# Handout 3: Advanced Classification - Production Systems at Scale

Machine Learning for Smarter Innovation

## 1 Handout 3: Advanced Classification - Production Systems at Scale

### 1.1 Learning Objectives

By the end of this handout, you will: - Build production-ready ML pipelines with proper architecture - Implement advanced optimization and ensemble techniques - Deploy models using modern containerization and orchestration - Monitor and maintain models in production environments - Handle real-world challenges at scale

### 1.2 Part 1: Production Pipeline Architecture

#### 1.2.1 1.1 Microservices Architecture for ML

```
# app.py - Flask microservice for model serving
from flask import Flask, request, jsonify
from flask_cors import CORS
import joblib
import numpy as np
import pandas as pd
from datetime import datetime
import redis
import json
import logging
from prometheus_flask_exporter import PrometheusMetrics

app = Flask(__name__)
CORS(app)
metrics = PrometheusMetrics(app)

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

# Initialize Redis for caching
redis_client = redis.Redis(host='localhost', port=6379, db=0, decode_responses
    =True)

# Model versioning
MODEL_VERSION = "v2.3.1"
MODEL_PATH = f"models/classifier_{MODEL_VERSION}.pkl"
```

```python
class ModelService:
    def __init__(self):
        self.model = None
        self.feature_names = None
        self.load_model()

    def load_model(self):
        """Load model with error handling and fallback"""
        try:
            model_data = joblib.load(MODEL_PATH)
            self.model = model_data['model']
            self.feature_names = model_data['feature_names']
            self.scaler = model_data['scaler']
            logger.info(f"Model {MODEL_VERSION} loaded successfully")
        except Exception as e:
            logger.error(f"Failed to load model: {e}")
            # Fallback to previous version
            self.load_fallback_model()

    def load_fallback_model(self):
        """Load fallback model version"""
        fallback_path = "models/classifier_v2.3.0.pkl"
        try:
            model_data = joblib.load(fallback_path)
            self.model = model_data['model']
            self.feature_names = model_data['feature_names']
            self.scaler = model_data['scaler']
            logger.warning("Loaded fallback model v2.3.0")
        except Exception as e:
            logger.critical(f"Failed to load fallback model: {e}")
            raise

    def preprocess(self, data):
        """Preprocessing pipeline with validation"""
        df = pd.DataFrame([data])

        # Validate required features
        missing_features = set(self.feature_names) - set(df.columns)
        if missing_features:
            raise ValueError(f"Missing features: {missing_features}")

        # Feature engineering
        df['efficiency_ratio'] = df['output'] / (df['cost'] + 1e-10)
        df['time_to_market'] = (pd.to_datetime('now') - pd.to_datetime(df['
    start_date'])).dt.days

        # Select and scale features
        X = df[self.feature_names]
        X_scaled = self.scaler.transform(X)

        return X_scaled

    def predict(self, data, user_id=None):
        """Make prediction with caching and logging"""
        # Check cache
        cache_key = f"prediction:{json.dumps(data, sort_keys=True)}"
        cached_result = redis_client.get(cache_key)

        if cached_result:
            logger.info(f"Cache hit for user {user_id}")
            return json.loads(cached_result)

        # Preprocess and predict
        X = self.preprocess(data)
        prediction = self.model.predict(X)[0]
        probability = self.model.predict_proba(X)[0].tolist()

        # Prepare response
        result = {
            'prediction': int(prediction),
            'probability': probability,
            'confidence': float(max(probability)),
            'model_version': MODEL_VERSION,
            'timestamp': datetime.now().isoformat()
        }

        # Cache result (TTL: 1 hour)
        redis_client.setex(cache_key, 3600, json.dumps(result))

        # Log prediction for monitoring
        self.log_prediction(data, result, user_id)

        return result

    def log_prediction(self, input_data, result, user_id):
        """Log predictions for monitoring and analysis"""
        log_entry = {
            'user_id': user_id,
            'input': input_data,
            'output': result,
            'timestamp': result['timestamp']
        }

        # Send to monitoring system (e.g., Kafka, CloudWatch)
        logger.info(f"Prediction logged: {log_entry}")

        # Store in database for analysis
        # db.predictions.insert(log_entry)

# Initialize model service
model_service = ModelService()

@app.route('/health', methods=['GET'])
def health_check():
    """Health check endpoint for load balancer"""
    return jsonify({
        'status': 'healthy',
        'model_version': MODEL_VERSION,
        'timestamp': datetime.now().isoformat()
    })

@app.route('/predict', methods=['POST'])
@metrics.counter('predictions_total', 'Total predictions',
                 labels={'model_version': lambda: MODEL_VERSION})
def predict():
    """Main prediction endpoint"""
    try:
        data = request.json
        user_id = request.headers.get('X-User-ID')

        # Input validation
        if not data:
            return jsonify({'error': 'No data provided'}), 400

        # Make prediction
        result = model_service.predict(data, user_id)
        return jsonify(result), 200

    except ValueError as e:
        logger.error(f"Validation error: {e}")
        return jsonify({'error': str(e)}), 400
    except Exception as e:
        logger.error(f"Prediction error: {e}")
        return jsonify({'error': 'Internal server error'}), 500

@app.route('/batch_predict', methods=['POST'])
def batch_predict():
    """Batch prediction endpoint for multiple inputs"""
    try:
        data_list = request.json.get('data', [])

        if not data_list:
            return jsonify({'error': 'No data provided'}), 400

        results = []
        for data in data_list:
            result = model_service.predict(data)
            results.append(result)

        return jsonify({'results': results}), 200

    except Exception as e:
        logger.error(f"Batch prediction error: {e}")
        return jsonify({'error': 'Batch processing failed'}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=False)
```

