AI Algorithms and Machine Learning for Mobile Apps

Practical Exercise – AI Integration, Neural Networks, and Intelligent Agents

Introduction to Mobile AI and Machine Learning

Course: Mobile Application Development (Chapter 10)
Level: Technical University Students
Duration: 9–11 hours self-study + 2–2.5 hours online discussion
Format: Individual Assignment with ML Model Implementation (7–9 pages)

Exercise Overview

In this exercise, you will master comprehensive AI and machine learning integration strategies for mobile applications: understanding fundamental ML algorithms suitable for mobile environments, implementing neural networks using TensorFlow Lite and Core ML, designing recommendation engines and personalization systems, exploring multi-agent systems for collaborative scenarios, and deploying intelligent agents that enhance user experience through predictive analytics and adaptive behavior.

Key Learning Outcomes

- Understand AI/ML fundamentals: supervised learning, unsupervised learning, reinforcement learning
- Master algorithm selection for mobile constraints (latency, memory, battery, storage)
- Implement neural networks using TensorFlow Lite (Android) and Core ML (iOS)
- Design and implement recommendation engines for personalized user experiences
- Understand multi-agent systems and their application in collaborative mobile scenarios
- Implement intelligent agents that learn user preferences and adapt behavior
- Integrate pre-trained models (BERT, MobileNet, etc.) for efficient mobile inference
- Develop on-device ML pipelines for privacy-preserving personalization
- Deploy real-time prediction systems with inference optimization techniques

- Conduct performance testing and optimization for mobile ML models
- Evaluate model accuracy, latency, and memory footprint trade-offs

---

Part A: AI/ML Fundamentals and Algorithm Selection

Task 1: ML Algorithms and Mobile Constraints (Self-Study: 2.5–3 hours)

Objective: Master fundamental ML algorithms and their suitability for mobile environments.

Instructions:

1. Machine Learning Paradigms Overview

Supervised Learning

- Training with labeled examples (input → output pairs)
- Goal: Learn mapping function from features to target
- Examples: Email classification (spam/not spam), house price prediction, image recognition
- Mobile use cases: User activity classification, recommendation ranking, content filtering

Unsupervised Learning

- Training with unlabeled data (discover patterns)
- Goal: Find hidden structure or groupings
- Examples: Clustering similar users, anomaly detection, topic modeling
- Mobile use cases: User segmentation, fraud detection, behavior clustering

Reinforcement Learning

- Agent learns through interaction and rewards/penalties
- Goal: Maximize cumulative reward through exploration
- Examples: Game AI, robot control, recommendation optimization
- Mobile use cases: Personalized notification timing, adaptive UI, game AI

Semi-Supervised Learning

- Combination of labeled and unlabeled data
- Goal: Leverage both for improved learning

- Mobile use cases: Active learning systems, smart data labeling

2. Algorithm Selection Framework for Mobile

Algorithm	Complexity	Latency	Memory	Accuracy	Best For	Mobile Rating
Logistic Regression	Low	<10ms	<5MB	Good	Binary classification, simple patterns	★★★★★
Decision Tree	Medium	10-50ms	5-20MB	Good	Feature interactions, interpretable	★★★★
Random Forest	High	50-200ms	20-100MB	Very Good	Complex patterns, ensemble learning	★★★
K-Nearest Neighbors	Medium	20-100ms	Variable	Good	Similarity-based, flexible	★★★
Support Vector Machine	High	50-500ms	10-50MB	Very Good	High-dimensional data	★★
Naive Bayes	Low	<10ms	<5MB	Good	Text classification, probability	★★★★★
Neural Network	Very High	50-1000ms	50-500MB	Excellent	Complex patterns, raw inputs	★★★ (with optimization)
Linear Regression	Low	<10ms	<5MB	Good	Continuous prediction	★★★★★
Gradient Boosting	High	100-500ms	20-200MB	Excellent	Tabular data, competitions	★★
K-Means Clustering	Medium	50-200ms	10-50MB	Good	User segmentation, grouping	★★★

3. Mobile ML Constraints and Trade-Offs

Latency Requirements:
Real-time (<100ms):
├─ User interaction feedback (typing prediction, autocomplete)
├─ On-device inference (classification, detection)
└─ Example: Predictive text input (Google Gboard)

Near Real-time (100-500ms):
├─ API results ranking and filtering
├─ Background model inference
└─ Example: Search results personalization

Batch (<1-5 seconds):
├─ Model updates, periodic scoring
├─ Background processing
└─ Example: Daily recommendation refresh

Memory Constraints:
Mobile Device Memory Breakdown:
├─ System OS: 1-2 GB
├─ Other apps running: 500MB-1GB
├─ Available for single app: 500MB-2GB
├─ App code/assets: 50-200MB
├─ Runtime data: 100-500MB
└─ ML models: 10-100MB (constraint!)

Battery Impact:
Operations ranked by battery drain:

1. Network I/O (400mW)
2. GPU computation (200-300mW)
3. CPU computation (100-200mW)
4. Reading from storage (50mW)
5. On-device inference (CPU): 50-100mW
6. On-device inference (GPU): 100-200mW
7. Memory access: 10-20mW

Optimization Strategies:

Quantization (Reduce precision):

- Float32 (32-bit) → Int8 (8-bit)
- Model size reduction: 25-75%
- Speed improvement: 2-4x
- Accuracy loss: <1% for most models

Pruning (Remove unimportant weights):

- Remove 30-50% of weights

- Model size: 30-50% reduction
- Speed: 10-30% improvement
- Accuracy loss: <1-2%

Knowledge Distillation:

- Train small model to mimic large model
- Model size: 10-100x smaller
- Inference speed: 2-10x faster
- Accuracy: Near-equivalent to large model

4. Common ML Algorithms for Mobile

Naive Bayes (Text Classification):

Example: Spam email detection

P(Spam | words) = P(words | Spam) * P(Spam) / P(words)

Training: Count word frequencies in spam vs. ham
Inference: Multiply probability of each word
Result: Probability email is spam

Logistic Regression (Binary Classification):

// Pseudocode: User engagement prediction
data class UserFeatures(
val sessionTime: Float, // minutes
val clickCount: Int, // number of clicks
val scrollDepth: Float, // percentage
val timeOfDay: Int // hour (0-23)
)

fun predictEngaged(features: UserFeatures): Float {
// Trained weights from historical data
val w0 = -0.5f // bias
val w1 = 0.01f // session time weight
val w2 = 0.02f // click count weight
val w3 = 0.03f // scroll depth weight
val w4 = 0.001f // time of day weight

```
val z = w0 + w1 * features.sessionTime +
        w2 * features.clickCount +
        w3 * features.scrollDepth +
        w4 * features.timeOfDay

// Sigmoid activation: outputs probability [0, 1]
return sigmoid(z)
```

}

fun sigmoid(x: Float): Float = 1.0f / (1.0f + exp(-x))

Decision Tree (Feature-Based Classification):

User Engagement Prediction Tree:

```
                [Session > 2min?]
                /               \
             YES                 NO
             |                   |
      [Clicks > 5?]          Return NOT_ENGAGED
      /           \
   YES             NO
   |               |
```

ENGAGED [Scroll > 50%?]
/
YES NO
| |
ENGAGED NOT_ENGAGED

K-Means Clustering (User Segmentation):

// Pseudocode: Segment users into 3 groups
data class UserProfile(val spendingAmount: Float, val sessionFrequency: Int)

class UserSegmenter {
val clusters = listOf(
Cluster(center = UserProfile(100f, 5)), // Power users
Cluster(center = UserProfile(50f, 2)), // Regular users
Cluster(center = UserProfile(10f, 0.5f)) // Casual users
)

```
fun segmentUser(user: UserProfile): String {
    val distances = clusters.map { cluster ->
        distance(user, cluster.center)
    }
    return when (distances.indexOf(distances.minOrNull())) {
        0 -> "POWER_USER"
        1 -> "REGULAR_USER"
        else -> "CASUAL_USER"
    }
}

fun distance(u1: UserProfile, u2: UserProfile): Float {
    val dSpend = (u1.spendingAmount - u2.spendingAmount).pow(2)
    val dFreq = (u1.sessionFrequency - u2.sessionFrequency).pow(2)
    return sqrt(dSpend + dFreq)
}
```

}

Deliverable:

- [ ] ML paradigm comparison table with mobile use cases
- [ ] Algorithm selection framework with 8+ algorithms
- [ ] Mobile constraints analysis (latency, memory, battery)
- [ ] Optimization techniques explanation with trade-offs
- [ ] Three implemented algorithm examples (Naive Bayes, Logistic Regression, K-Means)
- [ ] Performance characteristic summary for each algorithm

Task 2: Feature Engineering and Data Preparation (Self-Study: 2–2.5 hours)

Objective: Master feature engineering and data pipeline setup for mobile ML.

