

Modelo 4: Time-Period Model (Agregación Temporal)

Descripción del Modelo

Este modelo agrega datos por **períodos de tiempo** (día, semana, mes, trimestre, año). Calcula tendencias, estacionalidad, pronósticos y comparaciones período sobre período. Corresponde a la etapa **ATRIBUTOS** del workflow con GROUP BY fecha/mes/año.

Input

Output del Modelo 1 (Transaction Enriched)

Aggregation Keys

- Daily: GROUP BY fecha
- Weekly: GROUP BY año, semana_año
- Monthly: GROUP BY año, mes
- Quarterly: GROUP BY año, trimestre
- Yearly: GROUP BY año

Features Soportados

- **A6** - Pronóstico de Ventas Básico
- **A7** - Análisis de Tendencias Temporales
- **B2** - Análisis de Estacionalidad
- **B7** - Comparación con Períodos Anteriores
- **C3** - Alertas Predictivas (base temporal)
- **C8** - Pronósticos Multi-Factor (base temporal)

Cálculos del Modelo

1. Agregación Diaria

1.1. Métricas Básicas por Día

```
SELECT
    fecha,
    COUNT(trans_id) as num_transacciones,
    SUM(cantidad) as unidades_vendidas,
    SUM(total) as ingresos_dia,
    SUM(costo) as costo_dia,
    SUM(margen_absoluto) as margen_dia,
    AVG(total) as ticket_promedio_dia,
    COUNT(DISTINCT customer_id) as clientes_unicos_dia,
```

```

COUNT(DISTINCT producto) as productos_vendidos_dia
FROM transactions_enriched
GROUP BY fecha
ORDER BY fecha

```

Python:

```

def calculate_daily_metrics(df):
    """Calcula métricas agregadas por día"""
    daily = df.groupby('fecha').agg({
        'trans_id': 'count',
        'cantidad': 'sum',
        'total': ['sum', 'mean'],
        'costo': 'sum',
        'margen_absoluto': 'sum',
        'customer_id_clean': 'nunique',
        'producto_clean': 'nunique'
    }).reset_index()

    daily.columns = [
        'fecha', 'num_transacciones', 'unidades_vendidas',
        'ingresos_dia', 'ticket_promedio_dia', 'costo_dia',
        'margen_dia', 'clientes_unicos_dia', 'productos_vendidos_dia'
    ]

    # Componentes temporales
    daily['año'] = daily['fecha'].dt.year
    daily['mes'] = daily['fecha'].dt.month
    daily['dia_semana'] = daily['fecha'].dt.dayofweek + 1 # 1=Lunes
    daily['semana_año'] = daily['fecha'].dt.isocalendar().week
    daily['trimestre'] = daily['fecha'].dt.quarter

    # Día juliano
    daily['dia_juliano'] = (daily['fecha'] - daily['fecha'].min()).dt.days

    return daily

daily_metrics = calculate_daily_metrics(df_enriched)

```

2. Agregación Mensual

2.1. Métricas Básicas por Mes

```

SELECT
    año,
    mes,
    COUNT(trans_id) as num_transacciones,
    SUM(total) as ingresos_mes,
    SUM(costo) as costo_mes,
    SUM(margen_absoluto) as margen_mes,
    AVG(total) as ticket_promedio_mes,
    COUNT(DISTINCT customer_id) as clientes_unicos_mes,
    COUNT(DISTINCT producto) as productos_vendidos_mes,
    COUNT(DISTINCT fecha) as dias_activos
FROM transactions_enriched
GROUP BY año, mes
ORDER BY año, mes

```

Python:

```

def calculate_monthly_metrics(df):
    """Calcula métricas agregadas por mes"""
    monthly = df.groupby(['año', 'mes']).agg({
        'trans_id': 'count',
        'total': ['sum', 'mean'],
        'costo': 'sum',
        'margen_absoluto': 'sum',
        'customer_id_clean': 'nunique',
        'producto_clean': 'nunique',
        'fecha': 'nunique'
    }).reset_index()

    monthly.columns = [
        'año', 'mes', 'num_transacciones', 'ingresos_mes',
        'ticket_promedio_mes', 'costo_mes', 'margen_mes',
        'clientes_unicos_mes', 'productos_vendidos_mes', 'dias_activos'
    ]

    # Crear fecha para ordenamiento
    monthly['fecha_mes'] = pd.to_datetime(
        monthly[['año', 'mes']].assign(dia=1)
    )

    # Mes relativo (para análisis)
    base_año = monthly['año'].min()
    base_mes = monthly.loc[monthly['año'] == base_año, 'mes'].min()
    monthly['mes_relativo'] = (monthly['año'] - base_año) * 12 + (monthly['mes']

