# Building End-to-End MLOps Pipelines with DVC

Complete MLOps Implementation Guide

A Comprehensive Guide to Building Production-Ready
ML Pipelines with DVC Automation, Experiment Tracking,
and AWS Integration

**Sujil S**

sujil9480@gmail.com

December 25, 2025

# Contents

# 1 Introduction to MLOps Pipelines

## 1.1 What is MLOps?

**MLOps (Machine Learning Operations)** is a set of practices that combines Machine Learning, DevOps, and Data Engineering to deploy and maintain ML systems in production reliably and efficiently.

**Core MLOps Principles**

- **Automation**: Automate ML workflows and deployments
- **Reproducibility**: Ensure experiments can be recreated
- **Versioning**: Track code, data, and models
- **Monitoring**: Track model performance in production
- **Collaboration**: Enable team collaboration on ML projects

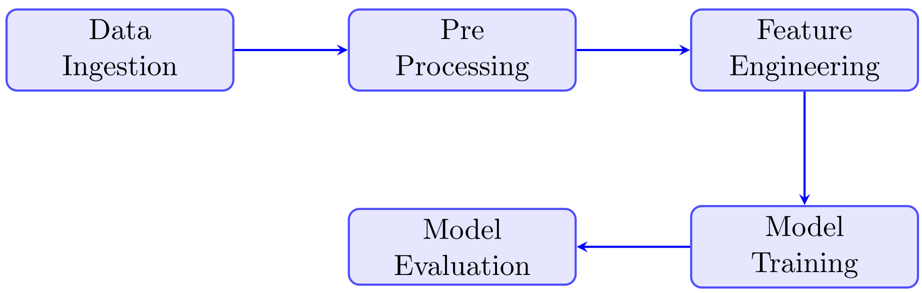
## 1.2 Course Agenda

This comprehensive guide covers the following key topics:

1. **End-to-End ML Pipeline**: Building complete ML workflows with logging and exception handling
2. **DVC Automation**: Automating pipelines using DVC YAML configuration
3. **Parameterization**: Adding configurable parameters for experimentation
4. **Experiment Tracking**: Using DVCLive for tracking experiments
5. **AWS Integration**: Setting up AWS S3 for data versioning

## 1.3 Basic ML Pipeline Components

A typical machine learning pipeline consists of the following stages:




### 1.3.1 Pipeline Stages Explained

1. **Data Ingestion**: Collecting and loading raw data from various sources
2. **Pre-Processing**: Cleaning, transforming, and preparing data
3. **Feature Engineering**: Creating meaningful features from raw data
4. **Model Training**: Training machine learning models on prepared data
5. **Model Evaluation**: Assessing model performance with metrics

## 1.4 Data Science vs MLOps Practice

| **Data Science Focus** | **MLOps Focus** |
|---|---|
| Pre-processing techniques | Coding best practices |
| Feature engineering methods | Robust pipeline architecture |
| Model hyperparameter tuning | Experiment tracking |
| Grid search and optimization | Automation with YAML |
| Algorithm selection | AWS/Cloud integration |
| Model performance | Reproducibility and versioning |

**Important Note**

MLOps extends data science by adding engineering rigor: version control, automation, testing, monitoring, and deployment practices that make ML systems production-ready.

# 2 Project Setup and Structure

## 2.1 Initial Repository Setup

### 2.1.1 Step 1: Create GitHub Repository

```
# On GitHub: Create new repository named "MLOPS-DVC-Project"
# Clone to local machine
git clone https://github.com/username/MLOPS-DVC-Project.git
cd MLOPS-DVC-Project
```

### 2.1.2 Step 2: Project Directory Structure

Create the following directory structure:

```
MLOPS-DVC-Project/
  Experiments/
    spam.csv
    mynotebook.ipynb
  src/
    Data_Ingestion.py
    Data_Pre_Processing.py
    Feature_Engineering.py
    Model_Building.py
    Model_Evaluation.py
  data/
    raw/
    interim/
    processed/
  models/
  reports/
  logs/
  .gitignore
  dvc.yaml
  params.yaml
  README.md
```

## 2.2 Initial Experiment Setup

The `Experiments/` folder contains the initial exploration work:

- **spam.csv**: Raw dataset for spam classification
- **mynotebook.ipynb**: Jupyter notebook with exploratory data analysis

**Purpose of Experiments Folder**

The experiments folder serves as a sandbox for:

- Exploratory data analysis (EDA)
- Trying different preprocessing techniques
- Testing various model architectures

- Validating assumptions before production code

## 2.3 Creating the Source Directory

```
# Create src directory
mkdir src
cd src
```

The `src/` directory will contain all production-ready Python scripts:

- Each component as a separate module
- Proper logging configuration
- Exception handling
- Modular and reusable code

## 2.4 Configuring .gitignore

Create a `.gitignore` file to exclude unnecessary files:

**Initial .gitignore Configuration**

```
# Data directories
data/
models/
reports/

# Python
__pycache__/
*.pyc
*.pyo
*.pyd
.Python
venv/
env/

# Logs
logs/
*.log

# Jupyter
.ipynb_checkpoints/

# IDE
.vscode/
.idea/

# OS
.DS_Store
Thumbs.db
```

```
# DVC
/dvc.lock
```

**Warning**

**Important**: Always add `data/`, `models/`, and `reports/` to .gitignore. DVC will handle versioning these large files, not Git.

# 3 Exploratory Data Analysis Phase

## 3.1 Understanding the Dataset

The spam classification dataset contains SMS messages labeled as spam or ham (not spam).

**Dataset Structure**

**Columns**:

- `v1`: Label (spam/ham)
- `v2`: SMS text message
- `Unnamed: 2, 3, 4`: Empty columns to be dropped

## 3.2 Notebook Workflow Overview

The `mynotebook.ipynb` follows this workflow:

1. **Import Libraries**: NumPy, Pandas, Matplotlib, NLTK
2. **Load Data**: Read CSV file
3. **Data Cleaning**: Drop unnecessary columns, rename columns
4. **Preprocessing**: Encode labels, remove duplicates
5. **Feature Engineering**: Text transformation, TF-IDF
6. **Model Training**: Train multiple classifiers
7. **Model Evaluation**: Compare accuracy and precision

## 3.3 Key Code Snippets from Notebook

### 3.3.1 Basic Imports and Setup

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

from wordcloud import WordCloud
import nltk
from nltk.corpus import stopwords

# Download NLTK data
nltk.download('stopwords')
nltk.download('punkt')
```

### 3.3.2 Data Loading and Cleaning

```python
# Load data
df = pd.read_csv('spam.csv')

# Drop unnecessary columns
```

```
df.drop(columns=['Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'],
        inplace=True)

# Rename columns
df.rename(columns={'v1': 'target', 'v2': 'text'}, inplace=True)

# Check duplicates
print(f"Duplicates: {df.duplicated().sum()}")
df = df.drop_duplicates(keep='first')
```

### 3.3.3 Text Transformation Function

```
from nltk.stem.porter import PorterStemmer
import string

ps = PorterStemmer()

def transform_text(text):
    # Lowercase transformation
    text = text.lower()

    # Tokenization
    text = nltk.word_tokenize(text)

    # Remove special characters
    text = [word for word in text if word.isalnum()]

    # Remove stopwords and punctuation
    text = [word for word in text
            if word not in stopwords.words('english')
            and word not in string.punctuation]

    # Stemming
    text = [ps.stem(word) for word in text]

    return " ".join(text)

# Apply transformation
df['transformed_text'] = df['text'].apply(transform_text)
```

### 3.3.4 Feature Engineering with TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Create TF-IDF features
tfidf = TfidfVectorizer(max_features=500)
X = tfidf.fit_transform(df['transformed_text']).toarray()
y = df['target'].values
```

### 3.3.5 Train-Test Split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
```

```
    X, y, test_size=0.20, random_state=2
)
```

## 3.4 Model Training and Comparison

### 3.4.1 Multiple Classifier Setup

```python
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import (RandomForestClassifier,
                              AdaBoostClassifier,
                              BaggingClassifier,
                              ExtraTreesClassifier,
                              GradientBoostingClassifier)
from xgboost import XGBClassifier

# Initialize classifiers
clfs = {
    'SVC': SVC(kernel="sigmoid", gamma=1.0),
    'KNN': KNeighborsClassifier(),
    'NB': MultinomialNB(),
    'DT': DecisionTreeClassifier(max_depth=5),
    'LR': LogisticRegression(solver='liblinear', penalty='l1'),
    'RF': RandomForestClassifier(n_estimators=50, random_state=2),
    'Adaboost': AdaBoostClassifier(n_estimators=50, random_state=2),
    'Bgc': BaggingClassifier(n_estimators=50, random_state=2),
    'ETC': ExtraTreesClassifier(n_estimators=50, random_state=2),
    'GBDT': GradientBoostingClassifier(n_estimators=50,
                                       random_state=2),
    'xgb': XGBClassifier(n_estimators=50, random_state=2)
}
```

### 3.4.2 Model Evaluation Function

```python
from sklearn.metrics import accuracy_score, precision_score

def train_classifier(clf, X_train, y_train, X_test, y_test):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    return accuracy, precision

# Train and evaluate all classifiers
for name, clf in clfs.items():
    accuracy, precision = train_classifier(
        clf, X_train, y_train, X_test, y_test
    )
    print(f"\nFor: {name}")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
```

**Important Note**

The notebook phase is exploratory. Once you identify the best approaches, refactor the code into production-ready modules in the `src/` directory with proper logging, error handling, and modularity.

# 4 Data Ingestion Module

## 4.1 Overview

The Data Ingestion module is responsible for:

- Loading raw data from source (URL, file, database)
- Basic preprocessing (dropping columns, renaming)
- Splitting data into train and test sets
- Saving processed data to designated directories

## 4.2 Complete Implementation

**src/Data_Ingestion.py**

```python
import pandas as pd
import os
from sklearn.model_selection import train_test_split
import logging

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Logging configuration
logger = logging.getLogger('Data_Ingestion')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Data_Ingestion.log')
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def load_data(data_url: str) -> pd.DataFrame:
    """Load data from a CSV file."""
    try:
        df = pd.read_csv(data_url)
        logger.debug('Data loaded from %s', data_url)
        return df
    except pd.errors.ParserError as e:
        logger.error('Failed to parse the CSV file: %s', e)
        raise
    except Exception as e:
```

```
        logger.error('Unexpected error while loading data: %s',
    e)
        raise


def preprocess_data(df: pd.DataFrame) -> pd.DataFrame:
    """Preprocess the data."""
    try:
        df.drop(columns=['Unnamed: 2', 'Unnamed: 3', 'Unnamed:
   4'],
                inplace=True)
        df.rename(columns={'v1': 'target', 'v2': 'text'},
                  inplace=True)
        logger.debug('Data preprocessing completed')
        return df
    except KeyError as e:
        logger.error('Missing column in dataframe: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error during preprocessing: %s
   ', e)
        raise


def save_data(train_data: pd.DataFrame,
              test_data: pd.DataFrame,
              data_path: str) -> None:
    """Save train and test datasets."""
    try:
        raw_data_path = os.path.join(data_path, 'raw')
        os.makedirs(raw_data_path, exist_ok=True)

        train_data.to_csv(
            os.path.join(raw_data_path, "train.csv"),
            index=False
        )
        test_data.to_csv(
            os.path.join(raw_data_path, "test.csv"),
            index=False
        )
        logger.debug('Train and test data saved to %s',
                     raw_data_path)
    except Exception as e:
        logger.error('Error while saving data: %s', e)
        raise


def main():
    try:
        test_size = 0.2
        data_url = 'https://raw.githubusercontent.com/...'

        df = load_data(data_url=data_url)
        final_df = preprocess_data(df)

        train_data, test_data = train_test_split(
```

```
            final_df, test_size=test_size, random_state=2
        )

        save_data(train_data, test_data, data_path='./data')
    except Exception as e:
        logger.error('Failed to complete data ingestion: %s', e)
        print(f"Error: {e}")


if __name__ == '__main__':
    main()
```

## 4.3 Key Components Explained

### 4.3.1 Logging Setup

- **Logger Name**: `Data_Ingestion`
- **Level**: `DEBUG` (captures all messages)
- **Handlers**:
  - Console handler: Outputs to terminal
  - File handler: Writes to `logs/Data_Ingestion.log`
- **Format**: Timestamp, logger name, level, message

### 4.3.2 Function Breakdown

1. **load_data()**:
   - Loads CSV from URL or file path
   - Handles parsing errors
   - Logs successful load
2. **preprocess_data()**:
   - Drops unnecessary columns
   - Renames columns to meaningful names
   - Handles missing column errors
3. **save_data()**:
   - Creates directory structure
   - Saves train and test CSVs
   - Logs save location
4. **main()**:
   - Orchestrates the entire process
   - Handles top-level exceptions
   - Entry point for script execution

## 4.4 Running the Module

