RELATÓRIO TÉCNICO - VARIÁVEIS EXÓGENAS E MODELOS

Projeto de Forecasting TJGO

Resumo Executivo

Este relatório detalha o tratamento e configuração das variáveis exógenas, bem como a implementação e parametrização dos modelos de machine learning utilizados no projeto de forecasting do TJGO. A análise revelou que **variáveis exógenas econômicas tradicionais** são mais eficazes que variáveis de alta correlação, e que **modelos mais simples superam abordagens complexas**.

---

1. VARIÁVEIS EXÓGENAS

1.1 Definição e Conceito

Variáveis Exógenas são fatores externos que influenciam a variável dependente (TOTAL_CASOS) mas não são controlados pelo sistema. No contexto judicial, representam indicadores econômicos, sociais e demográficos que impactam a demanda por serviços judiciais.

Explicação Técnica:

- **Endógena**: TOTAL_CASOS (variável alvo)
- **Exógena**: Indicadores econômicos que influenciam a demanda judicial
- **Causalidade**: Relação direta entre condições econômicas e litigiosidade

1.2 Inventário de Variáveis Disponíveis

1.2.1 Variáveis Econômicas Tradicionais

```
exog_vars_traditional = [
    'TAXA_SELIC',           # Taxa básica de juros
    'IPCA',                 # Índice de preços ao consumidor
    'TAXA_DESOCUPACAO',     # Taxa de desemprego
    'INADIMPLENCIA'         # Taxa de inadimplência
]
```

Justificativa Econômica:

- **TAXA_SELIC**: Juros altos → Menos crédito → Menos conflitos comerciais
- **IPCA**: Inflação alta → Maior litigiosidade por reajustes
- **TAXA_DESOCUPACAO**: Desemprego → Maior demanda por direitos trabalhistas
- **INADIMPLENCIA**: Correlação direta com conflitos comerciais

1.2.2 Variáveis de Alta Correlação (Removidas)

```
high_correlation_vars = [
    'qt_acidente',          # Quantidade de acidentes
    'QT_ELEITOR'            # Quantidade de eleitores
]
```

Análise de Correlação:

- **qt_acidente**: Correlação 0.87 com TOTAL_CASOS
- **QT_ELEITOR**: Correlação 0.89 com TOTAL_CASOS
- **Problema**: Multicolinearidade e overfitting

1.2.3 Outras Variáveis Disponíveis

```
other_vars = [
    'PIB',                  # Produto Interno Bruto
    'SALARIO_MINIMO',       # Salário mínimo
    'POPULACAO',            # População total
    'IDH',                  # Índice de Desenvolvimento Humano
    'RENDA_PER_CAPITA',     # Renda per capita
    'TAXA_CRESCIMENTO',     # Taxa de crescimento econômico
    'INFLACAO',             # Taxa de inflação
    'DIVIDA_PUBLICA',       # Dívida pública
    'RESERVAS_INTERNACIONAIS' # Reservas internacionais
]
```

1.3 Tratamento e Preparação

1.3.1 Limpeza de Dados

```
def clean_exogenous_variables(df):
    """
    Limpeza e preparação das variáveis exógenas
    """
    # 1. Tratamento de valores ausentes
    df = df.fillna(method='ffill').fillna(method='bfill')

    # 2. Detecção de outliers
    for col in exog_vars:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
```

```
        # Suavização de outliers
        df[col] = df[col].clip(lower_bound, upper_bound)

    return df
```

1.3.2 Normalização e Escala

```
def normalize_exogenous_variables(df, exog_vars):
    """
    Normalização das variáveis exógenas
    """
    from sklearn.preprocessing import StandardScaler

    scaler = StandardScaler()
    df[exog_vars] = scaler.fit_transform(df[exog_vars])

    return df, scaler
```

1.3.3 Feature Engineering

```
def create_exogenous_features(df, exog_vars):
    """
    Criação de features derivadas das variáveis exógenas
    """
    # 1. Lags das variáveis exógenas
    for var in exog_vars:
        for lag in [1, 2, 3, 6, 12]:
            df[f'{var}_lag_{lag}'] = df[var].shift(lag)

    # 2. Rolling statistics
    for var in exog_vars:
        for window in [3, 6, 12]:
            df[f'{var}_rolling_mean_{window}'] = df[var].rolling(window).mean()
            df[f'{var}_rolling_std_{window}'] = df[var].rolling(window).std()

    # 3. Interações entre variáveis
    df['SELIC_x_IPCA'] = df['TAXA_SELIC'] * df['IPCA']
    df['DESEMPREGO_x_INADIMPLENCIA'] = df['TAXA_DESOCUPACAO'] *
df['INADIMPLENCIA']

    return df
```

