TP1

1. Análisis Exploratorio y Preprocesamiento de Datos

a. Exploración inicial

- Dividimos el dataset original en train-test (80-20)
- Estudiamos el tipo de cada variable del dataset
- Preparamos los datasets (train y test)
  - Nos quedamos con
    - los tipos ‘Casa’, ‘Departamento’, ‘PH’
    - operación ‘Venta’
    - precio ‘USD’
    - ubicación ‘CABA’
  - Convertimos los campos de fechas
- Realizamos una descripción de los campos
- Calculamos medidas de resumen (media, mediana, q1, q3 y moda) para variables cuantitativas (‘property_rooms’ y ‘property_bedrooms’)
- Calculamos cantidades y frecuencias para variables cualitativas (‘place_l3’ → ciudad, ‘place_l4’ → barrio, ‘property_type’)
- Determinamos variables irrelevantes
  - id → no se usa
  - operation → sólo trabajamos con ‘Venta’
  - property_currency → sólo trabajamos con ‘USD’
  - place_l5 → la columna estaba vacía
  - place_l6 → la columna estaba vacía
  - property_title → no se usa
- Visualizamos los gráficos de distribución de cada variable
- Visualizamos la correlación entre todas las variables con un pairplot

b. Datos faltantes

i. A Nivel Columna → por cada variable

- Graficamos por cada variable el porcentaje de datos faltantes (columna vs. registro vacío)

ii. A Nivel Fila → por cada registro

- Graficamos el porcentaje de filas con n cantidad de columnas con datos faltantes (fila vs. cantidad columnas vacías)

iii. Reparación de Datos Faltantes

- place_l3 → llenamos con valores de place_l2 bajo el método de Hot Deck (deberíamos haber realizado Cold Deck y rellenar los valores según datos externos a partir de las columnas de latitud y longitud)
- place_l4 → llenamos con valores de place_l3 (ídem place_l3)

- place_l5 y place_l6 → cambiamos nulos por 0
- latitud y longitud → imputamos a partir de la media agrupando por place_l4
- property_surface_total y property_surface_covered → los reemplazamos entre sí bajo el método de Hot Deck
- property_rooms → reemplazamos por la media según la cantidad de dormitorios
- property_bedrooms → reemplazamos por la media según la cantidad de ambientes
- property_rooms y property_bedrooms → cuando faltan ambas, predecimos mediante regresión lineal property_rooms a partir de property_surface_covered y luego utilizando la media a property_bedrooms

c. Valores atípicos

i. Análisis Univariado

- Boxplot → estudiamos posibles outliers por encima de los límites superiores e inferiores de los cuartiles y borramos en caso de identificar valores atípicos
- Z Score → estudiamos los posibles valores atípicos que superan la regla de oro (|Z| > 3) y borramos aquellos que representaban un porcentaje menor al 1%
- Z Score Modificado → estudiamos los posibles valores atípicos que superan la regla de oro (|Z| > 3.5) y borramos aquellos que representaban un porcentaje menor al 1%

ii. Análisis Multivariado

- Boxplot según property_type → estudiamos posibles outliers por encima de los límites superiores e inferiores de los cuartiles y borramos en caso de identificar valores atípicos
- Distancia de Mahalanobis según property_type → estudiamos los valores de property_rooms y property_bedrooms contra property_surface_total y property_surface_covered y analizamos el dispersograma resultante para observar qué valores categorizar como outliers

d. Reducción de la dimensionalidad

- Para este punto ya tenemos eliminados los considerados outliers
- Eliminamos variables previamente determinadas como irrelevantes
- Encontramos 3 pares de variables altamente correlacionadas y eliminamos una
- Estudiamos variables con varianza baja pero no eliminamos ninguna

2. Agrupamiento

- Transformamos las variables cualitativas con One Hot Encoding
- Estudiamos si existe tendencia al clustering con la estadística de Hopkins (para valores cercanos a 0 significa que hay tendencia)
- Utilizamos el método del codo para definir qué cantidad de clusters utilizar

- Utilizamos el método de Silhouette para determinar qué tan óptima es la cantidad de clusters que elegimos a partir del método del codo}
- Analizamos la clasificación obtenida según cada variable y obtenemos que las más importantes para la clusterización son: ubicación, precio y superficie.

3. Clasificación

a. Construcción del target

- Construimos la variable tipo_precio evaluando la relación entre precio y metro cuadrado de tres categorías: alto, medio y bajo.
- Dividimos la variable pxm2 en 3 intervalos utilizando 3 alternativas distintas:
  - tipo_precio_1 → ordenamos los valores y partimos el datset en 3 partes iguales
  - tipo_precio_2 → evaluamos según cuartiles y asignamos q1, q2, q3 a bajo, medio, alto respectivamente
  - tipo_precio_3 → como la técnica anterior pero agrupando por tipo de propiedad
- Estudiamos cómo se comportan las distribuciones de las variables tipo_precio_1, tipo_precio_2 y tipo_precio_3 y vimos que la mediana del precio es distinta según el tipo de propiedad
- Concluimos que la técnica más efectiva es la que utilizamos para tipo_precio_3
- Agrupamos por KMeans con 3 clusters y por el target y vemos resultados similares
  - Los PH se distribuyen casi iguales a lo largo de las 3 categorías
  - El grupo de precios altos abarca casi todo el mapa
  - Es mucho menor la cantidad de registros de precios bajos

