Inventory Optimization Model

Author: Priyanshu Bhardwaj
Date: February 13, 2026
GitHub: trulypriyanshu/inventory-optimization-model

Business Impact

- Reduced excess inventory
- Reduced Stockouts
- Saved inventory costs
- Improved service levels
- Automated inventory optimization

Features

1. Comprehensive data cleaning and preprocessing
2. Statistical demand forecasting
3. ABC classification (Pareto analysis)
4. Obsolescence risk detection
5. Safety stock optimization
6. Inventory performance metrics
7. Visual analytics dashboard
8. Export to Excel and Power BI

```
# Import required libraries
import os
import pandas as pd
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
import warnings
warnings.filterwarnings('ignore')

# Set up display options
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
plt.style.use('seaborn-v0_8-darkgrid')

# Create output directory
output_dir = 'outputs'
os.makedirs(output_dir, exist_ok=True)

print("□ Libraries imported successfully")
print(f"□ Output directory created: {output_dir}")
```

1. Data Loading & Cleaning

Load and clean the Online Retail dataset with proper data validation.

```
def load_and_clean_data(filepath='data/Online Retail.xlsx'):
    """
    Load and clean the Online Retail dataset with proper data
validation
    """
    print("=" * 70)
    print("▯ LOADING AND CLEANING DATA")
    print("=" * 70)

    try:
        # Load data
        print(f"Loading data from: {filepath}")
        df = pd.read_excel(filepath)

        print(f"✓ Original dataset shape: {df.shape}")
        print(f"✓ Columns: {df.columns.tolist()}")
        print(f"✓ Memory usage: {df.memory_usage(deep=True).sum() /
1024**2:.2f} MB")

        # Standardize column names
        df.columns = [col.strip().replace(' ', '_') for col in
df.columns]

        # Store original stats
        original_rows = len(df)
        original_skus = df['StockCode'].nunique()

        # Create a clean copy
        df_clean = df.copy()

        # ========== DATA CLEANING STEPS ==========

        # 1. Remove cancelled orders
        mask_cancelled =
df_clean['InvoiceNo'].astype(str).str.startswith('C')
        cancelled_count = mask_cancelled.sum()
        df_clean = df_clean[~mask_cancelled]
        print(f"  Removed {cancelled_count:,} cancelled orders")

        # 2. Remove negative quantities and prices
        mask_negative_qty = df_clean['Quantity'] <= 0
        mask_negative_price = df_clean['UnitPrice'] <= 0
        negative_count = mask_negative_qty.sum() +
mask_negative_price.sum()
        df_clean = df_clean[(df_clean['Quantity'] > 0) &
```

```
(df_clean['UnitPrice'] > 0)]
        print(f"  Removed {negative_count:,} rows with negative
quantities/prices")

        # 3. Remove missing CustomerID
        missing_customer_before = df_clean['CustomerID'].isna().sum()
        df_clean = df_clean.dropna(subset=['CustomerID'])
        print(f"  Removed {missing_customer_before:,} rows with
missing CustomerID")

        # 4. Remove extreme outliers (99th percentile)
        q99_quantity = df_clean['Quantity'].quantile(0.99)
        mask_outliers = df_clean['Quantity'] > q99_quantity
        outlier_count = mask_outliers.sum()
        df_clean = df_clean[~mask_outliers]
        print(f"  Removed {outlier_count:,} extreme quantity outliers
(> {q99_quantity:.0f} units)")

        # ========== FEATURE ENGINEERING ==========

        # Convert InvoiceDate to datetime
        df_clean['InvoiceDate'] =
pd.to_datetime(df_clean['InvoiceDate'])

        # Create essential features
        df_clean['TotalValue'] = df_clean['Quantity'] *
df_clean['UnitPrice']
        df_clean['Date'] = df_clean['InvoiceDate'].dt.date
        df_clean['YearMonth'] =
df_clean['InvoiceDate'].dt.to_period('M')
        df_clean['DayOfWeek'] = df_clean['InvoiceDate'].dt.day_name()
        df_clean['Month'] = df_clean['InvoiceDate'].dt.month
        df_clean['Year'] = df_clean['InvoiceDate'].dt.year

        # ========== FINAL STATISTICS ==========

        cleaned_rows = len(df_clean)
        cleaned_skus = df_clean['StockCode'].nunique()
        data_retention = (cleaned_rows / original_rows) * 100

        print("\n CLEANING SUMMARY:")
        print(f"  Original rows: {original_rows:,}")
        print(f"  Cleaned rows: {cleaned_rows:,}")
        print(f"  Data retention: {data_retention:.1f}%")
        print(f"  Original SKUs: {original_skus:,}")
        print(f"  Cleaned SKUs: {cleaned_skus:,}")
        print(f"  Time range: {df_clean['InvoiceDate'].min().date()}
to {df_clean['InvoiceDate'].max().date()}")
        print(f"  Total revenue: $
{df_clean['TotalValue'].sum():,.2f}")
        print(f"  Avg transaction value: $
{df_clean['TotalValue'].mean():.2f}")

        # Create summary dictionary
        summary_stats = {
            'original_rows': original_rows,
            'cleaned_rows': cleaned_rows,
            'data_retention_pct': data_retention,
            'original_skus': original_skus,
            'cleaned_skus': cleaned_skus,
            'total_revenue': df_clean['TotalValue'].sum(),
            'time_range_start': df_clean['InvoiceDate'].min(),
            'time_range_end': df_clean['InvoiceDate'].max()
        }

        return df_clean, summary_stats

    except Exception as e:
        print(f"� Error loading data: {e}")
        raise

# Execute data loading
df_clean, summary_stats = load_and_clean_data('data/Online
Retail.xlsx')
```