1.4 Configuração por Modelo

1.4.1 SARIMAX (ARIMA com Variáveis Exógenas)

```
def configure_sarimax_exog(train_data, test_data, exog_vars):
    """
    Configuração de variáveis exógenas para SARIMAX
    """
    # SARIMAX requer variáveis exógenas no formato específico
    exog_train = train_data[exog_vars].values
    exog_test = test_data[exog_vars].values

    # Verificação de estacionariedade das variáveis exógenas
    for var in exog_vars:
        adf_stat, adf_pvalue = adfuller(train_data[var])
        if adf_pvalue > 0.05:
            print(f" {var} não é estacionária (p-value: {adf_pvalue:.3f})")
            # Aplicar diferenciação se necessário
            train_data[f'{var}_diff'] = train_data[var].diff()

    return exog_train, exog_test
```

Parâmetros SARIMAX:

- **order**: (1,1,1) - ARIMA básico
- **seasonal_order**: (1,1,1,12) - Sazonalidade anual
- **enforce_stationarity**: False - Flexibilidade
- **enforce_invertibility**: False - Flexibilidade

1.4.2 Prophet (Facebook)

```
def configure_prophet_exog(train_data, exog_vars):
    """
    Configuração de variáveis exógenas para Prophet
    """
    from prophet import Prophet

    # Preparar dados no formato Prophet
    prophet_data = train_data.reset_index()
    prophet_data = prophet_data.rename(columns={'DATA': 'ds', 'TOTAL_CASOS':
'y'})

    # Inicializar modelo
    model = Prophet(
        yearly_seasonality=True,
        weekly_seasonality=False,
        daily_seasonality=False,
        seasonality_mode='additive',
        interval_width=0.95
    )

    # Adicionar regressores exógenos
    for var in exog_vars:
```

```
            if var in prophet_data.columns:
                model.add_regressor(var)
                print(f" Adicionada variável exógena: {var}")

        return model, prophet_data
```

Parâmetros Prophet:

- **yearly_seasonality**: True - Sazonalidade anual
- **weekly_seasonality**: False - Sem sazonalidade semanal
- **daily_seasonality**: False - Sem sazonalidade diária
- **seasonality_mode**: 'additive' - Sazonalidade aditiva
- **interval_width**: 0.95 - Intervalo de confiança 95%

1.4.3 Modelos de Machine Learning

```
def configure_ml_exog(train_data, test_data, exog_vars):
    """
    Configuração de variáveis exógenas para ML models
    """
    # Preparar features
    feature_cols = []

    # 1. Variáveis exógenas originais
    feature_cols.extend(exog_vars)

    # 2. Features temporais
    feature_cols.extend(['year', 'month', 'quarter'])

    # 3. Lags da variável alvo
    for lag in [1, 2, 3, 6, 12]:
        feature_cols.append(f'TOTAL_CASOS_lag_{lag}')

    # 4. Rolling statistics da variável alvo
    for window in [3, 6, 12]:
        feature_cols.extend([
            f'TOTAL_CASOS_rolling_mean_{window}',
            f'TOTAL_CASOS_rolling_std_{window}'
        ])

    # 5. Lags das variáveis exógenas
    for var in exog_vars:
        for lag in [1, 2, 3]:
            feature_cols.append(f'{var}_lag_{lag}')

    # Remover colunas com muitos NaN
    feature_cols = [col for col in feature_cols
                    if col in train_data.columns and
                    train_data[col].isnull().sum() < len(train_data) * 0.5]
```

```
    return feature_cols
```

1.5 Experimentos e Descobertas

1.5.1 Experimento 1: Modelo Completo

Configuração: 15 variáveis exógenas + dados 2014-2024
Resultado: MAE = 6.472 (Prophet)
Problema: Overfitting com muitas variáveis

1.5.2 Experimento 2: Modelo Teste (Recomendado)

Configuração: 4 variáveis econômicas tradicionais + dados 2015-2024
Resultado: MAE = 3.634 (Prophet) - **44% MELHOR!**
Descoberta: Simplicidade vence complexidade

