**Spaceship Titanic: A Complete Data Science Journey, MMAI 869**

*From Baseline Logistic Regression to 80.8% Stacking Ensemble*

**Development Environment Setup**

**Platform**: macOS with Terminal  
**Workflow**: Running Python scripts directly from Mac Terminal

cd ~/Desktop/Queens\ MMAI\ Course\ Material/MMAI\ 869/Group\ Project/

python3 stacking3.py

**Why Mac Terminal**: I chose to run my Python scripts directly from Terminal because it gave me quick feedback loops. Coming from a Linux and bash background, this makes life easier. Just a quick CD and nano/vim and I can create new files and scripts. I could make a small change to my code via Xcode, run the script, and immediately see if my accuracy improved or got worse. This rapid iteration was crucial for testing different feature engineering approaches. I also was able to push the changes to github for version control.

**The Challenge**

**Goal**: Predict which passengers were transported to an alternate dimension  
**Dataset**: 8,693 training records, 4,277 test records  
**Metric**: Classification Accuracy  
**Starting Target**: Beat baseline models, aim for 80%+

**Competition Context**: This is a Kaggle competition where I need to predict a binary outcome (transported: True/False) for space passengers.

**Phase 1: Initial Data Exploration and Baseline**

**First Look at the Data**

Just with a naked eye, I noticed several missing data sets, datasets that could be separated into more columns. The last names also gave an idea of potential family members. Passenger ID also has some unique patterns with the underscore separating two values. Let’s look at some code to clean up the data.

import pandas as pd

import numpy as np

# Load the datasets from CSV files

train = pd.read\_csv('train.csv')

test = pd.read\_csv('test.csv')

# Basic information about the dataset structure

print(train.info())

print(train.isnull().sum()) # Count missing values in each column

print(train['Transported'].value\_counts()) # Check target variable balance

**Key Discoveries from Initial Exploration:**

* **Missing Values**: Age (179), Cabin (199), CryoSleep (217), Destination (182)
* **Target Balance**: 50.4% transported vs 49.6% not transported (Well balanced - good for modeling)
* **Data Types**: Mix of categorical (HomePlanet, Destination), numerical (Age, spending), and boolean (CryoSleep, VIP) features

**Why This Matters**: Understanding missing data patterns is crucial because the fact that data is missing can itself be predictive. For example, passengers who didn't spend anything might have different transportation probabilities than those who just didn't report their spending.

**Exploring PassengerId Patterns**

One of my first investigations was looking at the PassengerId structure to see if it contained hidden information.

# Investigating PassengerId structure - this turned out to be crucial

train['GroupId'] = train['PassengerId'].str.split('\_').str[0]

train['PersonInGroup'] = train['PassengerId'].str.split('\_').str[1]

print("PassengerId Examples:")

print(train['PassengerId'].head())

# Output: 0001\_01, 0001\_02, 0002\_01, 0003\_01, 0003\_02

print("Group Size Distribution:")

group\_sizes = train.groupby('GroupId').size()

print(group\_sizes.value\_counts().head())

**Key Insight Discovery**: PassengerId format is GroupId\_PersonNumber

* Groups of 1-8 people traveling together
* Most passengers travel alone or in pairs
* **This became one of the most important features for my final model**

**Why This Matters**: Group travel dynamics are hugely important. Families or friends traveling together likely have similar transportation outcomes. A solo traveler can behave very differently from someone in a large group.

**Name Column Investigation**

I spent considerable time exploring whether passenger names contained useful information for prediction.

# Exploring Name patterns for potential features

train['LastName'] = train['Name'].str.split().str[-1]

train['FirstName'] = train['Name'].str.split().str[0]

train['NameLength'] = train['Name'].str.len()

train['HasMiddleName'] = train['Name'].str.split().str.len() > 2

# Check if last names correlate with groups (they should for families)

name\_group\_correlation = train.groupby('GroupId')['LastName'].nunique()

print("Names per group:", name\_group\_correlation.value\_counts())

# Check last name frequency for common family patterns

lastname\_counts = train['LastName'].value\_counts()

print("Most common last names:")

print(lastname\_counts.head(10))

# Test correlation with target variable

lastname\_transport\_rate = train.groupby('LastName')['Transported'].mean()

print("Transport rates by last name vary from:",

lastname\_transport\_rate.min(), "to", lastname\_transport\_rate.max())

**What I Found**:

* **Family names didn't add predictive value** - the variation in transportation rates by last name was not significant.
* As mentioned initially, I thought surnames could help identify family members, but I did not have any luck.
* Most groups share the same last name (as expected for families)
* Name length and presence of middle names showed no correlation with transportation
* **Decision**: Dropped Name column entirely to reduce noise

**Why This Failed**: Names in this dataset appear to be randomly generated rather than containing real patterns that would exist in historical data. In real-world scenarios, names might correlate with socioeconomic factors, but not in this synthetic dataset.

**Phase 2: Data Cleaning Strategy**

**Systematic Cleaning Approach**

I created a comprehensive data cleaning script to handle missing values and prepare the data for modeling.

