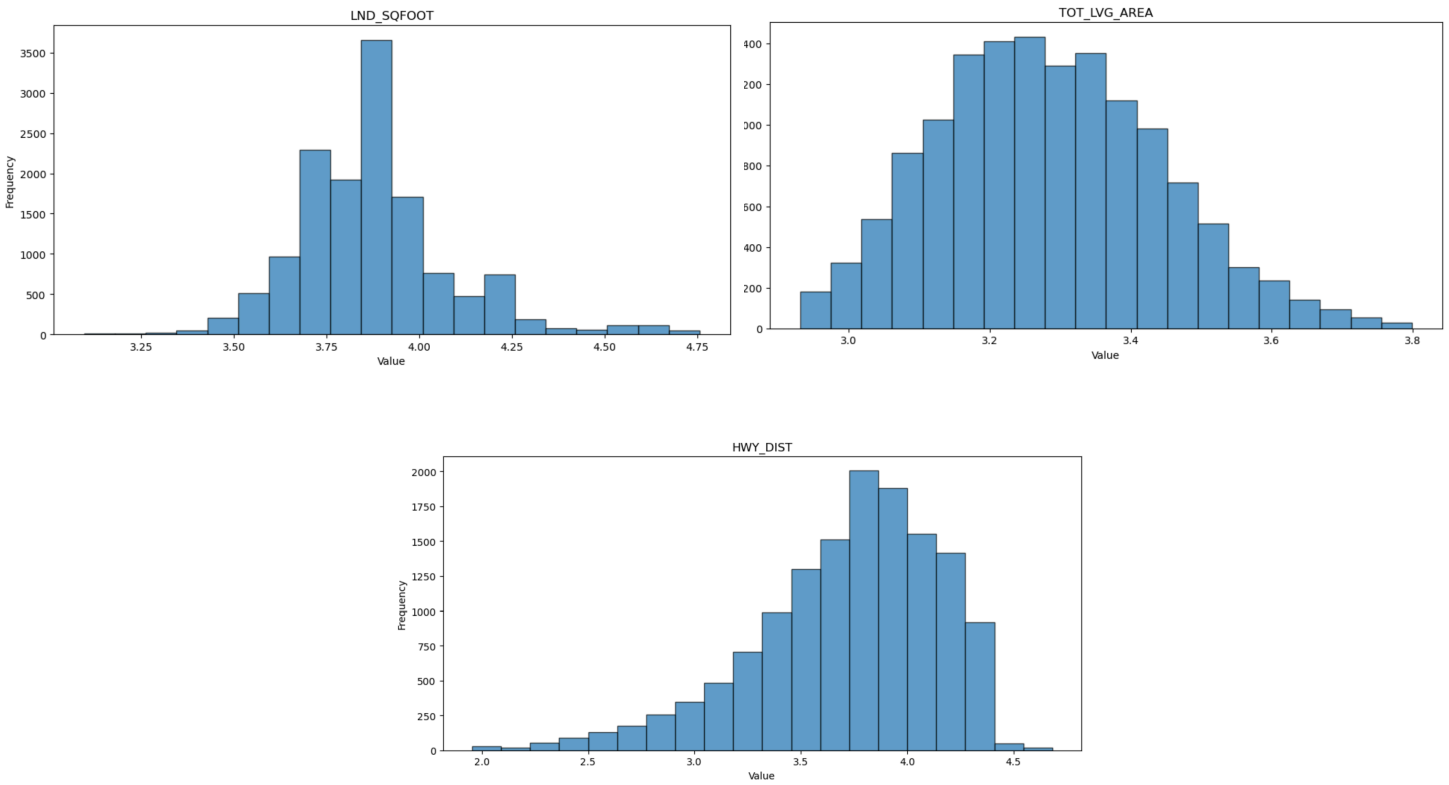
After reviewing all the histograms and boxplots, I haven't found any observation that I would consider an outlier, so no treatment is necessary.

Gaussianity

Gaussianity in the data is fundamental in machine learning as many methods like linear regression or KNN assume that features follow a normal distribution. To check how each variable is distributed, histograms have been plotted:



From what we can see, LND_SQFOOT, TOT_LVG_AREA and HWY_DIST seem to need a logarithmic transformation to look more gaussian. Here's how they look after the transformation:



As we can see, they all look much better, specially LND_SQFOOT and TOT_LVG_AREA, so the logarithmic transformation was well applied.

