Machine Learning Competitivo

Autor: Daniel Pereda

30 Septiembre 2021

Outline

- Competencia y resultados.
- Modelos más dominantes en datos tabulares (estructurados, bases relacionales, etc).
- Un poco de EDA
- Técnicas y librerías comúnmente utilizadas:
  - Validación adversarial.
  - Rapids
    - cuML: RandomForest, KMeans, T-NSE y UMAP.
    - cuDF: Básicamente Dask en GPU.
    - Entre otros.
  - Selección de features:
    - Leave One Feature Out (LOFO).
    - Añadir ruido.
  - Optimización de hiperparámetros:
    - Optuna.
  - Ensembling: (ver más en el repo)
    - Promediar semillas.
    - Stacking.
  - StratifiedKfold basado en clusters

Competencia: 30 Days of ML

- Datos tabulares:
  - 10 features categóricas.
  - 13 features continuas.
  - Target continuo.
  - No missing data.
  - 500.000 filas
- Métrica: RMSE.
- Leaderboard:
  - La diferencia entre el primer lugar y top 100 es de ~ 0.025 %.
  - La diferencia con alguien en el top 10%, es de ~ 0.27 %.

#	Team Name	Notebook	Team Members	Score
1	Kaggle Swags			0.71694
2	DDR			0.71695
	Best Entry ↑ submission scored 0.71694, which is not an improvement of your best score. Keep trying!			
3	Overfitting Public AND Private			0.71695
4	Ali Akyurek			0.71700
5	Neo Zhao			0.71701

LB score: Equipo con Rodrigo Assar y Diego Ramírez

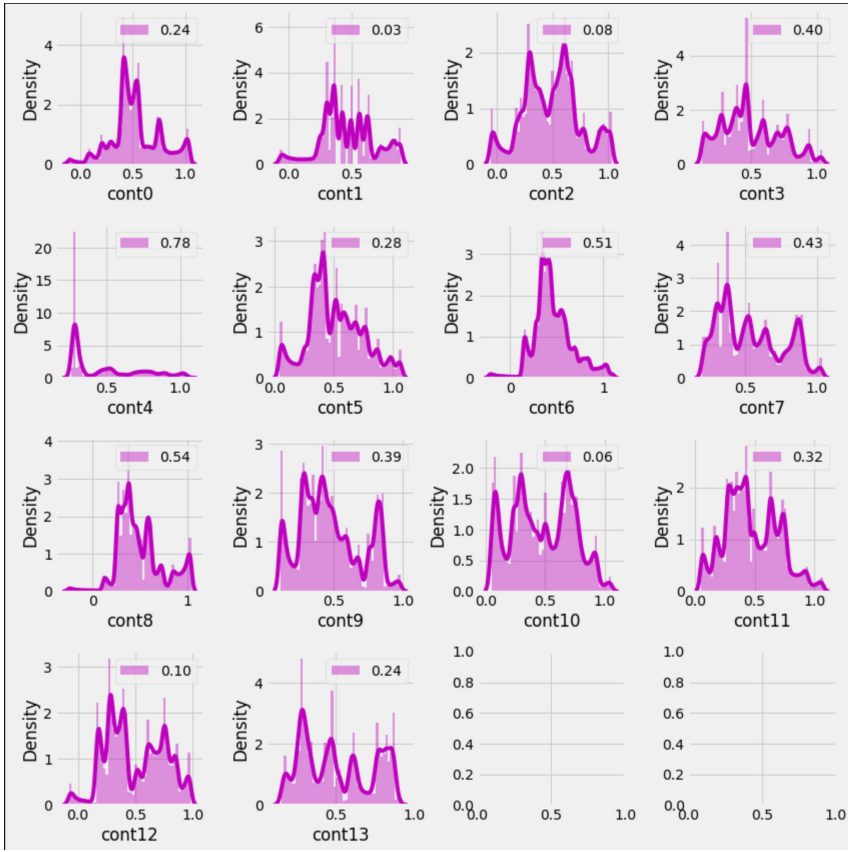
Modelos dominantes datos estructurados

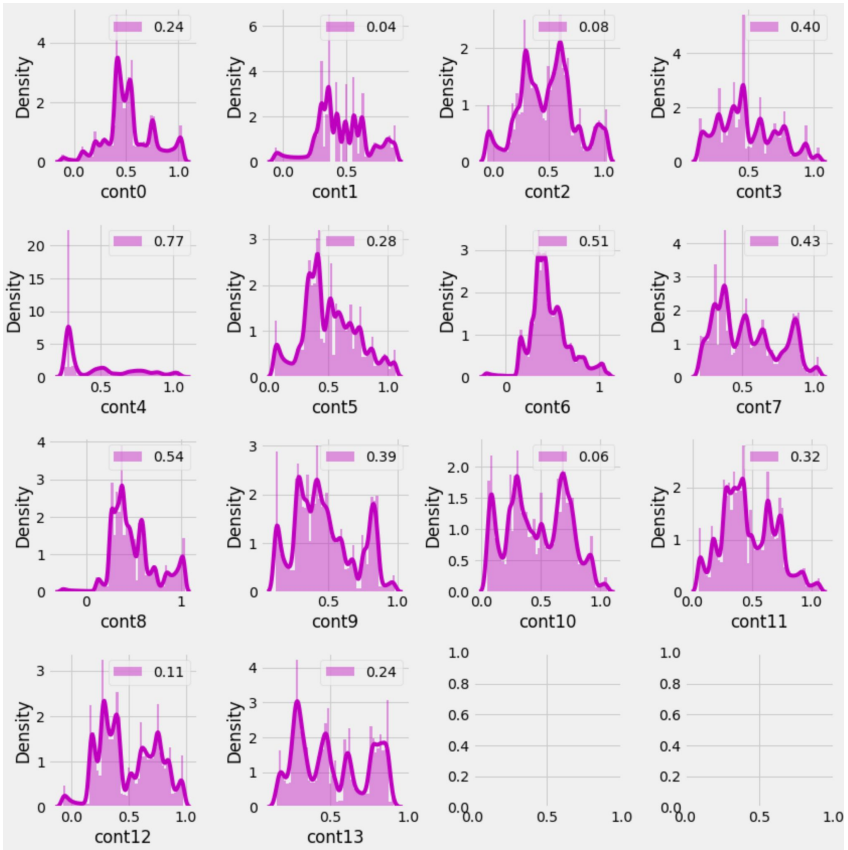
- Basados en gradient boosting: Una característica común es que no requieren imputación de datos, eliminación de outliers ni estandarización.
  - LightGBM:
    - Funciona por defecto con variables categóricas.
    - Muy rápido, incluso en CPU.
  - XGBoost:
    - Debemos hacer encoding de variables categóricas.
    - Rápido en GPU.
  - CatBoost:
    - Funciona por defecto con variables categóricas.
    - Más rápido que XGBoost, más lento que LightGBM.
- DeepLearning:
  - TabNet (Diciembre 2020): Mecanismo de atención secuencial, el cual selecciona features en cada iteración (ganando interpretabilidad). *Ver paper*

Name	Rossman	CoverType	Higgs	Gas	Eye	Gesture	YearPrediction	MSLR	Epsilon	Shrutime	Blastchar
XGBoost	490.18	3.13	21.62	2.18	**56.07**	80.64	77.98	55.43	11.12	13.82	20.39
NODE	488.59	4.15	21.19	2.17	68.35	92.12	76.39	55.72	**10.39**	14.61	21.40
DNF-Net	503.83	3.96	23.68	**1.44**	68.38	86.98	81.21	56.83	12.23	16.80	27.91
TabNet	**485.12**	3.01	**21.14**	1.92	67.13	96.42	83.19	56.04	11.92	14.94	23.72
1D-CNN	493.81	3.51	22.33	1.79	67.90	97.89	78.94	55.97	11.08	15.31	24.68
Simple Ensemble	488.57	3.19	22.46	2.36	58.72	89.45	78.01	55.46	11.07	13.61	21.18
Deep Ensemble w/o XGBoost	489.94	3.52	22.41	1.98	69.28	93.50	78.99	55.59	10.95	14.69	24.25
Deep Ensemble w XGBoost	485.33	**2.99**	22.34	1.69	59.43	**78.93**	**76.19**	**55.38**	11.18	**13.10**	**20.18**
		TabNet			DNF-Net		NODE			New datasets	

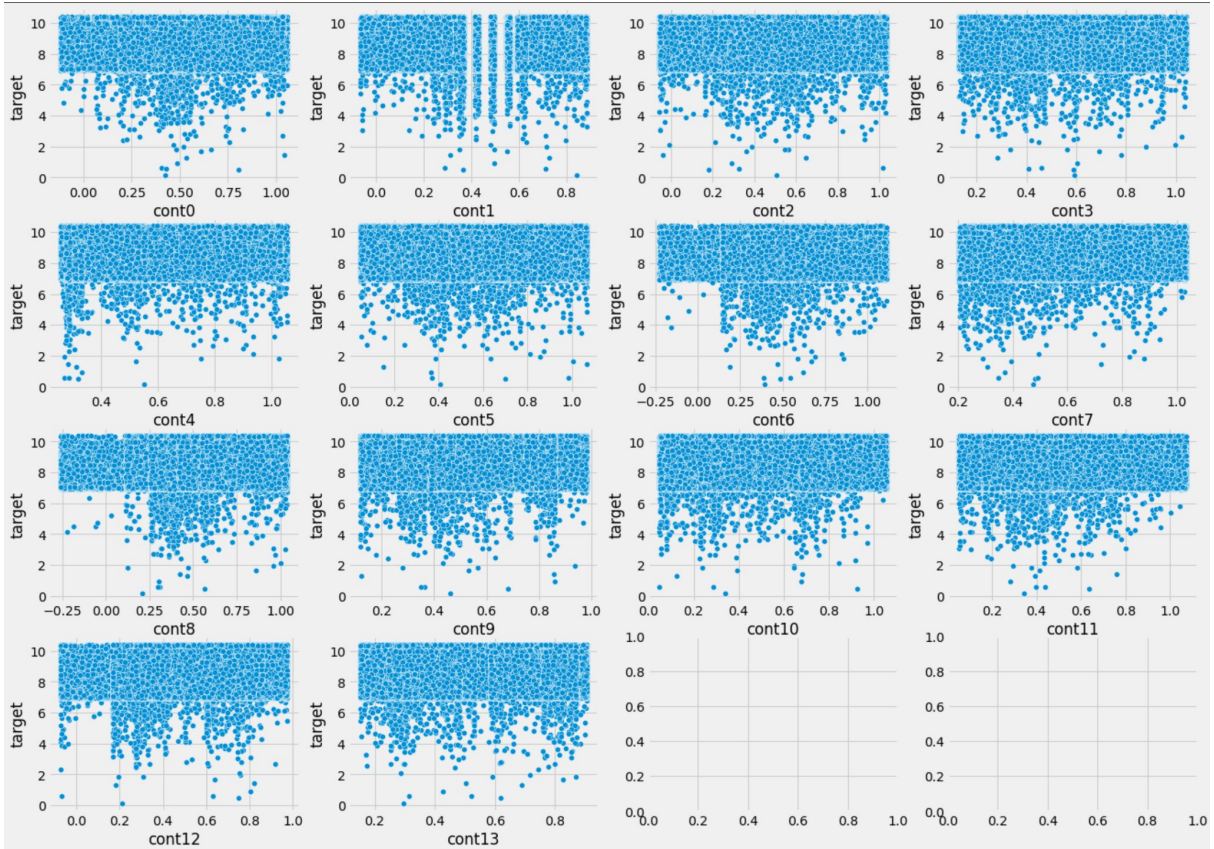
Tabular Data: Deep Learning is not all you need! (*preprint: Junio 2021*)

