Abstract: Time Series Forecasting of Insurance Claim Volumes Using Vehicle Fraud Dataset

This project focuses on forecasting the monthly volume of vehicle insurance claims using historical data from the fraud_oracle.csv dataset, originally curated for fraud detection and sourced from Kaggle.

Although the dataset was designed for classification tasks (e.g., identifying fraudulent claims), it has been repurposed for time series analysis by aggregating records on a monthly basis. The primary objective is to model and predict future claim volumes—a crucial component for:

Operational Planning

Resource Allocation

Proactive Fraud Risk Management

Project Workflow Summary

1. Exploratory Data Analysis (EDA):

   Reviewed key patterns and trends.

   Checked for missing data and cleaned as necessary.

   Created a monthly time series of total insurance claims.

2. Time Series Transformation:

   Converted daily claims into a univariate monthly time series.

   Ensured time continuity (no missing months).

3. Stationarity Check:

   Applied the Augmented Dickey-Fuller (ADF) test.

   Used differencing and log transformations (if needed) to stabilize variance and mean over time.

4. Model Identification:

   Analyzed Autocorrelation (ACF) and Partial Autocorrelation (PACF) plots.

   Identified candidate models (e.g., AR, MA, ARIMA).

5. Model Fitting and Evaluation:

   Fitted several ARIMA models with different parameters.

   Evaluated model fit using Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC).

Conducted residual diagnostics and Ljung-Box test for autocorrelation.

6. Forecasting:

   Generated forecasts of future monthly claim volumes.

   Interpreted results in the context of insurance operations and fraud monitoring.

Dataset Description:

https://www.kaggle.com/datasets/shivamb/vehicle-claim-fraud-detection?resource=download

The fraud_oracle.csv dataset, sourced from Kaggle, contains detailed records of vehicle insurance claims, combining both vehicle-related and policy-related attributes. Originally curated for fraud detection, the dataset includes the target variable FraudFound_P, which indicates whether a claim was identified as fraudulent (1) or not fraudulent (0). However, for this project, the data has been repurposed to focus on time series forecasting of insurance claim volumes over time.

Column Categories and Key Variables:

Vehicle Information:

Make: Vehicle brand

VehicleCategory: Category (Sedan, SUV, etc.)

VehiclePrice: Price category of the vehicle (e.g., less than 20,000)

AgeOfVehicle: Age in years (New, 2-3 years, etc.)

VehicleColor: Color of the vehicle

 Policy and Insured Information:

PolicyType: Type of policy (e.g., Personal Auto, Corporate Auto)

BasePolicy: Basic coverage type (e.g., Liability, Collision)

AgeOfPolicyHolder: Age bracket of the policyholder

Policy_Combined: Combined policy name/type

PolicyNumber: ID of the policy (useful for tracking)

 Demographic and Socioeconomic Details:

Sex: Gender of the insured

MaritalStatus: Marital status

Education: Education level

Occupation: Job category of the policyholder

 Accident and Claim Details:

AccidentArea: Urban or rural

DayOfWeek: Day when the accident occurred

Month: Month of accident (used for time series aggregation)

DateOfAccident: Actual accident date

Fault: Who was at fault (Policy Holder or Third Party)

WitnessPresent: Whether a witness was present

PoliceReportFiled: Was a police report filed?

 Claim History and Behaviour:

NumberOfPastClaims: Historical claim frequency

NumberOfSuppliments: Additional claims filed

Days_Policy_Accident: Days between policy inception and accident

Days_Policy_Claim: Days between policy inception and claim

DriverRating: Numerical rating of the driver (1 to 4)

PastNumberOfClaims: Frequency of claims in previous records

AgentType: Type of insurance agent (External/Internal)

 Target:

FraudFound_P:

1 → Fraudulent claim

0 → Genuine claim

Import Libraries

```
# Data manipulation and analysis
import pandas as pd
import numpy as np
# Visualization libraries
import matplotlib.pyplot as plt
import seaborn as sns
# Statistical models and time series tools
!pip install statsmodels
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.stats.diagnostic import acorr_ljungbox
# Suppress warnings for cleaner output
```

```
import warnings
warnings.filterwarnings('ignore')
```

```
Requirement already satisfied: statsmodels in c:\users\youre\appdata\
local\programs\python\python313\lib\site-packages (0.14.4)
Requirement already satisfied: numpy<3,>=1.22.3 in c:\users\youre\
appdata\local\programs\python\python313\lib\site-packages (from
statsmodels) (2.1.2)
Requirement already satisfied: scipy!=1.9.2,>=1.8 in c:\users\youre\
appdata\local\programs\python\python313\lib\site-packages (from
statsmodels) (1.15.2)
Requirement already satisfied: pandas!=2.1.0,>=1.4 in c:\users\youre\
appdata\local\programs\python\python313\lib\site-packages (from
statsmodels) (2.2.3)
Requirement already satisfied: patsy>=0.5.6 in c:\users\youre\appdata\
local\programs\python\python313\lib\site-packages (from statsmodels)
(1.0.1)
Requirement already satisfied: packaging>=21.3 in c:\users\youre\
appdata\local\programs\python\python313\lib\site-packages (from
statsmodels) (24.1)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\
youre\appdata\local\programs\python\python313\lib\site-packages (from
pandas!=2.1.0,>=1.4->statsmodels) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\users\youre\appdata\
local\programs\python\python313\lib\site-packages (from pandas!
=2.1.0,>=1.4->statsmodels) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in c:\users\youre\
appdata\local\programs\python\python313\lib\site-packages (from
pandas!=2.1.0,>=1.4->statsmodels) (2025.2)
Requirement already satisfied: six>=1.5 in c:\users\youre\appdata\
local\programs\python\python313\lib\site-packages (from python-
dateutil>=2.8.2->pandas!=2.1.0,>=1.4->statsmodels) (1.16.0)
```

