Flujo KNIME: Análisis de Sentimientos con RNN (LSTM)

📊 Visión General del Workflow

Este workflow implementa un **modelo de Deep Learning con RNN (LSTM)** para análisis de sentimientos en tweets sobre aerolíneas estadounidenses, clasificándolos en **positivo, negativo o neutral**.

Dataset: Kaggle Dataset con miles de posts de redes sociales sobre 6 aerolíneas de EE.UU. (2015)

🎯 Objetivo del Workflow

Construir un modelo predictivo que clasifique automáticamente el sentimiento de tweets de clientes hacia aerolíneas usando un enfoque basado en **léxico** (conteo de palabras positivas/negativas) procesado por una **red neuronal recurrente (LSTM)**.

🔷 Estructura del Workflow: 5 Secciones Principales

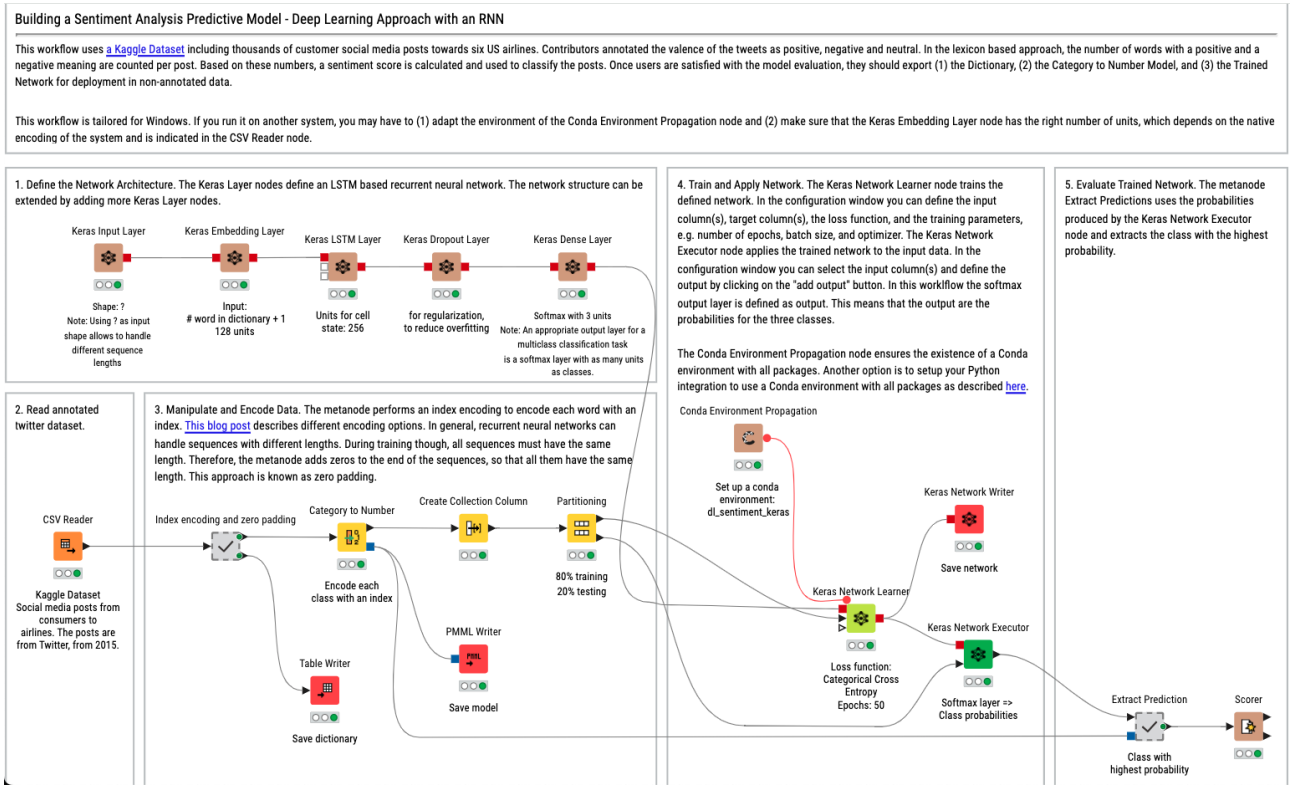
1. Define the Network Architecture

2. Read Annotated Twitter Dataset

3. Manipulate and Encode Data

4. Train and Apply Network

5. Evaluate Trained Network



📐 SECCIÓN 1: Define the Network Architecture

Arquitectura LSTM para Secuencias de Texto:

`Input Layer → Embedding Layer → LSTM Layer → Dropout Layer → Dense Layer (Softmax)`

1. Keras Input Layer

Parámetros:

- **Shape: ?**
    - El ? indica longitud variable de secuencia
    - Nota: "Using 7 as input shape allows to handle different sequence lengths"

Función:

- Acepta secuencias de texto de diferentes longitudes
- Cada secuencia representa un tweet tokenizado

2. Keras Embedding Layer

Parámetros:

- **Input: # word in dictionary + 1**
- **128 units**

Función:

- Convierte índices de palabras en vectores densos de 128 dimensiones
- Aprende representaciones semánticas de palabras durante el entrenamiento
- "+1" para manejar el token de padding (índice 0)

¿Qué hace?

`Palabra "good" (índice 42) → Vector de 128 números [0.23, -0.45, 0.67, ...]`

`Palabra "bad" (índice 89) → Vector de 128 números [-0.34, 0.12, -0.78, ...]`

