Análisis de Data Streams: Electricity Prices

Trabajo Final - Datos Temporales y Complejos

Javier Vela Tambo

2025-04-18

Resumen

En este trabajo se exploran técnicas de aprendizaje automático para data streams, con un enfoque en el dataset Elec2 que contiene datos de precios de electricidad en Nueva Gales del Sur, Australia. Se analizan algoritmos de clasificación como Hoeffding Trees y KNN, evaluando su rendimiento mediante métodos de evaluación holdout y prequential. También se implementan algoritmos de clustering como CluStream y DenStream. Se pone especial atención a la detección de concept drift, utilizando el algoritmo ADWIN para identificar cambios significativos en los patrones de datos a lo largo del tiempo.

Introducción

Este documento analiza flujos de datos (*data streams*) utilizando el dataset `Elec2`, que contiene información sobre precios de electricidad en Nueva Gales del Sur, Australia. Este conjunto de datos es ideal para evaluar algoritmos de aprendizaje automático debido a su naturaleza temporal y compleja.

Se utiliza la biblioteca de Python River para implementar y evaluar algoritmos de clasificación, detección de concept drift y clustering.

El documento está organizado en cuatro secciones principales (asociadas a cada uno de los ejercicios propuestos) además de la introducción y presentación del conjunto de datos: Técnica de Evaluación: *Holdout*, Técnica de Evaluación: *Prequential*, Detección de *Concept Drift* y Técnicas de *Clustering*.

> **Creación del Documento**
>
> El Jupyter Notebook ha sido convertido a un documento PDF utilizando Quarto. Algunas celdas, como las que contienen código para mostrar gráficos y ciertos resultados, se han ocultado para evitar redundancias y mantener el documento limpio.
> Se puede consultar el Notebook original en el repositorio de GitHub.

En la siguiente celda, se importan las librerías necesarias. Para la gestión de datos se utiliza `pandas`, para la visualización `matplotlib` y `seaborn`, para operaciones matemáticas `numpy`, y para el aprendizaje automático *online* se utiliza `river`.

La librería `river` es una biblioteca de aprendizaje automático en Python diseñada para trabajar con flujos de datos. Permite implementar algoritmos de aprendizaje automático que pueden adaptarse a datos que llegan de forma continua, lo que es ideal para el análisis de series temporales y la detección de cambios en los datos. Se utilizan los submódulos `river.datasets` para cargar el conjunto de datos, `river.cluster`, `river.neighbors` y `river.tree` para implementar los algoritmos de aprendizaje automático, `river.drift` para detectar *concept drift* y `river.compose`, `river.metrics`, `river.preprocessing` y `river.utils` para crear la pipeline de preprocesamiento, entrenamiento y evaluación.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from river import cluster
from river import compose
from river import datasets
from river import drift
from river import metrics
from river import neighbors
from river import preprocessing
from river import tree
from river import utils
```

Conjunto de Datos: `Elec2`

El dataset `Elec2` contiene datos sobre el precio de la electricidad en Nueva Gales del Sur, Australia e incluye la transferencia de electricidad entre Nueva Gales del Sur y Victoria. El conjunto de datos presenta una tarea de clasificación binaria, donde el objetivo es predecir si el precio de la electricidad subirá o bajará en función de las características de consumo eléctrico.

El conjunto de datos contiene 45,312 instancias desde el 7 de mayo de 1996 hasta el 5 de diciembre de 1998. Cada instancia representa un período de 30 minutos, lo que equivale a 48 instancias por día.

```
# Cargar el conjunto de datos Elec2
dataset = datasets.Elec2()
```

La Tabla 1 muestra las primeras 5 filas del conjunto de datos, con sus correspondientes valores para cada una de las columnas. Las características del conjunto de datos son las siguientes:

- `date`: Fecha de la medición.
- `day`: Día de la semana.
- `period`: Hora de la medición (1-48) en intervalos de media hora a lo largo de 24 horas.
- `nswprice`: Precio de la electricidad en Nueva Gales del Sur.
- `nswdemand`: Demanda de electricidad en Nueva Gales del Sur.
- `vicprice`: Precio de la electricidad en Victoria.
- `vicdemand`: Demanda de electricidad en Victoria.
- `transfer`: Transferencia programada de electricidad entre ambos estados.
- `target`: Variable objetivo que identifica el cambio de precio (UP = 1 o DOWN = 0) en Nueva Gales del Sur en relación con un promedio móvil de las últimas 24 horas.

La Tabla 2 proporciona un resumen estadístico detallado de las columnas del conjunto de datos. Este análisis destaca que todas las variables numéricas están normalizadas entre 0 y 1, lo que facilita la comparación entre ellas y mejora el rendimiento de los algoritmos de aprendizaje automático. Además, esta normalización ayuda a mitigar el impacto de valores extremos o anomalías en los datos.

Por otro lado, la Tabla 3 muestra que la variable objetivo `target` está razonablemente balanceada, con un 42% de ejemplos positivos (`True`) y un 58% de ejemplos negativos (`False`). Este balance es crucial para evitar que los modelos de clasificación desarrollen un sesgo hacia una clase específica, lo que podría comprometer su capacidad de generalización.

Finalmente, la Figura 1 ilustra cómo evolucionan las variables relacionadas con el precio y la demanda a lo largo de la serie temporal. Se observa que la distribución de estas variables cambia con el tiempo, lo que sugiere la presencia de *concept drift*. Además, se destaca que la variable `vicprice` presenta valores muy bajos debido a la normalización, que incluye un valor extremadamente alto, posiblemente anómalo.

Tabla 1: Primeras 5 filas del dataset Elec2

	date	day	period	nswprice	nswdemand	vicprice	vicdemand	transfer	target
0	0.0	2	0.000000	0.056443	0.439155	0.003467	0.422915	0.414912	True
1	0.0	2	0.021277	0.051699	0.415055	0.003467	0.422915	0.414912	True
2	0.0	2	0.042553	0.051489	0.385004	0.003467	0.422915	0.414912	True
3	0.0	2	0.063830	0.045485	0.314639	0.003467	0.422915	0.414912	True
4	0.0	2	0.085106	0.042482	0.251116	0.003467	0.422915	0.414912	False

Tabla 2: Información del dataset Elec2

	count	mean	std	min	max	unique
date	45312	0.50	0.34	0.0	1.0	933
day	45312	4.00	2.00	1.0	7.0	7
period	45312	0.50	0.29	0.0	1.0	48
nswprice	45312	0.06	0.04	0.0	1.0	4089
nswdemand	45312	0.43	0.16	0.0	1.0	5266
vicprice	45312	0.00	0.01	0.0	1.0	3798
vicdemand	45312	0.42	0.12	0.0	1.0	2846
transfer	45312	0.50	0.15	0.0	1.0	1878

Tabla 3: Distribución de clases de la variable objetivo en el dataset Elec2

Tabla 3

	Value	Proportion
0	False	58.0%
1	True	42.0%

Dado que se ha observado que el conjunto de datos presenta *concept drift*, se ha realizado un análisis de el concepto de deriva en las variables individuales del conjunto de datos. Para ello, se ha utilizado la técnica Adaptive Windowing (ADWIN) para detectar cambios en la distribución de las variables a lo largo del tiempo. Esta técnica permite identificar si hay cambios significativos en la media de una secuencia de datos, lo que puede indicar la presencia de *concept drift*. En la Figura 2 se muestran los resultados de este análisis, que indica que las variables `nswprice`, `nswdemand` y `vicdemand` presentan cambios significativos en su distribución a lo largo del tiempo, lo que sugiere la presencia de *concept drift* en estas variables. La variable `vicprice`, por otro lado, no muestra cambios significativos en su distribución mediante la técnica ADWIN. Esto se puede deber a que la variable `vicprice` tiene un rango de valores muy bajo debido a la normalización, lo que puede dificultar la detección de cambios significativos en su distribución.