EDA: Features numéricas

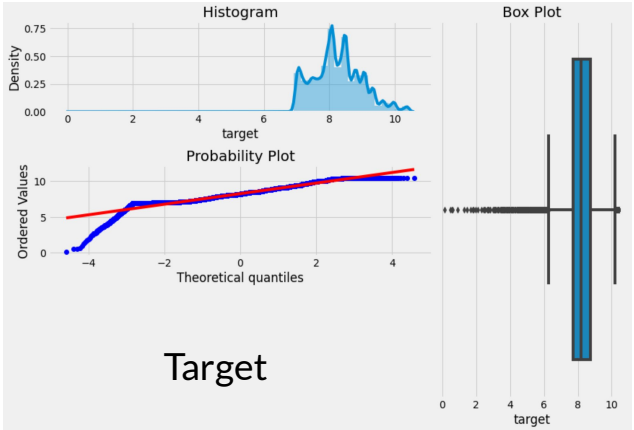


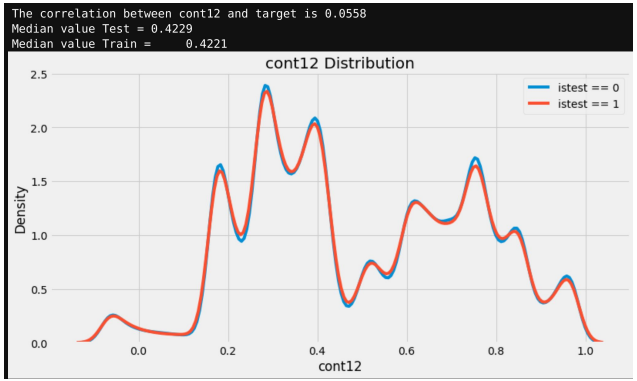


EDA: Features numéricas



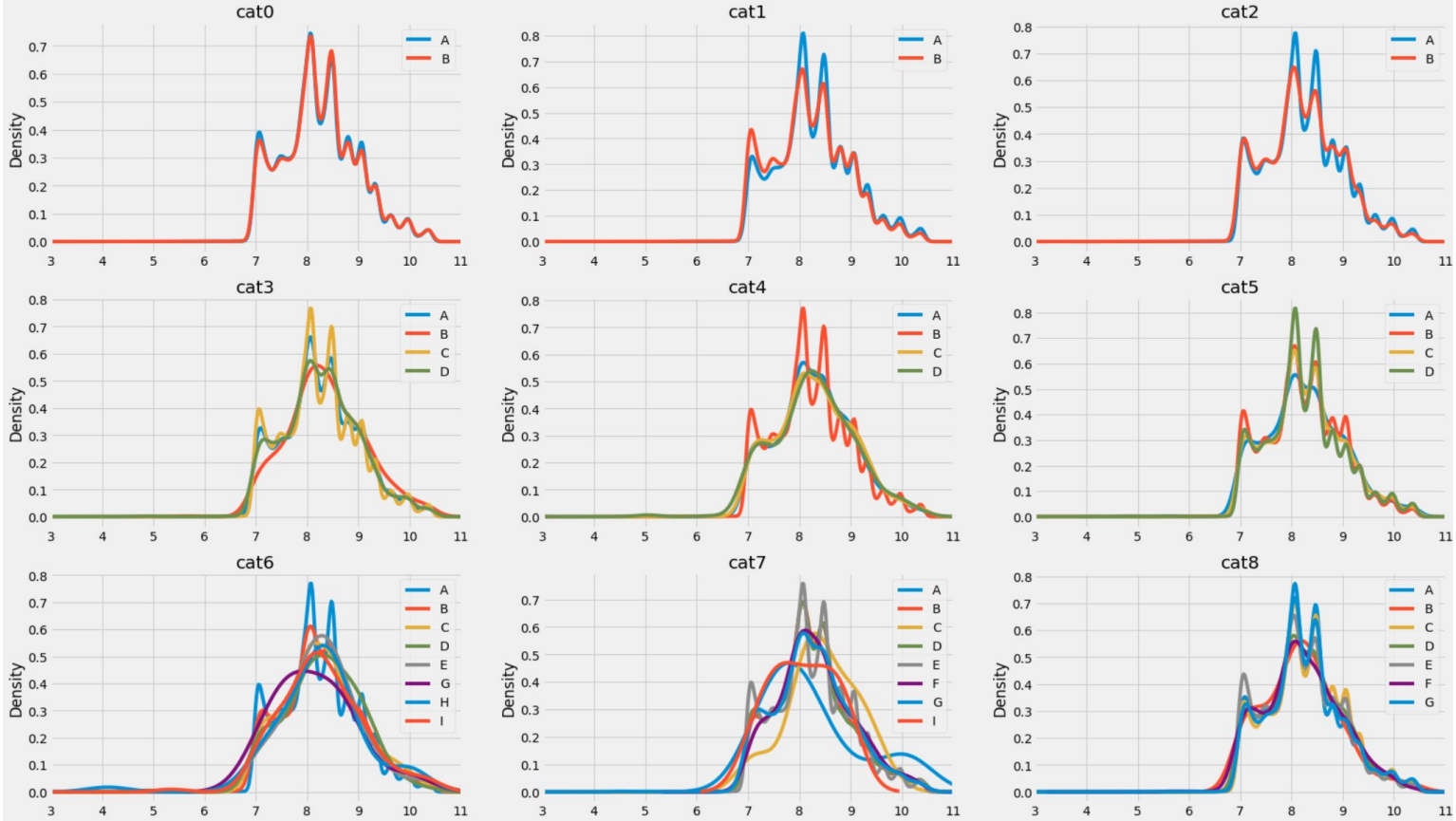
Scatter plots variables continuas y target





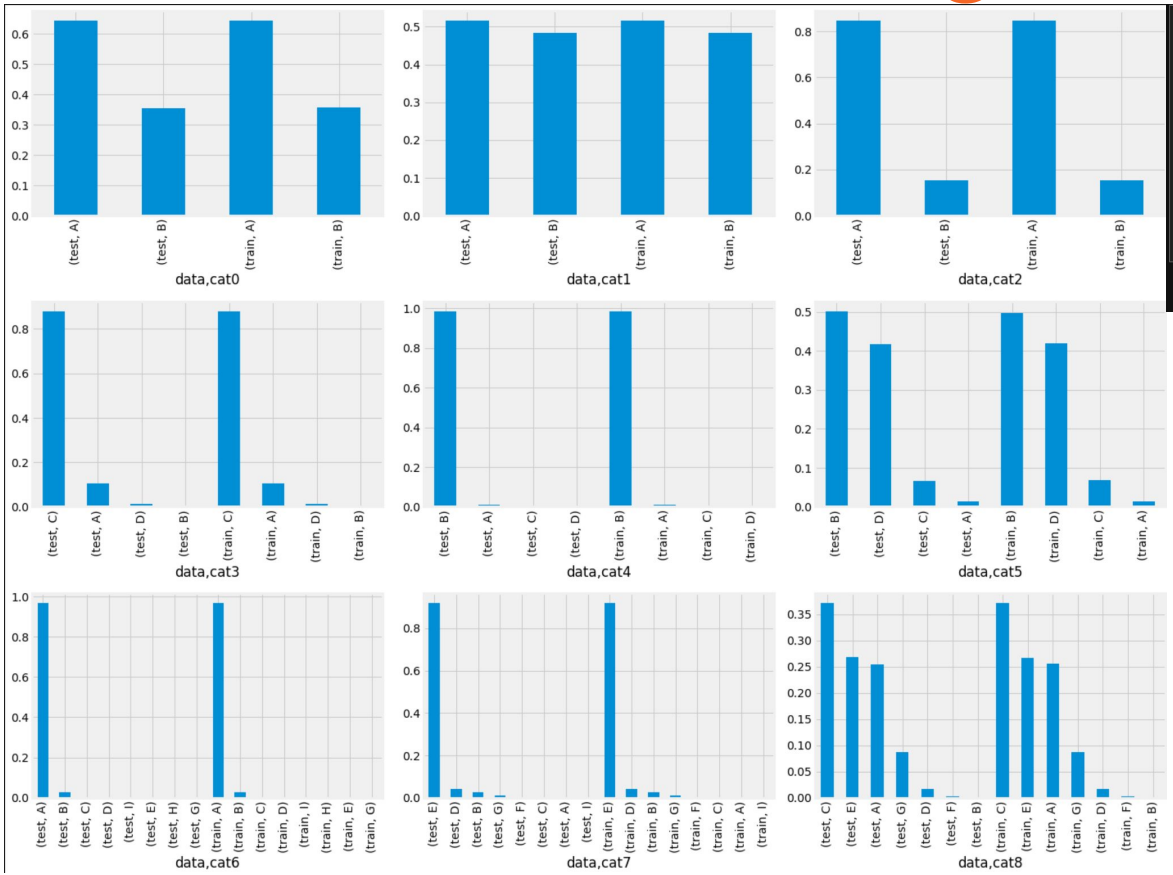
Comparar distribuciones en train y test

EDA: Features categóricas



Target kde por categoría: Intuición para crear features basadas en encodings

EDA: Features categóricas



Porcentaje categoría en train y test set

```
%%time

train["data"] = "train"
test["data"] = "test"
X = pd.concat([train[cat_cols + ["data"]],test[cat_cols + ["data"]]] )
fig, axes = plt.subplots(nrows=3, ncols = 3)
for i, cat_col in enumerate(cat_cols[:-1]):
    X.groupby("data")[cat_col].value_counts(
        dropna=False, normalize=True
    ).plot(kind="bar", ax=axes[i // 3, i % 3]
    );
```

```
CPU times: user 925 ms, sys: 22.2 ms, total: 947 ms
Wall time: 946 ms
```