---

3. Keras LSTM Layer

Parámetros:

- **Units for cell state: 256**

Función:

- Procesa la secuencia de palabras capturando dependencias temporales
- 256 unidades LSTM mantienen memoria de contexto largo
- Ideal para entender el sentimiento en el contexto completo del tweet

¿Por qué LSTM?

- Captura contexto: "not good" vs "very good"
- Maneja secuencias de longitud variable
- Recuerda información relevante a lo largo del tweet

4. Keras Dropout Layer

Parámetros:

- **For regularization, to reduce overfitting**

Función:

- Apaga aleatoriamente neuronas durante el entrenamiento
- Previene que el modelo memorice (overfitting)
- Mejora la generalización

5. Keras Dense Layer

Parámetros:

- **Softmax with 3 units**
- Nota: "Note: As appropriate output layer for a multiclass classification task"
- **Is a dense layer with as many units as classes**

Función:

- Capa de salida con 3 neuronas (3 clases)
- Softmax genera probabilidades que suman 1.0

- Clases: **Positivo**, **Negativo**, **Neutral**

🔷 SECCIÓN 2: Read Annotated Twitter Dataset

Flujo de carga:

```
CSV Reader → Kaggle Dataset (Social media posts from customers to airlines, 2015)
```

Contenido del dataset:

- Miles de tweets dirigidos a 6 aerolíneas de EE.UU.
- Tweets anotados con sentimiento: positivo, negativo, neutral
- Enfoque léxico: cuenta palabras positivas/negativas por post
- Score de sentimiento calculado basado en conteos

🔷 SECCIÓN 3: Manipulate and Encode Data

Flujo de preprocesamiento:

```
CSV Reader → Index encoding and zero padding → Category to Number →

Create Collection Column → Partitioning (80/20) → Table Writer + PMML Writer
```

Este es el procesamiento MÁS CRÍTICO para RNNs con texto.

---

1. Index Encoding and Zero Padding

Función crítica: Convierte texto en secuencias numéricas de longitud uniforme.