```
# Navigate to project root
cd MLOPS-DVC-Project

# Run data ingestion
python src/Data_Ingestion.py
```

**Expected Output**:

- `logs/Data_Ingestion.log` created
- `data/raw/train.csv` created
- `data/raw/test.csv` created
- Console displays debug messages

**Important Note**

Always test each module individually before integrating into the DVC pipeline. This ensures each component works correctly in isolation.

# 5 Data Pre-Processing Module

## 5.1 Overview

The Data Pre-Processing module handles:

- Label encoding (converting text labels to numeric)
- Removing duplicate entries
- Text transformation (lowercasing, tokenization, stemming)
- Removing stopwords and punctuation

## 5.2 Complete Implementation

**src/Data_Pre_Processing.py - Part 1**

```python
import os
import logging
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from nltk.stem.porter import PorterStemmer
from nltk.corpus import stopwords
import string
import nltk

nltk.download('stopwords')
nltk.download('punkt')

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Setting up logger
logger = logging.getLogger('Data_Pre_Processing')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Data_Pre_Processing.log'
    )
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def transform_text(text):
    """
```

```
    Transform text: lowercase, tokenize, remove stopwords,
    punctuation, and stem.
    """
    ps = PorterStemmer()

    # Convert to lowercase
    text = text.lower()

    # Tokenize
    text = nltk.word_tokenize(text)

    # Remove non-alphanumeric tokens
    text = [word for word in text if word.isalnum()]

    # Remove stopwords and punctuation
    text = [word for word in text
            if word not in stopwords.words('english')
            and word not in string.punctuation]

    # Stem the words
    text = [ps.stem(word) for word in text]

    # Join back into string
    return " ".join(text)
```

**src/Data_Pre_Processing.py - Part 2**

```
def preprocess_df(df, text_column='text',
                  target_column='target'):
    """
    Preprocess DataFrame: encode target, remove duplicates,
    transform text.
    """
    try:
        logger.debug('Starting preprocessing for DataFrame')

        # Encode the target column
        encoder = LabelEncoder()
        df[target_column] = encoder.fit_transform(df[
    target_column])
        logger.debug('Target column encoded')

        # Remove duplicate rows
        df = df.drop_duplicates(keep='first')
        logger.debug('Duplicates removed')

        # Apply text transformation
        df.loc[:, text_column] = df[text_column].apply(
            transform_text
        )
        logger.debug('Text column transformed')

        return df
    except KeyError as e:
        logger.error('Column not found: %s', e)
```

```
            raise
    except Exception as e:
        logger.error('Error during text normalization: %s', e)
        raise


def main(text_column='text', target_column='target'):
    """
    Main function: load raw data, preprocess, save processed.
    """
    try:
        # Load data from data/raw
        train_data = pd.read_csv('./data/raw/train.csv')
        test_data = pd.read_csv('./data/raw/test.csv')
        logger.debug('Data loaded properly')

        # Transform the data
        train_processed = preprocess_df(train_data,
                                        text_column,
                                        target_column)
        test_processed = preprocess_df(test_data,
                                       text_column,
                                       target_column)

        # Store in data/interim
        data_path = os.path.join("./data", "interim")
        os.makedirs(data_path, exist_ok=True)

        train_processed.to_csv(
            os.path.join(data_path, "train_processed.csv"),
            index=False
        )
        test_processed.to_csv(
            os.path.join(data_path, "test_processed.csv"),
            index=False
        )

        logger.debug('Processed data saved to %s', data_path)
    except FileNotFoundError as e:
        logger.error('File not found: %s', e)
    except pd.errors.EmptyDataError as e:
        logger.error('No data: %s', e)
    except Exception as e:
        logger.error('Failed to complete data transformation: %
    s', e)
        print(f"Error: {e}")


if __name__ == '__main__':
    main()
```

## 5.3 Text Transformation Pipeline

The `transform_text()` function implements a comprehensive NLP preprocessing pipeline:

1. **Lowercasing**: Converts all text to lowercase

- "Hello World" → "hello world"

2. **Tokenization**: Splits text into individual words
   - "hello world" → ["hello", "world"]
3. **Alphanumeric Filtering**: Removes special characters
   - ["hello", "world", "!"] → ["hello", "world"]
4. **Stopword Removal**: Removes common words
   - ["the", "quick", "fox"] → ["quick", "fox"]
5. **Stemming**: Reduces words to root form
   - ["running", "runs", "ran"] → ["run", "run", "run"]

## 5.4 Running the Module

```
# Run data preprocessing
python src/Data_Pre_Processing.py
```

**Expected Output**:

- `logs/Data_Pre_Processing.log` created
- `data/interim/train_processed.csv` created
- `data/interim/test_processed.csv` created

**Why Interim Directory?**

The data flow follows this structure:

- `data/raw/`: Original, untouched data
- `data/interim/`: Partially processed data
- `data/processed/`: Final features ready for modeling

This separation allows tracking transformations at each stage.

# 6 Feature Engineering Module

## 6.1 Overview

The Feature Engineering module:

- Applies TF-IDF (Term Frequency-Inverse Document Frequency)
- Converts text data into numerical feature vectors
- Creates fixed-size feature matrices for modeling
- Saves processed features with labels

## 6.2 Complete Implementation

**src/Feature_Engineering.py**

```python
import pandas as pd
import os
from sklearn.feature_extraction.text import TfidfVectorizer
import logging

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Logging configuration
logger = logging.getLogger('Feature_Engineering')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Feature_Engineering.log'
    )
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def load_data(file_path: str) -> pd.DataFrame:
    """Load data from a CSV file."""
    try:
        df = pd.read_csv(file_path)
        df.fillna('', inplace=True)
        logger.debug('Data loaded and NaNs filled from %s',
                     file_path)
        return df
    except pd.errors.ParserError as e:
```

```
        logger.error('Failed to parse the CSV file: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error while loading data: %s', e)
        raise


def apply_tfidf(train_data: pd.DataFrame,
                test_data: pd.DataFrame,
                max_features: int) -> tuple:
    """Apply TF-IDF to the data."""
    try:
        vectorizer = TfidfVectorizer(max_features=max_features)

        X_train = train_data['text'].values
        y_train = train_data['target'].values
        X_test = test_data['text'].values
        y_test = test_data['target'].values

        X_train_tfidf = vectorizer.fit_transform(X_train)
        X_test_tfidf = vectorizer.transform(X_test)

        train_df = pd.DataFrame(X_train_tfidf.toarray())
        train_df['label'] = y_train

        test_df = pd.DataFrame(X_test_tfidf.toarray())
        test_df['label'] = y_test

        logger.debug('TF-IDF applied and data transformed')
        return train_df, test_df
    except Exception as e:
        logger.error('Error during TF-IDF transformation: %s', e)
        raise


def save_data(df: pd.DataFrame, file_path: str) -> None:
    """Save the dataframe to a CSV file."""
    try:
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        df.to_csv(file_path, index=False)
        logger.debug('Data saved to %s', file_path)
    except Exception as e:
        logger.error('Unexpected error while saving data: %s', e)
        raise


def main():
    try:
        max_features = 50

        train_data = load_data('./data/interim/train_processed.csv')
        test_data = load_data('./data/interim/test_processed.
```

```
        csv')

            train_df, test_df = apply_tfidf(train_data, test_data,
                                            max_features)

            save_data(train_df,
                      os.path.join("./data", "processed",
                                   "train_tfidf.csv"))
            save_data(test_df,
                      os.path.join("./data", "processed",
                                   "test_tfidf.csv"))
        except Exception as e:
            logger.error('Failed to complete feature engineering: %
        s', e)
            print(f"Error: {e}")


if __name__ == '__main__':
        main()
```

## 6.3 Understanding TF-IDF

**TF-IDF (Term Frequency-Inverse Document Frequency)** measures the importance of a word in a document relative to a collection of documents.

**TF-IDF Formula**

**TF (Term Frequency)**: How often a word appears in a document

$$TF(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

**IDF (Inverse Document Frequency)**: How rare/common a word is across all documents

$$IDF(t) = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing term } t}\right)$$

**TF-IDF Score**:

$$\text{TF-IDF}(t, d) = TF(t, d) \times IDF(t)$$

### 6.3.1 Why TF-IDF?

- **Reduces weight of common words**: Words like "the", "is" get lower scores
- **Increases weight of distinctive words**: Rare words get higher scores
- **Creates fixed-size vectors**: Essential for machine learning models
- **Captures semantic importance**: Better than simple word counts

## 6.4 max_features Parameter

The `max_features` parameter controls dimensionality:

- **max_features=50**: Creates 50 features (top 50 important words)

- **Higher values**: More features, more information, higher complexity
- **Lower values**: Fewer features, less information, faster training

**Important Note**

The `max_features` value is a hyperparameter that can be tuned. We'll later make this configurable via `params.yaml` for easy experimentation.

### 6.5 Running the Module

```
# Run feature engineering
python src/Feature_Engineering.py
```

**Expected Output**:

- `logs/Feature_Engineering.log` created
- `data/processed/train_tfidf.csv` created (50 features + 1 label column)
- `data/processed/test_tfidf.csv` created (50 features + 1 label column)

# 7 Model Building Module

## 7.1 Overview

The Model Building module:

- Loads processed feature data
- Trains a RandomForest classifier
- Saves the trained model as a pickle file
- Logs the training process

## 7.2 Complete Implementation

**src/Model_Building.py**

```python
import os
import numpy as np
import pandas as pd
import pickle
import logging
from sklearn.ensemble import RandomForestClassifier

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Logging configuration
logger = logging.getLogger('Model_Building')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Model_Building.log')
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def load_data(file_path: str) -> pd.DataFrame:
    """Load data from a CSV file."""
    try:
        df = pd.read_csv(file_path)
        logger.debug('Data loaded from %s with shape %s',
                     file_path, df.shape)
        return df
    except pd.errors.ParserError as e:
```

```python
        logger.error('Failed to parse the CSV file: %s', e)
        raise
    except FileNotFoundError as e:
        logger.error('File not found: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error while loading data: %s', e)
        raise


def train_model(X_train: np.ndarray,
                y_train: np.ndarray,
                params: dict) -> RandomForestClassifier:
    """Train the RandomForest model."""
    try:
        if X_train.shape[0] != y_train.shape[0]:
            raise ValueError(
                "Number of samples in X_train and y_train must match"
            )

        logger.debug('Initializing RandomForest with params: %s',
                     params)
        clf = RandomForestClassifier(
            n_estimators=params['n_estimators'],
            random_state=params['random_state']
        )

        logger.debug('Model training started with %d samples',
                     X_train.shape[0])
        clf.fit(X_train, y_train)
        logger.debug('Model training completed')

        return clf
    except ValueError as e:
        logger.error('ValueError during model training: %s', e)
        raise
    except Exception as e:
        logger.error('Error during model training: %s', e)
        raise


def save_model(model, file_path: str) -> None:
    """Save the trained model to a file."""
    try:
        os.makedirs(os.path.dirname(file_path), exist_ok=True)

        with open(file_path, 'wb') as file:
            pickle.dump(model, file)
        logger.debug('Model saved to %s', file_path)
    except FileNotFoundError as e:
        logger.error('File path not found: %s', e)
        raise
    except Exception as e:
```

```
            logger.error('Error while saving model: %s', e)
            raise


def main():
    try:
        params = {'n_estimators': 25, 'random_state': 2}

        train_data = load_data('./data/processed/train_tfidf.
csv')
        X_train = train_data.iloc[:, :-1].values
        y_train = train_data.iloc[:, -1].values

        clf = train_model(X_train, y_train, params)

        model_save_path = 'models/model.pkl'
        save_model(clf, model_save_path)
    except Exception as e:
        logger.error('Failed to complete model building: %s', e
)
        print(f"Error: {e}")


if __name__ == '__main__':
    main()
```

## 7.3 Model Selection: RandomForest

**Why RandomForest?**

- **Robust**: Handles overfitting well through ensemble learning
- **Feature Importance**: Provides insights into which features matter
- **No Feature Scaling**: Works well without normalization
- **Handles Imbalanced Data**: Good for spam/ham classification
- **Parallel Training**: Can utilize multiple CPU cores

## 7.4 Hyperparameters Explained

- **n_estimators**: Number of decision trees in the forest
  - More trees = Better performance but slower training
  - Typical range: 50-200
- **random_state**: Ensures reproducibility
  - Same value = Same results across runs
  - Essential for experiment tracking

## 7.5 Model Persistence with Pickle

**Pickle** serializes Python objects to disk:

- **Saves entire model**: Including learned parameters
- **Quick loading**: No need to retrain
- **Version control friendly**: When combined with DVC

**Warning**

**Important .gitignore Rule**:

Add `models/` to .gitignore! Git should NOT track model files (they're binary and large). DVC will handle model versioning.

