D. Saul Jameson FromInputsToInsights.com

```
!pip install cmdstanpy --quiet
```

```
from cmdstanpy import install_cmdstan
install_cmdstan()
```

Show hidden output

```
import pandas as pd
import numpy as np

# Set seed for reproducibility
np.random.seed(42)

# Generate daily data over 3+ years
days = pd.date_range(start="2022-01-01", end="2025-03-31", freq="D")
n = len(days)

# Media spend with variation
google_ads_spend = np.random.normal(1500, 350, n)
facebook_ads_spend = np.random.normal(1200, 325, n)
tv_ads_spend = np.random.normal(1000, 550, n)
influencer_spend = np.random.normal(800, 200, n) + np.linspace(0, 300, n)

# Competitor activity
competitor_activity = np.random.choice([0, 1, 2, 3], size=n, p=[0.2, 0.3, 0.3, 0.2])

# Seasonality
seasonality = days.month.map({
    1: 0.9, 2: 0.9, 3: 1.0, 4: 1.0,
    5: 0.95, 6: 0.9, 7: 0.85, 8: 0.85,
    9: 1.05, 10: 1.2, 11: 1.4, 12: 1.6
}).values

# Black Friday boost
black_friday_boost = np.array([
    1.3 if d.month == 11 and 24 <= d.day <= 30 else 1.0
    for d in days
])

# Organic trend
organic_trend = np.linspace(2000, 5000, n)

# Add randomness to weaken deterministic ROAS mapping
google_ads_revenue = google_ads_spend * 1.2 + np.random.normal(0, 300, n)
facebook_ads_revenue = facebook_ads_spend * 0.95 + np.random.normal(0, 250, n)
tv_ads_revenue = tv_ads_spend * 0.6 + np.random.normal(0, 200, n)
influencer_revenue = influencer_spend * -1.6 + np.random.normal(0, 400, n)

# Total revenue
revenue = (
    google_ads_revenue +
    facebook_ads_revenue +
    tv_ads_revenue +
    influencer_revenue +
    seasonality * 200 +
    organic_trend +
    black_friday_boost * 150 -
    competitor_activity * 100 +
    np.random.normal(0, 10000, n)
)

# Final DataFrame
df = pd.DataFrame({
    "date": days,
    "google_ads": google_ads_spend,
    "facebook_ads": facebook_ads_spend,
    "tv_ads": tv_ads_spend,
    "influencer": influencer_spend,
    "google_ads_revenue": google_ads_revenue,
    "facebook_ads_revenue": facebook_ads_revenue,
```

```
    "tv_ads_revenue": tv_ads_revenue,
    "influencer_revenue": influencer_revenue,
    "competitor_activity": competitor_activity,
    "seasonality_index": seasonality,
    "black_friday_boost": black_friday_boost,
    "organic_trend": organic_trend,
    "revenue": revenue
})
```

```
# Define the features to include in the model
features = [
    "google_ads", "facebook_ads", "tv_ads", "influencer",
    "competitor_activity", "seasonality_index",
    "black_friday_boost", "organic_trend"
]

# Drop any rows with missing values (just in case)
df_model = df[features + ["revenue"]].dropna()

# Build the design matrix and target
X = df_model[features].values
y = df_model["revenue"].values
N, K = X.shape

# Create Stan-formatted dictionary
stan_data = {
    "N": N,
    "K": K,
    "X": X,
    "y": y
}
```

The inputs (X) and the target variable (y) are separated. X includes all predictor variables such as media spend and control features, while y represents the outcome variable, typically revenue or conversions. The number of observations (N) and the number of predictors (K) are counted to inform the model structure. A dictionary is then created to store the model inputs (N, K, X, y) in the format required by Bayesian modeling frameworks like Stan. This setup allows the model to estimate the effect of each media channel on the outcome by generating probability distributions for each coefficient.

```
stan_code = """
data {
  int<lower=0> N;          // number of observations
  int<lower=0> K;          // number of predictors
  matrix[N, K] X;          // feature matrix
  vector[N] y;             // revenue
}

parameters {
  real alpha;              // intercept
  vector[K] beta;          // coefficients
  real<lower=0> sigma;     // noise std dev
}

model {
  // Priors
  beta ~ normal(0, 1);
  alpha ~ normal(0, 10);
  sigma ~ normal(0, 20000);

  // Likelihood
  y ~ normal(X * beta + alpha, sigma);
}
"""

with open("bayesian_mmm.stan", "w") as f:
    f.write(stan_code)
```

This block defines the structure of a Bayesian linear regression model in Stan and saves it to a file named bayesian_mmm.stan.

