Entrega 1 Taller de Aprendizaje Automático 2025

Uriel Yaffé

I. Talleres 3 y 5 – Demanda de bicicletas

A. 1

La búsqueda de hiperparámetros se realizó mediante GridSearchCV sobre un DecisionTreeRegressor envuelto en un TransformedTargetRegressor con transformación logarítmica. Se utilizó RMSLE como métrica de evaluación ya que los valores a estimar son grandes y RMSLE controla bien la gran dispersión de valores de los datos.

```
diccionario = {
    'regressor__max_depth': [5,8,10,12,14,15,20],
    'regressor__min_samples_split' : [2,4,6,8,10],
    'regressor__min_samples_leaf': [1,2,4,8,12],
    'regressor__max_features': ['sqrt', 'log2', None]
    }
TT_regressor = TransformedTargetRegressor(
    regressor=DecisionTreeRegressor(random_state=2),
    func=np.log1p, inverse_func=np.expm1
)

grid_search = GridSearchCV(estimator=TT_regressor, param_grid=diccionario, cv=5, scoring=rmsle_scorer)

grid_search.fit(X_train, y_train)

best_model = grid_search.best_estimator_
best_depth = grid_search.best_params_['regressor__max_depth']

print("Mejores hiperparámetros:", grid_search.best_params_)
print("Mejor score RMSLE:", -grid_search.best_score_)
```

```
Mejores hiperparámetros: {'regressor__max_depth': 10, 'regressor__max_features': None, 'regressor__min_samples_leaf': 12, 'regressor__min_samples_split': 2}
Mejor score RMSLE: 0.6087007236395622
```

Fig. 1. GridSearch para el DecisionTreeRegressor

B. 2

Un árbol de decisión sin restricciones tiende a ajustarse excesivamente al conjunto de entrenamiento, lo que puede llevar a un problema de *overfitting*. En estos casos, una pequeña variación en los datos de entrada puede generar cambios significativos en la estructura del árbol y, por lo tanto, en sus predicciones.

La idea de combinar múltiples árboles de decisión, como ocurre en métodos de ensamblado tipo *bagging* o *random forests*, busca reducir este problema.
Al introducir aleatoriedad en el proceso de construcción de cada árbol (por ejemplo, seleccionando subconjuntos distintos de datos o de características), se genera un conjunto diverso de modelos.
Aunque un árbol individual puede cometer errores por sobreajuste, la combinación de muchos árboles con distintos sesgos permite que, en promedio, la predicción conjunta sea más robusta. De este modo, el error de un árbol puede ser corregido por la mayoría de los demás, mejorando la generalización del modelo.

La combinación o el promedio de modelos simples tiene mayor capacidad que cada modelo por separado y además, si los conjuntos de entrenamiento son independientes, la combinación reduce la varianza sin afectar el sesgo.

C. 3

Para el mejor estimador de la parte 3 creé un Pipeline con dos etapas. La primera etapa es de preprocesamiento, a los datos les hice un *StandarScaler()* para poder estandarizarlos. Decidí eliminar columnas que consideré irrelevantes o redundantes. Por ejemplo, descarté la columna ”temp” y mantuve ”atemp”, ya que ambas representan una medida de temperatura y contienen información similar, siendo innecesario conservar ambas.
Otra columna relevante a eliminar que genera redundancia es ’season’, ya que está relacionada con la temperatura, el clima. Por última, ’Holiday’ indica si el día es festivo, el cual ya tiene en cuenta la columna ”Working Day” Otra etapa de procesamiento importante que no incluí en el código son los SimpleImputers. Estos se usan para sustituir valores faltantes o valores que están puestos para rellenar (una valor característico como -999). En esta dataset no hay valores faltantes pero si se quiere volver a utilizar para otro dataset que se desconoce si hay valores faltantes, mejora mucho el modelo.

