Inteligencia de Negocio (2017-2018)
Grado en Ingeniería Informática
Universidad de Granada

Práctica 3

Juan José Sierra González
jjsierra103@gmail.com

10 de enero de 2018

Índice

Índice de figuras

Índice de tablas

1. Introducción

La última práctica de la asignatura de Inteligencia de Negocio consiste en participar en una competición dentro de la conocida plataforma de ciencia de datos Kaggle. Esta competición se realiza a nivel mundial pero los alumnos de la UGR competimos entre nosotros para ver quién consigue la mejor solución, es decir, obtener un error cuadrático medio menor sobre las predicciones de la variable respuesta. Como la competición es de la categoría "Getting started" dentro de la plataforma está indicado que es una competición orientada al aprendizaje, y se pueden encontrar muchos kernels y manuales de ayuda para facilitar la comprensión y el estudio del problema.

El problema al que nos enfrentamos es tratar de predecir el precio de aproximadamente 1500 viviendas de Ames, Iowa, a partir de otras 1500 viviendas de las que conocemos 79 variables descriptivas y una variable respuesta (el precio de venta de la casa) que será la que tendremos que predecir.

2. Tabla de resultados

En esta competición he realizado un total de **18 subidas a Kaggle**. Mi posición final fue la 456, con un error cuadrático medio de 0.11692. A continuación se mostrará una tabla con los resultados obtenidos en cada una de las subidas y una pequeña descripción del modelo empleado. Así se podrá observar de forma clara la evolución que se ha ido produciendo en los modelos y cómo esto ha influido menor o mayormente en el resultado obtenido.

Subida	Posición	Score	Fecha	Hora	Train RMSLE	Preprocesado	Algoritmos utilizados y parámetros
1	1049	0.12611	28/12/2017	17:30	ElasticNet: 0.11921 GBoosting: 0.07561	Eliminación características >50% NA y no correladas, imputación de NA en train, logaritmo etiquetas, dummies	ElasticNet(alpha=[0.0001..10], l1ratio=[0.01..0.99]) y GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=3)
2	1031	0.12559	31/12/2017	13:50	ElasticNet: 0.11921 GBoosting: 0.07588 XGBoost: 0.02117	Eliminación características >50% NA y no correladas, imputación de NA en train, logaritmo etiquetas, dummies	ElasticNet(alpha=[0.0001..10], l1ratio=[0.01..0.99]), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=3) y XGBoost (estimadores=3000, learning_rate=0.05, max_depth=3)
3	1031	0.13545	31/12/2017	13:59	XGBoost: 0.02117	Eliminación características >50% NA y no correladas, imputación de NA en train, logaritmo etiquetas, dummies	XGBoost (estimadores=3000, learning_rate=0.05, max_depth=3)
4	1016	0.13044	02/01/2018	12:00	ElasticNet: 0.12619 GBoosting: 0.07604 XGBoost: 0.02307	Filtrado características con muchos NA y con información duplicada y no correladas, imputación de NA en train y test, logaritmo etiquetas, dummies	ElasticNet(alpha=[0.0001..10], l1ratio=[0.01..0.99]), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=3) y XGBoost (estimadores=3000, learning_rate=0.05, max_depth=3)
5	1023	0.12832	02/01/2018	15:23	ElasticNet: 0.10232 GBoosting: 0.06869 XGBoost: 0.01971	Eliminación características >50% NA y no correladas, eliminación outliers, imputación de NA en train y test (utilizando mediana y moda), logaritmo etiquetas, dummies	ElasticNet(alpha=[0.0001..10], l1ratio=[0.01..0.99]), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=3) y XGBoost (estimadores=3000, learning_rate=0.05, max_depth=3)
6	1024	0.12690	02/01/2018	16:39	ElasticNet: 0.09642 GBoosting: 0.06507 XGBoost: 0.01717	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies	ElasticNet(alpha=[0.0001..10], l1ratio=[0.01..0.99]), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=3) y XGBoost (estimadores=3000, learning_rate=0.05, max_depth=3)
7	1039	0.12720	03/01/2018	15:05	ElasticNet: 0.09510 GBoosting: 0.10801 XGBoost: 0.06490	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox a todas las variables	ElasticNet(alpha=[0.0001..10], l1ratio=[0.01..0.99]), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=3) y XGBoost (estimadores=3000, learning_rate=0.05, max_depth=3)
8	1041	0.13346	03/01/2018	16:14	ElasticNet: 0.09705	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas	ElasticNet(alpha=[0.0001..10], l1ratio=[0.01..0.99])
9	492	0.11789	03/01/2018	18:38	0.1088	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, label encoding a algunas variables categóricas	Stacked Model con ElasticNet (alpha=0.0005, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=4), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)
10	502	0.11906	04/01/2018	9:33	0.1093	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, label encoding a algunas variables categóricas	Stacked Model con ElasticNet (alpha=0.0005, l1ratio=0.9), XGBoost (estimadores=2200, learning_rate=0.05, max_depth=3), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)
11	450	0.11712	04/01/2018	10:11	0.1085	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, label encoding a algunas variables categóricas	Stacked Model con ElasticNet (alpha=0.0005, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=4), XGBoost (estimadores=2200, learning_rate=0.05, max_depth=3), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)

Tabla 2.1: Resultados descriptivos de cada una de las subidas a Kaggle (subidas 1-11).