```
# In .gitignore
models/
```

## 7.6 Running the Module

```
# Run model building
python src/Model_Building.py
```

**Expected Output**:

- `logs/Model_Building.log` created
- `models/model.pkl` created
- Console displays training progress

# 8 Model Evaluation Module

## 8.1 Overview

The Model Evaluation module:

- Loads the trained model
- Loads test data
- Makes predictions
- Calculates evaluation metrics
- Saves metrics to JSON file

## 8.2 Complete Implementation

**src/Model_Evaluation.py**

```python
import os
import numpy as np
import pandas as pd
import pickle
import json
from sklearn.metrics import (accuracy_score, precision_score,
                             recall_score, roc_auc_score)
import logging

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Logging configuration
logger = logging.getLogger('Model_Evaluation')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Model_Evaluation.log')
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def load_model(file_path: str):
    """Load the trained model from a file."""
    try:
        with open(file_path, 'rb') as file:
```

```python
                model = pickle.load(file)
            logger.debug('Model loaded from %s', file_path)
            return model
        except FileNotFoundError:
            logger.error('File not found: %s', file_path)
            raise
        except Exception as e:
            logger.error('Unexpected error while loading model: %s'
    , e)
            raise


def load_data(file_path: str) -> pd.DataFrame:
    """Load data from a CSV file."""
    try:
        df = pd.read_csv(file_path)
        logger.debug('Data loaded from %s', file_path)
        return df
    except pd.errors.ParserError as e:
        logger.error('Failed to parse the CSV file: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error while loading data: %s',
    e)
        raise


def evaluate_model(clf, X_test: np.ndarray,
                   y_test: np.ndarray) -> dict:
    """Evaluate the model and return metrics."""
    try:
        y_pred = clf.predict(X_test)
        y_pred_proba = clf.predict_proba(X_test)[:, 1]

        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred)
        recall = recall_score(y_test, y_pred)
        auc = roc_auc_score(y_test, y_pred_proba)

        metrics_dict = {
            'accuracy': accuracy,
            'precision': precision,
            'recall': recall,
            'auc': auc
        }
        logger.debug('Model evaluation metrics calculated')
        return metrics_dict
    except Exception as e:
        logger.error('Error during model evaluation: %s', e)
        raise


def save_metrics(metrics: dict, file_path: str) -> None:
    """Save the evaluation metrics to a JSON file."""
    try:
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
```

```python

        with open(file_path, 'w') as file:
            json.dump(metrics, file, indent=4)
        logger.debug('Metrics saved to %s', file_path)
    except Exception as e:
        logger.error('Error while saving metrics: %s', e)
        raise


def main():
    try:
        clf = load_model('./models/model.pkl')
        test_data = load_data('./data/processed/test_tfidf.csv')

        X_test = test_data.iloc[:, :-1].values
        y_test = test_data.iloc[:, -1].values

        metrics = evaluate_model(clf, X_test, y_test)

        save_metrics(metrics, 'reports/metrics.json')
    except Exception as e:
        logger.error('Failed to complete model evaluation: %s', e)
        print(f"Error: {e}")


if __name__ == '__main__':
    main()
```

## 8.3 Evaluation Metrics Explained

### 8.3.1 Accuracy

Percentage of correct predictions:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

### 8.3.2 Precision

Of all predicted spam, how many are actually spam:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

### 8.3.3 Recall (Sensitivity)

Of all actual spam, how many did we catch:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

### 8.3.4 AUC (Area Under ROC Curve)

Measures model's ability to distinguish between classes:

- AUC = 1.0: Perfect classifier
- AUC = 0.5: Random guessing
- Higher is better

## 8.4 Why These Metrics?

| Metric | When to Prioritize |
|---|---|
| Accuracy | Balanced datasets, overall performance |
| Precision | When false positives are costly (e.g., marking legitimate emails as spam) |
| Recall | When false negatives are costly (e.g., missing actual spam) |
| AUC | Overall model discrimination ability, imbalanced datasets |

## 8.5 Metrics JSON Format

**reports/metrics.json Example**

```
{
    "accuracy": 0.9732,
    "precision": 0.9821,
    "recall": 0.9456,
    "auc": 0.9887
}
```

**Warning**

**Add reports/ to .gitignore**:

```
# In .gitignore
reports/
```

DVC will track metrics files, not Git.

## 8.6 Running the Module

```
# Run model evaluation
python src/Model_Evaluation.py
```

**Expected Output**:

- `logs/Model_Evaluation.log` created
- `reports/metrics.json` created with all metrics
- Console displays evaluation progress

# 9 Testing the Complete Pipeline Manually

## 9.1 Running All Components Sequentially

Before automating with DVC, verify each component works correctly:

```
# Step 1: Data Ingestion
python src/Data_Ingestion.py

# Step 2: Data Pre-Processing
python src/Data_Pre_Processing.py

# Step 3: Feature Engineering
python src/Feature_Engineering.py

# Step 4: Model Building
python src/Model_Building.py

# Step 5: Model Evaluation
python src/Model_Evaluation.py
```

## 9.2 Expected Directory Structure After Execution

```
MLOPS-DVC-Project/
  data/
    raw/
      train.csv
      test.csv
    interim/
      train_processed.csv
      test_processed.csv
    processed/
      train_tfidf.csv
      test_tfidf.csv
  models/
    model.pkl
  reports/
    metrics.json
  logs/
    Data_Ingestion.log
    Data_Pre_Processing.log
    Feature_Engineering.log
    Model_Building.log
    Model_Evaluation.log
```

## 9.3 Verification Checklist

1. **Data Files**: All CSV files created in correct directories
2. **Model File**: model.pkl exists in models/
3. **Metrics File**: metrics.json contains all 4 metrics

4. **Log Files**: Each component has its log file
5. **No Errors**: Check log files for any ERROR messages

## 9.4 Initial Git Commit

```
# Check .gitignore is properly configured
cat .gitignore

# Stage all source code
git add .gitignore
git add src/
git add Experiments/
git add README.md

# Commit
git commit -m "Initial commit: Add all pipeline components"

# Push to remote
git push origin main
```

**Important Note**

At this point, Git tracks:

- Source code (`src/`)
- Experiments (`Experiments/`)
- Configuration files (`.gitignore`)

Git does NOT track:

- Data files (`data/`)
- Models (`models/`)
- Reports (`reports/`)
- Logs (`logs/`)

DVC will handle these large files in the next section.

# 10 Setting Up DVC Pipeline (Without Parameters)

## 10.1 Understanding DVC Pipelines

**DVC Pipelines** automate ML workflows by:

- Defining stages and dependencies
- Tracking inputs and outputs
- Automatically detecting changes
- Running only what's necessary
- Creating reproducible workflows

## 10.2 Initialize DVC

```
# Initialize DVC in the project
dvc init
```

This creates:

- `.dvc/` directory: DVC configuration and cache
- `.dvcignore`: Similar to .gitignore for DVC
- Updates `.gitignore` to exclude DVC cache

## 10.3 Creating dvc.yaml File

The `dvc.yaml` file defines the entire pipeline:

**dvc.yaml (Basic Version)**

```
stages:
  data_ingestion:
    cmd: python src/Data_Ingestion.py
    deps:
    - src/Data_Ingestion.py
    outs:
    - data/raw

  data_preprocessing:
    cmd: python src/Data_Pre_Processing.py
    deps:
    - data/raw
    - src/Data_Pre_Processing.py
    outs:
    - data/interim

  feature_engineering:
    cmd: python src/Feature_Engineering.py
    deps:
    - data/interim
    - src/Feature_Engineering.py
    outs:
    - data/processed
```

```

  model_building:
    cmd: python src/Model_Building.py
    deps:
    - data/processed
    - src/Model_Building.py
    outs:
    - models/model.pkl

  model_evaluation:
    cmd: python src/Model_Evaluation.py
    deps:
    - models/model.pkl
    - src/Model_Evaluation.py
    metrics:
    - reports/metrics.json
```

## 10.4 Understanding dvc.yaml Structure

### 10.4.1 Stage Components

Each stage has:

1. **cmd**: Command to execute

```
cmd: python src/Data_Ingestion.py

```

2. **deps**: Dependencies (files that, if changed, trigger re-run)

```
deps:
- src/Data_Ingestion.py
- data/raw

```

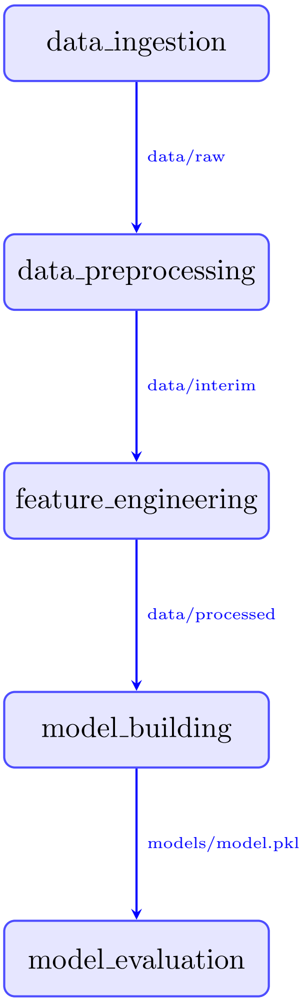
3. **outs**: Outputs (files/directories created by stage)

```
outs:
- data/interim

```

4. **metrics**: Metric files (special output for tracking)

```
metrics:
- reports/metrics.json

```

## 10.5 Stage Dependency Chain




## 10.6 Running the DVC Pipeline

```
# Run the entire pipeline
dvc repro
```

**What happens during** `dvc repro`:

1. DVC reads `dvc.yaml`
2. Checks if dependencies changed
3. Runs necessary stages in order
4. Tracks all outputs
5. Creates `dvc.lock` file

## 10.7 Understanding dvc.lock File

After running `dvc repro`, DVC creates `dvc.lock`:

**What is dvc.lock?**

`dvc.lock` is analogous to `package-lock.json` or `requirements.txt.lock`:

- **Locks exact versions**: Stores MD5 hashes of all files
- **Ensures reproducibility**: Same lock = same results
- **Tracked by Git**: Commit this file!
- **Auto-generated**: Never edit manually

**Sample dvc.lock Entry**

```
data_ingestion:
  cmd: python src/Data_Ingestion.py
  deps:
  - path: src/Data_Ingestion.py
    md5: a1b2c3d4e5f6g7h8i9j0
  outs:
  - path: data/raw
    md5: x1y2z3a4b5c6d7e8f9g0
```

## 10.8 Visualizing the Pipeline

```
# Visualize pipeline as a DAG (Directed Acyclic Graph)
dvc dag
```

**Example Output**:

```
    +---------------------+
    | data_ingestion      |
    +---------------------+
              *
              *
              *
  +-----------------------+
  | data_preprocessing    |
  +-----------------------+
              *
              *
              *
  +-----------------------+
  | feature_engineering   |
  +-----------------------+
              *
              *
              *
    +--------------------+
    | model_building     |
    +--------------------+
              *
              *
```

```
          *
+--------------------+
| model_evaluation   |
+--------------------+
```

## 10.9 Key DVC Concepts

### 10.9.1 The .dvc Directory

- **.dvc/cache/**: Stores all versions of tracked data
- **.dvc/config**: DVC configuration (remotes, etc.)
- **.dvc/.gitignore**: Prevents Git from tracking cache

**Cache Structure**

The `.dvc/cache/` contains:

- **Both tokens AND data**: Unlike remote storage
- **All versions**: Every experiment's data
- **Content-addressable**: Files named by MD5 hash
- **Deduplicated**: Identical files stored once

### 10.9.2 dvc.yaml vs dvc.lock vs .dvc files

| File | Purpose |
|---|---|
| dvc.yaml | Defines pipeline stages, dependencies, outputs |
| dvc.lock | Locks exact file versions (MD5 hashes) for reproducibility |
| <name>.dvc | Created by `dvc add`, tracks individual files/folders |

**Warning**

**Key Difference**:

- **dvc add**: Manually track files, creates `<name>.dvc`
- **dvc.yaml outputs**: Automatically tracked, uses `dvc.lock`

With pipelines, you DON'T need `dvc add` for outputs!

## 10.10 Intelligent Re-Running

DVC only re-runs changed stages:

```
# If you run dvc repro again without changes
dvc repro

# Output:
Stage 'data_ingestion' didn't change, skipping
Stage 'data_preprocessing' didn't change, skipping
Stage 'feature_engineering' didn't change, skipping
```

```
Stage 'model_building' didn't change, skipping
Stage 'model_evaluation' didn't change, skipping
Data and pipelines are up to date.
```

**When does DVC re-run a stage?**

- **Code changes**: Modify `src/` files
- **Dependency changes**: Input data changes
- **Parameter changes**: (covered in next section)
- **Manual force**: `dvc repro --force`

### 10.11 Committing Pipeline to Git