2. Demand Analysis

Analyze demand patterns for each SKU with statistical rigor.

```
def analyze_sku_demand(df_clean, min_days_with_sales=30):
    """
    Analyze demand patterns for each SKU with statistical rigor
    """
    print("\n" + "=" * 70)
    print("� ANALYZING SKU DEMAND PATTERNS")
    print("=" * 70)

    # Create daily demand dataset
    print("Creating daily demand aggregation...")
    daily_demand = df_clean.groupby(['StockCode', 'Date']).agg({
        'Quantity': 'sum',
        'TotalValue': 'sum',
        'UnitPrice': 'mean'
    }).reset_index()

    # Calculate comprehensive demand statistics per SKU
    print("Calculating SKU-level demand statistics...")
    sku_stats = daily_demand.groupby('StockCode').agg({
        'Quantity': ['mean', 'std', 'count', 'min', 'max', 'median'],
        'TotalValue': 'sum',
        'UnitPrice': 'mean'
    }).reset_index()

    # Flatten multi-level columns
    sku_stats.columns = [
        'StockCode',
        'AvgDailyDemand', 'DemandStd', 'DaysWithSales',
        'MinDailyDemand', 'MaxDailyDemand', 'MedianDailyDemand',
        'TotalRevenue', 'AvgUnitPrice'
    ]

    # Add product description
    product_info = (
        df_clean.groupby('StockCode')['Description']
        .agg(lambda x: x.mode().iloc[0] if not x.mode().empty else
x.iloc[0])
        .reset_index()
    )
    sku_stats = sku_stats.merge(product_info, on='StockCode',
how='left')

    # Calculate additional metrics
    sku_stats['DemandCV'] = sku_stats['DemandStd'] /
sku_stats['AvgDailyDemand']

    # Handle zero/negative demand and NaN values
    sku_stats['AvgDailyDemand'] =
sku_stats['AvgDailyDemand'].clip(lower=0.1)
    sku_stats['DemandStd'] =
sku_stats['DemandStd'].fillna(sku_stats['AvgDailyDemand'] * 0.3)
    sku_stats['DemandCV'] =
sku_stats['DemandCV'].fillna(0.5).clip(upper=3.0)

    # Calculate demand patterns
    sku_stats['DemandStability'] = np.where(
        sku_stats['DemandCV'] < 0.5, 'Stable',
        np.where(sku_stats['DemandCV'] < 1.0, 'Moderate', 'Volatile')
    )

    # Filter SKUs with sufficient data
    sufficient_mask = sku_stats['DaysWithSales'] >=
min_days_with_sales
    sufficient_skus = sku_stats[sufficient_mask].copy()
    insufficient_skus = sku_stats[~sufficient_mask].copy()

    print(f"\n□ DEMAND ANALYSIS RESULTS:")
    print(f"  Total SKUs analyzed: {len(sku_stats):,}")
    print(f"  SKUs with ≥{min_days_with_sales} days sales: 
{len(sufficient_skus):,} 
({len(sufficient_skus)/len(sku_stats)*100:.1f}%)")
    print(f"  SKUs with insufficient history: 
{len(insufficient_skus):,}")

    # Demand variability summary
    print(f"\n  Demand Variability Distribution:")
    for stability in ['Stable', 'Moderate', 'Volatile']:
        count = (sufficient_skus['DemandStability'] == 
stability).sum()
        pct = count / len(sufficient_skus) * 100
        print(f"    {stability}: {count:,} SKUs ({pct:.1f}%)")

    # Top SKUs by revenue
    print(f"\n  Top 5 SKUs by Revenue:")
    top_skus = sufficient_skus.nlargest(5, 'TotalRevenue')
[['StockCode', 'Description', 'TotalRevenue', 'AvgDailyDemand']]
    for idx, row in top_skus.iterrows():
        print(f"    {row['StockCode']}: ${row['TotalRevenue']:,.2f} 
({row['AvgDailyDemand']:.1f} units/day)")

    return sufficient_skus, daily_demand

# Execute demand analysis
sku_stats, daily_demand = analyze_sku_demand(df_clean, 
min_days_with_sales=30)
```

3. Obsolescence Risk Detection

Detect obsolescence risk from product descriptions.

```
def detect_obsolescence_risk(description):
    """
    Detect obsolescence risk from product description with detailed 
categorization
    """
    if pd.isna(description):
        return {'risk_level': 'Medium', 'risk_type': 'Unknown', 
'confidence': 0.5}

    desc_upper = str(description).upper()

    # Define risk patterns
    seasonal_terms = {
        'CHRISTMAS': 0.9, 'XMAS': 0.9, 'NOEL': 0.8,
        'EASTER': 0.8, 'EGG': 0.7,
        'HALLOWEEN': 0.8, 'PUMPKIN': 0.7,
        'SUMMER': 0.6, 'WINTER': 0.6, 'SPRING': 0.6, 'AUTUMN': 0.6,
        'SEASONAL': 0.8, 'FESTIVE': 0.7, 'HOLIDAY': 0.7,
        'DECORATION': 0.5
    }

    fashion_terms = {
        'RETRO': 0.7, 'VINTAGE': 0.6, 'TRENDY': 0.8,
        'FASHION': 0.7, 'STYLE': 0.5, 'DESIGNER': 0.6,
        'COLLECTION': 0.5, 'MODERN': 0.5, 'CONTEMPORARY': 0.5
    }

    perishable_terms = {
        'FOOD': 0.9, 'CHOCOLATE': 0.8, 'PERISHABLE': 0.9,
        'EXPIR': 0.9, 'FRESH': 0.7, 'BAKED': 0.7
    }

    # Calculate risk scores
    seasonal_score = sum(score for term, score in
seasonal_terms.items() if term in desc_upper)
    fashion_score = sum(score for term, score in fashion_terms.items()
if term in desc_upper)
    perishable_score = sum(score for term, score in
perishable_terms.items() if term in desc_upper)

    # Determine primary risk type and level
    max_score = max(seasonal_score, fashion_score, perishable_score)

    if max_score >= 0.8:
        risk_level = 'High'
        confidence = min(1.0, max_score)
    elif max_score >= 0.5:
        risk_level = 'Medium'
        confidence = max_score
    else:
        risk_level = 'Low'
        confidence = 0.3

    # Determine risk type
    if seasonal_score == max_score and seasonal_score > 0:
        risk_type = 'Seasonal'
    elif fashion_score == max_score and fashion_score > 0:
        risk_type = 'Fashion'
    elif perishable_score == max_score and perishable_score > 0:
        risk_type = 'Perishable'
    else:
        risk_type = 'Stable'
        confidence = 0.2

    return {
        'risk_level': risk_level,
        'risk_type': risk_type,
        'confidence': round(confidence, 2)
    }

def add_obsolescence_features(sku_stats):
    """
    Add obsolescence risk features to SKU statistics
    """
    print("\n ANALYZING OBSOLESCENCE RISKS")

    # Apply obsolescence detection
    risk_results = 
sku_stats['Description'].apply(detect_obsolescence_risk)

    # Extract to columns
    sku_stats['ObsolescenceRisk'] = [r['risk_level'] for r in 
risk_results]
    sku_stats['ObsolescenceType'] = [r['risk_type'] for r in 
risk_results]
    sku_stats['RiskConfidence'] = [r['confidence'] for r in 
risk_results]

    # Print summary
    risk_counts = sku_stats['ObsolescenceRisk'].value_counts()
    print(f"  Obsolescence Risk Distribution:")
    for risk_level in ['High', 'Medium', 'Low']:
        count = risk_counts.get(risk_level, 0)
        pct = count / len(sku_stats) * 100
        print(f"    {risk_level}: {count:,} SKUs ({pct:.1f}%)")

    return sku_stats

# Execute obsolescence analysis
sku_stats = add_obsolescence_features(sku_stats)
```