Normalization

Normalization (scaling) is essential in data analysis and machine learning, as it ensures that all features contribute equally, prevents numerical problems, and helps in the interpretability and visualization of data.
Basically, what is done is transforming all numerical data, so they fall within a certain range (in this case the range is [0,1]).
The transformation that I will apply is Min-Max Scaling.
Here's the data before and after the scaling (notice "min" and "max" values):

Before:

	LND_SQFOOT	TOT_LVG_AREA	SPEC_FEAT_VAL	RAIL_DIST	OCEAN_DIST	WATER_DIST	CNTR_DIST	SUBCNTR_DI	HWY_DIST	AGE
count	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000
mean	3.875	3.283	9562.493	8348.100	31685.715	11959.837	68496.619	41114.596	3.721	30.669
std	0.210	0.159	13890.968	6178.029	17583.666	11932.985	32000.577	22161.821	0.434	21.153
min	3.096	2.931	0.000	10.000	236.000	0.000	3825.000	1462.000	1.954	0.000
25%	3.732	3.167	810.000	3299.000	18096.750	2675.750	42848.500	23995.750	3.477	14.000
50%	3.875	3.274	2765.500	7105.500	28545.000	6922.000	65854.000	41109.500	3.790	26.000
75%	3.960	3.393	12352.250	12102.000	44298.000	19199.750	89345.750	53949.000	4.036	46.000
max	4.756	3.798	175020.000	29621.000	75744.000	50399.000	159976.000	110553.000	4.683	96.000

After:

	SALE_PRC	LND_SQFOOT	TOT_LVG_AREA	SPEC_FEAT_VAL	RAIL_DIST	OCEAN_DIST	WATER_DIST	CNTR_DIST	SUBCNTR_DI	HWY_DIST	
count	1.393e+04	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	13932.000	1393
mean	3.999e+05	0.469	0.406	0.055	0.282	0.417	0.237	0.414	0.363	0.647	
std	3.172e+05	0.127	0.183	0.079	0.209	0.233	0.237	0.205	0.203	0.159	
min	7.200e+04	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
25%	2.350e+05	0.383	0.272	0.005	0.111	0.237	0.053	0.250	0.207	0.558	
50%	3.100e+05	0.469	0.395	0.016	0.240	0.375	0.137	0.397	0.363	0.673	
75%	4.280e+05	0.520	0.532	0.071	0.408	0.584	0.381	0.548	0.481	0.763	
max	2.650e+06	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	

As we can observe, now all numerical variables fall within the same range, so we will surely be getting better results than if we hadn't performed this operation. Now there will not be any variable with more weight and influence than the rest

Dealing with Categorical variables

As I'll explain later, this dataset has a numerical variable as target, so I'll be fitting Regression Models. This type of model only accepts numerical variables as predictor variables, so something needs to be done with the current categorical variables.

The most common way to deal with them is to apply one-hot encoding, which basically creates as many new variables as there were initially categories and assigns each one 0 or 1 depending on whether the category was the one to which it originally belonged. I'll use the get_dummies function from pandas.
This procedure will be applied to the three categorical variables in our dataset (AVNO60PLUS, STRUCTURE_QUALITY and MONTH_SOLD):
Now the dataset looks like this:

```
AVNO60PLUS_soroll_acceptable          bool
AVNO60PLUS_soroll_no_acceptable       bool
STRUCTURE_QUALITY_1.0                 bool
STRUCTURE_QUALITY_2.0                 bool
STRUCTURE_QUALITY_3.0                 bool
STRUCTURE_QUALITY_4.0                 bool
STRUCTURE_QUALITY_5.0                 bool
MONTH_SOLD_abril                      bool
MONTH_SOLD_agost                      bool
MONTH_SOLD_desembre                   bool
MONTH_SOLD_febrer                     bool
MONTH_SOLD_gener                      bool
MONTH_SOLD_juliol                     bool
MONTH_SOLD_juny                       bool
MONTH_SOLD_maig                       bool
MONTH_SOLD_març                       bool
MONTH_SOLD_novembre                   bool
MONTH_SOLD_octubre                    bool
MONTH_SOLD_setembre                   bool
```

Modeling

Machine learning models are algorithms that are capable of recognizing patterns in data and making predictions. To do so, they need to be trained with a large dataset. We can commonly distinguish between two types of models, depending on the type of prediction needed to make:

- Classification problems. These types of problems have as a target a categorical variable, so the objective is to tell whether a certain point belongs or not to a certain category.
- Regression problems. The target in these problems is a numerical variable, so the goal is to predict the exact value a point takes.

