**📘 Complete Step-by-Step Guide: Air Traffic Clustering with Streamlit Deployment**

**🎯 Project Overview**

**Business Problem:** Airlines need to optimize operations and maximize profitability through data-driven passenger segmentation.

**Success Criteria:**

* 🎯 **ML Success:** Silhouette Score ≥ 0.7
* 📈 **Business Success:** 10-12% operational efficiency improvement
* 💰 **Economic Success:** 8% revenue increase

**📂 Project Structure**

air\_traffic\_clustering/

│

├── clustering\_pipeline.py # Backend ML pipeline

├── app.py # Streamlit deployment

├── requirements.txt # Python dependencies

├── AirTraffic\_Passenger\_Statistics.csv

└── README.md

**🔧 Setup Instructions**

**Step 1: Install Dependencies**

Create requirements.txt:

pandas==2.0.3

numpy==1.24.3

matplotlib==3.7.2

seaborn==0.12.2

scikit-learn==1.3.0

scipy==1.11.1

streamlit==1.28.0

plotly==5.17.0

sqlalchemy==2.0.20

pymysql==1.1.0

Install:

pip install -r requirements.txt

**Step 2: Run Streamlit App**

streamlit run app.py

**📖 CRISP-ML(Q) PHASES - DETAILED EXPLANATION**

**PHASE 1: BUSINESS & DATA UNDERSTANDING 🎯**

**What happens:**

* Load air traffic passenger data
* Understand business context
* Explore data structure
* Identify key variables

**Code Explanation:**

def load\_and\_explore\_data(self):

# Load CSV file containing air traffic data

self.df\_original = pd.read\_csv(self.data\_path)

**Line-by-line breakdown:**

1. pd.read\_csv() - Pandas function to read CSV files
2. self.data\_path - Path to the data file (provided during initialization)
3. self.df\_original - Stores the raw data in the class instance

# Display basic information

print(self.df\_original.head()) # Shows first 5 rows

print(self.df\_original.dtypes) # Shows data types of each column

print(self.df\_original.describe()) # Statistical summary

**What each line does:**

* .head() - Preview the data structure
* .dtypes - Check if columns are numeric/categorical/datetime
* .describe() - Get mean, median, std dev, min, max for numeric columns

# Check missing values

missing = self.df\_original.isnull().sum()

print(missing[missing > 0])

**Explanation:**

* .isnull() - Returns True for missing values
* .sum() - Counts missing values per column
* missing[missing > 0] - Shows only columns with missing data

**Business Insights Generated:**

if 'Operating Airline' in self.df\_original.columns:

print(f"Total Airlines: {self.df\_original['Operating Airline'].nunique()}")

* nunique() - Counts unique airlines (helps understand market diversity)
* This tells us how many different airlines we're analyzing

**PHASE 2: DATA PREPARATION 🔧**

**What happens:**

* Clean data (remove duplicates, handle missing values)
* Preprocess features (scaling, encoding)
* Apply dimensionality reduction (PCA)

**2.1 Data Cleaning**

def clean\_data(self):

# Create a copy to avoid modifying original data

self.df\_clean = self.df\_original.copy()

**Why .copy()?**

* Prevents accidental modification of original data
* Allows comparison before/after cleaning

# Remove duplicate rows

initial\_rows = len(self.df\_clean)

self.df\_clean = self.df\_clean.drop\_duplicates()

print(f"Removed {initial\_rows - len(self.df\_clean)} duplicate rows")

**Step-by-step:**

1. len(self.df\_clean) - Count rows before cleaning
2. .drop\_duplicates() - Removes exact duplicate rows
3. Calculate and print how many duplicates were removed

# Handle missing values in numeric columns

numeric\_cols = self.df\_clean.select\_dtypes(include=[np.number]).columns

for col in numeric\_cols:

if self.df\_clean[col].isnull().any():

self.df\_clean[col].fillna(self.df\_clean[col].median(), inplace=True)