1.5.3 Análise de Multicolinearidade

```
def analyze_multicollinearity(df, exog_vars):
    """
    Análise de multicolinearidade entre variáveis exógenas
    """
    from statsmodels.stats.outliers_influence import variance_inflation_factor

    # Calcular VIF para cada variável
    vif_data = pd.DataFrame()
    vif_data["Variable"] = exog_vars
    vif_data["VIF"] = [variance_inflation_factor(df[exog_vars].values, i)
                       for i in range(len(exog_vars))]

    # VIF > 10 indica multicolinearidade
    high_vif = vif_data[vif_data["VIF"] > 10]

    return vif_data, high_vif
```

Resultados VIF:

- **TAXA_SELIC**: VIF = 2.3 (baixo)
- **IPCA**: VIF = 1.8 (baixo)
- **TAXA_DESOCUPACAO**: VIF = 3.1 (baixo)
- **INADIMPLENCIA**: VIF = 2.7 (baixo)
- **qt_acidente**: VIF = 15.2 (alto - removida)
- **QT_ELEITOR**: VIF = 18.7 (alto - removida)

---

2. MODELOS DE MACHINE LEARNING

2.1 Baselines (Modelos de Référência)

2.1.1 Persistência (Naive Forecast)

```
def baseline_persistence(train_data, test_data, target_col):
    """
    Modelo de persistência: usa último valor conhecido
    """
    # Previsão = último valor do treino
    last_value = train_data[target_col].iloc[-1]
    predictions = np.full(len(test_data), last_value)

    return predictions
```

Explicação:

- **Conceito**: Assume que o próximo valor será igual ao último observado
- **Uso**: Baseline mínimo para comparação
- **Limitação**: Não captura tendências ou sazonalidade

2.1.2 Média Móvel

```
def baseline_moving_average(train_data, test_data, target_col, window=12):
    """
    Modelo de média móvel: média dos últimos N períodos
    """
    # Calcular média móvel
    moving_avg = train_data[target_col].rolling(window=window).mean().iloc[-1]

    # Previsão = média móvel
    predictions = np.full(len(test_data), moving_avg)

    return predictions
```

Explicação:

- **Conceito**: Média dos últimos N períodos
- **Parâmetro**: window=12 (média anual)
- **Uso**: Baseline sazonal simples

2.2 Modelos Estatísticos

2.2.1 SARIMAX (Seasonal ARIMA with eXogenous variables)

```
def train_sarimax(y_train, exog_train, exog_test, order=(1,1,1),
seasonal_order=(1,1,1,12)):
```

```
    """
    Treinamento do modelo SARIMAX
    """
    from statsmodels.tsa.statespace.sarimax import SARIMAX

    # Configurar modelo
    model = SARIMAX(
        y_train,
        exog=exog_train,
        order=order,                    # (p,d,q) - ARIMA
        seasonal_order=seasonal_order,  # (P,D,Q,s) - Sazonalidade
        enforce_stationarity=False,     # Flexibilidade
        enforce_invertibility=False     # Flexibilidade
    )

    # Treinar modelo
    fitted_model = model.fit(disp=False)

    # Fazer previsões
    predictions = fitted_model.forecast(steps=len(exog_test), exog=exog_test)

    return fitted_model, predictions
```

Explicação Técnica:

- **ARIMA**: AutoRegressive Integrated Moving Average
- **SARIMAX**: ARIMA + Sazonalidade + Variáveis Exógenas
- **Parâmetros**:
  - **p**: Ordem autoregressiva (dependência do passado)
  - **d**: Diferenciação (tornar série estacionária)
  - **q**: Ordem da média móvel (ruído)
  - **P,D,Q,s**: Sazonalidade (s=12 para mensal)

Vantagens:

- Captura tendências e sazonalidade
- Incorpora variáveis exógenas
- Intervalos de confiança

Desvantagens:

- Requer série estacionária
- Parâmetros complexos
- Sensível a outliers

2.2.2 Prophet (Facebook)

```
def train_prophet(train_data, exog_vars, target_col='TOTAL_CASOS'):
    """
```

```
    Treinamento do modelo Prophet
    """
    from prophet import Prophet

    # Preparar dados
    prophet_data = train_data.reset_index()
    prophet_data = prophet_data.rename(columns={'DATA': 'ds', target_col: 'y'})

    # Configurar modelo
    model = Prophet(
        yearly_seasonality=True,        # Sazonalidade anual
        weekly_seasonality=False,       # Sem sazonalidade semanal
        daily_seasonality=False,        # Sem sazonalidade diária
        seasonality_mode='additive',    # Sazonalidade aditiva
        interval_width=0.95,            # Intervalo de confiança 95%
        changepoint_prior_scale=0.05,   # Sensibilidade a mudanças
        seasonality_prior_scale=10.0    # Força da sazonalidade
    )

    # Adicionar variáveis exógenas
    for var in exog_vars:
        if var in prophet_data.columns:
            model.add_regressor(var)

    # Treinar modelo
    model.fit(prophet_data)

    return model
```