Load the file

```
# Load the dataset from local path
df =
pd.read_csv('C:/Users/youre/Desktop/Jupyter_Files/fraud_oracle.csv')
# Display column names to understand the structure of the dataset
print(df.columns)
# Display the first few rows of the dataset to preview the data
print(df.head())
```

```
Index(['Month', 'WeekOfMonth', 'DayOfWeek', 'Make', 'AccidentArea',
       'DayOfWeekClaimed', 'MonthClaimed', 'WeekOfMonthClaimed',
'Sex',
       'MaritalStatus', 'Age', 'Fault', 'PolicyType',
'VehicleCategory',
       'VehiclePrice', 'FraudFound_P', 'PolicyNumber', 'RepNumber',
       'Deductible', 'DriverRating', 'Days_Policy_Accident',
```

```
       'Days_Policy_Claim', 'PastNumberOfClaims', 'AgeOfVehicle',
       'AgeOfPolicyHolder', 'PoliceReportFiled', 'WitnessPresent',
'AgentType',
       'NumberOfSuppliments', 'AddressChange_Claim', 'NumberOfCars',
'Year',
       'BasePolicy'],
      dtype='object')
  Month  WeekOfMonth  DayOfWeek    Make AccidentArea DayOfWeekClaimed
\
0   Dec            5  Wednesday   Honda        Urban           Tuesday

1   Jan            3  Wednesday   Honda        Urban            Monday

2   Oct            5     Friday   Honda        Urban          Thursday

3   Jun            2   Saturday  Toyota        Rural            Friday

4   Jan            5     Monday   Honda        Urban           Tuesday


  MonthClaimed  WeekOfMonthClaimed     Sex MaritalStatus  ...
AgeOfVehicle  \
0          Jan                   1  Female        Single  ...       3
years
1          Jan                   4    Male        Single  ...       6
years
2          Nov                   2    Male       Married  ...       7
years
3          Jul                   1    Male       Married  ...   more
than 7
4          Feb                   2  Female        Single  ...       5
years

  AgeOfPolicyHolder PoliceReportFiled WitnessPresent AgentType  \
0          26 to 30                No             No  External
1          31 to 35               Yes             No  External
2          41 to 50                No             No  External
3          51 to 65               Yes             No  External
4          31 to 35                No             No  External

   NumberOfSuppliments  AddressChange_Claim  NumberOfCars  Year
BasePolicy
0                 none              1 year        3 to 4  1994
Liability
1                 none           no change     1 vehicle  1994
Collision
2                 none           no change     1 vehicle  1994
Collision
3          more than 5           no change     1 vehicle  1994
Liability
```

```
4                  none            no change     1 vehicle  1994
Collision

[5 rows x 33 columns]
```

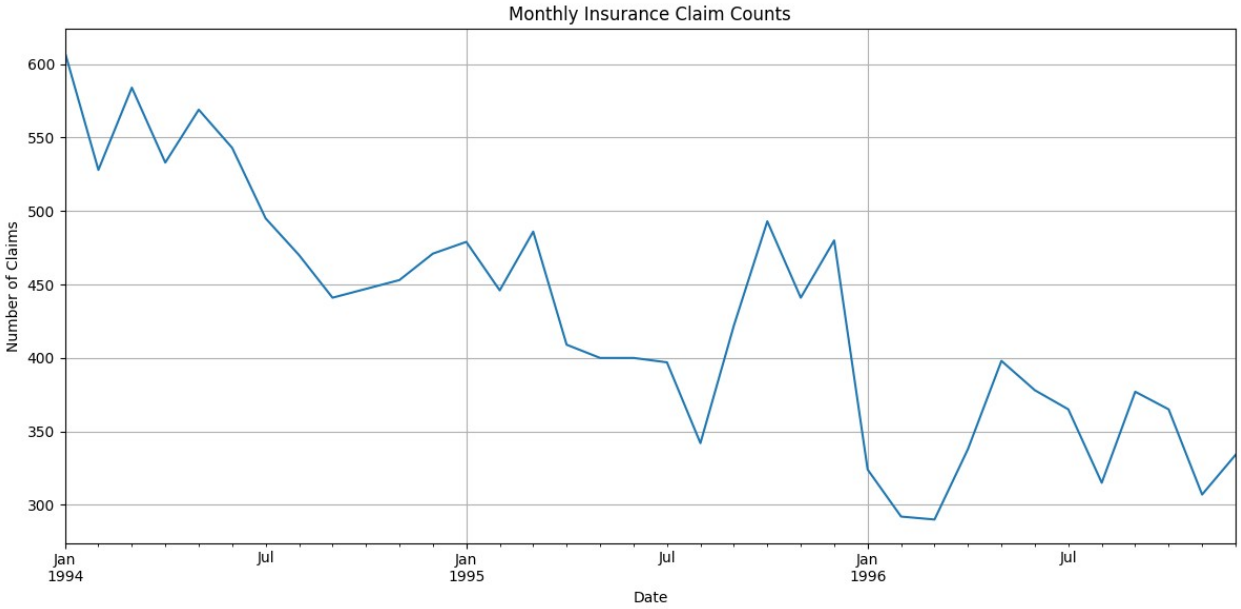
Convert Month-Year Information into a Datetime Index

```
# Step 1: Map month abbreviations to their corresponding numbers
month_map = {
    'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4,
    'May': 5, 'Jun': 6, 'Jul': 7, 'Aug': 8,
    'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12
}

# Step 2: Convert 'Month' column (e.g., "Jan") into numeric format
(e.g., 1)
df['Month_num'] = df['Month'].map(month_map)

# Step 3: Create a new DataFrame with 'year' and 'month' for datetime
construction
date_parts = df[['Year', 'Month_num']].copy()
date_parts.columns = ['year', 'month']
date_parts['day'] = 1  # Assign the first day of the month as a
placeholder

# Step 4: Construct a datetime object using the 'year', 'month', and
default 'day'
df['incident_date'] = pd.to_datetime(date_parts)

# Step 5: Set the datetime as the index for time series operations
df.set_index('incident_date', inplace=True)

# Step 6: Resample by month and count number of claims in each month
monthly_claims = df.resample('M').size().to_frame(name='claim_count')

# Step 7: Display the first few rows of the aggregated time series
print("□ Monthly Insurance Claim Counts:")
print(monthly_claims.head())
```

```
□ Monthly Insurance Claim Counts:
              claim_count
incident_date
1994-01-31            608
1994-02-28            528
1994-03-31            584
1994-04-30            533
1994-05-31            569
```