### 1.2.2 1.2 Feature Store Implementation

```
# feature_store.py - Centralized feature management
import pandas as pd
from datetime import datetime, timedelta
import hashlib
from typing import List, Dict, Any
import sqlalchemy
from sqlalchemy import create_engine
import redis
import json

class FeatureStore:
    def __init__(self, db_url: str, redis_host: str = 'localhost'):
        self.engine = create_engine(db_url)
        self.redis_client = redis.Redis(host=redis_host, port=6379, db=1)
        self.feature_registry = {}

    def register_feature(self, name: str, description: str,
                         computation: str, dependencies: List[str] = None):
        """Register a new feature in the store"""
        self.feature_registry[name] = {
            'description': description,
            'computation': computation,
            'dependencies': dependencies or [],
            'created_at': datetime.now(),
            'version': self._generate_version(computation)
        }
        # Store in database
        registry_df = pd.DataFrame([self.feature_registry[name]])
        registry_df['name'] = name
        registry_df.to_sql('feature_registry', self.engine,
                          if_exists='append', index=False)

    def compute_features(self, entity_id: str, feature_names: List[str],
                        point_in_time: datetime = None) -> Dict[str, Any]:
        """Compute features for a given entity"""
        if point_in_time is None:
            point_in_time = datetime.now()

        # Check cache first
        cache_key = self._get_cache_key(entity_id, feature_names,
point_in_time)
        cached = self.redis_client.get(cache_key)

        if cached:
            return json.loads(cached)

        features = {}

        for feature_name in feature_names:
            if feature_name not in self.feature_registry:
                raise ValueError(f"Unknown feature: {feature_name}")

            feature_def = self.feature_registry[feature_name]

            # Compute dependencies first
            dep_features = {}
            if feature_def['dependencies']:
                dep_features = self.compute_features(
                    entity_id, feature_def['dependencies'], point_in_time
                )

            # Execute feature computation
            feature_value = self._execute_computation(
                feature_def['computation'], entity_id, point_in_time,
dep_features
            )
            features[feature_name] = feature_value

        # Cache results
        self.redis_client.setex(
            cache_key, 3600, json.dumps(features, default=str)
        )

        return features

    def _execute_computation(self, computation: str, entity_id: str,
                           point_in_time: datetime, dependencies: Dict) -> Any
:
        """Execute feature computation logic"""
        # This would normally execute SQL or Python code
        # For demonstration, using a simple eval (NOT for production!)

        # Fetch raw data
        query = f"""
        SELECT * FROM transactions
        WHERE entity_id = '{entity_id}'
        AND created_at <= '{point_in_time}'
        """
        df = pd.read_sql(query, self.engine)

        # Apply computation
        # In production, use safe execution methods
        local_vars = {
            'df': df,
            'dependencies': dependencies,
            'pd': pd,
            'np': np
        }

        return eval(computation, {"__builtins__": {}}, local_vars)

    def _get_cache_key(self, entity_id: str, features: List[str],
                       timestamp: datetime) -> str:
        """Generate cache key for features"""
        feature_str = ','.join(sorted(features))
        key_str = f"{entity_id}:{feature_str}:{timestamp.date()}"
        return hashlib.md5(key_str.encode()).hexdigest()

    def _generate_version(self, computation: str) -> str:
        """Generate version hash for feature computation"""
        return hashlib.md5(computation.encode()).hexdigest()[:8]

# Example feature definitions
feature_store = FeatureStore('postgresql://user:pass@localhost/features')

# Register features
feature_store.register_feature(
    name='transaction_count_30d',
    description='Number of transactions in last 30 days',
    computation="len(df[df['created_at'] >= pd.Timestamp.now() - pd.Timedelta(
    days=30)])"
)

feature_store.register_feature(
    name='average_transaction_value',
    description='Average transaction value',
    computation="df['amount'].mean()"
)

feature_store.register_feature(
    name='risk_score',
    description='Computed risk score',
    computation="dependencies['transaction_count_30d'] * 0.3 + dependencies['
    average_transaction_value'] * 0.7",
    dependencies=['transaction_count_30d', 'average_transaction_value']
)
```