Instructions:

1. Feature Engineering Process

Feature Types:

Numerical Features:

- Raw numbers: User age, session duration, purchase amount
- Derived: Average session length, total spend, engagement score

Categorical Features:

- Categories: Device type, user segment, content genre
- Encoded: One-hot encoding, label encoding, embedding encoding

Temporal Features:

- Time-based: Hour of day, day of week, time since last action
- Sequence-based: User action history, behavior patterns

Text Features:

- Bag-of-words: Word frequency, TF-IDF scores
- Embeddings: Word2Vec, FastText, BERT embeddings

Image Features:

- Raw pixels (but expensive for mobile)
- CNN embeddings (feature vectors from neural networks)
- Extracted features (edges, colors, shapes)

2. Feature Engineering Pipeline

// Android example: Mobile ML feature pipeline
class FeatureEngineer {

```
// Raw data collection
data class UserSession(
    val userId: String,
    val timestamp: Long,
    val screenViewDuration: Long,
    val clickCount: Int,
    val scrollDistance: Float,
    val deviceType: String,
    val appVersion: String
)

// Feature extraction
fun extractFeatures(session: UserSession): FloatArray {
    val calendar = Calendar.getInstance().apply { timeInMillis =
session.timestamp }

    return floatArrayOf(
        // Numerical features (normalize to [0, 1])
        session.screenViewDuration.toFloat() / 60000,  // minutes (max
10)
        minOf(session.clickCount.toFloat() / 20, 1f),  // clicks (max
20)
        minOf(session.scrollDistance / 5000, 1f),      // scroll (max
5000px)

        // Temporal features (normalized)
        calendar.get(Calendar.HOUR_OF_DAY).toFloat() / 24,
        calendar.get(Calendar.DAY_OF_WEEK).toFloat() / 7,

        // Categorical features (one-hot encoded)
        if (session.deviceType == "PHONE") 1f else 0f,
        if (session.deviceType == "TABLET") 1f else 0f,

        // Version encoding
        if (session.appVersion.startsWith("4")) 1f else 0f,
        if (session.appVersion.startsWith("5")) 1f else 0f
    )
}

// Feature normalization (standardization)
fun normalizeFeatures(features: FloatArray, mean: FloatArray, std:
FloatArray): FloatArray {
    return FloatArray(features.size) { i ->
        (features[i] - mean[i]) / std[i]
    }
```

```
}

// Feature scaling (min-max normalization)
fun scaleFeatures(features: FloatArray, min: FloatArray, max:
FloatArray): FloatArray {
    return FloatArray(features.size) { i ->
        (features[i] - min[i]) / (max[i] - min[i])
    }
}
```

```
}
```

// iOS example using Swift
class FeatureEngineer {
struct UserSession {
let userId: String
let timestamp: Date
let screenViewDuration: TimeInterval
let clickCount: Int
let scrollDistance: Float
let deviceType: String
let appVersion: String
}

```
func extractFeatures(session: UserSession) -> [Float] {
    let calendar = Calendar.current
    let hour = calendar.component(.hour, from: session.timestamp)
    let dayOfWeek = calendar.component(.weekday, from:
session.timestamp)

    return [
        // Numerical features (normalized)
        Float(session.screenViewDuration) / 60,      // minutes
        Float(min(session.clickCount, 20)) / 20,     // clicks
        min(session.scrollDistance / 5000, 1),       // scroll

        // Temporal features
        Float(hour) / 24,
        Float(dayOfWeek) / 7,

        // Categorical (one-hot)
        session.deviceType == "PHONE" ? 1 : 0,
        session.deviceType == "TABLET" ? 1 : 0
    ]
}

func normalizeFeatures(_ features: [Float], mean: [Float], std:
[Float]) -> [Float] {
    return zip(zip(features, mean), std).map { (values, stdDev) in
        (values.0 - values.1) / stdDev
    }
```

```
}
```

}

3. Data Quality and Handling

Missing Values:

- Deletion: Remove samples with missing values (if <5% missing)
- Mean imputation: Replace with feature mean
- Model-based imputation: Use another model to predict missing values
- Forward fill: Use previous value (for time series)

Outliers:

- Statistical method: Values beyond 3 standard deviations
- IQR method: Values beyond 1.5 * IQR
- Domain knowledge: Business rules for invalid values
- Handling: Remove, cap, or transform

Class Imbalance:

- Downsampling: Remove samples from majority class
- Upsampling: Duplicate samples from minority class
- SMOTE: Generate synthetic minority samples
- Cost weighting: Give higher weight to minority class

4. Feature Storage and Deployment

// Store computed features efficiently on device
class FeatureStore {
fun saveFeatures(userId: String, features: FloatArray) {
val database = Room.databaseBuilder(context, FeatureDatabase::class.java,
"features").build()
val entity = FeatureEntity(userId, features, System.currentTimeMillis())
database.featureDao().insert(entity)
}

```
fun loadFeatures(userId: String): FloatArray? {
    val database = Room.databaseBuilder(context,
FeatureDatabase::class.java, "features").build()
    return database.featureDao().getLatestFeatures(userId)?.features
}

fun clearOldFeatures(maxAgeDays: Int) {
    val cutoffTime = System.currentTimeMillis() - (maxAgeDays * 24 * 60
* 60 * 1000)
    database.featureDao().deleteOlderThan(cutoffTime)
```

```
}

}

// Efficient data structures for mobile
@Entity(tableName = "features")
data class FeatureEntity(
@PrimaryKey val userId: String,
val features: FloatArray, // Serialized as BLOB
val timestamp: Long
)

@Dao
interface FeatureDao {
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insert(feature: FeatureEntity)

@Query("SELECT * FROM features WHERE userId = :userId ORDER BY
timestamp DESC LIMIT 1")
suspend fun getLatestFeatures(userId: String): FeatureEntity?

@Query("DELETE FROM features WHERE timestamp < :cutoffTime")
suspend fun deleteOlderThan(cutoffTime: Long)

}
```

Deliverable:

- [ ] Feature engineering framework with four feature types
- [ ] Feature extraction pipeline implementation (Android or iOS)
- [ ] Feature normalization and scaling examples
- [ ] Data quality handling guide (missing values, outliers, imbalance)
- [ ] Feature storage optimization strategy
- [ ] Performance characteristics of feature extraction

Part B: Neural Networks and Deep Learning on Mobile

Task 3: TensorFlow Lite and Core ML Integration (Self-Study: 2.5–3 hours)

Objective: Master neural network deployment on mobile using TensorFlow Lite and Core ML.

Instructions:

1. Neural Network Basics

Neural Network Architecture:

Input Layer Hidden Layers Output Layer
(5 neurons) (4 neurons) (2 neurons)

Input1 ----
Input2 ----+--> H1 ----
Input3 ----+--> H2 ----+--> Output1 (Probability)
Input4 ----+--> H3 ----+--> Output2 (Probability)
Input5 ----/ H4 ----/

Forward Pass:
z = Input × Weights + Bias
a = Activation(z)

Activation Functions:

- ReLU: max(0, x) — fast, sparse
- Sigmoid: 1/(1+e^-x) — probability output
- Tanh: (e^x - e^-x)/(e^x + e^-x) — zero-centered
- Softmax: e^x / sum(e^x) — multi-class probability

2. Android TensorFlow Lite Integration

Setup:

// build.gradle
dependencies {
implementation 'org.tensorflow:tensorflow-lite:2.13.0'
implementation 'org.tensorflow:tensorflow-lite-gpu-delegate:2.13.0'
implementation 'org.tensorflow:tensorflow-lite-support:0.4.4'
}

Model Inference (User Engagement Prediction):

import org.tensorflow.lite.Interpreter
import org.tensorflow.lite.support.tensorbuffer.TensorBuffer
import java.nio.MappedByteBuffer
import java.nio.channels.FileChannel

class EngagementPredictor(context: Context) {
private lateinit var interpreter: Interpreter
private val inputSize = 10 // Number of input features

```
init {
    // Load model from assets
    val modelBytes = loadModelFile(context, "engagement_model.tflite")
    interpreter = Interpreter(modelBytes)
```

```
}

private fun loadModelFile(context: Context, modelName: String):
MappedByteBuffer {
    val assetFileDescriptor = context.assets.openFd(modelName)
    val inputStream = assetFileDescriptor.createInputStream()
    val fileChannel = (inputStream as FileInputStream).channel
    val startOffset = assetFileDescriptor.startOffset
    val declaredLength = assetFileDescriptor.declaredLength
    return fileChannel.map(FileChannel.MapMode.READ_ONLY, startOffset,
declaredLength)
}

fun predictEngagement(features: FloatArray): EngagementResult {
    // Prepare input tensor
    val inputShape = intArrayOf(1, inputSize)
    val inputBuffer = TensorBuffer.createFixedSize(inputShape,
DataType.FLOAT32)
    inputBuffer.loadArray(features)

    // Allocate output buffer
    val outputShape = intArrayOf(1, 2)  // 2 classes: engaged, not
engaged
    val outputBuffer = TensorBuffer.createFixedSize(outputShape,
DataType.FLOAT32)

    // Run inference
    interpreter.run(inputBuffer.buffer, outputBuffer.buffer)

    // Parse results
    val output = outputBuffer.floatArray
    val engagementProb = output[0]
    val disengagementProb = output[1]

    return EngagementResult(
        isEngaged = engagementProb > 0.5f,
        engagementScore = engagementProb,
        confidence = maxOf(engagementProb, disengagementProb)
    )
}

fun close() {
    interpreter.close()
}
```