4. ABC Analysis (Pareto Analysis)

Classify SKUs based on revenue contribution.

```
def perform_abc_analysis(sku_stats, percentiles=(80, 95)):
    """
    Perform ABC analysis with configurable percentiles
    """
    print("\n" + "=" * 70)
    print(" PERFORMING ABC ANALYSIS (PARETO ANALYSIS)")
    print("=" * 70)

    a_threshold, b_threshold = percentiles
    # Sort by revenue descending
    sku_sorted = sku_stats.sort_values('TotalRevenue',
ascending=False).copy()

    # Calculate cumulative percentages
    sku_sorted['CumulativeRevenue'] =
sku_sorted['TotalRevenue'].cumsum()
    sku_sorted['CumulativePct'] = (sku_sorted['CumulativeRevenue'] /
                                   sku_sorted['TotalRevenue'].sum() *
100)

    # Assign ABC class based on cumulative percentage
    def assign_abc_class(cum_pct):
        if cum_pct <= a_threshold:
            return 'A'
        elif cum_pct <= b_threshold:
            return 'B'
        else:
            return 'C'

    sku_sorted['ABC_Class'] =
sku_sorted['CumulativePct'].apply(assign_abc_class)

    # Calculate statistics per ABC class
    abc_summary = sku_sorted.groupby('ABC_Class').agg({
        'StockCode': 'count',
        'TotalRevenue': 'sum',
        'AvgDailyDemand': 'mean',
        'AvgUnitPrice': 'mean',
        'DaysWithSales': 'mean'
    }).reset_index()

    abc_summary['Pct_SKUs'] = (abc_summary['StockCode'] /
abc_summary['StockCode'].sum()) * 100
    abc_summary['Pct_Revenue'] = (abc_summary['TotalRevenue'] /
abc_summary['TotalRevenue'].sum()) * 100
    abc_summary['Revenue_per_SKU'] = abc_summary['TotalRevenue'] /
abc_summary['StockCode']

    print(f"\n□ ABC ANALYSIS RESULTS ({a_threshold}/{b_threshold}
Percentiles):")
    print("-" * 60)
    print(f"{'Class':<6} {'SKUs':>8} {'% SKUs':>8} {'Revenue':>12} {'%
Revenue':>10} {'Rev/SKU':>12}")
    print("-" * 60)

    for _, row in abc_summary.iterrows():
        print(f"{row['ABC_Class']:<6} "
              f"{row['StockCode']:>8,} "
              f"{row['Pct_SKUs']:>7.1f}% "
              f"${row['TotalRevenue']:>11,.0f} "
              f"{row['Pct_Revenue']:>9.1f}% "
              f"${row['Revenue_per_SKU']:>11,.0f}")

    print("-" * 60)

    return sku_sorted, abc_summary

# Execute ABC analysis
sku_stats, abc_summary = perform_abc_analysis(sku_stats,
percentiles=(80, 95))
```

5. Lead Time Assignment

Assign realistic lead times based on ABC class and other factors.

```
def assign_lead_times(sku_stats):
    """
    Assign realistic lead times based on ABC class, obsolescence risk,
and other factors
    """
    print("\n" + "=" * 70)
    print(" ASSIGNING LEAD TIMES")
    print("=" * 70)

    # Base lead times by ABC class
    base_lead_times = {
        'A': {'min': 3, 'max': 7, 'mean': 5},
        'B': {'min': 7, 'max': 14, 'mean': 10},
        'C': {'min': 14, 'max': 30, 'mean': 21}
    }

    def calculate_lead_time(row):
        # Get base lead time based on ABC class
        base = base_lead_times[row['ABC_Class']]

        # Start with base mean
        lead_time = base['mean']

        # Adjust for obsolescence risk
        if row['ObsolescenceRisk'] == 'High':
            lead_time = max(base['min'], lead_time * 0.7)
        elif row['ObsolescenceRisk'] == 'Low':
            lead_time = min(base['max'], lead_time * 1.2)

        # Adjust for unit price
        if row['AvgUnitPrice'] > 100:
            lead_time *= 1.3
        elif row['AvgUnitPrice'] < 10:
            lead_time *= 0.8

        # Adjust for demand stability
        if row['DemandStability'] == 'Volatile':
            lead_time *= 0.8
        elif row['DemandStability'] == 'Stable':
            lead_time *= 1.2

        # Round and ensure within bounds
        lead_time = round(lead_time)
        lead_time = max(base['min'], min(base['max'], lead_time))

        return lead_time

    def calculate_lead_time_std(row):
        """Calculate lead time standard deviation"""
        base_std = row['LeadTimeDays'] * 0.2

        if row['ObsolescenceRisk'] == 'High':
            base_std *= 1.5

        desc = str(row['Description']).upper()
        if any(term in desc for term in ['IMPORT', 'CHINA', 'ASIA',
'EUROPE', 'ITALY']):
            base_std *= 1.8

        return round(base_std, 1)

    # Apply lead time calculations
    sku_stats['LeadTimeDays'] = sku_stats.apply(calculate_lead_time,
axis=1)
    sku_stats['LeadTimeStd'] =
sku_stats.apply(calculate_lead_time_std, axis=1)

    # Print summary statistics
    print(f"\n LEAD TIME DISTRIBUTION:")
    for abc_class in ['A', 'B', 'C']:
        class_data = sku_stats[sku_stats['ABC_Class'] == abc_class]
        if len(class_data) > 0:
            mean_lt = class_data['LeadTimeDays'].mean()
            std_lt = class_data['LeadTimeStd'].mean()
            print(f"  {abc_class}-items: {mean_lt:.1f} ± {std_lt:.1f}
days "
                  f"(range: {class_data['LeadTimeDays'].min()}-
{class_data['LeadTimeDays'].max()})")

    return sku_stats
```

```python
# Execute lead time assignment
sku_stats = assign_lead_times(sku_stats)
```