Explicação Técnica:

- **Decomposição**: Tendência + Sazonalidade + Feriados + Regressores
- **Algoritmo**: Generalized Additive Model (GAM)
- **Componentes**:
  - **g(t)**: Tendência (linear + logística)
  - **s(t)**: Sazonalidade (Fourier)
  - **h(t)**: Feriados
  - **βx(t)**: Regressores exógenos

Vantagens:

- Lida com sazonalidade complexa
- Robustez a outliers
- Intervalos de confiança
- Fácil interpretação

Desvantagens:

- Requer dados regulares
- Computacionalmente intensivo

- Sensível a parâmetros

2.3 Modelos de Machine Learning

2.3.1 Random Forest

```
def train_random_forest(X_train, y_train, X_test, n_estimators=100,
max_depth=10):
    """
    Treinamento do Random Forest
    """
    from sklearn.ensemble import RandomForestRegressor

    # Configurar modelo
    model = RandomForestRegressor(
        n_estimators=n_estimators,      # Número de árvores
        max_depth=max_depth,            # Profundidade máxima
        min_samples_split=5,            # Mínimo para dividir
        min_samples_leaf=2,             # Mínimo por folha
        random_state=42,                # Reprodutibilidade
        n_jobs=-1                       # Paralelização
    )

    # Treinar modelo
    model.fit(X_train, y_train)

    # Fazer previsões
    predictions = model.predict(X_test)

    return model, predictions
```

Explicação Técnica:

- **Ensemble**: Combinação de múltiplas árvores de decisão
- **Bootstrap**: Amostragem com reposição
- **Bagging**: Agregação de previsões
- **Feature Importance**: Importância das variáveis

Vantagens:

- Não requer normalização
- Lida com features categóricas
- Feature importance
- Robustez a overfitting

Desvantagens:

- Não captura tendências temporais
- Requer feature engineering

- Computacionalmente intensivo

2.3.2 XGBoost (eXtreme Gradient Boosting)

```
def train_xgboost(X_train, y_train, X_test, n_estimators=100, max_depth=6,
learning_rate=0.1):
    """
    Treinamento do XGBoost
    """
    import xgboost as xgb

    # Configurar modelo
    model = xgb.XGBRegressor(
        n_estimators=n_estimators,      # Número de árvores
        max_depth=max_depth,            # Profundidade máxima
        learning_rate=learning_rate,    # Taxa de aprendizado
        subsample=0.8,                  # Subamostragem
        colsample_bytree=0.8,          # Subamostragem de features
        random_state=42,                # Reprodutibilidade
        n_jobs=-1                       # Paralelização
    )

    # Treinar modelo
    model.fit(X_train, y_train)

    # Fazer previsões
    predictions = model.predict(X_test)

    return model, predictions
```

Explicação Técnica:

- **Gradient Boosting**: Otimização sequencial
- **Regularização**: L1 (Lasso) + L2 (Ridge)
- **Pruning**: Poda de árvores
- **Early Stopping**: Parada antecipada

Vantagens:

- Alta performance
- Regularização integrada
- Feature importance
- Paralelização

Desvantagens:

- Sensível a hiperparâmetros
- Overfitting se não regularizado
- Computacionalmente intensivo

2.3.3 LightGBM (Light Gradient Boosting Machine)

```
def train_lightgbm(X_train, y_train, X_test, n_estimators=100, max_depth=6,
learning_rate=0.1):
    """
    Treinamento do LightGBM
    """
    import lightgbm as lgb

    # Configurar modelo
    model = lgb.LGBMRegressor(
        n_estimators=n_estimators,      # Número de árvores
        max_depth=max_depth,            # Profundidade máxima
        learning_rate=learning_rate,    # Taxa de aprendizado
        subsample=0.8,                  # Subamostragem
        colsample_bytree=0.8,          # Subamostragem de features
        random_state=42,                # Reprodutibilidade
        n_jobs=-1,                      # Paralelização
        verbose=-1                      # Silencioso
    )

    # Treinar modelo
    model.fit(X_train, y_train)

    # Fazer previsões
    predictions = model.predict(X_test)

    return model, predictions
```