```
# Stage DVC files
git add dvc.yaml dvc.lock .dvc/.gitignore .dvc/config

# Commit
git commit -m "Add DVC pipeline automation"

# Push
git push origin main
```

**Important Note**

**What's in Git vs DVC now?**
**Git tracks**:

- Source code (`src/`)
- Pipeline definition (`dvc.yaml`)
- File version locks (`dvc.lock`)
- DVC configuration (`.dvc/config`)

**DVC tracks** (in `.dvc/cache/`):

- Data (`data/`)
- Models (`models/`)
- Reports (`reports/`)

# 11 Adding Configurable Parameters

## 11.1 Why Parameterize?

Hardcoded values in code are problematic:

- **Not experiment-friendly**: Changing values requires code edits
- **Not version-controlled**: Hard to track what values were used
- **Not DVC-aware**: DVC can't detect parameter changes
- **Not reproducible**: Unclear what parameters produced results

**Solution**: Centralize parameters in `params.yaml`

## 11.2 Creating params.yaml

**params.yaml**

```yaml
data_ingestion:
  test_size: 0.15

feature_engineering:
  max_features: 45

model_building:
  n_estimators: 20
  random_state: 2
```

## 11.3 Updating dvc.yaml with Parameters

**dvc.yaml (With Parameters)**

```yaml
stages:
  data_ingestion:
    cmd: python src/Data_Ingestion.py
    deps:
    - src/Data_Ingestion.py
    params:
    - data_ingestion.test_size
    outs:
    - data/raw

  data_preprocessing:
    cmd: python src/Data_Pre_Processing.py
    deps:
    - data/raw
    - src/Data_Pre_Processing.py
    outs:
    - data/interim

  feature_engineering:
    cmd: python src/Feature_Engineering.py
    deps:
```

```yaml
    - data/interim
    - src/Feature_Engineering.py
    params:
    - feature_engineering.max_features
    outs:
    - data/processed

  model_building:
    cmd: python src/Model_Building.py
    deps:
    - data/processed
    - src/Model_Building.py
    params:
    - model_building.n_estimators
    - model_building.random_state
    outs:
    - models/model.pkl

  model_evaluation:
    cmd: python src/Model_Evaluation.py
    deps:
    - models/model.pkl
    - src/Model_Evaluation.py
    metrics:
    - reports/metrics.json
```

## 11.4 Loading Parameters in Python Code

Add a `load_params()` function to each module:

**Parameter Loading Function**

```python
import yaml

def load_params(params_path: str) -> dict:
    """Load parameters from a YAML file."""
    try:
        with open(params_path, 'r') as file:
            params = yaml.safe_load(file)
        logger.debug('Parameters retrieved from %s', params_path)
        return params
    except FileNotFoundError:
        logger.error('File not found: %s', params_path)
        raise
    except yaml.YAMLError as e:
        logger.error('YAML error: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error: %s', e)
        raise
```

## 11.5 Updated Module Implementations

### 11.5.1 Data_Ingestion.py Changes

```
# Add at top
import yaml

# Add load_params function (shown above)

# Update main()
def main():
    try:
        # Load parameters
        params = load_params(params_path='params.yaml')
        test_size = params['data_ingestion']['test_size']

        data_url = 'https://raw.githubusercontent.com/...'
        df = load_data(data_url=data_url)
        final_df = preprocess_data(df)

        # Use parameterized test_size
        train_data, test_data = train_test_split(
            final_df, test_size=test_size, random_state=2
        )

        save_data(train_data, test_data, data_path='./data')
    except Exception as e:
        logger.error('Failed to complete data ingestion: %s', e)
        print(f"Error: {e}")
```

### 11.5.2 Feature_Engineering.py Changes

```
# Add at top
import yaml

# Add load_params function

# Update main()
def main():
    try:
        # Load parameters
        params = load_params(params_path='params.yaml')
        max_features = params['feature_engineering']['max_features']

        train_data = load_data('./data/interim/train_processed.csv')
        test_data = load_data('./data/interim/test_processed.csv')

        # Use parameterized max_features
        train_df, test_df = apply_tfidf(train_data, test_data,
                                        max_features)

        save_data(train_df, './data/processed/train_tfidf.csv')
        save_data(test_df, './data/processed/test_tfidf.csv')
    except Exception as e:
        logger.error('Failed to complete feature engineering: %s', e)
        print(f"Error: {e}")
```

### 11.5.3 Model_Building.py Changes

```python
# Add at top
import yaml

# Add load_params function

# Update main()
def main():
    try:
        # Load parameters
        params = load_params('params.yaml')['model_building']

        train_data = load_data('./data/processed/train_tfidf.csv')
        X_train = train_data.iloc[:, :-1].values
        y_train = train_data.iloc[:, -1].values

        # Use parameterized model parameters
        clf = train_model(X_train, y_train, params)

        model_save_path = 'models/model.pkl'
        save_model(clf, model_save_path)
    except Exception as e:
        logger.error('Failed to complete model building: %s', e)
        print(f"Error: {e}")
```

## 11.6 Testing Parameterized Pipeline

```
# Run pipeline with new parameters
dvc repro
```

**DVC now detects parameter changes**:

- If you change `params.yaml`, DVC knows
- Only affected stages re-run
- Parameters tracked in `dvc.lock`

## 11.7 Benefits of Parameterization

1. **Easy Experimentation**:

```yaml
# Change this:
model_building:
  n_estimators: 20

# To this:
model_building:
  n_estimators: 50

# Then run: dvc repro

```

2. **DVC Awareness**: DVC detects changes and re-runs appropriate stages
3. **Version Control**: Parameters tracked with code in Git

4. **Reproducibility**: `params.yaml + dvc.lock` = exact reproduction
5. **Documentation**: Clear record of hyperparameters used

## 11.8 Commit Parameterized Pipeline

```
# Stage changes
git add params.yaml dvc.yaml dvc.lock src/

# Commit
git commit -m "Add parameter configuration with params.yaml"

# Push
git push origin main
```

**Important Note**

**Important Behavior**:
If you run `dvc repro` without any changes:

```
Stage 'data_ingestion' didn't change, skipping
Stage 'data_preprocessing' didn't change, skipping
Stage 'feature_engineering' didn't change, skipping
Stage 'model_building' didn't change, skipping
Stage 'model_evaluation' didn't change, skipping
Data and pipelines are up to date.
```

DVC is smart enough to skip unnecessary work!

# 12 Experiment Tracking with DVCLive

## 12.1 What is DVCLive?

**DVCLive** is DVC's experiment tracking library:

- **Logs metrics**: Accuracy, loss, custom metrics
- **Logs parameters**: Hyperparameters used
- **Creates plots**: Training curves, confusion matrices
- **Integrates with DVC**: Automatic experiment versioning
- **Lightweight**: No external services needed

## 12.2 Installation

```
pip install dvclive
```

## 12.3 Updating Model_Evaluation.py

Add DVCLive integration to track experiments:

**Model_Evaluation.py with DVCLive**

```python
import os
import numpy as np
import pandas as pd
import pickle
import json
from sklearn.metrics import (accuracy_score, precision_score,
                             recall_score, roc_auc_score)
import logging
from dvclive import Live  # Add DVCLive
import yaml  # Add YAML

# ... (previous logging setup remains same) ...

# Add load_params function
def load_params(params_path: str) -> dict:
    """Load parameters from a YAML file."""
    try:
        with open(params_path, 'r') as file:
            params = yaml.safe_load(file)
        logger.debug('Parameters retrieved from %s',
    params_path)
        return params
    except FileNotFoundError:
        logger.error('File not found: %s', params_path)
        raise
    except yaml.YAMLError as e:
        logger.error('YAML error: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error: %s', e)
        raise
```

```python

# ... (other functions remain same) ...

def main():
    try:
        # Load parameters
        params = load_params('params.yaml')

        clf = load_model('./models/model.pkl')
        test_data = load_data('./data/processed/test_tfidf.csv')

        X_test = test_data.iloc[:, :-1].values
        y_test = test_data.iloc[:, -1].values

        metrics = evaluate_model(clf, X_test, y_test)
        save_metrics(metrics, 'reports/metrics.json')

        # DVCLive logging
        with Live(save_dvc_exp=True) as live:
            # Log metrics
            live.log_metric('accuracy', metrics['accuracy'])
            live.log_metric('precision', metrics['precision'])
            live.log_metric('recall', metrics['recall'])
            live.log_metric('auc', metrics['auc'])

            # Log parameters
            live.log_params(params)

    except Exception as e:
        logger.error('Failed to complete model evaluation: %s', e)
        print(f"Error: {e}")


if __name__ == '__main__':
    main()
```

## 12.4 Understanding DVCLive Code

### 12.4.1 The Live Context Manager

```python
with Live(save_dvc_exp=True) as live:
    # Logging code here
```

- **save_dvc_exp=True**: Saves experiment in DVC
- **Context manager**: Automatically handles setup/cleanup
- **Creates dvclive/ directory**: Stores experiment data

### 12.4.2 Logging Metrics

```python
live.log_metric('accuracy', metrics['accuracy'])
live.log_metric('precision', metrics['precision'])
```

Each metric is:

- Recorded in `dvclive/` folder
- Tracked by DVC for comparison
- Viewable in experiment tables

### 12.4.3 Logging Parameters

```
live.log_params(params)
```

Logs all parameters from `params.yaml`:

- Test size
- Max features
- Model hyperparameters

## 12.5 Running Experiments

```
# Run an experiment (instead of dvc repro)
dvc exp run
```

**What happens:**

1. Pipeline executes
2. DVCLive logs metrics and parameters
3. `dvclive/` directory created
4. Experiment saved in DVC's experiment database
5. Unique experiment ID generated

## 12.6 Viewing Experiments

### 12.6.1 Command Line

```
# Show all experiments
dvc exp show
```

**Example Output:**

```
============================================================
Experiment         accuracy  precision  recall    auc
============================================================
workspace          0.9732    0.9821     0.9456   0.9887
  exp-a1b2c        0.9651    0.9745     0.9382   0.9801
  exp-c3d4e        0.9689    0.9778     0.9421   0.9845
  exp-e5f6g        0.9712    0.9802     0.9444   0.9869
============================================================
```

### 12.6.2 VS Code Extension

Install DVC Extension in VS Code:

1. Open VS Code
2. Go to Extensions (Ctrl+Shift+X)
3. Search "DVC"
4. Install "DVC" by Iterative

**Features**:

- Visual experiment comparison table
- Click to view experiment details
- Sort by metrics
- Apply experiments to workspace

## 12.7 The dvclive/ Directory

After running `dvc exp run`, a `dvclive/` directory is created:

```
dvclive/
  metrics.json
  params.yaml
  plots/
```

**Important: dvclive/ is Temporary!**

**Key Concept**: The `dvclive/` folder is like a whiteboard:

- **Current experiment only**: Contains latest run's data
- **Gets overwritten**: Next run replaces contents
- **DVC saves copies**: Each experiment snapshot saved internally
- **Don't add to Git**: Let DVC handle it

Think of it as:

- `dvclive/` = temporary workspace
- DVC's internal storage = permanent photo album
- `dvc exp show` = views the photo album, not the workspace

## 12.8 Running Multiple Experiments

### 12.8.1 Experiment 1: Baseline

```yaml
# params.yaml
model_building:
  n_estimators: 20
  random_state: 2
```

```
dvc exp run
```

### 12.8.2 Experiment 2: More Trees

```yaml
# params.yaml
model_building:
  n_estimators: 50
  random_state: 2
```

```
dvc exp run
```

### 12.8.3 Experiment 3: Different Features

```yaml
# params.yaml
feature_engineering:
  max_features: 100

model_building:
  n_estimators: 50
  random_state: 2
```

```
dvc exp run
```

## 12.9 Comparing Experiments

```
# Show all experiments with metrics
dvc exp show

# Show differences between experiments
dvc exp diff exp-a1b2c exp-c3d4e
```

## 12.10 Applying an Experiment

If Experiment 2 has the best metrics:

```
# Apply experiment to workspace
dvc exp apply exp-c3d4e
```

**What happens**:

- Workspace updated with that experiment's code and parameters
- Files restored to that experiment's state
- **Note**: Experiment ID no longer exists after applying

**Warning**

**Critical Understanding**:

`dvc exp apply` does NOT create a Git commit!
It's temporary—like copying experiment → workspace.
To make it permanent:

1. Apply experiment: `dvc exp apply <exp-id>`
2. Commit to Git: `git add . && git commit`

Experiments are NOT Git commits—they're separate DVC entities.

## 12.11 Removing Experiments

