Machine Learning Fundamentals in Azure Databricks

ML Pipelines & Customer Churn Prediction - Teaching Notes

Nived Varma, Microsoft Certified Trainer

OPENING HOOK (3 minutes)

WHITEBOARD STEP 1: Draw the "ML Chaos vs ML Pipeline" comparison:

```
❌ TRADITIONAL APPROACH             ☑ PIPELINE APPROACH
Raw Data                            Raw Data
   ↓ (manual step)                     ↓
Clean Data                          [Pipeline Stage 1: Cleaner]
   ↓ (manual step)                     ↓
Transform Features                  [Pipeline Stage 2: Transformer]
   ↓ (manual step)                     ↓
Train Model                         [Pipeline Stage 3: Estimator]
   ↓ (error-prone!)                    ↓
Predictions                         Predictions (Automated!)
```

Example Scenario: Insurance company predicting customer churn - millions of policyholders, dozens of behavioral features, multiple preprocessing steps to identify customers likely to cancel policies.

SECTION 1: ML Pipeline Fundamentals (12 minutes)

WHITEBOARD STEP 2: Create the "Customer Churn Prediction" scenario:

```
PROBLEM: Predict customer churn in insurance
INPUT FEATURES:           TARGET:
- Policy type             - Churn (0=Stay, 1=Leave)
- Customer age
- Premium amount
- Claims frequency
- Customer segment (text)
- Years as customer
- Has multiple policies
```

A. What is an ML Pipeline?

WHITEBOARD STEP 3: Draw the "Pipeline Concept":

```
    ML PIPELINE = SERIES OF STAGES
```

```
Stage 1      Stage 2      Stage 3      Stage 4
[String   →  [Vector   →  [Standard  →  [Logistic
Indexer]     Assembler]   Scaler]       Regression]

"Premium" → 0 → [45,2500,1,5,0,2] → [0.8,0.6,0.2,0.3,0,0.4] → Churn: 0.23
```

Code Example:

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, StandardScaler
from pyspark.ml.classification import LogisticRegression

# Define stages for churn prediction
indexer = StringIndexer(inputCol="customer_segment", outputCol="segment_idx")
assembler = VectorAssembler(inputCols=["age", "premium_amount",
"claims_frequency",
                                       "years_customer", "segment_idx",
"multiple_policies"],
                           outputCol="features")
scaler = StandardScaler(inputCol="features", outputCol="scaled_features")
lr = LogisticRegression(featuresCol="scaled_features", labelCol="churn")

# Create pipeline
pipeline = Pipeline(stages=[indexer, assembler, scaler, lr])

# One line to train everything!
model = pipeline.fit(train_data)
```

B. Pipeline Benefits

WHITEBOARD STEP 4: Create benefits web diagram:

```
        PIPELINE BENEFITS
       /         |        \
Reproducibility  |   No Data Leakage
      |          |          |
Easy Deployment  |    Version Control
      |          |          |
  Scalability    |    Error Prevention

Benefits for Insurance:
- Consistent churn scoring
- Automated customer segmentation
- Real-time risk assessment
```

SECTION 2: Pipeline Stages Deep Dive (15 minutes)

A. Transformer vs Estimator Patterns

WHITEBOARD STEP 5: Draw the fundamental patterns:

```
TRANSFORMER PATTERN:
- Has transform() method
- No learning required
- Stateless operations
- Examples: StringIndexer, VectorAssembler

Input Data → [Transformer] → Transformed Data
"Premium"  →  [StringIndexer]  →  0

ESTIMATOR PATTERN:
- Has fit() method that returns Transformer
- Learning required
- Creates Model (which is a Transformer)
- Examples: LogisticRegression, StandardScaler

Training Data → [Estimator.fit()] → Model (Transformer)
Then: New Data → [Model.transform()] → Churn Predictions
```

B. Common Pipeline Stages

WHITEBOARD STEP 6: Create the "Pipeline Stages Toolkit":

```
SPARK ML PIPELINE STAGES

📊 DATA PREPARATION
├── StringIndexer (customer_segment → numbers)
├── OneHotEncoder (policy_type → binary vectors)
├── VectorAssembler (columns → feature vector)
├── StandardScaler (normalize premium amounts)
└── MinMaxScaler (scale claim frequencies to 0-1)

🔍 FEATURE SELECTION
├── ChiSqSelector (statistical churn indicators)
├── UnivariateFeatureSelector (correlation-based)
└── VectorSlicer (manual selection)

🤖 MACHINE LEARNING
├── LogisticRegression (churn probability)
├── RandomForestClassifier (ensemble churn prediction)
├── GBTClassifier (gradient boosted churn model)
└── LinearSVC (support vector churn classifier)
```

C. Stage Execution Flow

WHITEBOARD STEP 7: Show step-by-step data flow:

```
STEP-BY-STEP PIPELINE EXECUTION

Original Data:
| age | premium | claims | years | segment | multiple | churn |
| 45  |  2500   |   1    |   5   | Premium |   yes    |  ???  |

Step 1 - StringIndexer:
| age | premium | claims | years | segment_idx | multiple | churn |
| 45  |  2500   |   1    |   5   |      0      |   yes    |  ???  |

Step 2 - OneHotEncoder:
| age | premium | claims | years | segment_idx | multiple_enc | churn |
| 45  |  2500   |   1    |   5   |      0      |    [1,0]     |  ???  |

Step 3 - VectorAssembler:
| features                     | churn |
| [45, 2500, 1, 5, 0, 1, 0]    |  ???  |

Step 4 - StandardScaler:
| scaled_features                       | churn |
| [0.2, 1.8, -0.5, 0.1, 0, 1.0, 0]     |  ???  |

