Project description

Ask a buyer to describe their dream home and they probably won't start with basement ceiling height or proximity to an east-west rail line. But the data set from this playground contest proves that price negotiations are influenced by much more than the number of bedrooms or a white picket fence.

This dataset contains 79 explanatory variables describing almost every aspect of residential homes in Ames, Iowa.

Goal: It is your job to predict the sales price for each house using everything you have learned so far. If **you use a model not presented in class, you must justify it, explain how it works and describe precisely the role of each of the hyper-parameters**. For each Id in the test set, you must predict the value of the SalePrice variable.

Metric: Predictions are evaluated on Root-Mean-Squared-Error (RMSE) between the logarithm of the predicted value and the logarithm of the observed sales price. (Taking logs means that errors in predicting expensive houses and cheap houses will affect the result equally.)

Homework submission: You must upload a zip archive containing 3 files to``lms.univ-cotedazur.fr`:

- A `pdf` report describing for each of the selected features the treatment performed
- A `jupyter notebook` performing the preprocessing, each step of which is inserted into a sklearn or imblearn pipeline (you must leave traces of notebook executions. The first cell should have the number 1, the second the number 2, etc.)
- A `result.csv` should contain your prediction for each of the properties in tthe test set in the the following format:
    - `Id,SalePrice`
    - `1461,169000.9876`
    - `1462,187724.1233`
    - `1463,175221.1928`
    - `etc.`

In order to achieve this project, first of all I started by loading the training dataset:

```
# Read train files
import pandas as pd

df_train = pd.read_csv("train.csv", index_col=0)
print(len(df_train))
df_train.head(3)
```

```
1000
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	...	PoolArea	PoolQC	Fence	MiscFeature	MiscV
0	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	Inside	...	0	NaN	NaN	NaN	
1	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	FR2	...	0	NaN	NaN	NaN	
2	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	Inside	...	0	NaN	NaN	NaN	

3 rows × 80 columns

After that I split the data because we'll need the features/labels a nd the target variable:

```
X = df_train.drop(["SalePrice"], axis=1) #separating the labels and targets
```

```
y = df_train["SalePrice"] #separating the labels and targets
```

After that I needed to filter the numeric columns and the object columns because if we're going to be fitting our model, we need numeric values so to achieve that I thought about doing hot encoding on the non-numerical values in order to turn them into numeric ones

```
#Finding the numerical featues
numeric_cols = [colname for colname in X.columns if X[colname].dtype in ["int64", "float64"]]

print("there are  " + str(len(numeric_cols)) + " numerical columns in the dataframe")
print("\n")
print("Numerical columns are: {}".format(numeric_cols))
```

```
there are  36 numerical columns in the dataframe


Numerical columns are: ['MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemod
Add', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF',
'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd',
'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch',
'ScreenPorch', 'PoolArea', 'MiscVal', 'MoSold', 'YrSold']
```

```
# Filtering the other categorical features so I can do hot encoding on them because we can't fit object type cols
k = (X.dtypes == "object")

categorical_cols = list(k[k].index)

print("Categorical columns are: {}".format(categorical_cols))
```

```
Categorical columns are: ['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSl
ope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFin
Type1', 'BsmtFinType2', 'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual', 'Functional', 'FireplaceQ
u', 'GarageType', 'GarageFinish', 'GarageQual', 'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature', 'SaleTy
pe', 'SaleCondition']
```

Since now I know the non-numeric columns, I now need to find the exact number of unique values per column:

```
# Now it's time to find the exact  number of unique valyes per column so that I know how many values I should aim for

categorical_dict = {}

for i in X[categorical_dict].columns:
    categorical_dict[i] = len(X[i].unique())

categorical_dict = sorted(categorical_dict.items(), key=lambda x: x[1])