Explicação Técnica:

- **Leaf-wise Growth**: Crescimento por folha
- **Histogram-based**: Binning de features
- **GOSS**: Gradient-based One-Side Sampling
- **EFB**: Exclusive Feature Bundling

Vantagens:

- Muito rápido
- Baixo uso de memória
- Boa performance
- Regularização integrada

Desvantagens:

- Sensível a overfitting
- Requer tuning cuidadoso
- Menos interpretável

2.4 Otimização de Hiperparâmetros

2.4.1 Grid Search

```
def optimize_hyperparameters(model_class, X_train, y_train, param_grid):
    """
    Otimização de hiperparâmetros com Grid Search
    """
    from sklearn.model_selection import GridSearchCV, TimeSeriesSplit

    # Time Series Cross-Validation
    tscv = TimeSeriesSplit(n_splits=5)

    # Grid Search
    grid_search = GridSearchCV(
        estimator=model_class,
        param_grid=param_grid,
        cv=tscv,
        scoring='neg_mean_absolute_error',
        n_jobs=-1,
        verbose=1
    )

    # Executar busca
    grid_search.fit(X_train, y_train)

    return grid_search.best_estimator_, grid_search.best_params_
```

2.4.2 Parâmetros Otimizados

```
# Random Forest
rf_params = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10]
}

# XGBoost
xgb_params = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 6, 9],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.8, 0.9, 1.0]
}

# LightGBM
lgb_params = {
    'n_estimators': [50, 100, 200],
```

```
    'max_depth': [3, 6, 9],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.8, 0.9, 1.0]
}
```

2.5 Validação Temporal

2.5.1 Time Series Cross-Validation

```
def time_series_cv(model, X, y, n_splits=5):
    """
    Validação cruzada temporal
    """
    from sklearn.model_selection import TimeSeriesSplit
    from sklearn.metrics import mean_absolute_error

    tscv = TimeSeriesSplit(n_splits=n_splits)
    scores = []

    for train_idx, val_idx in tscv.split(X):
        X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
        y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

        # Treinar modelo
        model.fit(X_train, y_train)

        # Fazer previsões
        y_pred = model.predict(X_val)

        # Calcular métrica
        mae = mean_absolute_error(y_val, y_pred)
        scores.append(mae)

    return np.mean(scores), np.std(scores)
```

2.5.2 Walk-Forward Validation

```
def walk_forward_validation(model, X, y, train_size=0.8):
    """
    Validação walk-forward
    """
    n_train = int(len(X) * train_size)

    # Dados de treino
    X_train, y_train = X[:n_train], y[:n_train]

    # Dados de teste
```

```
    X_test, y_test = X[n_train:], y[n_train:]

    # Treinar modelo
    model.fit(X_train, y_train)

    # Fazer previsões
    y_pred = model.predict(X_test)

    return y_test, y_pred
```

---

3. CONFIGURAÇÃO FINAL E RESULTADOS

3.1 Configuração Vencedora

3.1.1 Variáveis Exógenas Selecionadas

```
final_exog_vars = [
    'TAXA_SELIC',           # Taxa básica de juros
    'IPCA',                 # Índice de preços
    'TAXA_DESOCUPACAO',     # Taxa de desemprego
    'INADIMPLENCIA'         # Taxa de inadimplência
]
```

Justificativa:

- **Baixa multicolinearidade**: VIF < 5 para todas
- **Relevância econômica**: Impacto direto na litigiosidade
- **Disponibilidade**: Dados consistentes e atualizados
- **Interpretabilidade**: Fácil compreensão pelos stakeholders

3.1.2 Modelo Vencedor: Prophet

```
# Configuração final do Prophet
best_model = Prophet(
    yearly_seasonality=True,        # Sazonalidade anual
    weekly_seasonality=False,     # Sem sazonalidade semanal
    daily_seasonality=False,        # Sem sazonalidade diária
    seasonality_mode='additive',    # Sazonalidade aditiva
    interval_width=0.95,           # Intervalo de confiança 95%
    changepoint_prior_scale=0.05,  # Sensibilidade a mudanças
    seasonality_prior_scale=10.0   # Força da sazonalidade
)

# Adicionar regressores exógenos
for var in final_exog_vars:
    best_model.add_regressor(var)
```