Proceso:

1. **Index encoding**: Cada palabra única del vocabulario recibe un índice

```
"flight" → 1

"delayed" → 2

"good" → 3

"service" → 4
```

2. **Zero padding**: Todas las secuencias se llevan a la misma longitud añadiendo ceros

```
Tweet corto: [1, 2, 3] → [1, 2, 3, 0, 0, 0, 0]

Tweet largo: [1, 2, 3, 4, 5, 6, 7] → [1, 2, 3, 4, 5, 6, 7]
```

¿Por qué?

- Las RNNs necesitan secuencias de igual longitud por batch
- El padding permite procesar tweets de diferentes tamaños juntos
- Referencia: "This blog post describes different encoding options"

2. Category to Number

Parámetros:

- **Encode each class with an index**

Función:

- Convierte etiquetas de sentimiento a números
- Ejemplo:

```
"positive" → 0

"negative" → 1

"neutral" → 2
```

3. Create Collection Column

Función:

- Agrupa la secuencia codificada con su etiqueta
- Prepara formato requerido por Keras en KNIME
- Cada fila = [secuencia de índices, etiqueta]

4. Partitioning

Parámetros:

- **80% training**
- **20% testing**

Función:

- División estándar para entrenamiento y evaluación
- Mantiene distribución de clases balanceada

5. Table Writer + PMML Writer

Parámetros:

- **Save dictionary**
- **Save model**

Función:

- **Table Writer**: Guarda el diccionario palabra→índice
- **PMML Writer**: Guarda el modelo de encoding para deployment
- Necesarios para aplicar el mismo encoding en datos nuevos

Deployment: Una vez entrenado, se exportan 3 elementos:

1. Dictionary (vocabulario)
2. Category to Number Model (encoding de clases)
3. Trained Network (red neuronal entrenada)

🔷 SECCIÓN 4: Train and Apply Network

Flujo de entrenamiento:

```
Conda Environment Propagation → Keras Network Learner →

Keras Network Executor → Keras Network Writer
```

---

1. Conda Environment Propagation

Parámetros:

- **Set up a conda environment: dl_sentiment_keras**

Función:

- Asegura que exista un entorno Conda con todas las dependencias

- Incluye TensorFlow, Keras, y bibliotecas necesarias
- Alternativa: integración Python con ambiente configurado (enlace here)

Nota técnica:

- Workflow diseñado para Windows
- En otros sistemas: ajustar Conda Environment Propagation node
- Verificar encoding del sistema en Keras Embedding Layer node

2. Keras Network Learner

Parámetros:

- **Loss function: Categorical Cross Entropy**
- **Epochs: 50**

Configuración en ventana:

- Define columnas de entrada y target
- Selecciona función de pérdida
- Configura parámetros de entrenamiento (epochs, batch size, optimizer)

Función:

- Entrena la LSTM con los datos de entrenamiento
- 50 épocas: pasa por todo el dataset 50 veces
- Categorical Cross Entropy: apropiada para clasificación multiclase

3. Keras Network Executor

Parámetros:

- **Softmax layer => Class probabilities**

Configuración:

- Selecciona columna de entrada
- Define output haciendo clic en "add output"
- Output: probabilidades para las 3 clases

Función:

- Aplica la red entrenada al conjunto de prueba
- Genera predicciones con probabilidades
- Output: [P(positivo), P(negativo), P(neutral)]

4. Keras Network Writer

Parámetros:

- **Save network**

Función:

- Guarda el modelo entrenado para deployment
- Formato estándar de Keras (.h5)
- Permite reutilizar sin reentrenar

🔷 SECCIÓN 5: Evaluate Trained Network

Flujo de evaluación:

```
Keras Network Executor → Extract Prediction → Scorer
```

1. Extract Prediction

Parámetros:

- **Class with highest probability**

Función:

- Extrae la clase con mayor probabilidad de las 3
- Ejemplo:

```
[0.05, 0.80, 0.15] → Clase "negative" (índice 1)

[0.70, 0.20, 0.10] → Clase "positive" (índice 0)
```

2. Scorer

Función:

- Evalúa el rendimiento del modelo
- Calcula métricas:

- **Accuracy**: Porcentaje de clasificaciones correctas
- **Precision, Recall, F1**: Por cada clase (pos/neg/neutral)
- **Confusion Matrix**: Distribución de errores entre clases

🎯 Concepto Clave: Enfoque Léxico + Deep Learning

¿Qué es el enfoque léxico?