```
# Cargar el conjunto de datos
dataset = datasets.Elec2()
# Columnas a analizar
features = ["nswprice", "nswdemand", "vicprice", "vicdemand"]
# Inicializar el detector de cambio ADWIN para cada característica
adwins = {f: drift.ADWIN() for f in features}
# Inicializar listas para almacenar los puntos de cambio detectados y los valores
drift_points = {f: [] for f in features}
values = {f: [] for f in features}
time_steps = []
```

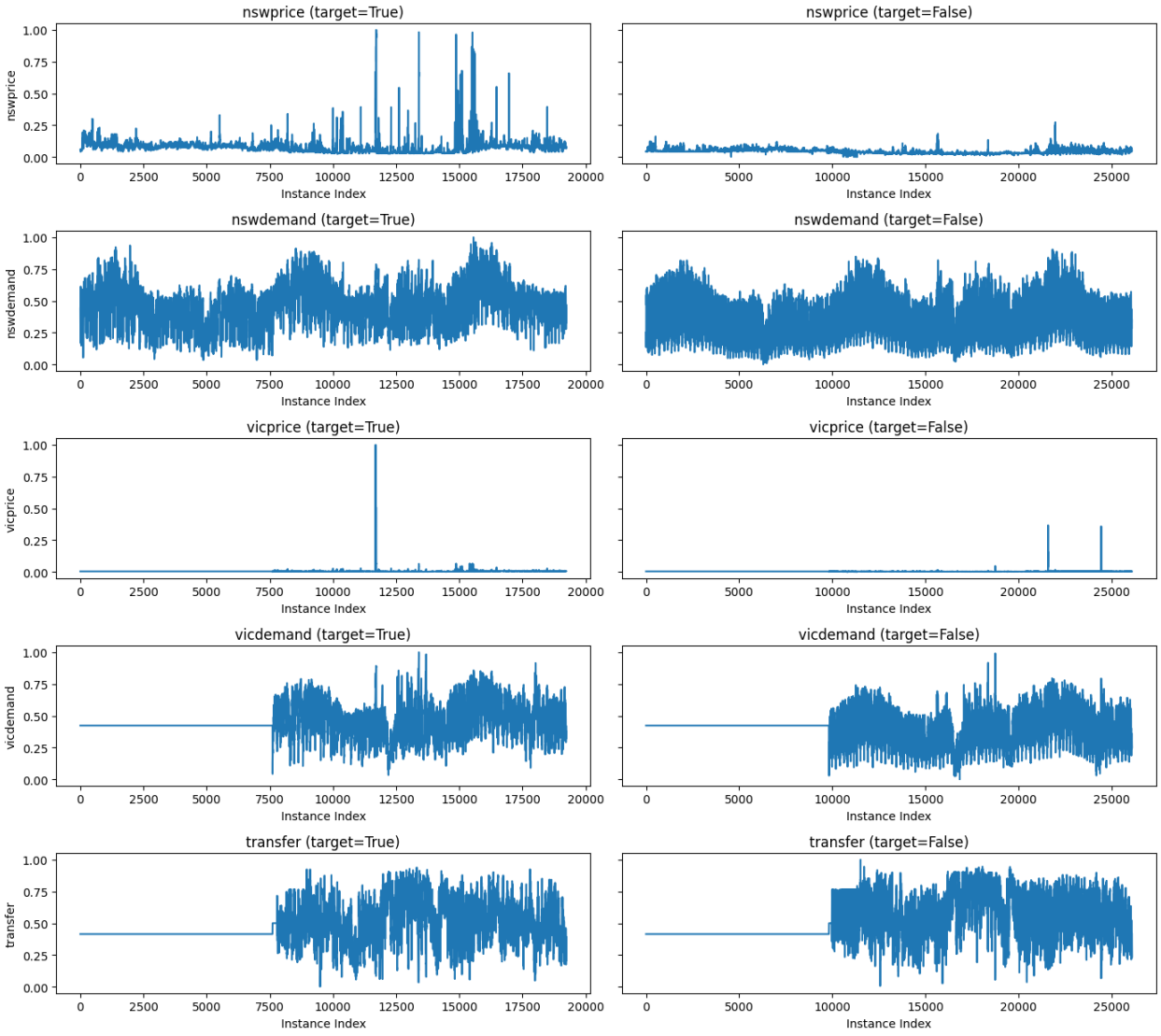

Figura 1: Distribución de variables del dataset Elec2

```python
# Iterar sobre el conjunto de datos
for i, (x, _) in enumerate(dataset):
    time_steps.append(i)
    # Actualizar el detector de cambio para cada característica
    for f in features:
        val = x[f]
        values[f].append(val)
        adwin = adwins[f]
        adwin.update(val)
        # Verificar si se ha detectado un cambio
        if adwin.drift_detected:
            drift_points[f].append(i)
```

Figura 2: Detección de Drift en algunas variables del dataset Elec2

1 Técnica de Evaluación: *Holdout*

Con el objetivo de crear dos modelos de aprendizaje automático en línea para la clasificación del dataset Elec2 mediante el método de evaluación *holdout*, se ha desarrollado la función `train_holdout`. Esta función permite evaluar periódicamente un modelo dado, utilizando un conjunto de prueba dinámico, a partir de un flujo de datos continuo.

La evaluación *holdout* se realiza con una proporción del 70% de las instancias para entrenamiento y el 30% restante para prueba. Los parámetros predeterminados de la función `train_holdout` están configurados para que, cada 1000 instancias procesadas, el modelo se entrene con las primeras 700 (70%) y se evalúe con las siguientes 300 (30%). En la Figura 3 se ilustran las particiones de entrenamiento y prueba realizadas en el conjunto de datos.

```python
def train_holdout(
    model, stream, metric_list, evaluation_interval=1000, test_window_size=300
):
    # Convertir el stream a un iterador
    stream_iter = iter(stream)
    steps = []
    results = []
    i = 1
    try:
        while True:
            x, y = next(stream_iter)
            # Aprender de una instancia
            model.learn_one(x, y)
            i += 1
            # Evaluar el modelo cada evaluation_interval
            if (i + test_window_size) % evaluation_interval == 0:
                # Obtener un conjunto de prueba de test_window_size instancias
                test_set = [next(stream_iter) for _ in range(test_window_size)]
                # Reiniciar las métricas
                for metric in metric_list:
                    metric.__init__()
                # Evaluar el modelo en el conjunto de prueba
                for x_test, y_test in test_set:
                    y_pred = model.predict_one(x_test)
                    # Actualizar las métricas
                    for metric in metric_list:
                        if y_pred is not None:
                            metric.update(y_test, y_pred)
                i += test_window_size
                steps.append(i)
                results.append([metric.get() for metric in metric_list])
    except StopIteration:
        pass
    return steps, results
```

Figura 3: Secciones del conjunto de datos utilizadas para entrenamiento y test

1.1 Hoeffding Tree con *Holdout*

El primer modelo utilizado es un clasificador basado en un Hoeffding Tree, un árbol de decisión diseñado para flujos de datos. Este modelo no almacena las instancias pasadas, sino que guarda únicamente la información necesaria para actualizar el árbol y realizar predicciones de manera eficiente.

El modelo se entrena con el conjunto de datos y se evalúa mediante la estrategia *Holdout*, utilizando las métricas de precisión (*accuracy*) y *F1-score*. Además, se incluye un preprocesamiento con `StandardScaler` para normalizar los datos y mejorar el rendimiento del modelo.

En la Figura 4 se observa la evolución de ambas métricas en cada evaluación periódica (cada 1000 instancias procesadas). Los resultados muestran fluctuaciones significativas en las métricas, posiblemente debido al *concept drift* presente en los datos. La precisión varía entre ~0.60 y ~0.95, mientras que el *F1-score* oscila entre ~0.20 y ~0.95. Esto sugiere que el modelo enfrenta dificultades para adaptarse a los cambios en la distribución de los datos a lo largo del tiempo.