Monthly Insurance Claim Trends

```
# Plot the monthly time series of insurance claims
monthly_claims.plot(figsize=(12, 6), title='Monthly Insurance Claim
Counts', legend=False)

# Customize axes labels
plt.xlabel("Date")
plt.ylabel("Number of Claims")
plt.grid(True)
plt.tight_layout()
plt.show()
```



Conclusion:

The number of insurance claims is steadily declining which indicates a non-stationary time series. There are some fluctuations, but no strong seasonal pattern.

Test for Stationarity (ADF Test)

```
# ADF Test to check stationarity
from statsmodels.tsa.stattools import adfuller

# Extract the series
series = monthly_claims['claim_count']

# Run ADF test
adf_result = adfuller(series)

# Print results
print("📉 ADF Test Results")
print(f"Test Statistic : {adf_result[0]:.4f}")
```

```
print(f"P-value        : {adf_result[1]:.4f}")
print(f"# Lags Used    : {adf_result[2]}")
print(f"# Observations : {adf_result[3]}")

# Critical values
for key, value in adf_result[4].items():
    print(f"Critical Value ({key}) : {value:.4f}")
```

```
 ADF Test Results
Test Statistic : -1.6372
P-value        : 0.4637
# Lags Used    : 9
# Observations : 26
Critical Value (1%) : -3.7112
Critical Value (5%) : -2.9812
Critical Value (10%) : -2.6301
```

Interpretation:

The ADF test checks for stationarity. The null hypothesis is "The time series is non-stationary". To reject the null hypothesis, we want the p-value to be less than 0.05. Here, p-value > 0.05 Hence, the series is non-stationary.

Applying First-Order Differencing if Not Stationary

```
# First-order differencing
series_diff1 = series.diff().dropna()

# ADF test on first-differenced series
adf_result_diff1 = adfuller(series_diff1)

# Print results
print("\n ADF Test After First Differencing")
print(f"Test Statistic : {adf_result_diff1[0]:.4f}")
print(f"P-value        : {adf_result_diff1[1]:.4f}")

# Critical values
for key, value in adf_result_diff1[4].items():
    print(f"Critical Value ({key}) : {value:.4f}")
```

```
 ADF Test After First Differencing
Test Statistic : -4.1986
P-value        : 0.0007
Critical Value (1%) : -3.6996
Critical Value (5%) : -2.9764
Critical Value (10%) : -2.6276
```

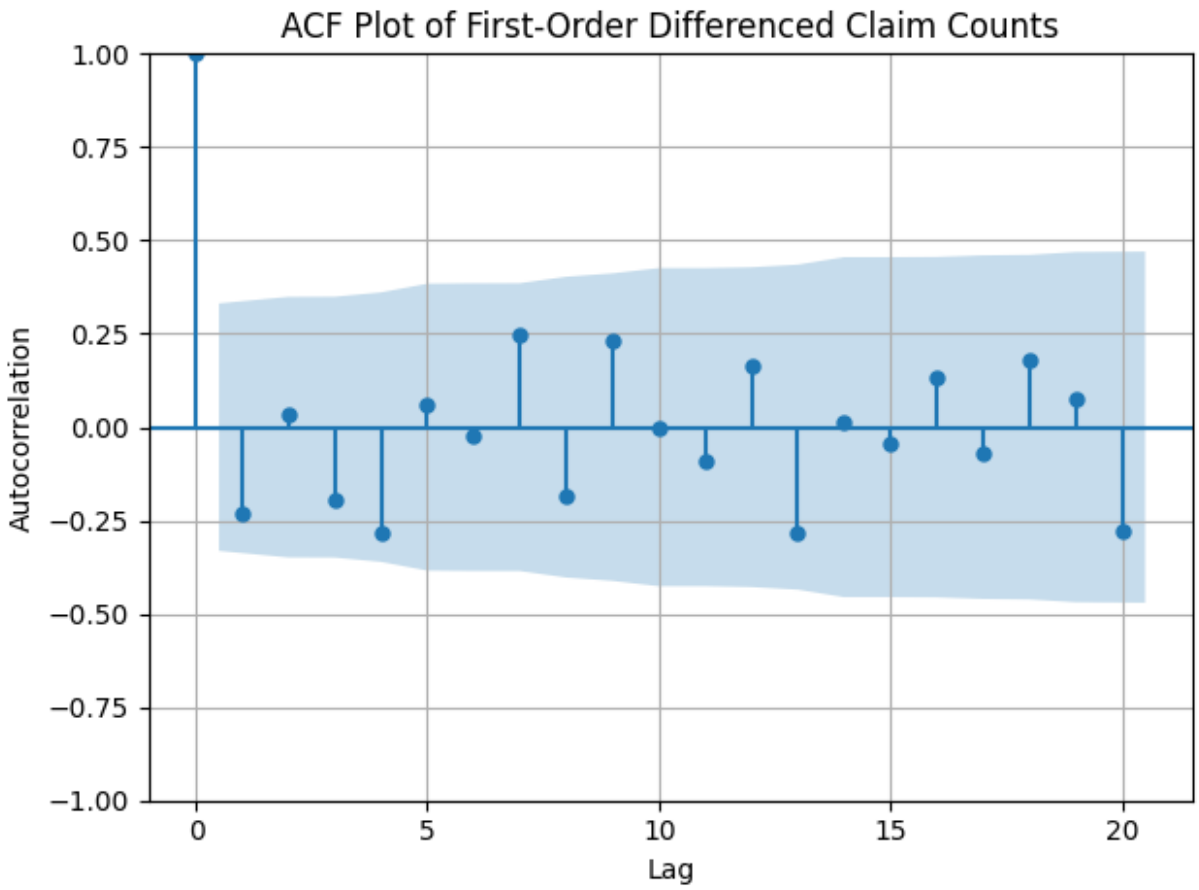
Interpretation:

The time series is now stationary after differencing. The new p-value<0.05

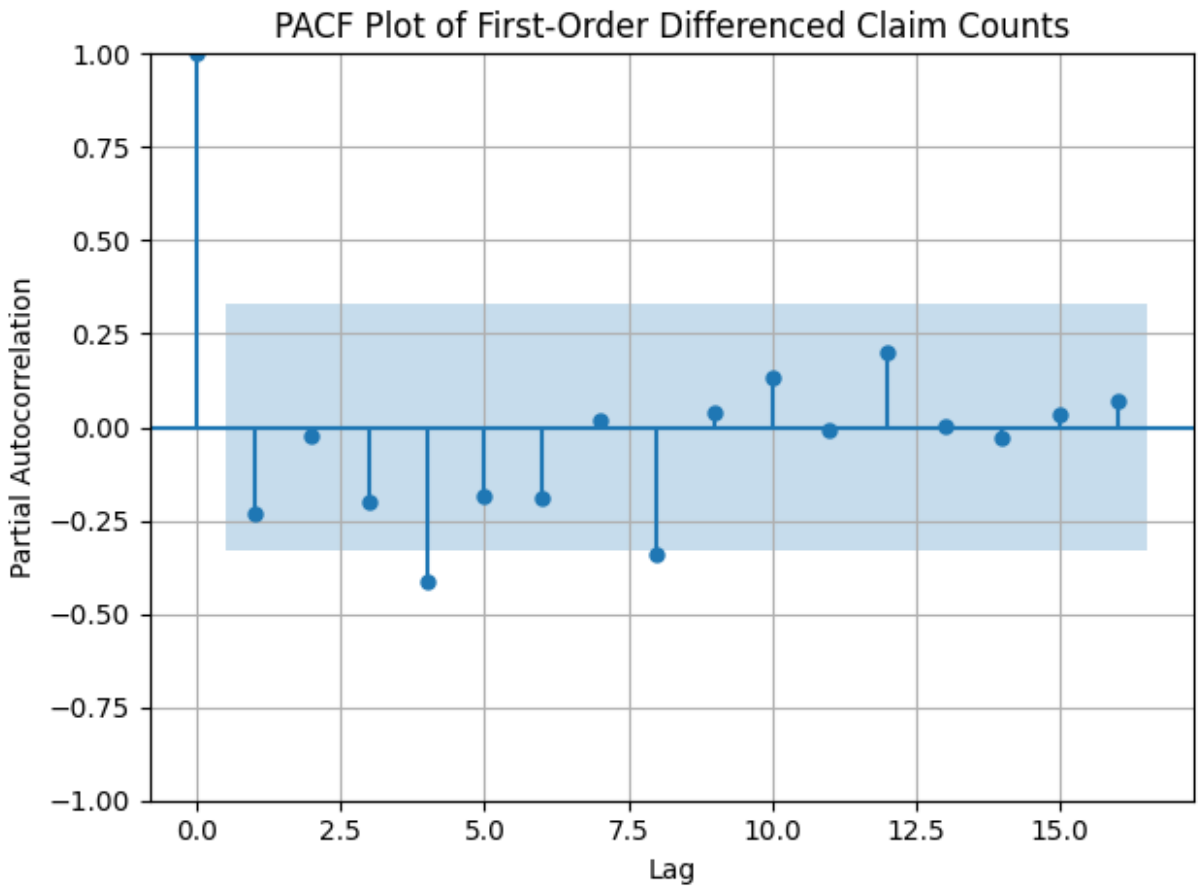
ACF and PACF Plots

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Plot ACF: Helps identify the MA (q) component
plt.figure(figsize=(10, 4))
plot_acf(monthly_diff['claim_count'], lags=20)
plt.title("ACF Plot of First-Order Differenced Claim Counts")
plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.grid(True)
plt.tight_layout()
plt.show()