Subida	Posición	Score	Fecha	Hora	Train RMSLE	Preprocesado	Algoritmos utilizados y parámetros
12	453	0.11722	04/01/2018	12:10	0.1078	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, label encoding a algunas variables categóricas, ranking de valores en variables categóricas	Stacked Model con ElasticNet (alpha=0.0005, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=4), XGBoost (estimadores=2200, learning_rate=0.05, max_depth=3), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)
13	447	0.11692	04/01/2018	15:11	0.1075	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, label encoding a algunas variables categóricas, ranking de valores en variables categóricas, simplificado del ranking y combinación de variables	Stacked Model con ElasticNet (alpha=0.0005, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=4), XGBoost (estimadores=2200, learning_rate=0.05, max_depth=3), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)
14	447	0.11793	04/01/2018	16:04	0.1073	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, ranking de valores en variables categóricas, simplificado del ranking y combinación de variables, label encoding a nuevas variables categóricas, añadiendo las simplificadas	Stacked Model con ElasticNet (alpha=0.0005, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=4), XGBoost (estimadores=2200, learning_rate=0.05, max_depth=3), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)
15	451	0.11765	04/01/2018	17:25	0.1069	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, ranking de valores en variables categóricas, simplificado del ranking y combinación de variables, label encoding a nuevas variables categóricas, añadiendo las simplificadas, transformaciones exponenciales de variables más correladas con la variable respuesta	Stacked Model con ElasticNet (alpha=0.0005, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=4), XGBoost (estimadores=2200, learning_rate=0.05, max_depth=3), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)
16	452	0.11774	04/01/2018	17:46	0.1069	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, ranking de valores en variables categóricas, simplificado del ranking y combinación de variables, label encoding a anteriores variables categóricas, añadiendo las simplificadas, transformaciones exponenciales de variables más correladas con la variable respuesta	Stacked Model con ElasticNet (alpha=0.0005, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.05, max_depth=4), XGBoost (estimadores=2200, learning_rate=0.05, max_depth=3), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)

Tabla 2.2: Resultados descriptivos de cada una de las subidas a Kaggle (subidas 12-16).

Subida	Posición	Score	Fecha	Hora	Train RMSLE	Preprocesado	Algoritmos utilizados y parámetros
17	454	0.11830	04/01/2018	18:53	0.1070	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, label encoding a anteriores variables categóricas, ranking de valores en variables categóricas, simplificado del ranking y combinación de variables	Stacked Model con ElasticNet (alpha=0.0002, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.02, max_depth=3), XGBoost (estimadores=2200, learning_rate=0.02, max_depth=3), Lasso (alpha=0.0005), y KernelRidge (alpha=0.6, grado=2, coef0=2.5)
18	456	0.11907	04/01/2018	19:25	0.1066	Eliminación outliers, imputación de NA en train y test para casi todas las variables (utilizando mediana y moda), logaritmo etiquetas, dummies, transformación box-cox solo a variables muy sesgadas, label encoding a anteriores variables categóricas, ranking de valores en variables categóricas, simplificado del ranking y combinación de variables	Stacked Model con ElasticNet (alpha=0.0002, l1ratio=0.9), GradientBoosting (estimadores=3000, learning_rate=0.02, max_depth=3) y Lasso (alpha=0.0005)

Tabla 2.3: Resultados descriptivos de cada una de las subidas a Kaggle (subidas 17-18).

3. Explicación de las subidas

En esta sección se incluirá un subapartado por cada subida, donde se indicará qué ha cambiado con respecto a la subida anterior, qué ha propiciado incluir estos cambios y qué resultados han reflejado en el problema para valorar si se ha mejorado el modelo o no.

3.1. Subida 1

Para la primera subida se ha utilizado como base el kernel de Sergei Neviadomski [1], que con un modelo sencillo es capaz de dejar la puntuación rondando la posición 1000 de la competición. El preprocesado que se realiza es esencialmente el mismo que hay en el script que se nos dio por defecto, ya que el principal propósito de realizar esta subida es tomarla como punto de partida y tratar de mejorar a partir de ahí.