Este workflow usa un enfoque **híbrido**:

1. **Preprocesamiento léxico:**
    - Cuenta palabras positivas por tweet
    - Cuenta palabras negativas por tweet
    - Calcula score de sentimiento basado en estos conteos
2. **Clasificación con LSTM:**
    - La LSTM aprende patrones en las secuencias de palabras
    - Captura contexto y dependencias temporales
    - Clasifica basándose en representaciones aprendidas

Ventaja de este enfoque:

- Combina conocimiento lingüístico (léxico) con aprendizaje automático (LSTM)
- La LSTM puede aprender matices que el léxico no captura
- Maneja negaciones, sarcasmo, y contexto complejo

🔬 Aspectos Técnicos Importantes

1. Zero Padding:

- Necesario porque tweets tienen diferentes longitudes
- Se añaden ceros al final de secuencias cortas
- La LSTM aprende a ignorar los ceros

2. Embedding Layer:

- Aprende representaciones de palabras durante el entrenamiento
- 128 dimensiones por palabra (hiperparámetro)

- Palabras similares tendrán embeddings similares

3. LSTM con 256 unidades:

- Capacidad de memoria para capturar dependencias largas
- Cell state mantiene información relevante
- Gates controlan qué información recordar/olvidar

4. Softmax para 3 clases:

- Output: 3 probabilidades que suman 1.0
- La clase con mayor probabilidad es la predicción

⚠️ Consideraciones Técnicas

Specific para Windows:

- El workflow está optimizado para Windows
- En otros sistemas operativos:
    1. Ajustar Conda Environment Propagation node
    2. Verificar encoding en Keras Embedding Layer node (depende del encoding nativo del sistema)

Dependencias requeridas:

- Entorno Conda con TensorFlow/Keras
- Bibliotecas de NLP para tokenización
- KNIME Deep Learning Extension

Flujo KNIME: Autoencoder para Detección de Fraude

📊 Visión General del Workflow

Este workflow implementa un **Autoencoder** para detección de fraude en transacciones con tarjetas de crédito, dividido en **5 secciones principales**:

1. **Keras Autoencoder Architecture** (Arquitectura del autoencoder)
2. **Data Preprocessing** (Preprocesamiento de datos)
3. **Training the Autoencoder** (Entrenamiento)
4. **Optimizing threshold K** (Optimización del umbral)
5. **Final Performance** (Evaluación final)

🎯 ¿Qué es un Autoencoder para Detección de Fraude?

