Diego Alberto Barceló Nieves
diego.barcelo@im.unam.mx
Mateo Tonatiuh Rodríguez Cervantes
mateotonatiuh@ciencias.unam.mx

Project 2 - House Prices

Inteligencia Artificial al Análisis
Supervisado de Datos
Esteban Avelardo Hernández Vargas

14-10-2021

1 Overview

In this project we are given two data sets[1], each containing information on 1,459 houses in a single city (Ames, Iowa), including 79 variables for each house. The first data set is intended for training our models, while the second is for testing them out. Our goal is to find the best regression model to predict the price of each home. Before modelling the data, we must:

1. Frame the problem and look at the big picture.
2. Prepare the data, checking for faulty entries (empty cells or cells with "nan"s or other values that do not make sense in the context of the problem) and dealing with them.
3. Explore the data with visual tools to gain some insight and intuition.

While modelling the data, we must:

1. Explore many different models and shortlist the best ones.
2. Fine-tune our models, and possibly combine more than one to greater effect.

2 Data Preparation

Correcting data types

After importing both data sets as Pandas data frames and concatenating them to ensure that all of the data will be pre-processed in the same manner with the following code

```
data_dir = Path("~/Data/House Prices/")
df_train = pd.read_csv(data_dir / "train.csv")
df_test = pd.read_csv(data_dir / "test.csv")
df_concat = pd.concat([df_train, df_test])
```

we started inspecting the data by using the function `df_concat.Attribute.unique()`, replacing `Attribute` with each of the dataframe attributes (i.e., columns) to see the unique values inputted in each column, as well as the column's data type. By doing this, we found out which columns had missing values (which appear as `nan`) and, by comparing the outputs with the project's `data_description.txt` file, we also discovered which columns had typos, as well as which had data types that did not correspond to their values. We then created the following function to correct the typos and rename columns whose original names start with a number, since Python has problems applying some functions to them:

```
def correct_typos(df):
    df["MSZoning"] = df["MSZoning"].replace({"C (all)": "C"})
    df["Neighborhood"] = df["Neighborhood"].replace({"NAmes": "Names"})
    df["BldgType"] = df["BldgType"].replace({"2fmCon": "2FmCon"})
```

[1] The data sets may be downloaded from https://www.kaggle.com/c/house-prices-advanced-regression-techniques.

```
    df["BldgType"] = df["BldgType"].replace({"Duplex": "Duplx"})
    df["HouseStyle"] = df["HouseStyle"].replace({"SLv1": "SLvl"})
    df["Exterior2nd"] = df["Exterior2nd"].replace({"Wd Shng": "WdShing"})
    df["Exterior2nd"] = df["Exterior2nd"].replace({"Brk Cmn": "BrkComm"})
    df.rename(columns={
        "1stFlrSF": "FirstFlrSF",
        "2ndFlrSF": "SecondFlrSF",
        "3SsnPorch": "ThreeSeasonPorch",
    }, inplace=True)
    return df
```

After further analysis of the `data_description.txt` file, we took notice of those attributes whose data types appear as numeric, but for which a categorical data type would be more appropriate, and created a list called `nominal_features` with said attributes. This list includes, for example, the attribute `'MSSubClass'`, since the numeric values inputted in this column are nothing but labels, and should not be interpreted as anything more than that. We also took notice of those attributes whose values are strings, but for which an ordinal (i.e. ordered) data type would be more appropriate, and divided them in two ways: we created a list named `Po_to_Ex` for columns with values ranging from `'Po'` (Poor) to `'Ex'` (Excellent) and with no missing values, such as `'ExterQual'`, and a dictionary named `scales` with pairs of ordinal attributes that do have missing values along with their appropriate ordinal scale, such as `'BsmtQual' : ['NA', 'Po', 'Fa', 'TA', 'Gd', 'Ex']`. Then, we created the following function

```
def correct_datatypes(df):
    for nom_feature in nominal_features:
        df[nom_feature] = df[nom_feature].astype("category")
    for ord_feature, scale in scales.items():
        df[ord_feature] = df[ord_feature].astype(CategoricalDtype(scale, ordered=True))
    for feature in po_to_ex:
        df[feature] = df[feature].replace({'Po': 1, 'Fa' : 2, 'TA' : 3, 'Gd' : 4, 'Ex' : 5})
    return df
```