## 1.3 Part 2: Advanced Optimization Techniques

### 1.3.1 2.1 AutoML Pipeline with Hyperparameter Optimization

```
# automl_pipeline.py
import optuna
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import numpy as np
import joblib
import json
from typing import Dict, Any, Tuple

class AutoMLClassifier:
    def __init__(self, time_budget: int = 3600, n_jobs: int = -1):
        self.time_budget = time_budget  # seconds
        self.n_jobs = n_jobs
        self.best_model = None
        self.best_params = None
        self.best_score = -np.inf
        self.study = None
        self.scaler = StandardScaler()

    def _create_model(self, trial: optuna.Trial, model_type: str):
        """Create model with Optuna trial parameters"""

        if model_type == 'random_forest':
            return RandomForestClassifier(
                n_estimators=trial.suggest_int('rf_n_estimators', 50, 500),
                max_depth=trial.suggest_int('rf_max_depth', 3, 20),
                min_samples_split=trial.suggest_int('rf_min_samples_split', 2,
 20),
                min_samples_leaf=trial.suggest_int('rf_min_samples_leaf', 1,
10),
                max_features=trial.suggest_categorical('rf_max_features', ['
sqrt', 'log2', None]),
                n_jobs=self.n_jobs
            )

        elif model_type == 'gradient_boosting':
            return GradientBoostingClassifier(
                n_estimators=trial.suggest_int('gb_n_estimators', 50, 300),
                learning_rate=trial.suggest_float('gb_learning_rate', 0.01,
0.3, log=True),
                max_depth=trial.suggest_int('gb_max_depth', 3, 10),
                subsample=trial.suggest_float('gb_subsample', 0.5, 1.0),
                min_samples_split=trial.suggest_int('gb_min_samples_split', 2,
 20)
            )

        elif model_type == 'svm':
            return SVC(
                C=trial.suggest_float('svm_C', 1e-3, 1e3, log=True),
                kernel=trial.suggest_categorical('svm_kernel', ['linear', 'rbf
', 'poly']),
                gamma=trial.suggest_categorical('svm_gamma', ['scale', 'auto'
]),
                probability=True
            )

        elif model_type == 'neural_network':
            n_layers = trial.suggest_int('nn_n_layers', 1, 3)
            layers = []
            for i in range(n_layers):
                layers.append(trial.suggest_int(f'nn_layer_{i}_size', 10, 200)
)

            return MLPClassifier(
                hidden_layer_sizes=tuple(layers),
                activation=trial.suggest_categorical('nn_activation', ['relu',
 'tanh']),
                solver=trial.suggest_categorical('nn_solver', ['adam', 'lbfgs'
]),
                alpha=trial.suggest_float('nn_alpha', 1e-5, 1e-1, log=True),
                learning_rate=trial.suggest_categorical('nn_learning_rate',
                                                       ['constant', 'adaptive'
]),
                max_iter=500
            )

    def objective(self, trial: optuna.Trial, X: np.ndarray, y: np.ndarray) ->
float:
        """Objective function for Optuna optimization"""

        # Select model type
        model_type = trial.suggest_categorical('model_type',
            ['random_forest', 'gradient_boosting', 'svm', 'neural_network'])

        # Create model
        model = self._create_model(trial, model_type)

        # Create pipeline with scaling for certain models
        if model_type in ['svm', 'neural_network']:
            pipeline = Pipeline([
                ('scaler', StandardScaler()),
                ('model', model)
            ])
        else:
            pipeline = Pipeline([
                ('model', model)
            ])

        # Evaluate using cross-validation
        cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
        scores = cross_val_score(pipeline, X, y, cv=cv, scoring='roc_auc',
n_jobs=self.n_jobs)

        return scores.mean()

    def fit(self, X: np.ndarray, y: np.ndarray) -> 'AutoMLClassifier':
        """Fit AutoML pipeline"""

        # Create Optuna study
        self.study = optuna.create_study(
            direction='maximize',
            sampler=optuna.samplers.TPESampler(seed=42),
            pruner=optuna.pruners.MedianPruner()
        )

        # Optimize
        self.study.optimize(
            lambda trial: self.objective(trial, X, y),
            timeout=self.time_budget,
            n_jobs=1  # Parallelism handled by sklearn
        )

        # Get best parameters
        self.best_params = self.study.best_params
        self.best_score = self.study.best_value

        # Train final model with best parameters
        model_type = self.best_params['model_type']
        # Create model with best parameters
        trial = optuna.trial.FixedTrial(self.best_params)
        model = self._create_model(trial, model_type)

        # Create final pipeline
        if model_type in ['svm', 'neural_network']:
            self.best_model = Pipeline([
                ('scaler', StandardScaler()),
                ('model', model)
            ])
        else:
            self.best_model = Pipeline([
                ('model', model)
            ])

        # Fit on full data
        self.best_model.fit(X, y)

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Make predictions"""
        return self.best_model.predict(X)

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Predict probabilities"""
        return self.best_model.predict_proba(X)

    def save_results(self, filepath: str):
        """Save optimization results and best model"""
        results = {
            'best_params': self.best_params,
            'best_score': self.best_score,
            'optimization_history': [
                {
                    'number': trial.number,
                    'value': trial.value,
                    'params': trial.params,
                    'state': str(trial.state)
                }
                for trial in self.study.trials
            ]
        }

        # Save results
        with open(f"{filepath}_results.json", 'w') as f:
            json.dump(results, f, indent=2)

        # Save model
        joblib.dump(self.best_model, f"{filepath}_model.pkl")

        # Save study for further analysis
        self.study.trials_dataframe().to_csv(f"{filepath}_trials.csv", index=
    False)

# Usage example
if __name__ == "__main__":
    from sklearn.datasets import make_classification
    from sklearn.model_selection import train_test_split

    # Generate data
    X, y = make_classification(n_samples=1000, n_features=20, n_informative
   =15,
                               n_redundant=5, n_classes=2, random_state=42)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
   random_state=42)

    # Run AutoML
    automl = AutoMLClassifier(time_budget=300)  # 5 minutes
    automl.fit(X_train, y_train)

    print(f"Best model: {automl.best_params['model_type']}")
    print(f"Best score: {automl.best_score:.4f}")
    print(f"Test score: {automl.score(X_test, y_test):.4f}")

    # Save results
    automl.save_results("automl_classifier")
```