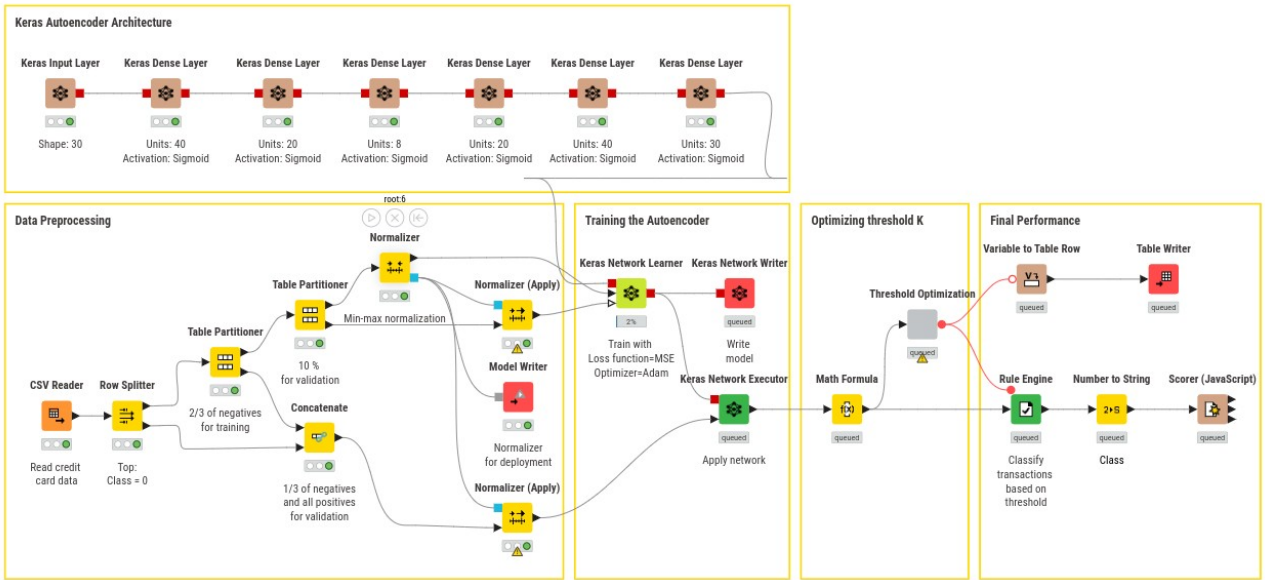
Concepto Clave:

Un autoencoder es una red neuronal que aprende a **reconstruir transacciones normales**. Cuando encuentra una transacción fraudulenta (anómala), no puede reconstruirla bien, generando un **error de reconstrucción alto**.

Estrategia de Detección:

Transacción Normal → Autoencoder → Reconstrucción Buena → Error Bajo → NO FRAUDE

Transacción Fraudulenta → Autoencoder → Reconstrucción Mala → Error Alto → FRAUDE



🔷 SECCIÓN 1: Keras Autoencoder Architecture

Arquitectura simétrica del Autoencoder:

Input(30) → Dense(40) → Dense(20) → Dense(8) → Dense(20) → Dense(40) → Dense(30)

←--------- ENCODER ---------→ ←-------- DECODER --------→

1. Keras Input Layer

Parámetros:

- **Shape: 30**

Función:

- Recibe vectores de 30 características de transacciones
- Cada transacción tiene 30 variables (monto, tiempo, ubicación, etc.)

ENCODER (Compresión):

2. Keras Dense Layer #1

Parámetros:

- **Units: 40**
- **Activation: Sigmoid**

Función:

- Primera capa de compresión
- Expande dimensionalidad temporalmente (30 → 40)

3. Keras Dense Layer #2

Parámetros:

- **Units: 20**
- **Activation: Sigmoid**

Función:

- Comprime información (40 → 20)
- Comienza a extraer características esenciales

4. Keras Dense Layer #3 (Bottleneck/Cuello de botella)

Parámetros:

- **Units: 8**
- **Activation: Sigmoid**

Función:

- **Capa de código/representación latente**

- Comprime las 30 características originales en solo **8 dimensiones**
- Fuerza a la red a aprender las características MÁS importantes
- Esta es la representación comprimida de la transacción

DECODER (Reconstrucción):

5. Keras Dense Layer #4

Parámetros:

- **Units: 20**
- **Activation: Sigmoid**

Función:

- Comienza la reconstrucción (8 → 20)
- Espejo de la capa del encoder

6. Keras Dense Layer #5

Parámetros:

- **Units: 40**
- **Activation: Sigmoid**

Función:

- Continúa expandiendo (20 → 40)
- Recupera información comprimida

7. Keras Dense Layer #6 (Output)

Parámetros:

- **Units: 30**
- **Activation: Sigmoid**

Función:

- **Capa de salida**: Reconstruye las 30 características originales
- La red intenta reproducir la entrada exacta
- Sigmoid asegura valores en rango [0, 1] después de normalización