```
datatset_stream = datasets.Elec2()
hoeffding_model = compose.Pipeline(
    preprocessing.StandardScaler(), tree.HoeffdingTreeClassifier()
)
metrics_list = [metrics.Accuracy(), metrics.F1(pos_val=True)]

hoeffding_holdout_steps, hoeffding_holdout_results = train_holdout(
    hoeffding_model, datatset_stream, metrics_list
)
```

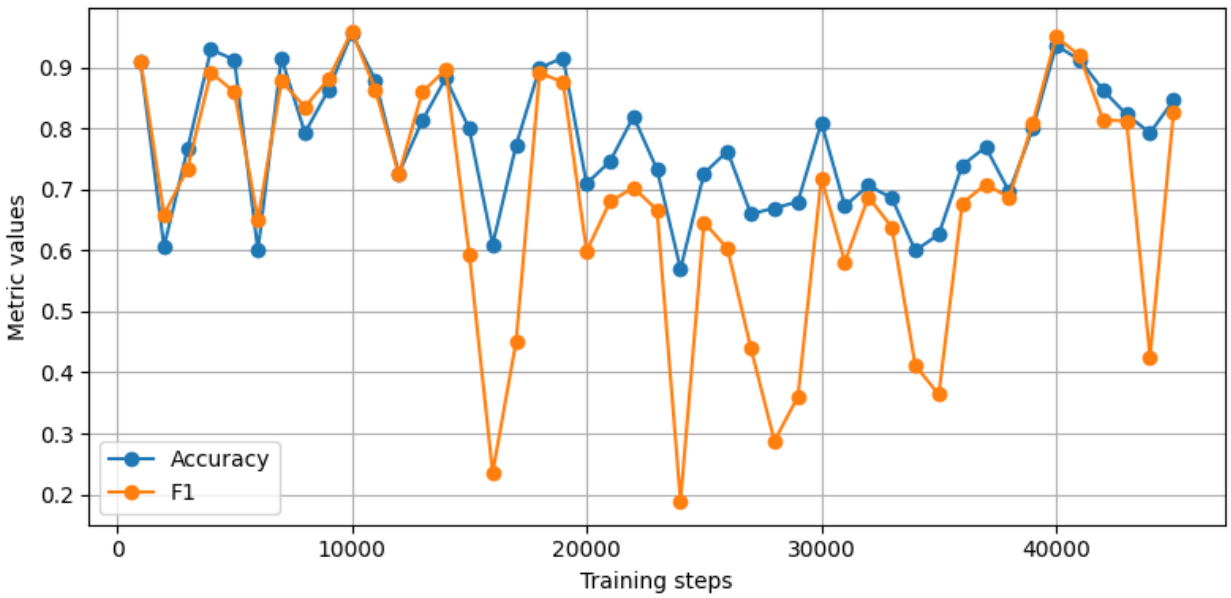

Figura 4: Resultados de la evaluación del modelo Hoeffding Tree utilizando el método de Holdout

1.2 KNN con *Holdout*

El segundo modelo entrenado es un clasificador KNN (*K-Nearest Neighbors*), basado en la premisa de que las instancias similares estarán cercanas en el espacio de características. Este modelo se entrena y evalúa utilizando las mismas técnicas y métricas que el modelo anterior.

La Figura 5 muestra la evolución de las métricas de precisión (*accuracy*) y *F1-score* en cada evaluación periódica. Al igual que el Hoeffding Tree, ambas métricas presentan fluctuaciones significativas. Los valores de precisión oscilan entre ~0.60 y ~0.90, mientras que el *F1-score* varía entre ~0.30 y ~0.90. Esto indica que el modelo KNN también enfrenta dificultades para adaptarse a los cambios en la distribución de los datos.

```
datatset_stream = datasets.Elec2()
neighbors_model = compose.Pipeline(
    preprocessing.StandardScaler(), neighbors.KNNClassifier()
)
metrics_list = [metrics.Accuracy(), metrics.F1(pos_val=True)]

neighbors_holdout_steps, neighbors_holdout_results = train_holdout(
    neighbors_model, datatset_stream, metrics_list
)
```

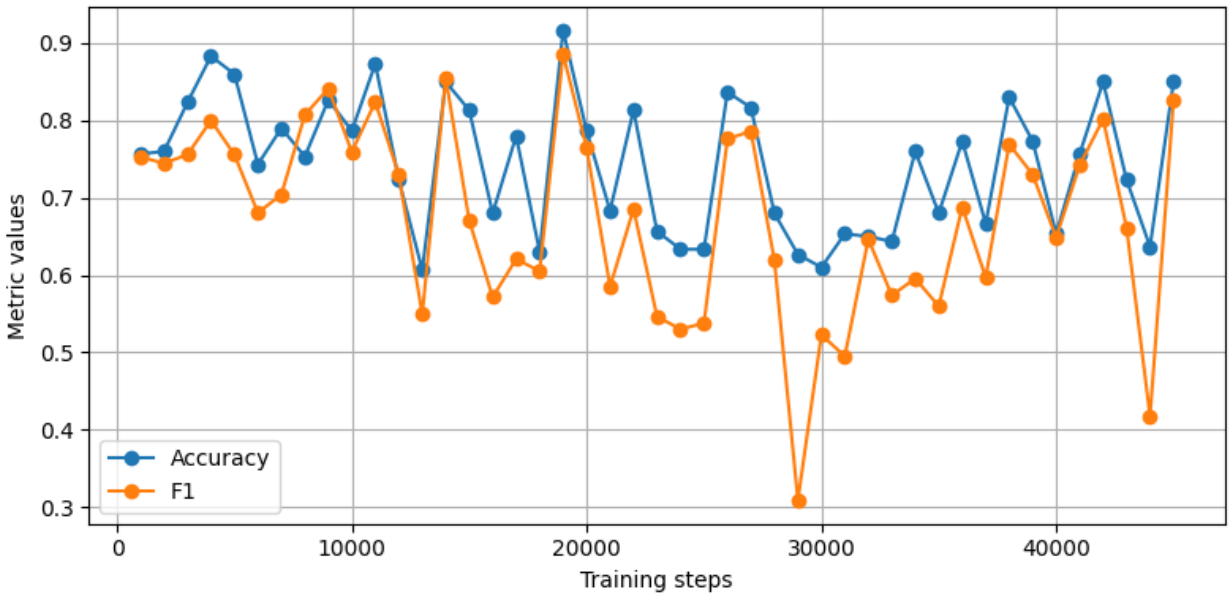

Figura 5: Resultados de la evaluación del modelo KNN utilizando el método de Holdout

1.3 Comparación de Hoeffding Tree y KNN con *Holdout*

Para la comparación de ambos modelos, se han obtenido el último valor y el valor promedio de las métricas de precisión (*accuracy*) y *F1-score* en la evaluación *holdout*. La Figura 6 muestra los resultados obtenidos para ambos modelos. Ambos modelos presentan un rendimiento similar en la última fase del entrenamiento y evaluación. Sin embargo, el modelo basado en un Hoeffding Tree muestra un rendimiento ligeramente superior en el promedio de ambas métricas a lo largo de todo el proceso. Esto sugiere que, aunque ambos modelos son capaces de adaptarse a los cambios en la distribución de los datos, el Hoeffding Tree puede ser más efectivo en este contexto específico.

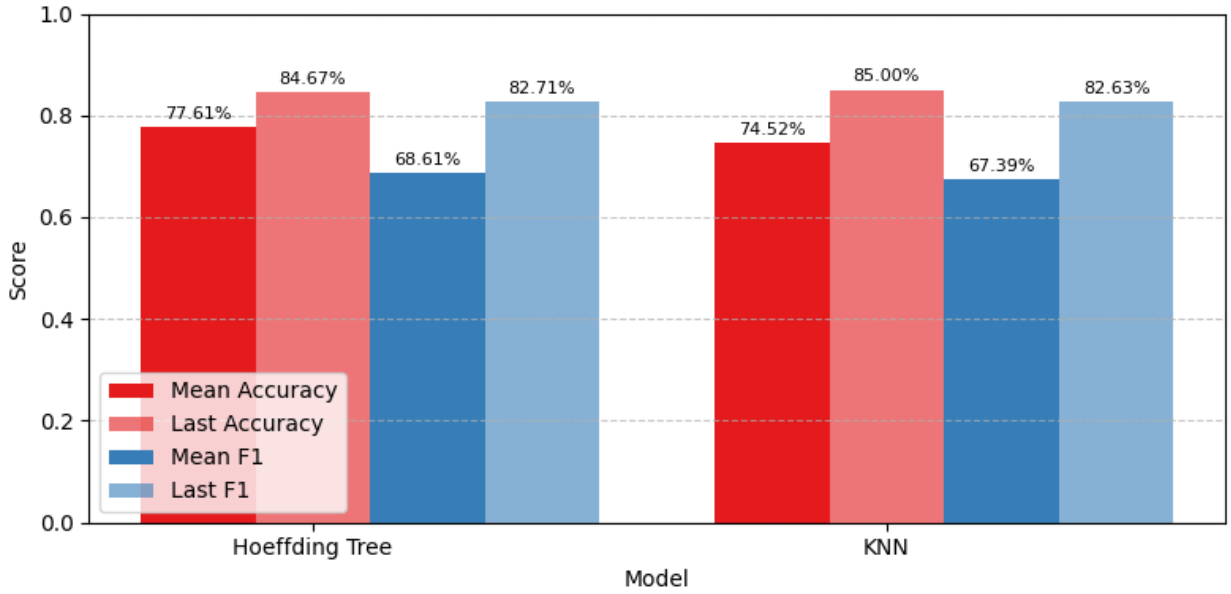

Figura 6: Comparación de resultados de modelos utilizando el método de Holdout

2 Técnica de Evaluación: *Prequential*

A continuación, se entrenan y evalúan los modelos utilizando la técnica de evaluación *prequential*. Esta técnica permite evaluar el rendimiento del modelo en tiempo real, utilizando cada instancia a medida que llega. Primero, se evalúa el modelo con la instancia actual y luego se utiliza esa misma instancia para actualizar el modelo. La función `train_prequential` implementa esta técnica, permitiendo evaluar el rendimiento del modelo en tiempo real.