Step 5 - LogisticRegression:
Churn Probability: 0.23 (23% likely to churn)
```

Complete Pipeline Code:

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import *
from pyspark.ml.classification import LogisticRegression

# Stage 1: Handle categorical features
segment_indexer = StringIndexer(inputCol="customer_segment",
outputCol="segment_idx")
policy_indexer = StringIndexer(inputCol="policy_type", outputCol="policy_idx")
multiple_encoder = OneHotEncoder(inputCol="multiple_policies",
outputCol="multiple_encoded")

# Stage 2: Assemble features
assembler = VectorAssembler(
    inputCols=["age", "premium_amount", "claims_frequency", "years_customer",
               "segment_idx", "policy_idx", "multiple_encoded"],
    outputCol="features"
)

# Stage 3: Scale features
scaler = StandardScaler(inputCol="features", outputCol="scaled_features")
```

```
# Stage 4: Train model
lr = LogisticRegression(featuresCol="scaled_features", labelCol="churn")

# Create and fit pipeline
pipeline = Pipeline(stages=[
    segment_indexer, policy_indexer, multiple_encoder,
    assembler, scaler, lr
])

# Train entire pipeline
pipeline_model = pipeline.fit(train_df)

# Make predictions (all stages applied automatically!)
predictions = pipeline_model.transform(test_df)
```

SECTION 3: Classification Models (15 minutes)

WHITEBOARD STEP 8: Set up classification scenario:

```
CLASSIFICATION PROBLEM: Will customer churn?

FEATURES:                    TARGET:
- Age                       - Churn (0=Stay, 1=Leave)
- Premium amount
- Claims in last year       GOAL: Predict probability
- Years as customer         of customer leaving
- Customer satisfaction
- Policy count
```

A. Logistic Regression for Classification

WHITEBOARD STEP 9: Draw the sigmoid function:

```
SIGMOID FUNCTION: P(churn=1) = 1 / (1 + e^(-z))
where z = w₁×age + w₂×premium + ... + wₙ×satisfaction + b

     1.0 ┤         ╭───── High churn risk
         │       /
P(churn) │     /
         │   /
     0.5 ┤/       ← Decision threshold
         │
     0.0 ┤         Low churn risk
         └─────────────── Risk Score (z)
        -∞      0      +∞

Key insight: Customer features → Churn probability
```

Code Example:

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Create pipeline for churn prediction
categorical_cols = ["customer_segment", "policy_type"]
numerical_cols = ["age", "premium_amount", "claims_frequency", "years_customer",
                  "satisfaction_score", "policy_count"]

# Preprocessing stages
segment_indexer = StringIndexer(inputCol="customer_segment",
outputCol="segment_idx")
policy_indexer = StringIndexer(inputCol="policy_type", outputCol="policy_idx")
assembler = VectorAssembler(inputCols=["segment_idx", "policy_idx"] +
numerical_cols,
                           outputCol="features")
scaler = StandardScaler(inputCol="features", outputCol="scaled_features")

# Classification model
lr_classifier = LogisticRegression(
    featuresCol="scaled_features",
    labelCol="churn",
    maxIter=10,
    regParam=0.01
)

# Build pipeline
churn_pipeline = Pipeline(stages=[segment_indexer, policy_indexer, assembler,
scaler, lr_classifier])
churn_model = churn_pipeline.fit(train_df)

# Generate predictions
predictions = churn_model.transform(test_df)
predictions.select("scaled_features", "churn", "probability",
"prediction").show(5)
```

B. Random Forest Classification

WHITEBOARD STEP 10: Visualize Random Forest concept:

```
RANDOM FOREST = ENSEMBLE OF DECISION TREES

Tree 1:             Tree 2:             Tree 3:
Premium > $3000?    Claims > 2?         Satisfaction < 3?
├─Yes: CHURN        ├─Yes: CHURN        ├─Yes: CHURN
└─No: STAY          └─No: STAY          └─No: STAY

VOTING FOR CUSTOMER ID 12345:
Tree 1: CHURN (75% confidence)
```

```
Tree 2: STAY (60% confidence)
Tree 3: CHURN (80% confidence)

FINAL PREDICTION: CHURN (72% average confidence)
Action: Flag for retention campaign
```

Code Example:

```
from pyspark.ml.classification import RandomForestClassifier

# Replace logistic regression with random forest
rf_classifier = RandomForestClassifier(
    featuresCol="scaled_features",
    labelCol="churn",
    numTrees=50,        # Number of trees in forest
    maxDepth=8,         # Maximum depth of each tree
    seed=42             # For reproducibility
)

# Same pipeline, different algorithm
rf_pipeline = Pipeline(stages=[segment_indexer, policy_indexer, assembler, scaler,
rf_classifier])
rf_model = rf_pipeline.fit(train_df)

# Get feature importances for business insights
feature_importances = rf_model.stages[-1].featureImportances
print("Most important churn indicators:", feature_importances)
```

SECTION 4: Regression Models (15 minutes)

WHITEBOARD STEP 11: Set up regression scenario:

```
REGRESSION PROBLEM: Predict Customer Lifetime Value (CLV)

FEATURES:                     TARGET:
- Age                        - CLV (continuous $)
- Premium amount
- Years as customer          GOAL: Predict total revenue
- Claims history             from customer over lifetime
- Policy count
- Satisfaction score
```

A. Linear Regression

WHITEBOARD STEP 12: Draw linear regression concept:

```
LINEAR REGRESSION: CLV = β₀ + β₁×Premium + β₂×Years + ... + ε

VISUALIZATION (2D example):
CLV $
 50k ┤      ●
     |    ● / ●
 30k ┤ ● /     ● (actual CLV points)
     | /
 20k ┤/
     |        / (best fit line)
 10k ┤      /
     └──────────────── Premium Amount
     1k   2k   3k

GOAL: Minimize Σ(actual CLV - predicted CLV)²
```

Code Example:

```python
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Regression pipeline for CLV prediction
numerical_features = ["age", "premium_amount", "years_customer", "claims_count",
                     "policy_count", "satisfaction_score"]

assembler = VectorAssembler(inputCols=numerical_features, outputCol="features")
scaler = StandardScaler(inputCol="features", outputCol="scaled_features")

# Linear regression model
lr_regressor = LinearRegression(
    featuresCol="scaled_features",
    labelCol="customer_lifetime_value",
    maxIter=20,
    regParam=0.1      # Regularization to prevent overfitting
)

# Build and train pipeline
clv_pipeline = Pipeline(stages=[assembler, scaler, lr_regressor])
clv_model = clv_pipeline.fit(train_df)