En primer lugar se eliminan aquellas variables que tienen aproximadamente el 50 % o más de los valores perdidos, y además se eliminan otras variables que el autor del kernel entiende no correladas con la variable respuesta, es decir, el precio de venta de la vivienda. Aquí se puede ver cómo se eliminan las variables usando las funcionalidades de Pandas y cuáles son esas variables.

```
features.drop(['Utilities', 'RoofMatl', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'Heating', 'LowQualFinSF',
               'BsmtFullBath', 'BsmtHalfBath', 'Functional', 'GarageYrBlt', 'GarageArea', 'GarageCond', 'WoodDeckSF',
               'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC', 'Fence', 'MiscFeature', 'MiscVal'],
              axis=1, inplace=True)
```

Además de esto, con el resto de variables que sí tienen NA entre los datos de train se hace una imputación de dichos valores, por ejemplo rellenando con la moda aquellas variables categóricas, con la media aquellas variables numéricas, y con 0 las variables numéricas que parecen denotar la ausencia de dicho valor. Incluso se categorizan aquellas variables numéricas que pueden actuar como tal. A continuación se incluye un ejemplo de imputación de valores perdidos.

```
# TotalBsmtSF  NA in pred. I suppose NA means 0
features['TotalBsmtSF'] = features['TotalBsmtSF'].fillna(0)

# Electrical NA in pred. filling with most popular values
features['Electrical'] = features['Electrical'].fillna(features['Electrical'].mode()[0])
```

Sobre las etiquetas (en este caso al no tratar un problema de clasificación es más adecuado llamarlas *variable respuesta*) se realiza una transformación logarítmica, con el fin de lograr que los valores estén menos sesgados y sigan una distribución más próxima a una normal.

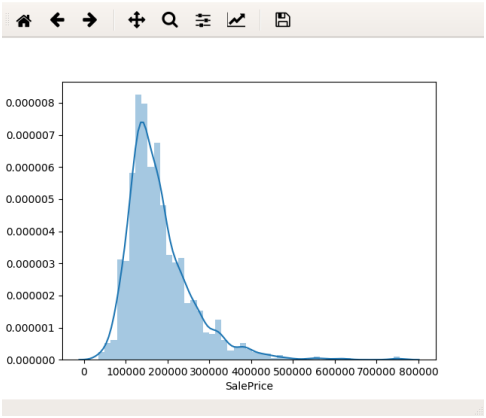

Figura 3.1: Valores sesgados de SalePrice.

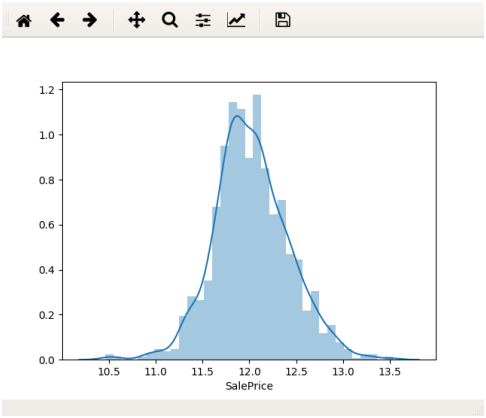

Figura 3.2: Valores no sesgados de SalePrice tras logaritmo.

Por último, como parte del preprocesado se han dividido las variables categóricas en "dummy variables", que son variables que corresponden a solo uno de los valores categóricos y que indican su presencia o no mediante un valor 0 ó 1. Aunque en este kernel se preprocesan dos variables con un dummies especial, la forma general de hacerlo es la siguiente.

```
# Getting Dummies from all other categorical vars
for col in features.dtypes[features.dtypes == 'object'].index:
    for_dummy = features.pop(col)
    features = pd.concat([features, pd.get_dummies(for_dummy, prefix=col)], axis=1)
```

Acabado el preprocesado, en el kernel de Sergei Neviadomski se utilizaban dos algoritmos para predecir los datos: ElasticNet y GradientBoosting. Sobre cada uno de ellos se realiza una cross-validation para obtener un error medio más representativo que no esté

influenciado por seleccionar unos datos en vez de otros para el conjunto de train. Para este caso se obtiene como valor de la variable respuesta el valor medio entre los que hayan encontrado cada uno de los algoritmos previamente indicados. Aquí se puede ver la definición de los modelos y los parámetros de cada uno en detalle.

```
# Elastic Net
ENSTest = linear_model.ElasticNetCV(alphas=[0.0001, 0.0005, 0.001, 0.01, 0.1, 1, 10],
            l1_ratio=[.01, .1, .5, .9, .99], max_iter=5000).fit(x_train_st, y_train_st)


# Gradient Boosting
GBest = ensemble.GradientBoostingRegressor(n_estimators=3000, learning_rate=0.05,
                                           max_depth=3, max_features='sqrt',
                                           min_samples_leaf=15, min_samples_split=10,
                                           loss='huber').fit(x_train, y_train)
```