Código que genera los gráficos

```
train.drop_duplicates(subset = cat_cols)
```

	cat0	cat1	cat2	cat3	cat4	cat5	cat6	cat7	cat8	cat9	...
0	B	B	B	C	B	B	A	E	C	N	...
1	B	B	A	A	B	D	A	F	A	O	...
2	A	A	A	C	B	D	A	D	A	F	...
3	B	B	A	C	B	D	A	E	C	K	...
4	A	A	A	C	B	D	A	E	A	N	...
...	...	...	...	...	...	...	...	...	...	...	...
299821	B	B	A	C	A	B	A	E	A	K	...
299843	A	A	A	C	A	D	A	B	A	L	...
299855	A	B	B	A	B	D	A	B	E	F	...
299901	B	A	A	C	A	A	A	E	C	H	...
299981	B	A	B	A	B	D	A	D	E	H	...

14782 rows × 26 columns

Solo ocurren 14.782 combinaciones de un total de 3.440.640

Validación adversarial

- La idea es comparar el nivel de similitud entre train y test en términos de la distribución de las features:
  - Si son difíciles de distinguir, entonces probablemente las distribuciones son similares y técnicas de validación usuales pueden ser aplicadas.
  - Si no, debemos buscar la(s) features que nos están dando problema(s) y arreglarlas (esto normalmente es complejo) o eliminarlas.
- También se puede utilizar después de la creación de features, para identificar de forma rápida si tenemos algún leakage o feature no útil.
- Utilizando XGBoost en GPU obtuvimos los siguientes resultados:

```
xgb_params = {"max_depth":5,
              "objective":'binary:logistic',
                "n_estimators": 1000,
                'booster': 'gbtree',
                 'tree_method':"gpu_hist",
                'gpu_id' : 0,
                'predictor': "gpu_predictor"}
```

```
0 0.59813 0.5002176628588649
1 0.60049 0.49997179401884984
2 0.59888 0.49991059269978066
3 0.58802 0.5003591266921229
4 0.60231 0.50014734112786
0.5975659999999999 0.5001213034794956
Dummy: 0.499526 0.49958379847011913
CPU times: user 29.1 s, sys: 611 ms, total: 29.7 s
Wall time: 27.6 s
```

- *Accuracy: 0.6*
- *ROC_AUC: 0.5*

Validación adversarial

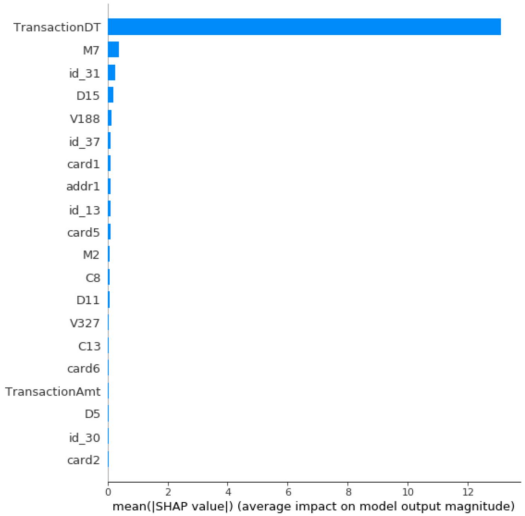
En términos de ML esto se resume a lo siguiente:

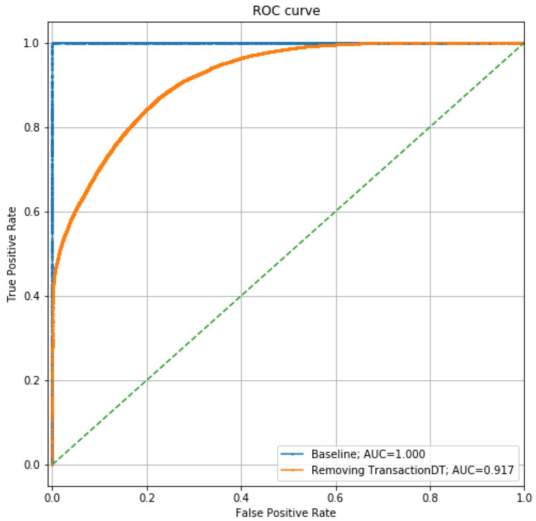
- Se elimina la columna *target* en *train*.
- Agregar feature binaria {0,1} que indica si la fila viene de train o test.
- *train[“istrain”] = 1* y *test[“istrain”] = 0*
- Se concatenan ambos datasets y genera train/test split.
- Se entrena un modelo de clasificación optimizando métrica *roc_auc*.
- Es bueno hacer un esquema de validación de tipo *kfold* o *StratifiedKfold* utilizando *“istrain?”*, esto último asegura que preservemos un porcentaje de muestras de ambas clases similar en cada fold.

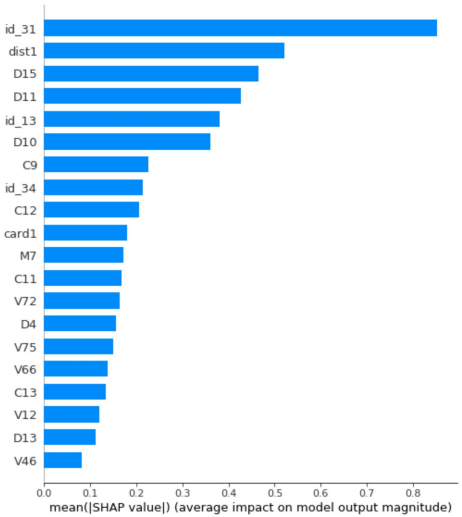
```
%%time

acc = []
auc = []
dummy_acc = []
dummy_auc = []
model_fi = []

for fold in range(5):
    xtrain =  df_all[df_all.kfold != fold].reset_index(drop=True)
    xvalid = df_all[df_all.kfold == fold].reset_index(drop=True)

    ytrain = xtrain.train
    yvalid = xvalid.train

    xtrain = xtrain[useful_features]
    xvalid = xvalid[useful_features]

    encoder = ce.OrdinalEncoder()
    xtrain = encoder.fit_transform(xtrain, ytrain)
    xvalid = encoder.transform(xvalid)

    model = XGBClassifier(**xgb_params, random_state = 0)
    model.fit(xtrain, ytrain, verbose=False,
              eval_set=[(xvalid, yvalid)],
              eval_metric = "auc",
              early_stopping_rounds=300
              )
    model_fi.append(model.feature_importances_)

    dummy_clf = DummyClassifier(strategy="uniform")
    dummy_clf.fit(xtrain, ytrain)

    preds_valid = model.predict(xvalid, ntree_limit = model.get_booster().best_ntree_limit)
    preds_dummy = dummy_clf.predict(xvalid)
    fold_acc = accuracy_score(yvalid, preds_valid)
    fold_auc = roc_auc_score(yvalid, preds_valid)
    acc.append(fold_acc)
    auc.append(fold_auc)
    dummy_acc.append(accuracy_score(yvalid, preds_dummy))
    dummy_auc.append(roc_auc_score(yvalid, preds_dummy))
    print(fold, fold_acc, fold_auc)
```

Cómo se vería el mal caso

- Problema: Serie de tiempo en donde test set está en el futuro
  - La feature día de transacción (*TransactionDT*) está primera en feature importances y shap values.
  - Eliminamos e iteramos otra vez.





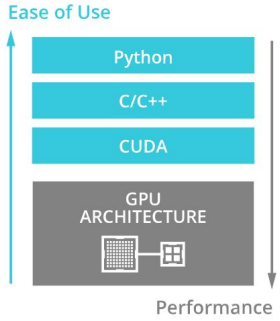


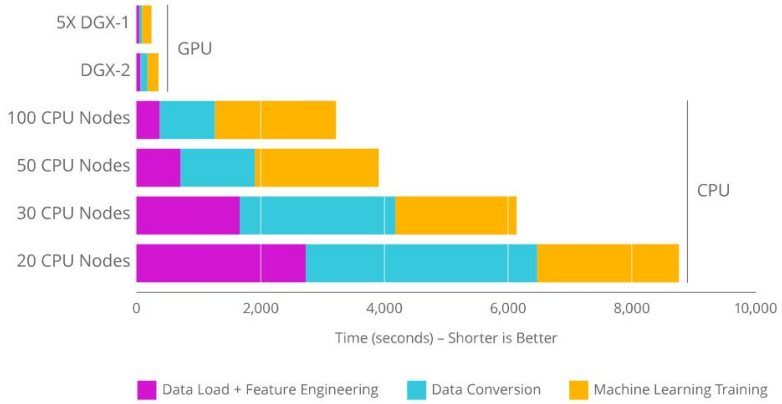
Fuente: https://towardsdatascience.com/adversarial-validation-ca69303543cd
Ver también: [How to use Adversarial Validation to Help Fix Overfitting]

NVIDIA RAPIDS

- cuML:
  - ¿Sklearn en GPU?
  - Ver algoritmos en github
- cuDF:
  - Dask en GPU Ver 10 min guide to cuDF
  - Competencia en Kaggle Accelerating Trading on GPU

```
import cudf

gdf = cudf.read_csv('path/to/file.csv')
for column in gdf.columns:
    print(gdf[column].mean())
```