b. Entrenamiento y predicción

- Eliminamos de train y test las variables de tipo_precio y cualquier otra que contenga información del precio de venta
- Dejamos como target tipo_precio_3
- Realizamos One Hot Encoding para las variables cualitativas
- Buscamos los mejores hiperparámetros con GridSearchCV en base al Accuracy porque es la métrica más intuitiva y queremos maximizar el porcentaje de observaciones

i. Modelo 1 → Árbol de Decisión

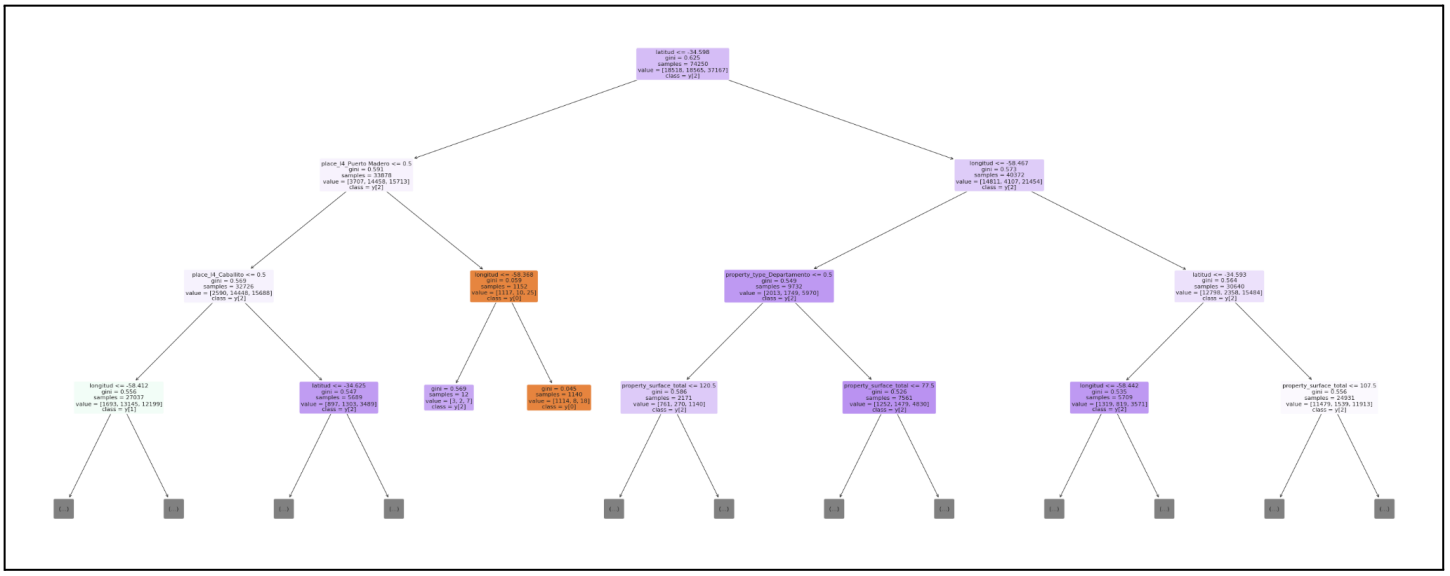
- Buscamos los mejores hiperparámetros:

```
param_grid = {
    'max_depth': [*range(21, 25, 1)],
    'criterion': ['gini', 'entropy'],
    'ccp_alpha': [0.00005, 0.0001, 0.0005]}

grid = GridSearchCV(tree.DecisionTreeClassifier(), param_grid, cv=5, scoring='accuracy')
grid.fit(x_train, y_train)
```

```
grid.best_params_
```

```
{'ccp_alpha': 0.0001, 'criterion': 'gini', 'max_depth': 21}
```

- Predecimos el set de entrenamiento y obtenemos:
  - Accuracy: 0.66
  - Precision: 0.68
  - Recall: 0.61
  - F1 Score: 0.63
- Predecimos el set de test y obtenemos:
  - Accuracy: 0.64
  - Precision: 0.66
  - Recall: 0.59
  - F1 Score: 0.61
- Vemos que las métricas dan muy similares
- Exportamos el árbol:



ii. Modelo 2 → Random Forest

- Buscamos los mejores hiperparámetros:

```
param_grid = {
    'n_estimators': [45, 55],
    'max_depth': [20, 25],
    'criterion': ['gini', 'entropy'],
    'ccp_alpha': [0.0001, 0.0005]}

grid = GridSearchCV(RandomForestClassifier(random_state=5), param_grid, cv=5, scoring='accuracy')
grid.fit(x_train, y_train)
```