```
# Remove specific experiment
dvc exp remove exp-a1b2c

# Remove all experiments except workspace
dvc exp gc
```

## 12.12 Best Practices for Experimentation

1. **Run experiments frequently**: After each significant change
2. **Use descriptive parameters**: Clear naming in `params.yaml`
3. **Compare before committing**: Use `dvc exp show` to choose best
4. **Apply + Commit best experiment**: Make it permanent in Git
5. **Clean up**: Remove failed/uninteresting experiments

# 13 AWS S3 Integration for Remote Storage

## 13.1 Why Remote Storage?

**Local DVC cache (.dvc/cache/) limitations**:

- **Machine-specific**: Lost if machine crashes
- **Not shareable**: Team members can't access
- **Storage constraints**: Limited by local disk space
- **No backup**: Single point of failure

**Remote storage (S3) benefits**:

- **Cloud backup**: Data safe in AWS
- **Team collaboration**: Everyone accesses same data
- **Scalable**: Unlimited storage
- **Reproducible**: Pull exact data versions anywhere

## 13.2 AWS Setup Prerequisites

1. AWS Account
2. IAM User with S3 permissions
3. Access Key ID
4. Secret Access Key

## 13.3 Step-by-Step AWS Configuration

### 13.3.1 Step 1: Create IAM User

1. Log into AWS Console
2. Navigate to IAM → Users
3. Click "Create User"
4. Set username (e.g., `dvc-user`)
5. Enable "Programmatic access"
6. Attach policy: `AmazonS3FullAccess`
7. Save Access Key ID and Secret Access Key

**Warning**

**Security Warning**:

- NEVER commit AWS credentials to Git
- Store securely in environment variables

- Use IAM roles in production
- Consider using AWS CLI profiles

### 13.3.2 Step 2: Create S3 Bucket

1. Navigate to S3 in AWS Console
2. Click "Create bucket"
3. Enter unique bucket name (e.g., `my-dvc-storage-bucket`)
4. Select region (e.g., `us-east-1`)
5. Keep default settings (or configure as needed)
6. Click "Create bucket"

**Important Note**

**Bucket Naming Rules**:

- Must be globally unique across all AWS
- 3-63 characters long
- Lowercase letters, numbers, hyphens only
- Cannot start/end with hyphen

### 13.3.3 Step 3: Install AWS CLI and DVC S3 Support

```
# Install AWS CLI tools
pip install awscli

# Install DVC with S3 support
pip install dvc[s3]
```

### 13.3.4 Step 4: Configure AWS Credentials

```
# Configure AWS CLI
aws configure
```

**You'll be prompted for**:

```
AWS Access Key ID [None]: <your-access-key-id>
AWS Secret Access Key [None]: <your-secret-access-key>
Default region name [None]: us-east-1
Default output format [None]: json
```

**This creates**:

- `~/.aws/credentials`: Stores access keys
- `~/.aws/config`: Stores configuration

### 13.3.5 Step 5: Add S3 Remote to DVC

```
# Add S3 bucket as DVC remote
dvc remote add -d dvcstore s3://my-dvc-storage-bucket

# Verify remote configuration
dvc remote list
```

**Expected Output**:

```
dvcstore    s3://my-dvc-storage-bucket
```

**Command Breakdown**:

- `dvc remote add`: Adds a remote storage location
- `-d`: Sets as default remote
- `dvcstore`: Name for this remote (can be anything)
- `s3://...`: S3 bucket URL

## 13.4 Pushing Data to S3

### 13.4.1 Initial Push

```
# Push all tracked data to S3
dvc push
```

**What happens**:

1. DVC reads `dvc.lock` to identify tracked files
2. Compares local cache with S3
3. Uploads missing/changed files
4. Stores files using content-addressable hashes

**Expected Output**:

```
Collecting                                        |8.00 [00:00, 2.50entry/s]
Pushing                                           |8.00 [00:05, 1.60file/s]
8 files pushed
```

## 13.5 Understanding S3 Storage Structure

In your S3 bucket, files are stored like:

```
my-dvc-storage-bucket/
  files/
    md5/
      a1/
        b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6
      x1/
        y2z3a4b5c6d7e8f9g0h1i2j3k4l5m6
      ...
```

- **Content-addressable**: Files named by MD5 hash
- **Deduplicated**: Same content = same hash = stored once
- **Efficient**: Only changed files uploaded

## 13.6 Pulling Data from S3

### 13.6.1 Cloning Project on New Machine

```
# Clone Git repository
git clone https://github.com/username/MLOPS-DVC-Project.git
cd MLOPS-DVC-Project

# Pull data from S3
dvc pull
```

**What happens**:

1. DVC reads `dvc.lock`
2. Downloads required files from S3
3. Populates local cache
4. Restores files to workspace

## 13.7 Complete Experiment Workflow with S3

**Full Workflow: Local → Git + DVC → S3**

```
# 1. Run experiment
dvc exp run

# 2. If satisfied, push data to S3
dvc push

# 3. Commit code to Git
git add .
git commit -m "Experiment: increased n_estimators to 50"

# 4. Push code to GitHub
git push origin main
```

## 13.8 Key Concepts: Git vs DVC vs S3

| System | Stores | Purpose |
|---|---|---|
| **Git** | Code, `dvc.yaml`, `dvc.lock`, `params.yaml` | Version control for code and metadata |
| **Local DVC Cache** | All data versions locally | Fast access, temporary |
| **S3 Remote** | All data versions in cloud | Backup, sharing, collaboration |

**Data Flow Visualization**

**Workspace → DVC Cache → S3 Remote**

- `dvc add` / `dvc exp run`: Workspace → DVC Cache
- `dvc commit`: Finalizes tracked outputs (no new cache unless manually modified)
- `dvc push`: DVC Cache → Remote Storage (S3/GCS/etc.)
- `dvc pull`: Remote Storage → DVC Cache
- `dvc checkout`: DVC Cache → Workspace

## 13.9 Configuring Remote in .dvc/config

After running `dvc remote add`, check `.dvc/config`:

**.dvc/config**

```
[core]
    remote = dvcstore

['remote "dvcstore"']
    url = s3://my-dvc-storage-bucket
```

**This file is tracked by Git**, so team members automatically use the same remote!

## 13.10 Verifying S3 Upload

1. Go to AWS S3 Console
2. Navigate to your bucket
3. Check `files/md5/` directory
4. Verify files exist (they'll be named by hash)

## 13.11 Alternative: Using DVC with Other Storage

DVC supports multiple storage backends:

- **Google Cloud Storage**: `dvc remote add -d myremote gs://bucket`
- **Azure Blob**: `dvc remote add -d myremote azure://container`
- **Google Drive**: `dvc remote add -d myremote gdrive://folder-id`
- **SSH/SFTP**: `dvc remote add -d myremote ssh://server/path`
- **Local/Network**: `dvc remote add -d myremote /mnt/storage`

# 14 What does dvc push actually push?

## 14.1 What Exactly Does `dvc push` Upload?

**Key Rule**

`dvc push` uploads **only the data objects referenced by the current workspace state**. It does **not** automatically upload all past experiments or all cached data.

### 14.1.1 How DVC Decides What to Push

When you run `dvc push`, DVC answers one question:

> *Which data objects are required to reproduce the current workspace?*

To determine this, DVC inspects:

- `dvc.lock` (if using `dvc.yaml` pipelines)
- `.dvc` files (if using `dvc add`)

Each of these files contains **hashes** that point to objects in `.dvc/cache/`. Only those cache objects are considered "needed".

### 14.1.2 What Gets Pushed

- Data files, models, or artifacts stored in `.dvc/cache/`
- Only the cache objects whose hashes appear in:
  - the current `dvc.lock`, and/or
  - the current `.dvc` files

**Important**: DVC does *not* push hash files or tokens themselves— it pushes the actual data objects referenced by those hashes.

### 14.1.3 What Does NOT Get Pushed

- Data from previous experiments that are not applied
- Data referenced only by discarded or local experiments
- Experiment history stored in `.git/refs/exps/`
- All cache contents by default

### 14.1.4 Example: Multiple Experiments

Assume you ran three experiments:

- Experiment 1 produced data version A
- Experiment 2 produced data version B
- Experiment 3 produced data version C

After applying Experiment 3, the current `dvc.lock` contains hashes pointing only to version C.

```
dvc push
```

**Result**:

- Only data version C is uploaded to remote storage
- Versions A and B remain local

#### 14.1.5 Pushing Data for All Commits

If you want to upload data referenced by **all Git commits** (not just the current one), use:

```
dvc push --all-commits
```

This scans Git history and uploads cache objects referenced by every committed state. **Note**: This still does not push local experiment history—only committed states.

### 14.2 Summary

**Warning**

**Key Takeaways**:

- `dvc push` uploads data referenced by the current workspace only
- It relies on `dvc.lock` and/or `.dvc` files
- Unapplied experiments are not pushed
- To push all committed data, use `dvc push --all-commits`

# 15 Complete MLOps Workflow Summary

## 15.1 Running Multiple Experiments with DVCLive

**How DVC Experiments Work**

When you run multiple experiments with `dvc exp run`:

**What Happens**:

1. Run Experiment 1 → DVC stores it locally in `.git/refs/exps/`
2. Run Experiment 2 → DVC stores it separately, Exp 1 remains
3. Run Experiment 3 → DVC stores it separately, Exp 1 & 2 remain
4. Use `dvc exp show` → See all experiments with metrics
5. Use `dvc exp diff` → Compare experiments

**Result**: All experiments are tracked locally and can be compared!

**DVC + DVCLive Benefits**:

- DVCLive automatically logs metrics, parameters, and plots
- DVC assigns each experiment a unique identifier
- Compare multiple experiments side-by-side
- Local experiment history preserved

**Warning**

**Important Limitation**: Experiment History is Local Only

**The Constraint**:

- DVC experiments live in `.git/refs/exps/` (local Git refs)
- `dvc push` does NOT upload experiment history to remote
- Only the **applied/promoted** experiment is committed to Git
- Team members cannot see your experiment history

**For centralized experiment tracking across teams**, use MLflow or similar tools. (We'll cover this in advanced sections.)

## 15.2 Correct Workflow: Compare and Apply Best Experiment

**Proper Workflow for Multiple Experiments**

```
# ==== RUN MULTIPLE EXPERIMENTS ====
# Experiment 1
vim params.yaml  # e.g., n_estimators: 20
dvc exp run -n "baseline-20"
```

```
# Experiment 2
vim params.yaml  # e.g., n_estimators: 50
dvc exp run -n "test-50"

# Experiment 3
vim params.yaml  # e.g., n_estimators: 100
dvc exp run -n "test-100"

# ==== COMPARE EXPERIMENTS ====
dvc exp show
# See all experiments with metrics side-by-side

dvc exp diff baseline-20 test-50
# Compare specific experiments

# ==== APPLY BEST EXPERIMENT ====
# Choose best performing experiment (e.g., test-50)
dvc exp apply test-50

# ==== COMMIT THE BEST VERSION ====
# 1. Push data to S3 (uploads cached outputs)
dvc push

# 2. Commit to Git (saves code + metadata)
git add dvc.lock params.yaml
git commit -m "Apply best experiment: n_estimators=50, accuracy=0.92"
git push origin main
```

**What Gets Committed**

**One Git Commit = One Promoted Experiment**:

- Only the **applied** experiment becomes permanent
- Other experiments remain local (not pushed to remote)
- Git history shows only promoted experiments
- Each team member's local experiments stay private

## 15.3 Why This Matters for Rollback

| What You CAN Rollback | What You CANNOT Rollback |
|---|---|
| Promoted experiments (via Git commits) | Unpromoted local experiments |
| Code, data, models of committed state | Alternative experiments (not preserved) |
| `git checkout + dvc pull` works | Unpromoted experiments (may be discarded) |
| Parameters and metrics in Git history | Local experiment refs (not preserved, may be discarded) |

**Warning**

**Key Understanding**: Git Commits Are Singular

When you `dvc exp apply` and commit:

- One Git commit = one experiment outcome
- Unpromoted experiments are not preserved and may be discarded
- Rolling back restores only committed experiments
- To preserve all experiments, you need external tracking (e.g., MLflow)

### 15.4 Rollback Procedure

To reproduce a previously committed experiment:

```
# 1. View commit history
git log --oneline

# Example output:
# a1b2c3d Apply best: n_estimators=100, accuracy=0.94
# d4e5f6g Apply best: n_estimators=50, accuracy=0.92
# h7i8j9k Baseline: n_estimators=20, accuracy=0.88

# 2. Checkout desired experiment
git checkout d4e5f6g

# 3. Pull corresponding data from S3
dvc pull

# 4. Verify
python src/Model_Evaluation.py
```

**What happens**:

1. `git checkout`: Restores code + `dvc.lock` for that commit
2. `dvc pull`: Uses `dvc.lock` to fetch exact data from S3
3. Everything restored: code, data, model, parameters

**Rollback is Only Possible For...**

- Experiments that were **applied** with `dvc exp apply`
- Experiments that were committed to Git
- Experiments whose data was pushed with `dvc push`

Unpromoted local experiments are not preserved and may be discarded!

## 15.5 Complete Command Reference

**Experiment Workflow**

```
# 1. Run multiple experiments
dvc exp run -n "exp-name"  # Repeat with different params

# 2. Compare experiments
dvc exp show               # View all experiments
dvc exp diff exp1 exp2     # Compare two experiments

# 3. Apply best experiment
dvc exp apply exp-name     # Promote to workspace

# 4. Commit the promoted experiment
dvc push                   # Upload cached data to S3
git add dvc.lock params.yaml
git commit -m "message"
git push                   # Upload to GitHub
```

**To Rollback**