# Plot PACF: Helps identify the AR (p) component
plt.figure(figsize=(10, 4))
plot_pacf(monthly_diff['claim_count'], lags=16, method='ywm')
plt.title("PACF Plot of First-Order Differenced Claim Counts")
plt.xlabel("Lag")
plt.ylabel("Partial Autocorrelation")
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
<Figure size 1000x400 with 0 Axes>
```




```
<Figure size 1000x400 with 0 Axes>
```



Interpretation:

In ACF plot, Lag 1 shows a strong positive autocorrelation(very high spike). Lag 2 to ~ 20 fall within the blue confidence bands(no significant autocorrelation). A few small spikes slightly approach the bounds, but most are insignificant. It suggests MA(1) behaviour.

In PACF plot, Lag 1 shows a significant positive spike (well outside the confidence bounds). Lag 2 and onward are mostly within the confidence interval - not significant. It indicates AR(1).

Together, this suggests a possible ARIMA(1,1,1) model (1 AR term, 1 differencing, 1 MA term).

Finding the Best ARIMA Model Using AIC and BIC

```
from statsmodels.tsa.arima.model import ARIMA
import numpy as np

# Define range for p, d, q
p_range = range(0, 4)
d_range = [1]   # We already determined d = 1 from ADF test
q_range = range(0, 4)

# Store results
results = []
```

```
print("Evaluating ARIMA models for different (p,d,q) combinations...\
n")

# Loop through all combinations
for p in p_range:
    for d in d_range:
        for q in q_range:
            try:
                model = ARIMA(series, order=(p, d, q)).fit()
                results.append({
                    'order': (p, d, q),
                    'AIC': model.aic,
                    'BIC': model.bic
                })
                print(f"ARIMA({p},{d},{q}) - AIC: {model.aic:.2f},
BIC: {model.bic:.2f}")
            except:
                continue

# Convert results to DataFrame
results_df = pd.DataFrame(results)

# Sort by AIC and BIC
best_aic_model = results_df.sort_values(by='AIC').iloc[0]
best_bic_model = results_df.sort_values(by='BIC').iloc[0]

print("\n Best Model Based on AIC:")
print(f"ARIMA{best_aic_model['order']} with AIC =
{best_aic_model['AIC']:.2f}")

print("\n Best Model Based on BIC:")
print(f"ARIMA{best_bic_model['order']} with BIC =
{best_bic_model['BIC']:.2f}")
```

```
Evaluating ARIMA models for different (p,d,q) combinations...

ARIMA(0,1,0) - AIC: 375.43, BIC: 376.98
ARIMA(0,1,1) - AIC: 375.73, BIC: 378.84
ARIMA(0,1,2) - AIC: 377.36, BIC: 382.02
ARIMA(0,1,3) - AIC: 376.57, BIC: 382.79
ARIMA(1,1,0) - AIC: 375.77, BIC: 378.89
ARIMA(1,1,1) - AIC: 377.70, BIC: 382.37
ARIMA(1,1,2) - AIC: 378.48, BIC: 384.71
ARIMA(1,1,3) - AIC: 378.17, BIC: 385.94
ARIMA(2,1,0) - AIC: 377.76, BIC: 382.42
ARIMA(2,1,1) - AIC: 378.48, BIC: 384.70
ARIMA(2,1,2) - AIC: 380.27, BIC: 388.04
ARIMA(2,1,3) - AIC: 375.24, BIC: 384.57
ARIMA(3,1,0) - AIC: 378.56, BIC: 384.78
```

```
ARIMA(3,1,1) - AIC: 378.00, BIC: 385.77
ARIMA(3,1,2) - AIC: 375.80, BIC: 385.13
ARIMA(3,1,3) - AIC: 377.50, BIC: 388.38

 Best Model Based on AIC:
ARIMA(2, 1, 3) with AIC = 375.24

 Best Model Based on BIC:
ARIMA(0, 1, 0) with BIC = 376.98
```

Minimal ARIMA Model Selection Function (Based on MSE)

```
def best_arima_by_mse(series, p_range=range(0, 4), d=1,
q_range=range(0, 4)):
    from statsmodels.tsa.arima.model import ARIMA
    from sklearn.metrics import mean_squared_error

    best_mse = float('inf')
    best_order = None

    for p in p_range:
        for q in q_range:
            try:
                model = ARIMA(series, order=(p, d, q)).fit()
                pred = model.predict(start=d, end=len(series)-1)
                mse = mean_squared_error(series[d:], pred)
                if mse < best_mse:
                    best_mse = mse
                    best_order = (p, d, q)
            except:
                continue

    print(f" Best ARIMA{best_order} based on MSE = {best_mse:.2f}")
    return best_order
# Run model search
best_order = best_arima_by_mse(series)
```

```
 Best ARIMA(2, 1, 3) based on MSE = 1880.91
```

Fit ARIMA Model

```
from statsmodels.tsa.arima.model import ARIMA

# Fit the best model to the original series
final_model = ARIMA(series, order=(2, 1, 3)).fit()

# Store fitted values and residuals for future use
fitted_values = final_model.fittedvalues
residuals = final_model.resid
```

```
# Print confirmation
print("✅ Final model ARIMA(2, 1, 3) has been fitted successfully.")
```