**Explanation:**

* select\_dtypes(include=[np.number]) - Selects only numeric columns (Year, Passenger Count)
* .median() - Middle value (less affected by outliers than mean)
* .fillna() - Replaces missing values
* inplace=True - Modifies the dataframe directly

# Handle missing values in categorical columns

categorical\_cols = self.df\_clean.select\_dtypes(include=['object']).columns

for col in categorical\_cols:

if self.df\_clean[col].isnull().any():

self.df\_clean[col].fillna(self.df\_clean[col].mode()[0], inplace=True)

**Explanation:**

* select\_dtypes(include=['object']) - Selects text columns (Airline, Region, Terminal)
* .mode()[0] - Most frequent value
* Used for categorical data because median/mean don't make sense

# Treat outliers using winsorization

if 'Passenger Count' in self.df\_clean.columns:

lower = self.df\_clean['Passenger Count'].quantile(0.01) # 1st percentile

upper = self.df\_clean['Passenger Count'].quantile(0.99) # 99th percentile

self.df\_clean['Passenger Count'] = self.df\_clean['Passenger Count'].clip(lower, upper)

**What is Winsorization?**

* Caps extreme values instead of removing them
* .quantile(0.01) - Value below which 1% of data falls
* .clip(lower, upper) - Replaces values below lower with lower, values above upper with upper
* **Example:** If lower=100 and upper=50000, a value of 1,000,000 becomes 50,000

**2.2 Feature Preprocessing**

def preprocess\_data(self):

# Identify feature types

numeric\_cols = self.df\_clean.select\_dtypes(include=[np.number]).columns.tolist()

categorical\_cols = self.df\_clean.select\_dtypes(include=['object']).columns.tolist()

**Why separate numeric and categorical?**

* They need different transformations
* Numeric: Scaling (making values comparable)
* Categorical: Encoding (converting text to numbers)

# Create preprocessing pipeline

self.preprocessor = ColumnTransformer(

transformers=[

('num', MinMaxScaler(), numeric\_cols),

('cat', OneHotEncoder(handle\_unknown='ignore', sparse\_output=False), categorical\_cols)

]

)

**What is ColumnTransformer?**

* Applies different transformations to different columns
* ('num', MinMaxScaler(), numeric\_cols):
  + Name: 'num'
  + Transformer: MinMaxScaler
  + Columns: All numeric columns

**MinMaxScaler Explained:**

Original value: 15000 passengers

Min value in column: 100

Max value in column: 50000

Scaled value = (15000 - 100) / (50000 - 100) = 0.298

All values become 0 to 1

**OneHotEncoder Explained:**

Original: Operating Airline = "United Airlines"

After encoding:

Operating Airline\_United Airlines = 1

Operating Airline\_Delta = 0

Operating Airline\_American = 0

Each unique value becomes a separate column

# Apply transformations

processed\_array = self.preprocessor.fit\_transform(self.df\_clean)

**What happens here:**

1. .fit() - Learns the parameters (min/max for scaling, categories for encoding)
2. .transform() - Applies the transformations
3. .fit\_transform() - Does both in one step
4. Returns a numpy array (not pandas DataFrame)

# Get feature names after encoding

num\_features = numeric\_cols

cat\_features = self.preprocessor.named\_transformers\_['cat'].get\_feature\_names\_out(categorical\_cols)

all\_features = num\_features + list(cat\_features)

**Why do we need feature names?**

* After OneHotEncoding, we have many more columns
* Original: 'Operating Airline' (1 column)
* After: 'Operating Airline\_United', 'Operating Airline\_Delta', etc. (50+ columns)
* get\_feature\_names\_out() retrieves all the new column names

# Create DataFrame

self.df\_processed = pd.DataFrame(

processed\_array,

columns=all\_features,

index=self.df\_clean.index

)

**Converting back to DataFrame:**

* processed\_array - The numpy array with transformed data
* columns=all\_features - Assigns column names
* index=self.df\_clean.index - Keeps original row indices

**2.3 Dimensionality Reduction (PCA)**

def apply\_dimensionality\_reduction(self, variance\_ratio=0.95):

self.pca = PCA(n\_components=variance\_ratio)

pca\_array = self.pca.fit\_transform(self.df\_processed)

**What is PCA (Principal Component Analysis)?**

**Simple Analogy:** Imagine you have 188 features (after one-hot encoding). PCA finds the "main directions" where data varies the most.