3.2. Subida 2

En la segunda subida no se ha cambiado el preprocesado, por lo que sigue teniendo los mismos pasos que en el apartado anterior (como se podía apreciar en la tabla 2.1). Lo único que ha cambiado entre la subida anterior y esta es que se ha añadido un nuevo algoritmo de regresión al modelo, **XGBoost**. Elijo añadirlo porque es un algoritmo conocido que obtiene buenos resultados, y los parámetros con los que lo inicializo son similares a los del otro regresor con boosting, Gradient Boosting.

```
# XGBoost
XGBest = xgb.XGBRegressor(max_depth=3, learning_rate=0.05,
                          n_estimators=3000).fit(x_train, y_train)
```

Con este nuevo algoritmo añadido al modelo, las predicciones mejoran ligeramente, pasando de 0.12611 a 0.12559.

3.3. Subida 3

Al ver que el modelo no mejora prácticamente nada decido probar XGBoost por separado, a pesar de que el error cuadrático medio que se obtiene en el conjunto de entrenamiento es muy bajo, lo que da lugar a pensar que con casi toda seguridad habrá sobreajuste.

En efecto, se produce un sobreajuste y en el conjunto de test el error se dispara hasta 0.13545. La idea de dejar XGBoost solo no da un buen resultado.

3.4. Subida 4

Para la cuarta subida decido hacer yo mi propio filtrado de características y no dejar las que aparecían en el primer kernel. Para ello me baso en la información sobre lo que significan las variables y sus respectivos valores en la página de la competición de Kaggle,

además de en algunas ayudas obtenidas del kernel explicativo de Pedro Marcelino [2]. El filtrado de variables se ha subdividido en varias partes, para diferenciar el motivo de filtrado de cada variable, como se puede apreciar. Previamente además se ha realizado una búsqueda sobre los datos para ver qué variables contienen más valores perdidos, y esos son los que se incluyen en el primer filtrado.

A continuación se incluyen el código del filtrado de atributos y la tabla con los atributos con mayor cantidad de valores perdidos, que han indicado cuáles deben pertenecer al grupo de filtrados.

	MissingValues	Percent
PoolQC	2909	0.996574
MiscFeature	2814	0.964029
Alley	2721	0.932169
Fence	2348	0.804385
FireplaceQu	1420	0.486468
LotFrontage	486	0.166495
GarageCond	159	0.054471
GarageQual	159	0.054471
GarageYrBlt	159	0.054471
GarageFinish	159	0.054471

```
# Checking for missing data, showing every variable with at least one missing value in train set
total_missing_data = features.isnull().sum().sort_values(ascending=False)
missing_data_percent = (features.isnull().sum()/features.isnull().count()).sort_values(ascending=False)
missing_data = pd.concat([total_missing_data, missing_data_percent], axis=1, keys=['Total', 'Percent'])
print(missing_data[missing_data['Percent']> 0])

# I get rid of the features that have a lot of missing data
features.drop(['PoolQC', 'MiscFeature', 'Alley', 'Fence', 'LotFrontage'], axis=1, inplace=True)

# Now I drop those features with duplicated information
features.drop(['GarageType', 'GarageYrBlt', 'GarageFinish', 'GarageArea', 'GarageQual', 'GarageCond',
               '3SsnPorch', 'ScreenPorch', 'BsmtQual', 'BsmtCond', 'Heating', 'LandSlope', 'Exterior1st',
               'Exterior2nd', 'KitchenAbvGr', 'BedroomAbvGr', 'Fireplaces'], axis=1, inplace=True)

# Now the same for those features that seem non-related to SalePrice, or do not give much information about it
features.drop(['BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtFullBath',
               'BsmtHalfBath', 'BsmtUnfSF', 'Utilities', 'Street', 'MasVnrType', 'MasVnrArea'], axis=1, inplace=True)
```

Cabe destacar que además en esta ocasión se están mirando también qué datos tienen valores perdidos tanto en train como en test, y por eso se tratan de corregir en el test también. El resultado obtenido no mejora el filtrado de variables que se realizaba en las anteriores subidas, quedándose en 0.13044.