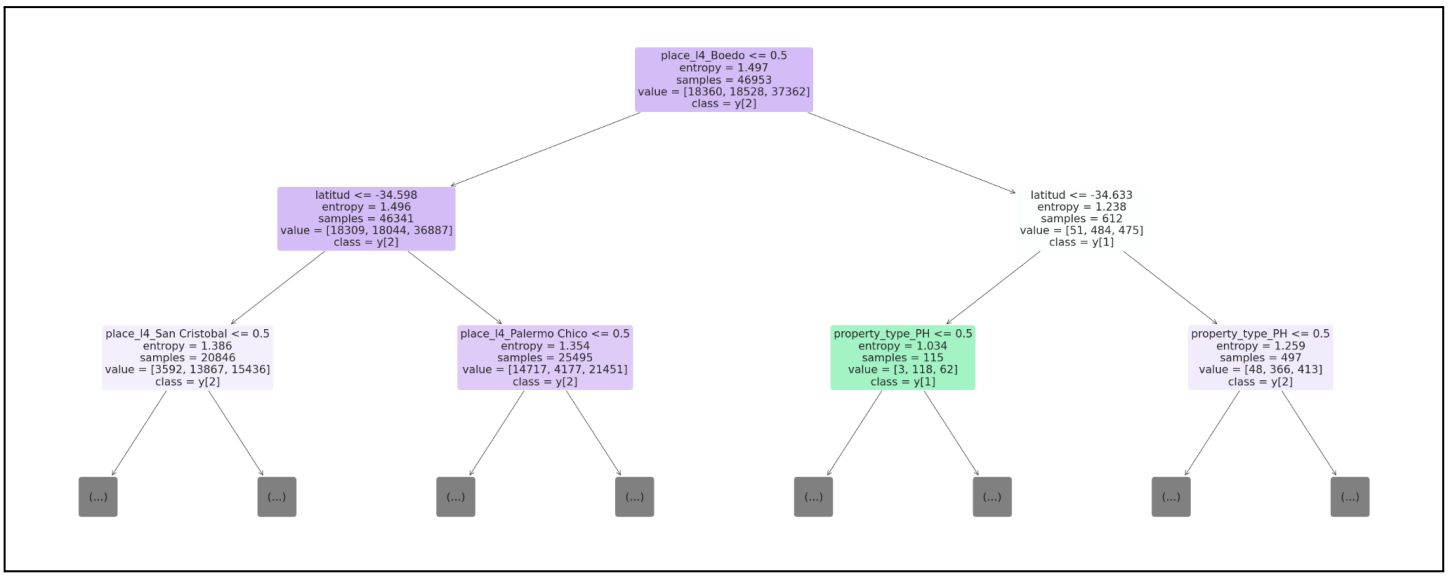
```
grid.best_params_
```

```
{'ccp_alpha': 0.0001,
 'criterion': 'entropy',
 'max_depth': 25,
 'n_estimators': 55}
```

- Predecimos el set de entrenamiento y obtenemos:
  - Accuracy: 0.75
  - Precision: 0.79
  - Recall: 0.71
  - F1 Score: 0.73
- Predecimos el set de test y obtenemos:
  - Accuracy: 0.68
  - Precision: 0.71
  - Recall: 0.63

- F1 Score: 0.65

- Vemos que las métricas dan muy similares
- Exportamos el árbol:



iii. Modelo 3 → K Nearest Neighbours

- Buscamos los mejores hiperparámetros:

```
param_grid = {
    'n_neighbors': [*range(2, 4, 1)],
    'algorithm': ['ball_tree', 'kd_tree', 'brute'],
    'weights': ['distance'],
    'n_jobs':[-1],
    'leaf_size':[*range(5, 10, 2)]}

grid = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5, scoring='accuracy')
grid.fit(x_train, y_train)
```

```
grid.best_params_
```

```
{'algorithm': 'kd_tree',
 'leaf_size': 5,
 'n_jobs': -1,
 'n_neighbors': 3,
 'weights': 'distance'}
```

- Predecimos el set de entrenamiento y obtenemos:
  - Accuracy: 1.0
  - Precision: 1.0
  - Recall: 1.0
  - F1 Score: 1.0
- Predecimos el set de test y obtenemos:
  - Accuracy: 0.48
  - Precision: 0.44
  - Recall: 0.44
  - F1 Score: 0.44

4. Regresión

a. Modelo 1 → KNN

- Entrenamos el dataset de train:
  - Accuracy: 0.193%
  - MSE: 72 billones

    - RMSE: 269 mil
    - Score: 0.424
- Entrenamos el dataset reducido:
    - Accuracy: 0.15%
    - MSE: 82 billones
    - RMSE: 286 mil
    - Score: 0.349
- Observamos que los modelos no son buenos predictores
- Optimizamos hiperparámetros usando RandomSearchCV con los parámetros:

```
params_grid = {'n_neighbors': range(1,100,25),
               'leaf_size': [*range(18, 23, 2)],
               'weights': ['distance','uniform'],
               'algorithm': ['ball_tree', 'kd_tree', 'brute'],
               'metric': ['euclidean','manhattan','chebyshev']}
```