**Example:**

* Feature 1: Passenger Count
* Feature 2: Year
* Feature 3-188: One-hot encoded airlines, regions, terminals

PCA might discover:

* Component 1: Mainly captures "traffic volume" (combines passenger count, major airlines)
* Component 2: Mainly captures "geographic patterns" (combines regions)
* Component 3: Mainly captures "temporal patterns" (combines year, month)

Instead of 188 features, we get ~20 components that capture 95% of information!

print(f"Dimensionality reduced:")

print(f" From: {self.df\_processed.shape[1]} features") # 188 features

print(f" To: {pca\_array.shape[1]} components") # 20 components

print(f" Variance explained: {self.pca.explained\_variance\_ratio\_.sum():.2%}") # 95%

**Why is this helpful?**

1. **Faster computation** - Less features = faster clustering
2. **Remove noise** - Minor variations removed
3. **Better visualization** - Can plot in 2D/3D
4. **Often better clustering** - Focuses on main patterns

**PHASE 3: MODEL BUILDING 🤖**

**What happens:**

* Test multiple clustering configurations
* Find optimal number of clusters
* Select best linkage method
* Compare original vs PCA data

**3.1 Testing Multiple Configurations**

def find\_optimal\_clusters(self, max\_clusters=10):

# Prepare data variations

datasets = {

'original': self.df\_processed, # All 188 features

'pca': self.apply\_dimensionality\_reduction() # ~20 components

}

**Why test both?**

* Original data: Complete information, but noisy
* PCA data: Compressed information, but cleaner
* Don't know which will cluster better until we test!

# Linkage methods to test

linkage\_methods = ['ward', 'complete', 'average']

**What are linkage methods?**

**1. Ward Linkage:**

* Minimizes variance within clusters
* Creates compact, equal-sized clusters
* **Best for:** Balanced segments
* **Example:** Creates 3 clusters of ~5000 records each

**2. Complete Linkage:**

* Minimizes maximum distance between clusters
* Creates well-separated clusters
* **Best for:** Distinct segments
* **Example:** Separates "international high-traffic" from "domestic low-traffic"

**3. Average Linkage:**

* Minimizes average distance between clusters
* Balanced between Ward and Complete
* **Best for:** Moderate separation

for data\_type, data in datasets.items(): # Loop through original and PCA

for method in linkage\_methods: # Loop through ward, complete, average

for k in range(2, max\_clusters + 1): # Test 2 to 10 clusters

**What this triple loop does:**

* Tests: 2 data types × 3 methods × 9 cluster counts = 54 models!
* Finds the best combination automatically

# Create and fit the model

model = AgglomerativeClustering(

n\_clusters=k,

metric='euclidean',

linkage=method

)

labels = model.fit\_predict(data)

**Line-by-line:**

1. n\_clusters=k - How many segments to create
2. metric='euclidean' - How to measure distance (straight-line distance)
3. linkage=method - Which linkage strategy to use
4. .fit\_predict(data) - Runs clustering and returns cluster labels

**What are cluster labels?**

Record 0: Cluster 2

Record 1: Cluster 0

Record 2: Cluster 2

Record 3: Cluster 1

...

Each record gets assigned to a cluster (0, 1, 2, etc.)