```

```
        return monthly.sort_values('fecha_mes')

monthly_metrics = calculate_monthly_metrics(df_enriched)
```

3. Análisis de Tendencias (Feature A7)

3.1. Comparaciones Período sobre Período

```
-- Mes vs mes anterior
WITH monthly_data AS (
    SELECT año, mes, SUM(total) as ingresos_mes
    FROM transactions_enriched
    GROUP BY año, mes
)
SELECT
    año,
    mes,
    ingresos_mes,
    LAG(ingresos_mes) OVER (ORDER BY año, mes) as ingresos_mes_anterior,
    ingresos_mes - LAG(ingresos_mes) OVER (ORDER BY año, mes) as cambio_absoluto,
    ((ingresos_mes - LAG(ingresos_mes) OVER (ORDER BY año, mes)) /
     LAG(ingresos_mes) OVER (ORDER BY año, mes) * 100) as cambio_porcentual
FROM monthly_data
ORDER BY año, mes
```

Python:

```
def calculate_period_comparisons(monthly_df):
    """Calcula comparaciones período sobre período"""
    df = monthly_df.copy()

    # Mes vs mes anterior
    df['ingresos_mes_anterior'] = df['ingresos_mes'].shift(1)
    df['cambio_mom_absoluto'] = df['ingresos_mes'] - df['ingresos_mes_anterior']
    df['cambio_mom_porcentual'] = (
        df['cambio_mom_absoluto'] / df['ingresos_mes_anterior'] * 100
    )

    # Mes vs mismo mes año anterior (YoY)
    df['ingresos_yoy'] = df['ingresos_mes'].shift(12)
    df['cambio_yoy_absoluto'] = df['ingresos_mes'] - df['ingresos_yoy']
    df['cambio_yoy_porcentual'] = (
        df['cambio_yoy_absoluto'] / df['ingresos_yoy'] * 100
    )
```

```

# Dirección de tendencia
df['tendencia'] = np.select(
    [
        df['cambio_mom_porcentual'] > 5,
        df['cambio_mom_porcentual'] < -5
    ],
    ['creciente', 'decreciente'],
    default='estable'
)

return df

monthly_metrics = calculate_period_comparisons(monthly_metrics)

```

3.2. Promedios Móviles

```

-- Promedio móvil 3 meses
AVG(ingresos_mes) OVER (
    ORDER BY año, mes
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
) as promedio_movil_3m

-- Promedio móvil 6 meses
AVG(ingresos_mes) OVER (
    ORDER BY año, mes
    ROWS BETWEEN 5 PRECEDING AND CURRENT ROW
) as promedio_movil_6m

```

Python:

```

def calculate_moving_averages(monthly_df):
    """Calcula promedios móviles"""
    df = monthly_df.copy()

    # Promedio móvil 3 meses
    df['promedio_movil_3m'] = df['ingresos_mes'].rolling(window=3, min_periods=1)

    # Promedio móvil 6 meses
    df['promedio_movil_6m'] = df['ingresos_mes'].rolling(window=6, min_periods=1)

    # Desviación estándar móvil (para bandas)
    df['std_movil_3m'] = df['ingresos_mes'].rolling(window=3, min_periods=1).std()

    return df

```

```
monthly_metrics = calculate_moving_averages(monthly_metrics)
```

3.3. Tendencia Lineal

```
-- Usando regresión lineal sobre mes_relativo  
-- En SQL complejo, mejor hacerlo en Python
```