As what I want to predict is the Sale Price of a house, my target variable is numerical and consequently, I have a Regression Problem.

Next I'll be showing all the models I've trained, with their corresponding metrics and my personal opinion on their performance.

Metrics

To assess the quality of a model, metrics need to be computed, so we can tell how well the model adapts to the data and how well it performs when compared to other models.

The best metric in Regression Model is R^2, which is a statistical measure that determines the proportion of variance in the target variable that can be explained by the predictor variables. In other words, R^2 shows how well the data fits the regression model (goodness of fit).

I've also computed in certain cases the MSE (Mean Square Error), specially to compare models, but on its own it's really hard to interpret and not a consistent metric

Data Splitting

When modeling in machine learning and data science, it's really important to divide the dataset into different subsets. The most common subsets are Training, Validation and Test. The Training set is used to train the model, the Validation set is used to tune hyperparameters and for giving us some assessment on how well the model adapts to unseen data, and finally, the Test set which is used to evaluate the final performance of the model on unseen data.
It's crucial that no data in one of the subsets is present in another subset, as the results shown wouldn't be legit.

As I have a lot of observations (13932), I've decided to divide my data into 60% training, 20% validation and 20% test.

Models

I've trained 7 different models, from less complex to more complex, in order to see which one is capable of making the better predictions. The models are Linear Regression, Linear Regression with Ridge Regularization, Linear Regression with Lasso Regularization, Decision Tree, Random Forest, Extra Trees and Voting Regressor.

Polynomial Regression

Polynomial Regression is an extension of Linear Regression where the relationship between the input features and the target variable is modeled as an n-th degree polynomial. Instead of fitting a straight line, this method fits a curve to the data. The goal is to minimize the difference between the actual and predicted values.

As my dataset is really large, I've only fitted polynomials of degrees 1 (linear regression) and 2. To do so I've used Sklearn's LinearRegression function, together with PolynomialFeatures to train polynomial regression models of different degrees. After trying first and second degree polynomials, and computing their metrics we observe the following R^2 for the training set (note that the first component of the vector is first degree polynomial and the second component for second degree polynomial) :

```
[0.698224496364116, 0.8745600096656174]
```

The R^2 for the validation set:

```
[0.6912932284998572, -9.492239266801368e+17]
```

The conclusions that we can extract are the following:

- Degree 1 has the worst training R2 but a similar validation R2. This implies that this model generalizes well, with close training and validation errors, and positive R-squared values. It's a good balance between bias and variance.

- Degree 2 has better training R2 but much worse validation R2. This implies that this model overfits the training data, as indicated by a much higher R2 in training, but a negative R2 in validation. The negative validation R-squared is a strong indication of overfitting. The negative R-squared value implies that the model performs worse than a horizontal line (a naive model predicting the mean of the target variable).

From what we see, polynomial regression with degree 1 has much better performance on unseen data, so it's the one I will stay with.

However, to get an even better understanding of how this model is performing, it is typically a good idea to apply Cross Validation. This technique evaluates the performance of a model on unseen data. It involves dividing the available data into multiple folds or subsets, using one of these folds as a validation set, and training the model on the remaining folds.

With 10 folds, the results are as follow:

	MSE	norm_MSE	R2
Fold 1	31308667750.009	0.294	0.706
Fold 2	33375913139.234	0.283	0.717
Fold 3	23976740393.288	0.289	0.711
Fold 4	35631133452.468	0.311	0.689
Fold 5	34590240248.736	0.346	0.654
Fold 6	37383936185.026	0.326	0.674
Fold 7	28440810205.285	0.305	0.695
Fold 8	27792018195.808	0.339	0.661
Fold 9	21067994807.459	0.249	0.751
Fold 10	33954417113.523	0.318	0.682
Mean	30752187149.084	0.306	0.694

We can see that this model has a mean validation R2 of around 69.4%, which although it's not awful, it could be much better.

To try to improve this linear regression model, I'll add some regularization.

Regularized Linear Regression

Regularized linear regression models are a type of linear regression that add a penalty to the size of the coefficients with the objective to improve performance and make it less likely to overfit.

In this project I've tried Ridge and Lasso regularizations.

Ridge Regularization

This regularization adds a penalty (λ - lambda) that larger the penalty, the more the coefficients are shrunk towards zero, making the model simpler and less likely to overfit.