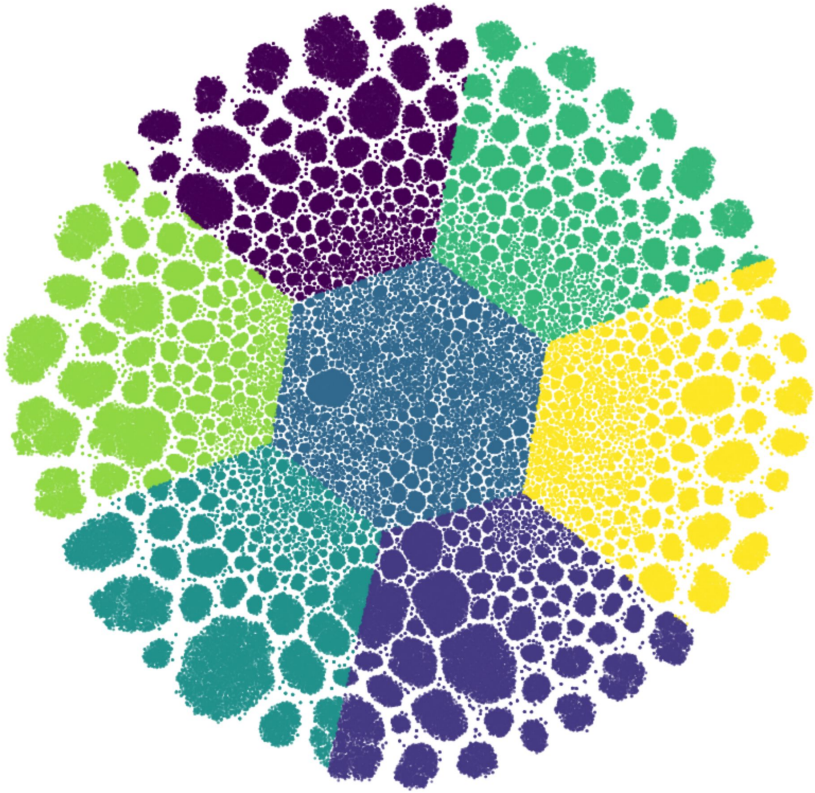


DGX-2 Corresponden a 16 NVIDIA V100 (2 petaFLOPS)

Ver poder de computo de distintas tarjetas

t-Distributed Stochastic Neighbor Embedding + KMeans

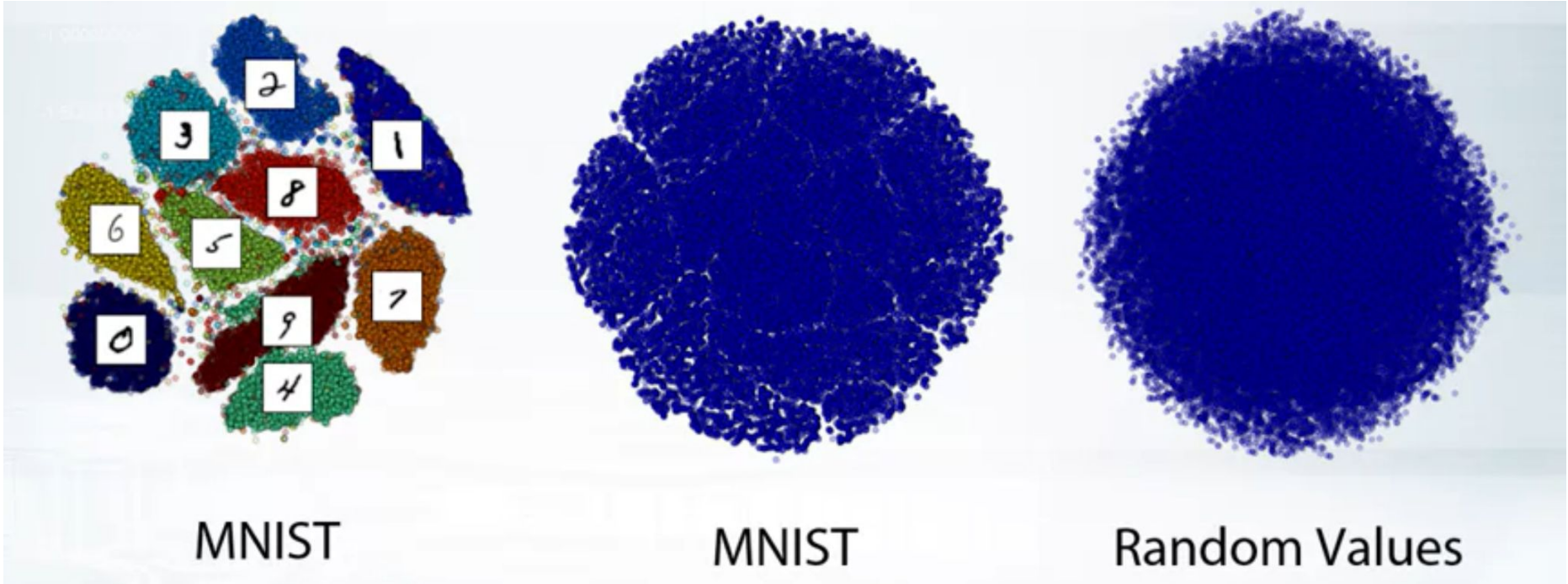
```
%%time

tsne = TSNE(n_components=2, perplexity=10, n_neighbors=100)
projection_2D = tsne.fit_transform(H)

kmeans = KMeans(n_clusters=7)
kmeans.fit(projection_2D)
labels = kmeans.labels_

plt.figure(figsize=(15, 15))
plt.scatter(projection_2D[:,0], projection_2D[:,1],
            c=labels,
            edgecolor='none',
            alpha=0.80,
            s=y_stratified.values)
```



- Técnica de reducción de dimensionalidad no lineal.
- Busca *similitudes* entre puntos en un espacio de alta dimensionalidad. (para esto utiliza distribuciones Gaussianas -> parámetro *perplexity*)
- [Página interactiva para aprender sobre TSNE]

```
CPU times: user 20.6 s, sys: 10.2 s, total: 30.8 s
Wall time: 30.8 s
```

TSNE en MNIST dataset



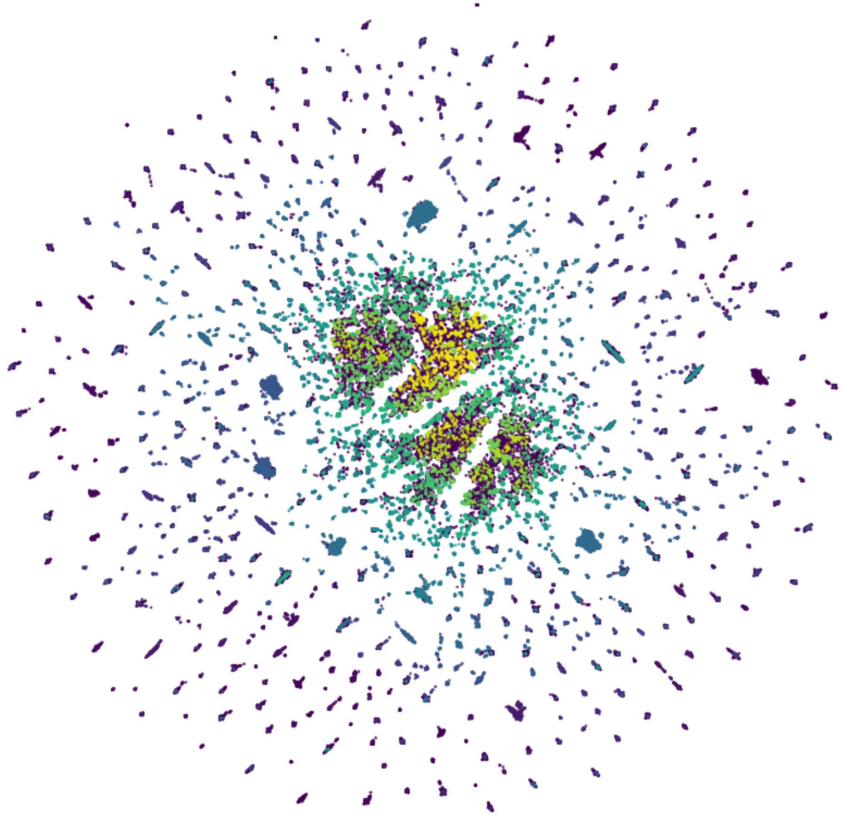
Efecto en mala elección del parámetro *perplexity*, en la izquierda es tomando el valor 150 y en el centro 3

Warning: Si tenemos más de 500 features, es mejor utilizar PCA o SVD para reducirlas y luego TSNE

Uniform Manifold Approximation and Projection+ HDBSCAN

```
from cuml.manifold import UMAP
# from cuml.experimental.cluster import HDBSCAN
from hdbscan import HDBSCAN

umap = UMAP(n_components=2, n_neighbors=30, min_dist= 0.2)
projection_2Dumap = umap.fit_transform(H)

hdbscan = HDBSCAN()
hdbscan.fit(projection_2Dumap)

labels = hdbscan.labels_

plt.figure(figsize=(15, 15))
plt.scatter(projection_2Dumap[:,0], projection_2Dumap[:,1],
            c=labels,
            edgecolor='none',
            alpha=0.80,
            s=y_stratified.values)
```

- Se construye una representación gráfica de alta dimensión de los datos y luego se optimiza una representación de baja dimensión con una estructura lo más similar posible.
- Similar a TSNE, pero utiliza un par de “trucos” para hacer el proceso más rápido
- n_neighbors: parámetro más importante
  - Bajo: Concentrarse en estructura local
  - Alto: Concentrarse en estructura global



```
CPU times: user 16.3 s, sys: 1.74 s, total: 18 s
Wall time: 19.1 s
```

[Link Libreria HDBSCAN]