```
def train_prequential(model, stream, metric_list, evaluation_interval=1000):
    steps = []
    results = []
    # Iterar sobre el flujo de datos
    for i, (x, y) in enumerate(stream, start=1):
        # Realizar la predicción
        y_pred = model.predict_one(x)
        # Actualizar las métricas
        for metric in metric_list:
            if y_pred is not None:
                metric.update(y, y_pred)
        # Reportar métricas de evaluación cada evaluation_interval
        if i % evaluation_interval == 0:
            results.append([metric.get() for metric in metric_list])
            steps.append(i)
        # Aprender de la instancia
        model.learn_one(x, y)
    return steps, results
```

2.1 Hoeffding Tree con *Prequential*

El primer modelo utilizado es un clasificador basado en un Hoeffding Tree, que se entrena y evalúa utilizando la técnica *prequential*. Al igual que en el caso anterior, se incluye un preprocesamiento con `StandardScaler` para normalizar los datos y mejorar el rendimiento del modelo.

Dado que, en el caso del entrenamiento con la técnica *prequential*, el modelo se evalúa cada vez que llega una nueva instancia, las métricas de precisión (*accuracy*) y *F1-score* se actualizan en cada iteración. Con el objetivo de que las métricas sean comparables con las obtenidas para el entrenamiento con *Holdout*, se utilizan métricas con una ventana de 1000 instancias. Esto significa que, cada 1000 instancias procesadas, se calcula el promedio de las métricas para evaluar el rendimiento del modelo en ese período.

La Figura 7 muestra la evolución de ambas métricas en cada evaluación periódica (cada 1000 instancias procesadas). Los resultados también muestran fluctuaciones significativas en las métricas, aunque no tan marcadas como en el caso de los modelos entrenados con *Holdout*. La precisión varía entre ~0.70 y ~0.90, mientras que el *F1-score* oscila entre ~0.50 y ~0.90.

```
datatset_stream = datasets.Elec2()
hoeffding_model = compose.Pipeline(
    preprocessing.StandardScaler(), tree.HoeffdingTreeClassifier()
)
metrics_list = [
    utils.Rolling(metrics.Accuracy(), window_size=1000),
    utils.Rolling(metrics.F1(pos_val=True), window_size=1000),
]

hoeffding_prequential_steps, hoeffding_prequential_results = train_prequential(
    hoeffding_model, datatset_stream, metrics_list
)
```

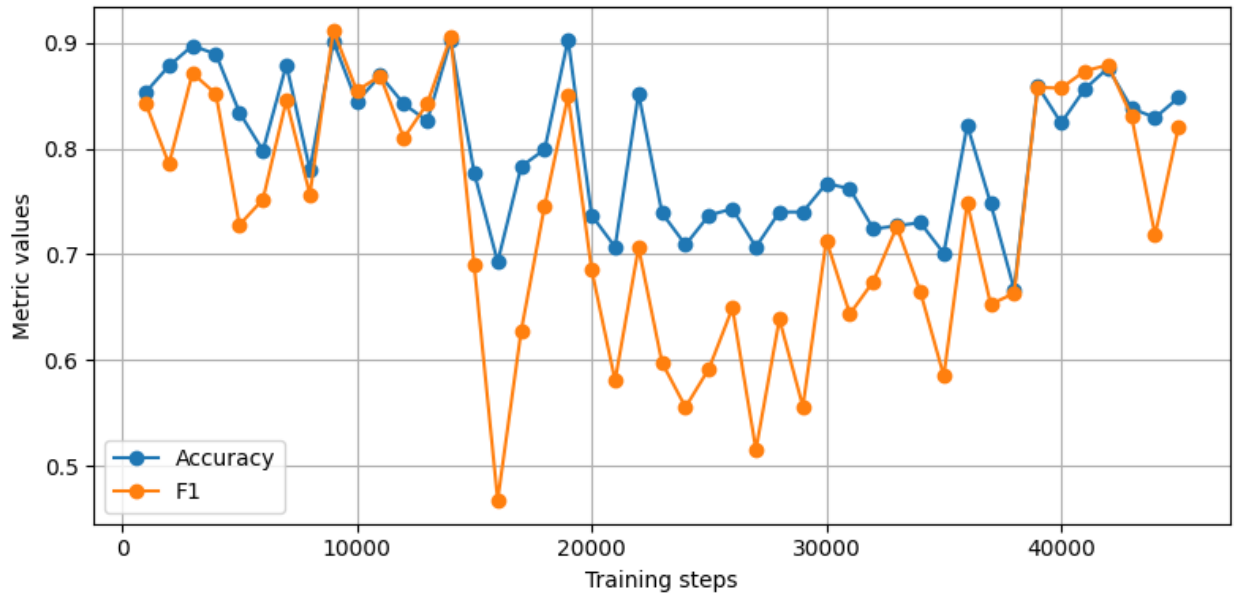

Figura 7: Resultados de la evaluación del modelo Hoeffding Tree utilizando el método Prequential

2.2 KNN con *Prequential*

El modelo KNN también se entrena y evalúa utilizando la técnica *prequential* con la misma configuración y métricas de evaluación que el modelo anterior. La Figura 8 muestra la evolución de las métricas de precisión (*accuracy*) y *F1-score* en cada evaluación periódica. Los valores de precisión oscilan entre ~0.75 y ~0.95, mientras que el *F1-score* varía entre ~0.70 y ~0.95.