print(cat_dict)
```

```
[('Street', 2), ('Utilities', 2), ('CentralAir', 2), ('Alley', 3), ('LandSlope', 3), ('PavedDrive', 3), ('PoolQC',
3), ('LotShape', 4), ('LandContour', 4), ('ExterQual', 4), ('Heating', 4), ('KitchenQual', 4), ('GarageFinish', 4),
('MiscFeature', 4), ('MSZoning', 5), ('LotConfig', 5), ('BldgType', 5), ('RoofStyle', 5), ('MasVnrType', 5), ('ExterC
ond', 5), ('BsmtQual', 5), ('BsmtCond', 5), ('BsmtExposure', 5), ('HeatingQC', 5), ('Electrical', 5), ('Fence', 5),
('Condition2', 6), ('RoofMatl', 6), ('Foundation', 6), ('FireplaceQu', 6), ('GarageQual', 6), ('GarageCond', 6), ('Sa
leCondition', 6), ('BsmtFinType1', 7), ('BsmtFinType2', 7), ('Functional', 7), ('GarageType', 7), ('HouseStyle', 8),
('Condition1', 9), ('SaleType', 9), ('Exterior1st', 11), ('Exterior2nd', 15), ('Neighborhood', 25)]
```

After that I had to proceed with the separation of the ordinal and nominal categorical features:

```
In [57]: #separating the ordinal and nominal columns

         categorical_ordinal = ["Utilities", "ExterQual", "ExterCond",
                                "BsmtQual", "BsmtCond", "BsmtExposure",
                                "BsmtFinType1", "BsmtFinType2", "HeatingQC",
                                "KitchenQual", "FireplaceQu","GarageFinish",
                                "GarageQual", "GarageCond", "PoolQC", "Fence"]

         categorical_nominal = ["MSZoning", "Street", "Alley", "LotShape",
                                "LandContour", "LotConfig","LandSlope", "Condition1",
                                "Condition2", "BldgType", "HouseStyle","RoofStyle",
                                "RoofMatl", "MasVnrType", "Foundation", "Heating",
                                "CentralAir","Electrical", "Functional", "GarageType",
                                "PavedDrive", "SaleType", "SaleCondition"]
```

Following that, I had to check the number of NaN values and how many of them exist in each column of the training dataframe:

```
: # Time to clean the data and find how many of the columns have NAN values and how many of them are in each col

  nan_values = {}

  for j in X[categorical_ordinal + categorical_nominal].columns:
      if X[j].isnull().sum() > 0:
          nan_values[j] = X[j].isnull().sum()

  print(nan_values)
```

```
{'BsmtQual': 24, 'BsmtCond': 24, 'BsmtExposure': 25, 'BsmtFinType1': 24, 'BsmtFinType2': 25, 'FireplaceQu': 478, 'Gar
ageFinish': 56, 'GarageQual': 56, 'GarageCond': 56, 'PoolQC': 998, 'Fence': 806, 'Alley': 935, 'MasVnrType': 6, 'Gara
geType': 56}
```

After that I got to the most important part which is replacing the non-numeric values by a numeric value and that's going to be done through hot encoding of the values to certain values such as : 0,1,2,3… depending on how many unique values the column has:

```
# Now it's time to use a function which will allow us to convert all the non numerical values to numerical values
#I did that by encoding them as following :

def encode_categorical_ordinal(X):
    X["ExterQual"][X["ExterQual"] == "Ex"] = 5
    X["ExterQual"][X["ExterQual"] == "Gd"] = 4
    X["ExterQual"][X["ExterQual"] == "TA"] = 3
    X["ExterQual"][X["ExterQual"] == "Fa"] = 2
    X["ExterQual"][X["ExterQual"] == "Po"] = 1

    X["ExterCond"][X["ExterCond"] == "Ex"] = 5
    X["ExterCond"][X["ExterCond"] == "Gd"] = 4
    X["ExterCond"][X["ExterCond"] == "TA"] = 3
    X["ExterCond"][X["ExterCond"] == "Fa"] = 2
    X["ExterCond"][X["ExterCond"] == "Po"] = 1

    X["BsmtQual"][X["BsmtQual"] == "Ex"] = 5
    X["BsmtQual"][X["BsmtQual"] == "Gd"] = 4
    X["BsmtQual"][X["BsmtQual"] == "TA"] = 3
    X["BsmtQual"][X["BsmtQual"] == "Fa"] = 2
    X["BsmtQual"][X["BsmtQual"] == "Po"] = 1
    X["BsmtQual"][X["BsmtQual"].isnull()] = 0