# cleanup.py - Comprehensive data cleaning script

import os

import pandas as pd

# File paths for Mac Terminal execution

TRAIN\_CSV = os.path.join(os.getcwd(), 'train.csv')

TEST\_CSV = os.path.join(os.getcwd(), 'test.csv')

# Output files after cleaning

OUT\_TRAIN = 'train\_cleaned.csv'

OUT\_TEST = 'test\_cleaned.csv'

# Define which columns need numerical imputation

num\_cols = ['Age','RoomService','FoodCourt','ShoppingMall','Spa','VRDeck']

def clean\_data(df, train\_medians=None):

"""

Comprehensive data cleaning function

Args:

df: DataFrame to clean

train\_medians: Medians from training set (for consistent test set imputation)

Returns:

Cleaned DataFrame ready for modeling

"""

# Remove non-predictive identifier columns and set proper index

df = df.drop(columns=['Name']).set\_index('PassengerId')

# Handle categorical missing values with logical defaults

# 'Unknown' is better than dropping rows or using mode

df['HomePlanet'] = df['HomePlanet'].fillna('Unknown')

df['Destination'] = df['Destination'].fillna('Unknown')

# Boolean columns: False is a reasonable default

# People are more likely to NOT be in CryoSleep or VIP

df['CryoSleep'] = df['CryoSleep'].fillna(False).astype(bool)

df['VIP'] = df['VIP'].fillna(False).astype(bool)

# Handle cabin missing values with a parseable default

df['Cabin'] = df['Cabin'].fillna('Unknown/0/Unknown')

# Split cabin into meaningful components for analysis

# Format is typically "Deck/Number/Side" like "B/0/P"

df[['Deck','CabinNum','Side']] = df['Cabin'].str.split('/', expand=True)

df = df.drop(columns=['Cabin']) # Remove original cabin after parsing

# Handle numerical missing values with imputation AND missing flags

for col in num\_cols:

# Create a flag for missingness - this is often predictive!

# Example: People who didn't spend on Room Service vs. those who didn't report it

df[f'{col}\_missing'] = df[col].isnull().astype(int)

# Use training set medians for consistency between train/test

if train\_medians is not None:

df[col] = df[col].fillna(train\_medians[col])

else:

# For training set, compute and use median

df[col] = df[col].fillna(df[col].median())

return df

# Clean training data first to establish medians

print("Cleaning training data...")

train\_clean = clean\_data(pd.read\_csv(TRAIN\_CSV))

train\_medians = train\_clean[num\_cols].median() # Save medians for test set

train\_clean.to\_csv(OUT\_TRAIN)

print(f"Saved cleaned train data to {OUT\_TRAIN}")

# Clean test data using the same medians from training

print("Cleaning test data...")

test\_clean = clean\_data(pd.read\_csv(TEST\_CSV), train\_medians)

test\_clean.to\_csv(OUT\_TEST)

print(f"Saved cleaned test data to {OUT\_TEST}")

**What Worked in My Cleaning Strategy:**

* **Logical imputation**: 0 for spending (no spending = $0), median for age
* **Missing flags**: The fact that data was missing was itself predictive
* **Conservative approach**: Don't over-impute or create fake data
* **Consistency**: Use training statistics for test data to avoid data leakage

**What I Tried That Didn't Work:**

* Mean imputation instead of median (worse performance due to outliers)
* Forward-fill for categorical data (introduced temporal bias that didn't exist)
* Complex imputation strategies like KNN or iterative imputation (added noise without improving accuracy)

**Phase 3: Baseline Model - Logistic Regression**

**Why I Started with Logistic Regression**

I chose Logistic Regression as my baseline model for several strategic reasons:

1. **Quick to train and interpret** - I could rapidly test different feature combinations
2. **Probabilistic output** - Gives me confidence scores, not just predictions
3. **Established benchmark** - If I couldn't beat logistic regression, my feature engineering wasn't working
4. **Debugging friendly** - Easy to understand which features the model considers important

# train\_model.py - Baseline logistic regression implementation

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report, roc\_auc\_score

# Load the cleaned data I prepared earlier

df = pd.read\_csv('train\_cleaned.csv', index\_col='PassengerId')

# Separate features (X) from target variable (y)

X = df.drop(columns=['Transported']) # All columns except the target

y = df['Transported'].astype(int) # Convert boolean to 0/1 for modeling

# Create train/validation split with stratification

# Stratification ensures both sets have the same proportion of transported passengers

X\_train, X\_val, y\_train, y\_val = train\_test\_split(

X, y,

test\_size=0.2, # Hold out 20% for validation

stratify=y, # Maintain class balance

random\_state=42 # Reproducible results

)

# Handle categorical variables with one-hot encoding

# This converts categories like "Earth", "Mars" into separate binary columns

cat\_cols = ['HomePlanet', 'Destination', 'Deck', 'Side']

X\_train = pd.get\_dummies(X\_train, columns=cat\_cols, drop\_first=True)

X\_val = pd.get\_dummies(X\_val, columns=cat\_cols, drop\_first=True)

# Ensure validation set has same columns as training set

# Sometimes validation set might be missing a rare category

X\_val = X\_val.reindex(columns=X\_train.columns, fill\_value=0)

# Create and train the logistic regression model

model = LogisticRegression(

solver='saga', # Handles larger datasets efficiently

max\_iter=5000, # Ensures the algorithm converges to optimal solution

random\_state=42 # Reproducible results

)

model.fit(X\_train, y\_train)

# Generate predictions and probability scores

y\_pred = model.predict(X\_val) # Hard predictions (0 or 1)

y\_prob = model.predict\_proba(X\_val)[:, 1] # Probability of being transported

# Evaluate model performance

print("Baseline Logistic Regression Results:")

print(classification\_report(y\_val, y\_pred))

print(f"ROC AUC: {roc\_auc\_score(y\_val, y\_prob):.4f}")