TSNE vs UMAP

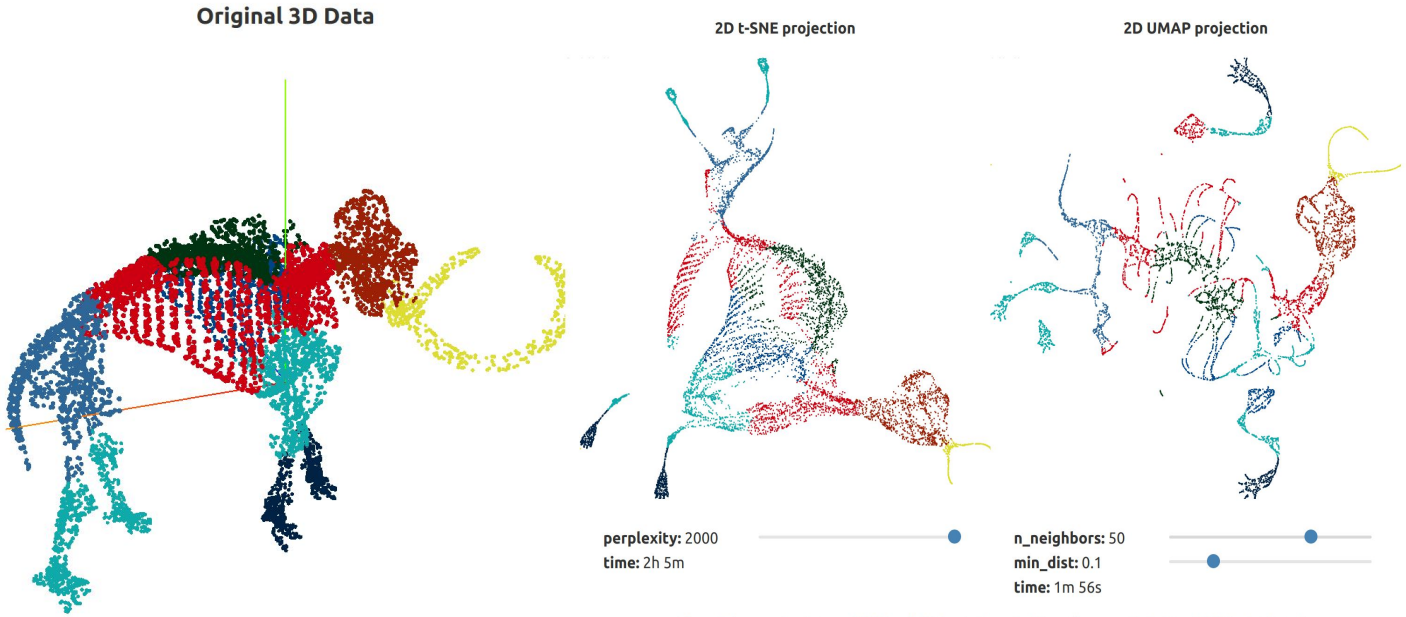


Figure 6: A comparison between UMAP and t-SNE projections of a 3D woolly mammoth skeleton (50,000 points) into 2 dimensions, with various settings for parameters. Notice how much more global structure is preserved with UMAP, particularly with larger values of `n_neighbors`.

Usualmente UMAP es mejor en preservar la estructura global en la proyección final.

Esto significa que la distancia intercluster son potencialmente más significativas que en TSNE.

Notar la diferencia en tiempos de ejecución ~60 veces más rápido!

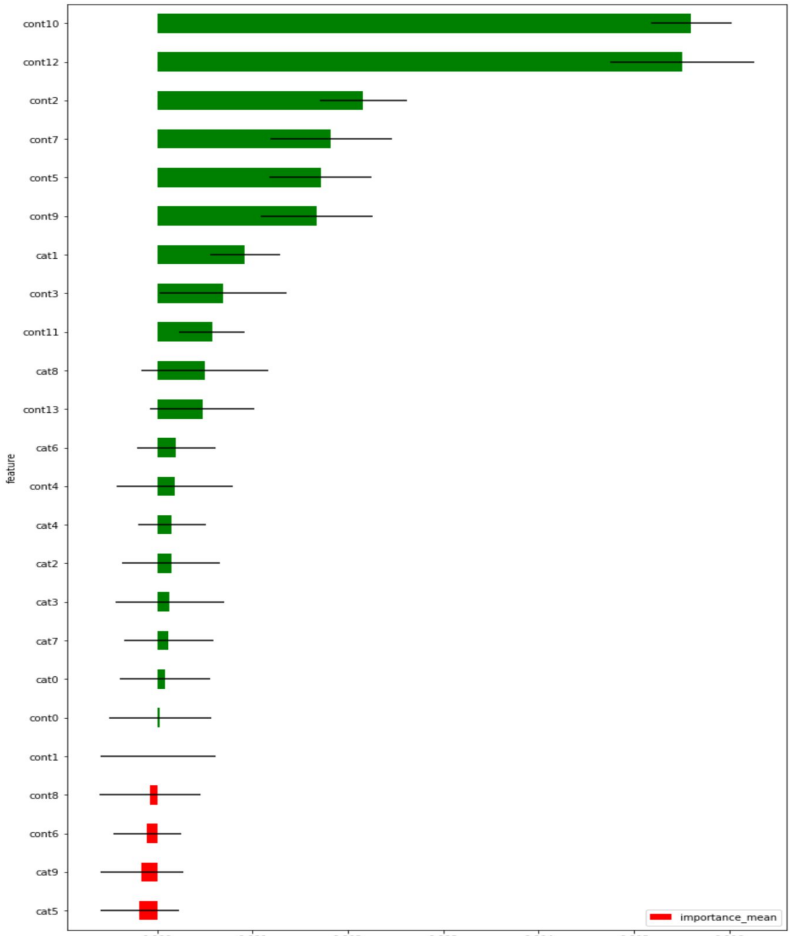
Ver más: Comparación interactiva TSNE y UMAP en distintos datasets

Leave One Feature Out (LOFO)

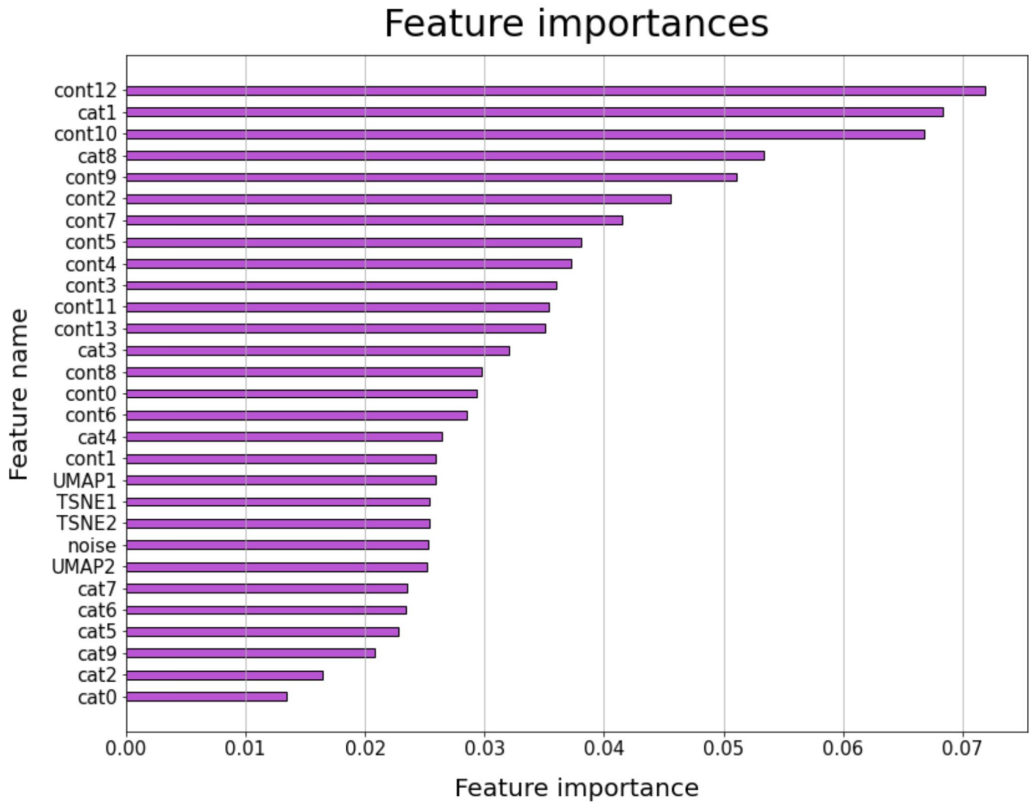
```
kf = KFold(n_splits=10, shuffle=True, random_state=42)

features = ['cat0', 'cat1', 'cat2', 'cat3', 'cat4', 'cat5', 'cat6', 'cat7',
       'cat8', 'cat9', 'cont0', 'cont1', 'cont2', 'cont3', 'cont4', 'cont5',
       'cont6', 'cont7', 'cont8', 'cont9', 'cont10', 'cont11', 'cont12',
       'cont13']

model = XGBRegressor(tree_method ="gpu_hist", gpu_id = 0, predictor = "gpu_predictor")
dataset = Dataset(df=train, target="target", features=features)
lofo_imp = LOFOImportance(dataset, cv=kf, scoring="neg_root_mean_squared_error", model = model)
importance_df_xgb = lofo_imp.get_importance()
plot_importance(importance_df_xgb, figsize=(12, 20))
```

```
100% 24/24 [05:00<00:00, 12.49s/it]
```

- Evalúa la performance del modelo utilizando todas las features, luego, de manera iterativa remueve features una a una.
- Se pueden agrupar features. Especialmente útil para features con alta cardinalidad como TFIDF o OHE.
- No favorece features granulares.
- Generaliza bien a test set no vistos.
- Es agnóstica al modelo.
- Da importancia negativa a features que de ser incluídas, disminuyen la performance.
- Existe otro paquete popular en Kaggle llamado [boruta-shap]



Añadir ruido blanco



- Notamos que las features con importancia similar o menor al ruido blanco son las que en LOFO tenían valores verdes pequeños o valores rojos.
- Tanto LOFO como Boruta-Shap intentan resolver el problema de encontrar “todas las features relevantes”.
- Esta técnica busca resolver el problema de “la cantidad óptima mínima de features”.
- Este tipo de paradigma es el utilizado por la mayoría de las técnicas.