}

data class EngagementResult(
val isEngaged: Boolean,
val engagementScore: Float,

val confidence: Float
)

// Usage
val predictor = EngagementPredictor(context)
val features = floatArrayOf(2.5f, 15f, 0.8f, 14f, 3f, 1f, 0f, 0f, 1f, 0f)
val result = predictor.predictEngagement(features)

if (result.isEngaged && result.confidence > 0.85f) {
showEngagementContent()
} else {
showRetentionOffer()
}

GPU Acceleration:

class OptimizedEngagementPredictor(context: Context) {
private lateinit var interpreter: Interpreter
private var gpuDelegate: GpuDelegate? = null

```
init {
    val modelBytes = loadModelFile(context, "engagement_model.tflite")

    // Enable GPU acceleration
    gpuDelegate = GpuDelegate()
    val options = Interpreter.Options().addDelegate(gpuDelegate)

    interpreter = Interpreter(modelBytes, options)
}

fun close() {
    interpreter.close()
    gpuDelegate?.close()
}
```

}

3. iOS Core ML Integration

Setup:

import CoreML
import Vision

class EngagementPredictor {
let model: EngagementModel

```
init() {
    self.model = try! EngagementModel(configuration: .init())
}

func predictEngagement(features: [Double]) -> EngagementResult {
    // Prepare input (model expects specific format)
```

```
    let input = EngagementModelInput(
        sessionTime: features[0],
        clickCount: features[1],
        scrollDepth: features[2],
        timeOfDay: features[3],
        dayOfWeek: features[4],
        isPhone: features[5],
        isTablet: features[6],
        isVersion4: features[7],
        isVersion5: features[8],
        score: 0  // Dummy, will be predicted
    )

    // Run inference
    guard let output = try? model.prediction(input: input) else {
        return EngagementResult(isEngaged: false, score: 0, confidence:
0)
    }

    return EngagementResult(
        isEngaged: output.isEngaged == 1,
        score: Float(output.engagementScore),
        confidence: Float(output.confidence)
    )
}
```

}

struct EngagementResult {
let isEngaged: Bool
let score: Float
let confidence: Float
}

// Usage
let predictor = EngagementPredictor()
let features = [2.5, 15, 0.8, 14, 3, 1, 0, 0, 1, 0]
let result = predictor.predictEngagement(features: features)

if result.isEngaged && result.confidence > 0.85 {
showEngagementContent()
}

Model Updates:

// Update model on background thread
DispatchQueue.global().async {
// Download new model from server
let newModelURL = self.downloadLatestModel()

```
// Verify model hash for security
if self.verifyModelHash(url: newModelURL) {
```

```
    // Replace old model with new one
    try? FileManager.default.replaceItemAt(
        self.modelURL,
        withItemAt: newModelURL
    )

    // Reload predictor on main thread
    DispatchQueue.main.async {
        self.predictor = EngagementPredictor()
    }
}
```

}

4. Performance Optimization

Model Quantization:

// During model conversion (Python TensorFlow)
import tensorflow as tf

def quantize_model(saved_model_dir, output_file):
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

```
# Enable post-training quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Optional: Set representative dataset for better accuracy
# converter.representative_dataset = representative_data_gen

quantized_model = converter.convert()

with open(output_file, 'wb') as f:
    f.write(quantized_model)

# Result: 25% model size, 2-4x faster, <1% accuracy loss
```

Lazy Loading:

class LazyModelProvider(context: Context) {
private var predictor: EngagementPredictor? = null
private val context = context

```
fun getPredictor(): EngagementPredictor {
    if (predictor == null) {
        predictor = EngagementPredictor(context)
    }
    return predictor!!
}

fun releaseModel() {
```

```
    predictor?.close()
    predictor = null
}
```

}

Deliverable:

- [ ] Neural network architecture explanation with diagrams
- [ ] Android TensorFlow Lite implementation (complete example)
- [ ] iOS Core ML implementation (complete example)
- [ ] GPU acceleration setup and benefits
- [ ] Model quantization explanation and implementation
- [ ] Performance benchmarks (latency, memory, battery) for optimized models

---

Task 4: Pre-trained Models and Transfer Learning (Self-Study: 2–2.5 hours)

Objective: Master integration of pre-trained models for efficient mobile ML.

Instructions:

1. Common Pre-trained Models for Mobile

Image Classification (MobileNet):

MobileNet v2 Comparison:

Model Size Latency Accuracy
MobileNet 3.5 MB 10-30ms 71.3%
MobileNetV2 14 MB 30-50ms 72%
Inception 90 MB 200+ms 76.7%
ResNet 170 MB 500+ms 76%

Mobile choice: MobileNet (optimized for mobile)

Natural Language Processing (BERT):

BERT Model Sizes:

BERT-Base 340 MB 1000-2000ms Excellent
BERT-Small 27 MB 200-500ms Good
DistilBERT 26 MB 100-300ms Good
MobileBERT 25 MB 50-150ms Good

Mobile choice: MobileBERT (4.3% smaller, 4.3x faster)

Object Detection (YOLO, SSD):

Detection Model Comparison:

Model Size Latency Accuracy
YOLOv3-Tiny 34 MB 100-200ms 76% mAP
SSD-MobileNet 16 MB 50-100ms 72% mAP
EfficientDet 60-260 MB 100-1000ms 85%+ mAP

Mobile choice: SSD-MobileNet (fast, accurate)

2. Transfer Learning for Custom Tasks

Android Transfer Learning (Image Classification Fine-tuning):

// Fine-tune MobileNet for custom app classification
class CustomImageClassifier(context: Context) {
private val tfLiteModel = loadModel(context, "mobilenet_v2.tflite")

```
fun classifyImage(bitmap: Bitmap): ClassificationResult {
    // Preprocess image
    val tensor = TensorImage(DataType.UINT8)
    tensor.load(bitmap)

    // Extract features (not final classification)
    val tfLiteInterpreter = Interpreter(tfLiteModel)

    // Get intermediate layer output (features)
    val features = FloatArray(1280)  // MobileNet feature vector size

    // Run inference to get features
    val input = arrayOf<Any>(tensor.buffer)
    val output = mapOf(0 to features)
    tfLiteInterpreter.runForMultipleInputsOutputs(input, output)

    // Classify using fine-tuned head
    val prediction = classifyWithCustomHead(features)

    return ClassificationResult(
        label = prediction.label,
        confidence = prediction.score
    )
}

private fun classifyWithCustomHead(features: FloatArray): Prediction {
    // Trained on device-specific examples
    val weights = floatArrayOf(...)  // Your fine-tuned weights
    val bias = floatArrayOf(...)

    val score = sigmoid(features.zip(weights).sumOf { it.first *
it.second }.toFloat() + bias[0])

    return Prediction(
        label = if (score > 0.5) "CustomClass1" else "CustomClass2",
```

```
        score = score
    )
}
```

}

iOS Transfer Learning:

import CoreML
import Vision

class CustomImageClassifier {
let featureExtractor: VNCoreMLRequest
let customClassifier: CustomHeadModel

```
init() {
    // Load pre-trained feature extractor (MobileNet)
    let model = try! MobileNetV2(configuration: .init())
    featureExtractor = VNCoreMLRequest(model: model.model) { request,
error in
        // Extract features
    }

    // Load custom classification head
    customClassifier = try! CustomHeadModel(configuration: .init())
}

func classifyImage(_ image: UIImage) -> String {
    // Step 1: Extract features using MobileNet
    let features = extractFeatures(image)

    // Step 2: Classify with custom head
    let input = CustomHeadModelInput(features: features)
    let output = try! customClassifier.prediction(input: input)

    return output.classLabel
}

private func extractFeatures(_ image: UIImage) -> MLMultiArray {
    let request = VNCoreMLRequest(model: featureExtractor.model)
    let handler = VNImageRequestHandler(cgImage: image.cgImage!)
    try! handler.perform([request])

    // Extract intermediate layer
    return request.results?.first as? MLMultiArray ?? MLMultiArray()
}
```

}

3. Model Selection Framework

Decision Tree for Model Selection:

```
Task Type?
├─ Classification
│  ├─ Images → MobileNet / EfficientNet
│  ├─ Text → MobileBERT / DistilBERT
│  └─ Tabular → Logistic Regression / Tree Ensemble
├─ Detection
│  ├─ Objects → SSD-MobileNet / YOLOv3-Tiny
│  ├─ Text → CRAFT / DBNet
│  └─ Faces → Face-Detection-LFFD
├─ Segmentation
│  ├─ Semantic → DeepLabV3-MobileNet
│  ├─ Instance → Mask R-CNN (heavyweight)
│  └─ Portrait → MediaPipe Selfie Segmenter
└─ NLP
├─ Sentiment → DistilBERT / MobileBERT
├─ Intent → MobileBERT-based custom
└─ NER → MobileBERT-based custom
```

Deliverable:

- [ ] Comparison table of popular pre-trained models
- [ ] Transfer learning explanation with diagram
- [ ] Android custom classifier implementation (fine-tuning example)
- [ ] iOS custom classifier implementation (fine-tuning example)
- [ ] Model selection framework for different tasks
- [ ] Performance benchmarks for transfer learning scenarios

Part C: Recommendation Engines and Personalization

Task 5: Recommendation Engines and Collaborative Filtering (Self-Study: 2.5–3 hours)

Objective: Master recommendation engines and AI-driven personalization strategies.