- Buscamos para el dataset de train:

Calculamos para 5 iteraciones y 5 folds:

```
randomcv = random_cv_hyper_param_optimization(5, 5, KNeighborsRegressor(), params_grid, x_train, y_train)
```

Los mejores hiperparámetros encontrados son:

```
print(randomcv.best_params_)
```

```
{'weights': 'uniform', 'n_neighbors': 51, 'metric': 'chebyshev', 'leaf_size': 22, 'algorithm': 'ball_tree'}
```

- Entrenamos y obtenemos:
    - Accuracy: 0.0%
    - MSE: 70 billones
    - RMSE: 265 mil
    - Score: 0.441
- Buscamos para el dataset reducido:

```
randomcv = random_cv_hyper_param_optimization(5, 5, KNeighborsRegressor(), params_grid, x_train, y_train)
```

Los mejores hiperparámetros encontrados son:

```
print(randomcv.best_params_)
```

```
{'weights': 'uniform', 'n_neighbors': 76, 'metric': 'euclidean', 'leaf_size': 20, 'algorithm': 'ball_tree'}
```

- Entrenamos y obtenemos:
    - Accuracy: 0.0%
    - MSE: 78 billones
    - RMSE: 280 mil
    - Score: 0.376
- Vemos que no se obtienen buenos puntajes de predicción a pesar de que mejoraron luego de la optimización de hiperparámetros

b. Modelo 2 → XG Boost

- Entrenamos el dataset de train:
    - Accuracy: 0.0%
    - MSE: 53 billones

    - RMSE: 228 mil
    - Score: 0.424
- Entrenamos el dataset reducido:
    - Accuracy: 0.0%
    - MSE: 56 billones
    - RMSE: 237 mil
    - Score: 0.555
- Observamos que los modelos no son buenos predictores
- Optimizamos hiperparámetros usando RandomSearchCV con los parámetros:

```
params_grid = {'learning_rate': [0.05, 0.10, 0.15, 0.20, 0.25, 0.30 ],
               'max_depth': [3, 4, 5, 6, 8, 10, 12, 15],
               'min_child_weight': [1, 3, 5, 7],
               'gamma': [ 0.0, 0.1, 0.2 , 0.3, 0.4 ],
               'colsample_bytree' : [ 0.3, 0.4, 0.5 , 0.7 ]}
```

- Buscamos para el dataset de train:

Probemos con 5 iteraciones y 5 folds:

```
randomcv = random_cv_hyper_param_optimization(5, 5, XGBRegressor(), params_grid, x_train, y_train)
```

Los mejores hiperparámetros encontrados son:

```
print(randomcv.best_params_)
```

```
{'min_child_weight': 5, 'max_depth': 6, 'learning_rate': 0.3, 'gamma': 0.1, 'colsample_bytree': 0.3}
```

- Entrenamos y obtenemos:
    - Accuracy: 0.0%
    - MSE: 52 billones
    - RMSE: 229 mil
    - Score: 0.584
- Buscamos para el dataset reducido:

Probemos con 5 iteraciones y 5 folds:

```
randomcv = random_cv_hyper_param_optimization(5, 5, XGBRegressor(), params_grid, x_train, y_train)
```

Los mejores hiperparámetros encontrados son:

```
print(randomcv.best_params_)
```

```
{'min_child_weight': 3, 'max_depth': 10, 'learning_rate': 0.1, 'gamma': 0.4, 'colsample_bytree': 0.7}
```

- Entrenamos y obtenemos:
    - Accuracy: 0.0%
    - MSE: 51 billones
    - RMSE: 227 mil
    - Score: 0.591
- Observamos que el Score del modelo alcanza aproximadamente un 60% y resulta mejor la regresión con el dataset reducido que con el de train

c. Modelo 3 → AdaBoost

- Entrenamos el dataset de train:
    - Accuracy: 0.0%
    - MSE: 102 billones

    - RMSE: 319 mil
    - Score: 0.191
- Entrenamos el dataset reducido:
    - Accuracy: 0.0%
    - MSE: 785 billones
    - RMSE: 280 mil
    - Score: 0.378
- Observamos que los modelos no son buenos predictores
- Optimizamos hiperparámetros usando RandomSearchCV con los parámetros:

```
params_grid = {'n_estimators': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 20],
               'learning_rate': [(0.97 + x / 100) for x in range(0, 8)]}
```

- Buscamos para el dataset de train:

Probemos con 5 iteraciones y 5 folds:

```
randomcv = random_cv_hyper_param_optimization(5, 5, AdaBoostRegressor(), params_grid, x_train, y_train)
```

Los mejores hiperparámetros encontrados son:

```
print(randomcv.best_params_)
```

```
{'n_estimators': 10, 'learning_rate': 1.02}
```

- Entrenamos y obtenemos:
    - Accuracy: 0.0%
    - MSE: 65 billones
    - RMSE: 256 mil
    - Score: 0.481
- Buscamos para el dataset reducido:

Probemos con 5 iteraciones y 5 folds:

```
randomcv = random_cv_hyper_param_optimization(5, 5, AdaBoostRegressor(), params_grid, x_train, y_train)
```

Los mejores hiperparámetros encontrados son:

```
print(randomcv.best_params_)
```

```
{'n_estimators': 6, 'learning_rate': 1.0}
```

- Entrenamos y obtenemos:
    - Accuracy: 0.0%
    - MSE: 73 billones
    - RMSE: 271 mil
    - Score: 0.417
- Observamos que las estimaciones mejoran pero no alcanzan el Score de los dos previos → es el menos óptimo

TP2

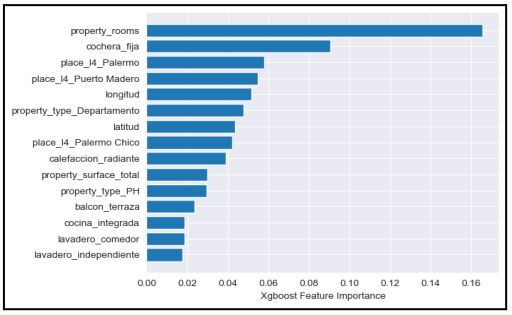
1. Procesamiento del Lenguaje Natural

a. Ampliación del Dataset

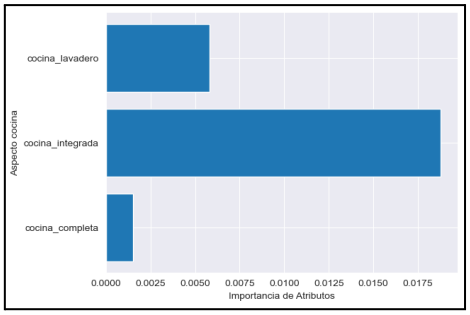
- Ampliamos el dataset utilizado para entrenamiento en el TP1 y adicionamos la columna property_description
- Estudiamos cuáles son las 100 palabras más comunes en la columna property_description y vemos que hay etiquetas html, letras en mayúsculas y minúsculas, con tilde, números, etc.
- Filtrando con ReGex, utilizamos técnicas de reducción de la dimensionalidad:
    - eliminamos etiquetas html
    - transformamos las palabras a minúsculas
    - quitamos los tildes
    - eliminamos símbolos
    - eliminamos espacios múltiples
    - eliminamos palabras sin significado (stop words)
- Nuevamente estudiamos cuáles son las 100 palabras más comunes y seleccionamos las que consideramos relevantes: cocina, pisos, calefaccion, expensas, lavadero, balcon, cochera y aire
- Estudiamos cuáles son las palabras más comunes para cada uno de estos aspectos y seleccionamos los que consideramos como relevantes
- Agregamos al dataset de train cada uno de los aspectos como columnas y como valores el que se encontró para el aspecto en cuestión

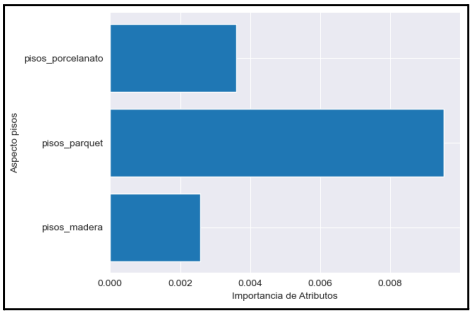
b. Modelos

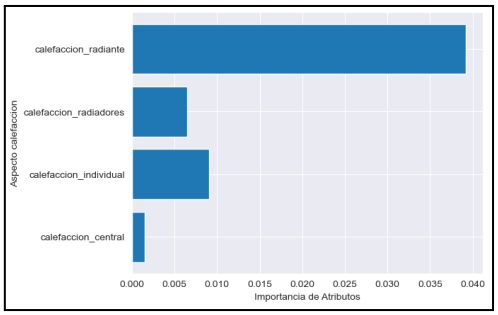
- Entrenamos un modelo XG Boost para regresión con el nuevo dataset ampliado utilizando los mismos hiperparámetros que TP1:
    - ➢ min_child_weight = 5
    - ➢ max_depth = 6
    - ➢ learning_rate = 0.3
    - ➢ gamma = 0.1
    - ➢ colsample_bytree = 0.3
    - ■ MSE: 53 billones
    - ■ RMSE: 230 mil
    - ■ Score: 58.0%
- Estudiamos la importancia de features:

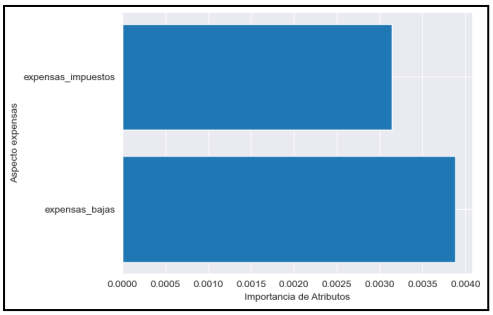


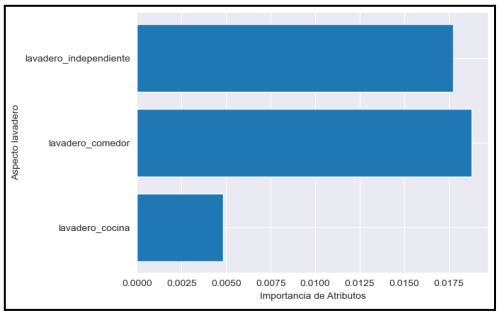
- Estudiamos la importancia de features por aspecto:

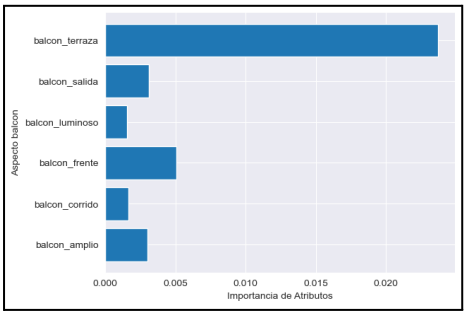


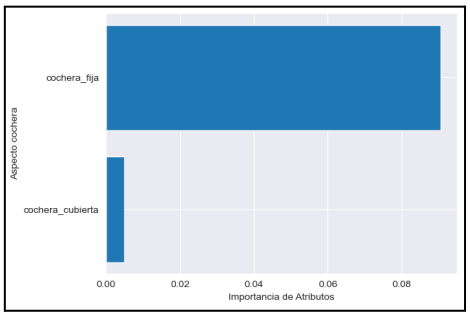


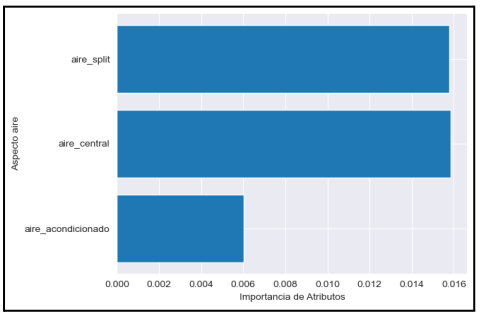










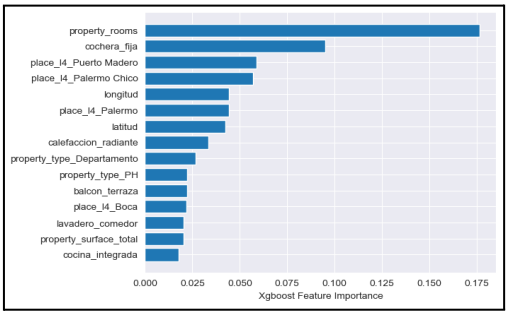


- Realizamos una optimización de hiperparámetros:

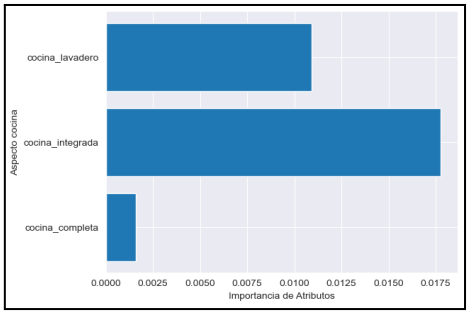
```
params_grid = {'learning_rate': [0.20, 0.25, 0.30],
               'max_depth': [4, 6, 8, 10],
               'min_child_weight': [1, 3, 5, 7, 9],
               'gamma': [0.1, 0.3, 0.5],
               'colsample_bytree' : [0.3, 0.5, 0.7]}

randomCV = RandomizedSearchCV(estimator = XGBRegressor(),
                              param_distributions = params_grid,
                              scoring = make_scorer(r2_score),
                              cv = 5,
                              n_iter = 20)

randomCV.fit(df_dummies, df_train_y_regresion)
```

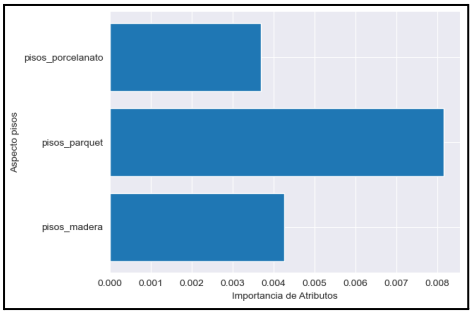
  - ➢ min_child_weight: 1
  - ➢ max_depth: 8
  - ➢ learning_rate: 0.3
  - ➢ gamma: 0.1
  - ➢ colsample_bytree: 0.3
  - MSE: 53 billones
  - RMSE: 230 mil