# Calculate silhouette score

score = metrics.silhouette\_score(data, labels)

**Silhouette Score Calculation:**

For each record:

1. Calculate average distance to other points in **same cluster** (a)
2. Calculate average distance to points in **nearest other cluster** (b)
3. Silhouette = (b - a) / max(a, b)

**Interpretation:**

* Score close to 1: Point is far from other clusters (good)
* Score close to 0: Point is on border between clusters (poor)
* Negative score: Point is probably in wrong cluster (very poor)

**Average all points = Overall Silhouette Score**

# Update best model if this is better

if score > self.best\_score:

self.best\_score = score

self.best\_model = model

self.best\_params = {

'data\_type': data\_type,

'method': method,

'n\_clusters': k,

'data': data

}

**What this does:**

* Keeps track of the best configuration found so far
* At the end, we have the optimal model!

**PHASE 4: MODEL EVALUATION 📊**

**4.1 Applying Final Model**

def evaluate\_model(self):

# Get final labels

data = self.best\_params['data']

labels = self.best\_model.labels\_

# Add cluster labels to original data

self.df\_clustered = self.df\_clean.copy()

self.df\_clustered['Cluster'] = labels

**Why add to original data?**

* We clustered on scaled/encoded data
* But we want to interpret results using original values
* Now each record has: Original airline name, passenger count, etc. + Cluster ID

**4.2 Cluster Analysis**

# Cluster Distribution

print("\n📈 Cluster Distribution:")

cluster\_counts = pd.Series(labels).value\_counts().sort\_index()

for cluster, count in cluster\_counts.items():

pct = (count / len(labels)) \* 100

print(f" Cluster {cluster}: {count:,} records ({pct:.1f}%)")

**Example Output:**

Cluster 0: 5,234 records (34.9%) # International, high-traffic

Cluster 1: 6,123 records (40.8%) # Domestic, medium-traffic

Cluster 2: 3,650 records (24.3%) # Regional, low-traffic

# Business Metrics by Cluster

if 'Passenger Count' in self.df\_clustered.columns:

for cluster in sorted(self.df\_clustered['Cluster'].unique()):

cluster\_data = self.df\_clustered[self.df\_clustered['Cluster'] == cluster]

total\_passengers = cluster\_data['Passenger Count'].sum()

avg\_passengers = cluster\_data['Passenger Count'].mean()

**What this reveals:**

* **Cluster 0:** Total: 45M passengers, Avg: 8,600 per flight → High-volume routes
* **Cluster 1:** Total: 30M passengers, Avg: 4,900 per flight → Medium routes
* **Cluster 2:** Total: 8M passengers, Avg: 2,200 per flight → Low-volume routes

# Top airlines

if 'Operating Airline' in cluster\_data.columns:

top\_airline = cluster\_data['Operating Airline'].mode().values[0]

print(f" Top Airline: {top\_airline}")

**Business Insight:**

* **Cluster 0:** United Airlines → Focus on international expansion
* **Cluster 1:** Southwest Airlines → Optimize domestic operations
* **Cluster 2:** Regional carriers → Consider consolidation

**PHASE 5: DEPLOYMENT (STREAMLIT) 🚀**

**5.1 Streamlit App Structure**

st.set\_page\_config(

page\_title="Air Traffic Clustering",

page\_icon="✈️",

layout="wide",

initial\_sidebar\_state="expanded"

)

**What this does:**

* page\_title - Browser tab title
* page\_icon - Emoji in browser tab
* layout="wide" - Uses full screen width
* initial\_sidebar\_state="expanded" - Sidebar visible on load

**5.2 File Upload**

uploaded\_file = st.file\_uploader(

"Upload Air Traffic CSV",

type=['csv'],

help="Upload your AirTraffic\_Passenger\_Statistics.csv file"

)

**How it works:**

1. User clicks "Browse files"
2. Selects CSV file from computer
3. File is uploaded to Streamlit server
4. uploaded\_file contains the file object

@st.cache\_data

def load\_data(file):

df = pd.read\_csv(file)

return df

**What is @st.cache\_data?**

* **Without caching:** Every time user interacts, file is re-read (slow!)
* **With caching:** File read once, stored in memory, reused (fast!)