3.5. Subida 5

Para la quinta subida vuelvo a utilizar el filtrado de características inicial. Además, en esta ocasión elimino outliers por primera vez. En la primera figura se pueden considerar outliers los dos datos que tienen un valor alto de área perteneciente a la vivienda, y sin

embargo su precio de venta es reducido. Estos datos se pueden filtrar con la simple línea de código que se muestra a continuación, y deja el dataset más uniforme, como se aprecia en la segunda figura.

```
# Deleting outliers
train = train.drop(train[(train['GrLivArea']>4000) & (train['SalePrice']<300000)].index)
```

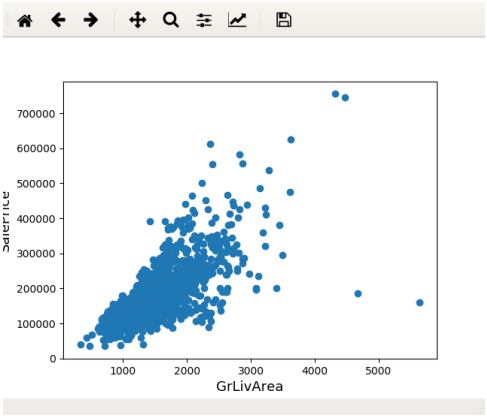

Figura 3.3: Dataset con outliers.

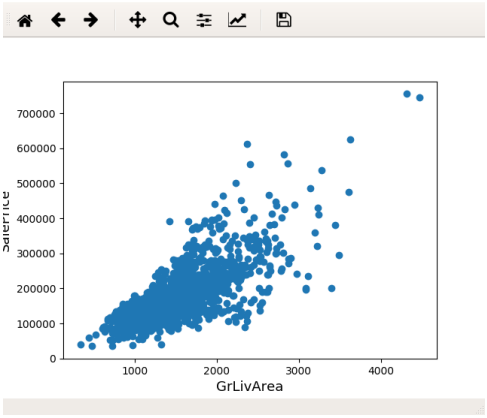

Figura 3.4: Dataset sin outliers.

Además de eliminar los outliers, se ha añadido la mediana a la hora de determinar los valores perdidos de variables numéricas, en lugar de la media. Además, se ha hecho de forma que determine la mediana del atributo en función del resto de valores de ese atributo para los ejemplos que pertenecen a su mismo grupo en OverallQual, que es una de las variables más correladas con la variable respuesta.

```
# LotFrontage NA filling with median according to its OverallQual value
median = features.groupby('OverallQual')['LotFrontage'].transform('median')
features['LotFrontage'] = features['LotFrontage'].fillna(median)
```

El objetivo de hacer esto es que siendo OverallQual una de las variables más determinantes a la hora de predecir el precio de la vivienda, si siguiesen dichas variables numéricas con datos perdidos la misma distribución se podrían obtener buenos resultados. Sin embargo, por la poca cantidad de datos perdidos en dichas variables o por no realizar el preprocesado de forma adecuada, no mejora como estaba previsto, quedando en un error cuadrático medio de 0.12832.

3.6. Subida 6

Viendo que el filtrado de variables parecía seguir restándome información, en este paso decido no filtrar casi ninguna (solamente Utilities, que tiene el mismo valor para casi todas las instancias) y hacer la imputación de valores perdidos correctamente para cada

variable según aparece en la página de la competición.

Esto da un resultado mejor que en la anterior subida, 0.12690, pero sigue sin mejorar el mejor modelo hasta el momento, el de la subida 2.

3.7. Subida 7

En la séptima subida se me ocurre realizar una transformación Box-cox sobre las variables numéricas del dataset. La transformación box-cox se usa para reducir el sesgo en variables muy sesgadas, pero en este caso se han aplicado sobre todas las variables numéricas, como se indica a continuación.

```
# Box-cox transformation
for col in features_normalized.dtypes[features_normalized.dtypes != "object"].index:
    features_normalized[col] = boxcox1p(features_normalized[col], 0.15)
```

Esta transformación sin embargo no mejora los resultados obtenidos, quedando el error cuadrático medio en 0.12720.

3.8. Subida 8

Sabiendo que Box-cox es útil sobre todo para reducir sesgo en variables muy sesgadas, decido aplicar la transformación solamente en aquellas variables sesgadas por un valor por encima de 0.75, como pude ver que se recomendaba hacer en el kernel de Serigne [3].

Este es el fragmento de código en el que se puede observar cómo se realiza la transformación sobre las variables más sesgadas del dataset.

```
# Check the skew of all numerical features
skewed_features = features[numeric_features].apply(lambda x: skew(x.dropna())).sort_values(ascending=False)
print("\nSkew in numerical features: \n")
skewness = pd.DataFrame({'Skew' :skewed_features})
print(skewness.head(10))

# Box-cox
skewness = skewness[abs(skewness) > 0.75]

skewed_features = skewness.index
lam = 0.15
for feat in skewed_features:
    features[feat] = boxcox1p(features[feat], lam)
```

Además, para esta estimación, viendo lo bajos que resultaban los errores cuadráticos medios de GradientBoosting y XGBoost, he decidido probar con ElasticNet solamente, ya que la diferencia entre el RMSLE en train y en test no parece tan grande. Sin embargo, el resultado no es lo bueno que podría esperar, y obtengo un 0.13346.