# Make predictions
predictions = clv_model.transform(test_df)

# Show predictions vs actual
predictions.select("customer_lifetime_value", "prediction").show(10)
```

B. Random Forest Regression

WHITEBOARD STEP 13: Compare linear vs non-linear patterns:

```
LINEAR RELATIONSHIP:         NON-LINEAR RELATIONSHIP:
CLV                          CLV
  ↑    ●                       ↑     ●●
  │ ●   ●                      │   ●●  ●
  │    ●   ●                   │ ●●      ●●
  │       ●   ●                │●           ●●
  └─────────→ Premium          └─────────────→ Premium

Linear Regression: ✓          Linear Regression: ✗
Random Forest: ✓              Random Forest: ✓

Business insight: High-value customers may have
non-linear CLV patterns based on policy combinations
```

Code Example:

```
from pyspark.ml.regression import RandomForestRegressor

# Random Forest regression for CLV
rf_regressor = RandomForestRegressor(
    featuresCol="scaled_features",
    labelCol="customer_lifetime_value",
    numTrees=100,       # More trees for better accuracy
    maxDepth=12,        # Deeper trees for complex patterns
    seed=42
)

# Same preprocessing, different algorithm
rf_clv_pipeline = Pipeline(stages=[assembler, scaler, rf_regressor])
rf_clv_model = rf_clv_pipeline.fit(train_df)

# Compare predictions
lr_predictions = clv_model.transform(test_df)
rf_predictions = rf_clv_model.transform(test_df)

print("Linear Regression vs Random Forest CLV Predictions:")
comparison = lr_predictions.select("customer_lifetime_value",
                                   col("prediction").alias("lr_pred")) \
                          .join(rf_predictions.select("customer_lifetime_value",

col("prediction").alias("rf_pred")),
                                 on="customer_lifetime_value")
comparison.show(10)
```

SECTION 5: Evaluation Metrics (18 minutes)

A. Classification Metrics

WHITEBOARD STEP 14: Create the "Classification Metrics Dashboard":

```
CHURN PREDICTION EVALUATION METRICS

CONFUSION MATRIX:
                Predicted
                Stay  Churn
Actual Stay   [TN]   [FP]    TN = Correctly predicted stays
Actual Churn  [FN]   [TP]    TP = Correctly predicted churn
                              FN = Missed churners (costly!)
                              FP = False churn alerts

ACCURACY = (TP + TN) / (TP + TN + FP + FN)
"Overall correctness"

PRECISION = TP / (TP + FP)
"Of predicted churners, how many actually churned?"

RECALL = TP / (TP + FN)
"Of actual churners, how many did we catch?"

F1-SCORE = 2 × (Precision × Recall) / (Precision + Recall)
"Balanced metric for churn prediction"
```

Real-World Example: WHITEBOARD STEP 15: Insurance business impact:

```
BUSINESS IMPACT OF CHURN PREDICTION:

High Precision Important:     High Recall Important:
- Retention campaigns        - Revenue protection
- Targeted discounts         - Market share defense
- Resource allocation        - Competitive analysis

False Positive = Unnecessary  False Negative = Lost
retention spend              high-value customer

Cost of false retention campaign: $50
Cost of losing high-value customer: $5,000

Business Strategy: Optimize for Recall!
```

Code Example:

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator,
MulticlassClassificationEvaluator

# Binary classification metrics for churn
binary_evaluator = BinaryClassificationEvaluator(
```

```
    labelCol="churn",
    rawPredictionCol="rawPrediction"
)

# Area Under ROC Curve
auc = binary_evaluator.evaluate(predictions, {binary_evaluator.metricName:
"areaUnderROC"})
print(f"AUC: {auc:.3f}")

# Area Under Precision-Recall Curve
aupr = binary_evaluator.evaluate(predictions, {binary_evaluator.metricName:
"areaUnderPR"})
print(f"AUPR: {aupr:.3f}")

# Multi-class metrics
multi_evaluator = MulticlassClassificationEvaluator(
    labelCol="churn",
    predictionCol="prediction"
)

accuracy = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName:
"accuracy"})
precision = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName:
"weightedPrecision"})
recall = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName:
"weightedRecall"})
f1 = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: "f1"})

print(f"Churn Prediction Accuracy: {accuracy:.3f}")
print(f"Precision: {precision:.3f}")
print(f"Recall: {recall:.3f}")
print(f"F1-Score: {f1:.3f}")

# Business metrics
churn_rate = predictions.filter(col("prediction") == 1.0).count() /
predictions.count()
print(f"Predicted Churn Rate: {churn_rate:.1%}")
```

B. Regression Metrics

WHITEBOARD STEP 16: Draw regression metrics visualization:

```
CLV PREDICTION EVALUATION METRICS

MEAN ABSOLUTE ERROR (MAE):
Average |actual CLV - predicted CLV|
●───●───●───● (actual CLV values)
 \   / \   /  (prediction errors)
  ● ●   ● ●    (predicted CLV)

MAE = (|e₁| + |e₂| + ... + |eₙ|) / n
```

```
Easy to interpret: "Off by $X on average"

ROOT MEAN SQUARED ERROR (RMSE):
√(Σ(actual CLV - predicted CLV)² / n)
Penalizes large CLV prediction errors heavily

R² (R-SQUARED):
1 - (SS_res / SS_tot)
0 = no better than average CLV
1 = perfect CLV prediction
```