🔷 SECCIÓN 2: Data Preprocessing

Flujo de preprocesamiento:

CSV Reader → Row Splitter → Table Partitioner → Normalizer →
Concatenate → Model Writer → Normalizer (Apply) × 2

Estrategia de Datos:

Este es el aspecto MÁS CRÍTICO del autoencoder para detección de fraude:

El autoencoder se entrena SOLO con transacciones NORMALES (no fraudulentas)

1. CSV Reader

Parámetros:

- **Read credit card data**

Función:

- Carga el dataset de transacciones con tarjeta de crédito
- Contiene transacciones normales y fraudulentas etiquetadas

2. Row Splitter

Parámetros:

- **Top: Class = 0** (transacciones normales)

Función:

- **Separa transacciones normales (Class = 0)**
- Las fraudulentas (Class = 1) se descartan para el entrenamiento
- Solo queremos que la red aprenda cómo son las transacciones NORMALES

3. Table Partitioner

Parámetros:

- **2/3 of negatives for training**

Función:

- Divide las transacciones normales:
    - 2/3 para entrenamiento
    - 1/3 para validación

4. Table Partitioner (segundo)

Parámetros:

- **10% for validation**

Función:

- Del set de validación, separa 10% adicional
- Usado para ajustar el umbral de detección

5. Normalizer

Parámetros:

- **Min-max normalization**

Función:

- Normaliza las características al rango [0, 1]
- Se ajusta SOLO con transacciones normales de entrenamiento

- Crítico para que el autoencoder funcione correctamente

6. Concatenate

Parámetros:

- **1/3 of negatives and all positives for validation**

Función:

- Combina:
  - 1/3 de transacciones normales (no usadas en training)
  - TODAS las transacciones fraudulentas
- Este set mixto se usa para validación y encontrar el umbral

7. Model Writer

Función:

- Prepara la arquitectura del autoencoder para el entrenamiento

8. Normalizer (Apply) - Para entrenamiento

Parámetros:

- **Normalizer for deployment**

Función:

- Aplica normalización a datos de entrenamiento usando parámetros aprendidos

9. Normalizer (Apply) - Para validación

Función:

- Aplica la MISMA normalización al set de validación
- Usa los mismos parámetros del training (evita data leakage)

🔷 SECCIÓN 3: Training the Autoencoder

Flujo de entrenamiento:

Keras Network Learner → Keras Network Writer → Keras Network Executor

1. Keras Network Learner

Parámetros:

- **Train with Loss function: MSE**
- **Optimizer: Adam**

Función:

- Entrena el autoencoder SOLO con transacciones normales
- Loss MSE (Mean Squared Error): mide error de reconstrucción
- Objetivo: minimizar diferencia entre entrada y salida
- La red aprende a reconstruir transacciones normales perfectamente

2. Keras Network Writer

Parámetros:

- **Write model**

Función:

- Guarda el modelo entrenado
- Permite reutilizar el autoencoder sin reentrenar

3. Keras Network Executor

Parámetros:

- **Apply network**

Función:

- Aplica el autoencoder al set de validación (normales + fraudes)
- Genera reconstrucciones para TODAS las transacciones
- Las transacciones fraudulentas tendrán mayor error de reconstrucción

🔷 SECCIÓN 4: Optimizing Threshold K

Flujo de optimización del umbral:

Math Formula → Threshold Optimization → Rule Engine →

Number to String → Scorer (JavaScript)

Concepto: ¿Qué es el threshold K?