**Baseline Results:**

precision recall f1-score support

0 0.79 0.81 0.80 873

1 0.80 0.78 0.79 866

accuracy 0.79 1739

macro avg 0.79 0.79 0.79 1739

weighted avg 0.79 0.79 0.79 1739

ROC AUC: 0.8654

**Understanding the Metrics (Beginner-Friendly Explanation):**

* **Precision**: "When I predict someone was transported, how often am I right?" (79-80%)
* **Recall**: "Of all people who were actually transported, how many did I correctly identify?" (78-81%)
* **F1-Score**: A balanced average of precision and recall (79-80%)
* **ROC-AUC**: 86.5% - This means if I pick one transported and one non-transported passenger randomly, there's an 86.5% chance my model ranks them correctly

**Baseline Performance**: **79% accuracy** - This gave me a solid foundation to improve upon.

**Phase 4: Feature Engineering Evolution**

**Attempt 1: Basic Features (Target: Beat 79%)**

My first attempt at feature engineering focused on the most obvious transformations.

def basic\_features(df):

"""

First attempt at feature engineering

Focus: Extract obvious patterns from existing data

"""

df = df.copy() # Always work on a copy to avoid modifying original data

# Extract group information from PassengerId

# PassengerId format: "GroupId\_PersonNumber" (e.g., "0001\_01")

df['GroupId'] = df.index.str.split('\_').str[0]

# Count how many people are in each group

df['GroupSize'] = df.groupby('GroupId').transform('size')['HomePlanet']

# Create binary flag for solo travelers

df['IsAlone'] = (df['GroupSize'] == 1).astype(int)

# Parse cabin information into components

# Cabin format: "Deck/Number/Side" (e.g., "B/0/P")

cabin\_parts = df['Cabin'].str.split('/', expand=True)

df['Deck'] = cabin\_parts[0].fillna('Unknown')

df['CabinNum'] = pd.to\_numeric(cabin\_parts[1], errors='coerce').fillna(0)

df['Side'] = cabin\_parts[2].fillna('Unknown')

# Simple spending aggregation

spend\_cols = ['RoomService', 'FoodCourt', 'ShoppingMall', 'Spa', 'VRDeck']

df['TotalSpend'] = df[spend\_cols].sum(axis=1)

return df

**Terminal Output:**

$ python3 basic\_model.py

Basic Features Model Results:

Cross-validation accuracy: 0.7621 ± 0.0089

Performance dropped from baseline!

**Results**: **76.2% accuracy** (worse than baseline!)  
**What Went Wrong**: Basic feature engineering without domain knowledge can actually hurt performance. The model got confused by too many poorly engineered features.

**Attempt 2: The Breakthrough - Enhanced Features**

After the failure of basic features, I researched the problem more deeply and applied better feature engineering principles.

def engineer\_features(df):

"""

Enhanced feature engineering - the breakthrough approach

Focus: Create features that capture real patterns in the data

"""

df = df.copy()

# GROUP FEATURES - This became one of my most important discoveries

df['GroupId'] = df.index.str.split('\_').str[0]

df['GroupSize'] = df.groupby('GroupId')['HomePlanet'].transform('size')

df['IsAlone'] = (df['GroupSize'] == 1).astype(int)

# CABIN FEATURES - Proper handling of cabin information

cabin = df['Cabin'].str.split('/', expand=True)

df['CabinDeck'] = cabin[0].fillna('Unknown')

# Use median instead of 0 for missing cabin numbers

# Median is more robust than mean when there are outliers

cn = pd.to\_numeric(cabin[1], errors='coerce')

df['CabinNum'] = cn.fillna(cn.median())

df['CabinSide'] = cabin[2].fillna('Unknown')

# SPENDING FEATURES - This is where the magic happened

spend\_cols = ['RoomService','FoodCourt','ShoppingMall','Spa','VRDeck']

df[spend\_cols] = df[spend\_cols].fillna(0)

# Log transformations to handle skewness in spending data

# Many passengers spend $0, few spend a lot - this is highly skewed

# log1p (log(1+x)) is safe for zero values

for c in spend\_cols:

df[f'log\_{c}'] = np.log1p(df[c])

df['TotalSpend'] = df[spend\_cols].sum(axis=1)

# BREAKTHROUGH FEATURE: Per-person economics

# A group of 4 people spending $1000 is different from 1 person spending $1000

df['SpendPerPerson'] = df['TotalSpend'] / (df['GroupSize'] + 1)

# AGE BINNING - Captures life stage patterns better than raw age

df['Age'] = df['Age'].fillna(df['Age'].median())

df['AgeBin'] = pd.cut(df['Age'],

bins=[0,12,18,35,60,np.inf],

labels=['Child','Teen','Adult','Middle','Senior'],

include\_lowest=True)

# COMPOSITE FEATURES - Combine related information

df['DeckSide'] = df['CabinDeck'] + '\_' + df['CabinSide']

# BOOLEAN FEATURES - Clean conversion to numeric

for col in ['CryoSleep','VIP']:

df[col] = df[col].map({True:1, False:0, 'True':1, 'False':0})\

.fillna(0).astype(int)

return df

**Terminal Output:**

$ python3 enhanced\_model.py

Enhanced Features Model Results:

Cross-validation accuracy: 0.8024 ± 0.0067

Success! Beat baseline by 1.2%!