Code Example:

```python
from pyspark.ml.evaluation import RegressionEvaluator

# Create evaluator for CLV predictions
clv_evaluator = RegressionEvaluator(
    labelCol="customer_lifetime_value",
    predictionCol="prediction"
)

# Calculate different metrics
rmse = clv_evaluator.evaluate(predictions, {clv_evaluator.metricName: "rmse"})
mae = clv_evaluator.evaluate(predictions, {clv_evaluator.metricName: "mae"})
r2 = clv_evaluator.evaluate(predictions, {clv_evaluator.metricName: "r2"})

print(f"CLV Prediction RMSE: ${rmse:,.0f}")
print(f"CLV Prediction MAE: ${mae:,.0f}")
print(f"CLV Prediction R²: {r2:.3f}")

# Business interpretation
print(f"Model explains {r2*100:.1f}% of CLV variation")
print(f"Average CLV prediction error: ${mae:,.0f}")

# Custom evaluation for business insights
from pyspark.sql.functions import abs, pow, avg, when

# Calculate error bands for business decisions
evaluation_df = predictions.withColumn("clv_error", col("customer_lifetime_value")
- col("prediction")) \
                           .withColumn("abs_error", abs(col("clv_error"))) \
                           .withColumn("error_percentage",
                                      abs(col("clv_error")) /
col("customer_lifetime_value") * 100)

# Business quality metrics
within_10_percent = evaluation_df.filter(col("error_percentage") < 10).count() /
evaluation_df.count()
within_20_percent = evaluation_df.filter(col("error_percentage") < 20).count() /
evaluation_df.count()
```

```
print(f"CLV predictions within 10%: {within_10_percent:.1%}")
print(f"CLV predictions within 20%: {within_20_percent:.1%}")
```

C. Cross-Validation for Robust Evaluation

WHITEBOARD STEP 17: Illustrate cross-validation:

```
5-FOLD CROSS-VALIDATION FOR CHURN PREDICTION

Original Data: [████████████████████] (All customers)

Fold 1: [Test Customers][Train Customers][Train][Train][Train]
Fold 2: [Train][Test Customers][Train][Train][Train]
Fold 3: [Train][Train][Test Customers][Train][Train]
Fold 4: [Train][Train][Train][Test Customers][Train]
Fold 5: [Train][Train][Train][Train][Test Customers]

Result: 5 churn prediction scores → Average ± Std Dev
More reliable than single holdout test!

Business benefit: Confident model performance estimates
```

Code Example:

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Create parameter grid for churn model hyperparameter tuning
param_grid = ParamGridBuilder() \
    .addGrid(lr_classifier.regParam, [0.01, 0.1, 1.0]) \
    .addGrid(lr_classifier.maxIter, [10, 20, 50]) \
    .build()

# Cross-validator for churn prediction
cross_validator = CrossValidator(
    estimator=churn_pipeline,
    estimatorParamMaps=param_grid,
    evaluator=binary_evaluator,
    numFolds=5,  # 5-fold cross-validation
    seed=42
)

# Fit with cross-validation
cv_churn_model = cross_validator.fit(train_df)

# Best model metrics
best_churn_model = cv_churn_model.bestModel
best_predictions = best_churn_model.transform(test_df)
best_auc = binary_evaluator.evaluate(best_predictions)
```

```
print(f"Best Churn Model AUC after CV: {best_auc:.3f}")
print(f"Best parameters: {cv_churn_model.getEstimatorParamMaps()
[cv_churn_model.avgMetrics.index(max(cv_churn_model.avgMetrics))]}")

# Business confidence intervals
cv_scores = cv_churn_model.avgMetrics
import numpy as np
mean_score = np.mean(cv_scores)
std_score = np.std(cv_scores)
print(f"Model performance: {mean_score:.3f} ± {std_score:.3f}")
print(f"95% confidence interval: [{mean_score - 2*std_score:.3f}, {mean_score +
2*std_score:.3f}]")
```

SECTION 6: Model Comparison & Selection (10 minutes)

WHITEBOARD STEP 18: Create model comparison table:

```
CHURN PREDICTION MODEL COMPARISON

CLASSIFICATION MODELS:
Model               AUC     Precision  Recall   F1-Score  Training Time  Business
Impact
Logistic Regression 0.842    0.756     0.634     0.690      2.3s          Baseline
Random Forest       0.869    0.781     0.745     0.763      8.7s
+$50K/month
Gradient Boosting   0.891    0.803     0.778     0.790     25.1s
+$75K/month

CLV PREDICTION MODELS:
Model               RMSE      MAE       R²        Training Time  Business Value
Linear Regression   $4,231   $3,106    0.734      1.9s          Interpretable
Random Forest       $3,642   $2,765    0.801     12.4s          Good accuracy
Gradient Boosting   $3,287   $2,234    0.823     25.1s          Best performance

WINNER: Gradient Boosting for both (best business ROI)
```

Code Example - Model Comparison Pipeline:

```
from pyspark.ml.classification import GBTClassifier
import time

def evaluate_churn_model(pipeline, train_data, test_data, model_name):
    """Evaluate a churn prediction model and return business metrics"""
    start_time = time.time()

    # Train model
    model = pipeline.fit(train_data)
    training_time = time.time() - start_time
```

```
    # Make predictions
    predictions = model.transform(test_data)

    # Calculate metrics
    evaluator = BinaryClassificationEvaluator(labelCol="churn",
rawPredictionCol="rawPrediction")
    auc = evaluator.evaluate(predictions)

    # Business metrics
    total_customers = predictions.count()
    predicted_churners = predictions.filter(col("prediction") == 1.0).count()
    actual_churners = predictions.filter(col("churn") == 1.0).count()

    return {
        'model': model_name,
        'auc': auc,
        'training_time': training_time,
        'predicted_churn_rate': predicted_churners / total_customers,
        'actual_churn_rate': actual_churners / total_customers
    }

# Define churn models to compare
churn_models = {
    'Logistic Regression': Pipeline(stages=[segment_indexer, policy_indexer,
assembler, scaler,

LogisticRegression(featuresCol="scaled_features", labelCol="churn")]),
    'Random Forest': Pipeline(stages=[segment_indexer, policy_indexer, assembler,
scaler,

RandomForestClassifier(featuresCol="scaled_features", labelCol="churn",
numTrees=50)]),
    'Gradient Boosting': Pipeline(stages=[segment_indexer, policy_indexer,
assembler, scaler,

GBTClassifier(featuresCol="scaled_features", labelCol="churn", maxIter=20)])
}

# Evaluate all models
results = []
for name, pipeline in churn_models.items():
    result = evaluate_churn_model(pipeline, train_df, test_df, name)
    results.append(result)
    print(f"{name}: AUC={result['auc']:.3f}, Churn Rate=
{result['predicted_churn_rate']:.1%}, Time={result['training_time']:.1f}s")

