Housing Prices Competition for Kaggle Learn Users (pt. 2)

Pablo de la Asunción Cumbrera Conde

En el siguiente proyecto, continuación de 'Housing Prices Competition for Kaggle Learn Users (pt.1)', profundizaremos en la mejora de modelos desarrollados para la competicion Housing Prices Competition for Kaggle Learn Users sobre la aplicación de Machine Learning para la predicción de precios de viviendas en Ames, Iowa.

La descripción e introducción de la competición reza así:

- *Pídale a un comprador de vivienda que describa la casa de sus sueños, y probablemente no comenzará con la altura del techo del sótano o la proximidad a un ferrocarril que cruce todo el país. No obstante, este conjunto de datos para esta distendida competición demuestra que influye mucho más sobre el precio las negociaciones que el número de dormitorios o una valla blanca.*

Objetivos para la competición:

- Con 79 variables explicativas que describen (casi) todos los aspectos de las viviendas residenciales en Ames, Iowa, esta competencián desafía al usuario a predecir el precio final de cada casa.

Habilidades a poner en práctica:

- Ingeniería creativa de funciones
- Técnicas de regresión avanzadas como 'Random Forest' y 'Gradient Boosting'.
- Limpieza de datos y optimización de modelos

Contenidos

- Update y set del conjunto de datos
- Métodos y valoración de sustitución de valores ausentes
- Creación de pre-procesamiento de datos a través de pipeline
- Construcción Random Forest con pipeline
- Construcción XGBoost
- Comparación modelos y métodos
- Conclusión

Update y set del conjunto de datos

In [92]:
```
import pandas as pd
from sklearn.model_selection import train_test_split

# Leemos datos
X_full = pd.read_csv('file:///C:/Users/pablo/OneDrive/Escritorio/Data/Data%20sets/housingprice_train.csv', index_col='Id')
X_test_full = pd.read_csv('file:///C:/Users/pablo/OneDrive/Escritorio/Data/Data%20sets/housingprice_test.csv', index_col='Id')

# Obtención de objetivos y predictores
y = X_full.SalePrice
features = ['LotArea', 'YearBuilt', '1stFlrSF', '2ndFlrSF', 'FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd']
X = X_full[features].copy()
X_test = X_test_full[features].copy()

# Para ello primero separamos las filas con valores ausentes, haciendo a un lado objetivos y predictores.
X_full.dropna(axis=0, subset=['SalePrice'], inplace=True)
y = X_full.SalePrice
X_full.drop(['SalePrice'], axis=1, inplace=True)


# Para mantener la simpleza, solo utilizaremos predictores numéricos.
X = X_full.select_dtypes(exclude=['object'])
X_test = X_test_full.select_dtypes(exclude=['object'])


# Separación de valores del set para entrenamiento
X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=0.2,
                                                      random_state=0)
X_full.head()
```

Out[92]:

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	...	ScreenPorch	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	MoSold	YrSold	SaleType	SaleCondition
Id																					
1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	Inside	...	0	0	NaN	NaN	NaN	0	2	2008	WD	Normal
2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	FR2	...	0	0	NaN	NaN	NaN	0	5	2007	WD	Normal
3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	Inside	...	0	0	NaN	NaN	NaN	0	9	2008	WD	Normal
4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	Corner	...	0	0	NaN	NaN	NaN	0	2	2006	WD	Abnorml
5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	FR2	...	0	0	NaN	NaN	NaN	0	12	2008	WD	Normal

5 rows × 79 columns

In [93]:
```
X_test_full.head()
```

Out[93]:

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	...	ScreenPorch	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	MoSold	YrSold	SaleType	SaleCondition
Id																					
1461	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	AllPub	Inside	...	120	0	NaN	MnPrv	NaN	0	6	2010	WD	Normal
1462	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	AllPub	Corner	...	0	0	NaN	NaN	Gar2	12500	6	2010	WD	Normal
1463	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	AllPub	Inside	...	0	0	NaN	MnPrv	NaN	0	3	2010	WD	Normal
1464	60	RL	78.0	9978	Pave	NaN	IR1	Lvl	AllPub	Inside	...	0	0	NaN	NaN	NaN	0	6	2010	WD	Normal
1465	120	RL	43.0	5005	Pave	NaN	IR1	HLS	AllPub	Inside	...	144	0	NaN	NaN	NaN	0	1	2010	WD	Normal

5 rows × 79 columns

In [94]:
```
X_train.head()
```

Out[94]:

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	...	GarageArea	WoodDeckSF	OpenPorchSF	EnclosedPorch	3SsnPorch	ScreenPorch	Poo
Id																		
619	20	90.0	11694	9	5	2007	2007	452.0	48	0	...	774	0	108	0	0	260	
871	20	60.0	6600	5	5	1962	1962	0.0	0	0	...	308	0	0	0	0	0	
93	30	80.0	13360	5	7	1921	2006	0.0	713	0	...	432	0	0	44	0	0	
818	20	NaN	13265	8	5	2002	2002	148.0	1218	0	...	857	150	59	0	0	0	
303	20	118.0	13704	7	5	2001	2002	150.0	0	0	...	843	468	81	0	0	0	

5 rows × 36 columns

Utilizaremos el set de entrenamiento de la X

Realizamos una inserción en los valores que faltan (NaN- Not A Number) a través de la prueba de diversos métodos para probar cuál es el óptimo. Para evaluar las diferentes formas de aproximación utilizaremos el 'MAE'.

In [95]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

# Función para comprobar la exactitud de las aproximaciones
def score_dataset(X_train, X_valid, y_train, y_valid):
    model = RandomForestRegressor(n_estimators=100, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

Utilizaremos el set de entrenamiento de la X

In [96]:
```
print(X_train.shape)

# Number of missing values in each column of training data# Fill in the line below: get names of columns with missing values
cols_with_missing = [col for col in X_train.columns
                     if X_train[col].isnull().any()]

# Drop columns in training and validation data
reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1) # Your code here

# Check your answers
missing_val_count_by_column = (X_train.isnull().sum())
print(missing_val_count_by_column[missing_val_count_by_column > 0])
```

```
(1168, 36)
LotFrontage    212
MasVnrArea       6
GarageYrBlt     56
dtype: int64
```

Tenemos 1168 entradas conocidas, divididas en 37 columnas. En adición, faltan 212 de la sección 'LotFrontage' ó de 'MasVnrArea' y 56 de 'GarageYrBlt' ¿Qué porcentaje de nuestros datos suponen estos faltantes?

In [97]:
```
#Sumamos los datos y los dividimos entre en total
a=(212 + 6 + 56) / (212 + 6 +56 + 1168)
a
```

Out[97]:
```
0.19113573407202217
```

Métodos de sustitución de valores ausentes

1.Dado que los datos restantes no suponen una gran cantidad del total (por debajo del 20%) probaremos si la mejor forma de lidiar con ellos sería eliminar estas entradas.

In [166]:
```
cols_with_missing = [col for col in X_train.columns
                     if X_train[col].isnull().any()]

# Eliminamos estas columnas en los DOS SET DE DATOS (No sólo en uno) entrenamiento y validación.
reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)
```

In [167]:
```
print("MAE (Sin columnas de valores vacios):")
print(score_dataset(reduced_X_train, reduced_X_valid, y_train, y_valid))
```

```
MAE (Sin columnas de valores vacios):
17837.82570776256
```

1. Probamos el método de la sustitución de los valores restantes por valores medios de cada columna (Imputation)

In [168]:
```
from sklearn.impute import SimpleImputer

# Imputation
my_imputer = SimpleImputer()
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
imputed_X_valid = pd.DataFrame(my_imputer.transform(X_valid))

# Imputation: Eliminación de valores ausentes; sustitución.
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns
```

In [169]:
```
print("MAE (Imputation):")
print(score_dataset(imputed_X_train, imputed_X_valid, y_train, y_valid))
```

```
MAE (Imputation):
18062.894611872147
```

Los resultados arrojan la siguiente información: El primer método (Dropping) ha funcionado, en este caso mejor que el segundo método (Imputation)

- MAE DROP: 17837.82
- MAE IMPUTATION- 18062.89

Creación de pre-procesamiento de datos a través de pipeline

Desarrollamos nuestro pipeline

Un pipeline es una forma sencilla de mantener el código de modelado organizado y preprocesar los datos. Por lo tanto, agrupa los pasos de pre-procesamiento y modelado para que podamos unir todos estos pasos en uno solo.

Aunque no es obligatorio, de hecho muchos científicos no realizan pipelines, esta forma de proceder tiene muchas ventajas como:

- Código más limpio: tener en cuenta los datos en cada paso del preprocesamiento puede resultar complicado. Con un pipeline no se necesita realizar un seguimiento manual de sus datos de capacitación y validación en cada paso.
- Menos errores: como una aplicación incorrecta o el olvido de un paso.
- Más fácil de producir: puede ser sorprendentemente difícil hacer la transición de un modelo de un prototipo a algo que se pueda implementar a escala, el pipeline puede ayudar en ésto.
- Más opciones para la validación del modelo.