Instructions:

1. Recommendation Algorithms

Content-Based Filtering:

User → Item Features → Similarity → Recommendations

Example: Movie recommendation based on genres
User watched: [Action, Sci-Fi]
Item to recommend: Genres = [Action, Sci-Fi, Adventure]
Similarity score: 2/3 = 0.67 (match on 2 genres)

Pros:

- No "cold start" for new items
- Transparent recommendations
- Works with unpopular items

Cons:

- Limited discovery (similar items only)
- Requires good item features
- Can't leverage user behavior patterns

Collaborative Filtering:

Similar Users → Item Ratings → Recommendations

User A rated: {Movie1: 5★, Movie2: 4★, Movie3: 3★}
User B rated: {Movie1: 5★, Movie2: 4★, Movie4: 5★}

Similarity(A, B) = 2 common high ratings → HIGH
→ Recommend Movie4 to User A

Hybrid Approach (Recommended for Mobile):

Content-Based (20%)
↓
Collaborative (40%) → Combined Score → Ranking
↓
Popularity (40%)

Benefits:

- Handles cold start (new users/items)
- Diverse recommendations
- Balanced between discovery and relevance

2. Android Recommendation Engine

class RecommendationEngine(context: Context) {
private val database = AppDatabase.getInstance(context)
private val userPreferences = UserPreferencesStore(context)

```
// Recommendation data structure
data class Recommendation(
    val itemId: String,
    val score: Float,
    val reason: String
```

```
)

fun getRecommendations(userId: String, limit: Int = 5):
List<Recommendation> {
    // Get user's interaction history
    val userHistory = database.interactionDao().getUserHistory(userId)
    val userFeatures = extractUserFeatures(userHistory)

    // Get all items
    val allItems = database.itemDao().getAllItems()

    // Score each item
    val scoredItems = allItems.map { item ->
        val contentScore = calculateContentScore(userFeatures, item)
        val collaborativeScore = calculateCollaborativeScore(userId,
item)
        val popularityScore = calculatePopularityScore(item)

        val totalScore = (contentScore * 0.2f +
                          collaborativeScore * 0.4f +
                          popularityScore * 0.4f)

        Recommendation(
            itemId = item.id,
            score = totalScore,
            reason = selectRecommendationReason(contentScore,
collaborativeScore)
        )
    }

    // Return top recommendations
    return scoredItems.sortedByDescending { it.score }.take(limit)
}

private fun calculateContentScore(userFeatures: UserFeatures, item:
Item): Float {
    // Calculate similarity between user preferences and item features
    return cosineSimilarity(userFeatures.vector, item.featureVector)
}

private fun calculateCollaborativeScore(userId: String, item: Item):
Float {
    // Find similar users and their ratings for this item
    val similarUsers = database.userDao().findSimilarUsers(userId,
limit = 10)

    if (similarUsers.isEmpty()) return 0f

    val ratings = similarUsers.mapNotNull { user ->
        database.ratingDao().getRating(user.id, item.id)
```

```
    }

    if (ratings.isEmpty()) return 0f

    // Average rating from similar users (normalized to [0, 1])
    return ratings.map { it.score }.average().toFloat() / 5.0f
}

private fun calculatePopularityScore(item: Item): Float {
    // Normalize popularity: engagement count / max count
    val allItemCounts = database.itemDao().getAllPopularityCounts()
    val maxCount = allItemCounts.maxOrNull() ?: 1
    return item.engagementCount.toFloat() / maxCount
}

private fun cosineSimilarity(vec1: FloatArray, vec2: FloatArray): Float
{
    val dotProduct = vec1.zip(vec2).sumOf { it.first * it.second
}.toFloat()
    val magnitude1 = sqrt(vec1.sumOf { it * it }.toFloat())
    val magnitude2 = sqrt(vec2.sumOf { it * it }.toFloat())

    return if (magnitude1 > 0 && magnitude2 > 0) {
        dotProduct / (magnitude1 * magnitude2)
    } else 0f
}

private fun selectRecommendationReason(content: Float, collaborative:
Float): String {
    return when {
        collaborative > 0.7f → "Users like you enjoyed this"
        content > 0.7f → "Based on your interests"
        else → "Trending now"
    }
}
```

```
}

// Usage
val engine = RecommendationEngine(context)
val recommendations = engine.getRecommendations(userId = "user123", limit = 5)

recommendations.forEach { rec ->
println("${rec.itemId}: ${rec.score} - ${rec.reason}")
}
```

3. iOS Recommendation Implementation

```
class RecommendationEngine {
let database: AppDatabase
let userPreferences: UserPreferencesStore
```

```
struct Recommendation {
    let itemId: String
    let score: Float
    let reason: String
}

func getRecommendations(userId: String, limit: Int = 5) ->
[Recommendation] {
    // Get user history
    let userHistory = database.getUserHistory(userId)
    let userVector = extractUserVector(from: userHistory)

    // Score all items
    let allItems = database.getAllItems()
    var recommendations: [Recommendation] = []

    for item in allItems {
        let contentScore = calculateContentScore(userVector,
item.vector)
        let collaborativeScore = calculateCollaborativeScore(userId,
itemId: item.id)
        let popularityScore = calculatePopularityScore(item)

        let totalScore = (contentScore * 0.2 +
                          collaborativeScore * 0.4 +
                          popularityScore * 0.4)

        let reason: String
        if collaborativeScore > 0.7 {
            reason = "Users like you enjoyed this"
        } else if contentScore > 0.7 {
            reason = "Based on your interests"
        } else {
            reason = "Trending now"
        }

        recommendations.append(Recommendation(
            itemId: item.id,
            score: Float(totalScore),
            reason: reason
        ))
    }

    return recommendations.sorted { $0.score > $1.score
}.prefix(limit).map { $0 }
}

private func calculateContentScore(_ userVector: [Float], _ itemVector:
[Float]) -> Double {
    let dotProduct = zip(userVector, itemVector).reduce(0) { $0 + $1.0
```

```
* $1.1 }
    let mag1 = sqrt(userVector.reduce(0) { $0 + $1 * $1 })
    let mag2 = sqrt(itemVector.reduce(0) { $0 + $1 * $1 })

    return mag1 > 0 && mag2 > 0 ? Double(dotProduct) / Double(mag1 *
mag2) : 0
}

private func calculateCollaborativeScore(_ userId: String, itemId:
String) -> Double {
    let similarUsers = database.findSimilarUsers(userId, limit: 10)
    let ratings = similarUsers.compactMap { user in
        database.getRating(userId: user.id, itemId: itemId)
    }

    guard !ratings.isEmpty else { return 0 }

    let average = ratings.reduce(0) { $0 + $1 } / Double(ratings.count)
    return average / 5.0
}

private func calculatePopularityScore(_ item: Item) -> Double {
    let allCounts = database.getAllEngagementCounts()
    let maxCount = allCounts.max() ?? 1
    return Double(item.engagementCount) / Double(maxCount)
}
```

```
}
```

4. A/B Testing Recommendations

// A/B test different recommendation strategies
class RecommendationABTest {
enum Strategy { CONTENT_BASED, COLLABORATIVE, HYBRID }

```
fun assignUserToVariant(userId: String): Strategy {
    // Hash-based consistent assignment
    val hash = userId.hashCode().toLong() and 0xFFFFFFFFL
    return when (hash % 3) {
        0L -> Strategy.CONTENT_BASED
        1L -> Strategy.COLLABORATIVE
        else -> Strategy.HYBRID
    }
}

fun recordMetric(userId: String, itemId: String, metric: String, value:
Float) {
    val variant = assignUserToVariant(userId)
    // Log: variant, metric, value for analysis
```

```
}
```

}

Deliverable:

- [ ] Recommendation algorithm comparison table
- [ ] Content-based filtering explanation with example
- [ ] Collaborative filtering explanation with example
- [ ] Hybrid approach implementation (Android or iOS)
- [ ] Cosine similarity calculation and usage
- [ ] A/B testing framework for recommendation strategies
- [ ] Performance metrics for recommendation quality

Task 6: Personalization and Adaptive User Experience (Self-Study: 2–2.5 hours)

Objective: Master AI-driven personalization and adaptive UI/UX systems.