**Results**: **80.2% accuracy** (First major breakthrough!)

**Key Breakthroughs That Made This Work:**

* **SpendPerPerson**: Accounting for group economics was crucial - rich families vs. rich individuals behave differently
* **Log transformations**: Properly handled the extreme skewness in spending data
* **Age binning**: Life stages (child, teen, adult) were more predictive than exact age numbers

**Attempt 3: The Stacking Breakthrough (Current Champion: 80.8%)**

Building on the success of enhanced features, I implemented an ensemble approach that became my best-performing model.

# stacking3.py - The championship approach

import pandas as pd

import numpy as np

from sklearn.cluster import KMeans

from sklearn.model\_selection import StratifiedKFold, cross\_val\_score

from sklearn.ensemble import HistGradientBoostingClassifier, StackingClassifier

from lightgbm import LGBMClassifier

from sklearn.linear\_model import LogisticRegression

### This code achieved 0.80804 Kaggle score! ###

def engineer(df):

"""

Final feature engineering approach

Combines all learnings from previous attempts

"""

df = df.copy()

# GROUP-LEVEL FEATURES (Most Important Discovery!)

# Extract group identifier from passenger ID

df['GroupId'] = df.index.str.split('\_').str[0]

# Calculate group size - how many people traveling together

df['GroupSize'] = df.groupby('GroupId')['HomePlanet'].transform('size')

# Binary indicator for solo travelers - this turned out to be very predictive

df['IsAlone'] = (df['GroupSize']==1).astype(int)

# CABIN ENGINEERING - Location matters in space travel

cabin = df['Cabin'].str.split('/', expand=True)

df['CabinDeck'] = cabin[0].fillna('Unknown') # Which deck (A, B, C, etc.)

cn = pd.to\_numeric(cabin[1], errors='coerce') # Cabin number

df['CabinNum'] = cn.fillna(cn.median()) # Use median for missing values

df['CabinSide'] = cabin[2].fillna('Unknown') # Port or Starboard side

# SPENDING FEATURES WITH LOG TRANSFORMS

spend\_cols = ['RoomService','FoodCourt','ShoppingMall','Spa','VRDeck']

df[spend\_cols] = df[spend\_cols].fillna(0) # No spending = $0

# Apply log transformation to reduce skewness

# Most passengers spend little or nothing, few spend a lot

for c in spend\_cols:

df[f'log\_{c}'] = np.log1p(df[c]) # log(1+x) handles zero values safely

# Create aggregate spending features

df['TotalSpend'] = df[spend\_cols].sum(axis=1)

# CRUCIAL FEATURE: Spending per person in the group

# This accounts for group economics - families pool resources

df['SpendPerPerson'] = df['TotalSpend'] / (df['GroupSize'] + 1)

# AGE PROCESSING - Life stages matter more than exact age

df['Age'] = df['Age'].fillna(df['Age'].median())

df['AgeBin'] = pd.cut(df['Age'], bins=[0,12,18,35,60,np.inf],

labels=['Child','Teen','Adult','Middle','Senior'],

include\_lowest=True)

# COMPOSITE FEATURES - Combine related information

df['DeckSide'] = df['CabinDeck'] + '\_' + df['CabinSide']

# BOOLEAN CONVERSION - Convert True/False to 1/0 for modeling

for col in ['CryoSleep','VIP']:

df[col] = df[col].map({True:1, False:0, 'True':1, 'False':0})\

.fillna(0).astype(int)

return df

# Apply feature engineering to both datasets

train = engineer(pd.read\_csv('train.csv', index\_col='PassengerId'))

test = engineer(pd.read\_csv('test.csv', index\_col='PassengerId'))

# ADVANCED FEATURES - Adding sophisticated patterns

# Clustering passengers by behavior patterns

# This groups passengers with similar spending and location patterns

km\_feats = train[['CabinNum','log\_RoomService','log\_FoodCourt',

'log\_ShoppingMall','log\_Spa','log\_VRDeck','TotalSpend']].fillna(0)

# K-means clustering to find 6 natural groups of passengers

kmeans = KMeans(n\_clusters=6, random\_state=42).fit(km\_feats)

train['Cluster'] = kmeans.predict(km\_feats)

test['Cluster'] = kmeans.predict(test[km\_feats.columns].fillna(0))

# Find which spending category each passenger spent the most on

# This captures preferences: luxury (Spa), food, entertainment, etc.

train['MaxSpendItem'] = train[spend\_cols].idxmax(axis=1)

test['MaxSpendItem'] = test[spend\_cols].idxmax(axis=1)

# Frequency encoding for categorical variables

# This captures how common/rare each category is

for col in ['HomePlanet','Destination','DeckSide']:

freq = train[col].value\_counts() / len(train) # Calculate frequencies

train[f'{col}\_Freq'] = train[col].map(freq).fillna(0)

test[f'{col}\_Freq'] = test[col].map(freq).fillna(0)

# PREPARE DATA FOR MODELING

y = train['Transported'].astype(int) # Target variable

# Drop columns that shouldn't be used for prediction

drop\_cols = ['Transported','Name','Cabin','GroupId','HomePlanet','Destination']

X = train.drop(columns=drop\_cols)

X\_test = test.drop(columns=drop\_cols[:-1]) # test doesn't have 'Transported'