Python:

```
from sklearn.linear_model import LinearRegression  
  
def calculate_trend(monthly_df):  
    """Calcula tendencia lineal"""  
    df = monthly_df.copy()  
  
    # Preparar datos para regresión  
    X = df['mes_relativo'].values.reshape(-1, 1)  
    y = df['ingresos_mes'].values  
  
    # Filtrar NaNs  
    mask = ~np.isnan(y)  
    X_clean = X[mask]  
    y_clean = y[mask]  
  
    # Regresión lineal  
    model = LinearRegression()  
    model.fit(X_clean, y_clean)  
  
    # Predicción de tendencia  
    df['tendencia_lineal'] = model.predict(X)  
    df['pendiente_tendencia'] = model.coef_[0]  
  
    return df  
  
monthly_metrics = calculate_trend(monthly_metrics)
```

4. Análisis de Estacionalidad (Feature B2)

4.1. Índices Estacionales por Mes

```
-- Índice estacional = Promedio del mes / Promedio general  
WITH monthly_avg AS (  

```

```

SELECT mes, AVG(ingresos_mes) as avg_ingresos_mes
FROM monthly_metrics
GROUP BY mes
),
overall_avg AS (
    SELECT AVG(ingresos_mes) as avg_general
    FROM monthly_metrics
)
SELECT
    mes,
    avg_ingresos_mes,
    avg_general,
    (avg_ingresos_mes / avg_general) as indice_estacional
FROM monthly_avg CROSS JOIN overall_avg
ORDER BY mes

```

Python:

```

def calculate_seasonality_indices(monthly_df):
    """Calcula índices de estacionalidad por mes"""
    # Promedio por mes (across all years)
    monthly_avg = monthly_df.groupby('mes')['ingresos_mes'].mean().reset_index()
    monthly_avg.columns = ['mes', 'avg_ingresos_mes']

    # Promedio general
    avg_general = monthly_df['ingresos_mes'].mean()

    # Índice estacional
    monthly_avg['indice_estacional'] = monthly_avg['avg_ingresos_mes'] / avg_general

    # Categorizar
    monthly_avg['categoria_estacional'] = pd.cut(
        monthly_avg['indice_estacional'],
        bins=[0, 0.9, 1.1, np.inf],
        labels=['bajo', 'normal', 'alto']
    )

    return monthly_avg

seasonality_indices = calculate_seasonality_indices(monthly_metrics)

```

4.2. Estacionalidad por Día de Semana

```

def calculate_dow_seasonality(daily_df):
    """Calcula estacionalidad por día de semana"""

```

```

# Promedio por día de semana
dow_avg = daily_df.groupby('dia_semana')['ingresos_dia'].mean().reset_index()
dow_avg.columns = ['dia_semana', 'avg_ingresos_dia']

# Nombres de días
dow_names = {1: 'Lunes', 2: 'Martes', 3: 'Miércoles', 4: 'Jueves',
              5: 'Viernes', 6: 'Sábado', 7: 'Domingo'}
dow_avg['dia_nombre'] = dow_avg['dia_semana'].map(dow_names)

# Índice
avg_general = daily_df['ingresos_dia'].mean()
dow_avg['indice_estacional'] = dow_avg['avg_ingresos_dia'] / avg_general

return dow_avg

dow_seasonality = calculate_dow_seasonality(daily_metrics)

```

5. Pronósticos Básicos (Feature A6)

5.1. Pronóstico por Promedio Móvil

```

def forecast_moving_average(monthly_df, periods=1):
    """Pronóstico simple usando promedio móvil"""
    last_3m_avg = monthly_df['ingresos_mes'].tail(3).mean()

    forecast = {
        'metodo': 'promedio_movil_3m',
        'pronostico': last_3m_avg,
        'periodo_siguiete': monthly_df['mes_relativo'].max() + 1
    }

    return forecast

forecast_ma = forecast_moving_average(monthly_metrics)

```

5.2. Pronóstico por Tendencia + Estacionalidad

```

def forecast_trend_seasonal(monthly_df, seasonality_indices, periods=1):
    """Pronóstico usando tendencia lineal + estacionalidad"""
    from sklearn.linear_model import LinearRegression

    # Tendencia
    X = monthly_df['mes_relativo'].values.reshape(-1, 1)
    y = monthly_df['ingresos_mes'].values