Instructions:

1. Personalization Strategies

User Profiling:

```
// User preference model
data class UserProfile(
val userId: String,
val preferences: PreferencesVector, // Float array of learned preferences
val behaviors: BehaviorPatterns, // User behavior model
val demographics: Demographics, // Age, location, device type
val lastUpdated: Long
)

data class PreferencesVector(
val genrePreferences: FloatArray, // [0-1] for each genre
val contentLengthPreference: Float, // 0-1 (short-long)
val socialWeight: Float, // 0-1 (trending weight)
val personalizedWeight: Float // 0-1 (personalization weight)
)

data class BehaviorPatterns(
val clickThroughRate: Float,
val avgSessionLength: Long,
val peakActivityHour: Int,
val weeklyEngagement: FloatArray // 7 values
)
```

Dynamic Content Ranking:

class PersonalizedRanker(val userProfile: UserProfile) {

```
fun rankContent(items: List<ContentItem>): List<ContentItem> {
    return items.map { item ->
        val relevanceScore = calculateRelevance(item, userProfile)
        val timinessScore = calculateTimelineScore(item)
        val noveltyScore = calculateNovelty(item, userProfile)

        val finalScore = (relevanceScore * 0.6f +
                          timinessScore * 0.2f +
                          noveltyScore * 0.2f)

        item to finalScore
    }.sortedByDescending { it.second }
     .map { it.first }
}

private fun calculateRelevance(item: ContentItem, profile:
UserProfile): Float {
    // Score based on user's preference vector
    return cosineSimilarity(item.featureVector,
profile.preferences.genrePreferences)
}

private fun calculateTimelineScore(item: ContentItem): Float {
    // Recent content scores higher (decay over time)
    val ageHours = (System.currentTimeMillis() - item.publishedAt) /
(1000 * 3600)
    return (1 / (1 + 0.1f * ageHours)).coerceIn(0f, 1f)
}

private fun calculateNovelty(item: ContentItem, profile: UserProfile):
Float {
    // Content user hasn't seen similar items before
    val seen = profile.behaviors.viewedItems.contains(item.id)
    return if (seen) 0.3f else 1f
}
```

}

2. Adaptive UI Personalization

// Adaptive layout selection
class AdaptiveUIController(val userProfile: UserProfile) {

```
fun getLayoutVariant(screenSize: ScreenSize, userSegment: UserSegment):
LayoutVariant {
    return when {
        // Power users: advanced features
```

```
        userSegment == UserSegment.POWER_USER -> {
            when (screenSize) {
                ScreenSize.PHONE -> LayoutVariant.PHONE_ADVANCED
                ScreenSize.TABLET -> LayoutVariant.TABLET_ADVANCED
            }
        }
        // New users: simplified, onboarding
        userSegment == UserSegment.NEW_USER -> {
            when (screenSize) {
                ScreenSize.PHONE -> LayoutVariant.PHONE_ONBOARDING
                ScreenSize.TABLET -> LayoutVariant.TABLET_ONBOARDING
            }
        }
        // Regular users: standard experience
        else -> {
            when (screenSize) {
                ScreenSize.PHONE -> LayoutVariant.PHONE_STANDARD
                ScreenSize.TABLET -> LayoutVariant.TABLET_STANDARD
            }
        }
    }
}

fun getRecommendedFeatures(userSegment: UserSegment): List<Feature> {
    // Show features most likely to engage user
    return when (userSegment) {
        UserSegment.POWER_USER -> listOf(
            Feature.ADVANCED_FILTERS,
            Feature.CUSTOM_SORTING,
            Feature.SAVE_COLLECTIONS
        )
        UserSegment.NEW_USER -> listOf(
            Feature.TUTORIAL,
            Feature.SUGGESTED_ITEMS,
            Feature.HELP_CENTER
        )
        UserSegment.CASUAL_USER -> listOf(
            Feature.TRENDING_SECTION,
            Feature.PUSH_RECOMMENDATIONS
        )
    }
}
```

}

enum class UserSegment {
NEW_USER, // <7 days
CASUAL_USER, // <1 session/week
REGULAR_USER, // 1-3 sessions/week

POWER_USER // >3 sessions/week
}

Timing Optimization:

class NotificationOptimizer(val userProfile: UserProfile) {

```
fun getOptimalNotificationTime(userId: String): Long {
    // Predict user engagement: pick hour with peak engagement
    val peakHour = userProfile.behaviors.peakActivityHour
    val nextOccurrence = calculateNextOccurenceOfHour(peakHour)

    return nextOccurrence
}

fun shouldShowNotification(contentType: String): Boolean {
    // Only notify about content user has shown interest in
    val relevanceThreshold = 0.6f
    val estimatedRelevance = estimateRelevance(contentType,
userProfile)

    return estimatedRelevance > relevanceThreshold
}

private fun estimateRelevance(contentType: String, profile:
UserProfile): Float {
    // Check user's historical engagement with this content type
    return
profile.preferences.genrePreferences[contentTypeToIndex(contentType)]
}
```

}

3. Privacy-Preserving Personalization

// On-device personalization (data stays on device)
class PrivatePersonalizer {

```
fun personalizeLocally(
    userInteractions: List<Interaction>,
    items: List<ContentItem>
): List<ContentItem> {
    // All computation happens on-device
    // No user data sent to server

    // Extract user preferences locally
    val preferences =
extractPreferencesFromInteractions(userInteractions)

    // Score items based on local preferences
    return items.map { item ->
```

```
        item to calculateScore(item, preferences)
    }.sortedByDescending { it.second }
     .map { it.first }
}

private fun extractPreferencesFromInteractions(interactions:
List<Interaction>): Preferences {
    // Aggregate user behavior without storing raw data
    val likeCount = interactions.count { it.type == LIKE }
    val viewCount = interactions.count { it.type == VIEW }
    val shareCount = interactions.count { it.type == SHARE }

    return Preferences(
        engagementScore = likeCount.toFloat() / (viewCount + 1),
        shareability = shareCount.toFloat() / (likeCount + 1)
    )
}
```

}

Deliverable:

- [ ] User profiling data structures with preference vectors
- [ ] Personalized content ranking algorithm with three scoring components
- [ ] Adaptive UI implementation for different user segments
- [ ] Notification timing optimization framework
- [ ] Privacy-preserving on-device personalization example
- [ ] A/B testing setup for personalization strategies

Part D: Multi-Agent Systems and Intelligent Agents

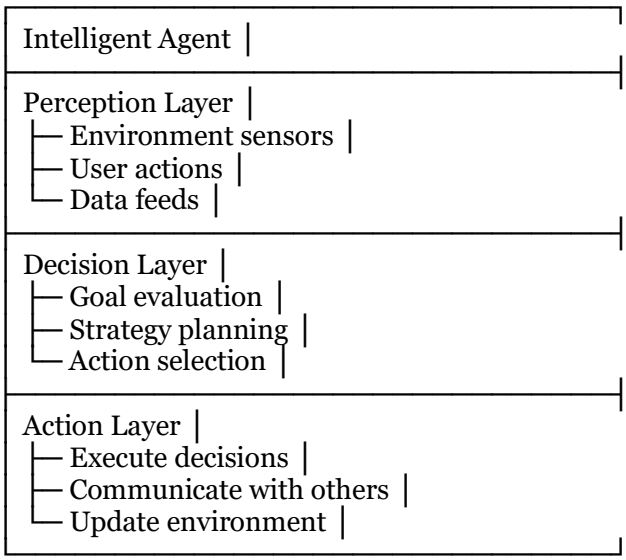
Task 7: Multi-Agent Systems and Collaborative Scenarios (Self-Study: 2–2.5 hours)

Objective: Master multi-agent systems design and implementation for mobile collaborative apps.

Instructions:

1. Multi-Agent System Concepts

Agent Architecture:



Agent Types:

Reactive Agent:

- Respond immediately to environment
- No complex reasoning
- Example: Immediate push notification response

Deliberative Agent:

- Plan actions based on goals
- Reason about consequences
- Example: Smart expense categorization

Collaborative Agent:

- Work with other agents
- Negotiate and share information
- Example: Team task scheduling

2. Android Multi-Agent Implementation

```
// Multi-agent expense management system
sealed class Agent {
abstract val agentId: String
abstract fun perceive(environment: Environment): List<Percept>
abstract fun decide(percepts: List<Percept>): Action
abstract fun act(action: Action, environment: Environment)
}

// Categorization Agent
class CategorizationAgent : Agent() {
override val agentId = "categorizer"
```

```
override fun perceive(environment: Environment): List<Percept> {
    return environment.getUncategorizedExpenses()
        .map { Percept.ExpenseDetected(it) }
}

override fun decide(percepts: List<Percept>): Action {
    return percepts.mapNotNull { percept ->
        if (percept is Percept.ExpenseDetected) {
            val category = predictCategory(percept.expense)
            Action.CategorizeExpense(percept.expense.id, category)
        } else null
    }.firstOrNull() ?: Action.Idle
}

override fun act(action: Action, environment: Environment) {
    if (action is Action.CategorizeExpense) {
        environment.updateExpenseCategory(action.expenseId,
action.category)
    }
}

private fun predictCategory(expense: Expense): String {
    // Use ML model to predict category
    val features = extractExpenseFeatures(expense)
    return categoryClassifier.predict(features)
}

}
```