    X["BsmtCond"][X["BsmtCond"] == "Ex"] = 5
    X["BsmtCond"][X["BsmtCond"] == "Gd"] = 4
    X["BsmtCond"][X["BsmtCond"] == "TA"] = 3
    X["BsmtCond"][X["BsmtCond"] == "Fa"] = 2
    X["BsmtCond"][X["BsmtCond"] == "Po"] = 1
    X["BsmtCond"][X["BsmtCond"].isnull()] = 0

    X["BsmtExposure"][X["BsmtExposure"] == "Gd"] = 4
    X["BsmtExposure"][X["BsmtExposure"] == "Av"] = 3
    X["BsmtExposure"][X["BsmtExposure"] == "Mn"] = 2
    X["BsmtExposure"][X["BsmtExposure"] == "No"] = 1
```

Since it's requested to use a pipeline, I created a pipeline for the categorical columns using Pipeline from the sklearn library while of course using the BaseEstimator and transformerMixin along with SimpleInputer and ColumnTransformer which will help us easily manage the creation of the pipeline:

```
# Time to create the pipeline of categorical cols

from sklearn.base import BaseEstimator, TransformerMixin

class OrdinalEncoder(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return encode_categorical_ordinal(X)
```

```
# Now it's time to apply one hot encoding on some of the features

from sklearn.preprocessing import OneHotEncoder

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

cat_ordinal_transformer = Pipeline(steps=[("ordinal_encoder", OrdEncoder())])

cat_nominal_transformer = Pipeline(steps=[("imputer", SimpleImputer(strategy="constant")),("oh_encoder", OneHotEncoder(
```

After that I created a numerical transformer that using the SimpleImputer function which will help us replace the NaN values with a strategic value such as the median, mean....

```
numerical_transformation = SimpleImputer(strategy="median")
#I used SimpleImputer because it's an already existing function that will allow us
#to replace the NaN values with a strategic values such as the median, mean....
```

```
# Creating a pre-processing pipeline
pre_processor = ColumnTransformer(
    transformers=[
        ("numerical", numerical_transformation, numeric_cols),
        ("categorical_ordinal", cat_ordinal_transformer, categorical_ordinal),
        ("categorical_nominal", cat_nominal_transformer, categorical_nominal)
    ]
)
```

Now it's time to create the pre-processing pipeline for the categorical ordinal and nominal values:

```
# Creating a pre-processing pipeline
pre_processor = ColumnTransformer(
    transformers=[
        ("numerical", numerical_transformation, numeric_cols),
        ("categorical_ordinal", cat_ordinal_transformer, categorical_ordinal),
        ("categorical_nominal", cat_nominal_transformer, categorical_nominal)
    ]
)
```

And then I decided to aim for RandomForests because it suits the objective of this project and using GridSearch which will implement a fit and a score method which will allow us to implement score samples and more:

```
from sklearn.ensemble import RandomForestRegressor

rand_forest = RandomForestRegressor(n_estimators=100, random_state=0) #Creating the model
```

```
sklearn.model_selection import GridSearchCV

_parameters = [{"n_estimators": [50, 100, 500, 750, 1000]}]

_search = GridSearchCV(rnd_forest, grid_parameters, cv=3, scoring="neg_root_mean_squared_error")

sed GridSearchCV because it implements a fit and a score method which will allow us to implement score samples and more
```

```
overall_pipeline = Pipeline(steps=[("preprocessor", pre_processor),("grid_search", grid_search)])

overall_pipeline.fit(X, y)
```

Time to test our overall pipeline transformer and we can also see how gridsearch is useful to get the best parameters, estimator, and the best score to aim for