### 1.3.2 2.2 Advanced Ensemble with Stacking

```
# ensemble_stacking.py
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.model_selection import KFold
import numpy as np
from typing import List, Tuple
import pandas as pd

class AdvancedStackingClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, base_models: List[Tuple[str, BaseEstimator]],
                 meta_model: BaseEstimator, cv: int = 5,
                 use_probas: bool = True, passthrough: bool = False):
        """
        Advanced stacking classifier with cross-validation

        Args:
            base_models: List of (name, model) tuples
            meta_model: Meta-learner model
            cv: Number of CV folds for creating meta features
            use_probas: Use probability predictions if available
            passthrough: Include original features in meta model
        """
        self.base_models = base_models
        self.meta_model = meta_model
        self.cv = cv
        self.use_probas = use_probas
        self.passthrough = passthrough
        self.fitted_models_ = []

    def fit(self, X: np.ndarray, y: np.ndarray) -> 'AdvancedStackingClassifier
   ':
        """Fit stacking classifier"""
        n_samples = X.shape[0]
        n_models = len(self.base_models)

        # Determine number of meta features per model
        n_classes = len(np.unique(y))
        if self.use_probas and n_classes > 2:
            n_meta_features_per_model = n_classes
        else:
            n_meta_features_per_model = 1 if n_classes == 2 else n_classes
        # Initialize meta features array
        meta_features = np.zeros((n_samples, n_models *
    n_meta_features_per_model))

        # Create meta features using cross-validation
        kf = KFold(n_splits=self.cv, shuffle=True, random_state=42)

        for fold_idx, (train_idx, val_idx) in enumerate(kf.split(X)):
            X_train_fold, X_val_fold = X[train_idx], X[val_idx]
            y_train_fold = y[train_idx]

            for model_idx, (name, model) in enumerate(self.base_models):
                # Clone and fit model on fold
                model_clone = clone(model)
                model_clone.fit(X_train_fold, y_train_fold)

                # Generate predictions for validation fold
                if self.use_probas and hasattr(model_clone, 'predict_proba'):
                    fold_preds = model_clone.predict_proba(X_val_fold)
                    if n_classes == 2:
                        fold_preds = fold_preds[:, 1:2]  # Use positive class
    only
                else:
                    fold_preds = model_clone.predict(X_val_fold).reshape(-1,
    1)

                # Store in meta features
                start_idx = model_idx * n_meta_features_per_model
                end_idx = start_idx + n_meta_features_per_model
                meta_features[val_idx, start_idx:end_idx] = fold_preds

        # Train base models on full dataset
        self.fitted_models_ = []
        for name, model in self.base_models:
            model_clone = clone(model)
            model_clone.fit(X, y)
            self.fitted_models_.append((name, model_clone))

        # Prepare meta training data
        if self.passthrough:
            meta_X = np.hstack([meta_features, X])
        else:
            meta_X = meta_features

        # Train meta model
        self.meta_model.fit(meta_X, y)

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Make predictions"""
        meta_features = self._create_meta_features(X)

        if self.passthrough:
            meta_X = np.hstack([meta_features, X])
        else:
            meta_X = meta_features

        return self.meta_model.predict(meta_X)

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Predict probabilities"""
        meta_features = self._create_meta_features(X)
        if self.passthrough:
            meta_X = np.hstack([meta_features, X])
        else:
            meta_X = meta_features

        return self.meta_model.predict_proba(meta_X)

    def _create_meta_features(self, X: np.ndarray) -> np.ndarray:
        """Create meta features for new data"""
        n_samples = X.shape[0]
        meta_features = []

        for name, model in self.fitted_models_:
            if self.use_probas and hasattr(model, 'predict_proba'):
                preds = model.predict_proba(X)
                if preds.shape[1] == 2:
                    preds = preds[:, 1:2]
            else:
                preds = model.predict(X).reshape(-1, 1)

            meta_features.append(preds)

        return np.hstack(meta_features)

# Helper function for cloning
from sklearn.base import clone

# Example usage with diverse models
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from xgboost import XGBClassifier

# Define base models
base_models = [
    ('rf', RandomForestClassifier(n_estimators=100, random_state=42)),
    ('gb', GradientBoostingClassifier(n_estimators=100, random_state=42)),
    ('svm', SVC(probability=True, random_state=42)),
    ('nb', GaussianNB()),
    ('xgb', XGBClassifier(n_estimators=100, random_state=42))
]

# Define meta model
meta_model = LogisticRegression(random_state=42)

# Create stacking classifier
stacker = AdvancedStackingClassifier(
    base_models=base_models,
    meta_model=meta_model,
    cv=5,
    use_probas=True,
    passthrough=False
)
```