```
datatset_stream = datasets.Elec2()
neighbors_model = compose.Pipeline(
    preprocessing.StandardScaler(), neighbors.KNNClassifier()
)
metrics_list = [
    utils.Rolling(metrics.Accuracy(), window_size=1000),
    utils.Rolling(metrics.F1(pos_val=True), window_size=1000),
]

neighbors_prequential_steps, neighbors_prequential_results = train_prequential(
    neighbors_model, datatset_stream, metrics_list
)
```

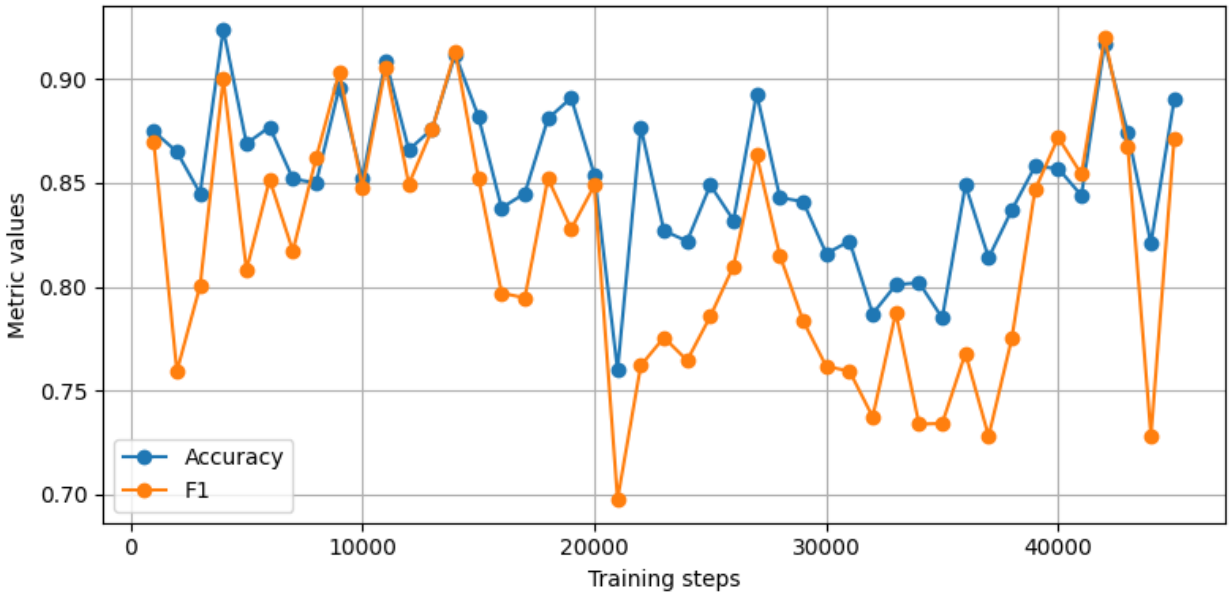

Figura 8: Resultados de la evaluación del modelo KNN utilizando el método Prequential

2.3 Comparación de Hoeffding Tree y KNN con *Prequential*

La Figura 9 muestra los resultados obtenidos para ambos modelos. En este caso, el modelo KNN presenta un rendimiento superior en ambas métricas, tanto en la última fase del entrenamiento como en el promedio de todas las fases. Esto sugiere que el modelo KNN es más efectivo en este contexto específico y puede adaptarse mejor a los cambios en la distribución de los datos a lo largo del tiempo.

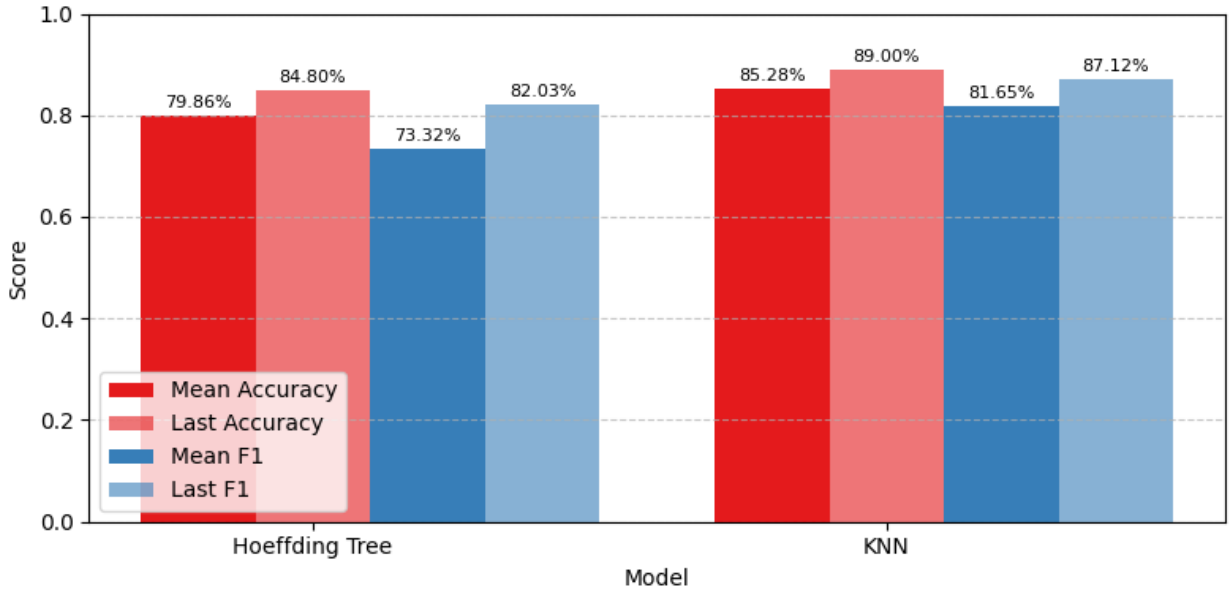

Figura 9: Comparación de resultados de modelos utilizando el método Prequential

3 Detección de *Concept Drift*

Como se ha observado en el análisis del conjunto de datos, el dataset `Elec2` presenta *concept drift*. Además, se ha mencionado que el modelo Hoeffding Tree y el modelo KNN presentan fluctuaciones significativas en las métricas de evaluación, lo que sugiere que ambos modelos enfrentan dificultades para adaptarse a los cambios en la distribución de los datos a lo largo del tiempo.

La función `train_prequential_drift` implementa un enfoque de evaluación *prequential* con detección de *concept drift* utilizando el algoritmo ADWIN. Durante el proceso, el modelo se entrena y evalúa iterativamente sobre un flujo de datos continuo. En cada iteración, se realiza una predicción con el modelo actual y se actualizan las métricas de evaluación proporcionadas. Simultáneamente, se utiliza ADWIN para detectar cambios significativos en la distribución de los datos, basándose en los errores de predicción. Si se detecta un *drift*, el detector se reinicia y el modelo se restablece a su estado original para adaptarse a la nueva distribución de los datos.