Optuna

```python
def objective(trial):
    scores = []
    for fold in [4, 8]:
        lgb_params = {
                "task": "train",
                "boosting_type": "gbdt",
                "objective": "regression",
                "metric": "rmse",
                "learning_rate": trial.suggest_float("learning_rate", 1e-2, 0.25, log=True),
                "num_leaves": trial.suggest_int('num_leaves', 2, 256),
                "max_depth": trial.suggest_int("max_depth", 2, 12),
                "reg_alpha": trial.suggest_loguniform("reg_alpha", 1e-8, 100.0), # 'lambda_l1'
                "reg_lambda": trial.suggest_loguniform("reg_lambda", 1e-8, 100.0), #'lambda_l2'
                'feature_fraction': trial.suggest_uniform('feature_fraction', 0.2, 1.0),
                'bagging_fraction': trial.suggest_uniform('bagging_fraction', 0.2, 1.0),
                'bagging_freq': trial.suggest_int('bagging_freq', 1, 7),
                'min_child_samples': trial.suggest_int('min_child_samples', 5, 1000),
                'max_bin': trial.suggest_int('max_bin', 60, 255),
                "verbosity": -1,
                "bagging_seed": 42,
                "feature_fraction_seed": 42,
                "seed": 42,
                'device': 'gpu',
                'gpu_platform_id': 0,
                'gpu_device_id': 0
        }

        xtrain = df[df.kfold != fold].reset_index(drop=True)
        xvalid = df[df.kfold == fold].reset_index(drop=True)

        ytrain = xtrain.target
        yvalid = xvalid.target

        xtrain = xtrain[useful_features]
        xvalid = xvalid[useful_features]

        lgb_train = lgb.Dataset(xtrain, label=ytrain)
        lgb_valid = lgb.Dataset(xvalid, label=yvalid)

        pruning_callback = optuna.integration.LightGBMPruningCallback(trial, "rmse")
        model = lgb.train(lgb_params, lgb_train, valid_sets = [lgb_valid],
                          verbose_eval = False,
                          early_stopping_rounds = 500,
                          callbacks=[pruning_callback])
        preds_valid = model.predict(xvalid)

        rmse = mean_squared_error(yvalid, preds_valid, squared=False)
        scores.append(rmse)
    return np.mean(scores)
```

- Optimización de hiperparámetros mediante técnicas de optimización bayesiana y algoritmos genéticos. La elección del algoritmo es nuestra.
- Pruning Callbacks y más!

```
%%time
study = optuna.create_study(direction='minimize')
optuna.logging.set_verbosity(optuna.logging.WARNING)
study.optimize(objective, n_trials=1024)

best_trial=study.best_trial.params
print('Number of finished trials:', len(study.trials))
best_trial
```

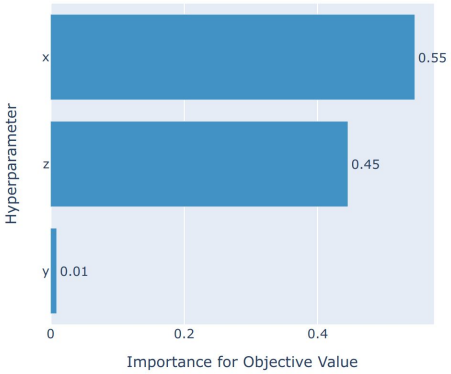
```
Number of finished trials: 1024
CPU times: user 52min 20s, sys: 1min 55s, total: 54min 15s
Wall time: 9min 4s
```

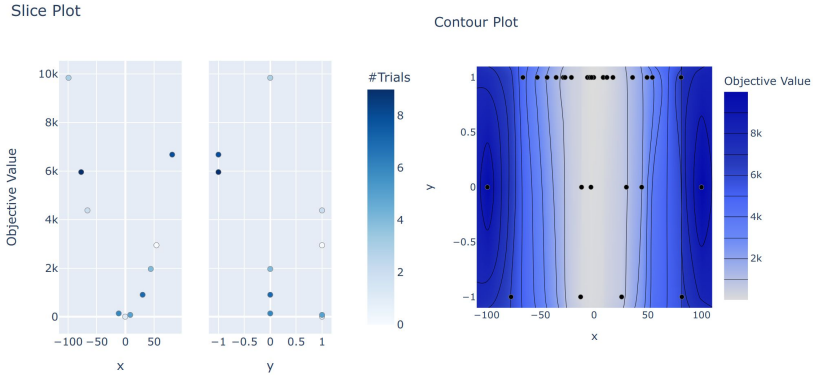
```
{'learning_rate': 0.23699787848918066,
 'num_leaves': 20,
 'max_depth': 3,
 'reg_alpha': 8.339166259513034e-08,
 'reg_lambda': 8.1226810334296e-06,
 'feature_fraction': 0.6174290449900618,
 'bagging_fraction': 0.9743730076588037,
 'bagging_freq': 5,
 'min_child_samples': 815,
 'max_bin': 156,
 'subsample': 0.6233259980070975,
 'colsample_bytree': 0.4159864166643203}
```

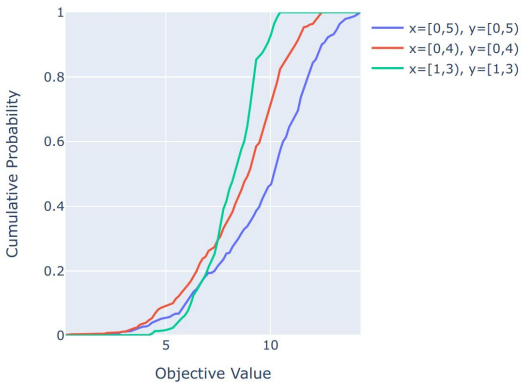
Optuna: Visualizaciones y otros

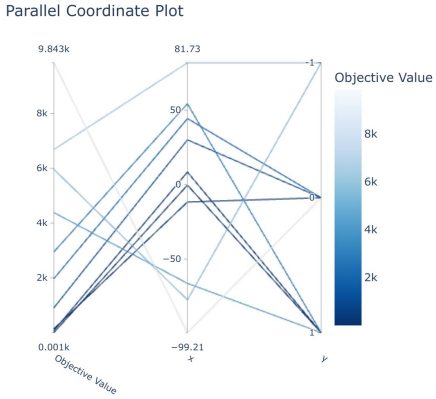
```
study.trials_dataframe().sort_values(by = "value").head(10)
```

	number	value	datetime_start	datetime_complete	duration	params_bagging_fraction
979	979	0.719468	2021-08-30 14:43:09.391219	2021-08-30 14:43:11.104505	0 days 00:00:01.713286	0.974373
352	352	0.719469	2021-08-30 14:38:03.356339	2021-08-30 14:38:04.686899	0 days 00:00:01.330560	0.921198
304	304	0.719472	2021-08-30 14:37:39.767114	2021-08-30 14:37:41.125778	0 days 00:00:01.358664	0.899713
463	463	0.719481	2021-08-30 14:38:48.088544	2021-08-30 14:38:49.542265	0 days 00:00:01.453721	0.979805
234	234	0.719491	2021-08-30 14:36:42.680724	2021-08-30 14:36:44.063550	0 days 00:00:01.382826	0.978024
235	235	0.719516	2021-08-30 14:36:44.063681	2021-08-30 14:36:45.409749	0 days 00:00:01.346068	0.960282
247	247	0.719517	2021-08-30 14:36:55.229956	2021-08-30 14:36:56.560002	0 days 00:00:01.330046	0.910473
201	201	0.719517	2021-08-30 14:36:24.719548	2021-08-30 14:36:26.103033	0 days 00:00:01.383485	0.984907
770	770	0.719520	2021-08-30 14:41:17.205343	2021-08-30 14:41:19.149733	0 days 00:00:01.944390	0.989210
590	590	0.719521	2021-08-30 14:39:38.776588	2021-08-30 14:39:40.277359	0 days 00:00:01.500771	0.983429










```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=10)
```

[Ver más]

Ensembling: Promediar semillas

```
%%time

avg_final_valid_predictions = 0
avg_final_test_predictions = 0
seeds = 30

for seed in range(seeds):
    print(20*"=","Running seed:", seed, 20*"=")
    lgb_params = {'n_estimators': 20000,
                  'n_jobs': 14,
                  'verbosity': -1,
                  'task': 'train',
                  'boosting_type': 'gbdt',
                  'objective': 'regression',
                  'metric': 'rmse',
                  'bagging_seed': seed,
                  'feature_fraction_seed': seed,
                  'seed': seed}
    lgb_params.update(best_trial)

    final_valid_predictions, final_test_predictions = lgb_model(train, test,
                                                                useful_features, lgb_params,
                                                               early_stopping_rounds=100,
                                                               verbose = False)

    avg_final_valid_predictions += pd.Series(final_valid_predictions)/seeds
    avg_final_test_predictions += np.mean(np.column_stack(final_test_predictions), axis=1) / seeds
```

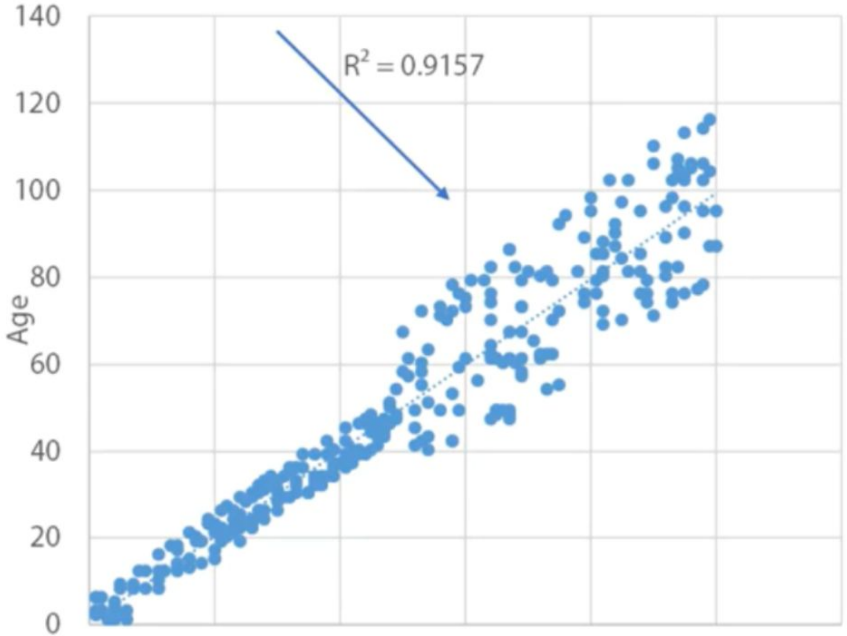
```
==================== Running seed: 0 ====================
fold:0, train_rmse: 17.658139372012194, folds_rmse: 21.541501592956404, gap: 3.8834
fold:1, train_rmse: 16.469264082405576, folds_rmse: 21.896301600454546, gap: 5.427
fold:2, train_rmse: 17.29671323266425, folds_rmse: 22.48841803929068, gap: 5.1917
fold:3, train_rmse: 15.365116755205412, folds_rmse: 23.30610802200228, gap: 7.941
fold:4, train_rmse: 14.628416515148697, folds_rmse: 22.259264381149457, gap: 7.6308
22.298318727170674, 0.5979451810997765, 6.014788735683448
==================== Running seed: 1 ====================
fold:0, train_rmse: 17.850510371350232, folds_rmse: 21.92163807629656, gap: 4.0711
fold:1, train_rmse: 15.974170494115318, folds_rmse: 21.920579889525754, gap: 5.9464
fold:2, train_rmse: 15.501243446278083, folds_rmse: 22.451956873528708, gap: 6.9507
fold:3, train_rmse: 13.508062957554522, folds_rmse: 23.480880594577396, gap: 9.9728
fold:4, train_rmse: 15.102688321944472, folds_rmse: 22.19266939708869, gap: 7.09
22.393544966203425, 0.5783517598165064, 6.806209847954896 5
==================== Running seed: 2 ====================
fold:0, train_rmse: 17.322228750607454, folds_rmse: 21.821862433114095, gap: 4.4996
fold:1, train_rmse: 14.580564547737302, folds_rmse: 22.078741702893907, gap: 7.4982
fold:2, train_rmse: 17.117854388949493, folds_rmse: 22.42077437528872, gap: 5.3029
fold:3, train_rmse: 12.673419074849988, folds_rmse: 23.62031374140203, gap: 10.9469
fold:4, train_rmse: 15.877155209977879, folds_rmse: 22.3747568515555, gap: 6.4976
22.46328982085085, 0.6176821757593857, 6.949045426426426
```

```
score = []
for fold in range(5):
    valid, pred = np.column_stack(train_merge[train_merge["kfold"] == fold][["target", "pred_lgb"]].to_numpy())
    rmse = mean_squared_error(valid, pred, squared = False)
    score.append(rmse)
    print("RMSE:", rmse)

print("Avg RMSE:", np.mean(score))
```