## 1.4 Part 3: Deployment Strategies

### 1.4.1 3.1 Docker Containerization

```
# Dockerfile
FROM python:3.9-slim

# Set working directory
WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements
COPY requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Create non-root user
RUN useradd -m -u 1000 mluser && chown -R mluser:mluser /app
USER mluser

# Expose port
EXPOSE 5000

# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:5000/health || exit 1

# Run application
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "--workers", "4", "--timeout", "120
  ", "app:app"]
```

### 1.4.2 3.2 Kubernetes Deployment

```
# kubernetes-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-classifier-deployment
  labels:
    app: ml-classifier
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ml-classifier
  template:
    metadata:
      labels:
        app: ml-classifier
    spec:
      containers:
      - name: ml-classifier
        image: ml-classifier:v2.3.1
        ports:
        - containerPort: 5000
```

```yaml
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1000m"
        env:
        - name: MODEL_VERSION
          value: "v2.3.1"
        - name: REDIS_HOST
          value: "redis-service"
        livenessProbe:
          httpGet:
            path: /health
            port: 5000
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health
            port: 5000
          initialDelaySeconds: 5
          periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: ml-classifier-service
spec:
  selector:
    app: ml-classifier
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
  type: LoadBalancer
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: ml-classifier-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: ml-classifier-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

## 1.5 Part 4: Monitoring and Maintenance

### 1.5.1 4.1 Model Monitoring Dashboard