# Find best model for business
best_model = max(results, key=lambda x: x['auc'])
print(f"\nBest Churn Model: {best_model['model']} (AUC: {best_model['auc']:.3f})")
```

SECTION 7: Production Deployment (8 minutes)

WHITEBOARD STEP 19: Draw production workflow:

```
PRODUCTION CHURN PREDICTION WORKFLOW

Development:                 Production:
[Customer Data] → [Pipeline] → [Model]  [New Customers] → [Saved Model] → [Churn
Scores]
       ↓                                        ↓              ↓               ↓
[Training]                              [Batch Scoring]   [Load Model]
[Retention Actions]

BUSINESS APPLICATIONS:
✓ Daily churn risk scoring
✓ Automated retention campaigns
✓ Customer segmentation
✓ Proactive customer service
✓ CLV-based resource allocation
```

Code Example:

```
# Save the best churn prediction model
best_churn_pipeline_model = churn_models['Gradient Boosting'].fit(train_df)
churn_model_path = "/models/insurance_churn_predictor_v1"
best_churn_pipeline_model.save(churn_model_path)

# Load model for inference
from pyspark.ml.pipeline import PipelineModel
loaded_churn_model = PipelineModel.load(churn_model_path)

# Batch churn scoring function
def batch_churn_scoring(model, new_customers_path, output_path):
    """Apply churn model to new customers and save risk scores"""
    new_customers = spark.read.parquet(new_customers_path)
    churn_predictions = model.transform(new_customers)

    # Create business-ready output
    risk_scores = churn_predictions.select("customer_id",

col("probability").getItem(1).alias("churn_probability"),

col("prediction").alias("churn_prediction"),
                                          "features") \
                                  .withColumn("risk_category",
                                            when(col("churn_probability") > 0.8,
"High Risk")
                                            .when(col("churn_probability") > 0.5,
"Medium Risk")
                                            .otherwise("Low Risk")) \
```

```
                                   .withColumn("scoring_date", current_timestamp())
\
                                   .withColumn("model_version", lit("v1"))

    risk_scores.write.mode("overwrite").parquet(output_path)
    return risk_scores.count()

# Real-time churn scoring function
def predict_customer_churn(model, customer_id, age, premium, claims, years,
segment, policy_type, satisfaction):
    """Predict churn probability for a single customer"""
    from pyspark.sql.types import StructType, StructField, StringType,
IntegerType, DoubleType

    schema = StructType([
        StructField("customer_id", StringType()),
        StructField("age", IntegerType()),
        StructField("premium_amount", DoubleType()),
        StructField("claims_frequency", IntegerType()),
        StructField("years_customer", IntegerType()),
        StructField("customer_segment", StringType()),
        StructField("policy_type", StringType()),
        StructField("satisfaction_score", DoubleType())
    ])

    single_customer = spark.createDataFrame([(customer_id, age, premium, claims,
years, segment, policy_type, satisfaction)], schema)
    prediction = model.transform(single_customer)
    churn_prob = prediction.select("probability").collect()[0][0][1]  # Get
probability of churn (class 1)
    return churn_prob

# Example churn prediction
churn_probability = predict_customer_churn(loaded_churn_model, "CUST_12345", 45,
2500, 1, 5, "Premium", "Auto", 7.5)
print(f"Customer CUST_12345 churn probability: {churn_probability:.1%}")

# Business action trigger
if churn_probability > 0.7:
    print("Action: Trigger high-value retention campaign")
elif churn_probability > 0.4:
    print("Action: Schedule proactive customer service call")
else:
    print("Action: Continue standard customer journey")
```

SECTION 8: Best Practices & Tips (7 minutes)

WHITEBOARD STEP 20: Create best practices checklist:

```
✅ INSURANCE ML PIPELINE BEST PRACTICES

🔧 DEVELOPMENT:
  ├── Always use pipelines (no manual steps!)
  ├── Fit transformers on training data only
  ├── Use cross-validation for model selection
  ├── Log all experiments and parameters
  └── Version your data and models

🚀 PRODUCTION:
  ├── Monitor churn prediction drift
  ├── Implement automated model retraining
  ├── Use A/B testing for retention strategies
  ├── Have model rollback mechanisms ready
  └── Log all predictions for audit compliance

⚠️ INSURANCE-SPECIFIC PITFALLS:
  ├── Regulatory compliance requirements
  ├── Seasonal churn pattern changes
  ├── Economic factor impact on models
  ├── Customer lifecycle stage variations
  └── Policy renewal timing effects

💼 BUSINESS INTEGRATION:
  ├── Align predictions with business cycles
  ├── Consider retention campaign costs
  ├── Integrate with CRM systems
  ├── Measure actual retention ROI
  └── Provide model explainability for decisions
```

Code Example - MLflow Integration:

```
import mlflow
import mlflow.spark
from mlflow.tracking import MlflowClient

# Start MLflow experiment for insurance churn
mlflow.set_experiment("insurance_churn_prediction")

# Log churn model training with MLflow
with mlflow.start_run(run_name="churn_gradient_boosting_v1"):
    # Train model
    gbt_churn_pipeline = Pipeline(stages=[segment_indexer, policy_indexer,
assembler, scaler,

GBTClassifier(featuresCol="scaled_features", labelCol="churn")])
    gbt_churn_model = gbt_churn_pipeline.fit(train_df)

    # Make predictions
    predictions = gbt_churn_model.transform(test_df)
```

```
    # Calculate metrics
    evaluator = BinaryClassificationEvaluator(labelCol="churn",
rawPredictionCol="rawPrediction")
    auc = evaluator.evaluate(predictions)

    # Business metrics
    predicted_churn_rate = predictions.filter(col("prediction") == 1.0).count() /
predictions.count()
    high_risk_customers = predictions.filter(col("probability").getItem(1) >
0.8).count()

    # Log parameters
    mlflow.log_param("algorithm", "GBTClassifier")
    mlflow.log_param("num_features", len(numerical_cols) + len(categorical_cols))
    mlflow.log_param("max_iter", 20)
    mlflow.log_param("business_domain", "insurance_churn")

    # Log metrics
    mlflow.log_metric("auc", auc)
    mlflow.log_metric("predicted_churn_rate", predicted_churn_rate)
    mlflow.log_metric("high_risk_customers", high_risk_customers)