El **threshold K** es el umbral de error de reconstrucción que separa transacciones normales de fraudulentas:

Error de reconstrucción < K → Transacción NORMAL

Error de reconstrucción ≥ K → Transacción FRAUDULENTA

1. Math Formula

Función:

- Calcula el **error de reconstrucción** para cada transacción
- Fórmula típica: `MSE = mean((original - reconstruida)²)`
- Cada transacción obtiene un score de error

2. Threshold Optimization

Función:

- Prueba diferentes valores de K en el set de validación
- Busca el K óptimo que maximiza:
    - **Detección de fraudes** (alta recall para fraudes)

    - **Minimiza falsos positivos** (baja clasificación errónea de normales)
- Balancea precision y recall

Ejemplo de optimización:

K = 0.01 → Detecta 60% fraudes, 2% falsos positivos

K = 0.05 → Detecta 85% fraudes, 5% falsos positivos ✓ MEJOR

K = 0.10 → Detecta 95% fraudes, 15% falsos positivos

3. Rule Engine

Parámetros:

- **Classify transactions based on threshold**

Función:

- Aplica el umbral K optimizado
- Regla: `IF error >= K THEN "Fraud" ELSE "Normal"`
- Genera predicciones binarias

4. Number to String

Parámetros:

- **Class**

Función:

- Convierte las predicciones numéricas (0/1) a texto ("Normal"/"Fraud")
- Prepara datos para el scorer

5. Scorer (JavaScript)

Función:

- Evalúa el rendimiento del modelo:
    - **Accuracy**: Porcentaje de clasificaciones correctas
    - **Precision**: De las predichas como fraude, cuántas lo son realmente
    - **Recall**: De los fraudes reales, cuántos detectamos
    - **F1-Score**: Media armónica de precision y recall
    - **Confusion Matrix**: Distribución de predicciones

🔷 SECCIÓN 5: Final Performance

Flujo de evaluación final:

Variable to Table Row → Table Writer → [Métricas guardadas]

1. Variable to Table Row

Función:

- Convierte el mejor threshold K en una fila de tabla

- Documenta el valor óptimo encontrado

2. Table Writer

Función:

- Guarda los resultados finales del modelo
- Incluye threshold óptimo y métricas de rendimiento
- Permite reproducibilidad y documentación

🎯 ¿Por Qué Funciona el Autoencoder para Detección de Fraude?

Principio Fundamental:

1. **Entrenamiento con normalidad:**
   - El autoencoder aprende la "firma" de transacciones normales
   - Se vuelve experto en reconstruir patrones normales
2. **Detección de anomalías:**
   - Transacciones fraudulentas tienen patrones diferentes
   - El autoencoder NO aprendió esos patrones
   - No puede reconstruirlas bien → Error alto
3. **Threshold como decisor:**
   - Errores bajos = Comportamiento normal
   - Errores altos = Comportamiento anómalo (fraude)

📊 Comparación: Autoencoder vs Clasificación Supervisada

Aspecto	Clasificación Supervisada	Autoencoder (este workflow)
Datos de entrenamiento	Necesita fraudes etiquetados	Solo necesita transacciones normales
Desbalance de clases	Problema crítico (pocos fraudes)	No es problema
Nuevos tipos de fraude	No detecta si no los vio	Detecta cualquier anomalía
Interpretabilidad	Alta (features importantes)	Media (error de reconstrucción)
Objetivo	Aprender "qué es fraude"	Aprender "qué es normal"

💡 Conceptos Clave

1. Arquitectura Simétrica:

30 → 40 → 20 → [8] → 20 → 40 → 30

- **Encoder**: Comprime información (30 → 8)
- **Bottleneck**: Representación comprimida (8 dimensiones)
- **Decoder**: Reconstruye (8 → 30)

2. Bottleneck de 8 dimensiones:

- Fuerza a la red a comprimir 30 características en solo 8
- Solo las características MÁS esenciales sobreviven
- Si la transacción es anómala, la reconstrucción falla

3. Sigmoid en todas las capas:

- Mantiene valores en rango [0, 1]
- Compatible con normalización min-max
- Suaviza las activaciones

4. Entrenamiento solo con Class = 0:

- **Clase 0**: Transacciones normales (la mayoría)
- **Clase 1**: Fraudes (ignorados durante entrenamiento)
- El modelo nunca "ve" fraudes durante el aprendizaje