To find the penalty (or hyperparameter) that gives the best results I've defined a vector of the most common lambdas and applied at the same time cross-validation, so for each lambda, a whole cross validation is performed.

With 5 folds and lambdas = [1e-10,1e-5,1e-4,1e-3,1e-2,0.1, 0.5,1,5,10,50,100], this has been the result:

	mean MSE	mean norm_MSE	mean R2
Lambda=0.1	30537339782.753	0.304	0.696
Lambda=0.01	30537637491.578	0.304	0.696
Lambda=0.001	30537678918.679	0.304	0.696
Lambda=0.0001	30537683180.192	0.304	0.696
Lambda=1e-05	30537683607.534	0.304	0.696
Lambda=1e-10	30537683655.029	0.304	0.696
Lambda=0.5	30538352791.142	0.304	0.696
Lambda=1	30544089633.262	0.305	0.695
Lambda=5	30673595029.995	0.306	0.694
Lambda=10	30884699495.298	0.308	0.692
Lambda=50	32361359833.892	0.322	0.678
Lambda=100	34187857474.539	0.34	0.66

As we can see, this regularization hasn't improved nearly anything about the models, so I'll try the other regularization.

Lasso Regularization

Lasso is almost identical to Ridge regularization, except for the fact it can shrink some coefficients to exactly zero. This is like performing feature selection by excluding less important features from the model.

Same strategy has been applied (5 folds and same lambdas) and this has been the best configuration:

```
LassoCV
LassoCV(alphas=[1e-10, 1e-05, 0.0001, 0.001, 0.01, 0.1, 0.5, 1, 5, 10, 50, 100],
        cv=5)
```

```
Best lambda: 10.0 R2 score: 0.6955219769574434
```

As we can see, there's not much improvement.

Conclusion

If we compare these three linear regression models. we have the following table:

	lr	ridge_cv	lasso_cv
CV R2	0.694	0.696	0.696
Train R2	0.697	0.698	0.698
lambda	0.000	0.100	10.000

As we can see, there's been really little improvement in the cross validation R-squared. This may be due to the fact that first degree polynomial regression wasn't overfitting much and thus adding regularization hasn't had much of an impact.

Still the best validation R^2 is significantly low (69,6%), so I will implement more complex models to see if performance is improved.

Ensembles

In the context of machine learning, ensembles are a set of methods that combine multiple learning algorithms to obtain a better predictive performance.

I've particularly implemented various ensembles, such as Decision Tree and Random Forest amongst others.

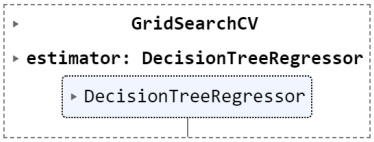
Decision Tree Regressor

Decision Tree Regressor is an ensemble and a machine learning technique that constructs a tree-like model to predict continuous numerical values. The model can be implemented from the Scikit Learn library as DecisionTreeRegressor. This function needs multiple hyperparameters that specify the desired maximum depth, the minimum samples required to be at a leaf node and many other characteristics. That's why I've decided to use the function GridSearchCV, that allows me to iterate through all the possible hyperparameters, to find the combination that gives the best possible R^2.

With the following hyperparameters tested,

```
param_grid = {
    'criterion': ['squared_error', 'absolute_error'],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 3, 5],
}
```

the best combination found is the following:




```
0:08:03.668042
Best parameters: {'criterion': 'squared_error', 'max_depth': None, 'min_samples_leaf': 5, 'min_samples_split': 2}
```

(Criterion : squared_error; this is the function used to measure the quality of a split)
(Max depth : None; means that nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples)

If we compute its metrics we obtain:

```
Training Mean Squared Error: 4632984153.71342
Training R^2 Score: 0.9538504027238367
Validation Mean Squared Error: 17775567757.46878
Validation R^2 Score: 0.8322918570847707
```

These metrics represent a huge improvement from the best previous model, as training R-square is now 95%, and most importantly, validation R-squared is now above 83%.

However, the model is still overfitting, so I'll implement other models that are better at preventing overfitting and work better with unseen data.

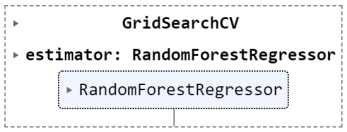
Random Forest Regressor

Random Forest is an ensemble method that builds multiple decision trees (usually trained on different sets of the data and with different subsets of features) and combines their predictions. The final prediction is often the average of the predictions from all the trees, which helps improve accuracy and reduce overfitting.