    # Log model
    mlflow.spark.log_model(gbt_churn_model, "churn_model")

    print(f"Churn model logged with AUC: {auc:.3f}")
    print(f"Predicted churn rate: {predicted_churn_rate:.1%}")
    print(f"High-risk customers identified: {high_risk_customers}")
```

SECTION 9: Advanced Pipeline Techniques (5 minutes)

A. Feature Selection in Churn Pipelines

WHITEBOARD STEP 21: Show feature selection integration:

```
CHURN FEATURE SELECTION PIPELINE

Raw Features (15) → [Feature Selection] → Top Features (8) → [Churn Model]
                          ↓
                   ChiSqSelector
                   (Statistical churn indicators)
                          ↓
                Remove weak predictors
                Focus on key churn drivers
                Improve model interpretability

Key Churn Indicators Often Selected:
- Claims frequency (high correlation)
- Satisfaction scores (strong predictor)
```

```
- Premium amount (threshold effects)
- Years as customer (loyalty curve)
```

Code Example:

```
from pyspark.ml.feature import ChiSqSelector

# Enhanced churn pipeline with feature selection
enhanced_churn_pipeline = Pipeline(stages=[
    # Preprocessing
    StringIndexer(inputCol="customer_segment", outputCol="segment_idx"),
    StringIndexer(inputCol="policy_type", outputCol="policy_idx"),
    VectorAssembler(inputCols=numerical_cols + ["segment_idx", "policy_idx"],
outputCol="features"),
    StandardScaler(inputCol="features", outputCol="scaled_features"),

    # Feature selection for churn prediction
    ChiSqSelector(featuresCol="scaled_features", outputCol="selected_features",
                  numTopFeatures=8),  # Keep only top 8 churn predictors

    # Model training
    RandomForestClassifier(featuresCol="selected_features", labelCol="churn",
numTrees=100)
])

# Train with feature selection
enhanced_churn_model = enhanced_churn_pipeline.fit(train_df)

# Get selected features for business insights
feature_selector = enhanced_churn_model.stages[-2]  # Get the ChiSqSelector stage
selected_indices = feature_selector.selectedFeatures
all_feature_names = numerical_cols + ["segment_idx", "policy_idx"]
selected_features = [all_feature_names[i] for i in selected_indices]
print("Most important churn predictors:", selected_features)
```

B. Hyperparameter Tuning for Business Optimization

Code Example:

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Create parameter grid optimized for business metrics
param_grid = ParamGridBuilder() \
    .addGrid(enhanced_churn_pipeline.getStages()[-1].numTrees, [50, 100, 200]) \
    .addGrid(enhanced_churn_pipeline.getStages()[-1].maxDepth, [8, 12, 16]) \
    .addGrid(enhanced_churn_pipeline.getStages()[-2].numTopFeatures, [6, 8, 10]) \
    .build()

# Custom evaluator for business optimization (maximize recall for churn)
```

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
recall_evaluator = MulticlassClassificationEvaluator(
    labelCol="churn",
    predictionCol="prediction",
    metricName="weightedRecall"  # Optimize for catching churners
)

# Cross-validation optimizing for business recall
cv = CrossValidator(
    estimator=enhanced_churn_pipeline,
    estimatorParamMaps=param_grid,
    evaluator=recall_evaluator,
    numFolds=3,
    parallelism=4
)

# Find best parameters for churn detection
best_churn_model = cv.fit(train_df)
print(f"Best recall-optimized parameters:")
print(f"Trees: {best_churn_model.bestModel.stages[-1].getNumTrees()}")
print(f"Features: {best_churn_model.bestModel.stages[-2].getNumTopFeatures()}")
```

SECTION 10: Real-World Insurance Case Study (8 minutes)

WHITEBOARD STEP 22: Draw the complete system architecture:

```
INSURANCE CUSTOMER RETENTION SYSTEM

CUSTOMER DATA SOURCES:
- Policy transactions, claims history
- Customer service interactions
- Payment patterns, demographics
- External credit/risk data

BUSINESS DATA:
- Market competition, pricing
- Economic indicators, seasonality
- Product portfolio, profitability

              ↓
    [FEATURE ENGINEERING PIPELINE]
              ↓
         [ML PIPELINE]
    Churn Prediction + CLV Estimation
              ↓
     [BUSINESS DECISION ENGINE]
              ↓
         [RETENTION ACTIONS]
              ↓
      [ROI MEASUREMENT]
```

Complete End-to-End Insurance Example:

```
# Insurance customer retention pipeline
from pyspark.ml.recommendation import ALS
from pyspark.ml.classification import GBTClassifier

# Step 1: Comprehensive Feature Engineering Pipeline
insurance_feature_pipeline = Pipeline(stages=[
    # Handle categorical features
    StringIndexer(inputCol="customer_segment", outputCol="segment_idx"),
    StringIndexer(inputCol="policy_type", outputCol="policy_idx"),
    StringIndexer(inputCol="state", outputCol="state_idx"),
    OneHotEncoder(inputCols=["segment_idx", "policy_idx", "state_idx"],
                  outputCols=["segment_encoded", "policy_encoded",
"state_encoded"]),

    # Assemble all features including derived ones
    VectorAssembler(inputCols=["age", "premium_amount", "claims_frequency",
"years_customer",
                               "satisfaction_score", "policy_count",
"claim_amount_last_year",
                               "payment_delay_frequency", "competitor_quotes",
                               "segment_encoded", "policy_encoded",
"state_encoded"],
                   outputCol="features"),

    # Scale features
    StandardScaler(inputCol="features", outputCol="scaled_features")
])

# Step 2: Multi-objective pipeline (churn + CLV)
churn_stage = GBTClassifier(
    featuresCol="scaled_features",
    labelCol="churn",
    probabilityCol="churn_probability",
    maxIter=20
)

# Step 3: Create complete business pipeline
complete_insurance_pipeline =
Pipeline(stages=insurance_feature_pipeline.getStages() + [churn_stage])

# Step 4: Train with comprehensive MLflow tracking
with mlflow.start_run(run_name="insurance_retention_system_v1"):
    # Train model
    retention_model = complete_insurance_pipeline.fit(train_df)