3.2 Performance Comparativa

Modelo	MAE	RMSE	R^2	Variáveis Exógenas
Prophet (Teste)	**3.634**	**4.597**	**0.339**	4 econômicas tradicionais
Prophet (Completo)	6.472	7.313	-0.245	15 variáveis (incluindo alta correlação)
Random Forest	6.827	7.874	-0.939	4 econômicas + features temporais
XGBoost	7.669	8.918	-1.487	4 econômicas + features temporais
LightGBM	7.464	8.876	-1.464	4 econômicas + features temporais
SARIMAX	9.416	11.290	-2.986	4 econômicas tradicionais

3.3 Lições Aprendidas

3.3.1 Variáveis Exógenas

1. **Qualidade > Quantidade**: 4 variáveis bem escolhidas > 15 variáveis
2. **Multicolinearidade**: Variáveis altamente correlacionadas diminuem performance
3. **Relevância Econômica**: Variáveis econômicas tradicionais são mais eficazes
4. **Feature Engineering**: Lags e rolling statistics são essenciais

3.3.2 Modelos

1. **Simplicidade vence complexidade**: Prophet simples > modelos complexos
2. **Validação Temporal**: Crucial para séries temporais
3. **Interpretabilidade**: Modelos interpretáveis são preferíveis
4. **Robustez**: Modelos robustos a outliers são mais confiáveis

---

4. RECOMENDAÇÕES TÉCNICAS

4.1 Implementação em Produção

4.1.1 Pipeline de Dados

```
def production_pipeline():
    """
    Pipeline de produção para previsões
    """
    # 1. Carregar dados atualizados
    data = load_latest_data()

    # 2. Preparar variáveis exógenas
    exog_data = prepare_exogenous_variables(data)
```

```
    # 3. Treinar modelo
    model = train_prophet_model(exog_data)

    # 4. Fazer previsões
    forecast = model.predict(future_data)

    # 5. Validar previsões
    validation_results = validate_forecast(forecast)

    return forecast, validation_results
```

4.1.2 Monitoramento

```
def monitor_model_performance():
    """
    Monitoramento da performance do modelo
    """
    # 1. Calcular métricas recentes
    recent_mae = calculate_recent_mae()

    # 2. Detectar drift
    if detect_data_drift():
        send_alert("Data drift detected")

    # 3. Verificar performance
    if recent_mae > threshold:
        send_alert("Model performance degraded")

    # 4. Retreinar se necessário
    if should_retrain():
        retrain_model()
```

4.2 Manutenção e Atualização

4.2.1 Retreinamento Automático

- **Frequência**: Mensal
- **Trigger**: Performance degradada ou novos dados
- **Validação**: Cross-validation temporal
- **Deploy**: A/B testing com modelo anterior

4.2.2 Monitoramento de Variáveis Exógenas

- **Disponibilidade**: Verificar atualização mensal
- **Qualidade**: Detectar outliers e missing values
- **Relevância**: Avaliar correlação com variável alvo
- **Substituição**: Identificar novas variáveis relevantes

5. CONCLUSÕES

5.1 Descobertas Principais

1. **Variáveis Exógenas Econômicas Tradicionais** são mais eficazes que variáveis de alta correlação
2. **Modelos Simples** superam abordagens complexas (princípio da parcimônia)
3. **Prophet** é superior para séries temporais com sazonalidade
4. **Feature Engineering** é crucial para modelos de ML
5. **Validação Temporal** é essencial para séries temporais

5.2 Recomendações Finais

1. **Usar Prophet** com 4 variáveis econômicas tradicionais
2. **Implementar retreinamento mensal** automático
3. **Monitorar performance** continuamente
4. **Expandir gradualmente** para outros tipos de processo
5. **Documentar decisões** para reprodutibilidade

5.3 Próximos Passos

1. **Implementação em produção** com monitoramento
2. **Expansão para outros tribunais** usando metodologia
3. **Desenvolvimento de dashboard** executivo
4. **Treinamento da equipe** técnica
5. **Pesquisa de novas variáveis** exógenas relevantes

Equipe e Contato

- **Autores** Eng. Manuel Lucala Zengo - DIACDE - TJGO
- **Mentoria** - Fernando Ribeiro Trindade, Dra. Deborah Silva Alves Fernandes e Marcio Giovane
- **Metodologia** - CRISP-DM adaptada para séries temporais

Data: Outubro de 2025
Versão: 1.0
Status: Em Avaliacao