**Example:**

* First load: Takes 2 seconds to read CSV
* User clicks a button: Data loaded instantly from cache
* User uploads new file: Cache cleared, new file read

**5.3 Interactive Parameters**

n\_clusters = st.slider(

"Number of Clusters",

min\_value=2,

max\_value=10,

value=3,

help="Select the number of segments to create"

)

**What this creates:**

* Interactive slider on screen
* User can drag to select 2-10 clusters
* Default value is 3
* When user changes slider, entire app re-runs with new value

linkage\_method = st.selectbox(

"Linkage Method",

options=['ward', 'complete', 'average'],

index=0,

help="Ward: Minimizes variance within clusters"

)

**Dropdown menu:**

* Shows 3 options
* index=0 - 'ward' selected by default
* User can change selection
* Help text shown on hover

**5.4 Running Clustering**

if st.button("🚀 Run Clustering Analysis", type="primary"):

with st.spinner("Building clustering model..."):

model, labels, silhouette\_score = apply\_clustering(

clustering\_data,

n\_clusters,

linkage\_method

)

# Store in session state

st.session\_state['model'] = model

st.session\_state['labels'] = labels

**What is session state?**

* **Problem:** Streamlit reruns entire script on every interaction
* **Solution:** Session state persists data between reruns
* **Example:** Store clustering results so they don't recompute every time

**Flow:**

1. User clicks "Run Clustering"
2. Button returns True
3. Code inside if block executes
4. st.spinner() shows "Building clustering model..." message
5. Clustering runs
6. Results stored in session state
7. Button click ends
8. Results remain in session state for visualizations

**5.5 Interactive Visualizations**

fig = px.bar(

x=cluster\_sizes.index,

y=cluster\_sizes.values,

labels={'x': 'Cluster', 'y': 'Number of Records'},

title='Records per Cluster',

color=cluster\_sizes.values,

color\_continuous\_scale='Blues'

)

st.plotly\_chart(fig, use\_container\_width=True)

**Why Plotly instead of Matplotlib?**

* **Interactive:** Hover to see values, zoom, pan
* **Better looking:** Modern, professional appearance
* **Responsive:** Adapts to screen size

**5.6 Download Results**

csv = df\_clustered.to\_csv(index=False)

st.download\_button(

label="📥 Download Clustered Data (CSV)",

data=csv,

file\_name="air\_traffic\_clustered.csv",

mime="text/csv",

use\_container\_width=True

)

**What happens:**

1. df\_clustered.to\_csv() - Converts DataFrame to CSV string
2. st.download\_button() - Creates download button
3. User clicks button
4. Browser downloads file with cluster assignments
5. File can be opened in Excel, imported to database, etc.

**PHASE 6: MONITORING & MAINTENANCE 🔍**

**Key Metrics to Track:**

# Track performance over time

st.metric(

"Silhouette Score",

f"{silhouette\_score:.4f}",

delta=f"{silhouette\_score - 0.7:.4f}"

)

**What to monitor:**

1. **Silhouette Score:** Is clustering quality maintained?
2. **Cluster Sizes:** Are clusters balanced or drifting?
3. **Business Metrics:** Are passenger counts changing per cluster?

**When to retrain:**

* New data collected (monthly/quarterly)
* Silhouette score drops below threshold
* Business patterns change (new airlines, routes)
* Cluster sizes become very imbalanced

**🎯 BUSINESS IMPACT MEASUREMENT**

**Operational Efficiency (10-12% improvement)**

**Before Clustering:**

* One-size-fits-all approach
* Resources spread evenly across all routes
* No targeted optimization

**After Clustering:**

* **Cluster 0 (High-volume):** Allocate more staff, larger aircraft
* **Cluster 1 (Medium):** Standard operations
* **Cluster 2 (Low-volume):** Smaller aircraft, reduced frequency

**Measurement:**

# Calculate efficiency gain

baseline\_cost\_per\_passenger = total\_cost / total\_passengers

optimized\_cost\_per\_passenger = (cluster\_cost) / total\_passengers

efficiency\_gain = (baseline - optimized) / baseline \* 100

**Revenue Increase (8%)**

**Strategies by Cluster:**

* **High-volume cluster:** Premium services, higher pricing
* **Medium cluster:** Balanced pricing, loyalty programs
* **Low-volume cluster:** Dynamic pricing, cost reduction