This model can also be implemented from Scikit Learn with the function RandomForestRegressor. As with the Decision Tree, it's necessary to do some hyperparameter tuning to find the combination that gives the best result. GridSearchCV has also been used here.
With this hyperparameters,

```
param_grid_rf = {
    'n_estimators': [50, 100],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 3, 5],
}
```

the best combination found has been:




```
0:20:56.205326
Best parameters for Random Forest: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 100}
```

And its metrics:

```
Random Forest Training Mean Squared Error: 1902412338.1972291
Random Forest Training R^2 Score: 0.981049889153919
Random Forest Validation Mean Squared Error: 10119679146.643074
Random Forest Validation R^2 Score: 0.9045232973856271
```

These results are really good and the model has improved a lot when comparing it to a single Decision Tree, as now validation R-squared sits at an outstanding 90%. Definitely, implementing a more complex model that tries multiple Decision Trees instead of just one, helps reduce variance and overfitting, leading to better performing models.

However, there's still room for improvement, as the model is still overfitting a little bit.

Extra Trees Regressor

Extra Trees is a similar model to Random Forest, but with a significant difference in how the trees are built. In Extra Trees, the splitting points are chosen even more randomly, which can lead to more diverse trees and, in some cases, better performance.

The model has been implemented as before and with the following hyperparameters:

```
param_grid_et = {
    'n_estimators': [150, None],
    'max_depth': [100, None],
    'min_samples_split': [4, 6],
    'min_samples_leaf': [2, 4],
}
```

The best combination of hyperparameters is

```
0:01:10.703305
Best parameters for Extra Trees: {'max_depth': 100, 'min_samples_leaf': 2, 'min_samples_split': 4, 'n_estimators': 150}
```

and has the following metrics:

```
Extra Trees Training Mean Squared Error: 874768035.8162951
Extra Trees Training R^2 Score: 0.9912863521169989
Extra Trees Validation Mean Squared Error: 9910652273.917093
Extra Trees Validation R^2 Score: 0.9064954149079787
```

As we can see, the performance has even improved a little bit from the Random Forest model, as it has an even higher R-squared, and an even lower MSE. Additionally, as it can be seen just above the best combination, this is also the fastest model of all the ensembles tried, which reflects just how good and efficient Extra Trees can be.

Voting Regressor

The last model I will try is Voting Regressor, which is an ensemble method that combines predictions from multiple different models. The final prediction is the average of all the predictions.

To implement this model, first we need to define which models are going to take part in computing this final prediction. If we use the following models,

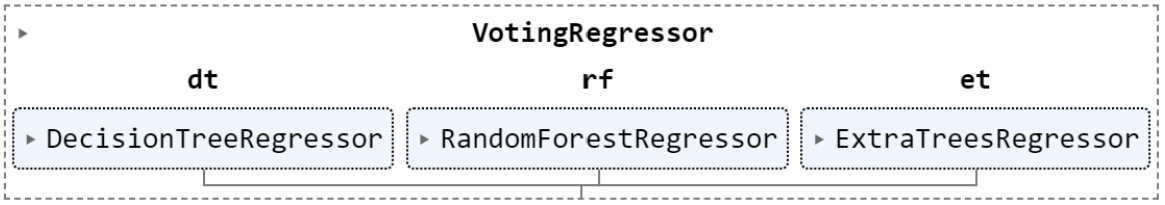
```
# Define individual regressors
reg1 = LinearRegression()
reg2 = DecisionTreeRegressor(random_state=42)
reg3 = RandomForestRegressor(random_state=42)
reg4 = ExtraTreesRegressor(random_state=42)
```

and fit the Voting Regressor, we obtain the following metrics:


```
Voting Regressor Training Mean Squared Error: 2432464942.7575006
Voting Regressor Training R^2 Score: 0.9757699845775065
Voting Regressor Validation Mean Squared Error: 11960533815.022825
Voting Regressor Validation R^2 Score: 0.8871552829276315
```

As we can see, the results haven got worse in comparison to Random Forest and Extra Trees Regressor. This could be mainly due to the fact that I'm using the LinearRegression model, which wasn't good at all, and this can be leading to bad results.