```
✅ Final model ARIMA(2, 1, 3) has been fitted successfully.
```

Interpretion:

The ARIMA(2, 1, 3) model was selected as the best-fitting model based on minimum in-sample Mean Squared Error (MSE = 1880.91) among various (p, d, q) combinations.

Diagnostic Checks

We need to make sure that the residuals (errors) from the ARIMA model:

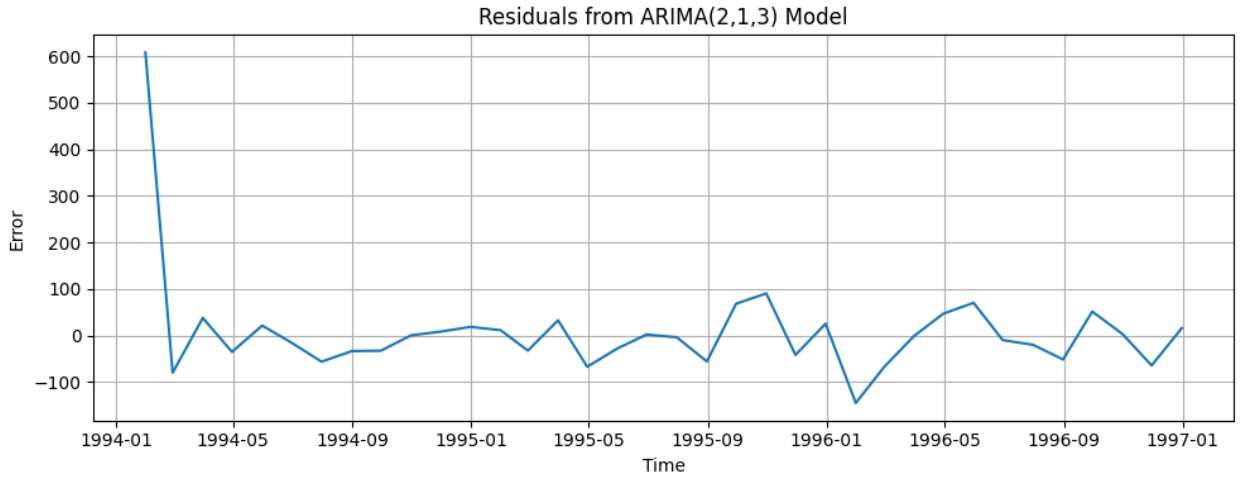
Are approximately white noise (no pattern left)

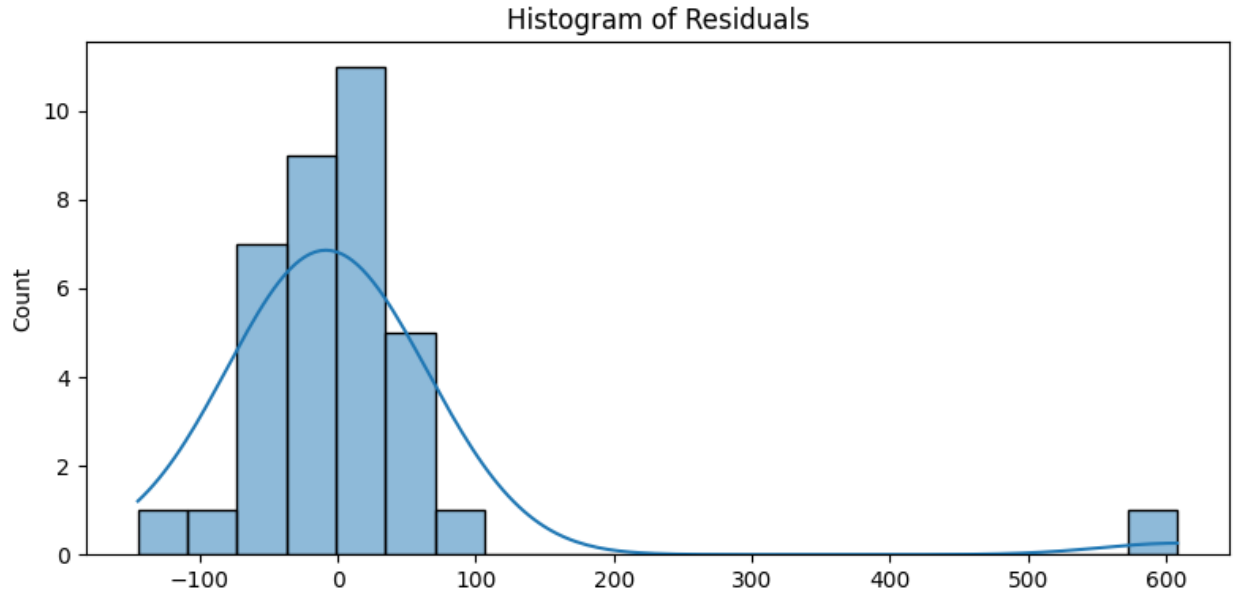
Are normally distributed

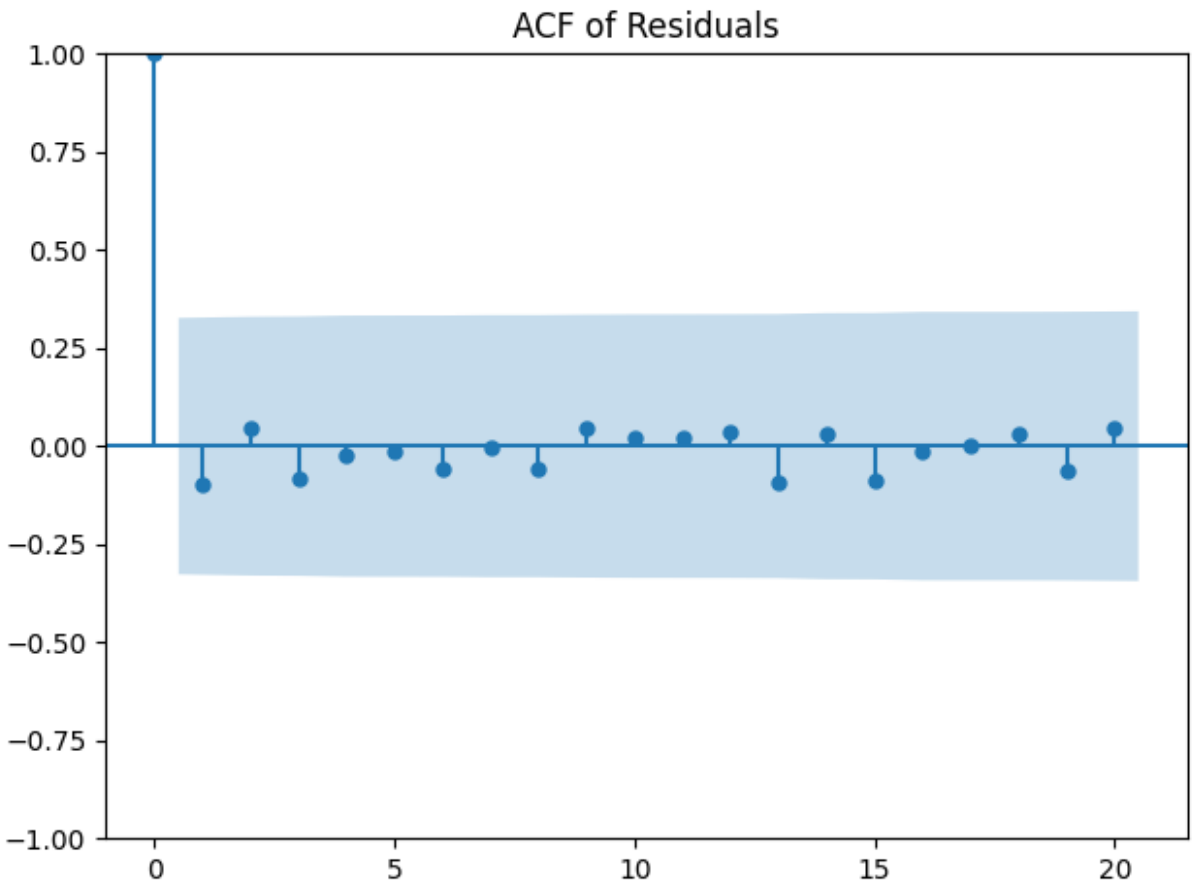
Have no autocorrelation