The data block declares the inputs needed for the model:

N is the number of observations.

K is the number of predictor variables.

X is the feature matrix with media and control inputs.

y is the outcome variable, such as revenue.

The parameters block defines what the model will learn:

alpha is the intercept.

beta is a vector of coefficients for the K predictors.

sigma is the standard deviation of the noise (model uncertainty), constrained to be positive.

The model block sets the prior beliefs and the likelihood:

Priors assume beta values come from a normal distribution centered at 0 with standard deviation 1, implying an assumption that most media channels have small effects but allowing for variation.

The intercept alpha has a broader prior, allowing for a wider range of baseline revenue.

The sigma prior allows for large noise variance, reflecting uncertainty in revenue predictions.

The likelihood statement tells the model that the observed revenue y is normally distributed around the predicted values X * beta + alpha, with noise level sigma.

Finally, the code writes this model specification to a .stan file, which can later be compiled and used for Bayesian inference.

Double-click (or enter) to edit

```
from cmdstanpy import CmdStanModel



# Compile the Stan model
model = CmdStanModel(stan_file="bayesian_mmm.stan")

# Fit the model using MCMC
fit = model.sample(
    data=stan_data,
    chains=4,
    parallel_chains=4,
    iter_sampling=1000,
    iter_warmup=500,
    seed=42,
    show_progress=True
)
```

Show hidden output

The Stan model is compiled from the bayesian_mmm.stan file using CmdStanModel, which prepares the model for sampling. Once compiled, the model is fit using Markov Chain Monte Carlo (MCMC) via the .sample() method.

The model runs 4 chains in parallel to improve the robustness and convergence diagnostics of the posterior estimates. Each chain draws 1,000 samples after a warmup phase of 500 iterations. The 500 warmup iterations allow the sampler to adapt to the shape of the posterior distribution and tune internal parameters like step size. The choice of 1,000 post-warmup samples per chain (4,000 total) balances computational efficiency with the need for enough samples to produce stable posterior estimates.

A random seed is set to 42 for reproducibility, ensuring that results can be replicated exactly. Running multiple chains and setting a reasonable number of warmup and sampling iterations are best practices in Bayesian inference to ensure the model converges properly and returns reliable estimates. Additional arguments like adapt_delta, max_treedepth, or thin could also be used for fine-tuning if convergence diagnostics suggest it.

```
# Extract posterior samples into a DataFrame
posterior = fit.draws_pd()
```

This line extracts the posterior samples generated by the MCMC process into a pandas DataFrame.

Each row in the DataFrame represents a single draw from the posterior distribution, and each column corresponds to a parameter in the model—such as alpha (intercept), beta[1], beta[2], etc., and sigma (noise).

This format makes it easy to explore, visualize, and summarize the results using standard Python tools. For example, you can compute means, credible intervals, or plot distributions for each parameter to understand the estimated effect of each media channel on revenue.

```
# Get all beta columns from the posterior
beta_cols = [col for col in posterior.columns if col.startswith("beta[")]

# Match the order of features
feature_names = [
    "google_ads", "facebook_ads", "tv_ads", "influencer",
    "competitor_activity", "seasonality_index",
    "black_friday_boost", "organic_trend"
]

# Map beta[i] to real feature names
rename_map = {f"beta[{i+1}]": feature_names[i] for i in range(len(feature_names))}
media_df = posterior[beta_cols].rename(columns=rename_map)

# Show summary stats for all predictors
media_summary = media_df.describe(percentiles=[0.025, 0.5, 0.975]).T
media_summary = media_summary.rename(columns={
    "mean": "Mean Coefficient",
    "std": "Std Dev",
    "2.5%": "2.5%",
    "50%": "Median",
    "97.5%": "97.5%"
})

media_summary.round(4)
```