which changes the data type of the attributes in `nominal_features` to categorical instead of numerical and the data type of the attributes in `scales` to an *ordered* categorical type. Additionally, this function replaces the values `'Po', 'Fa', 'TA', 'Gd', 'Ex'` with `1,2,3,4` and `5`, respectively, since through further experimentation we found that this lead to the best results.

Handling missing values

To decide how to handle missing values, we first labelled them with the following function

```
def label_missing_values(df):
    df['KitchenQual'] = df['KitchenQual'].fillna(3)
    df['GarageCars'] = df['GarageCars'].fillna(0)
    for name in df.select_dtypes("category"):
        df[name] = df[name].cat.add_categories('NaN').fillna('NaN')
    return df
```

which filled all of the missing values in categorical columns with `'NaN'`. Futhermore, we decided to fill in missing values for `'KitchenQual'` with the number `3` (equivalent to an average kitchen) and `'GarageCars'` with `0`, since the houses missing a value for this attribute were the same missing values for other garage-related attributes, most probably indicating they do not have a garage. We then used the following code

```
Attribute = 'BsmtFinType1'
df_concat.groupby(Attribute).size().plot(kind='pie', autopct='%.2f', ylabel=Attribute)
```

changing the value of Attribute to that of each column to yield pie charts that allowed us to visualize the distribution of values of each attribute. An example is shown in Figure 1.

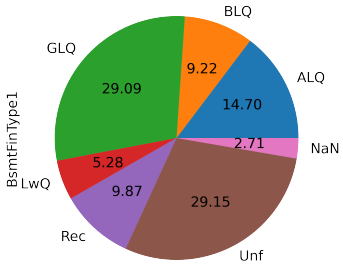

Figure 1: Pie chart plot used to visualize the value distribution of the `'BsmtFinType1'` column.

From these visualizations, we noticed that many attributes that *could* have an `'Na'` (Not Available) value according to the `data_description.txt` file but did not have a single value of this type, indicating that this value was simply ommited, thus leaving empty cells. We then created a list `NaNtoNA` with these attributes. Similarly, we noticed other attributes for which it made sense to change the `'NaN'` values for another value, specific to each case; we recorded these attributes along with theier "default" values in a dictionary named `NaNtoOther`. We then created the following function to implement these changes.

```
def handle_missing_values(df):
    df_concat['BldgType'].replace({'Twnhs' : 'TwnhsI'})
    for name in NaNtoNA:
        df[name] = df[name].replace({'NaN' : 'NA'})
    for name, value in NaNtoOther.items():
        df[name] = df[name].replace({'NaN' : value})
    return df
```

Since many of the functions that we wanted to use to analyze the data could not work with categorical data (in particular, the function that calculated the dataframe's correlation matrix), we encoded all of the information in columns with categorical data type with the Pandas `get_dummies()` function, which is a One Hot Encoder. Finally, we divided our pre-processed dataframe —which was originally a concatenation of the training and testing data sets— into its constituent parts, thus obtaining versions of our original traning and testing data sets which were now prepared for data analysis. To ensure that the previous data preparation worked, we calculated the correlation matrix of the prepared data frame, obtaining that it now worked with all the features present in the data, not only the numerical ones.