6. Service Level Calculations

Calculate target and achievable service levels.

```python
def calculate_service_level_target(row):
    """
    Calculate appropriate service level target based on ABC class and
obsolescence risk
    """
    # Base service levels by ABC class
    base_service = {
        'A': 0.97,
        'B': 0.95,
        'C': 0.90
    }

    target = base_service.get(row['ABC_Class'], 0.95)

    # Adjust for obsolescence risk
    if row['ObsolescenceRisk'] == 'High':
        target = max(0.85, target - 0.05)
    elif row['ObsolescenceRisk'] == 'Low':
        target = min(0.99, target + 0.02)

    return round(target, 3)

def calculate_achievable_service_level(row):
    """
    Calculate what service level current stock can actually provide
    """
    from scipy import stats

    # Mean and std of demand during lead time
    mean_demand_lt = row['AvgDailyDemand'] * row['LeadTimeDays']
    std_demand_lt = np.sqrt(
        row['LeadTimeDays'] * (row['DemandStd'] ** 2) +
        (row['AvgDailyDemand'] ** 2) * (row['LeadTimeStd'] ** 2)
    )

    if std_demand_lt == 0 or mean_demand_lt == 0:
        return 0.5

    # Z-score for current stock position
    z_actual = (row['CurrentStock'] - mean_demand_lt) / std_demand_lt
    # Service level = Probability(demand ≤ current stock)
    service_level = stats.norm.cdf(z_actual)

    return round(max(0, min(1, service_level)), 3)
```

7. Safety Stock & Reorder Point Calculation

Calculate optimal safety stock and reorder points.

```
def calculate_safety_stock(row):
    """
    Calculate safety stock using statistical formula with service
level target
    """
    from scipy import stats

    # Get service level target
    service_level = row['ServiceLevelTarget']

    # Get Z-score for service level
    z_score = stats.norm.ppf(service_level)

    # Extract components
    D = row['AvgDailyDemand']
    σ_D = max(row['DemandStd'], D * 0.1)
    L = row['LeadTimeDays']
    σ_L = row['LeadTimeStd']

    # Safety stock calculation
    ss = z_score * np.sqrt(L * (σ_D ** 2) + (D ** 2) * (σ_L ** 2))

    # Apply minimum safety stock
    min_ss = D * 0.5

    return round(max(min_ss, ss))
```

8. Stock Status Determination

Determine if SKUs are optimally stocked, overstocked, or understocked.

```
def determine_stock_status(row):
    """
    Determine stock status with CORRECT LOGIC (no double-counting
safety stock)
    """
    # Calculate key metrics
    days_on_hand = row['CurrentStock'] / max(row['AvgDailyDemand'],
0.1)
```

```python
    # Define business rules by ABC class
    business_rules = {
        'A': {
            'max_days': 30,
            'tolerance_pct': 1.10,
            'reason': 'High-value items need tight control'
        },
        'B': {
            'max_days': 45,
            'tolerance_pct': 1.20,
            'reason': 'Balanced approach for moderate-value items'
        },
        'C': {
            'max_days': 60,
            'tolerance_pct': 1.30,
            'reason': 'Low-value items, ordering efficiency matters'
        }
    }

    rules = business_rules.get(row['ABC_Class'], business_rules['C'])

    # Calculate tolerance threshold
    tolerance_threshold = row['ReorderPoint'] * rules['tolerance_pct']

    # Determine status
    status_reasons = []

    # Check for UNDERSTOCKED
    if row['CurrentStock'] < row['ReorderPoint']:
        shortage = row['ReorderPoint'] - row['CurrentStock']
        status_reasons.append(f"Understocked by {int(shortage)}
units")
        return 'Understocked', "; ".join(status_reasons)

    # Check for OVERSTOCKED
    overstock_conditions = []

    if days_on_hand > rules['max_days']:
        overstock_conditions.append(f"{days_on_hand:.1f} days
inventory > {rules['max_days']} day limit")

    if row['CurrentStock'] > tolerance_threshold:
        overstock_conditions.append(f"Stock {row['CurrentStock']} >
tolerance {tolerance_threshold:.0f}")

    if overstock_conditions:
        status_reasons.append(f"Overstocked: {',
'.join(overstock_conditions)}")
```

```
        return 'Overstocked', "; ".join(status_reasons)

    # Otherwise OPTIMAL
    status_reasons.append(f"Within {row['ABC_Class']}-item
guidelines")
    return 'Optimal', "; ".join(status_reasons)
```