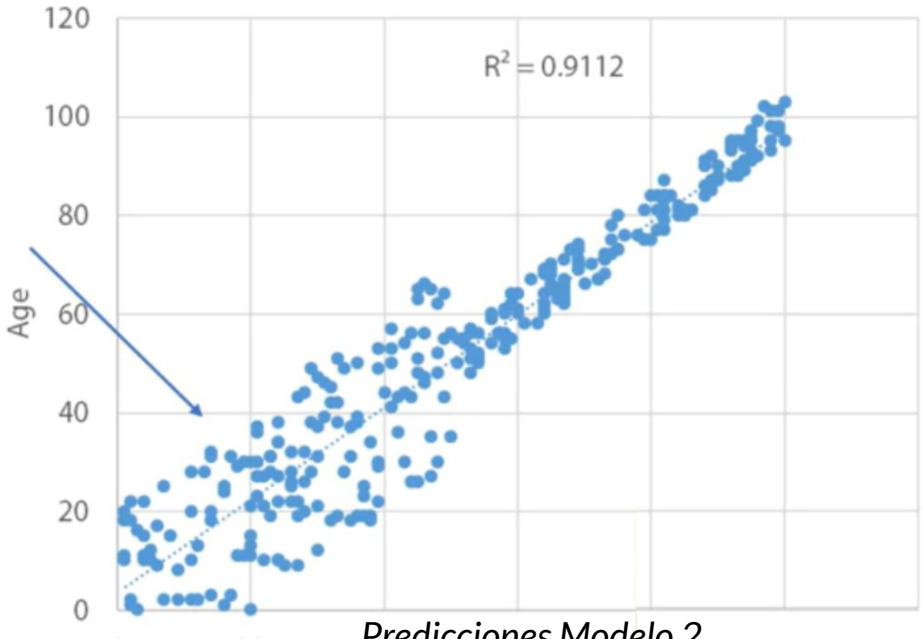
```
RMSE: 21.263085048877237
RMSE: 21.550283514208225
RMSE: 21.920939254519375
RMSE: 22.883274821267285
RMSE: 21.868735925968366
Avg RMSE: 21.8972637129681
```

- Es super simple, solo cambiar parámetros relacionados a random state / seed.
- En este caso logramos un boost en ~ 0.4 lo que corresponde a una disminución en 1.8% solo por promediar.
- Cuando un modelo tarda un tiempo considerable en entrenar, la alternativa es usar una semilla distinta en cada fold.

Ensembling: Stacking



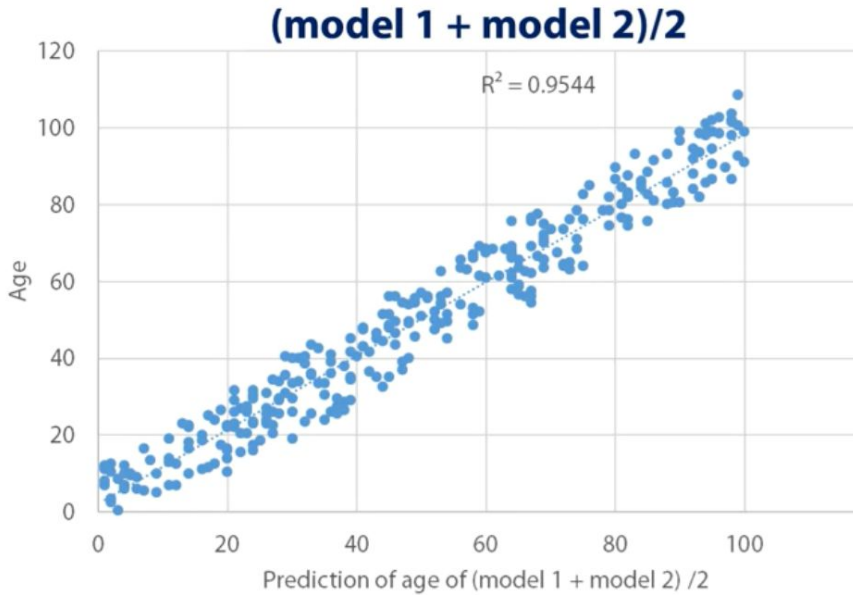
Predicciones Modelo 1

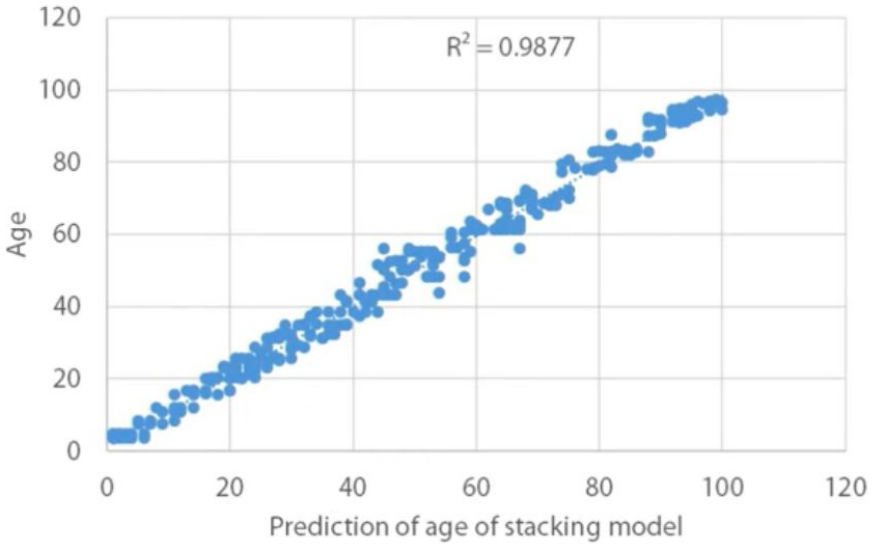


Predicciones Modelo 2

- Modelos fuertes y diversos (distinto algoritmo, distintas features, distintos hiperparámetros, etc)
- Notamos que en promedio tienen una performance similar, sin embargo, son muy distintos!
- Óptimo teórico: Si la edad (AGE) es menor a 50 usar modelo 1, si no usar modelo 2.

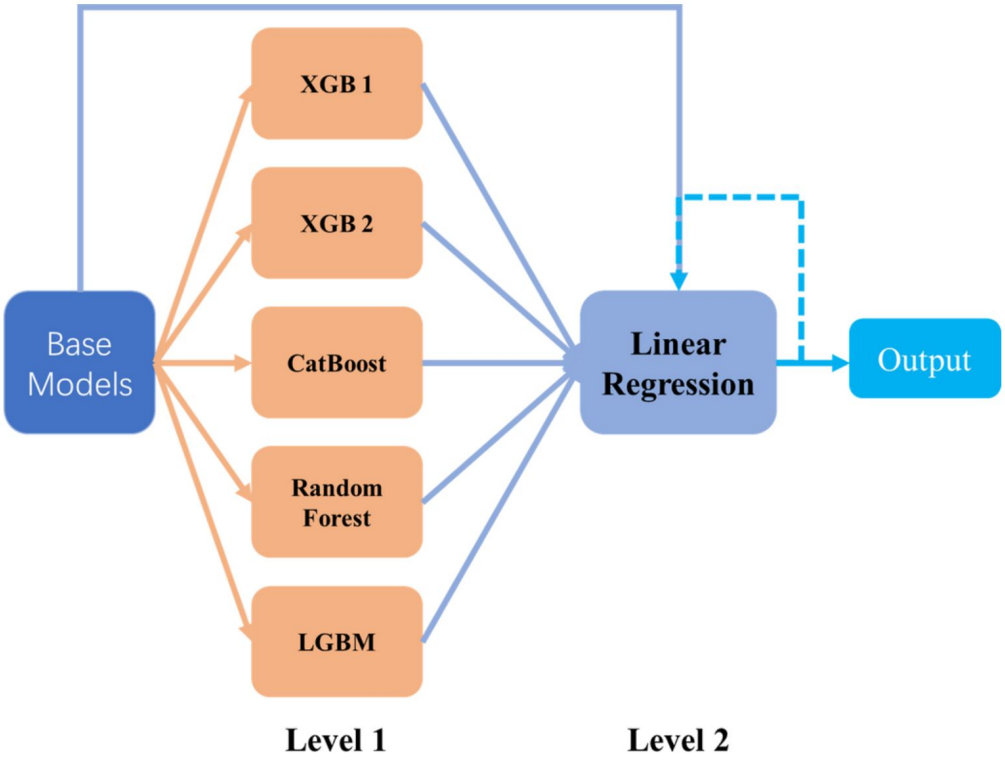
Ensembling: Stacking





- En primera instancia, podríamos tomar un promedio simple, pues ambos modelos tienen métricas similares.
- Stacking intenta encontrar el óptimo teórico, que no podemos ver, pues requiere usar la variable a predecir.
- En este caso una regresión lineal de los 2 modelos logra una mejora en performance considerable.