// Budget Monitoring Agent
class BudgetMonitorAgent : Agent() {
override val agentId = "budget_monitor"

```
override fun perceive(environment: Environment): List<Percept> {
    val budgets = environment.getAllBudgets()
    val spending = environment.getCurrentSpending()

    return budgets.map { budget ->
        val percentUsed = spending[budget.category] ?: 0f /
budget.limit
        Percept.BudgetStatus(budget, percentUsed)
    }
}

override fun decide(percepts: List<Percept>): Action {
    for (percept in percepts) {
        if (percept is Percept.BudgetStatus) {
            return when {
                percept.percentUsed > 0.9f ->
Action.AlertBudgetExceeded(percept.budget.category)
```

```
                percept.percentUsed > 0.7f ->
Action.WarnBudgetWarning(percept.budget.category)
                else -> Action.Idle
            }
        }
    }
    return Action.Idle
}

override fun act(action: Action, environment: Environment) {
    when (action) {
        is Action.AlertBudgetExceeded -> environment.sendAlert(
            "Budget exceeded for ${action.category}!"
        )
        is Action.WarnBudgetWarning -> environment.sendNotification(
            "Warning: 70% of ${action.category} budget used"
        )
        else -> {} // Idle
    }
}
```

}

// Recommendation Agent
class RecommendationAgent : Agent() {
override val agentId = "recommender"

```
override fun perceive(environment: Environment): List<Percept> {
    val userBehavior = environment.getUserBehavior()
    val availableOffers = environment.getAvailableOffers()

    return listOf(Percept.OpportunitiesDetected(userBehavior,
availableOffers))
}

override fun decide(percepts: List<Percept>): Action {
    if (percepts[0] is Percept.OpportunitiesDetected) {
        val (behavior, offers) = percepts[0] as
Percept.OpportunitiesDetected
        val matching = offers.filter { it.matchesUserProfile(behavior)
}
        return Action.RecommendOffers(matching)
    }
    return Action.Idle
}

override fun act(action: Action, environment: Environment) {
    if (action is Action.RecommendOffers) {
        action.offers.forEach { offer ->
            environment.showRecommendation(offer)
```

```
        }
    }
}
```

}

// Multi-Agent Coordinator
class MultiAgentCoordinator(
val agents: List<Agent>,
val environment: Environment
) {
fun runCycle() {
// Perception phase
val allPercepts = agents.flatMap { it.perceive(environment) }

```
    // Decision phase
    val decisions = agents.map { agent ->
        agent.agentId to agent.decide(allPercepts)
    }

    // Conflict resolution (if multiple agents want to act)
    val resolvedActions = resolveConflicts(decisions)

    // Action phase
    resolvedActions.forEach { (agentId, action) ->
        val agent = agents.find { it.agentId == agentId }
        agent?.act(action, environment)
    }
}

private fun resolveConflicts(decisions: List<Pair<String, Action>>):
List<Pair<String, Action>> {
    // Example: Prioritize budget alerts over recommendations
    return decisions.sortedBy { (_, action) ->
        when (action) {
            is Action.AlertBudgetExceeded -> 0
            is Action.WarnBudgetWarning -> 1
            is Action.RecommendOffers -> 2
            else -> 3
        }
    }
}
```

}

// Usage
val environment = ExpenseEnvironment()
val agents = listOf(
CategorizationAgent(),
BudgetMonitorAgent(),
RecommendationAgent()

)
val coordinator = MultiAgentCoordinator(agents, environment)

// Run agent cycle
coordinator.runCycle()

Data Classes:

sealed class Percept {
data class ExpenseDetected(val expense: Expense) : Percept()
data class BudgetStatus(val budget: Budget, val percentUsed: Float) : Percept()
data class OpportunitiesDetected(val behavior: UserBehavior, val offers: List<Offer>) :
Percept()
}

sealed class Action {
object Idle : Action()
data class CategorizeExpense(val expenseId: String, val category: String) : Action()
data class AlertBudgetExceeded(val category: String) : Action()
data class WarnBudgetWarning(val category: String) : Action()
data class RecommendOffers(val offers: List<Offer>) : Action()
}

3. Agent Communication Protocol

// Agent-to-agent messaging
class AgentMessageBroker {
private val messageQueues = mutableMapOf<String, Queue<Message>>()

```
data class Message(
    val senderId: String,
    val recipientId: String,
    val content: String,
    val timestamp: Long
)

fun sendMessage(message: Message) {
    messageQueues.getOrPut(message.recipientId) { LinkedList() }
        .offer(message)
}

fun receiveMessages(agentId: String): List<Message> {
    return messageQueues[agentId]?.drainTo(mutableListOf()) ?:
emptyList()
}
```

}

// Agent negotiation example
class NegotiatingAgent(val agentId: String, val broker: AgentMessageBroker) {
fun proposeTask(taskId: String, otherAgentId: String) {
val proposal = AgentMessageBroker.Message(
senderId = agentId,

recipientId = otherAgentId,
content = "PROPOSE_TASK:$taskId",
timestamp = System.currentTimeMillis()
)
broker.sendMessage(proposal)
}

```
fun respondToProposal(proposalId: String, otherAgentId: String, accept:
Boolean) {
    val response = AgentMessageBroker.Message(
        senderId = agentId,
        recipientId = otherAgentId,
        content = if (accept) "ACCEPT:$proposalId" else
"REJECT:$proposalId",
        timestamp = System.currentTimeMillis()
    )
    broker.sendMessage(response)
}

fun processIncomingMessages() {
    val messages = broker.receiveMessages(agentId)
    for (message in messages) {
        when {
            message.content.startsWith("PROPOSE_TASK") ->
handleProposal(message)
            message.content.startsWith("ACCEPT") ->
handleAcceptance(message)
            message.content.startsWith("REJECT") ->
handleRejection(message)
        }
    }
}

private fun handleProposal(message: AgentMessageBroker.Message) {
    // Evaluate if we can accept task
    val canAccept = evaluateCapacity()
    respondToProposal(message.content, message.senderId, canAccept)
}

private fun handleAcceptance(message: AgentMessageBroker.Message) {
    // Task assignment confirmed
}

private fun handleRejection(message: AgentMessageBroker.Message) {
    // Find alternate agent to handle task
}

private fun evaluateCapacity(): Boolean {
    // Check if agent has capacity for new task
    return true
```

```
}
```

}

Deliverable:

- [ ] Multi-agent architecture diagram with agent types
- [ ] Android multi-agent system implementation (3+ agents)
- [ ] Agent perception-decision-action cycle explanation
- [ ] Conflict resolution strategy for multiple agents
- [ ] Agent communication protocol and messaging system
- [ ] Real-world multi-agent scenario walkthrough

Task 8: Intelligent Agents and Adaptive Behavior (Self-Study: 1.5–2 hours)

Objective: Master intelligent agent design with learning and adaptation.

Instructions:

1. Learning Agents

// Agent that learns from user feedback
class LearningAgent(val agentId: String) {
private val preferences = mutableMapOf<String, Float>()
private val feedbackHistory = mutableListOf<Feedback>()

```
data class Feedback(val action: String, val reward: Float, val
timestamp: Long)

fun recordFeedback(action: String, reward: Float) {
    feedbackHistory.add(Feedback(action, reward,
System.currentTimeMillis()))

    // Update preference for this action
    val currentPref = preferences.getOrDefault(action, 0.5f)
    val alpha = 0.1f  // Learning rate
    val newPref = currentPref + alpha * (reward - currentPref)
    preferences[action] = newPref
}

fun selectAction(availableActions: List<String>): String {
    // Epsilon-greedy exploration
    val epsilon = 0.1f  // 10% chance to explore

    return if (Math.random() < epsilon) {
```

```
        // Explore: random action
        availableActions.random()
    } else {
        // Exploit: best known action
        availableActions.maxByOrNull { preferences.getOrDefault(it,
0.5f) }
            ?: availableActions.first()
    }
}

fun getPredictedReward(action: String): Float {
    return preferences.getOrDefault(action, 0.5f)
}
```

```
}

// Usage
val agent = LearningAgent("notification_agent")

// User sees notification and opens it (positive reward)
agent.recordFeedback("notification_at_2pm", reward = 1.0f)

// User sees notification but ignores it (negative reward)
agent.recordFeedback("notification_at_6pm", reward = 0.1f)

// Agent learns: notifications at 2pm are better
val bestTime = agent.selectAction(listOf(
"notification_at_2pm",
"notification_at_6pm",
"notification_at_8pm"
))
```