9. Comprehensive Inventory Metrics

Calculate all inventory metrics and simulate current stock levels.

```
def calculate_inventory_metrics(sku_stats):
    """
    Calculate comprehensive inventory metrics for each SKU
    """
    print("\n" + "=" * 70)
    print("□ CALCULATING INVENTORY METRICS")
    print("=" * 70)

    # 1. Calculate service level targets
    print("Calculating service level targets...")
    sku_stats['ServiceLevelTarget'] =
sku_stats.apply(calculate_service_level_target, axis=1)

    # 2. Calculate safety stock
    print("Calculating safety stock...")
    sku_stats['SafetyStock'] = sku_stats.apply(calculate_safety_stock,
axis=1)

    print("Calculating reorder points...")
    sku_stats['ReorderPoint'] = (sku_stats['AvgDailyDemand'] *
sku_stats['LeadTimeDays'] +
                                sku_stats['SafetyStock']).round(0)

    # 3. Simulate current stock
    print("Simulating current stock levels...")
    np.random.seed(42)
    sku_stats['CurrentStock'] = (sku_stats['ReorderPoint'] *
                                np.random.uniform(0.5, 2.0,
len(sku_stats))).round(0)

    # 4. Calculate achievable service level
    print("Calculating achievable service levels...")
    sku_stats['ServiceLevelAchievable'] =
sku_stats.apply(calculate_achievable_service_level, axis=1)

    # 5. Determine stock status
    print("Determining stock status...")
    status_results = sku_stats.apply(
        lambda row: pd.Series(determine_stock_status(row)),
        axis=1
    )
    sku_stats['StockStatus'] = status_results[0]
    sku_stats['StatusReason'] = status_results[1]

    # 6. Calculate excess/deficit stock
    print("Calculating excess/deficit quantities...")

    def calculate_excess_deficit(row):
        if row['StockStatus'] == 'Overstocked':
            rules = {'A': 1.10, 'B': 1.20, 'C': 1.30}
            tolerance = rules.get(row['ABC_Class'], 1.20)
            threshold = row['ReorderPoint'] * tolerance
            excess = max(0, row['CurrentStock'] - threshold)
            return round(excess), 0
        elif row['StockStatus'] == 'Understocked':
            deficit = max(0, row['ReorderPoint'] -
row['CurrentStock'])
            return 0, round(deficit)
        else:
            return 0, 0

    excess_deficit = sku_stats.apply(
        lambda row: pd.Series(calculate_excess_deficit(row)),
        axis=1
    )
    sku_stats['ExcessStock'] = excess_deficit[0]
    sku_stats['DeficitStock'] = excess_deficit[1]

    # 7. Calculate inventory value and costs
    print("Calculating inventory values and costs...")
    sku_stats['InventoryValue'] = (sku_stats['CurrentStock'] *
sku_stats['AvgUnitPrice']).round(2)
    sku_stats['ExcessValue'] = (sku_stats['ExcessStock'] *
sku_stats['AvgUnitPrice']).round(2)
    sku_stats['DeficitValue'] = (sku_stats['DeficitStock'] *
sku_stats['AvgUnitPrice']).round(2)

    # Calculate holding costs (25% annual carrying cost)
    daily_holding_rate = 0.25 / 365
    sku_stats['DailyHoldingCost'] = (sku_stats['InventoryValue'] *
daily_holding_rate).round(2)
    sku_stats['ExcessHoldingCost'] = (sku_stats['ExcessValue'] *
daily_holding_rate).round(2)

    # 8. Calculate key performance metrics
    sku_stats['StockoutRisk'] = (1 -
sku_stats['ServiceLevelAchievable']).round(3)
```

```
    sku_stats['ServiceLevelGap'] = (sku_stats['ServiceLevelTarget'] -

sku_stats['ServiceLevelAchievable']).round(3)

    # Days of inventory
    sku_stats['DaysOnHand'] = (sku_stats['CurrentStock'] /

sku_stats['AvgDailyDemand'].clip(lower=0.1)).round(1)

    # Inventory turns (annual)
    sku_stats['AnnualTurns'] = (365 * sku_stats['AvgDailyDemand'] /

sku_stats['CurrentStock'].clip(lower=1)).round(1)

    # Print summary statistics
    print(f"\n□ INVENTORY METRICS SUMMARY:")
    print(f"  Total SKUs analyzed: {len(sku_stats):,}")

    status_counts = sku_stats['StockStatus'].value_counts()
    for status in ['Optimal', 'Overstocked', 'Understocked']:
        count = status_counts.get(status, 0)
        pct = count / len(sku_stats) * 100
        print(f"  {status}: {count:,} SKUs ({pct:.1f}%)")

    total_excess = sku_stats['ExcessValue'].sum()
    total_deficit = sku_stats['DeficitValue'].sum()
    total_inventory = sku_stats['InventoryValue'].sum()

    print(f"\n  Total Inventory Value: ${total_inventory:,.2f}")
    print(f"  Total Excess Value: ${total_excess:,.2f}")
    print(f"  Total Deficit Value: ${total_deficit:,.2f}")
    print(f"  Excess % of Total:
{total_excess/total_inventory*100:.1f}%")

    # Calculate potential savings
    annual_excess_cost = sku_stats['ExcessHoldingCost'].sum() * 365
    print(f"\n□ POTENTIAL ANNUAL SAVINGS:")
    print(f"  From reducing excess inventory: $
{annual_excess_cost:,.2f}")

    return sku_stats

# Execute inventory metrics calculation
sku_stats = calculate_inventory_metrics(sku_stats)
```