```
def train_prequential_drift(model, stream, metric_list, evaluation_interval=1000):
    steps = []
    results = []
    # Inicializar el detector de drift
    drift_detector = drift.ADWIN()
    drift_points = []
    # Guardar el modelo original para reiniciarlo
    original_model = model.clone()
    # Iterar sobre el flujo de datos
    for i, (x, y) in enumerate(stream, start=1):
        # Realizar la predicción
        y_pred = model.predict_one(x)
        # Actualizar las métricas
        for metric in metric_list:
            if y_pred is not None:
                metric.update(y, y_pred)
        # Detectar drift
        if y_pred is not None:
            error = 0 if y == y_pred else 1
            drift_detector.update(error)
            if drift_detector.drift_detected:
                drift_points.append(i)
                # Reiniciar el detector de drift
                drift_detector = drift.ADWIN()
                # Reiniciar el modelo
                model = original_model.clone()
        # Reportar métricas de evaluación cada evaluation_interval
        if i % evaluation_interval == 0:
            results.append([metric.get() for metric in metric_list])
            steps.append(i)
        # Aprender de la instancia
        model.learn_one(x, y)
    return steps, results, drift_points
```

3.1 Hoeffding Tree con Detección de *Concept Drift*

Se ha entrenado un modelo Hoeffding Tree con detección de *concept drift* utilizando la función `train_prequential_drift`. Se utiliza el mismo preprocesado y métricas de evaluación que en la sección anterior. La Figura 10 muestra la evolución de las métricas de precisión y *F1-score* en cada evaluación periódica. Los resultados muestran que el modelo Hoeffding Tree con detección de *concept drift* presenta un rendimiento más estable en comparación con el modelo sin detección de *drift*. La precisión varía entre ~0.70 y ~0.90, mientras que el *F1-score* oscila entre ~0.55 y ~0.90. Esto sugiere que la detección de *concept drift* permite al modelo adaptarse mejor a los cambios en la distribución de los datos a lo largo del tiempo.

En cuanto a la detección de *concept drift*, la figura resalta los momentos en los que se detectó un cambio significativo en la distribución de los datos. Estos momentos se indican con líneas verticales en la gráfica, lo que permite observar cómo el modelo se adapta a los cambios en la distribución de los datos a lo largo del tiempo. En el caso del Hoeffding Tree, se detectaron muchos cambios, lo que sugiere que el modelo es muy sensible a los cambios en la distribución de los datos. Los cambios detectados son tan abundantes que no permiten visualizar realmente los cambios en las métricas al reiniciar el modelo.

```
datatset_stream = datasets.Elec2()
hoeffding_model = compose.Pipeline(
    preprocessing.StandardScaler(), tree.HoeffdingTreeClassifier()
)
metrics_list = [
    utils.Rolling(metrics.Accuracy(), window_size=1000),
    utils.Rolling(metrics.F1(pos_val=True), window_size=1000),
]

(
    hoeffding_prequential_drift_steps,
    hoeffding_prequential_drift_results,
    hoeffding_drift_points,
) = train_prequential_drift(hoeffding_model, datatset_stream, metrics_list)
```

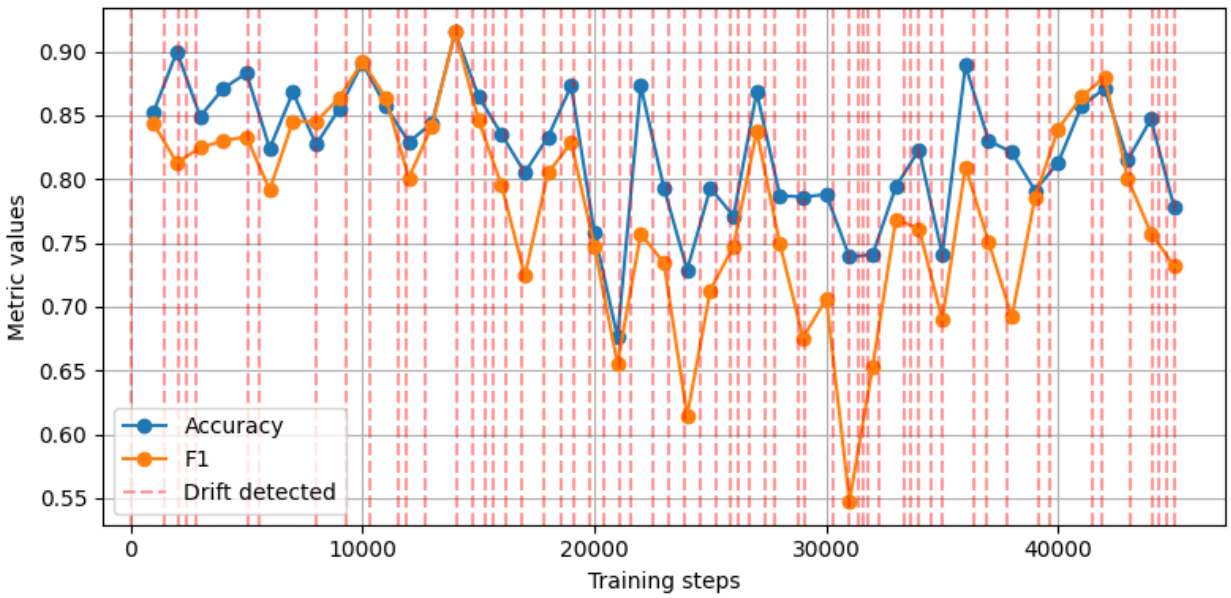

Figura 10: Resultados de la evaluación del modelo Hoeffding Tree utilizando el método Prequential con detección de Drift

3.2 KNN con Detección de *Concept Drift*

El modelo KNN también se entrena y evalúa utilizando la función `train_prequential_drift` con detección de *concept drift*. La Figura 11 muestra la evolución de las métricas de precisión y *F1-score* en cada evaluación periódica. Los resultados muestran que el modelo KNN con detección de *concept drift* presenta un rendimiento significativamente más estable en comparación con el modelo sin detección de *drift*. La precisión varía entre ~0.80 y ~0.95, mientras que el *F1-score* oscila entre ~0.75 y ~0.95.

La detección de *concept drift* en el modelo KNN también muestra momentos en los que se detectó un cambio significativo en la distribución de los datos. En el caso del KNN, se detectaron menos cambios que en el caso del Hoeffding Tree, lo que sugiere que el modelo es menos sensible a los cambios en la distribución de los datos. Además, tras los cambios marcados por las líneas verticales, se observa un cambio significativo en la tendencia de las métricas, lo que indica que el cambio en la distribución de los datos ha afectado al rendimiento del modelo.

```
datatset_stream = datasets.Elec2()
neighbors_model = compose.Pipeline(
    preprocessing.StandardScaler(), neighbors.KNNClassifier()
)
metrics_list = [
    utils.Rolling(metrics.Accuracy(), window_size=1000),
    utils.Rolling(metrics.F1(pos_val=True), window_size=1000),
]