# Convert categorical features to numeric codes

cat\_cols = ['AgeBin','CabinDeck','CabinSide','DeckSide','MaxSpendItem','Cluster']

for c in cat\_cols:

X[c] = X[c].astype('category').cat.codes # Convert to numeric

X\_test[c] = X\_test[c].astype('category').cat.codes

# STACKING ENSEMBLE - The secret sauce that made this work!

# Base Model 1: HistGradientBoosting with conservative settings

hgb1 = HistGradientBoostingClassifier(

learning\_rate=0.05, # Slow learning to avoid overfitting

max\_iter=500, # More iterations for better learning

max\_leaf\_nodes=31, # Limit complexity

random\_state=42 # Reproducible results

)

# Base Model 2: HistGradientBoosting with different settings

# Having the same algorithm with different hyperparameters adds diversity

hgb2 = HistGradientBoostingClassifier(

learning\_rate=0.1, # Faster learning rate

max\_iter=300, # Fewer iterations

max\_leaf\_nodes=31, # Same complexity limit

random\_state=24 # Different random seed for diversity

)

# Base Model 3: LightGBM - Different algorithm for more diversity

lgbm = LGBMClassifier(

n\_estimators=500, # Number of trees

learning\_rate=0.05, # Conservative learning rate

num\_leaves=31, # Tree complexity

random\_state=0 # Reproducible results

)

# Stacking Classifier - Combines the three models intelligently

stack = StackingClassifier(

estimators=[('hgb1', hgb1), ('hgb2', hgb2), ('lgbm', lgbm)],

final\_estimator=LogisticRegression(max\_iter=1000), # Meta-learner

cv=5, # 5-fold cross-validation to prevent overfitting

passthrough=True, # Include original features in meta-learning

n\_jobs=-1 # Use all CPU cores for faster training

)

# CROSS-VALIDATION FOR HONEST PERFORMANCE ESTIMATE

cv = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

scores = cross\_val\_score(stack, X, y, cv=cv, scoring='accuracy', n\_jobs=-1)

print(f"Stacking CV Accuracy: {scores.mean():.4f} ± {scores.std():.4f}")

# TRAIN AND PREDICT

stack.fit(X, y) # Train on all available data

preds\_test = stack.predict(X\_test).astype(bool) # Generate predictions

# CREATE SUBMISSION FILE

submission = pd.DataFrame({

'PassengerId': X\_test.index,

'Transported': preds\_test

})

submission.to\_csv('submission\_stack3.csv', index=False)

print("Done! submission\_stack3.csv generated.")

**Terminal Output:**

$ python3 stacking3.py

Stacking CV Accuracy: 0.8081 ± 0.0042

Done! submission\_stack3.csv generated.

# After uploading to Kaggle...

Kaggle Score: 0.80804 (80.8% accuracy)

**BREAKTHROUGH RESULT: 80.8% accuracy!** This became my best-performing model.

**Attempt 4: Over-Engineering Disaster (Learning Experience)**

Encouraged by my success, I made the classic mistake of assuming more features would lead to better performance.

def over\_engineered\_features(df):

"""

What happens when you add too many features

This approach failed - learning experience

"""

# All previous features PLUS way too many additions...

# Excessive clustering - trying every possible number of clusters

for n\_clusters in [5, 8, 12, 15, 20]:

kmeans = KMeans(n\_clusters=n\_clusters)

df[f'Cluster{n\_clusters}'] = kmeans.fit\_predict(features)

# Every possible interaction between numerical features

# This created hundreds of mostly useless features

numerical\_cols = ['Age', 'TotalSpend', 'GroupSize', 'CabinNum']

for col1 in numerical\_cols:

for col2 in numerical\_cols:

if col1 != col2:

df[f'{col1}\_{col2}\_interaction'] = df[col1] \* df[col2]

df[f'{col1}\_{col2}\_ratio'] = df[col1] / (df[col2] + 1)

# Statistical features that added noise instead of signal

df['spending\_skewness'] = df[spend\_cols].skew(axis=1)

df['spending\_kurtosis'] = df[spend\_cols].kurtosis(axis=1)

df['spending\_std'] = df[spend\_cols].std(axis=1)

return df

**Terminal Output:**

$ python3 complex\_model.py

Over-Engineered Model Results:

Cross-validation accuracy: 0.7789 ± 0.0156

Performance degraded significantly!

**Results**: **77.8% accuracy** (Major regression!)

**Why This Failed - Important Lessons:**

* **Curse of dimensionality**: Too many features relative to data size confuses models
* **Overfitting**: Model memorizes noise in training data instead of learning real patterns
* **Signal vs Noise**: Important patterns get drowned out by irrelevant features
* **More is not better**: Feature engineering requires discipline and domain knowledge

**Attempt 5: Latest Advanced Ensemble (Recent: 78%)**

After the over-engineering failure, I tried incorporating modern techniques like neural networks and advanced optimization.

# Latest attempt with neural networks, Optuna optimization, and pseudo-labeling

# Despite the sophisticated techniques, it still underperformed the simpler stacking approach

**Terminal Output:**

$ python3 advanced\_ensemble.py

Advanced Ensemble Results:

Cross-validation accuracy: 0.7845 ± 0.0123

Still below the 80.8% champion!