```



```

model = LinearRegression()
model.fit(X, y)

# Predecir siguiente período
next_period = monthly_df['mes_relativo'].max() + 1
next_month = (monthly_df['mes'].iloc[-1] % 12) + 1 # Próximo mes

# Tendencia para próximo período
trend_forecast = model.predict([[next_period]])[0]

# Ajustar por estacionalidad
seasonal_factor = seasonality_indices.loc[
    seasonality_indices['mes'] == next_month,
    'indice_estacional'
].values[0]

forecast = trend_forecast * seasonal_factor

return {
    'metodo': 'tendencia_estacional',
    'tendencia': trend_forecast,
    'factor_estacional': seasonal_factor,
    'pronostico': forecast,
    'mes_pronostico': next_month
}

forecast_ts = forecast_trend_seasonal(monthly_metrics, seasonality_indices)

```

6. Agregaciones por Otras Dimensiones Temporales

6.1. Agregación Semanal

```

def calculate_weekly_metrics(df):
    """Calcula métricas agregadas por semana"""
    weekly = df.groupby(['año', 'semana_año']).agg({
        'trans_id': 'count',
        'total': ['sum', 'mean'],
        'customer_id_clean': 'nunique',
        'fecha': ['min', 'max']
    }).reset_index()

    weekly.columns = [
        'año', 'semana', 'num_transacciones', 'ingresos_semana',
        'ticket_promedio', 'clientes_unicos', 'fecha_inicio', 'fecha_fin'
    ]

```

```

    ]

    return weekly

weekly_metrics = calculate_weekly_metrics(df_enriched)

```

6.2. Agregación Trimestral

```

def calculate_quarterly_metrics(df):
    """Calcula métricas agregadas por trimestre"""
    quarterly = df.groupby(['año', 'trimestre']).agg({
        'trans_id': 'count',
        'total': ['sum', 'mean'],
        'margen_absoluto': 'sum',
        'customer_id_clean': 'nunique'
    }).reset_index()

    quarterly.columns = [
        'año', 'trimestre', 'num_transacciones', 'ingresos_trimestre',
        'ticket_promedio', 'margen_trimestre', 'clientes_unicos'
    ]

    # Etiqueta trimestre
    quarterly['trimestre_label'] = 'Q' + quarterly['trimestre'].astype(str) + ' '

    return quarterly

quarterly_metrics = calculate_quarterly_metrics(df_enriched)

```

7. Detección de Anomalías Temporales

7.1. Z-Score por Día

```

def detect_daily_anomalies(daily_df, threshold=2):
    """Detecta anomalías en ventas diarias"""
    df = daily_df.copy()

    # Calcular Z-score
    df['ingresos_mean'] = df['ingresos_dia'].mean()
    df['ingresos_std'] = df['ingresos_dia'].std()
    df['z_score'] = (df['ingresos_dia'] - df['ingresos_mean']) / df['ingresos_std']

    # Detectar anomalías
    df['es_anomalia'] = (abs(df['z_score']) > threshold).astype(int)

```

```

df['tipo_anomalia'] = np.select(
    [df['z_score'] > threshold, df['z_score'] < -threshold],
    ['spike_positivo', 'spike_negativo'],
    default='normal'
)

return df

daily_metrics = detect_daily_anomalies(daily_metrics)

```

Output Schema

Daily Metrics

```

{
    'fecha': date,
    'año': int,
    'mes': int,
    'dia_semana': int,
    'semana_año': int,
    'trimestre': int,
    'dia_juliano': int,

    'num_transacciones': int,
    'unidades_vendidas': int,
    'ingresos_dia': float,
    'costo_dia': float,
    'margen_dia': float,
    'ticket_promedio_dia': float,
    'clientes_unicos_dia': int,
    'productos_vendidos_dia': int,

    # Anomalías
    'ingresos_mean': float,
    'ingresos_std': float,
    'z_score': float,
    'es_anomalia': int,
    'tipo_anomalia': str
}