neighbors_prequential_steps, neighbors_prequential_results, neighbors_drift_points = (
    train_prequential_drift(neighbors_model, datatset_stream, metrics_list)
)
```

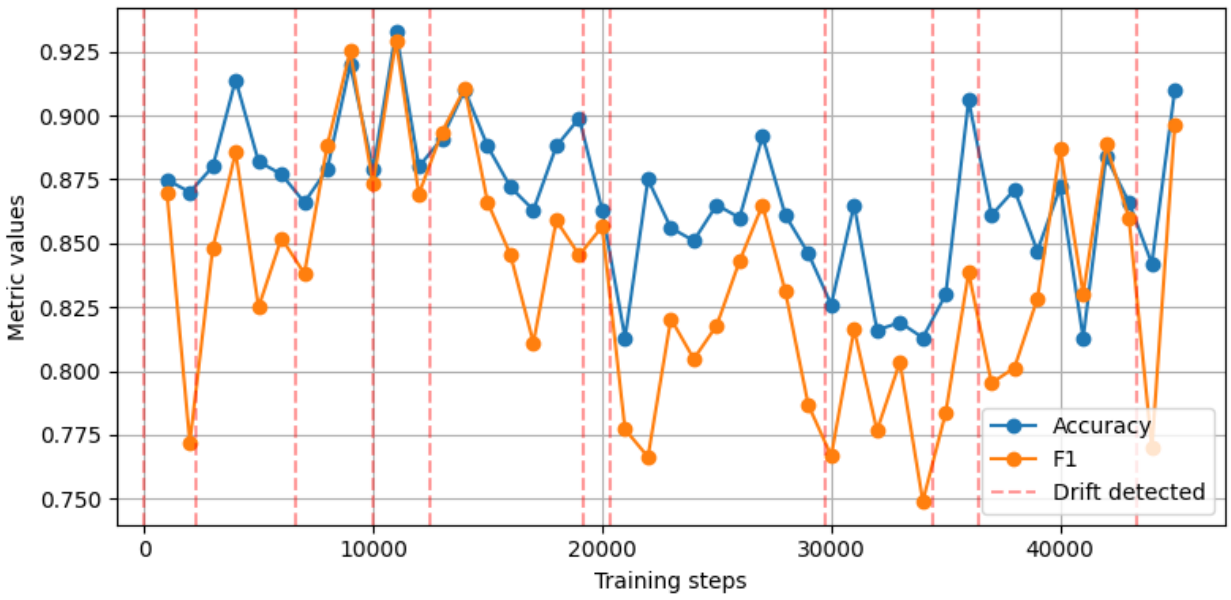

Figura 11: Resultados de la evaluación del modelo KNN utilizando el método Prequential con detección de Drift

3.3 Comparación de Hoeffding Tree y KNN con Detección de *Concept Drift*

La Figura 12 muestra los resultados obtenidos para ambos modelos. En este caso, el modelo KNN presenta un rendimiento muy superior en ambas métricas, tanto en la última fase del entrenamiento como en el promedio de todas las fases. Esto sugiere que el modelo KNN es más efectivo en este contexto específico y puede adaptarse mejor a los cambios en la distribución de los datos a lo largo del tiempo. Además, se ha visto como el modelo KNN, detectaba menos cambios en la distribución de sus errores que el modelo Hoeffding Tree, lo que sugiere que el modelo KNN es menos sensible a los cambios en la distribución de los datos. Esto puede ser una ventaja en algunos casos, ya que un modelo menos sensible a los cambios puede ser más robusto y menos propenso a sobreajustarse a los datos.

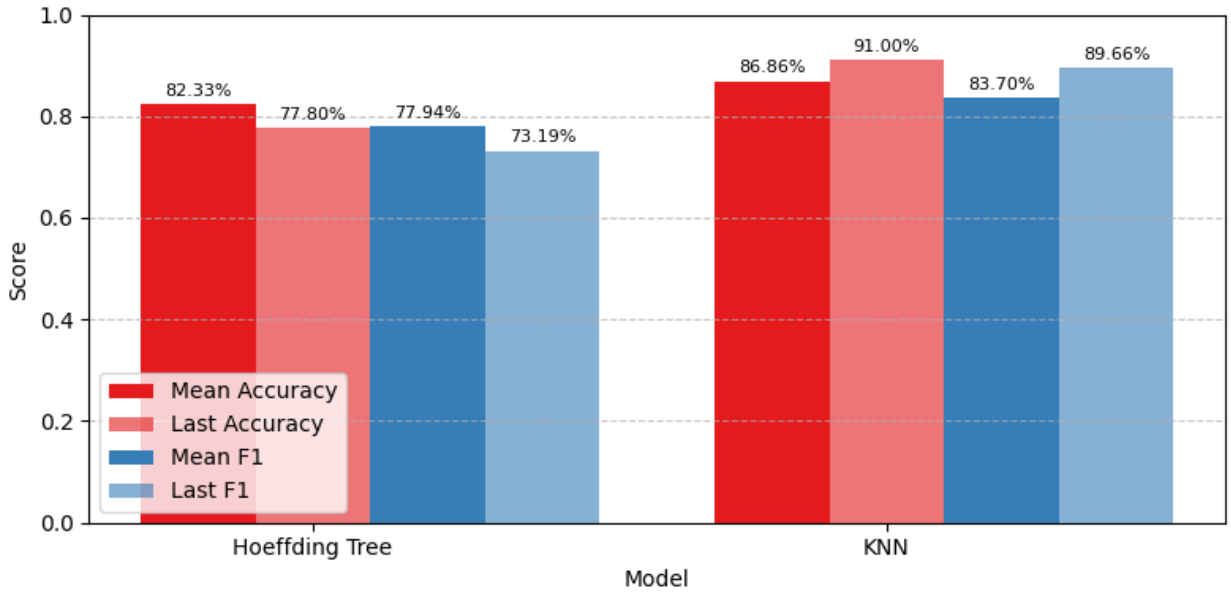

Figura 12: Comparación de resultados de modelos utilizando el método Prequential con detección de Drift

4 Técnicas de *Clustering*

Por último, se han implementado dos técnicas de *clustering* para analizar el conjunto de datos `Elec2`. Estas técnicas permiten agrupar las instancias en función de sus características, lo que puede ayudar a identificar patrones y tendencias en los datos. Las técnicas de *clustering* utilizadas son CluStream y DenStream, ambas implementadas en el módulo `river.cluster`. Para el entrenamiento y evaluación de los modelos de *clustering*, se ha utilizado la función `train_clustering` con la técnica *Prequential*, que permite evaluar el rendimiento del modelo en tiempo real, utilizando cada instancia a medida que llega.

```python
def train_clustering(model, stream, metrics_list, evaluation_interval=1000):
    steps = []
    results = []
    # Iterar sobre el flujo de datos
    for i, (x, y) in enumerate(stream, start=1):
        # Realizar la predicción
        y_pred = model.predict_one(x)
        # Actualizar las métricas
        for metric in metrics_list:
            if len(model.centers) > 1 and y_pred >= 0:
                metric.update(x, y_pred, model.centers)
        # Reportar métricas de evaluación cada evaluation_interval
        if i % evaluation_interval == 0:
            results.append([metric.get() for metric in metrics_list])
            steps.append(i)
        # Aprender de la instancia
        model.learn_one(x)
    return steps, results
```

4.1 Clustering con CluStream

El primer modelo de *clustering* utilizado es CluStream, un algoritmo de *clustering* diseñado para trabajar con flujos de datos. Para evaluar el rendimiento del modelo de *clustering* se utiliza la métrica de Silhouette, que mide la calidad del agrupamiento. La métrica de Silhouette varía entre -1 y 1, donde un valor cercano a 1 indica que las instancias están bien agrupadas y separadas de otros grupos, mientras que un valor cercano a -1 indica que las instancias están mal agrupadas. Al igual que los modelos anteriores, se reporta su rendimiento cada 1000 instancias procesadas.

La Figura 13 muestra la evolución de la métrica de Silhouette en cada evaluación periódica. Los resultados muestran que el modelo CluStream presenta un rendimiento estable, donde en las primeras 10000 iteraciones rápidamente baja a por debajo de 0.275 y se estabiliza en torno a 0.29, donde se mantiene con una tendencia ascendente hasta llegar a 0.31 al final del entrenamiento. Esto sugiere que el modelo CluStream es capaz de adaptarse a los cambios en la distribución de los datos a lo largo del tiempo y mejorar su rendimiento a medida que se procesan más instancias.

```
datatset_stream = datasets.Elec2()
clustream_model = cluster.CluStream()
metrics_list = [metrics.Silhouette()]