```
1. git checkout <hash>  # Restore code
2. dvc pull             # Restore data
```

**Advanced: Push All Commits' Data**

```
# Upload data for all Git commits (not just current)
dvc push --all-commits
```

## 15.6 Understanding DVCLive and Local Experiments

**The Experiment Lab Analogy**

Think of DVC experiments like a lab notebook:

**Current Workspace (dvclive/)**:

- Shows the active experiment's metrics and plots
- Gets overwritten when you run a new experiment
- Temporary workspace (don't add to Git)

**Experiment History (Local Git Refs)**:

- DVC stores all experiments in `.git/refs/exps/`
- Each experiment keeps its own metrics and parameters
- `dvc exp show` displays all experiments
- **Local only** - not pushed to GitHub/S3

**Committed Experiments (Git History)**:

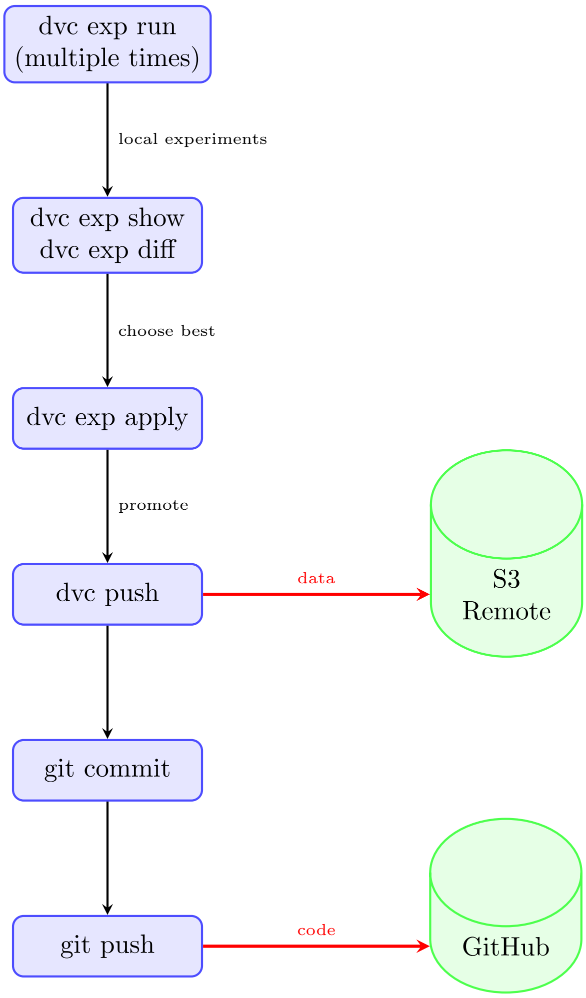
- Only applied experiments enter Git history
- Permanent and shareable across team
- Can be rolled back anytime

**Warning**

**Key Points**:

- `dvclive/` folder shows current state only
- DVC remembers all experiments locally
- Only promoted experiments survive in remote/Git
- For centralized experiment tracking, use MLflow

## 15.7 Workflow Visualization



## 15.8 When to Use MLflow

**Limitations of DVC Experiments**

DVC experiments are excellent for local iteration, but have limitations:

**What DVC Provides**:

- Local experiment tracking and comparison
- Data and model versioning
- Reproducibility of promoted experiments

**What DVC Does NOT Provide**:

- Centralized experiment history across team
- Remote storage of all experiment runs
- Web UI for comparing experiments
- Experiment history persistence beyond local refs

**Solution**: For centralized experiment tracking, use MLflow alongside DVC. (This will be covered in advanced sections.)

# 16 Common Issues and Troubleshooting

## 16.1 DVC Issues

### 16.1.1 Issue: "Stage didn't change, skipping"

**Problem**: DVC won't re-run stages even after changes.

**Solutions**:

```
# Force re-run entire pipeline
dvc repro --force

# Force re-run specific stage
dvc repro --force model_building

# Clear cache and re-run
dvc remove <stage>
dvc repro
```

### 16.1.2 Issue: "Failed to push data"

**Problem**: `dvc push` fails with S3 errors.

**Solutions**:

1. Check AWS credentials:

```
aws s3 ls s3://your-bucket-name
```

2. Verify remote configuration:

```
dvc remote list
cat .dvc/config
```

3. Check bucket permissions in AWS Console

4. Reconfigure AWS:

```
aws configure
```

### 16.1.3 Issue: "File not found" during dvc pull

**Problem**: `dvc pull` can't find files in S3.

**Cause**: You never ran `dvc push` for that version.

**Solution**:

- Cannot recover if data never pushed
- Re-run the experiment
- Follow complete workflow (commit → push)

## 16.2 Git Issues

### 16.2.1 Issue: Large files in Git

**Problem**: Accidentally committed data/models to Git.

**Solution**:

```
# Remove from Git tracking
git rm -r --cached data/
git rm -r --cached models/

# Update .gitignore
echo "data/" >> .gitignore
echo "models/" >> .gitignore

# Commit removal
git commit -m "Remove large files from Git tracking"
git push origin main
```

### 16.2.2 Issue: Merge conflicts in dvc.lock

**Problem**: Multiple team members modified pipeline.

**Solution**:

```
# Accept their version
git checkout --theirs dvc.lock

# Or accept your version
git checkout --ours dvc.lock

# Then re-run pipeline
dvc repro

# Commit resolved lock
git add dvc.lock
git commit -m "Resolve dvc.lock conflict"
```

## 16.3 Python Issues

### 16.3.1 Issue: Import errors

**Problem**: Cannot import modules.

**Solution**:

```
# Install requirements
pip install -r requirements.txt

# Or install individually
pip install pandas scikit-learn nltk pyyaml dvclive
```

### 16.3.2 Issue: NLTK data not found

**Problem**: `stopwords` or `punkt` not found.

**Solution**:

```
import nltk
nltk.download('stopwords')
nltk.download('punkt')
```

## 16.4 AWS Issues

### 16.4.1 Issue: "Access Denied" on S3

**Problem**: IAM user lacks permissions.

**Solution**:

1. Go to IAM in AWS Console
2. Find your user
3. Attach policy: `AmazonS3FullAccess`
4. Or create custom policy with s3:GetObject, s3:PutObject, s3:ListBucket

### 16.4.2 Issue: Wrong region

**Problem**: Bucket in different region than configured.

**Solution**:

```
# Check bucket region in S3 console
# Update AWS config
aws configure set region <correct-region>
```

## 16.5 Experiment Tracking Issues

### 16.5.1 Issue: Experiments not showing

**Problem**: `dvc exp show` displays no experiments.

**Cause**: Used `dvc repro` instead of `dvc exp run`.

**Solution**:

- Use `dvc exp run` for experiments
- `dvc repro` is for production pipeline execution

### 16.5.2 Issue: DVCLive not logging

**Problem**: No metrics in `dvclive/`.

**Solution**:

1. Verify DVCLive installed: `pip install dvclive`
2. Check code has `with Live(save_dvc_exp=True):`
3. Ensure metrics logged: `live.log_metric(...)`
4. Check for exceptions in logs

# 17 MLOps Best Practices

## 17.1 Code Organization Best Practices

### 17.1.1 Modular Design Principles

1. **Single Responsibility Principle**:
   - One component per file
   - Each module does one thing well
   - Clear separation of concerns

**Good vs Bad Structure**

**No Bad - Everything in one file**:

```
pipeline.py (2000 lines)
    - Data loading
    - Preprocessing
    - Feature engineering
    - Model training
    - Evaluation
```

**Yes Good - Modular structure**:

```
src/
+-- Data_Ingestion.py       (200 lines)
+-- Data_Pre_Processing.py  (250 lines)
+-- Feature_Engineering.py  (180 lines)
+-- Model_Building.py       (150 lines)
+-- Model_Evaluation.py     (200 lines)
```

2. **Reusable Functions**:

```python
# Good: Reusable utility functions
def load_params(params_path: str) -> dict:
    """Load parameters from YAML file."""
    with open(params_path, 'r') as f:
        return yaml.safe_load(f)

def setup_logger(name: str, log_file: str) -> logging.Logger:
    """Configure and return a logger."""
    logger = logging.getLogger(name)
    # ... setup code
    return logger

```

3. **Type Hints and Docstrings**:

```python
def train_model(
    X_train: np.ndarray,
    y_train: np.ndarray,
    params: dict
) -> RandomForestClassifier:
    """
```

```
    Train a RandomForest classifier.

    Args:
        X_train: Training features of shape (n_samples, n_features)
        y_train: Training labels of shape (n_samples,)
        params: Dictionary containing model hyperparameters

    Returns:
        Trained RandomForestClassifier instance

    Raises:
        ValueError: If X_train and y_train shapes don't match
    """
    # Implementation
    pass

```

### 17.1.2 Logging Best Practices

1. **Appropriate Log Levels**:

```
logger.debug('Detailed information for debugging')
logger.info('General informational messages')
logger.warning('Warning messages for potential issues')
logger.error('Error messages for failures')
logger.critical('Critical errors requiring immediate attention')

```

2. **Structured Logging**:

```
# Good: Structured, informative logging
logger.info(f'Training started with {X_train.shape[0]} samples')
logger.info(f'Parameters: {params}')
logger.debug(f'Feature shape: {X_train.shape}')
logger.info(f'Training completed in {elapsed_time:.2f}s')

# Bad: Vague logging
logger.info('Training done')

```

3. **Log File Organization**:

```
logs/
+-- Data_Ingestion_2024-12-20.log
+-- Data_Pre_Processing_2024-12-20.log
+-- Feature_Engineering_2024-12-20.log
+-- Model_Building_2024-12-20.log
+-- Model_Evaluation_2024-12-20.log
```

### 17.1.3 Error Handling Patterns

```
def load_data(file_path: str) -> pd.DataFrame:
    """Load data with comprehensive error handling."""
```

```
    try:
        # Attempt operation
        df = pd.read_csv(file_path)
        logger.info(f'Successfully loaded data from {file_path}')
        return df

    except FileNotFoundError:
        # Specific exception handling
        logger.error(f'File not found: {file_path}')
        raise

    except pd.errors.ParserError as e:
        # Another specific exception
        logger.error(f'Failed to parse CSV: {e}')
        raise

    except Exception as e:
        # Catch-all for unexpected errors
        logger.error(f'Unexpected error loading data: {e}')
        raise

    finally:
        # Cleanup code (if needed)
        logger.debug('Load data operation completed')
```

## 17.2 Data Management Best Practices

### 17.2.1 Directory Structure Philosophy

**Three-Tier Data Structure**

**data/raw/**: Original, immutable data

- Never modify these files
- Treat as read-only
- Can always reproduce from source

**data/interim/**: Intermediate processing

- Partially processed data
- Reusable across experiments
- Checkpoint for long pipelines

**data/processed/**: Final features

- Ready for model training
- Feature matrices
- Final transformations applied

### 17.2.2 Data Versioning Strategy

1. **Always Use DVC for Data**:

```
# Either track data directly:
dvc add data/processed

# OR track data via pipelines:
dvc.yaml:
  outs:
    - data/processed
```

2. **Never Commit Large Files to Git**:

**Warning**

Files larger than 100MB should NEVER go into Git:

- Git repositories become slow
- Clone times increase dramatically
- GitHub blocks files ¿100MB
- Use DVC instead!

3. **Avoid Manual Data Version Directories**:
   - Prefer DVC hashes over folder-based versioning
   - Use Git commits and DVC cache for true version history
   - Manual version folders are optional and discouraged at scale

### 17.2.3 Data Validation

```python
def validate_data(
    df: pd.DataFrame,
    expected_columns: list,
    min_rows: int = 100
) -> bool:
    """
    Validate data meets expectations.