3 Data Analysis

Feature Selection

Once we obtained the correlation matrix with the prepared data set, we were able to determine which features are relevant (and useful) to perform the regression models fitting. The theoretical and computational procedure we implement consists of two steps: first, choose a set of features that have an absolute value correlation with the sale price greater than certain amount $n_{SP} \in (0, 1)$,; secondly, check each feature couple from this set and compute their absolute value correlation and, if it is greater that certain amount $n_{corr} \in$

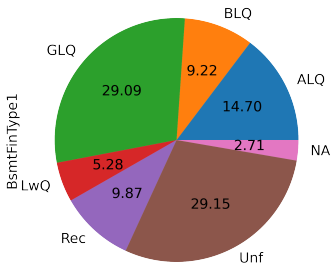

Figure 2: Pie chart plot for the `'BsmtFinType1'` column after applying the `handle_missing_values()` function to our dataframe. The `data_description.txt` file indicated that this attribute could have a `'NA'` (Not Available) value; however, no such value was found, as seen in Figure 1. Furthermore, other basement-related attributes had a similar percentage of missing values, indicating that houses with no basement available had simply been left with an empty cell in all basement-related columns.

$(0, 1)$, then remove the feature that has the least correlation with the sale price. This procedure is meant to reduce multicollinearity and improve the linear regression models.

To perform this calculation, we created the `selectfeatures(df,n_SP,n_corr)` function, that takes the data frame (`df`) and the desire values for n_{SP} (`n_SP`) and n_{corr} (`n_corr`), and returns an array of strings with the features that satisfy the previously described conditions called `relevnt_features`. Here, we show a short step-by-step example of this selection procedure with $n_{SP} = 0.65$ and $n_{corr} = 0.7$:

```
['OverallQual', 'ExterQual', 'GrLivArea', 'KitchenQual']
feature1= OverallQual
feature2= ExterQual
corr[ OverallQual , ExterQual ]= 0.7262784907641455
['OverallQual', 'ExterQual', 'GrLivArea', 'KitchenQual']
corr[ OverallQual , Sale Prices ]= 0.7909816005838047
corr[ ExterQual , Sale Prices ]= 0.6826392416562591
feature removed= ExterQual
['OverallQual', 'GrLivArea', 'KitchenQual']
feature2= GrLivArea
feature2= KitchenQual
['OverallQual', 'GrLivArea', 'KitchenQual']
['OverallQual', 'GrLivArea', 'KitchenQual']
feature1= GrLivArea
feature2= OverallQual
feature2= KitchenQual
['OverallQual', 'GrLivArea', 'KitchenQual']
feature1= KitchenQual
feature2= OverallQual
feature2= GrLivArea
relevant_features=['OverallQual', 'GrLivArea', 'KitchenQual']
```

We can see that, in this example, the conditions used are too restrictive, and thus return a small set of features.

4 Predictive model selection

We used three multi-linear regression models from the `sklearn` python library to perform the sale prices prediction:

- **Linear Regression** (`sklearn.linear_model.LinearRegression`): performs an ordinary least squares Linear Regression. Fits a linear model with coefficients $w = (w_1 1, \cdots, w_p)$ to minimize the residual sum of squares between the observed targets in the data set, and the targets predicted by the linear approximation.
- **Lasso** (`sklearn.linear_model.Lasso`): Linear Model trained with L1 prior as regularizer (aka the Lasso). The optimization objective for Lasso is:

  ```
  (1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
  ```

  The constant `alpha` multiplies the L1 term. Its default value is 1.0. `alpha` $= 0$ is equivalent to an ordinary least square, solved by the `LinearRegression` object.
- **Ridge**[2] (`sklearn.linear_model.Ridge`): Linear least squares with L2 regularization. Minimizes the objective function:

  ```
  ||y - Xw||^2_2 + alpha * ||w||^2_2
  ```

  This model solves a regression model where the loss function is the linear least squares function and regularization is given by the L2-norm. The variable `alpha` represents the regularization strength, which must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization.