2. Context-Aware Agents

```
// iOS context-aware agent
class ContextAwareAgent {
let locationManager = CLLocationManager()
let motionManager = CMMotionManager()

enum UserContext {
    case atHome
    case atWork
    case commuting
    case exercising
    case unknown
}

func detectContext() -> UserContext {
    let location = getCurrentLocation()
    let motion = getCurrentMotion()
    let time = Date().hour
```

```
    // Multi-factor context detection
    return when {
        isAtWork(location, time) -> .atWork
        isAtHome(location) -> .atHome
        motion.speed > 5 -> .commuting
        motion.isExerciseActivity -> .exercising
        else -> .unknown
    }
}

func adaptToContext(context: UserContext) {
    switch context {
    case .atHome:
        showPersonalRecommendations()
        enablePrivacyMode()

    case .atWork:
        hideNonWorkContent()
        reduceBatteryUsage()

    case .commuting:
        useAudioNotifications()
        pauseVideoContent()

    case .exercising:
        showFitnessTracker()
        disableScreenTimeout()

    case .unknown:
        useSafeDefaults()
    }
}
```

}

3. Goal-Oriented Agents

// Agent with explicit goals
class GoalOrientedAgent(val agentId: String) {
sealed class Goal {
data class MaximizeEngagement(val targetScore: Float = 0.8f) : Goal()
data class MinimizeNotifications(val maxPerDay: Int = 5) : Goal()
data class OptimizeRetention(val targetRetention: Float = 0.7f) : Goal()
}

```
private var currentGoal: Goal = Goal.MaximizeEngagement()
private var goalProgress = 0f

fun setPrimaryGoal(goal: Goal) {
    currentGoal = goal
    goalProgress = 0f
```

```
}

fun updateProgress(metric: Float) {
    goalProgress = metric

    // Check if goal is satisfied
    when (currentGoal) {
        is Goal.MaximizeEngagement -> {
            if (metric >= (currentGoal as
Goal.MaximizeEngagement).targetScore) {
                onGoalAchieved()
            }
        }
        is Goal.MinimizeNotifications -> {
            if (metric <= (currentGoal as
Goal.MinimizeNotifications).maxPerDay) {
                onGoalAchieved()
            }
        }
        is Goal.OptimizeRetention -> {
            if (metric >= (currentGoal as
Goal.OptimizeRetention).targetRetention) {
                onGoalAchieved()
            }
        }
    }
}

fun selectActionForGoal(availableActions: List<Action>): Action {
    return when (currentGoal) {
        is Goal.MaximizeEngagement -> {
            // Prioritize actions that maximize engagement
            availableActions.maxByOrNull { getEngagementScore(it) }
                ?: availableActions.first()
        }
        is Goal.MinimizeNotifications -> {
            // Filter out notification actions
            availableActions.filterNot { it is NotificationAction }
                .firstOrNull() ?: availableActions.first()
        }
        is Goal.OptimizeRetention -> {
            // Select actions that improve retention
            availableActions.maxByOrNull { getRetentionScore(it) }
                ?: availableActions.first()
        }
    }
}

private fun onGoalAchieved() {
    // Trigger new goal or switch strategy
```

```
}

private fun getEngagementScore(action: Action): Float = 0f
private fun getRetentionScore(action: Action): Float = 0f
```

}

Deliverable:

- [ ] Learning agent implementation with reward feedback
- [ ] Epsilon-greedy exploration-exploitation strategy
- [ ] Context detection and adaptation framework
- [ ] Goal-oriented agent with multiple goals
- [ ] Performance metrics for agent learning
- [ ] Real-world scenario: agent that learns notification timing

---

Part E: Evaluation and Deployment

Task 9: Model Evaluation and Performance Metrics (Self-Study: 1.5–2 hours)

Objective: Master ML model evaluation and mobile performance optimization.

Instructions:

1. ML Evaluation Metrics

Classification Metrics:

Confusion Matrix:
Predicted Positive Predicted Negative
Actual Positive True Positive False Negative
(TP) (FN)
Actual Negative False Positive True Negative
(FP) (TN)

Accuracy = (TP + TN) / (TP + FN + FP + TN)
Precision = TP / (TP + FP) — "How many predictions were correct?"
Recall = TP / (TP + FN) — "How many actual positives were found?"
F1-Score = 2 * (Precision * Recall) / (Precision + Recall)

Recommendation Metrics:

Precision@K = (# relevant items in top K) / K
Recall@K = (# relevant items in top K) / (# total relevant items)

Mean Reciprocal Rank (MRR) = average position of first relevant item
Normalized Discounted Cumulative Gain (NDCG) = quality weighted by position

Mobile-Specific Metrics:

data class MobileMetrics(
// Accuracy metrics
val accuracy: Float,
val precision: Float,
val recall: Float,

```
// Performance metrics
val latency: Long,          // milliseconds
val memory: Long,           // megabytes
val batteryDrain: Float,    // milliwatts
val modelSize: Long,        // kilobytes

// Quality metrics
val fpRate: Float,          // false positive rate
val fnRate: Float,          // false negative rate
val confidenceThreshold: Float
```

)

2. Android Model Evaluation

class ModelEvaluator {

```
fun evaluateModel(
    model: TensorFlowLiteModel,
    testDataset: List<Pair<FloatArray, String>>
): MobileMetrics {
    val predictions = mutableListOf<String>()
    val confidences = mutableListOf<Float>()
    val latencies = mutableListOf<Long>()
    val memoryUsage = mutableListOf<Long>()

    // Run inference on test set
    for ((features, _) in testDataset) {
        val startTime = SystemClock.elapsedRealtimeNanos()
        val startMemory = Runtime.getRuntime().totalMemory()

        val result = model.predict(features)
        predictions.add(result.label)
        confidences.add(result.confidence)

        latencies.add((SystemClock.elapsedRealtimeNanos() - startTime)
/ 1_000_000)
        memoryUsage.add(Runtime.getRuntime().totalMemory() -
startMemory)
    }
```

```
    // Calculate metrics
    val accuracy = calculateAccuracy(predictions, testDataset.map {
it.second })
    val precision = calculatePrecision(predictions, testDataset.map {
it.second })
    val recall = calculateRecall(predictions, testDataset.map {
it.second })

    val avgLatency = latencies.average().toLong()
    val avgMemory = memoryUsage.average().toLong()
    val avgConfidence = confidences.average()

    return MobileMetrics(
        accuracy = accuracy,
        precision = precision,
        recall = recall,
        latency = avgLatency,
        memory = avgMemory,
        batteryDrain = estimateBatteryDrain(avgLatency),
        modelSize = model.sizeInBytes,
        fpRate = calculateFalsePositiveRate(predictions,
testDataset.map { it.second }),
        fnRate = calculateFalseNegativeRate(predictions,
testDataset.map { it.second }),
        confidenceThreshold = avgConfidence
    )
}

private fun calculateAccuracy(predicted: List<String>, actual:
List<String>): Float {
    return predicted.zip(actual).count { it.first == it.second
}.toFloat() / predicted.size
}

private fun calculatePrecision(predicted: List<String>, actual:
List<String>): Float {
    val tp = predicted.zip(actual).count { it.first == it.second &&
it.first == "positive" }
    val fp = predicted.count { it == "positive" } - tp
    return if (tp + fp > 0) tp.toFloat() / (tp + fp) else 0f
}

private fun calculateRecall(predicted: List<String>, actual:
List<String>): Float {
    val tp = predicted.zip(actual).count { it.first == it.second &&
it.first == "positive" }
    val fn = actual.count { it == "positive" } - tp
    return if (tp + fn > 0) tp.toFloat() / (tp + fn) else 0f
}
```

```
private fun estimateBatteryDrain(latencyMs: Long): Float {
    // CPU power consumption estimate: ~100mW per 100ms inference
    return (latencyMs / 100).toFloat() * 100
}
```

}

3. Cross-Platform Benchmarking

// Benchmark models across devices
class CrossDeviceBenchmark {

```
data class DeviceResult(
    val deviceModel: String,
    val osVersion: String,
    val latency: Long,
    val memory: Long,
    val accuracy: Float
)

fun benchmarkOnDevice(model: TensorFlowLiteModel): DeviceResult {
    val device = Build.MODEL
    val osVersion = Build.VERSION.RELEASE

    val startTime = SystemClock.elapsedRealtimeNanos()
    val startMemory = Runtime.getRuntime().totalMemory()

    // Run multiple inference iterations
    repeat(100) {
        model.predict(generateRandomFeatures())
    }

    val totalLatency = SystemClock.elapsedRealtimeNanos() - startTime
    val totalMemory = Runtime.getRuntime().totalMemory() - startMemory

    return DeviceResult(
        deviceModel = device,
        osVersion = osVersion,
        latency = totalLatency / 100_000_000,  // Average per inference
        memory = totalMemory,
        accuracy = 0.95f  // From test set
    )
}
```

}

Deliverable:

- [ ] Classification metrics explanation (accuracy, precision, recall, F1)

- [ ] Recommendation metrics (Precision@K, Recall@K, NDCG)
- [ ] Mobile-specific metrics framework
- [ ] Model evaluation implementation (Android or iOS)
- [ ] Cross-device benchmarking framework
- [ ] Performance targets and optimization strategies

Task 10: Deployment and Continuous Improvement (Self-Study: 1.5–2 hours)

Objective: Master model deployment strategies and continuous improvement.