Para la segunda parte del Pipeline utilicé un Voting Regressor, un modelo de ensamble que combina las predicciones de varios regresores (en este caso, Random Forest y XGBoost,los predictores que mejor desempeño tuvieron a lo largo del Taller) promediando sus resultados. Para ambos predictores usé el mejor modelo encontrado mediante GridSearch en las partes anteriores.

```
from xgboost import XGBRegressor
param_grid = {
    "n_estimators": [50, 100, 200, 300],
    "learning_rate": [0.01, 0.05, 0.1, 0.2]
}

xgb_model = XGBRegressor(objective="reg:squarederror", random_state=2)


grid_search = GridSearchCV(
    estimator=xgb_model,
    param_grid=param_grid,
    scoring=rmsle_scorer,
    cv=5,
    n_jobs=-1
)

grid_search.fit(X_train, y_train)


best_model4 = grid_search.best_estimator_
best_xgb_params = grid_search.best_params_
best_rmsle = grid_search.best_score_
```

Fig. 2. GridSearch para encontrar los mejores Hiperpárametros del XGBRegressor

Para el GridSearch mostrado en la Figura 3, decidí separarlo en dos GridSearch secuenciales donde cada GridSearch busca diferentes hiperparámetros y el segundo GridSearch usa los hiperpárametros óptimos encontrados por el primer GridSearch. Esto, aunque no sea la forma más eficiente, se

```
diccionario = {
    'max_depth': [5,8,10,12],
    'n_estimators' : [10,50,100,150]
    }

grid_search = GridSearchCV(estimator=RandomForestRegressor(random_state=2), param_grid=diccionario, cv=5)

grid_search.fit(X_train, y_train)

best_model = grid_search.best_estimator_
best_depth = grid_search.best_params_['max_depth']


best_n = grid_search.best_params_['n_estimators']
```

```
diccionario2 = {
    'min_samples_split': [2, 5, 10, 15, 20],
    'min_samples_leaf': [1, 2, 4, 10, 15, 30]
    }


best_rf = RandomForestRegressor(max_depth=best_depth, n_estimators=best_n,n_jobs=-1,random_state=2)

grid_search2 = GridSearchCV(estimator=best_rf, param_grid=diccionario2, cv=5)

grid_search2.fit(X_train, y_train)

best_model2 = grid_search2.best_estimator_
best_leaf = grid_search2.best_params_['min_samples_leaf']
best_sample = grid_search2.best_params_['min_samples_split']
```

Fig. 3. GridSearch para encontrar los mejores hiperpárametros del RandomForest

ahorra mucho tiempo de cómputo. (El GridSearch con cuatro hiperparametros con cuatro valores cada uno y un CV=5, se entrenan 1280 modelos, mientras que hacer dos GridSearch secuenciales se entrenan 160 modelos)

```
rf_model = RandomForestRegressor(**best_params2)
xgb_model = XGBRegressor(**best_xgb_params)

voting_model = VotingRegressor(estimators=[
    ("random_forest", rf_model),
    ("xgboost", xgb_model)
])
drop_cols = ['temp','holiday','season']

num_cols = [col for col in X_train.columns if col not in drop_cols]

preprocessor = ColumnsTransformer(transformers=[
    ('scaler',StandardScaler(), num_cols)
    ,('dropper','drop',drop_cols)
])

pipeline = Pipeline([
    ('preprocessing', preprocessor),
    ("ensemble", voting_model)
])
```

Fig. 4. Pipeline del mejor modelo del Taller 3

Le hice un cross validate para obtener el rendimiento

```
]: score_tot = cross_validate(pipeline,X_train,y_train,cv=5,scoring=rmsle_scorer)

]: print(score_tot['test_score'].mean()*-1)

0.5600355629278501
```

En el Taller 5, el mejor modelo fue hallado por una búsqueda de hiperparametros por Optuna. Se utilizó Optuna para encontrar los mejores hiperparámetros para la red neuronal. Los hiperparámetros óptimos encontrados fueron , n_hidden: 5 capas ocultas, n_neurons: 207 neuronas por capa, learning_rate: 0.000567, optimizer_name: ”adam” y activation_name: ”relu”.