5. Error de reconstrucción como score de anomalía:

$$\text{Error} = \Sigma(\text{original} - \text{reconstruida})^2 / n\_features$$

- Bajo error → Normal
- Alto error → Anómalo (posible fraude)

🔬 Flujo Completo de Detección

Fase 1: Entrenamiento

1. Cargar transacciones con tarjeta de crédito
2. Filtrar SOLO transacciones normales (Class = 0)
3. Normalizar características
4. Entrenar autoencoder para reconstruir transacciones normales
5. Guardar modelo

Fase 2: Optimización

1. Aplicar autoencoder a set de validación (normales + fraudes)
2. Calcular error de reconstrucción para cada transacción
3. Probar diferentes thresholds K
4. Seleccionar K que maximiza detección con mínimos falsos positivos

Fase 3: Producción

1. Nueva transacción llega
2. Normalizar usando parámetros del training
3. Pasar por autoencoder
4. Calcular error de reconstrucción
5. Comparar con threshold K
6. **Decisión: Fraude o Normal**

⚠️ Ventajas del Autoencoder para Fraude

✅ **Aprende de la normalidad**: No necesita muchos ejemplos de fraude
✅ **Detecta nuevos fraudes**: Cualquier patrón anómalo se detecta
✅ **Robusto al desbalance**: El 99% de transacciones son normales
✅ **Adaptable**: Se puede reentrenar con nuevos patrones normales
✅ **Unsupervised/Semi-supervised**: No requiere etiquetado exhaustivo

📈 Métricas Esperadas

Con este autoencoder en datasets de fraude típicos:

- **Accuracy**: 95-98% (la mayoría de transacciones son normales)
- **Recall de fraude**: 75-85% (detecta la mayoría de fraudes)
- **Precision de fraude**: 60-75% (algunos falsos positivos)
- **F1-Score**: 0.65-0.80

Trade-off crítico:

- Threshold bajo → Detecta más fraudes pero más falsos positivos
- Threshold alto → Menos falsos positivos pero se escapan fraudes

El threshold optimization busca el balance óptimo para el negocio.

📚 Fuentes y Recursos

Artículos Científicos:

Machine Learning para Marketing en KNIME Hub

- Villarroel Ordenes, F., & Silipo, R. (2021). Machine learning for marketing on the KNIME Hub: The development of a live repository for marketing applications. *Journal of Business Research*, 137(1), 393-410.
- DOI: 10.1016/j.jbusres.2021.08.036
- KNIME Hub Space: Machine Learning and Marketing

Libros y Guías:

Codeless Deep Learning with KNIME

- Autoras: Kathrin Melcher & Rosaria Silipo
- Editorial: Packt Publishing (Community Edition)
- Descripción: Guía completa para construir, entrenar y desplegar arquitecturas de redes neuronales profundas usando KNIME Analytics Platform
- KNIME Hub Space: Codeless Deep Learning with KNIME

Recursos Adicionales en KNIME Hub:

1. **Space de Machine Learning y Marketing**
    - Repositorio en vivo con aplicaciones prácticas de ML para marketing
    - Workflows y ejemplos reproducibles

- URL: https://hub.knime.com/knime/spaces/Machine%20Learning%20and%20Marketing/~JyadcetnSt5U1vcw/

2. **Space de Deep Learning sin Código**
    - Ejemplos prácticos de CNNs, LSTMs, Autoencoders y más
    - Workflows listos para usar del libro "Codeless Deep Learning with KNIME"
    - URL: https://hub.knime.com/kathrin/spaces/Codeless%20Deep%20Learning%20with%20KNIME/~yMp8GBkT0Xwzx5X2/

Plataforma Principal:

KNIME Analytics Platform

- Plataforma open-source para ciencia de datos y machine learning
- Website oficial: www.knime.com
- Documentación: docs.knime.com