3.9. Subida 9

Llegados a este punto, tras prácticamente no mejorar en 8 subidas, decido empezar a hacer el script desde cero, ayudándome de aquellos fragmentos que haya notado que me han servido de otras subidas previas, para asegurarme de que sé en todo momento qué se está haciendo con el dataset.

Ahora, aparte de los pasos indicados anteriormente, bien limitados y limpiados de otras variables inservibles o enrevesadas, he añadido el label encoding a algunas variables categóricas, también inspirado en el kernel de Serigne [3]. Esto cambia los valores categóricos de cada atributo por valores enteros entre 0 y número de clases-1.

```
# Label encoding to some categorical features
categorical_features = ('FireplaceQu', 'BsmtQual', 'BsmtCond', 'GarageQual', 'GarageCond',
        'ExterQual', 'ExterCond','HeatingQC', 'PoolQC', 'KitchenQual', 'BsmtFinType1',
        'BsmtFinType2', 'Functional', 'Fence', 'BsmtExposure', 'GarageFinish', 'LandSlope',
        'LotShape', 'PavedDrive', 'Street', 'Alley', 'CentralAir', 'MSSubClass', 'OverallCond',
        'YrSold', 'MoSold')
lbl = LabelEncoder()

for col in categorical_features:
    lbl.fit(list(features[col].values))
    features[col] = lbl.transform(list(features[col].values))
```

Aunque el principal cambio de esta subida es modificar la forma de predecir los datos finales, ya que ahora utilizo un stacked model sencillo que define las funciones básicas fit y predict para todos los modelos que lo construyen, y que se crea como se muestra a continuación. Es la misma estrategia que he seguido hasta ahora, haciendo el modelo "medio" entre los distintos predictores, pero agrupado en una clase que permite hacer las operaciones más fácilmente.

```
# Class AveragingModels
# This class is Serigne's simplest way of stacking the prediction models, by
# averaging them. We are going to use it as it represents the same that we have
# been using in the late submissions, but this applies perfectly to rmsle_cv function.
class AveragingModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, models):
        self.models = models

    # we define clones of the original models to fit the data in
    def fit(self, X, y):
        self.models_ = [clone(x) for x in self.models]

        # Train cloned base models
        for model in self.models_:
            model.fit(X, y)

        return self

    #Now we do the predictions for cloned models and average them
    def predict(self, X):
        predictions = np.column_stack([
            model.predict(X) for model in self.models_
        ])
        return np.mean(predictions, axis=1)
```

Para este modelo se han utilizado ElasticNet y GradientBoosting, y además se han añadido el algoritmo Lasso y el algoritmo KernelRidge, con los siguientes parámetros.

```
enet_model = make_pipeline(RobustScaler(), ElasticNet(alpha=0.0005, l1_ratio=.9, random_state=101010))
```

```
gb_model = ensemble.GradientBoostingRegressor(n_estimators=3000, learning_rate=0.05,
                                              max_depth=4, max_features='sqrt',
                                              min_samples_leaf=15, min_samples_split=10,
                                              loss='huber', random_state =101010)
```

```
lasso_model = make_pipeline(RobustScaler(), Lasso(alpha =0.0005, random_state=101010))
```

```
krr_model = KernelRidge(alpha=0.6, kernel='polynomial', degree=2, coef0=2.5)
```

El ensamblado de modelos, junto al preprocesamiento más depurado, me llevan a mejorar considerablemente por primera vez, colocándome en la posición 492 con un error de 0.11789.

3.10. Subida 10

En la siguiente subida solamente se ha cambiado el modelo ensamblado, sustituyendo el GradientBoosting por el XGBoost, con lo cual queda conformado por Lasso, ElasticNet, XGBoost y KernelRidge. A continuación se muestra la definición del modelo XGBoost, basado en lo aportado por Serigne en su kernel [3].

```
xgb_model = xgb.XGBRegressor(colsample_bytree=0.4603, gamma=0.0468,
                             learning_rate=0.05, max_depth=3,
                             min_child_weight=1.7817, n_estimators=2200,
                             reg_alpha=0.4640, reg_lambda=0.8571,
                             subsample=0.5213, silent=1, nthread = -1)
```

El modelo no aporta mejores resultados, quedándose en 0.11906, por lo que no se avanza en la Leaderboard.