```
# monitoring.py
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
from typing import Dict, List, Tuple
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from scipy import stats
import warnings

class ModelMonitor:
    def __init__(self, reference_data: pd.DataFrame,
                 reference_predictions: np.ndarray):
        """Initialize monitor with reference data"""
        self.reference_data = reference_data
        self.reference_predictions = reference_predictions
        self.reference_stats = self._compute_statistics(reference_data)
        self.alerts = []

    def _compute_statistics(self, data: pd.DataFrame) -> Dict:
        """Compute statistical properties of data"""
        stats_dict = {}

        for column in data.columns:
            if data[column].dtype in ['int64', 'float64']:
                stats_dict[column] = {
                    'mean': data[column].mean(),
                    'std': data[column].std(),
                    'min': data[column].min(),
                    'max': data[column].max(),
                    'q25': data[column].quantile(0.25),
                    'q50': data[column].quantile(0.50),
                    'q75': data[column].quantile(0.75)
                }
            else:
                stats_dict[column] = {
                    'unique_values': data[column].nunique(),
                    'mode': data[column].mode()[0] if len(data[column].mode())
     > 0 else None
                }

        return stats_dict

    def detect_drift(self, current_data: pd.DataFrame,
                     threshold: float = 0.05) -> Dict[str, bool]:
        """Detect data drift using statistical tests"""
        drift_results = {}

        for column in current_data.columns:
            if column not in self.reference_data.columns:
                continue

            if current_data[column].dtype in ['int64', 'float64']:
                # Kolmogorov-Smirnov test for numerical features
                ks_statistic, p_value = stats.ks_2samp(
                    self.reference_data[column].dropna(),
                    current_data[column].dropna()
                )
                drift_detected = p_value < threshold
            else:
                # Chi-square test for categorical features
                ref_counts = self.reference_data[column].value_counts()
                curr_counts = current_data[column].value_counts()

                # Align categories
                all_categories = set(ref_counts.index) | set(curr_counts.index
)
                ref_aligned = [ref_counts.get(cat, 0) for cat in
all_categories]
                curr_aligned = [curr_counts.get(cat, 0) for cat in
all_categories]

                chi2, p_value = stats.chisquare(curr_aligned, ref_aligned)
                drift_detected = p_value < threshold

            drift_results[column] = {
                'drift_detected': drift_detected,
                'p_value': p_value,
                'test_statistic': ks_statistic if current_data[column].dtype
in ['int64', 'float64'] else chi2
            }

            if drift_detected:
                self.alerts.append({
                    'type': 'data_drift',
                    'feature': column,
                    'p_value': p_value,
                    'timestamp': datetime.now()
                })

        return drift_results

    def monitor_predictions(self, current_predictions: np.ndarray,
                            current_timestamps: pd.Series) -> Dict:
        """Monitor prediction distribution changes"""

        # Prediction drift
        ks_statistic, p_value = stats.ks_2samp(
            self.reference_predictions,
            current_predictions
        )

        # Prediction statistics
        pred_stats = {
            'mean': np.mean(current_predictions),
            'std': np.std(current_predictions),
            'drift_detected': p_value < 0.05,
            'p_value': p_value
        }

        # Check for sudden changes
        if len(current_predictions) > 100:
            recent_mean = np.mean(current_predictions[-50:])
            older_mean = np.mean(current_predictions[-100:-50])

            if abs(recent_mean - older_mean) > 2 * np.std(self.
reference_predictions):
                self.alerts.append({
                    'type': 'prediction_shift',
                    'recent_mean': recent_mean,
                    'older_mean': older_mean,
                'timestamp': datetime.now()
            })

        return pred_stats

    def create_monitoring_dashboard(self, current_data: pd.DataFrame,
                                   current_predictions: np.ndarray,
                                   performance_metrics: Dict) -> go.Figure:
        """Create interactive monitoring dashboard"""

        fig = make_subplots(
            rows=3, cols=2,
            subplot_titles=('Prediction Distribution', 'Feature Drift Scores',
                           'Model Performance Over Time', 'Alert Timeline',
                           'Feature Importance Changes', 'Prediction Confidence
'),
            specs=[[{'type': 'histogram'}, {'type': 'bar'}],
                  [{'type': 'scatter'}, {'type': 'scatter'}],
                  [{'type': 'bar'}, {'type': 'violin'}]]
        )