In [161]:
```
# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient but arbitrary)
categorical_cols = [cname for cname in X_train.columns if
                    X_train[cname].nunique() < 10 and
                    X_train[cname].dtype == "object"]

# Select numerical columns
numerical_cols = [cname for cname in X_train.columns if
                  X_train[cname].dtype in ['int64', 'float64']]

# Keep selected columns only
my_cols = categorical_cols + numerical_cols
X_trained = X_train[my_cols].copy()
X_valid = X_valid[my_cols].copy()
X_test = X_test_full[my_cols].copy()
```

In [182]:
```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Preprocesamos datos numéricos
numerical_transformer = SimpleImputer(strategy='constant')

# Preprocesamos datos categóricos
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Procesamiento previo de paquetes para datos numéricos y categóricos
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])
```

A través de este pipeline, construimos un modelo Random Forest

In [183]:
```
# Definimos el modelo
model = RandomForestRegressor(n_estimators=100, random_state=0)

# Agrupamos código de preprocesamiento y modelado en una canalización
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('model', model)
                     ])

# Preprocesamos los datos de validación y obtenemos predicciones
preds = clf.predict(X_valid)

print('MAE:', mean_absolute_error(y_valid, preds))
```

```
MAE: 18017.665970319635
```

Construcción modelo XGBoost

Construimos un modelo de Gradiend Boosting. Este es un método que pasa por ciclos para agregar modelos de forma iterativa a un conjunto.

El ciclo empieza por inicializar el conjunto con un solo modelo, cuyas predicciones pueden ser bastante ingenuas. (Incluso si sus predicciones son tremendamente inexactas, las adiciones posteriores al conjunto abordarán esos errores).

In [186]:
```
from xgboost import XGBRegressor
my_model_1 = XGBRegressor(random_state=0)

# Fit the model
my_model_1.fit(X_trained, y_train)

predictions_1 = my_model_1.predict(X_valid)
```

In [139]:
```
mae_1 = mean_absolute_error(predictions_1, y_valid)
print("Mean Absolute Error:" , mae_1)
```

```
Mean Absolute Error: 18482.946088398974
```

Mejoramos el modelo.

In [194]:
```
# Definimos el modelo
my_model_2 = XGBRegressor(n_estimators=1000,random_state=0, learning_rate=0.1)

# Ventimos el modelo
my_model_2.fit(X_train, y_train)

# Obtenemos predicciones
predictions_2 = my_model_2.predict(X_valid)

# Calculamos MAE
mae_2 = mean_absolute_error(predictions_2, y_valid)
print("Mean Absolute Error:" , mae_2)
```

```
Mean Absolute Error: 17249.479331165668
```

Comparación de modelos y métodos

Incluimos a XGBoost el pre procesamiento que hicimos en primera instancia. (Método DROP)

In [198]:
```
# Ventimos el modelo
my_model_2.fit(reduced_X_train, y_train)

# Obtenemos predicciones
predictions_2 = my_model_2.predict(reduced_X_valid)

# Calculamos MAE
mae_2 = mean_absolute_error(predictions_2, y_valid)
print("Mean Absolute Error:" , mae_2)
```

```
Mean Absolute Error: 16401.953069244863
```

Incluimos a XGBoost el pre procesamiento Pipeline.

In [190]:
```
my_model_2_pipe = Pipeline(steps=[('preprocessor', preprocessor),
                      ('model', my_model_2)
                     ])


# Preprocesamos training data
my_model_2_pipe.fit(X_trained, y_train)

# Preprocesamos los datos de validación y obtenemos predicciones
predictions = my_model_2_pipe.predict(X_valid)

print('MAE:', mean_absolute_error(y_valid, predictions))
```

```
MAE: 17581.454061429795
```

Conclusiones:

- Modelo Random Forest Pipeline: 18017.7
- Modelo XGboost: 17249.5
- Modelo XGboost Drop: 16402
- Modelo XGboost Pipeline: 17581.5

Usamos por tanto el método XGboost 16401.953 para la competición.

In [192]:
```
# Guardamos las predicciones
output = pd.DataFrame({'Id': X_test.index,
                       'SalePrice': preds_test})
output.to_csv('submissionboost.csv', index=False)
```

In [ ]:
```

```