	count	Mean Coefficient	Std Dev	min	2.5%	Median	97.5%	max
google_ads	4000.0	0.6577	0.5483	-1.1928	-0.4034	0.6543	1.7910	3.2006
facebook_ads	4000.0	0.4118	0.5902	-1.8660	-0.7328	0.4070	1.5895	2.6206
tv_ads	4000.0	0.6204	0.4445	-0.9380	-0.2305	0.6285	1.4954	2.0539
influencer	4000.0	-0.5921	0.7878	-3.6642	-2.1580	-0.5835	0.9229	1.8438
competitor_activity	4000.0	0.0003	1.0051	-4.3587	-1.8984	0.0095	2.0089	4.0249
seasonality_index	4000.0	0.0027	0.9790	-3.9551	-1.8881	0.0019	1.9459	3.8139
black_friday_boost	4000.0	-0.0039	0.9764	-4.9502	-1.8725	-0.0102	1.8746	4.2054
organic_trend	4000.0	1.0891	0.2670	0.1040	0.5696	1.0860	1.6125	2.1068

google_ads: The mean coefficient is 0.6577, suggesting that, on average, a one-unit increase in Google Ads spend is associated with a 0.66 unit increase in revenue, holding other variables constant. The 95% credible interval (2.5% to 97.5%) ranges from -0.4034 to 3.2006, showing some uncertainty but skewed toward a positive effect.

influencer: The mean coefficient is -0.5921, indicating that influencer campaigns may have had a negative association with revenue on average. The credible interval from -2.1580 to 1.8438 crosses zero, meaning the model is less certain about the direction of the effect.

```
# Filter last 4 weeks (March 2025)
march_data = df[df["date"] >= "2025-03-01"]

# Compute average for each channel
base_spend = march_data[[
    "google_ads", "facebook_ads", "tv_ads", "influencer"
]].mean()

# Create dummy April weekly plan (4 weeks)
april_plan = pd.DataFrame([base_spend] * 4)
april_plan["week"] = [f"Week {i+1}" for i in range(4)]
april_plan = april_plan[["week", "google_ads", "facebook_ads", "tv_ads", "influencer"]]

april_plan
```