En el *pipeline*, se eliminaron las columnas `temp` y `workingday` del conjunto de entrenamiento. Luego, se normalizaron las variables de entrada utilizando `StandardScaler`, mientras que las etiquetas (`y`) también fueron escaladas con un segundo `StandardScaler` para asegurar una coherencia del modelo durante el entrenamiento.

Los datos fueron divididos en un conjunto de entrenamiento (90%) y uno de prueba (10%). A su vez, el conjunto de entrenamiento fue subdividido en un conjunto de entrenamiento final (90%) y uno de validación (10%), utilizado para ajustar los parámetros internos del modelo y prevenir sobreajuste.

Se construyó un modelo secuencial utilizando `Keras`, el cual fue entrenado con los datos escalados previamente. Posteriormente, para evaluar su desempeño, se aplicó la transformación inversa a las predicciones y a los valores reales del conjunto de prueba, con el fin de calcular el RMSLE, el cual tuvo un desempeño mucho peor que los arboles de decisión, con un RMSLE de 1.46

Esta diferencia en el desempeño se me ocurre que puede deberse a que, a diferencia de las redes neuronales, a los árboles de decisión no solo se les ajustaron los mejores hiperparámetros, sino que también se les aplicó bagging. Esta técnica introduce aleatoriedad al entrenar múltiples modelos sobre diferentes subconjuntos del conjunto de entrenamiento (obtenidos con reemplazo), lo cual ayuda a reducir la varianza y mejora la generalización del modelo.

Tal vez, si hubiera entrenado las redes neuronales con técnicas similares que introducen aleatoriedad y robustez, como dropout (se mencionó en el teórico), podría haber obtenido un mejor desempeño.

II. Taller 4 – Detección de anomalías

A. 1

El concepto de ”detección de anomalías” en este contexto se refiere a identificar comportamientos inusuales, en este caso, en las conexiones de la red. Lo que le da sentido es que la mayoría de tráfico en cualquier red es tráfico no malicioso, lo que hace que si aparece alguna conexión diferente a las usuales, se vean como una anomalía. Estas ”anomalías”, al ser eventos raros, no se cuenta con mucha información sobre ellas. Por esta razón, en el conjunto que se nos proporciona para entrenar el modelo, todas las conexiones son normales. Al tener un conjunto de entrenamiento compuesto únicamente por conexiones normales, el modelo se entrena a partir de estos datos ”normales”, y luego, se ajusta el punto de operación para identificar posibles anomalías.

B. 2

Para la creación del Pipeline comencé definiendo imputers tanto para variables categóricas como numéricas. Si bien el conjunto de entrenamiento no contiene datos faltantes, considero necesario incluir estas etapas para cubrir posibles valores ausentes en el conjunto de prueba, ya que no tenemos certeza sobre su calidad.

```
cat_imputer = SimpleImputer(strategy="most_frequent")
num_imputer = SimpleImputer(strategy='constant', fill_value=-1)
```

Luego, para las columnas numéricas se utilizó un StandardScaler(), ya que quiero que todas las variables estén en una escala comparable. Esto previene que algunas características dominen sobre otras debido a sus rangos de valores más amplios. Por ejemplo, la columna '$srcbytes$' representa la cantidad de bytes enviados por el host, y puede tener una gran variabilidad. Esta variación no debería tener tanto peso comparado con una columna como '$numroot$', que indica la cantidad de veces que se intentó acceder con privilegios de root, algo mucho más relevante para detectar anomalías.