**Measurement:**

# Revenue per cluster

revenue\_by\_cluster = df\_clustered.groupby('Cluster').agg({

'Passenger Count': 'sum',

'Revenue': 'sum' # If available

})

revenue\_per\_passenger = revenue\_by\_cluster['Revenue'] / revenue\_by\_cluster['Passenger Count']

**🚀 DEPLOYMENT CHECKLIST**

**Before Deployment:**

* [ ] Test with sample data
* [ ] Verify all visualizations work
* [ ] Check file upload limits (Streamlit default: 200MB)
* [ ] Test download functionality
* [ ] Validate success criteria calculations

**Deployment Options:**

**1. Streamlit Cloud (Free):**

# Push to GitHub

git init

git add .

git commit -m "Initial commit"

git push origin main

# Deploy on streamlit.io

# Connect GitHub repo

# Select app.py

# Click Deploy

**2. Local Server:**

streamlit run app.py --server.port 8501

**3. Docker Container:**

FROM python:3.9

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

CMD ["streamlit", "run", "app.py"]

**📊 EXPECTED RESULTS**

**Typical Silhouette Scores:**

| **Scenario** | **Expected Score** | **Status** |
| --- | --- | --- |
| Poor clustering | 0.1 - 0.3 | ❌ Needs improvement |
| Fair clustering | 0.3 - 0.5 | ⚠️ Acceptable |
| Good clustering | 0.5 - 0.7 | ✅ Good |
| Excellent clustering | 0.7 - 1.0 | ✅ Excellent |

**Common Issues & Solutions:**

**Issue 1: Low Silhouette Score (< 0.3)**

* **Solution:** Enable PCA, try different linkage methods
* **Code:** use\_pca = True, test all 3 linkage methods

**Issue 2: Imbalanced Clusters**

* **Solution:** Adjust number of clusters, use ward linkage
* **Code:** Try n\_clusters = 4 or 5, linkage='ward'

**Issue 3: Too many features after encoding**

* **Solution:** Feature selection, higher PCA variance threshold
* **Code:** Remove low-variance features, pca\_variance = 0.90

**🎓 KEY TAKEAWAYS**

**Technical Skills Learned:**

1. ✅ End-to-end ML pipeline (CRISP-ML(Q))
2. ✅ Hierarchical clustering implementation
3. ✅ Data preprocessing (scaling, encoding)
4. ✅ Dimensionality reduction (PCA)
5. ✅ Model evaluation (Silhouette score)
6. ✅ Interactive web deployment (Streamlit)

**Business Skills Learned:**

1. ✅ Translating business problems to ML problems
2. ✅ Defining success criteria (ML, Business, Economic)
3. ✅ Generating actionable insights from clusters
4. ✅ Communicating results to stakeholders

**📝 FINAL NOTES**

**This complete solution provides:**

* ✅ Production-ready code with error handling
* ✅ Interactive web interface for non-technical users
* ✅ Comprehensive documentation
* ✅ Automated optimization (finds best parameters)
* ✅ Business-focused insights and recommendations
* ✅ Export functionality for further analysis

**Success Factors:**

1. **Data Quality:** Clean, complete data = better clusters
2. **Parameter Tuning:** Test multiple configurations
3. **Business Validation:** Technical results must make business sense
4. **User Experience:** Intuitive interface for adoption
5. **Monitoring:** Regular evaluation and retraining

**🔗 RESOURCES**

* **Streamlit Documentation:** https://docs.streamlit.io
* **Scikit-learn Clustering:** https://scikit-learn.org/stable/modules/clustering.html
* **CRISP-ML(Q):** https://ml-ops.org/content/crisp-ml

**🎉 Congratulations! You now have a complete, production-ready clustering solution with deployment!**