3.11. Subida 11

El siguiente paso que doy es ensamblar todos los modelos que he probado hasta ahora, incluyendo tanto GradientBoosting y XGBoost como KernelRidge, ElasticNet y Lasso. Además los parámetros de XGBoost han sido adaptados con respecto a los que daba

Serigne para dejarlos más simplificados, como se ve a continuación.

```
xgb_model = xgb.XGBRegressor(colsample_bytree=0.2, gamma=0.0,
                             learning_rate=0.05, max_depth=3,
                             min_child_weight=1.7, n_estimators=2200,
                             reg_alpha=0.9, reg_lambda=0.6,
                             subsample=0.5, silent=1, seed=101010)
```

El error cuadrático medio obtenido es de 0.11712, ascendiendo hasta la posición 450.

3.12. Subida 12

Para el camino a tomar en las siguientes subidas me inspiré en otro kernel didáctico, en este caso el publicado por "Juliencs" en la plataforma Kaggle [4]. En este kernel se dan pautas de tratamiento y transformación de las características para conseguir datos de mejor calidad que puedan ayudar a mejorar el modelo.

Para la primera subida ayudándome de este kernel he comenzado planteando yo un label encoding personal para aquellas variables en las que considero que los distintos valores categóricos pueden marcarse por un orden, y lo he establecido así. Además, teniendo en cuenta solo incluir el valor 0 cuando equivale a la no presencia de dicha característica en la vivienda (por ejemplo no hay sótano, o no hay piscina).

A continuación muestro en unas líneas algunos ejemplos de cómo he codificado estos valores para que representen las diferencias que me interesan.

```
# Let's rank those categorical features that can be understood to have an order
# Criterion: give higher ranking to better feature values
features = features.replace({'Street' : {'Grvl':1, 'Pave':2},
                             'Alley' : {'NoAccess':0, 'Grvl':1, 'Pave':2},
                             'LotShape' : {'I33':1, 'IR2':2, 'IR1':3, 'Reg':4},
                             'LandContour' : {'Low':1, 'HLS':2, 'Bnk':3, 'Lvl':4},
                             'LotConfig' : {'FR3':1, 'FR2':2, 'CulDSac':3, 'Corner':4, 'Inside':5},
                             'LandSlope' : {'Gtl':1, 'Mod':2, 'Sev':3},
                             'HouseStyle' : {'1Story':1, '1.5Fin':2, '1.5Unf':3, '2Story':4, '2.5Fin':5, '2.5Unf':6, 'SFoyer':7, 'SLvl':8},
                             'ExterQual' : {'Po':1, 'Fa':2, 'TA':3, 'Gd':4, 'Ex':5},
                             'ExterCond' : {'Po':1, 'Fa':2, 'TA':3, 'Gd':4, 'Ex':5},
                             'BsmtQual' : {'NoBsmt':0, 'Po':1, 'Fa':2, 'TA':3, 'Gd':4, 'Ex':5},
                             'BsmtCond' : {'NoBsmt':0, 'Po':1, 'Fa':2, 'TA':3, 'Gd':4, 'Ex':5},
```

El resultado reflejado, 0.11722, no mejora la mejor solución y me deja en el puesto 453.

3.13. Subida 13

El siguiente paso continuando con la línea de trabajo decidida en la subida anterior es simplificar los valores codificados de estas variables. Es decir, agrupar los valores codificados de las variables para reducir su complejidad.

En el siguiente fragmento de código se muestra cómo se simplifica una variable de calidad para reducir su dimensionalidad.

```
# Now we try to simplify some of the ranked features, reducing its number of values
features['SimplifiedOverallCond'] = features.OverallCond.replace({1:1, 2:1, 3:1, # bad
                                                                  4:2, 5:2, 6:2, # average
                                                                  7:3, 8:3, # good
                                                                  9:4, 10:4 # excellent
                                                                 })
```

Haciendo esto con algunas de las variables que he codificado anteriormente obtengo una serie de nuevas variables "simplificadas". Ahora he generado una serie de nuevas variables de interés que he encontrado oportunas analizando el dominio de variables del problema. Por ejemplo, la suma de las variables $FullBath + HalfBath/2$ puede equivaler a una variable TotalBaths. Además, he generado scores personalizados para valorar algunas de las características esenciales de la vivienda, como el score del garaje o el del sótano. También se han realizado las mismas características utilizando las variables simplificadas.

```
# Overall Score of the exterior (and simplified exterior)
features['ExterScore'] = features['ExterQual'] * features['ExterCond']
features['SimplifiedExterScore'] = features['SimplifiedExterQual'] * features['SimplifiedExterCond']

# Overall Score of the pool (and simplified pool)
features['PoolScore'] = features['PoolQC'] * features['PoolArea']
features['SimplifiedPoolScore'] = features['SimplifiedPoolQC'] * features['PoolArea']
```

Esta nueva generación de características ayuda a mejorar el modelo, hasta obtener un error cuadrático medio en test de 0.11692, alcanzando la posición 447.