```
overall_pipeline = Pipeline(steps=[("preprocessor", pre_processor),("grid_search", grid_search)])

overall_pipeline.fit(X, y)
```

```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('numerical',
                                                  SimpleImputer(strategy='median'),
                                                  ['MSSubClass', 'LotFrontage',
                                                   'LotArea', 'OverallQual',
                                                   'OverallCond', 'YearBuilt',
                                                   'YearRemodAdd', 'MasVnrArea',
                                                   'BsmtFinSF1', 'BsmtFinSF2',
                                                   'BsmtUnfSF', 'TotalBsmtSF',
                                                   '1stFlrSF', '2ndFlrSF',
                                                   'LowQualFinSF', 'GrLivArea',
                                                   'BsmtFullBath',
                                                   'BsmtHalfBath', 'Fu...
```

Moment of truth, time to create a function which will allow us to evaluate our model and decide to wither the score is good and the results are accurate or not

```
from sklearn.metrics import mean_squared_error

def model_score(X, y, model): #I create an evaluation function which will allow us to evaluate the model
    y_pred = model.predict(X)
    return mean_squared_error(y, y_pred, squared=False)
```

```
model_score(X, y, overall_pipeline)
```

```
11054.742722657827
```

```
print(grid_search.best_params_, grid_search.best_estimator_, -grid_search.best_score_)
#Now we use gridsearch to get the best parameters, estimator and the best score to aim for
```

```
{'n_estimators': 1000} RandomForestRegressor(n_estimators=1000, random_state=0) 0.023192683865020713
```

Then I decided to use xgboost because due to its many positive features XGBoost is a powerful and widely used tool for building high-performance machine learning models, and is particularly well-suited for tasks involving large datasets and high-dimensional data.

XGBoost is fast and efficient: It has a highly optimized implementation of gradient boosting that makes it faster than many other implementations of the algorithm. XGBoost is flexible: It provides a range of parameters that can be fine-tuned to achieve the desired model performance, such as the learning rate, the number of trees, and the depth of the trees. XGBoost is accurate: It has been shown to perform well on a variety of tasks and consistently ranks among the top models in machine learning competitions:

```
#I use xgboost because it implements of gradient boosting trees algorithm which will help us in predicting the house pr

xgb = XGBRegressor(n_jobs=6, random_state=0)

grid_parameters = [
    {"n_estimators": [100, 300, 500, 700, 1000], "learning_rate": [0.005, 0.01, 0.05, 0.1]}]

grid_search = GridSearchCV(xgb, grid_parameters, cv=3,scoring="neg_root_mean_squared_error")

overall_pipeline = Pipeline(steps=[
    ("preprocessor", pre_processor),
    ("grid_search", grid_search)
])

overall_pipeline.fit(X, y)
```

```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('numerical',
                                                  SimpleImputer(strategy='median'),
                                                  ['MSSubClass', 'LotFrontage',
                                                   'LotArea', 'OverallQual',
                                                   'OverallCond', 'YearBuilt',
```

Finally, it's time to load the test dataset to test my model, and as a result I got a csv file containing the prediction the model had on the test data:

```
# Read test file
df_test = pd.read_csv("test.csv", index_col=0)
print(len(df_test))
df_test.head(3)
```

```
460
```

id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	...	ScreenPorch	PoolArea	PoolQC	Fence	Mi
1000	20	RL	74.0	10206	Pave	NaN	Reg	Lvl	AllPub	Corner	...	0	0	NaN	NaN	
1001	30	RL	60.0	5400	Pave	NaN	Reg	Lvl	AllPub	Corner	...	0	0	NaN	NaN	
1002	20	RL	75.0	11957	Pave	NaN	IR1	Lvl	AllPub	Inside	...	0	0	NaN	NaN	

3 rows × 79 columns

```
X_test = pre_processor.transform(df_test)
```

```
y_pred = grid_search.predict(X_test)
```

```
df_results = pd.DataFrame({"Id": df_test.index,"SalePrice": y_pred})
df_results.to_csv("results.csv", index_label='id')
df_results.head()
```

	Id	SalePrice
0	1000	81688.328125
1	1001	84638.812500

Results.csv:

results

id	Id	SalePrice
0	1000	81688.33
1	1001	84638.81
2	1002	240351.75
3	1003	146560.02
4	1004	185222.39
5	1005	134797.11
6	1006	168405.75
7	1007	85872.28