	week	google_ads	facebook_ads	tv_ads	influencer
0	Week 1	1543.814851	1141.810251	1150.361907	1119.338649
1	Week 2	1543.814851	1141.810251	1150.361907	1119.338649
2	Week 3	1543.814851	1141.810251	1150.361907	1119.338649
3	Week 4	1543.814851	1141.810251	1150.361907	1119.338649

```
# Use mean coefficients from posterior to simulate revenue impact
mean_coefs = media_summary["Mean Coefficient"].to_dict()

# Apply coefficients to estimate weekly revenue
april_plan["estimated_revenue"] = (
    april_plan["google_ads"] * mean_coefs["google_ads"] +
    april_plan["facebook_ads"] * mean_coefs["facebook_ads"] +
    april_plan["tv_ads"] * mean_coefs["tv_ads"] +
    april_plan["influencer"] * mean_coefs["influencer"]
)

# Total estimated revenue for April
current_revenue_april = april_plan["estimated_revenue"].sum()
current_revenue_april
```

```
np.float64(6146.153626434751)
```

```
from itertools import product

# Use mean coefficients from the posterior
mean_coefs = media_summary["Mean Coefficient"].to_dict()

# Define budget cap: April can spend up to 20% more than current total
current_total_budget = april_plan[["google_ads", "facebook_ads", "tv_ads", "influencer"]].sum().sum()
max_total_budget = current_total_budget * 1.2

# Updated search grid with higher resolution and wider range
search_space = {
    "google_ads": np.linspace(0.6, 1.4, 81),
    "facebook_ads": np.linspace(0.6, 1.4, 81),
    "tv_ads": np.linspace(0.6, 1.4, 81),
    "influencer": np.linspace(0.6, 1.4, 81),
}

# Base weekly average
base_weekly = april_plan[["google_ads", "facebook_ads", "tv_ads", "influencer"]].mean()

# Grid search for optimal spend plan
best_plan = None
best_revenue = -np.inf

for g, f, t, i in product(search_space["google_ads"],
                          search_space["facebook_ads"],
                          search_space["tv_ads"],
                          search_space["influencer"]):
    test_weekly = {
        "google_ads": g * base_weekly["google_ads"],
        "facebook_ads": f * base_weekly["facebook_ads"],
        "tv_ads": t * base_weekly["tv_ads"],
        "influencer": i * base_weekly["influencer"],
    }
    test_total = sum(test_weekly.values()) * 4
    if test_total > max_total_budget:
        continue

    # Add slight noise to simulate uncertainty and prevent flat solutions
    beta_jitter = {k: np.random.normal(mean_coefs[k], 0.01) for k in mean_coefs}

    expected_revenue = (
        test_weekly["google_ads"] * beta_jitter["google_ads"] +
        test_weekly["facebook_ads"] * beta_jitter["facebook_ads"] +
        test_weekly["tv_ads"] * beta_jitter["tv_ads"] +
        test_weekly["influencer"] * beta_jitter["influencer"]
    ) * 4
```

```
        if expected_revenue > best_revenue:
            best_revenue = expected_revenue
            best_plan = test_weekly.copy()

    # Apply optimal plan
    suggested_plan = april_plan.copy()
    for channel in best_plan:
        suggested_plan[channel] = best_plan[channel]

    # Recalculate estimated revenue
    suggested_plan["estimated_revenue"] = (
        suggested_plan["google_ads"] * mean_coefs["google_ads"] +
        suggested_plan["facebook_ads"] * mean_coefs["facebook_ads"] +
        suggested_plan["tv_ads"] * mean_coefs["tv_ads"] +
        suggested_plan["influencer"] * mean_coefs["influencer"]
    )

    suggested_revenue_april = suggested_plan["estimated_revenue"].sum()
    suggested_revenue_april
```

```
np.float64(10434.35742427961)
```

```
    # Build a summary table with channel-specific impact
    channels = ["google_ads", "facebook_ads", "tv_ads", "influencer"]
    channel_names = ["Google Ads", "Facebook Ads", "TV Ads", "Influencer Marketing"]

    summary_df = pd.DataFrame({
        "Channel": channel_names,
        "Current_Spend_($)": april_plan[channels].sum().values,
        "Suggested_Spend_($)": suggested_plan[channels].sum().values
    })

    # Calculate % change in spend
    summary_df["Change_(%)"] = (
        (summary_df["Suggested_Spend_($)"] - summary_df["Current_Spend_($)"]) /
        summary_df["Current_Spend_($)"] * 100
    ).round(1)

    # Estimate revenue per channel from suggested changes
    expected_revenue_impact = []
    for channel in channels:
        spend_change = summary_df.loc[summary_df["Channel"] == channel_names[channels.index(channel)], "Suggested_Spend_($)"].values[0] - \
                       summary_df.loc[summary_df["Channel"] == channel_names[channels.index(channel)], "Current_Spend_($)"].values[0]
        impact = spend_change * mean_coefs[channel]
        expected_revenue_impact.append(round(impact, 2))

    summary_df["Expected_Revenue_Impact_($)"] = expected_revenue_impact

    summary_df.round(2)
```

	Channel	Current_Spend_($)	Suggested_Spend_($)	Change_(%)	Expected_Revenue_Impact_($)
0	Google Ads	6175.26	8645.36	40.0	1624.70
1	Facebook Ads	4567.24	6028.76	32.0	601.91
2	TV Ads	4601.45	6257.97	36.0	1027.64
3	Influencer Marketing	4477.35	2731.19	-39.0	1033.95

Select Key Channels: A list of media channels (channels) is defined along with human-readable names (channel_names). These are the focus of the spend optimization analysis.

Build Initial Summary Table: A DataFrame is created with:

Current media spend pulled from april_plan, representing the baseline budget.

Suggested media spend from suggested_plan, which could come from optimization logic (e.g., maximizing ROI using posterior estimates).

Calculate % Change in Spend: The percent change is calculated for each channel to quantify how much more or less budget is being recommended. This helps stakeholders understand how much reallocation is being proposed.

Estimate Revenue Impact by Channel: For each channel, the difference between suggested and current spend is multiplied by the mean coefficient (mean_coefs[channel]) from the Bayesian MMM model.

This shows the expected marginal return on the additional or reduced investment, assuming linear impact based on model estimates.