Para las columnas categóricas se aplicó un OneHotEncoder, que convierte las categorías en variables binarias para que puedan ser interpretadas por los modelos. La principal desventaja es que, si una columna tiene muchas categorías distintas (como podría ser el nombre de los servicios o tipos de ataques), el OneHotEncoder genera una columna por cada categoría, lo que aumenta significativamente la dimensionalidad del conjunto de datos.

```
procesamiento_numerico = Pipeline([
        ("imputacion", num_imputer),
        ("Scaler", StandardScaler() ) ])
procesamiento_categorico = Pipeline([
        ("imputacion", cat_imputer),
        ("one_hot", OneHotEncoder(handle_unknown="ignore")) ])

preprocesador = ColumnTransformer([
    ('num_imputer', procesamiento_numerico, columnas_numericas),
    ('cat_imputer', procesamiento_categorico, columnas_categoricas)
], remainder='drop')
```

Fig. 5. Pipeline para la detección de anomalías

Para finalizar, pase los datos por el pipeline creado para poder hacer una matriz de correlación entre sus columnas. La matriz nos muestra qué tan dependiente es una columna con cada una de las otras. Como umbral, decidí dropear una de cada par de columnas que tengan más de $\%90$ de correlación entre sí

```
mask = (correlation_matrix.abs() > 0.9) & (np.abs(correlation_matrix) != 1)

highly_correlated_pairs = []

for column in correlation_matrix.columns:
    correlated_columns = correlation_matrix.index[mask[column]].tolist()
    for col in correlated_columns:
        if column != col:
            highly_correlated_pairs.append((column, col, correlation_matrix.loc[column, col]))
```

C. 3

Las curvas de *Precision vs Recall* en K-Means y PCA tienen formas similares. A medida que aumenta el percentil del umbral, la precisión se incrementa gradualmente, mientras que el *Recall* permanece constante en 1 hasta aproximadamente el percentil 70%, donde comienza a disminuir. En la gráfica de PCA, el *Recall* cae lentamente a partir de ese punto, y después

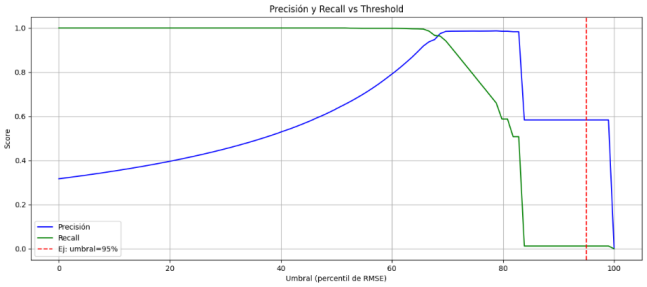

Fig. 6. Curvas de Precisison vs Recall de PCA

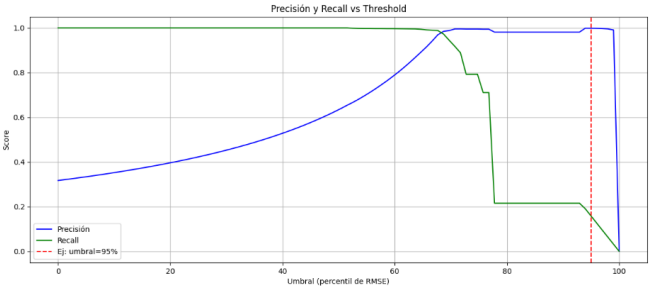

Fig. 7. Curvas de Precisison vs Recall de K-MEANS

del percentil 80, su valor se vuelve prácticamente nulo, lo cual indica que casi no se detectan anomalías verdaderas en umbrales altos. En cambio, en K-Means, si bien el *Recall* también decrece, lo hace de manera más abrupta y permanece en valores no nulos durante un mayor rango de percentiles, lo que indica una mayor persistencia en la detección de anomalías. Respecto a la precisión, ambas estrategias alcanzan valores altos en la región entre los percentiles 70 y 80. Sin embargo, en K-Means, la precisión se mantiene alta por más valores de percentil. En PCA, aunque también se logra una alta precisión, esta cae abruptamente junto con el *Recall* después del percentil 80. PCA ofrece mejor desempeño en general en los umbrales intermedios (Entre 70–80$\%$), tomando en consideración tanto precisión como *Recall*, mientras que K-Means resulta más estable y tolerante en un rango más amplio de umbrales, aunque con menor *Recall* en los mejores puntos de precisión. Esta diferencia en el comportamiento podría deberse a que PCA, al reducir la dimensionalidad, pierde cierta información que en K-Means aún está presente. Esto haría que la precisión y el *Recall* en PCA colapsen más abruptamente frente a cambios en el umbral, mientras que K-Means, al trabajar en el espacio completo, retiene una mayor capacidad de discriminación en percentiles altos.