Instructions:

1. A/B Testing ML Models

class ModelABTest {
enum class ModelVariant { CONTROL, TREATMENT_V1, TREATMENT_V2 }

```
fun assignUserToVariant(userId: String): ModelVariant {
    // Consistent hashing: same user always gets same variant
    val hash = userId.hashCode().toLong() and 0xFFFFFFFF
    return when (hash % 3) {
        0L -> ModelVariant.CONTROL
        1L -> ModelVariant.TREATMENT_V1
        else -> ModelVariant.TREATMENT_V2
    }
}

fun recordMetric(
    userId: String,
    metric: String,
    value: Float,
    timestamp: Long = System.currentTimeMillis()
) {
    val variant = assignUserToVariant(userId)

    // Log: userId, variant, metric, value, timestamp
    logToAnalytics(
        variant = variant.name,
        metric = metric,
        value = value,
        timestamp = timestamp
    )
}

fun analyzeResults(): TestResults {
```

```
    // After running for 2 weeks
    val controlResults = fetchMetrics(ModelVariant.CONTROL)
    val treatment1Results = fetchMetrics(ModelVariant.TREATMENT_V1)
    val treatment2Results = fetchMetrics(ModelVariant.TREATMENT_V2)

    return TestResults(
        controlMetrics = controlResults,
        treatmentMetrics = listOf(treatment1Results,
treatment2Results),
        statisticalSignificance = calculateSignificance(
            controlResults,
            treatment1Results
        )
    )
}
```

}

2. Model Update Strategy

// Safe model update with rollback
class SafeModelUpdate {

```
fun updateModel(newModelPath: String, verificationDataset:
List<Pair<FloatArray, String>>) {
    // Step 1: Verify new model
    val newModel = loadModel(newModelPath)
    val newMetrics = evaluateModel(newModel, verificationDataset)

    val currentModel = loadCurrentModel()
    val currentMetrics = evaluateModel(currentModel,
verificationDataset)

    // Step 2: Check if new model is better
    if (newMetrics.accuracy > currentMetrics.accuracy + 0.01f &&  // At
least 1% improvement
        newMetrics.latency < currentMetrics.latency * 1.2f) {    // Not
slower than 20%

        // Step 3: Canary deployment (5% of users)
        deployCanary(newModel, fractionOfUsers = 0.05f)

        // Step 4: Monitor metrics for 24 hours
        Thread.sleep(24 * 60 * 60 * 1000)

        val canaryMetrics = fetchCanaryMetrics()

        if (canaryMetrics.crashRate < 0.01f &&  // <1% crash rate
            canaryMetrics.engagement > currentMetrics.engagement *
0.95f) {  // Not worse than 5%
```

```
            // Step 5: Full rollout
            deployGlobally(newModel)
            archiveOldModel(currentModel)
        } else {
            // Rollback
            deployModel(currentModel)
            logWarning("Canary failed, rolled back to previous model")
        }
    } else {
        logWarning("New model not better than current, skipping
update")
    }
}
```

}

3. Monitoring and Alerting

class ModelMonitor {
data class HealthCheck(
val modelAccuracy: Float,
val inferenceLatency: Long,
val crashRate: Float,
val userEngagement: Float,
val timestamp: Long
)

```
fun monitorModel(
    model: TensorFlowLiteModel,
    sampleSize: Int = 1000
): HealthCheck {
    val sampleData = fetchSampleData(sampleSize)

    val accuracy = evaluateAccuracy(model, sampleData)
    val latency = measureLatency(model, sampleData)
    val crashes = measureCrashRate()
    val engagement = measureEngagement()

    return HealthCheck(
        modelAccuracy = accuracy,
        inferenceLatency = latency,
        crashRate = crashes,
        userEngagement = engagement,
        timestamp = System.currentTimeMillis()
    )
}

fun checkHealth(health: HealthCheck) {
    // Alert if metrics degrade
    val thresholds = mapOf(
```

```
        "accuracy" to 0.85f,
        "latency" to 100L,
        "crashRate" to 0.01f,
        "engagement" to 0.70f
    )

    if (health.modelAccuracy < thresholds["accuracy"]!!) {
        alertMetricsDegraded("Model accuracy dropped to
${health.modelAccuracy}")
    }

    if (health.inferenceLatency > thresholds["latency"]!!) {
        alertMetricsDegraded("Inference latency increased to
${health.inferenceLatency}ms")
    }

    if (health.crashRate > thresholds["crashRate"]!!) {
        alertCritical("Model causing crashes: ${health.crashRate *
100}%")
    }
}
```

}

Deliverable:

- [ ] A/B testing framework for ML models
- [ ] Variant assignment strategy (consistent hashing)
- [ ] Safe model update procedure with verification
- [ ] Canary deployment strategy
- [ ] Rollback mechanism
- [ ] Model monitoring and health check implementation
- [ ] Alert thresholds and escalation procedures

Evaluation Criteria

Criteria	Excellent (9–10)	Good (7–8)	Acceptable (5–6)	Needs Improvement
Algorithm Knowledge	All ML algorithms mastered	6+ algorithms understood	3-4 algorithms understood	Single algorithm
Mobile Constraints	Comprehensive analysis	Good understanding	Basic awareness	Minimal consideration

Neural Networks	TFLite & Core ML mastered	1 platform solid	Basic implementation	Minimal implementation
Recommendation Engine	Complete hybrid system	Content + collaborative	Single approach	Basic recommendation
Multi-Agent Systems	4+ coordinated agents	2-3 agents working	Single agent	Minimal agent concept
Code Quality	Clean, optimized implementations	Generally good	Readable, lacks optimization	Difficult to follow
Performance Evaluation	Comprehensive metrics	Multiple metrics analyzed	Basic metrics	Minimal evaluation
Deployment Strategy	A/B testing, monitoring, rollback	Testing and monitoring	Basic deployment	Manual deployment
Communication	Clear, detailed explanations	Generally clear	Readable, lacks depth	Disorganized
Practical Application	Real-world applicable solutions	Mostly practical	Basic applicability	Theoretical only

---

Deliverable Checklist

Submit as DOCX (7–9 pages):

- [ ] Task 1: Algorithm framework + mobile constraints + feature engineering
- [ ] Task 2: Feature extraction pipeline + normalization + data quality
- [ ] Task 3: TensorFlow Lite + Core ML integration + GPU acceleration
- [ ] Task 4: Pre-trained models + transfer learning examples
- [ ] Task 5: Recommendation engine implementation (hybrid approach)
- [ ] Task 6: Personalization strategies + adaptive UI + timing optimization
- [ ] Task 7: Multi-agent system with 3+ agents + communication protocol
- [ ] Task 8: Learning agents + context-aware agents + goal-oriented agents
- [ ] Task 9: Model evaluation metrics + benchmarking framework
- [ ] Task 10: A/B testing + safe deployment + monitoring

Resources & References

[1] Google. (2024). TensorFlow Lite Documentation. https://www.tensorflow.org/lite

[2] Apple. (2024). Core ML Documentation. https://developer.apple.com/coreml/

[3] TensorFlow. (2024). Machine Learning Crash Course. https://developers.google.com/machine-learning/crash-course

[4] Google. (2024). Android ML Kit. https://developers.google.com/ml-kit

[5] Apple. (2024). Create ML. https://developer.apple.com/create-ml/

[6] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

[7] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

[8] Ng, A., & Jordan, M. (2002). On discriminative vs. generative classifiers. *Advances in Neural Information Processing Systems*, 14.

[9] Wooldridge, M. (2002). *Introduction to Multiagent Systems*. Wiley.

[10] Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

[11] Kaggle. (2024). Machine Learning Competitions. https://www.kaggle.com/

[12] Papers with Code. (2024). Machine Learning Research. https://paperswithcode.com/

Timeline Guidance

Self-Study (9–11 hours):

- Hours 1–3: Task 1–2 (algorithms, feature engineering)
- Hours 3–5.5: Task 3–4 (neural networks, transfer learning)
- Hours 5.5–7.5: Task 5–6 (recommendations, personalization)
- Hours 7.5–9: Task 7–8 (multi-agent systems, intelligent agents)
- Hours 9–11: Task 9–10 (evaluation, deployment)

Online Sessions (2–2.5 hours):

- 0.5 hours: ML fundamentals and algorithm selection
- 0.5 hours: Neural networks and transfer learning demonstration
- 0.5 hours: Recommendation engines and personalization walkthrough
- 0.5 hours: Multi-agent systems and deployment strategies

Extension Tasks (Optional)

1. Implement federated learning for privacy-preserving model training
2. Build custom neural network architecture for specific domain
3. Create reinforcement learning agent for game/simulation
4. Implement model compression techniques (pruning, quantization)
5. Develop on-device active learning system
6. Build real-time anomaly detection system
7. Implement multi-modal ML system (text + images)
8. Create production ML pipeline with monitoring dashboard