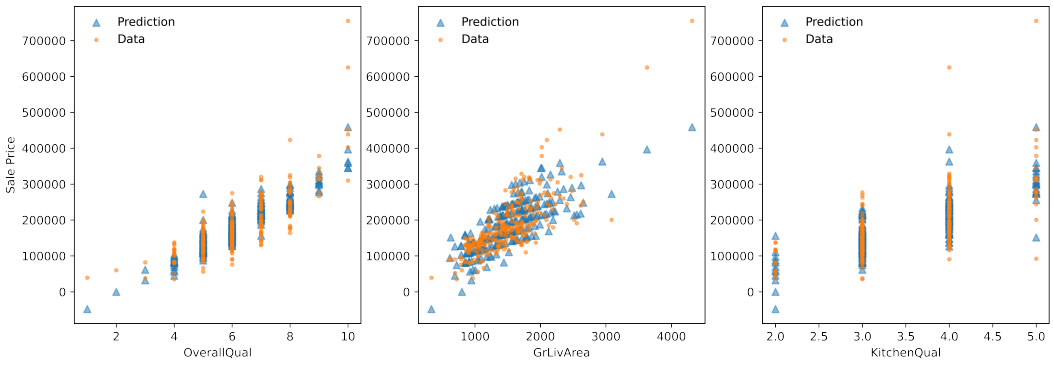

Figure 3: Linear regression sale price predictions compared with the test data for the different features.

To show a short example, we will use the features obtained in the last section to fit a linear regression model. To perform the fitting, we split the train data set in training and testing data using the `sklearn.model_selection.train_test_split` function. Then, we train the model with the training data and compare the result with the testing data, as shown in Figure 3.

[2] Also known as Ridge Regression or Tikhonov regularization.

For each model, we compute its coefficient of determination R^2, defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares and v is the total sum of squares. The best possible score is 1.0, and the value can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of 'y', disregarding the input features, would get a R^2 score of 0.0. For example, the score obtained for the linear model, trained only with the features found in the last section was 0.77361.

In order to choose the best fitting features for the models, we take runs of different values for the minimum features correlation with Sale Price and the maximum correlation allowed between features. For each case we perform one of the linear regression models from above, and compute the coefficient of determination (score), obtaining the best feature selection and regression model when this values reaches its maximum. In Figure 4 we can see an example of this procedure.

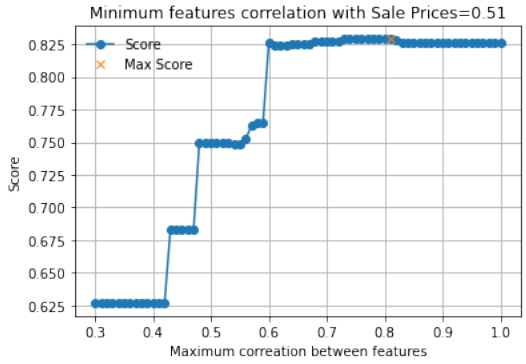

Figure 4: Plot of the Ridge model score against the maximum correlation between features (n_{corr}) ranging from 0.3 to 1 for a fixed value of $n_{sp} = 0.51$. In this example we found a maximum score value= 0.8290 at maximum between correlation n_{corr}= 0.81.

5 Results

We found that the best model is the Ridge model, and the best feature selection occurred when $n_{sp} = 0.49$ and $n_{corr} = 0.81$. With these values we found that the relevant features were

```
['OverallQual',
 'YearBuilt',
 'YearRemodAdd',
 'ExterQual',
 'TotalBsmtSF',
 'GrLivArea',
 'FullBath',
 'KitchenQual',
 'GarageCars',
 'Foundation_PConc',
 'BsmtQual_Ex']
```

The maximum score reached in the validation process was 0.82984. We then trained the Ridge model with these features using the complete prepared training data set. Finally, we predicted the house sale prices by applying this model to the same selected features in the testing data set and exported the results to a file named `SALE_PRICES.csv`.

6 Conclusions

Working with a big data set is complicated in many ways. Dirty or corrupted data can lead to mistakes, and suboptimal data preparation can lead to parts of the data being misinterpreted or not being taken into account during the analysis. Therefore, it is vital to inspect and visualize the data to get familiarized with it and understand how to best prepare it for analysis. After the data is ready for analysis, having a clear strategy to select the relevant features is also important in order to get the bets results, as is using several different models in order to find the most fitting one to the problem at hand.