D. 4

Si tuviera que quedarme con alguna de estas dos estrategias, viendo las gráficas de *Precision vs Recall*, me quedaría con *K-Means*. Para el escenario de detección de anomalías, se necesita crear un modelo que pueda identificar con solvencia las anomalías, tratando de dejar pasar la menor cantidad posible.

Por esta razón, se busca un modelo con un *recall* alto. Obviamente, debido al compromiso entre *precision* y *recall*, la *precision* no será muy buena, lo que generará varias falsas alarmas. Este es el precio que se paga, ya que es preferible detectar como una amenaza un mensaje sin malicia (el cual puede ser reenviado sin consecuencias) que no detectar un ataque de tipo *DoS*.

En el caso de *K-Means*, en el intervalo de percentil alrededor del 65% (antes de que se crucen las curvas), el *recall* se mantiene muy cerca de 1, lo que indica que casi todas las anomalías son detectadas. Aunque la *precision* no es perfecta (aproximadamente 90%), considero que es un valor más que aceptable con la justificación brindada anteriormente.

Para obtener un punto de operación adecuado, recorro los valores de percentiles y me quedo con aquellos que me otorguen un *recall* mayor al $99{,}5\%$ (valor que considero adecuado según lo justificado anteriormente).

Como la función de *Recall* es decreciente a medida que aumenta el percentil, el valor del umbral que cumple con la condición Recall $\geq 0{,}995$ y que me deja el mejor valor de *Precision* corresponde al mayor valor de percentil dentro de esa condición.

```python
from sklearn.metrics import recall_score
import numpy as np

mejor_recall, mejor_umbral = 0, None
for p in np.linspace(0, 100, 100):
    umbral = np.percentile(scores_val, p)
    y_pred = (scores_val > umbral).astype(int)
    recall = recall_score(y_val2, y_pred)


    if recall >= 0.995:
        mejor_recall = recall
        mejor_umbral = p

print(f"Mejor recall >= 0.995 = {mejor_recall:.4f} en percentil {mejor_umbral:.1f}")
```

```
Mejor recall >= 0.995 = 0.9953 en percentil 63.6
```

Fig. 8. Código para elegir el umbral óptimo

E. 5

Luego de elegir el umbral óptimo, evaluamos nuestro modelo en el conjunto de *test*.

Al hacerlo, observamos que el *Recall* se mantuvo alrededor del 90% en dicho conjunto, pero la *Precision* sobre las muestras que no eran amenazas disminuyó. Esto era esperable, ya que el criterio con el que se eligió el punto de operación no era mantener una *Precision* alta.

	precision	recall
Normal	0.74	0.94
Ataque	0.98	0.92

Fig. 9. Precision y Recall en Test

Al usar la tasa de aciertos por categoría, se puede notar que hay algunas anomalías que son detectadas de manera perfecta (tasa = 1.0), mientras que otras no superan una tasa del 10%.

Para investigar si existía alguna correlación entre la tasa de aciertos y la presencia de anomalías en el conjunto de validación, separé y promedié la tasa de acierto de las anomalías que sí fueron utilizadas para validación y aquellas que solo aparecían en el conjunto de test.

El promedio de acierto para clases en validación es 0.890%, mientras que el promedio de acierto para clases NO vistas en validación es 0.749%. Esto nos indica que tiende a detectar mucho mejor a las anomalías que se uso para validación.