The result is rounded and added as a new column, giving a dollar-value estimate of how each change in spend would affect revenue.

```
from scipy.stats import norm

# Key mapping from display name to model variable
channel_key_map = {
    "Google Ads": "google_ads",
    "Facebook Ads": "facebook_ads",
    "TV Ads": "tv_ads",
    "Influencer Marketing": "influencer"
}

# Confidence calculation
def calc_confidence(mean, std):
    if std == 0:
        return 100.0
    z_score = abs(mean / std)
    confidence = (1 - 2 * norm.cdf(-z_score)) * 100
    return round(confidence, 1)

# Initialize output fields
confidence_vals = []
logic_vals = []
explanation_vals = []

# Helper dicts
mean_lookup = media_summary["Mean Coefficient"].to_dict()
conf_lookup = media_summary["Std Dev"].to_dict()

# Match columns
current_col = [col for col in summary_df.columns if "Current" in col][0]
suggested_col = [col for col in summary_df.columns if "Suggested" in col][0]
impact_col = [col for col in summary_df.columns if "Expected_Revenue_Impact" in col][0]

# Build output
for display_name in summary_df["Channel"]:
    key = channel_key_map[display_name]
    mean = mean_lookup[key]
    std = conf_lookup[key]
    conf = calc_confidence(mean, std)

    logic = f"β = {mean:.3f} ± {std:.3f}"

    current_spend = summary_df.loc[summary_df["Channel"] == display_name, current_col].values[0]
    suggested_spend = summary_df.loc[summary_df["Channel"] == display_name, suggested_col].values[0]
    delta = suggested_spend - current_spend
    delta_abs = abs(delta)

    impact = summary_df.loc[summary_df["Channel"] == display_name, impact_col].values[0]

    if mean > 0:
        explanation = (
            f"The model sees room to grow — with {conf}% confidence, it recommends investing "
            f"${delta_abs:,.0f} more, expecting ${impact:,.0f} in net gain — "
            f"a return of ${mean:.2f} per $1 spent."
        )
    elif mean < 0:
        explanation = (
        f"This spend is losing ${abs(mean):.2f} per $1 — with {conf}% confidence, "
        f"the model recommends reallocating ${delta_abs:,.0f} to better-performing channels."
        )
    else:
        explanation = f"Effect uncertain — model makes no recommendation."

    confidence_vals.append(conf)
    logic_vals.append(logic)
    explanation_vals.append(explanation)

# Drop any accidental UI-formatted columns
summary_df = summary_df.drop(columns=[
    "Confidence (%)", "Logic (Math)", "Explanation (English)"
], errors="ignore")
```

```
# Overwrite the intended columns with updated values
summary_df["Confidence_"] = confidence_vals
summary_df["Logic_Math"] = logic_vals
summary_df["Explanation_English"] = explanation_vals

summary_df.round(2)
```

	Channel	Current_Spend_	Suggested_Spend_	Change_	Expected_Revenue_Impact_	Confidence_	Logic_Math	Explanation_English
0	Google Ads	6175.26	8645.36	40.0	1624.70	77.0	β = 0.658 ± 0.548	The model sees room to grow — with 77.0% confi...
1	Facebook Ads	4567.24	6028.76	32.0	601.91	51.5	β = 0.412 ± 0.590	The model sees room to grow — with 51.5% confi...
2	TV Ads	4601.45	6257.97	36.0	1027.64	83.7	β = 0.620 ±	The model sees room to

```
from google.colab import auth
auth.authenticate_user()



PROJECT_ID = "Redacted"
DATASET_ID = "dSaulJamesonPortfolio"

TABLE_MAIN = "bayesianMMM_data"
TABLE_SUMMARY = "bayesianMMM_april_summary"



from google.cloud import bigquery
from google.oauth2 import service_account

from pandas_gbq import to_gbq

# Push main time series data
to_gbq(
    dataframe=df,
    destination_table=f"{DATASET_ID}.{TABLE_MAIN}",
    project_id=PROJECT_ID,
    if_exists="replace"
)
```

```
100%|██████████| 1/1 [00:00<00:00, 2024.28it/s]
```

```
# Make BigQuery-safe column names
summary_df.columns = (
    summary_df.columns
    .str.replace(" ", "_")
    .str.replace(r"[^\w_]", "", regex=True)
)



# Remove duplicate columns by keeping the first occurrence
summary_df = summary_df.loc[:, ~summary_df.columns.duplicated()]

# Push to BigQuery
from pandas_gbq import to_gbq

to_gbq(
    dataframe=summary_df,
    destination_table=f"{DATASET_ID}.{TABLE_SUMMARY}",
    project_id=PROJECT_ID,
    if_exists="replace"
)
```

```
100%|██████████| 1/1 [00:00<00:00, 4559.03it/s]
```