    Checks:
        - Required columns exist
        - Minimum row count met
        - No completely empty columns
        - Correct data types
    """
    logger.info('Starting data validation')

    # Check columns
    missing_cols = set(expected_columns) - set(df.columns)
    if missing_cols:
        logger.error(f'Missing columns: {missing_cols}')
        raise ValueError(f"Missing required columns: {missing_cols}"
    )

    # Check row count
    if len(df) < min_rows:
```

```python
            logger.error(f'Only {len(df)} rows, expected {min_rows}')
            raise ValueError(f"Insufficient rows: {len(df)} < {min_rows}")

        # Check for empty columns
        empty_cols = df.columns[df.isnull().all()].tolist()
        if empty_cols:
            logger.warning(f'Empty columns found: {empty_cols}')

        # Check data types
        type_issues = []
        expected_types = {'text': 'object', 'target': ('int64', 'float64')}
        for col, expected_dtype in expected_types.items():
            if df[col].dtype not in expected_dtype:
                type_issues.append(f'{col}: {df[col].dtype}')

        if type_issues:
            logger.error(f'Type mismatches: {type_issues}')
            raise TypeError(f"Data type issues: {type_issues}")

        logger.info('Data validation passed')
        return True
```

## 17.3 Experiment Management Best Practices

### 17.3.1 Naming Conventions

1. **Descriptive Commit Messages**:

**Good Commit Messages**

```
Yes "Promote: n_estimators=100, accuracy=0.97"
Yes "Feature: Add TF-IDF with bigrams, precision +0.03"
Yes "Fix: Correct data leakage in preprocessing pipeline"
Yes "Baseline: RandomForest with default params, acc=0.95"
```

**Warning**

**Bad Commit Messages**:

```
No "Update model"
No "Fix bug"
No "Changes"
No "WIP"
No "asdfasdf"
```

2. **Experiment Naming Pattern**:

```
exp-<date>-<feature>-<value>

Examples:
exp-2024-12-20-n_estimators-100
```

```
exp-2024-12-20-max_features-200
exp-2024-12-20-baseline
```

### 17.3.2 Parameter Management

1. **All Hyperparameters in params.yaml**:

```
# No Bad: Hardcoded in code
clf = RandomForestClassifier(n_estimators=50, max_depth=10)

# Yes Good: From params.yaml
params = load_params('params.yaml')['model_building']
clf = RandomForestClassifier(**params)
```

2. **Document Parameter Choices**:

```
# params.yaml with comments
model_building:
  n_estimators: 50       # Increased from 20, improved accuracy
  max_depth: 10          # Prevent overfitting on small dataset
  min_samples_split: 5   # Baseline value
  random_state: 2        # For reproducibility
```

3. **Track Parameter Changes**:

```
# In commit messages
git commit -m "Exp: n_estimators 20→50, max_depth 5→10"
```

### 17.3.3 Metric Tracking

1. **Track Multiple Metrics**:

```
with Live(save_dvc_exp=True) as live:
    # Primary metrics
    live.log_metric('accuracy', accuracy)
    live.log_metric('precision', precision)
    live.log_metric('recall', recall)
    live.log_metric('f1_score', f1)
    live.log_metric('auc', auc)

    # Secondary metrics
    live.log_metric('training_time', elapsed_time)
    live.log_metric('num_features', X_train.shape[1])
    live.log_metric('num_samples', X_train.shape[0])

    # Log all parameters
    live.log_params(params)
```

2. **Business-Relevant Metrics**:

```python
# Don't just track ML metrics
# Track business impact
business_metrics = {
    'false_positive_rate': fp_rate,
    'false_negative_rate': fn_rate,
    'cost_per_error': calculate_cost(fp, fn),
    'expected_savings': calculate_savings(tp, tn)
}
```

## 17.4 Collaboration Best Practices

### 17.4.1 Documentation Standards

**Essential README.md Sections**

```
# Project Name

## Overview
Brief description of the ML problem and solution.

## Setup Instructions
1. Clone repository
2. Create virtual environment
3. Install dependencies
4. Configure AWS credentials
5. Run pipeline

## Project Structure
Explain directory organization.

## Data Description
- Source of data
- Features and target
- Data collection process

## Pipeline Stages
Describe each stage of the pipeline.

## Experiments
Table of experiments with results.

## How to Run
- Training: `dvc repro`
- Experiments: `dvc exp run`
- Rollback: Instructions

## Troubleshooting
Common issues and solutions.

## Team Members
```

```
Contact information.

## License
Project license.
```

### 17.4.2 Code Review Process

1. **Pull Request Template**:

```
## Changes
- What was changed
- Why it was changed

## Experiment Results
- Metric improvements
- Parameter changes

## Testing
- What was tested
- Test results

## Checklist
- [ ] Code follows style guide
- [ ] Tests pass
- [ ] Documentation updated
- [ ] dvc repro runs successfully (only for projects with dvc.yaml)
```

2. **Review Checklist**:
   - Code is modular and readable
   - Proper error handling
   - Logging is comprehensive
   - Parameters in params.yaml
   - No hardcoded values
   - Data not committed to Git
   - Tests exist and pass
   - Documentation updated

### 17.4.3 Team Communication

1. **Experiment Log**:

| Date | Experimenter | Changes | Result |
|---|---|---|---|
| 2024-12-20 | Alice | n_estimators=100 | acc=0.97 |
| 2024-12-19 | Bob | Add bigrams | acc=0.96 |
| 2024-12-18 | Alice | Baseline | acc=0.95 |

2. **Weekly Sync Meetings**:

- Review experiment results
- Discuss blockers
- Plan next experiments
- Share learnings

3. **Slack/Communication Channels**:

```
#ml-experiments: Share experiment results
#ml-questions: Ask technical questions
#ml-alerts: Pipeline failures, important updates
```

## 17.5 Security and Compliance

### 17.5.1 Credential Management

**Warning**

**NEVER Commit These to Git**:

- AWS Access Keys / Secret Keys
- API tokens
- Database passwords
- Private keys
- OAuth tokens
- Any secrets or credentials

**Secure Credential Management**

```python
# Yes Good: Use environment variables
import os

AWS_ACCESS_KEY = os.getenv('AWS_ACCESS_KEY_ID')
AWS_SECRET_KEY = os.getenv('AWS_SECRET_ACCESS_KEY')
API_KEY = os.getenv('API_KEY')

# Load from .env file (never commit .env!)
from dotenv import load_dotenv
load_dotenv()

# No Bad: Hardcoded credentials
AWS_ACCESS_KEY = "AKIAIOSFODNN7EXAMPLE"  # DON'T DO THIS!
```

**.env file (add to .gitignore)**:

```
AWS_ACCESS_KEY_ID=your_access_key
AWS_SECRET_ACCESS_KEY=your_secret_key
API_KEY=your_api_key
```

### 17.5.2 Data Privacy

1. **Sensitive Data Handling**:

```python
def anonymize_data(df: pd.DataFrame) -> pd.DataFrame:
    """Remove PII (Personally Identifiable Information)."""
    # Remove or hash sensitive columns
    df = df.drop(columns=['email', 'phone', 'ssn'])

    # Hash user IDs
    df['user_id'] = df['user_id'].apply(
        lambda x: hashlib.sha256(str(x).encode()).hexdigest()
    )

    return df

```

2. **Data Access Controls**:

- Use IAM roles with minimal permissions
- Separate dev/staging/prod environments
- Audit data access logs
- Implement data retention policies

## 17.6 Performance Optimization

### 17.6.1 Caching Strategies

1. **DVC Cache Optimization**:

```bash
# Use hardlinks for faster cache
dvc config cache.type hardlink,symlink

# Set cache directory on faster disk
dvc cache dir /path/to/ssd/cache

# Shared cache for team
dvc config cache.dir /shared/dvc/cache
dvc config cache.shared group

```

2. **Partial Pipeline Execution**:

```bash
# Run only specific stage
dvc repro model_building

# Run from specific stage onwards
dvc repro --downstream model_building

```

3. **Parallel Processing**:

```python
# Use all CPU cores
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(
    n_estimators=100,
```

```
    n_jobs=-1  # Use all cores
)

# Parallel data loading
import multiprocessing as mp

with mp.Pool(processes=4) as pool:
    results = pool.map(process_file, file_list)

```

### 17.6.2 Memory Management

```
def load_data_chunked(file_path: str, chunksize: int = 10000):
    """Load large CSV in chunks to manage memory."""
    chunks = []
    for chunk in pd.read_csv(file_path, chunksize=chunksize):
        # Process chunk
        processed = preprocess_chunk(chunk)
        chunks.append(processed)

    return pd.concat(chunks, ignore_index=True)

# Clear memory after use
import gc
del large_dataframe
gc.collect()
```

## 17.7 Common Pitfalls to Avoid

**Warning**

**Top 10 Common Mistakes in MLOps**:

1. Committing large data or models to Git instead of using DVC
2. Promoting or committing experiments prematurely instead of comparing them first
3. Hardcoding parameters instead of using `params.yaml`
4. Not pushing data to remote storage, making commits unreproducible
5. Editing `dvc.lock` manually
6. Committing AWS credentials, API keys, or secrets to version control
7. Using `dvc repro` when experiment tracking with `dvc exp run` is intended
8. Ignoring logs and metrics, which are critical for debugging and evaluation
9. Skipping data validation at pipeline stages
10. Failing to document experiment assumptions and results

**Best Practice Checklist**

Before committing code, verify:

- All parameters are defined in `params.yaml`
- No hardcoded configuration values in code
- Logging is configured in all critical modules
- Error handling is implemented
- Type hints and docstrings are added where appropriate
- Data files are excluded via `.gitignore`
- No credentials or secrets are present in the codebase
- Pipelines run successfully (`dvc repro`, if applicable)
- Best experiment is applied and committed (not all experiments)
- Documentation and experiment notes are updated

# 18 Quick Reference Guide

## 18.1 Essential DVC Commands

Pipeline Management

```
dvc init                 # Initialize DVC in repository
dvc repro                # Run entire pipeline
dvc repro --force        # Force re-run all stages
dvc repro <stage>        # Run specific stage
dvc dag                  # Visualize pipeline as DAG
dvc status               # Check pipeline status
dvc status -c            # Check status with cloud comparison
```

Data Management

```
dvc add <file>           # Track file/directory with DVC
dvc commit               # Save workspace changes to cache
dvc push                 # Upload data to remote storage
dvc pull                 # Download data from remote storage
dvc checkout             # Restore files from cache
dvc fetch                # Download without checkout
dvc gc                   # Garbage collect unused cache
```

Remote Storage

```
dvc remote add -d <name> <url>     # Add default remote
dvc remote list                     # List all remotes
dvc remote modify <name> <key> <v>  # Modify remote config
dvc remote remove <name>            # Remove remote
dvc remote rename <old> <new>       # Rename remote

# Examples:
dvc remote add -d s3store s3://bucket
dvc remote add -d gdrive gdrive://folder-id
dvc remote add -d local /mnt/storage
```

Experiment Tracking

```
dvc exp run              # Run experiment (saves automatically)
dvc exp show             # Show all experiments in table
dvc exp show --no-pager # Show without pagination
dvc exp diff             # Compare experiments
dvc exp diff <exp1> <exp2>  # Compare specific experiments
dvc exp apply <id>       # Apply experiment to workspace
dvc exp remove <id>      # Remove specific experiment
dvc exp gc               # Clean up all experiments
dvc exp list             # List all experiments
```

Git Integration

```
git add dvc.yaml dvc.lock params.yaml  # Stage DVC files
git commit -m "message"                 # Commit to Git
git push origin main                    # Push to GitHub

# Typical workflow
git add .
git commit -m "Experiment: description"
git push
```

## 18.2 Complete Workflow Cheat Sheet

### 18.2.1 Initial Project Setup

Step 1: Project Initialization

```
# 1. Create and clone Git repository
git clone https://github.com/username/project.git
cd project

# 2. Create virtual environment
python -m venv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate     # Windows

# 3. Install dependencies
pip install pandas scikit-learn nltk pyyaml
pip install dvc dvclive
pip install dvc[s3]       # For AWS S3
pip install awscli        # AWS CLI

# 4. Initialize DVC
dvc init

# 5. Create directory structure
mkdir -p src data/raw data/interim data/processed models reports logs

# 6. Configure .gitignore
cat >> .gitignore << EOF
data/
models/
reports/
logs/
venv/
__pycache__/
*.pyc
dvclive/
EOF

# 7. Initial commit
```

```
git add .
git commit -m "Initial project setup"
git push origin main
```

### 18.2.2 Full Experiment Workflow

Step 2: Running Experiments

```
# === RUN MULTIPLE EXPERIMENTS ===

# 1. Modify parameters
vim params.yaml

# 2. Run experiment
dvc exp run

# 3. View and compare results
dvc exp show

# === REPEAT STEPS 1{3 FOR MULTIPLE EXPERIMENTS ===

# 4. Choose the best experiment

# 5. Apply (promote) the best experiment
dvc exp apply <exp-id>

# 6. Commit promoted state to Git
git add dvc.lock params.yaml
git commit -m "Promote best experiment: n_estimators=50, acc=0.97"

# 7. Push data to remote storage
dvc push

# 8. Push code to GitHub
git push origin main
```

### 18.2.3 Rollback to a Promoted (Committed) Experiment

Step 3: Rollback Procedure

```
# 1. View commit history
git log --oneline

# Example output:
# a1b2c3d Promote best: n_estimators=100, acc=0.98
# d4e5f6g Promote best: n_estimators=50, acc=0.97
# h7i8j9k Promote best: n_estimators=20, acc=0.95

# 2. Checkout desired promoted experiment (Git commit)
git checkout d4e5f6g
```

```
# 3. Pull corresponding data from remote storage
dvc pull

# 4. Verify the restored experiment
python src/Model_Evaluation.py
cat reports/metrics.json

# 5. Return to the latest version (optional)
git checkout main
dvc pull
```

### 18.2.4 AWS S3 Setup Workflow

Step 4: AWS Integration

```
# 1. Install AWS tools
pip install dvc[s3]
pip install awscli

# 2. Configure AWS credentials
aws configure
# Enter: Access Key ID
# Enter: Secret Access Key
# Enter: Region (e.g., us-east-1)
# Enter: Output format (json)