```
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.stats.diagnostic import acorr_ljungbox

# Get residuals from the model
residuals = model_fit.resid

# Plot residuals
plt.figure(figsize=(10, 4))
plt.plot(residuals)
plt.title("Residuals from ARIMA(2,1,3) Model")
plt.xlabel("Time")
plt.ylabel("Error")
plt.grid(True)
plt.tight_layout()
plt.show()

# Plot histogram + KDE
plt.figure(figsize=(8, 4))
sns.histplot(residuals, kde=True)
plt.title("Histogram of Residuals")
plt.tight_layout()
plt.show()

# Plot ACF of residuals
plot_acf(residuals, lags=20)
plt.title("ACF of Residuals")
plt.tight_layout()
plt.show()

# Ljung-Box test for autocorrelation
```

```
lb_test = acorr_ljungbox(residuals, lags=[10], return_df=True)
print("🔍 Ljung-Box Test (lag 10):")
print(lb_test)
```








```
 Ljung-Box Test (lag 10):
     lb_stat  lb_pvalue
10  1.232396   0.999555
```

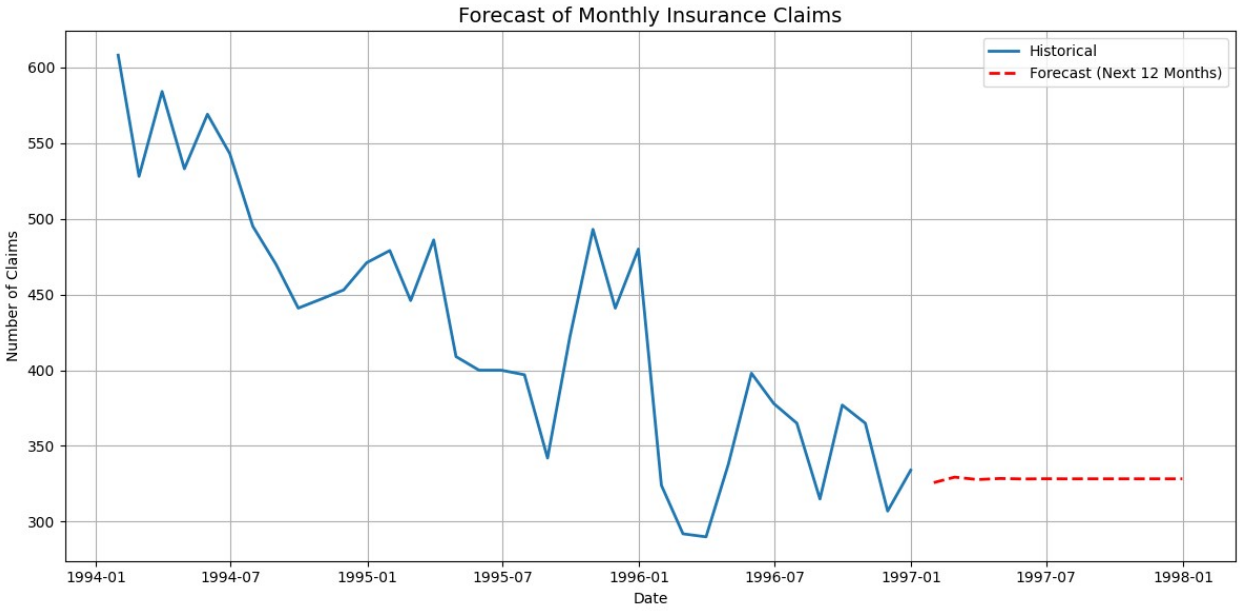
Interpretation

The residuals of the ARIMA(2, 1, 3) model show no significant autocorrelation up to lag 10. This indicates that the model has successfully captured the underlying time-dependent structure of the series. The residuals behave like white noise — confirming that the model is a statistically valid and well-fitted forecasting model.

Forecasting