clustream_steps, clustream_results = train_clustering(
    clustream_model, datatset_stream, metrics_list
)
```

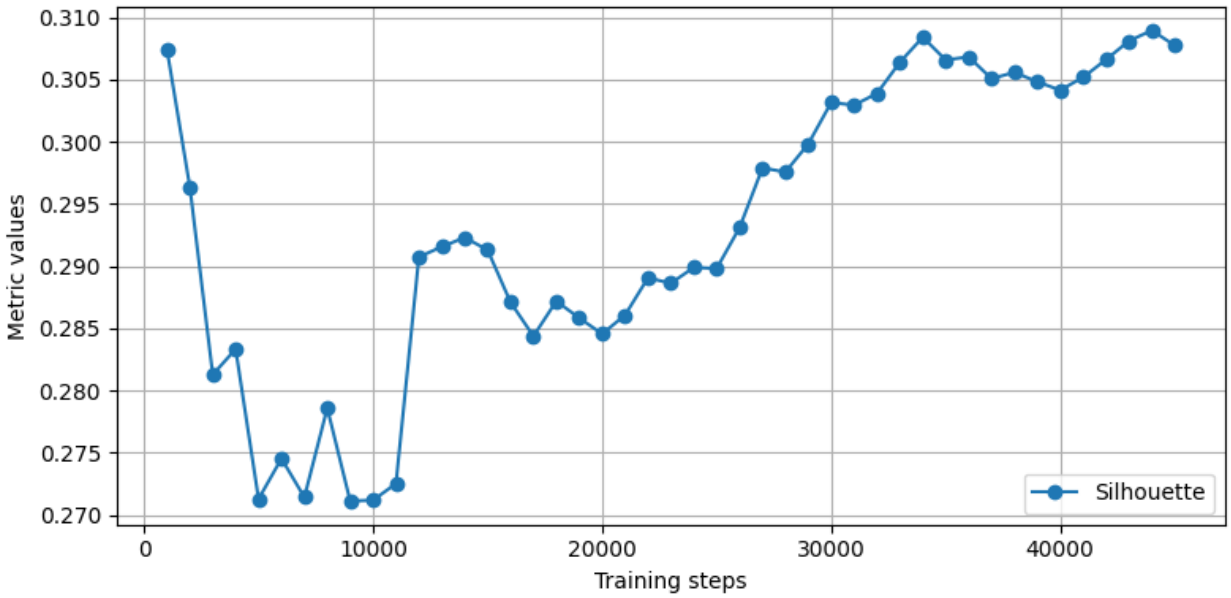

Figura 13: Resultados de la evaluación del modelo CluStream

4.2 Clustering con DenStream

El segundo modelo de *clustering* utilizado es DenStream, otro algoritmo de *clustering* diseñado para trabajar con flujos de datos. Al igual que en el caso anterior, se utiliza la métrica de Silhouette para evaluar el rendimiento del modelo de *clustering*. La Figura 14 muestra la evolución de la métrica de Silhouette en cada evaluación periódica. Los resultados muestran que el modelo DenStream presenta un rendimiento más estable frente a CluStream, donde en las primeras 10000 iteraciones baja a por debajo de 0.32 e inmediatamente vuelve a subir con una tendencia ascendente hasta llegar a 0.36 al final del entrenamiento.

```
datatset_stream = datasets.Elec2()
denstream_model = cluster.DenStream()
metrics_list = [metrics.Silhouette()]

denstream_steps, denstream_results = train_clustering(
    denstream_model, datatset_stream, metrics_list
)
```

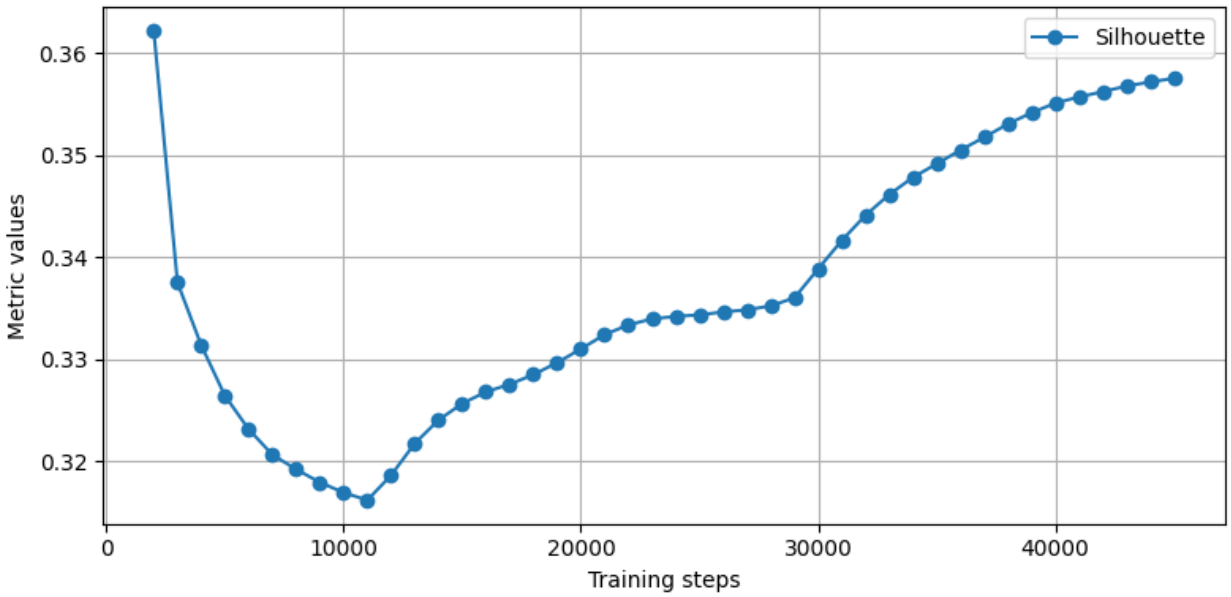

Figura 14: Resultados de la evaluación del modelo DenStream

4.3 Comparación de Clustering con CluStream y DenStream

La Figura 15 muestra los resultados obtenidos para ambos modelos. En este caso, el modelo DenStream presenta un rendimiento superior en la métrica de Silhouette, lo que sugiere que el modelo DenStream es más efectivo en este contexto específico y puede adaptarse mejor a los cambios en la distribución de los datos a lo largo del tiempo. Además, se ha visto como el modelo DenStream, sufría menos fluctuaciones en su rendimiento que el modelo CluStream, lo que sugiere que el modelo DenStream es menos sensible a los cambios en la distribución de los datos.

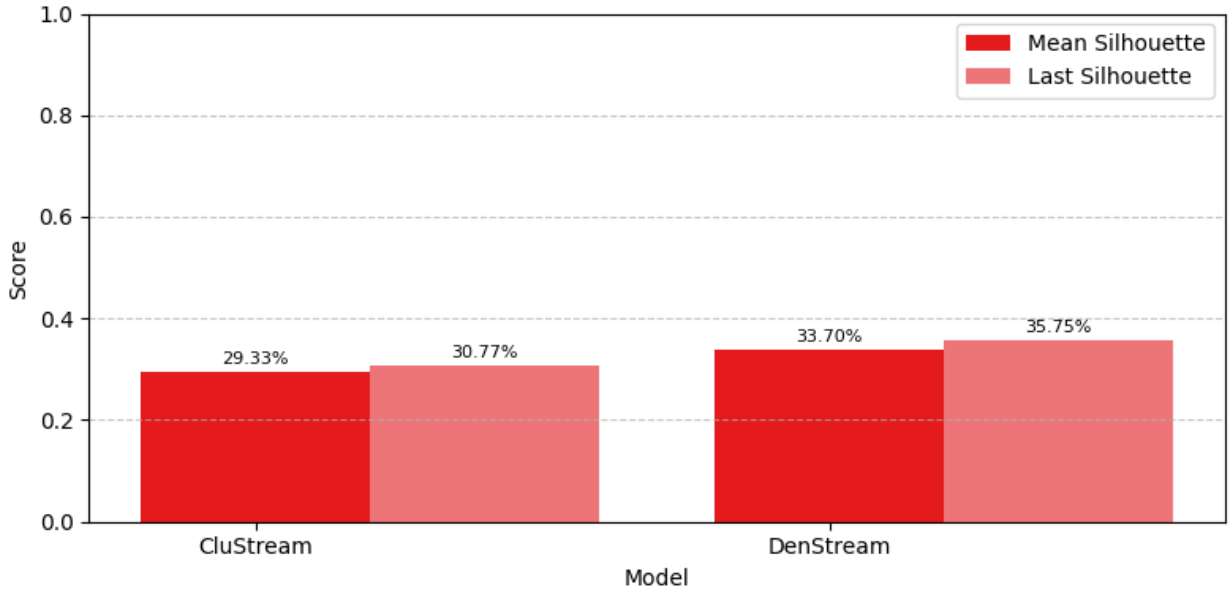

Figura 15: Comparación de resultados de modelos de clustering