# 3. Add S3 remote to DVC
dvc remote add -d dvcstore s3://your-bucket-name

# 4. Verify configuration
dvc remote list
# Output: dvcstore    s3://your-bucket-name

# 5. Test connection
dvc push

# 6. Commit DVC config to Git
git add .dvc/config
git commit -m "Add S3 remote storage"
git push origin main
```

### 18.3 Project Structure Template

```
MLOPS-DVC-Project/
|
+-- Experiments/              # Exploration phase
|   +-- spam.csv
|   +-- mynotebook.ipynb
|
+-- src/                      # Production code
|   +-- Data_Ingestion.py
|   +-- Data_Pre_Processing.py
|   +-- Feature_Engineering.py
|   +-- Model_Building.py
|   +-- Model_Evaluation.py
|
+-- data/                     # Data files (DVC tracked)
|   +-- raw/                  # Original data
|   +-- interim/              # Intermediate processing
|   +-- processed/            # Final features
|
+-- models/                   # Trained models (DVC tracked)
|   +-- model.pkl
|
+-- reports/                  # Metrics and reports (DVC tracked)
|   +-- metrics.json
|
+-- logs/                     # Application logs
|   +-- Data_Ingestion.log
|   +-- Data_Pre_Processing.log
|   +-- Feature_Engineering.log
|   +-- Model_Building.log
|   +-- Model_Evaluation.log
|
+-- dvclive/                  # DVCLive temporary files
|   +-- metrics.json
|   +-- params.yaml
|   +-- plots/
|
+-- .dvc/                     # DVC configuration
|   +-- cache/                # Local data cache
|   +-- config                # Remote storage config
|   +-- .gitignore
|
+-- dvc.yaml                  # Pipeline definition
+-- dvc.lock                  # Pipeline lock file
+-- params.yaml               # Hyperparameters
+-- .gitignore                # Git ignore rules
+-- .dvcignore                # DVC ignore rules
+-- requirements.txt          # Python dependencies
+-- README.md                 # Project documentation
```

## 18.4 Key File Purposes

| File | Purpose |
|---|---|
| **dvc.yaml** | Defines pipeline stages, dependencies, outputs, parameters |
| **dvc.lock** | Locks exact file versions using MD5 hashes |
| **params.yaml** | Centralized hyperparameter configuration |
| **.gitignore** | Tells Git what NOT to track (data, models, logs) |
| **.dvcignore** | Tells DVC what NOT to track |
| **.dvc/config** | DVC remote storage configuration |
| **.dvc/cache/** | Local cache storing all data versions |
| **dvclive/** | Temporary experiment metrics (current run only) |
| **requirements.txt** | Python package dependencies |

## 18.5 Git vs DVC Tracking

| Git Tracks | DVC Tracks |
|---|---|
| Source code (`src/*.py`) | Data files (`data/**`) |
| `dvc.yaml` (pipeline definition) | Model files (`models/*.pkl`) |
| `dvc.lock` (version locks) | Reports (`reports/*.json`) |
| `params.yaml` (hyperparameters) | Large binary files |
| `.gitignore`, `.dvcignore` | Outputs defined in `dvc.yaml` |
| `.dvc/config` (DVC settings) | Intermediate data artifacts |
| Documentation (`README.md`) | Feature matrices |
| `requirements.txt` | Preprocessed datasets |

## 18.6 params.yaml Template

**Complete params.yaml Example**

```
# Data Ingestion Parameters
data_ingestion:
  test_size: 0.15
  random_state: 2
  data_url: "https://raw.githubusercontent.com/.../spam.csv"

# Data Preprocessing Parameters
data_preprocessing:
  remove_stopwords: true
  apply_stemming: true
  lowercase: true

# Feature Engineering Parameters
feature_engineering:
  max_features: 45
  ngram_range: [1, 2]
  min_df: 2
  max_df: 0.95

# Model Building Parameters
model_building:
  model_type: "RandomForest"
  n_estimators: 20
  max_depth: null
```

```
  min_samples_split: 2
  min_samples_leaf: 1
  random_state: 2
  n_jobs: -1

# Model Evaluation Parameters
model_evaluation:
  metrics:
    - accuracy
    - precision
    - recall
    - f1_score
    - auc
```

## 18.7 dvc.yaml Template

**Complete dvc.yaml with All Features**

```
stages:
  data_ingestion:
    cmd: python src/Data_Ingestion.py
    deps:
      - src/Data_Ingestion.py
    params:
      - data_ingestion.test_size
      - data_ingestion.random_state
    outs:
      - data/raw

  data_preprocessing:
    cmd: python src/Data_Pre_Processing.py
    deps:
      - data/raw
      - src/Data_Pre_Processing.py
    params:
      - data_preprocessing
    outs:
      - data/interim

  feature_engineering:
    cmd: python src/Feature_Engineering.py
    deps:
      - data/interim
      - src/Feature_Engineering.py
    params:
      - feature_engineering.max_features
      - feature_engineering.ngram_range
    outs:
      - data/processed

  model_building:
    cmd: python src/Model_Building.py
    deps:
      - data/processed
```

```
      - src/Model_Building.py
    params:
      - model_building
    outs:
      - models/model.pkl

  model_evaluation:
    cmd: python src/Model_Evaluation.py
    deps:
      - models/model.pkl
      - data/processed
      - src/Model_Evaluation.py
    params:
      - model_evaluation.metrics
    metrics:
      - reports/metrics.json:
          cache: false
    plots:
      - reports/plots/confusion_matrix.png
```

## 18.8 Common Command Combinations

Frequently Used Command Sequences

```
# Complete experiment cycle
dvc exp run && dvc commit && dvc push && git add . && \
git commit -m "Experiment" && git push

# Quick status check
dvc status && git status

# Full pipeline re-run
dvc repro --force && dvc push && git add dvc.lock && \
git commit -m "Pipeline re-run" && git push

# Sync with team
git pull && dvc pull

# Clean up experiments
dvc exp gc && dvc gc

# View pipeline and experiments
dvc dag && dvc exp show
```

# 19 Conclusion and Key Takeaways

## 19.1 What You've Accomplished

Throughout this comprehensive guide, you've built a complete, production-ready MLOps pipeline. Let's review the major achievements:

1. **End-to-End ML Pipeline**:
   - Data Ingestion with error handling
   - Data Pre-Processing with text transformation
   - Feature Engineering using TF-IDF
   - Model Building with RandomForest
   - Model Evaluation with comprehensive metrics
2. **Pipeline Automation**:
   - Automated workflow using DVC
   - Intelligent dependency tracking
   - Reproducible experiments
   - Parameter-driven configuration
3. **Version Control Mastery**:
   - Git for code versioning
   - DVC for data versioning
   - AWS S3 for cloud backup
   - Complete rollback capability
4. **Experiment Tracking**:
   - DVCLive integration
   - Metric comparison across experiments
   - Parameter tracking
   - Visual experiment comparison
5. **Production Readiness**:
   - Comprehensive logging
   - Error handling
   - Modular architecture
   - Team collaboration setup

## 19.2 The MLOps Journey: From Notebook to Production

**The Transformation**

**Where You Started**:

- Jupyter notebooks
- Local experiments

- Manual tracking
- Irreproducible results
- Solo development

**Where You Are Now**:

- Production-ready code
- Automated pipelines
- Systematic tracking
- Fully reproducible
- Collaboration-ready

## 19.3 Core Principles You've Learned

### 19.3.1 1. Version Everything

| Component | Tool | What's Tracked |
|---|---|---|
| Code | Git | Python scripts, configs |
| Data | DVC | Raw, interim, processed |
| Models | DVC | Trained model files |
| Parameters | Git | params.yaml |
| Metrics | DVC | Evaluation results |
| Dependencies | Git | requirements.txt |

### 19.3.2 2. Automate Repetitive Tasks

- **Before**: Manually run 5 Python scripts in sequence
- **After**: `dvc repro` runs entire pipeline

### 19.3.3 3. Make Everything Reproducible

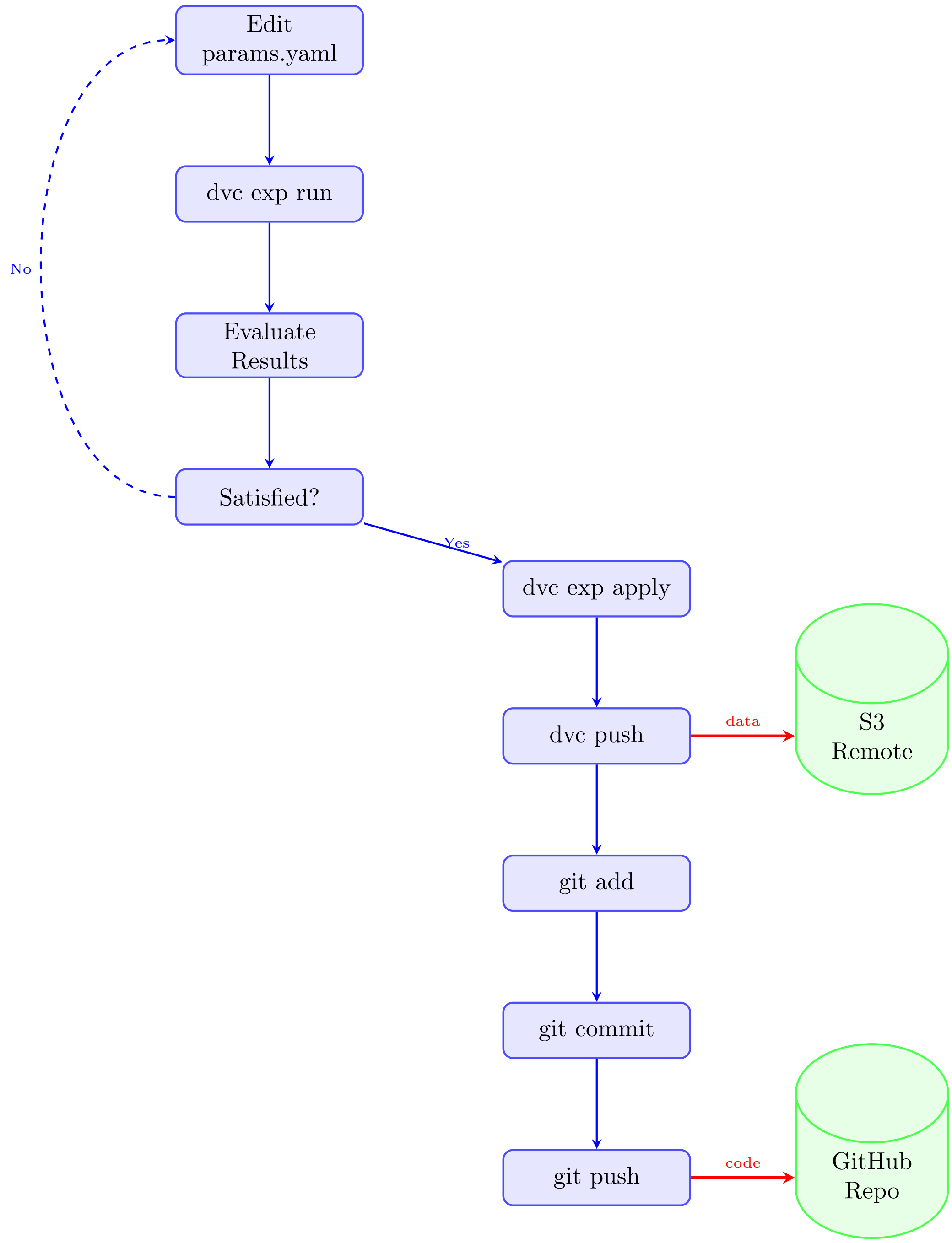
**The Golden Rule**:

```
git checkout <commit-hash>
dvc pull
# => Exact same code, data, and results
```

### 19.3.4 4. Track Experiments Systematically

| Experiment | Parameters | Accuracy | Decision |
|---|---|---|---|
| Exp 1 | n_estimators=20 | 0.950 | Baseline |
| Exp 2 | n_estimators=50 | 0.970 | Improved |
| Exp 3 | n_estimators=100 | 0.972 | Selected |
| Exp 4 | n_estimators=200 | 0.971 | Overfitting |

## 19.4 The Complete Workflow Visualization




## 19.5 Key Lessons Learned

1. **Data is as Important as Code**:
   - Models depend on data
   - Data changes over time
   - Version control applies to both
2. **Automation Saves Time**:
   - Initial setup takes time
   - Long-term benefits are huge

- Consistency across team

3. **Logging is Essential**:
   - Debug issues faster
   - Understand what happened
   - Track pipeline progress

4. **Parameters in Config Files**:
   - Easy experimentation
   - Clear documentation
   - Version controlled

5. **Rollback Capability is Powerful**:
   - Revert to any **promoted and committed** experiment
   - Compare past and current results reliably
   - Enables safe and reproducible experimentation