    # Evaluate on multiple business metrics
    predictions = retention_model.transform(test_df)

    # Churn prediction metrics
```

```
    auc = BinaryClassificationEvaluator(labelCol="churn",
rawPredictionCol="rawPrediction").evaluate(predictions)

    # Business impact calculations
    high_risk_churners = predictions.filter(col("churn_probability").getItem(1) >
0.8).count()
    medium_risk_churners = predictions.filter(
        (col("churn_probability").getItem(1) > 0.5) &
        (col("churn_probability").getItem(1) <= 0.8)
    ).count()

    # Revenue impact estimation
    avg_customer_value = train_df.agg(avg("customer_lifetime_value")).collect()[0]
[0]
    potential_revenue_at_risk = high_risk_churners * avg_customer_value

    # Log comprehensive metrics
    mlflow.log_metric("churn_auc", auc)
    mlflow.log_metric("high_risk_customers", high_risk_churners)
    mlflow.log_metric("medium_risk_customers", medium_risk_churners)
    mlflow.log_metric("revenue_at_risk", potential_revenue_at_risk)

    # Log business parameters
    mlflow.log_param("retention_threshold", 0.5)
    mlflow.log_param("high_risk_threshold", 0.8)
    mlflow.log_param("avg_customer_value", avg_customer_value)

    # Log model
    mlflow.spark.log_model(retention_model, "retention_model")

    print(f"Retention Model AUC: {auc:.3f}")
    print(f"High-risk customers: {high_risk_churners}")
    print(f"Revenue at risk: ${potential_revenue_at_risk:,.0f}")

# Step 5: Production business decision function
def business_retention_decision(customer_id, churn_probability, customer_value):
    """Make business decision based on churn risk and customer value"""

    # Retention campaign costs
    basic_campaign_cost = 50
    premium_campaign_cost = 200
    vip_campaign_cost = 500

    # Decision logic based on risk and value
    if churn_probability > 0.8:
        if customer_value > 10000:
            return {
                "action": "VIP Retention Campaign",
                "cost": vip_campaign_cost,
                "expected_roi": customer_value * 0.7 - vip_campaign_cost,
                "priority": "High"
            }
        elif customer_value > 5000:
            return {
```

```
                "action": "Premium Retention Campaign",
                "cost": premium_campaign_cost,
                "expected_roi": customer_value * 0.6 - premium_campaign_cost,
                "priority": "Medium"
            }
        else:
            return {
                "action": "Basic Retention Campaign",
                "cost": basic_campaign_cost,
                "expected_roi": customer_value * 0.4 - basic_campaign_cost,
                "priority": "Low"
            }
    elif churn_probability > 0.5:
        if customer_value > 8000:
            return {
                "action": "Proactive Customer Service",
                "cost": 25,
                "expected_roi": customer_value * 0.3 - 25,
                "priority": "Medium"
            }
        else:
            return {
                "action": "Automated Email Campaign",
                "cost": 5,
                "expected_roi": customer_value * 0.2 - 5,
                "priority": "Low"
            }
    else:
        return {
            "action": "Monitor",
            "cost": 0,
            "expected_roi": 0,
            "priority": "None"
        }

# Example business decision
decision = business_retention_decision("CUST_12345", 0.75, 8500)
print(f"Business Decision: {decision}")
```

SECTION 11: Performance Optimization (5 minutes)

WHITEBOARD STEP 23: Create performance optimization guide:

```
🚀 INSURANCE ML PIPELINE PERFORMANCE OPTIMIZATION

💾 DATA OPTIMIZATIONS:
  ├── Use Delta Lake for customer data versioning
  ├── Partition by policy renewal dates
  ├── Cache customer feature tables
  ├── Optimize joins with policy lookup tables
```

```
    └── Incremental feature computation

⚡ MODEL OPTIMIZATIONS:
    ├── Feature selection reduces churn model complexity
    ├── Early stopping in GBT training
    ├── Parallel cross-validation for hyperparameters
    ├── Model ensembling for critical decisions
    └── Batch scoring optimization

🔧 BUSINESS OPTIMIZATIONS:
    ├── Real-time vs batch prediction optimization
    ├── Customer segmentation for targeted models
    ├── Seasonal model retraining schedules
    ├── A/B testing framework for campaigns
    └── ROI-based model selection criteria
```

Code Example:

```
# Performance-optimized insurance pipeline

# 1. Optimize data loading and caching
from delta.tables import DeltaTable

# Use Delta Lake for customer data
customer_features_delta = DeltaTable.forPath(spark, "/delta/customer_features")
train_df = customer_features_delta.toDF().filter(col("snapshot_date") == "2024-01-
01")
train_df.cache()  # Cache training data

# 2. Optimize Spark configuration for insurance workloads
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")  # Handle customer
data skew

# 3. Incremental feature engineering
def incremental_feature_update(base_path, new_data_path, output_path):
    """Update customer features incrementally"""
    base_features = spark.read.delta(base_path)
    new_data = spark.read.parquet(new_data_path)

    # Compute new features only for changed customers
    updated_features = new_data.join(base_features, "customer_id", "left_anti") \
                              .union(base_features.join(new_data, "customer_id",
"left_semi"))

    # Write back to Delta Lake
    updated_features.write.format("delta").mode("overwrite").save(output_path)

# 4. Parallel model training with optimal resource allocation
cv_optimized = CrossValidator(
    estimator=complete_insurance_pipeline,
```

```
    estimatorParamMaps=param_grid,
    evaluator=recall_evaluator,
    numFolds=3,
    parallelism=spark.sparkContext.defaultParallelism,  # Use all available cores
    seed=42
)

# 5. Efficient batch churn scoring
def optimized_batch_churn_scoring(model, customer_batch_size=50000):
    """Optimized batch churn scoring for large customer base"""

    # Read customer data in optimized format
    customers = spark.read.format("delta").load("/delta/active_customers") \
                   .repartition(200)  # Optimal partitioning

    total_customers = customers.count()
    num_batches = (total_customers + customer_batch_size - 1) //
customer_batch_size

    for batch_id in range(num_batches):
        batch_customers = customers.limit(customer_batch_size).offset(batch_id *
customer_batch_size)