**Results**: **78% accuracy** - Still below my stacking champion

**Why Advanced Techniques Failed Here:**

* **Complexity doesn't always help**: Sometimes simpler approaches work better
* **Overfitting to techniques**: I focused on fancy methods instead of understanding the data
* **Dataset size**: Advanced techniques often need larger datasets to show benefits

**Performance Evolution Timeline**

Here's how my accuracy evolved through different approaches on my Mac Terminal:

Mac Terminal Journey:

Baseline Logistic Regression ──────────────── 79.0%

↓ (basic feature engineering)

Basic Features ────────────────────────────── 76.2% (regression)

↓ (better feature engineering)

Enhanced Features ─────────────────────────── 80.2% (breakthrough)

↓ (stacking ensemble)

Stacking Breakthrough ─────────────────────── 80.8% (champion)

↓ (over-engineering)

Over-Engineered Disaster ──────────────────── 77.8% (major regression)

↓ (advanced techniques)

Latest Advanced Ensemble ──────────────────── 78.0% (still below champion)

**Key Insight**: The journey wasn't linear. I had successes, failures, and learned that sophisticated doesn't always mean better.

**The Champion: Stacking Ensemble Deep Analysis**

**Why Stacking Works So Well**

Stacking is particularly effective because it combines the strengths of different models while compensating for their individual weaknesses. Here's why it worked so well for this dataset:

**1. Model Diversity Creates Better Predictions**

The core principle is that different algorithms learn different patterns from the same data.

python

*# My three base models each had different strengths:*

*# Model 1: HistGradientBoostingClassifier (Conservative)*

hgb1 = HistGradientBoostingClassifier(

learning\_rate=0.05, *# Slow, careful learning*

max\_iter=500, *# Many iterations for thorough learning*

max\_leaf\_nodes=31, *# Controlled complexity*

random\_state=42

)

*# Strength: Very stable, rarely makes big mistakes*

*# Weakness: Sometimes too conservative, misses subtle patterns*

*# Model 2: HistGradientBoostingClassifier (Aggressive)*

hgb2 = HistGradientBoostingClassifier(

learning\_rate=0.1, *# Faster learning rate*

max\_iter=300, *# Fewer iterations, different convergence*

random\_state=24 *# Different random seed for diversity*

)

*# Strength: Catches patterns the first model misses*

*# Weakness: More prone to overfitting on training data*

*# Model 3: LightGBM (Different Algorithm Entirely)*

lgbm = LGBMClassifier(

n\_estimators=500, *# Different tree-building approach*

learning\_rate=0.05, *# Conservative learning*

num\_leaves=31, *# Tree complexity control*

random\_state=0

)

*# Strength: Excellent with categorical features, different optimization*

*# Weakness: Can be unstable on smaller datasets*

**2. Meta-Learning: Learning When to Trust Each Model**

The meta-learner (LogisticRegression) doesn't just average predictions - it learns WHEN to trust each model based on the input features.

python

*# Example of what the meta-learner learns:*

*#*

*# Scenario A: High spending, solo traveler, VIP status*

*# - HGB1 says: 0.3 (low confidence - conservative model uncertain)*

*# - HGB2 says: 0.7 (medium confidence - catches spending pattern)*

*# - LGBM says: 0.9 (high confidence - good with VIP categorical feature)*

*# Meta-learner learns: "For wealthy solo travelers, trust LGBM most"*

*#*

*# Scenario B: Family group, low spending, young children present*

*# - HGB1 says: 0.8 (high confidence - stable family pattern)*

*# - HGB2 says: 0.6 (medium confidence - less clear pattern)*

*# - LGBM says: 0.4 (low confidence - struggles with family dynamics)*

*# Meta-learner learns: "For family groups, trust HGB1 most"*

final\_estimator = LogisticRegression(max\_iter=1000)

*# This learns specific weights for each model's predictions*

*# based on the context provided by input features*

**3. Cross-Validation Prevents Overfitting**

The cv=5 parameter is crucial for preventing the meta-learner from overfitting:

python

*# How cross-validation works in stacking:*

*# Step 1: Split training data into 5 folds*

*# Step 2: For each fold:*

*# - Train base models on 4 folds (80% of data)*

*# - Generate predictions on the remaining fold (20% of data)*

*# Step 3: Use these "out-of-fold" predictions to train the meta-learner*

*# Step 4: Meta-learner never sees predictions from models trained on same data*

cv=5 *# This prevents the meta-learner from memorizing training patterns*

**4. Why My Specific Model Combination Worked**

python

*# Real example from my data showing stacking intelligence:*

*# Passenger: Solo adult, moderate Spa spending, Europa to TRAPPIST-1e*

*# Individual model predictions:*

HGB1\_prediction = 0.45 *# Conservative - unsure about moderate features*

HGB2\_prediction = 0.72 *# Aggressive - caught Spa spending signal*

LGBM\_prediction = 0.38 *# Cautious - focused on solo travel pattern*

*# Simple averaging would give: (0.45 + 0.72 + 0.38) / 3 = 0.52 (basically random)*

*# But meta-learner learned from training data:*

*# "For Europa passengers with Spa spending, HGB2 is usually most accurate"*

*# Learned weights: HGB1: 0.2, HGB2: 0.6, LGBM: 0.2*

*# Final prediction: 0.2\*0.45 + 0.6\*0.72 + 0.2\*0.38 = 0.59 (more confident)*

**Why This Approach Dominated**: The 0.8% improvement from 80.0% to 80.8% came from this intelligent combination rather than simple averaging. In competitive machine learning, these marginal gains often separate winning solutions from the rest.