```

Monthly Metrics

```

{
    'año': int,

```

```

'mes': int,
'fecha_mes': date,
'mes_relativo': int,

'num_transacciones': int,
'ingresos_mes': float,
'costo_mes': float,
'margen_mes': float,
'ticket_promedio_mes': float,
'clientes_unicos_mes': int,
'productos_vendidos_mes': int,
'dias_activos': int,

# Comparaciones
'ingresos_mes_anterior': float,
'cambio_mom_absoluto': float,
'cambio_mom_porcentual': float,
'ingresos_yoy': float,
'cambio_yoy_absoluto': float,
'cambio_yoy_porcentual': float,
'tendencia': str, # creciente/decreciente/estable

# Promedios móviles
'promedio_movil_3m': float,
'promedio_movil_6m': float,
'std_movil_3m': float,

# Tendencia
'tendencia_lineal': float,
'pendiente_tendencia': float
}

```

Seasonality Indices

```

{
    'mes': int, # 1-12
    'avg_ingresos_mes': float,
    'indice_estacional': float,
    'categoria_estacional': str # bajo/normal/alto
}

```

Implementación Completa

```

def calculate_time_period_models(df_enriched):
    """
    Calcula todos los modelos temporales

    Returns:
    -----
    dict con daily, weekly, monthly, quarterly, seasonality
    """

    # 1. Daily
    daily = calculate_daily_metrics(df_enriched)
    daily = detect_daily_anomalies(daily)

    # 2. Weekly
    weekly = calculate_weekly_metrics(df_enriched)

    # 3. Monthly
    monthly = calculate_monthly_metrics(df_enriched)
    monthly = calculate_period_comparisons(monthly)
    monthly = calculate_moving_averages(monthly)
    monthly = calculate_trend(monthly)

    # 4. Quarterly
    quarterly = calculate_quarterly_metrics(df_enriched)

    # 5. Seasonality
    seasonality_monthly = calculate_seasonality_indices(monthly)
    seasonality_dow = calculate_dow_seasonality(daily)

    # 6. Forecasts
    forecast_ma = forecast_moving_average(monthly)
    forecast_ts = forecast_trend_seasonal(monthly, seasonality_monthly)

    return {
        'daily': daily,
        'weekly': weekly,
        'monthly': monthly,
        'quarterly': quarterly,
        'seasonality_monthly': seasonality_monthly,
        'seasonality_dow': seasonality_dow,
        'forecasts': {
            'moving_average': forecast_ma,
            'trend_seasonal': forecast_ts
        }
    }

```

```
# Uso
time_models = calculate_time_period_models(df_enriched)
```

Visualizaciones Sugeridas

```
import matplotlib.pyplot as plt

def plot_trend_analysis(monthly_df):
    """Gráfico de tendencia con promedios móviles"""
    fig, ax = plt.subplots(figsize=(12, 6))

    ax.plot(monthly_df['fecha_mes'], monthly_df['ingresos_mes'],
            label='Ingresos Mensuales', marker='o')
    ax.plot(monthly_df['fecha_mes'], monthly_df['promedio_movil_3m'],
            label='Promedio Móvil 3M', linestyle='--')
    ax.plot(monthly_df['fecha_mes'], monthly_df['tendencia_lineal'],
            label='Tendencia', linestyle=':')

    ax.set_xlabel('Fecha')
    ax.set_ylabel('Ingresos')
    ax.set_title('Análisis de Tendencia de Ventas')
    ax.legend()
    ax.grid(True, alpha=0.3)

    return fig


def plot_seasonality(seasonality_df):
    """Gráfico de índices estacionales"""
    fig, ax = plt.subplots(figsize=(10, 6))

    ax.bar(seasonality_df['mes'], seasonality_df['indice_estacional'])
    ax.axhline(y=1.0, color='r', linestyle='--', label='Promedio')
    ax.set_xlabel('Mes')
    ax.set_ylabel('Índice Estacional')
    ax.set_title('Estacionalidad por Mes')
    ax.set_xticks(range(1, 13))
    ax.legend()

    return fig
```

Dependencies

Input Requirements:

-  Output de Modelo 1 (Transaction Enriched)

External Libraries:

- pandas
 - numpy
 - scikit-learn (LinearRegression)
 - matplotlib / plotly (visualizaciones)
-

Notas

- Modelos diarios/semanales útiles para monitoreo operacional
- Modelos mensuales/trimestrales útiles para reportes ejecutivos
- Estacionalidad requiere mínimo 12 meses de datos para ser confiable
- Pronósticos mejorar con datos de 24+ meses