10. Data Visualization

Create comprehensive visualizations of the inventory analysis.

```
def create_visualizations(sku_stats, abc_summary):
    """
    Create comprehensive visualizations for the analysis
    """
    print("\n" + "=" * 70)
    print("▯ CREATING VISUALIZATIONS")
    print("=" * 70)

    # Create figure with multiple subplots
    fig = plt.figure(figsize=(20, 16))

    # 1. PARETO CHART
    print("Creating Pareto chart...")
    ax1 = plt.subplot(3, 3, 1)

    pareto_data = sku_stats.sort_values('TotalRevenue',
ascending=False).copy()
    pareto_data['CumulativePct'] =
(pareto_data['TotalRevenue'].cumsum() /
                                   pareto_data['TotalRevenue'].sum() *
100)
    pareto_data['ItemNumber'] = range(1, len(pareto_data) + 1)

    show_top = min(100, len(pareto_data))
    bars = ax1.bar(pareto_data['ItemNumber'].iloc[:show_top],
                   pareto_data['TotalRevenue'].iloc[:show_top],
                   color='skyblue', alpha=0.7)
    ax1.set_xlabel('SKU Rank (by Revenue)', fontsize=10)
    ax1.set_ylabel('Revenue ($)', color='blue', fontsize=10)
    ax1.tick_params(axis='y', labelcolor='blue')
    ax1.set_title('Pareto Chart: Revenue Distribution', fontsize=12,
fontweight='bold')

    ax1_twin = ax1.twinx()
    ax1_twin.plot(pareto_data['ItemNumber'].iloc[:show_top],
                  pareto_data['CumulativePct'].iloc[:show_top],
                  'r-', linewidth=2)
    ax1_twin.set_ylabel('Cumulative %', color='red', fontsize=10)
    ax1_twin.tick_params(axis='y', labelcolor='red')
    ax1_twin.axhline(y=80, color='green', linestyle='--', alpha=0.7)
    ax1_twin.axhline(y=95, color='orange', linestyle='--', alpha=0.7)

    # 2. ABC CLASS DISTRIBUTION
    print("Creating ABC distribution pie chart...")
    ax2 = plt.subplot(3, 3, 2)
    abc_counts = abc_summary.set_index('ABC_Class')['StockCode']
    colors = {'A': '#FFD700', 'B': '#C0C0C0', 'C': '#CD7F32'}

    pie_colors = [colors.get(cls, '#999999') for cls in
abc_counts.index]
```

```
    wedges, texts, autotexts = ax2.pie(abc_counts.values, 
labels=abc_counts.index,
                                       colors=pie_colors, 
autopct='%1.1f%%',
                                       startangle=90)
    ax2.set_title('ABC Class Distribution', fontsize=12, 
fontweight='bold')

    # 3. STOCK STATUS BY ABC CLASS
    print("Creating stock status by ABC class chart...")
    ax3 = plt.subplot(3, 3, 3)

    status_by_abc = pd.crosstab(sku_stats['ABC_Class'], 
sku_stats['StockStatus'])
    status_by_abc = status_by_abc.reindex(['A', 'B', 'C'])
    status_by_abc = status_by_abc[['Understocked', 'Optimal', 
'Overstocked']]

    colors_status = {'Understocked': '#FF6B6B', 'Optimal': '#4ECDC4', 
'Overstocked': '#45B7D1'}
    color_list = [colors_status[col] for col in status_by_abc.columns]

    status_by_abc.plot(kind='bar', stacked=True, ax=ax3, 
color=color_list, alpha=0.8)
    ax3.set_xlabel('ABC Class', fontsize=10)
    ax3.set_ylabel('Number of SKUs', fontsize=10)
    ax3.set_title('Stock Status by ABC Class', fontsize=12, 
fontweight='bold')
    ax3.legend(title='Stock Status')
    ax3.tick_params(axis='x', rotation=0)

    # 4. SERVICE LEVEL ACHIEVEMENT
    print("Creating service level analysis chart...")
    ax4 = plt.subplot(3, 3, 4)

    sample_size = min(200, len(sku_stats))
    sample_data = sku_stats.sample(sample_size, random_state=42)

    color_map = {'A': 0, 'B': 1, 'C': 2}
    colors_abc = [color_map.get(cls, 0) for cls in 
sample_data['ABC_Class']]

    scatter = ax4.scatter(sample_data['ServiceLevelTarget'] * 100,
                         sample_data['ServiceLevelAchievable'] * 100,
                         c=colors_abc, cmap='viridis', s=50, 
alpha=0.6)

    ax4.plot([0, 100], [0, 100], 'r--', alpha=0.5)
    ax4.set_xlabel('Target Service Level (%)', fontsize=10)
    ax4.set_ylabel('Achievable Service Level (%)', fontsize=10)
    ax4.set_title('Service Level: Target vs Achievable', fontsize=12,
fontweight='bold')
    ax4.grid(True, alpha=0.3)
    ax4.set_xlim(80, 100)
    ax4.set_ylim(80, 100)

    cbar = plt.colorbar(scatter, ax=ax4)
    cbar.set_label('ABC Class (0=A, 1=B, 2=C)', fontsize=9)

    # 5. DEMAND VARIABILITY ANALYSIS
    print("Creating demand variability chart...")
    ax5 = plt.subplot(3, 3, 5)

    demand_summary = sku_stats.groupby(['DemandStability',
'ABC_Class']).size().unstack()
    demand_summary = demand_summary.reindex(['Stable', 'Moderate',
'Volatile'])
    demand_summary = demand_summary[['A', 'B', 'C']]

    demand_summary.plot(kind='bar', ax=ax5, color=['#FFD700',
'#C0C0C0', '#CD7F32'], alpha=0.8)
    ax5.set_xlabel('Demand Stability', fontsize=10)
    ax5.set_ylabel('Number of SKUs', fontsize=10)
    ax5.set_title('Demand Stability by ABC Class', fontsize=12,
fontweight='bold')
    ax5.legend(title='ABC Class')
    ax5.tick_params(axis='x', rotation=0)

    # 6. OBSOLESCENCE RISK ANALYSIS
    print("Creating obsolescence risk chart...")
    ax6 = plt.subplot(3, 3, 6)

    obsolescence_summary = sku_stats.groupby(['ObsolescenceRisk',
'ABC_Class']).size().unstack()
    obsolescence_summary = obsolescence_summary.reindex(['High',
'Medium', 'Low'])
    obsolescence_summary = obsolescence_summary[['A', 'B', 'C']]

    obsolescence_summary.plot(kind='bar', ax=ax6, color=['#FFD700',
'#C0C0C0', '#CD7F32'], alpha=0.8)
    ax6.set_xlabel('Obsolescence Risk', fontsize=10)
    ax6.set_ylabel('Number of SKUs', fontsize=10)
    ax6.set_title('Obsolescence Risk by ABC Class', fontsize=12,
fontweight='bold')
    ax6.legend(title='ABC Class')
    ax6.tick_params(axis='x', rotation=0)

    # 7. EXCESS INVENTORY VALUE
    print("Creating excess inventory chart...")
    ax7 = plt.subplot(3, 3, 7)
    excess_by_abc = sku_stats[sku_stats['StockStatus'] ==
'Overstocked'].groupby('ABC_Class')['ExcessValue'].sum()
    excess_by_abc = excess_by_abc.reindex(['A', 'B', 'C']).fillna(0)

    colors_excess = {'A': '#FF6B6B', 'B': '#FFA726', 'C': '#66BB6A'}
    bar_colors = [colors_excess.get(cls, '#999999') for cls in
excess_by_abc.index]

    bars = ax7.bar(excess_by_abc.index, excess_by_abc.values,
color=bar_colors, alpha=0.8)
    ax7.set_xlabel('ABC Class', fontsize=10)
    ax7.set_ylabel('Excess Inventory Value ($)', fontsize=10)
    ax7.set_title('Excess Inventory Value by ABC Class', fontsize=12,
fontweight='bold')
    ax7.yaxis.set_major_formatter(plt.FuncFormatter(lambda x, p: f'$
{x:,.0f}'))

    # 8. INVENTORY TURNS
    print("Creating inventory turns chart...")
    ax8 = plt.subplot(3, 3, 8)

    turns_by_abc = sku_stats.groupby('ABC_Class')
['AnnualTurns'].mean()
    turns_by_abc = turns_by_abc.reindex(['A', 'B', 'C'])

    ax8.bar(turns_by_abc.index, turns_by_abc.values,
            color=['#FFD700', '#C0C0C0', '#CD7F32'], alpha=0.8)
    ax8.set_xlabel('ABC Class', fontsize=10)
    ax8.set_ylabel('Average Annual Inventory Turns', fontsize=10)
    ax8.set_title('Inventory Turnover by ABC Class', fontsize=12,
fontweight='bold')

    # 9. LEAD TIME DISTRIBUTION
    print("Creating lead time distribution chart...")
    ax9 = plt.subplot(3, 3, 9)

    leadtime_data = [sku_stats[sku_stats['ABC_Class'] == cls]
['LeadTimeDays']
                     for cls in ['A', 'B', 'C']]

    box = ax9.boxplot(leadtime_data, labels=['A', 'B', 'C'],
patch_artist=True)

    colors_box = ['#FFD700', '#C0C0C0', '#CD7F32']
    for patch, color in zip(box['boxes'], colors_box):
        patch.set_facecolor(color)
        patch.set_alpha(0.7)

    ax9.set_xlabel('ABC Class', fontsize=10)
    ax9.set_ylabel('Lead Time (Days)', fontsize=10)
    ax9.set_title('Lead Time Distribution by ABC Class', fontsize=12,
fontweight='bold')

    plt.tight_layout()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = f'{output_dir}/inventory_analysis_{timestamp}.png'
    plt.savefig(filename, dpi=300, bbox_inches='tight')

    print(f"\n Visualizations saved as: {filename}")
    plt.show()

    return fig

# Execute visualization
fig = create_visualizations(sku_stats, abc_summary)
```