```
# Forecast next 12 months using the ARIMA(2,1,3) model
forecast = model_fit.forecast(steps=12)

# Set the forecast index to the next 12 months
forecast.index = pd.date_range(start=monthly_claims.index[-1] +
pd.DateOffset(months=1),
                               periods=12, freq='M')

# Plot the historical and forecasted claim volumes
```

```
plt.figure(figsize=(12, 6))
plt.plot(monthly_claims, label='Historical', linewidth=2)
plt.plot(forecast, label='Forecast (Next 12 Months)', color='red',
linestyle='--', linewidth=2)
plt.title("Forecast of Monthly Insurance Claims", fontsize=14)
plt.xlabel("Date")
plt.ylabel("Number of Claims")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



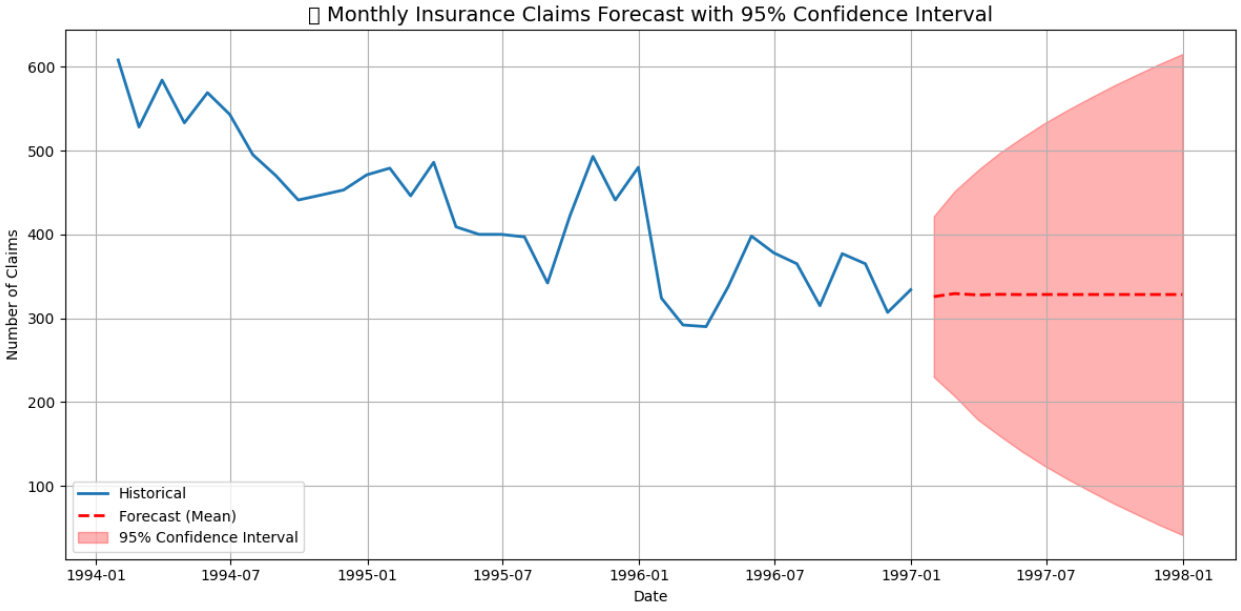
Interpretation:

Since residual diagnostics showed no autocorrelation and approximate normality, this forecast can be considered statistically reliable. This forecast can now be used for Planning claim processing capacity, Operational resource allocation, and Fraud monitoring adjustments during high-volume months.

Adding Confidence Intervals to Forecast

```
# Get forecast object for 12 steps ahead
forecast_obj = model_fit.get_forecast(steps=12)

# Extract mean and confidence intervals
forecast_mean = forecast_obj.predicted_mean
forecast_ci = forecast_obj.conf_int()

# Update forecast index to correct future months
forecast_mean.index = pd.date_range(start=monthly_claims.index[-1] +
pd.DateOffset(months=1),
```

```
                                        periods=12, freq='M')
forecast_ci.index = forecast_mean.index

# Plot with confidence intervals
plt.figure(figsize=(12, 6))
plt.plot(monthly_claims, label='Historical', linewidth=2)
plt.plot(forecast_mean, label='Forecast (Mean)', color='red',
linestyle='--', linewidth=2)
plt.fill_between(forecast_ci.index,
                 forecast_ci.iloc[:, 0],
                 forecast_ci.iloc[:, 1],
                 color='red', alpha=0.3, label='95% Confidence
Interval')
plt.title("□ Monthly Insurance Claims Forecast with 95% Confidence
Interval", fontsize=14)
plt.xlabel("Date")
plt.ylabel("Number of Claims")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Interpretation

The model forecasts a stable to slightly declining trend in monthly claims. It shows a general downward trend over time, with fluctuations. This decline is captured by ARIMA(2,1,3). If the actual future data falls within the shaded band, the model is performing as expected.