Now I will try the same Voting Regressor method, but without including the Linear Regression model, which could worsen the general model. Here are the results:




```
Voting Regressor Training Mean Squared Error: 156898374.85016942
Voting Regressor Training R^2 Score: 0.9984371203154632
Voting Regressor Validation Mean Squared Error: 10423473698.031572
Voting Regressor Validation R^2 Score: 0.9016570699471408
```

As we can see, the results have improved a lot, but are still similar to the ones obtained with RF or ET.
It seems like the training R-squared can't be improved, with the models seen in class. Maybe a more complex model can perform better, but its implementation would be much harder and probably not worth it. Still +90% in R^2 is a lot and the model is way beyond what I expected initially.

Final Model

To decide which model is the best here's a table to compare them:

Model/Metric	**Training R^2**	**Validation R^2**
Linear regression	0.697	0.694
Ridge Regularization	0.698	0.696
Lasso Regularization	0.698	0.696
Decision Tree Regressor	0.9538	0.832
Random Forest Regressor	0.981	0.9045
Extra Trees Regressor	0.9912	0.9065
Voting Regressor (with Linear Regression)	0.9758	0.887
Voting Regressor (without Linear Regression)	0.9984	0.9016

Analyzing this table, I can tell that there seems to be a tie between Random Forest, Extra Trees and Voting Regressor without Linear Regression, as they all have almost identical values for both the training and validation R^2.

To decide which model to choose, I have focused on another metric that I haven't included here: Time and computational complexity.

If we compare how much time each algorithm has taken, we observe a clear winner: Extra Trees Regressor, with a time of 1 minute and 10 seconds. Next there's Decision Tree with more than 8 minutes and finally Random Forest with more than 20 minutes. On the other hand, Voting Regressor is also a more complex model that involves the three other ensemble methods, so it's not preferable if results are the same.

Thi is why I've chosen as the best model the **Extra Trees Regressor**, both for its **performance** and **computational cost**.

Estimation of generalization performance

To estimate the generalization performance of the Extra Trees Regressor, I will compute its metrics with the best combination of hyperparameters and the Test subset that we reserved some time ago:

```
best_model_et = ExtraTreesRegressor(max_depth=100, min_samples_leaf=2, min_samples_split=4, n_estimators=150, random_state=42
best_model_et.fit(X_train, y_train)

y_pred = best_model_et.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared Score:", r2)
```

```
ExtraTreesRegressor
ExtraTreesRegressor(max_depth=100, min_samples_leaf=2, min_samples_split=4,
                    n_estimators=150, random_state=42)
```

```
Mean Squared Error: 8799917894.911224
R-squared Score: 0.9085290754893243
```

As we can see, my estimation of the generalization performance is a R^2 **= 0.9086**, which is an excellent value for predicting the Sale Price of a house in Miami.

Conclusions

As we have seen in this report, I have been able to implement a predictive model capable of doing predictions with an excellent performance (R^2 = 0.9086).
The results obtained throughout the project indicate that the dataset required a complex model (in this case Ensembles) to better understand the data relations.

I have personally learnt in detail how the whole process of implementing a machine learning model is carried out, from preprocessing to modeling.

I'm looking forward to designing and implementing other models as I really like the idea of predicting the outcome of an observation given a dataset (AI is the future).

(With this project it has been included the complete Jupyter Notebook code)

Exhibit

Here are the libraries I've used in this project. To install them, simply execute:
%pip install *library*

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sn
import pandas as pd
from collections import Counter
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
pd.set_option('display.precision', 3)

from pandas import read_csv
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import KNeighborsRegressor
from sklearn.neighbors import LocalOutlierFactor
from sklearn.metrics import mean_squared_error
from sklearn import preprocessing
from sklearn.preprocessing import PolynomialFeatures
from pandas.plotting import scatter_matrix
from scipy.stats import boxcox
from statsmodels.genmod.generalized_linear_model import GLM
from pandas import read_csv
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.tree import DecisionTreeRegressor
import graphviz

from sklearn.tree import export_graphviz
from sklearn.ensemble import RandomForestRegressor, VotingRegressor,
ExtraTreesRegressor
from imblearn.under_sampling import RandomUnderSampler
from sklearn.model_selection import GridSearchCV, KFold
from time import time
from datetime import timedelta
import seaborn as sns
from matplotlib import pyplot as plt
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
from sklearn.linear_model import LinearRegression, Ridge, RidgeCV, Lasso, LassoCV
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import MinMaxScaler
from statsmodels.genmod.generalized_linear_model import GLM
import statsmodels.api as sm
from scipy import stats
```