11. Export Results

Export all results to Excel and CSV files for further analysis.

```
def export_results(sku_stats, summary_stats, abc_summary):
    """
    Export all results to Excel and CSV files for further analysis
    """
    print("\n" + "=" * 70)
    print(" EXPORTING RESULTS")
    print("=" * 70)

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    # EXPORT TO EXCEL
    excel_filename =
f'{output_dir}/inventory_optimization_results_{timestamp}.xlsx'

    with pd.ExcelWriter(excel_filename, engine='openpyxl') as writer:
        # Sheet 1: SKU-Level Analysis
        print("Creating SKU-level analysis sheet...")
        sku_export = sku_stats[[
            'StockCode', 'Description', 'ABC_Class',
            'ObsolescenceRisk', 'ObsolescenceType', 'DemandStability',
            'AvgDailyDemand', 'DemandStd', 'DemandCV',
'DaysWithSales',
            'AvgUnitPrice', 'TotalRevenue',
            'LeadTimeDays', 'LeadTimeStd',
            'ServiceLevelTarget', 'SafetyStock', 'ReorderPoint',
            'CurrentStock', 'DaysOnHand', 'AnnualTurns',
            'ServiceLevelAchievable', 'StockoutRisk',
'ServiceLevelGap',
            'StockStatus', 'StatusReason',
            'ExcessStock', 'DeficitStock', 'ExcessValue',
'DeficitValue',
            'InventoryValue', 'DailyHoldingCost', 'ExcessHoldingCost'
        ]].copy()

        sku_export['Priority'] = sku_export['ABC_Class'].map({'A': 1,
'B': 2, 'C': 3})
        sku_export = sku_export.sort_values(['Priority',
'ExcessValue'], ascending=[True, False])
        sku_export = sku_export.drop('Priority', axis=1)

        sku_export.to_excel(writer, sheet_name='SKU_Analysis',
index=False)

        # Sheet 2: Summary Statistics
        print("Creating summary statistics sheet...")
        summary_df = pd.DataFrame([
            ['Total SKUs Analyzed', len(sku_stats)],
            ['Total Revenue', f"$
{sku_stats['TotalRevenue'].sum():,.2f}"],
            ['Total Inventory Value', f"$
{sku_stats['InventoryValue'].sum():,.2f}"],
            ['Total Excess Value', f"$
{sku_stats['ExcessValue'].sum():,.2f}"],
            ['Total Deficit Value', f"$
{sku_stats['DeficitValue'].sum():,.2f}"],
            ['Potential Annual Savings', f"$
{sku_stats['ExcessHoldingCost'].sum() * 365:,.2f}"],
            ['Data Retention',
f"{summary_stats['data_retention_pct']:.1f}%"],
            ['Time Period',
f"{summary_stats['time_range_start'].date()} to
{summary_stats['time_range_end'].date()}"],
            ['ABC Class - A Items',
f"{abc_summary[abc_summary['ABC_Class']=='A']
['StockCode'].values[0]:,} SKUs
({abc_summary[abc_summary['ABC_Class']=='A']
['Pct_Revenue'].values[0]:.1f}% revenue)"],
            ['ABC Class - B Items',
f"{abc_summary[abc_summary['ABC_Class']=='B']
['StockCode'].values[0]:,} SKUs
({abc_summary[abc_summary['ABC_Class']=='B']
['Pct_Revenue'].values[0]:.1f}% revenue)"],
            ['ABC Class - C Items',
f"{abc_summary[abc_summary['ABC_Class']=='C']
['StockCode'].values[0]:,} SKUs
({abc_summary[abc_summary['ABC_Class']=='C']
['Pct_Revenue'].values[0]:.1f}% revenue)"],
            ['Stock Status - Optimal',
f"{(sku_stats['StockStatus']=='Optimal').sum():,} SKUs
({(sku_stats['StockStatus']=='Optimal').sum()/len(sku_stats)*100:.1f}
%)"],
            ['Stock Status - Overstocked',
f"{(sku_stats['StockStatus']=='Overstocked').sum():,} SKUs
({(sku_stats['StockStatus']=='Overstocked').sum()/len(sku_stats)*100:.
1f}%)"],
            ['Stock Status - Understocked',
f"{(sku_stats['StockStatus']=='Understocked').sum():,} SKUs
({(sku_stats['StockStatus']=='Understocked').sum()/len(sku_stats)*100:
.1f}%)"]
        ], columns=['Metric', 'Value'])

        summary_df.to_excel(writer, sheet_name='Summary', index=False)

        # Sheet 3: ABC Class Summary
        print("Creating ABC summary sheet...")
        abc_summary.to_excel(writer, sheet_name='ABC_Summary',
index=False)

        # Sheet 4: Recommendations
        print("Creating recommendations sheet...")
        recommendations =
sku_stats[sku_stats['StockStatus'].isin(['Overstocked',
'Understocked'])].copy()

        def generate_recommendation(row):
            if row['StockStatus'] == 'Overstocked':
                action = "Reduce stock"
                target = f"Aim for: {row['ReorderPoint']:.0f} units"
                reason = f"Currently {row['CurrentStock']:.0f} units
({row['ExcessStock']:.0f} excess)"
            else:
                action = "Increase stock"
                target = f"Aim for: {row['ReorderPoint']:.0f} units"
                reason = f"Currently {row['CurrentStock']:.0f} units
({row['DeficitStock']:.0f} deficit)"

            priority = {'A': 'High', 'B': 'Medium', 'C':
'Low'}.get(row['ABC_Class'], 'Medium')

            return pd.Series({
                'Priority': priority,
                'Action': action,
                'Target': target,
                'Reason': reason,
                'Impact': f"${row['ExcessValue'] +
row['DeficitValue']:,.2f}"
            })

        rec_details = recommendations.apply(generate_recommendation,
axis=1)
        recommendations = pd.concat([recommendations[['StockCode',
'Description', 'ABC_Class']],
                                    rec_details], axis=1)

        recommendations = recommendations.sort_values(['Priority',
'Impact'], ascending=[True, False])
        recommendations.to_excel(writer, sheet_name='Recommendations',
index=False)

    print(f"□ Excel file exported: {excel_filename}")

    # EXPORT TO CSV (FOR POWER BI)
    csv_filename =
f'{output_dir}/inventory_data_for_powerbi_{timestamp}.csv'

    powerbi_data = sku_stats[[
        'StockCode', 'Description', 'ABC_Class', 'ObsolescenceRisk',
        'AvgDailyDemand', 'DemandStd', 'DemandCV',
        'LeadTimeDays', 'SafetyStock', 'ReorderPoint',
        'CurrentStock', 'StockStatus',
        'ExcessStock', 'DeficitStock', 'ExcessValue', 'DeficitValue',
        'InventoryValue', 'ServiceLevelTarget',
'ServiceLevelAchievable',
        'AnnualTurns', 'DaysOnHand'
    ]].copy()

    powerbi_data['StatusColor'] = powerbi_data['StockStatus'].map({
        'Optimal': '#4ECDC4',
        'Overstocked': '#FF6B6B',
        'Understocked': '#45B7D1'
    })

    powerbi_data['ActionPriority'] = powerbi_data.apply(
        lambda x: 'Immediate' if x['ABC_Class'] == 'A' and
x['StockStatus'] != 'Optimal'
        else 'High' if x['ABC_Class'] == 'B' and x['StockStatus'] !=
'Optimal'
        else 'Medium' if x['StockStatus'] != 'Optimal'
        else 'Low', axis=1
    )

    powerbi_data.to_csv(csv_filename, index=False)
    print(f"□ CSV file for Power BI exported: {csv_filename}")
```

```python
    return excel_filename, csv_filename

# Execute export
excel_file, csv_file = export_results(sku_stats, summary_stats,
abc_summary)
```