Arquitectura Stacking Competencia



- Base models:
  - XGBoost con distintos hiperparámetros, encodings y features.
  - LightGBM optimizado con LOFO y Optuna
  - CatBoost optimizado con LOFO y Optuna
  - TabNet
- Level 1:
  - XGBoost: Base models
  - XGBoost: Base feature + base models
  - CatBoost
  - RandomForest
  - LightGBM
- Level 2:
  - LinearRegression:
    - *Positive = True*
    - *fit_intercept=False*
  - PseudoLabeling

```python
for fold in range(10):
    xtrain =  df[df.kfold != fold].reset_index(drop=True)
    xvalid = df[df.kfold == fold].reset_index(drop=True)
    xtest = test.copy()

    valid_ids = xvalid.id.values.tolist()

    ytrain = xtrain.target
    yvalid = xvalid.target

    xtrain = xtrain[useful_features]
    xvalid = xvalid[useful_features]

    ordinal_encoder = OrdinalEncoder()
    xtrain[object_cols] = ordinal_encoder.fit_transform(xtrain[object_cols])
    xvalid[object_cols] = ordinal_encoder.transform(xvalid[object_cols])
    xtest[object_cols] = ordinal_encoder.transform(xtest[object_cols])


    model = XGBRegressor(**xgb_params, random_state=fold)

    model.fit(xtrain, ytrain, verbose=False,
              eval_set=[(xvalid, yvalid)],
              eval_metric="rmse",
              early_stopping_rounds=1000
             )

    preds_train = model.predict(xtrain)
    preds_valid = model.predict(xvalid)
    test_preds = model.predict(xtest)

    final_test_predictions.append(test_preds)
    final_valid_predictions.update(dict(zip(valid_ids, preds_valid)))

    train_rmse = mean_squared_error(ytrain, preds_train, squared=False)
    rmse = mean_squared_error(yvalid, preds_valid, squared=False)
    gap = ((rmse - train_rmse)/train_rmse*100)

    scores.append(rmse)
    gaps.append(gap)
    print(f"fold:{fold}, train_rmse: {train_rmse}, folds_rmse: {rmse}, gap: {gap.round(4)}")


print(f"{np.mean(scores)}, {np.std(scores)}, {np.mean(gaps)}")
final_valid_predictions = pd.DataFrame.from_dict(final_valid_predictions, orient="index").reset_index()
final_valid_predictions.columns = ["id", "pred_xgb"]
final_valid_predictions.to_csv("level1/train_pred_xgb.csv", index=False)

sample_submission.target = np.mean(np.column_stack(final_test_predictions), axis=1)
sample_submission.columns = ["id", "pred_xgb"]
sample_submission.to_csv("level1/test_pred_xgb.csv", index=False)
```

- Guardar out of fold IDs

- Usar solo las features útiles
- Encoding variables categóricas

- Hacer predicciones en test set y en validación.
- Guardar IDs y predicciones oof

- Crear dataframe con todas las predicciones oof y guardarlo.
- Guardar predicciones en test set

Ensembling: Stacking

```
files = listdir("level1")

train_preds = [file for file in files if ("train" in file)]
test_preds = [file for file in files if ("test" in file)]

for i, file in tqdm(enumerate(train_preds)):
    df_aux = pd.read_csv(f"level1/{file}")
    df_aux.columns = ["id", f"pred_{i}"]
    df = df.merge(df_aux, on="id", how="left")

for i, file in tqdm(enumerate(test_preds)):
    df_aux = pd.read_csv(f"level1/{file}")
    df_aux.columns = ["id", f"pred_{i}"]
    test = test.merge(df_aux, on="id", how="left")

useful_features = [c for c in df.columns if c.startswith("pred")]
test = test[useful_features]

del df_aux; gc.collect();
```

- Leer .csv con predicciones nivel 1 tanto para train y test.
- Agregar estas features en train y test

```
final_test_predictions = []
final_valid_predictions = {}
scores = []
for fold in range(10):
    xtrain =  df[df.kfold != fold].reset_index(drop=True)
    xvalid = df[df.kfold == fold].reset_index(drop=True)
    xtest = test.copy()

    valid_ids = xvalid.id.values.tolist()

    ytrain = xtrain.target
    yvalid = xvalid.target

    xtrain = xtrain[useful_features]
    xvalid = xvalid[useful_features]

    model = XGBRegressor(random_state = fold, **xgb_params)
    model.fit(xtrain, ytrain, early_stopping_rounds=800, eval_set=[(xvalid, yvalid)], verbose=1000)
    preds_valid = model.predict(xvalid)
    test_preds = model.predict(xtest)
    final_test_predictions.append(test_preds)
    final_valid_predictions.update(dict(zip(valid_ids, preds_valid)))
    rmse = mean_squared_error(yvalid, preds_valid, squared=False)
    print(fold, rmse)
    scores.append(rmse)

print(np.mean(scores), np.std(scores))
final_valid_predictions = pd.DataFrame.from_dict(final_valid_predictions, orient="index").reset_index()
final_valid_predictions.columns = ["id", "pred_xgb_v2"]
final_valid_predictions.to_csv("level2_lunes/train_pred_xgb_v2.csv", index=False)

sample_submission.target = np.mean(np.column_stack(final_test_predictions), axis=1)
sample_submission.columns = ["id", "pred_xgb_v2"]
sample_submission.to_csv("level2_lunes/test_pred_xgb_v2.csv", index=False)
```

Hacemos lo mismo de antes, solo que ahora se llama nivel 2

```python
for i, file in tqdm(enumerate(train_preds)):
    df_aux = pd.read_csv(f"level2_lunes/{file}")
    df_aux.columns = ["id", f"pred_{i}"]
    df = df.merge(df_aux, on="id", how="left")

for i, file in tqdm(enumerate(test_preds)):
    df_aux = pd.read_csv(f"level2_lunes/{file}")
    df_aux.columns = ["id", f"pred_{i}"]
    test = test.merge(df_aux, on="id", how="left")
```

- Cargamos predicciones de nivel 2 y las añadimos como features.

- Las ponderamos usando Regresión Lineal

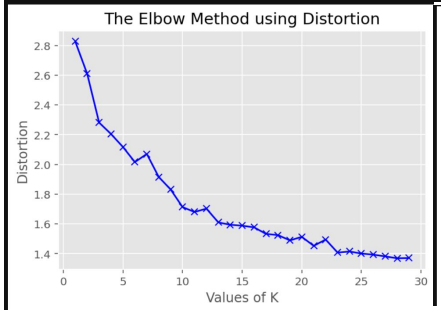
- Guardamos predicciones finales para ser subidas a la competencia.

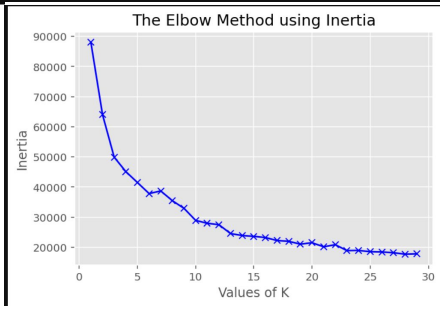
```python
final_predictions = []
scores = []
final_test_predictions = []
final_valid_predictions = {}

for fold in range(10):
    xtrain =  df[df.kfold != fold].reset_index(drop=True)
    xvalid = df[df.kfold == fold].reset_index(drop=True)
    xtest = test_lin.copy()

    valid_ids = xvalid.id.values.tolist()

    ytrain = xtrain.target
    yvalid = xvalid.target

    xtrain = xtrain[linear_feat]
    xvalid = xvalid[linear_feat]

    model = LinearRegression(fit_intercept=False)
    model.fit(xtrain, ytrain)

    preds_valid = model.predict(xvalid)
    test_preds = model.predict(xtest)
    final_predictions.append(test_preds)
    final_test_predictions.append(test_preds)
    final_valid_predictions.update(dict(zip(valid_ids, preds_valid)))
    rmse = mean_squared_error(yvalid, preds_valid, squared=False)
    print(fold, rmse, model.coef_, sum(model.coef_))
    scores.append(rmse)

print(np.mean(scores), np.std(scores))

sample_submission.target = np.mean(np.column_stack(final_predictions), axis=1)
sample_submission.to_csv("sub_level2/submission_martes_lowest.csv", index=False)

final_valid_predictions = pd.DataFrame.from_dict(final_valid_predictions, orient="index").reset_index()
final_valid_predictions.columns = ["id", "pred_lin"]
final_valid_predictions.to_csv("sub_level2/train_pred_lin.csv", index=False)

sample_submission.target = np.mean(np.column_stack(final_test_predictions), axis=1)
sample_submission.columns = ["id", "pred_lin"]
sample_submission.to_csv("sub_level2/test_pred_lin.csv", index=False)
```

StratifiedKfold con clusters

```
%%time

distortions = []
inertias = []
mapping1 = {}
mapping2 = {}
K = range(1, 30)

for k in K:
    # Building and fitting the model
    kmeanModel = KMeans(n_clusters=k).fit(X)
    kmeanModel.fit(X)

    distortions.append(sum(np.min(cdist(X, kmeanModel.cluster_centers_,
                                        'euclidean'), axis=1)) / X.shape[0])
    inertias.append(kmeanModel.inertia_)

    mapping1[k] = sum(np.min(cdist(X, kmeanModel.cluster_centers_,
                                   'euclidean'), axis=1)) / X.shape[0]
    mapping2[k] = kmeanModel.inertia_
```

```
CPU times: user 1.93 s, sys: 10.7 ms, total: 1.94 s
Wall time: 1.94 s
```





- Si tenemos que elegir a ciegas un valor entonces 2 veces el número de features funciona bien en general.
- En este caso gracias a RAPIDS podemos testear 30 valores en segundos.
- Finalmente usamos los labels para estratificar nuestros folds.
- Ayuda a disminuir *sampling bias* y estabilizar métricas (no siempre)

```
fold_idxs = []
skf = StratifiedKFold(n_splits=5,
                      shuffle=True,
                      random_state=42)
y_stratified = km.labels_
fold_idxs.append(list(skf.split(train_og, y_stratified)))
```

¿Preguntas / Comentarios ?

Gracias por su tiempo.