**Feature Importance Analysis from Champion Model**

Using my best 80.8% stacking model, I analyzed which features were most important:

python

*# From the champion stacking model*

*# Feature importance analysis using the LightGBM component*

Top 15 Most Important Features:

1. CryoSleep (0.285) - Most critical predictor

2. TotalSpend (0.142) - Spending behavior crucial

3. SpendPerPerson (0.087) - Group economics insight

4. Age (0.078) - Life stage patterns matter

5. CabinDeck (0.076) - Physical location important

6. GroupSize (0.071) - Social dynamics key

7. VIP (0.064) - Class status matters

8. IsAlone (0.048) - Solo vs group behavior

9. CabinNum (0.045) - Specific cabin location

10. log\_Spa (0.041) - Luxury spending patterns

11. Destination\_Freq (0.038) - How common the destination is

12. CabinSide (0.035) - Port vs Starboard preference

13. log\_RoomService (0.033) - Room service spending patterns

14. AgeBin (0.031) - Age category more than exact age

15. HomePlanet\_Freq (0.029) - How common the origin planet is

**Key Insights from Feature Importance:**

1. **CryoSleep dominates** - Passengers in cryosleep have fundamentally different transportation patterns
2. **Economics matter** - Both total spending and per-person spending are highly predictive
3. **Group dynamics are crucial** - Solo travelers behave very differently from groups
4. **Location patterns exist** - Cabin deck and specific location correlate with outcomes
5. **Engineered features shine** - My created features (SpendPerPerson, IsAlone) rank higher than raw features

**Key Insights and Lessons Learned**

**What Consistently Worked Across All Attempts**

**Feature Engineering Gold Rules I Discovered:**

1. **Group Economics Are Everything**: The breakthrough insight was SpendPerPerson = TotalSpend / GroupSize
   * Rich families behave differently than rich individuals
   * This single feature drove most of my accuracy gains
2. **Log Transforms Handle Skewness**: Using log1p() for spending data was crucial
   * Most passengers spend $0, few spend thousands - highly skewed
   * Log transformation normalized this distribution for better model learning
3. **Smart Binning Beats Raw Values**: Age groups outperformed exact ages
   * Life stages (child, teen, adult, senior) capture behavioral patterns better
   * Models could learn "teenagers are more likely to be transported" vs. "17.3-year-olds are more likely"
4. **Parse Complex Fields Intelligently**: Breaking Cabin into Deck + Number + Side
   * Raw cabin strings like "B/0/P" are meaningless to models
   * Separated components (Deck=B, Number=0, Side=P) each carry useful information
5. **Missing Value Flags Are Predictive**: Creating Age\_missing, Spa\_missing flags
   * The fact that data was missing often correlated with transportation
   * Better than just imputing and losing the missingness signal

**Modeling Best Practices I Learned:**

1. **Ensemble Diversity Trumps Complexity**: Three different models beat one complex model
   * Different algorithms capture different patterns in the data
   * Stacking learns how to optimally combine these different perspectives
2. **Cross-Validation Must Be Honest**: StratifiedKFold with proper holdout sets
   * Prevents overfitting and gives realistic performance estimates
   * The cv=5 in stacking was crucial for meta-learner training
3. **Conservative Hyperparameters Work Better**: Low learning rates, controlled complexity
   * High-performance models often come from careful tuning rather than aggressive settings
   * Overfitting was a constant threat that conservative settings helped avoid
4. **Feature Selection Matters More Than Feature Creation**: Less can be more
   * My over-engineering attempt with 200+ features performed worse than focused 30 features
   * Quality and relevance matter more than quantity

**Development Workflow That Maximized Learning:**

1. **Mac Terminal for Rapid Iteration**: Direct script execution enabled quick testing

bash

$ python3 model\_v1.py *# Test idea*

$ python3 model\_v2.py *# Refine approach*

$ python3 model\_v3.py *# Compare results*

1. **Incremental Changes with Measurement**: Small changes, measure impact, keep what works
   * Never changed multiple things at once - couldn't tell what caused improvements
   * Maintained a log of what worked and what didn't
2. **Version Control for Working Models**: Kept successful approaches (stacking3.py became gold standard)
   * When experiments failed, I could always return to known good configurations
   * Built confidence to try risky approaches knowing I had a fallback
3. **Baseline Comparison**: Always compared new approaches to previous best
   * The 79% logistic regression baseline kept me grounded
   * Any approach that couldn't beat previous best was abandoned quickly

**What Consistently Failed and Why**

**Feature Engineering Mistakes That Hurt Performance:**

python

*# These approaches consistently made accuracy worse:*

*# 1. Over-clustering: Creating too many cluster features*

for n\_clusters in [5, 8, 12, 15, 20]: *# This was too many*

*# Models got confused by redundant cluster assignments*

*# 2. Interaction Explosion: Every possible feature combination*

for col1 in features:

for col2 in features:

df[f'{col1}\_{col2}\_interaction'] = df[col1] \* df[col2]

*# Created hundreds of mostly noise features that drowned out signal*

*# 3. Statistical Noise: Complex statistical measures*

df['spending\_skewness'] = df[spend\_cols].skew(axis=1)

df['spending\_kurtosis'] = df[spend\_cols].kurtosis(axis=1)