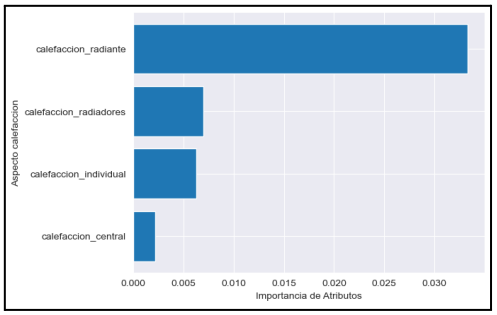
    - Score: 57.9%
- Estudiamos la importancia de features:

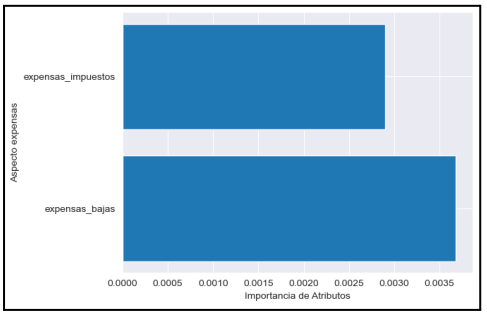


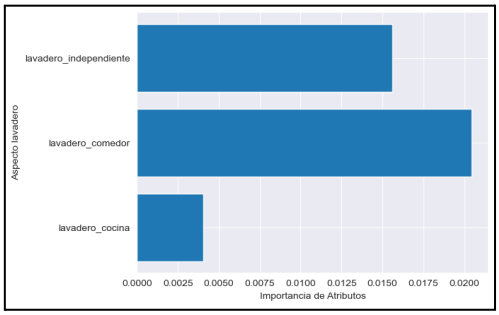
- Estudiamos la importancia de features por aspecto:

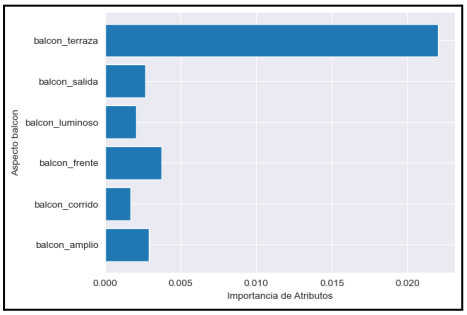


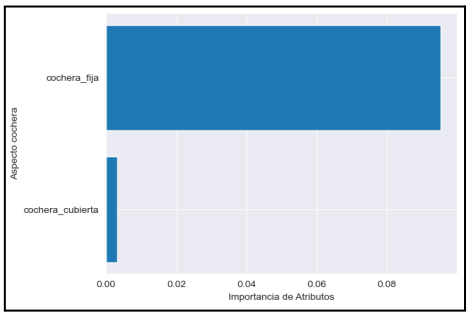


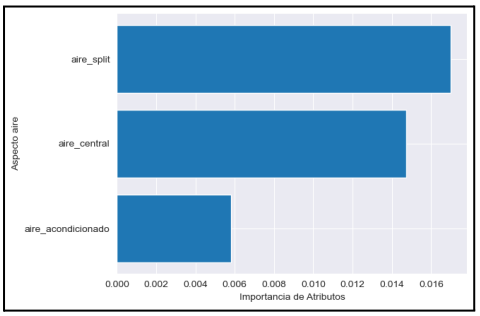












2. Redes Neuronales

a. Regresión

- Sacamos columnas de ids
- Normalizamos todos los valores con StandardScaler
- Buscamos el mejor modelo con GridSearchCV:

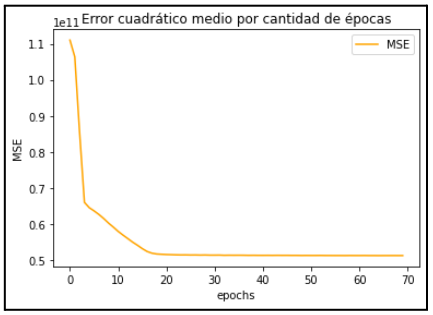
```
param_grid = dict(
    hidden_layers=[2, 4],
    first_layer_nodes = [x_train_regresion.shape[1], math.ceil(cantidad_de_columnas * (2/3))],
    last_layer_nodes = [5],
    activation_func = ['sigmoid', 'relu'],
    batch_size = [750],
    epochs = [70],
    optimizer=['RMSprop'],
)

grid = GridSearchCV(
    estimator = modelo,
    param_grid = param_grid,
    cv=5,
    error_score='raise',
    scoring='neg_mean_squared_error',
    verbose=0,
)
```

- activation_func: relu
- batch_size: 750
- epochs: 70
- first_layer_nodes: 69
- hidden_layers: 4
- last_layer_nodes: 5
- optimizer: RMSprop

- Error absoluto promedio: 65.5%
- Error absoluto cuadrado promedio: 51 millones

i. Predecir precio de la propiedad

- Predecimos el dataset de train y obtenemos:



- MSE: 107 millones
- RMSE: 327 mil

b. Clasificación

- Sacamos columnas de ids
- Normalizamos todos los valores con Z Score
- Aplicamos One Hot Encoding a la columna target para que tenga 3 y coincida con el output del modelo
- Buscamos el mejor modelo con GridSearchCV:

Creamos un modelo base, necesario según la documentación de SciKeras - KerasClassifier. Este modelo base será editado por GridSearchCV al buscar los hiperparámetros.

```
base_model = KerasClassifier(
    crear_modelo,
    loss_func="categorical_crossentropy",
    extra_hidden_layers=1,
    last_layer_nodes=1,
    activation_func='sigmoid',
)
```

Definimos los hiperparámetros posibles. Además de los necesarios por la función crear_modelo, agregamos diferentes optimizadores, learning rates y cantidad de épocas.

```
param_grid = dict(
    extra_hidden_layers=[1, 4],
    last_layer_nodes = [5, 10, 20],
    activation_func = ['relu', 'softmax'],
    batch_size = [750],
    epochs = [20],
    optimizer__learning_rate = [0.001],
    optimizer = ["adam", "sge"],
    loss_func = ["categorical_crossentropy"],
)

gs = GridSearchCV(base_model,
                  param_grid = param_grid,
                  cv=5,
                  scoring=make_scorer(my_categorical_accuracy),
                  verbose=0,
                  error_score='raise'
)
```

- Obtenemos un modelo secuencial de 4 capas densas

i. Predecir tipo precio

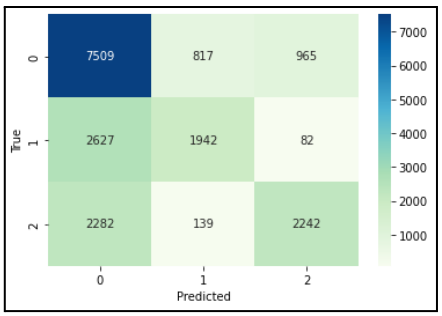
- Entrenamos el dataset de train y obtenemos:
  - Accuracy: 0.57
  - Precision: 0.58
  - Recall: 0.5
  - F1 Score: 0.51

3. Ensamble de Modelos Híbridos

- Preparamos la variable target bajo la misma convención que para el TP1 → pxm2 agrupando por tipo de propiedad en 3 intervalos, 25% a bajo, 50% a medio y el otro 25% restante a alto

a. Ensamble 1 (Clasificación) → Voting

- Usamos el dataset reducido del TP1
- Usamos el tipo de votación hard (regla de la mayoría)
- Definimos el ensamble con los mismos empleados en el TP1:
  - Árbol de decisión
    - ccp_alpha: 0.0001
    - criterion: gini
    - max_depth: 21
  - Random forest
    - ccp_alpha: 0.0001
    - criterion: entropy
    - max_depth: 25
    - n_estimators: 55
  - KNN

        - algorithm: kd_tree
        - leaf_size: 5
        - n_jobs: -1
        - n_neighbors: 3
        - weights: distance
- Entrenamos, predecimos y obtenemos:
    - Accuracy: 0.63
    - Precision: 0.65
    - Recall: 0.57
    - F1 Score: 0.59
    - 

b. Ensamble 2 (Regresión) → Stacking

- Usamos el dataset reducido del TP1
- Modelos base:
    - KNN
        - weights: uniform
        - n_neighbors: 51
        - metric: chebyshev
        - leaf_size: 22
        - algorithm: ball_tree
    - XGBoost
        - min_child_weight: 5
        - max_depth: 6
        - learning_rate: 0.3
        - gamma: 0.1
        - colsample_bytree: 0.3
    - Adaboost
        - n_estimators: 10
        - learning_rate: 1.02
- Meta-modelo: Gradient Boosting
- Entrenamos, predecimos y obtenemos:
    - MSE: 74 millones
    - RMSE: 272 mil
    - R2 Score: 0.410
- Vemos que el MSE se redujo a la mitad que en el TP1

Conclusiones Métricas

Clasificación					
		Accuracy	Precision	Recall	F1 Score
TP1	Decision Tree Classifier	0,64	0,66	0,59	0,61
	Random Forest	0,68	0,71	0,63	0,65
	K Nearest Neighbours	0,48	0,44	0,44	0,44
TP2	Redes Neuronales	0,57	0,58	0,5	0,51
	Ensamble Voting	0,63	0,65	0,57	0,59

Regresión					
			MSE	RMSE	Score
TP1	K Nearest Neighbours	Train	72.776.601.006	269.771	0,42
		Reduc	82.249.853.450	286.792	0,35
		Train v2	70.586.895.109	265.681	0,44
		Reduc v2	78.861.262.549	280.822	0,38
	XG Boost	Train	52.361.303.578	228.825	0,59
		Reduc	56.201.046.301	237.067	0,56
		Train v2	52.506.772.528	229.143	0,58
		Reduc v2	51.650.994.876	227.268	0,59
	Ada Boost	Train	102.158.666.520	319.622	0,19
		Reduc	78.579.238.461	280.319	0,38
		Train v2	65.537.678.112	256.003	0,48
		Reduc v2	73.694.782.959	271.467	0,42
TP2	Redes Neuronales	Train	107.144.176.406	327.328	
	Ensamble Stacking	Train	74.468.989.103	272.890	0,41