        # 1. Prediction Distribution
        fig.add_trace(
            go.Histogram(x=self.reference_predictions, name='Reference',
                        opacity=0.5, marker_color='blue'),
            row=1, col=1
        )
        fig.add_trace(
            go.Histogram(x=current_predictions, name='Current',
                        opacity=0.5, marker_color='red'),
            row=1, col=1
        )

        # 2. Feature Drift Scores
        drift_results = self.detect_drift(current_data)
        features = list(drift_results.keys())
        p_values = [drift_results[f]['p_value'] for f in features]

        fig.add_trace(
            go.Bar(x=features[:10], y=p_values[:10], name='P-values',
                  marker_color=['red' if p < 0.05 else 'green' for p in
p_values[:10]]),
            row=1, col=2
        )

        # 3. Model Performance Over Time (mock data for example)
        dates = pd.date_range(end=datetime.now(), periods=30, freq='D')
        mock_accuracy = np.random.normal(0.85, 0.02, 30)

        fig.add_trace(
            go.Scatter(x=dates, y=mock_accuracy, mode='lines+markers',
                      name='Accuracy', line=dict(color='purple')),
            row=2, col=1
        )

        # Add threshold line
        fig.add_hline(y=0.80, line_dash="dash", line_color="red",
                     annotation_text="Minimum Threshold", row=2, col=1)

        # 4. Alert Timeline
        if self.alerts:
            alert_times = [a['timestamp'] for a in self.alerts[-20:]]
            alert_types = [a['type'] for a in self.alerts[-20:]]
            fig.add_trace(
                go.Scatter(x=alert_times, y=range(len(alert_times)),
                          mode='markers', marker=dict(size=10),
                          text=alert_types, name='Alerts'),
                row=2, col=2
            )

        # 5. Feature Importance Changes (mock data)
        features = ['Feature1', 'Feature2', 'Feature3', 'Feature4', 'Feature5'
    ]
        original_importance = np.random.random(5)
        current_importance = original_importance + np.random.normal(0, 0.05,
    5)

        fig.add_trace(
            go.Bar(x=features, y=original_importance, name='Original',
                  marker_color='blue', opacity=0.6),
            row=3, col=1
        )
        fig.add_trace(
            go.Bar(x=features, y=current_importance, name='Current',
                  marker_color='red', opacity=0.6),
            row=3, col=1
        )

        # 6. Prediction Confidence Distribution
        confidence_scores = np.abs(current_predictions - 0.5) * 2  # Convert
    to confidence

        fig.add_trace(
            go.Violin(y=confidence_scores, name='Confidence',
                     box_visible=True, meanline_visible=True),
            row=3, col=2
        )

        # Update layout
        fig.update_layout(
            title='Model Monitoring Dashboard',
            showlegend=True,
            height=900,
            width=1400
        )

        return fig

# Example usage
if __name__ == "__main__":
    # Create mock reference data
    reference_data = pd.DataFrame(
        np.random.randn(1000, 10),
        columns=[f'feature_{i}' for i in range(10)]
    )
    reference_predictions = np.random.random(1000)

    # Initialize monitor
    monitor = ModelMonitor(reference_data, reference_predictions)

    # Create mock current data
    current_data = pd.DataFrame(
        np.random.randn(500, 10) + 0.1,  # Slight drift
        columns=[f'feature_{i}' for i in range(10)]
    )
    current_predictions = np.random.random(500) + 0.05

    # Detect drift
    drift_results = monitor.detect_drift(current_data)
    print("Drift detected in features:",
          [f for f, r in drift_results.items() if r['drift_detected']])

    # Create dashboard
    fig = monitor.create_monitoring_dashboard(
        current_data, current_predictions,
        {'accuracy': 0.85, 'precision': 0.82, 'recall': 0.88}
    )
    fig.show()
```

## 1.6 Part 5: Complete Production Pipeline

### 1.6.1 5.1 End-to-End MLOps Pipeline