3.14. Subida 14

Durante algunas de las siguientes subidas he cometido un despiste, y es modificar las variables a las que afecta el label encoding a pesar de obtener de ello el mismo resultado que ya he hecho a mano con el codificado de variables categóricas. En esta subida se añaden variables de las generadas en la subida anterior, sin modificar el resultado. El error, 0.11793, empeora al anterior a pesar de que en el conjunto de train mejorase el error cuadrático medio.

3.15. Subida 15

En esta subida continúo el preprocesamiento que aconsejaba el kernel de Juliencs [4], ahora realizando transformaciones exponenciales sobre algunas de las variables categóricas. El criterio para elegir cuáles son las que debo modificar ha sido escoger algunas de las más correladas con el precio de venta de la vivienda. He podido obtener esta información mediante la función "corr" de la librería Pandas.

	Correlation to SalePrice
SalePrice	1.000000
TotalSF	0.825326
OverallQual	0.821405
GrLivArea	0.725211
ExterQual	0.682226
GarageCars	0.681033
TotalBaths	0.676678
KitchenQual	0.669990
GarageArea	0.656129
TotalBsmtSF	0.647563

Sobre dichas variables se han realizado algunas transformaciones cuadráticas y cúbicas, como las que se ven a continuación.

```
features['SquaredOverallQual'] = features['OverallQual'] ** 2
features['CubedOverallQual'] = features['OverallQual'] ** 3

features['SquaredTotalSF'] = features['TotalSF'] ** 2
features['CubedTotalSF'] = features['TotalSF'] ** 3
```

El score obtenido en train es menor pero sin embargo 0.11765 no mejora la máxima puntuación obtenida.

3.16. Subida 16

Para esta subida viendo que lo que he hecho con el label encoding no arregla nada, trato de añadir las transformaciones exponenciales al preprocesado con el que se logró la mejor puntuación, el de la subida 13. El error sin embargo no cambia a mejor y se queda en 0.11774.

3.17. Subida 17

Llegado este punto, y viendo que lo último que he probado a hacer con el preprocesado no cambia los resultados, decido eliminar las transformaciones exponenciales (volver de nuevo al preprocesado de la subida más alta en la clasificación) y probar experimentalmente algunas combinaciones de parámetros para optimizar algo el error cuadrático medio en los datos de train.

Así, los parámetros que mejor resultado han dado en el conjunto de entrenamiento (0.1070) para todos los algoritmos del modelo ensamblado han sido los siguientes.

```
enet_model = make_pipeline(RobustScaler(), ElasticNet(alpha=0.0002, l1_ratio=.9, random_state=101010))


gb_model = ensemble.GradientBoostingRegressor(n_estimators=3000, learning_rate=0.02,
                                   max_depth=3, max_features='sqrt',
                                   min_samples_leaf=15, min_samples_split=10,
                                   loss='huber', random_state =101010)


xgb_model = xgb.XGBRegressor(colsample_bytree=0.2, gamma=0.0,
                             learning_rate=0.02, max_depth=3,
                             min_child_weight=1.7, n_estimators=2200,
                             reg_alpha=0.9, reg_lambda=0.6,
                             subsample=0.5, silent=1, seed=101010)


lasso_model = make_pipeline(RobustScaler(), Lasso(alpha=0.0005, random_state=101010))


krr_model = KernelRidge(alpha=0.6, kernel='polynomial', degree=2, coef0=2.5)
```

El error obtenido en train es menor pero en el Leaderboard obtengo un error de 0.11830, por lo que sigo sin ascender.

3.18. Subida 18

La última subida tiene lugar cuando observo que construir el modelo ensamblado con menos modelos da un mejor resultado en el cross-validation. Si utilizo únicamente ElasticNet, GradientBoosting y Lasso reduzco el error cuadrático medio hasta 0.1066, sin embargo en el test de Kaggle el error asciende hasta 0.11907, por lo que mi posición final es la 456 en la competición.

Referencias

[1] Sergei Neviadomski, How to get Top 25 % with simple model (sklearn), Kaggle, `https://www.kaggle.com/neviadomski/how-to-get-to-top-25-with-simple-model-sklearn/notebook`

[2] Pedro Marcelino, Comprehensive data exploration with Python, Kaggle, `https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python/notebook`

[3] Serigne, Stacked Regressions: Top 4 % on LeaderBoard, Kaggle, `https://www.kaggle.com/serigne/stacked-regressions-top-4-on-leaderboard`

[4] Juliencs, A study on Regression applied to the Ames dataset, Kaggle, `https://www.kaggle.com/juliencs/a-study-on-regression-applied-to-the-ames-dataset`