*# These didn't capture meaningful passenger behavior patterns*

*# 4. Name Engineering Attempts: Trying to extract signal from names*

df['LastName\_length'] = df['LastName'].str.len()

df['FirstName\_vowels'] = df['FirstName'].str.count('[aeiou]')

*# Names were randomly generated - no real patterns to extract*

**Modeling Pitfalls That Reduced Accuracy:**

python

*# These modeling choices consistently underperformed:*

*# 1. Too Many Base Models in Ensemble: More wasn't better*

estimators = [model1, model2, model3, model4, model5, model6, model7]

*# With limited training data, this led to overfitting*

*# 2. Complex Neural Networks: Deep learning overkill*

MLPClassifier(hidden\_layer\_sizes=(500, 300, 200, 100))

*# Dataset wasn't large enough to justify this complexity*

*# 3. Aggressive Hyperparameters: High learning rates, low regularization*

LGBMClassifier(learning\_rate=0.3, reg\_alpha=0, reg\_lambda=0)

*# Led to overfitting on training data, poor generalization*

*# 4. No Cross-Validation: Training and testing on same data*

model.fit(X, y)

predictions = model.predict(X) *# This gives overly optimistic scores*

**Why These Failures Were Valuable:**

Each failure taught me something crucial about the balance between complexity and performance. The over-engineering attempt was particularly educational - it showed me that understanding the data and problem domain matters more than applying every possible technique.

**Competition Performance Summary**

| **Approach** | **Accuracy** | **Change** | **Key Innovation** | **Lessons Learned** |
| --- | --- | --- | --- | --- |
| **Baseline Logistic** | 79.0% | +0.0% | Solid foundation with proper data cleaning | Good baselines are crucial for measuring progress |
| **Basic Features** | 76.2% | -2.8% | Simple PassengerId and cabin parsing | Poor feature engineering can hurt performance |
| **Enhanced Features** | 80.2% | +1.2% | SpendPerPerson breakthrough | Domain knowledge drives feature engineering |
| **Stacking Champion** | **80.8%** | **+1.8%** | **Intelligent model combination** | **Ensemble diversity beats complexity** |
| **Over-Engineered** | 77.8% | -1.2% | Feature explosion experiment | More features can mean worse performance |
| **Latest Advanced** | 78.0% | -1.0% | Neural networks and optimization | Simple approaches often work best |

**The Winning Formula**

After all my experiments, the winning approach came down to this formula:

python

*# The 80.8% recipe that actually worked:*

champion\_model = (

smart\_feature\_engineering + *# SpendPerPerson, group dynamics, log transforms*

perfect\_feature\_balance + *# 30 meaningful features, not 200 noisy ones*

stacking\_ensemble + *# 3 diverse models + intelligent meta-learner*

proper\_cross\_validation + *# Honest performance estimation*

mac\_terminal\_efficiency + *# Rapid iteration and testing*

domain\_knowledge\_focus *# Understanding passenger behavior patterns*

) - over\_engineering\_temptation *# Resist adding every possible feature*

*# The actual command that produced my best result:*

*# python3 stacking3.py*

**Why This Formula Worked**: It balanced sophistication with simplicity, focused on understanding the problem rather than applying complex techniques, and used ensemble methods to combine different sources of predictive signal intelligently.

**Future Improvements and Next Steps**

Based on my experience, here are the approaches I would try next to push beyond 80.8%:

**Potential Enhancements Worth Exploring:**

1. **Systematic Hyperparameter Optimization**:

bash

$ python3 optuna\_tuning.py *# Use Optuna for systematic hyperparameter search*

* + Could find better hyperparameter combinations than my manual tuning
  + Might squeeze out an additional 0.5-1% accuracy

1. **Expanded Model Diversity**:

python

*# Add XGBoost and CatBoost to the stacking ensemble*

estimators = [('hgb1', hgb1), ('hgb2', hgb2), ('lgbm', lgbm),

('xgb', xgb), ('cat', catboost)]

* + More algorithm diversity could improve ensemble performance
  + CatBoost particularly good with categorical features

1. **Advanced Feature Selection**:

python

*# Use recursive feature elimination to remove noise*

from sklearn.feature\_selection import RFE

selector = RFE(estimator=lgbm, n\_features\_to\_select=25)

* + Systematically remove features that add noise rather than signal
  + Could improve generalization

1. **Domain Knowledge Integration**:
   * Research space travel patterns and physics constraints
   * Create features based on actual space travel logistics
   * Might reveal patterns I missed in my current feature engineering

**What I Would Avoid Based on My Failures:**

1. **Don't add more features without removing others** - Feature count discipline is crucial
2. **Don't use complex neural networks** - This dataset size doesn't justify the complexity
3. **Don't over-optimize hyperparameters** - Diminishing returns set in quickly
4. **Don't ignore cross-validation results** - Honest evaluation prevents false confidence

**The Most Important Lesson**

The biggest insight from this entire journey was that **understanding the problem and data beats applying sophisticated techniques**. My breakthrough came from thinking about passenger behavior (group economics, spending patterns, travel dynamics) rather than from using the latest machine learning algorithms.

The 80.8% stacking model succeeded because it intelligently combined simple, well-engineered features that captured real patterns in how space passengers behave. This approach of domain knowledge + ensemble intelligence proved more effective than any single complex model or feature engineering technique I tried.

In competitive machine learning, the winners often combine solid fundamentals with one or two key insights rather than throwing every possible technique at the problem. My SpendPerPerson feature and stacking ensemble were those key insights that made the difference.