12. Final Report

Display key performance indicators and actionable insights.

```python
print("\n" + "=" * 70)
print("□ PROJECT COMPLETION REPORT")
print("=" * 70)

# Calculate key metrics
total_skus = len(sku_stats)
overstocked_skus = (sku_stats['StockStatus'] == 'Overstocked').sum()
understocked_skus = (sku_stats['StockStatus'] == 'Understocked').sum()
optimal_skus = (sku_stats['StockStatus'] == 'Optimal').sum()

total_excess_value = sku_stats['ExcessValue'].sum()
total_deficit_value = sku_stats['DeficitValue'].sum()
total_inventory_value = sku_stats['InventoryValue'].sum()

annual_savings = sku_stats['ExcessHoldingCost'].sum() * 365

# A-items performance
a_items = sku_stats[sku_stats['ABC_Class'] == 'A']
a_optimal_pct = (a_items['StockStatus'] == 'Optimal').sum() /
len(a_items) * 100 if len(a_items) > 0 else 0

print(f"\n□ KEY PERFORMANCE INDICATORS:")
print(f"  1. Inventory Health: {optimal_skus/total_skus*100:.1f}% SKUs
optimally stocked")
print(f"  2. A-Item Performance: {a_optimal_pct:.1f}% of A-items
optimally stocked")
print(f"  3. Excess Inventory: ${total_excess_value:,.2f}
({total_excess_value/total_inventory_value*100:.1f}% of total)")
print(f"  4. Service Level Gap:
{(sku_stats['ServiceLevelGap'].mean()*100):.1f}% average gap")
print(f"  5. Potential Annual Savings: ${annual_savings:,.2f}")

print(f"\n□ ACTIONABLE INSIGHTS:")

# Insight 1: Top overstocked A-items
top_overstocked_a = sku_stats[(sku_stats['ABC_Class'] == 'A') &
                             (sku_stats['StockStatus'] ==
'Overstocked')]
if len(top_overstocked_a) > 0:
    top_overstocked_a = top_overstocked_a.nlargest(3, 'ExcessValue')
    print(f"  1. Top 3 Overstocked A-items (High Priority):")
    for idx, row in top_overstocked_a.iterrows():
        print(f"     • {row['StockCode']}: {row['ExcessStock']:.0f}
units excess (${row['ExcessValue']:.2f})")

# Insight 2: Understocked high-service items
understocked_high_service = sku_stats[(sku_stats['StockStatus'] ==
'Understocked') &
                                     (sku_stats['ServiceLevelTarget']
> 0.95)]
if len(understocked_high_service) > 0:
    print(f"  2. High-Service Items Understocked:")
    for idx, row in understocked_high_service.head(3).iterrows():
        print(f"     • {row['StockCode']}: Service gap
{row['ServiceLevelGap']*100:.1f}%")

# Insight 3: Seasonal obsolescence risk
seasonal_high_risk = sku_stats[(sku_stats['ObsolescenceRisk'] ==
'High') &
                              (sku_stats['ObsolescenceType'] ==
'Seasonal')]
if len(seasonal_high_risk) > 0:
    print(f"  3. Seasonal Items with High Obsolescence Risk:")
    print(f"     • {len(seasonal_high_risk)} seasonal items
identified")
    print(f"     • Consider implementing clearance strategies")

print(f"\n□ OUTPUT FILES:")
print(f"  • {excel_file} - Complete analysis with multiple sheets")
print(f"  • {csv_file} - Optimized for Power BI dashboard")

print(f"\n□ PROJECT COMPLETED SUCCESSFULLY!")
```

Sample Recommendations

View top 5 recommendations for A-items with issues.

```
print("\n" + "=" * 70)
print("□ SAMPLE RECOMMENDATIONS")
print("=" * 70)

# Get top 5 recommendations (A items with issues)
recommendations = sku_stats[sku_stats['ABC_Class'] == 'A']
recommendations = recommendations[recommendations['StockStatus'] !=
'Optimal']

if len(recommendations) > 0:
    recommendations = recommendations.nlargest(5, 'ExcessValue')
    for idx, row in recommendations.iterrows():
        print(f"\n□ {row['StockCode']} - {row['Description']
[:50]}...")
        print(f"   Status: {row['StockStatus']}")
        print(f"   Current: {row['CurrentStock']:.0f} units")
        print(f"   Target: {row['ReorderPoint']:.0f} units")

        if row['StockStatus'] == 'Overstocked':
            print(f"   Excess: {row['ExcessStock']:.0f} units ($
{row['ExcessValue']:.2f})")
            print(f"   Action: Reduce inventory by
{row['ExcessStock']:.0f} units")
        else:
            print(f"   Deficit: {row['DeficitStock']:.0f} units ($
{row['DeficitValue']:.2f})")
            print(f"   Action: Increase inventory by
{row['DeficitStock']:.0f} units")

        print(f"   Impact: ${row['ExcessValue'] +
row['DeficitValue']:.2f}")
else:
    print("All A-items are optimally stocked! Great inventory
management.")
```