```
# mlops_pipeline.py
import mlflow
import mlflow.sklearn
from mlflow.tracking import MlflowClient
import yaml
from pathlib import Path
import hashlib
import json

class MLOpsPipeline:
    def __init__(self, config_path: str):
        with open(config_path, 'r') as f:
            self.config = yaml.safe_load(f)

        mlflow.set_tracking_uri(self.config['mlflow']['tracking_uri'])
        mlflow.set_experiment(self.config['mlflow']['experiment_name'])
        self.client = MlflowClient()

    def train_pipeline(self, data_path: str):
        """Complete training pipeline with MLflow tracking"""

        with mlflow.start_run() as run:
            # Log parameters
            mlflow.log_params(self.config['model']['parameters'])

            # Data versioning
            data_hash = self._hash_file(data_path)
            mlflow.log_param('data_version', data_hash)

            # Load and preprocess data
            X_train, X_test, y_train, y_test = self._load_data(data_path)

            # Feature engineering
            X_train_feat, X_test_feat = self._engineer_features(X_train,
    X_test)

            # Train model
            model = self._train_model(X_train_feat, y_train)

            # Evaluate
            metrics = self._evaluate_model(model, X_test_feat, y_test)
            mlflow.log_metrics(metrics)
            # Save model
            mlflow.sklearn.log_model(
                model,
                "model",
                registered_model_name=self.config['model']['name']
            )

            # Promote to staging if performance threshold met
            if metrics['accuracy'] > self.config['promotion']['
    accuracy_threshold']:
                self._promote_model(run.info.run_id, 'Staging')

            return run.info.run_id

    def deploy_pipeline(self, model_version: str, environment: str):
        """Deploy model to specified environment"""

        # Load model from registry
        model_uri = f"models:/{self.config['model']['name']}/{model_version}"
        model = mlflow.sklearn.load_model(model_uri)

        # Generate deployment configuration
        deployment_config = {
            'model_version': model_version,
            'environment': environment,
            'replicas': self.config['deployment'][environment]['replicas'],
            'resources': self.config['deployment'][environment]['resources']
        }

        # Deploy to Kubernetes
        self._deploy_to_k8s(model, deployment_config)

        # Set up monitoring
        self._setup_monitoring(model_version, environment)

        # Update model registry
        self.client.transition_model_version_stage(
            name=self.config['model']['name'],
            version=model_version,
            stage='Production' if environment == 'prod' else 'Staging'
        )

    def _hash_file(self, filepath: str) -> str:
        """Generate hash of data file for versioning"""
        with open(filepath, 'rb') as f:
            return hashlib.md5(f.read()).hexdigest()

    def _promote_model(self, run_id: str, stage: str):
        """Promote model to specified stage"""
        model_version = self.client.search_model_versions(
            f"run_id='{run_id}'"
        )[0].version

        self.client.transition_model_version_stage(
            name=self.config['model']['name'],
            version=model_version,
            stage=stage
        )
```

### 1.6.2 5.2 Configuration Management

```yaml
# config.yaml
mlflow:
  tracking_uri: http://localhost:5000
  experiment_name: innovation_classifier

model:
  name: innovation_classifier
  type: RandomForestClassifier
  parameters:
    n_estimators: 100
    max_depth: 10
    min_samples_split: 5

data:
  train_path: data/train.csv
  test_path: data/test.csv
  features:
    numerical:
      - novelty_score
      - market_size
      - team_experience
      - development_time
    categorical:
      - category
      - region

preprocessing:
  scaler: StandardScaler
  encoder: OneHotEncoder

promotion:
  accuracy_threshold: 0.85

deployment:
  staging:
    replicas: 2
    resources:
      memory: 512Mi
      cpu: 500m
  production:
    replicas: 5
    resources:
      memory: 1Gi
      cpu: 1000m

monitoring:
  drift_threshold: 0.05
  performance_threshold: 0.80
  alert_email: ml-team@company.com
```

## 1.7 Summary

This advanced handout covered: 1. **Production Architecture**: Microservices, feature stores, caching 2. **Advanced Optimization**: AutoML, hyperparameter tuning, ensemble methods 3. **Deployment**: Docker, Kubernetes, scaling strategies 4. **Monitoring**: Drift detection, performance tracking, alerting 5. **MLOps Pipeline**: End-to-end automation with MLflow

Key takeaways for production ML: - Always version your models and data - Implement comprehensive monitoring from day one - Use containerization for reproducible deployments - Automate retraining based on performance metrics - Design for failure with fallback models - Cache predictions for improved latency - Use feature stores for consistency across teams

Production ML is about reliability, scalability, and maintainability - not just accuracy!