        # Score batch
        batch_predictions = model.transform(batch_customers)

        # Write results with business metadata
        scored_batch = batch_predictions.select(
            "customer_id",
            col("churn_probability").getItem(1).alias("churn_risk_score"),
            col("prediction").alias("churn_prediction"),
            current_timestamp().alias("scoring_timestamp"),
            lit(f"batch_{batch_id}").alias("batch_id")
        )

        # Write to Delta Lake for downstream consumption
        scored_batch.write.format("delta") \
                   .mode("append") \
                   .option("mergeSchema", "true") \
                   .save("/delta/churn_scores")

        print(f"Processed batch {batch_id + 1}/{num_batches}")

# 6. Monitor performance metrics
def monitor_model_performance():
    """Monitor churn model performance in production"""
    current_scores = spark.read.format("delta").load("/delta/churn_scores")

    # Calculate performance drift
    score_distribution = current_scores.groupBy("churn_prediction").count()
    avg_risk_score = current_scores.agg(avg("churn_risk_score")).collect()[0][0]

    print(f"Current churn prediction distribution:")
    score_distribution.show()
```

```
    print(f"Average churn risk score: {avg_risk_score:.3f}")

    # Alert if distribution changes significantly
    if avg_risk_score > 0.6:  # Historical average was 0.45
        print("⚠  ALERT: Churn risk scores significantly higher than historical average")
        print("   Action: Consider model retraining or market analysis")
```

CLOSING & CALL TO ACTION (5 minutes)

WHITEBOARD STEP 24: Create the learning roadmap:

```
🎯 YOUR INSURANCE ML MASTERY ROADMAP

FOUNDATION (Week 1-2):          INTERMEDIATE (Week 3-4):
✓ Customer churn pipelines       → Advanced feature engineering
✓ Classification vs CLV          → Business metric optimization
✓ Insurance evaluation metrics  → Cross-validation strategies
✓ Basic model comparison       → Regulatory compliance

ADVANCED (Week 5-6):           PRODUCTION (Week 7-8):
→ Ensemble churn models         → Real-time scoring systems
→ Customer segmentation         → A/B testing campaigns
→ Performance optimization      → ROI measurement & tracking
→ Retention decision engines    → Automated model monitoring

🚀 INSURANCE ML CHALLENGES:
1. Build churn model for YOUR insurance domain
2. Implement customer lifetime value prediction
3. Create automated retention decision system
4. Set up A/B testing for campaign effectiveness
5. Develop regulatory-compliant model documentation!
```

Final Insurance ML Code Challenge:

```
# YOUR CHALLENGE: Complete this insurance ML pipeline template!

def build_insurance_ml_system(business_objective="churn_prevention"):
    """
    Build a complete insurance ML system

    TODO for students:
    1. Add domain-specific insurance features
    2. Implement business-optimized algorithms
    3. Include regulatory compliance measures
    4. Add ROI-based decision logic
    5. Create A/B testing framework
    """
```

```
    # Feature engineering for insurance
    insurance_features = [
        # TODO: Add your insurance-specific transformers
        # Examples: policy_age_calculator, claim_frequency_aggregator
    ]

    # Model selection based on business objective
    if business_objective == "churn_prevention":
        models = [
            LogisticRegression(featuresCol="features", labelCol="churn"),
            GBTClassifier(featuresCol="features", labelCol="churn"),
            # TODO: Add ensemble methods
        ]
        evaluation_metric = "areaUnderROC"
    elif business_objective == "clv_optimization":
        models = [
            LinearRegression(featuresCol="features",
labelCol="customer_lifetime_value"),
            GBTRegressor(featuresCol="features",
labelCol="customer_lifetime_value"),
            # TODO: Add advanced regression models
        ]
        evaluation_metric = "r2"

    # TODO: Implement business decision logic
    # TODO: Add compliance and audit trails
    # TODO: Include A/B testing framework
    # TODO: Create ROI measurement system

    return best_business_pipeline

# Start building YOUR insurance ML system today!
# my_insurance_system = build_insurance_ml_system("churn_prevention")
```

Key Takeaways:

```
💡 INSURANCE ML SUCCESS FACTORS:
✓ Pipelines ensure regulatory compliance and auditability
✓ Business metrics matter more than technical metrics
✓ Customer retention ROI drives model selection
✓ Real-time scoring enables proactive interventions
✓ A/B testing validates campaign effectiveness

🎯 YOUR NEXT STEPS:
1. Identify high-value customer segments in your data
2. Build churn prediction pipeline with business rules
3. Implement automated retention decision system
4. Measure actual business impact and ROI
5. Scale to real-time customer interaction systems

Questions? Let's solve insurance challenges together! 💼
```

APPENDIX: Quick Reference

```python
# Essential Insurance ML Pipeline Pattern
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, StandardScaler
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# 1. Define stages for churn prediction
segment_indexer = StringIndexer(inputCol="customer_segment",
outputCol="segment_idx")
assembler = VectorAssembler(inputCols=["age", "premium", "claims", "segment_idx"],
outputCol="features")
scaler = StandardScaler(inputCol="features", outputCol="scaled_features")
churn_model = GBTClassifier(featuresCol="scaled_features", labelCol="churn")

# 2. Create pipeline
pipeline = Pipeline(stages=[segment_indexer, assembler, scaler, churn_model])

# 3. Train with business focus
trained_pipeline = pipeline.fit(train_data)

# 4. Predict churn risks
predictions = trained_pipeline.transform(test_data)

# 5. Evaluate with business metrics
evaluator = BinaryClassificationEvaluator(labelCol="churn",
rawPredictionCol="rawPrediction")
auc = evaluator.evaluate(predictions)

# 6. Deploy for business decisions
trained_pipeline.save("/models/churn_model_v1")

# 7. Business decision logic
def retention_decision(churn_prob, customer_value):
    if churn_prob > 0.8 and customer_value > 10000:
        return "VIP_retention_campaign"
    elif churn_prob > 0.5:
        return "standard_retention_offer"
    else:
        return "monitor_only"
```

TOTAL PRESENTATION TIME: ~90 minutes BUSINESS FOCUS: Insurance Customer Retention & Value Optimization