Machine Learning for Search: Complete Study Guide

Faire ML Interview Preparation

Alex Yang
Principal Software Engineer, Roblox

Prepared: November 24, 2025

Contents

1 Introduction & Study Approach

1.1 What This Guide Covers

This is a **study guide**, not a cheatsheet. It's designed to help you deeply understand ML concepts for search, recommendations, and ads systems from first principles. You should:

- **Read actively:** Take notes, work through examples
- **Code along:** Implement the algorithms yourself
- **Practice explaining:** Teach concepts out loud
- **Connect to experience:** Relate to your past projects

1.2 Interview Format at Faire (Expected)

Based on typical ML for Search interviews, expect:

1. **ML System Design** (45-60 min)
   - "Design a search ranking system"
   - "Build a recommendation engine for products"
   - "Design an ads click prediction system"
2. **ML Fundamentals** (30-45 min)
   - Theory questions (bias-variance, regularization, etc.)
   - Metric selection and evaluation
   - Model selection trade-offs
3. **ML Coding** (30-60 min, possibly)
   - Implement algorithm from scratch
   - Calculate metric (AUC, NDCG)
   - Feature engineering problem

1.3 Recommended Study Schedule

Day	Hours	Topics
1	4-5	ML Fundamentals, Metrics, Learning to Rank
2	4-5	Model Architectures, Feature Engineering
3	4-5	System Design Framework, Practice Problems
4	3-4	ML Coding, Implementation Practice
5	2-3	Review, Mock Interview Practice

2 ML Fundamentals: Core Concepts

2.1 The Bias-Variance Tradeoff

2.1.1 What Is It?

The bias-variance tradeoff is the fundamental tension in machine learning between two types of errors:

Bias = Error from wrong assumptions
Variance = Error from sensitivity to training data

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

2.1.2 Understanding Through Examples

Example

High Bias (Underfitting):
Imagine fitting a linear model to data that's actually quadratic:

- Model: $y = ax + b$
- True relationship: $y = x^2$
- Problem: Model can't capture curvature (wrong assumption)
- Result: Poor performance on both training and test data

High Variance (Overfitting):
Imagine fitting a 10th degree polynomial to data with only 5 points:

- Model: $y = a_{10}x^{10} + ... + a_1x + b$
- Problem: Model memorizes noise in training data
- Result: Perfect on training data, terrible on test data

2.1.3 How to Identify in Practice

Symptom	High Bias	High Variance
Train Error	High (e.g., 20%)	Low (e.g., 1%)
Test Error	High (e.g., 22%)	High (e.g., 25%)
Gap	Small	Large

2.1.4 Solutions

For High Bias (Underfitting):

- Add more features or feature interactions
- Increase model complexity (more layers, higher degree)

- Reduce regularization
- Try different model family (linear → tree-based → neural net)

For High Variance (Overfitting):

- Get more training data
- Add regularization (L1, L2, dropout)
- Reduce model complexity
- Early stopping
- Ensemble methods (reduce variance)

Key Insight

Interview Tip: When asked about model performance issues, ALWAYS start by diagnosing bias vs variance:

1. Look at train error vs test error
2. Identify which problem you have
3. Propose targeted solutions
4. Explain why each solution helps

2.2 Regularization: L1 vs L2

2.2.1 What Is Regularization?

Regularization adds a penalty term to the loss function to discourage overly complex models:

$$\text{Loss}_{\text{total}} = \text{Loss}_{\text{data}} + \lambda \times \text{Penalty}$$

Where λ controls the strength of regularization.

2.2.2 L1 Regularization (Lasso)

$$\text{Penalty}_{\text{L1}} = \sum_{i=1}^{n} |w_i|$$

Key Properties:

- Encourages **sparse solutions** (many weights = 0)
- Acts as feature selection
- Non-differentiable at zero (use subgradient methods)

When to Use:

- Many irrelevant or redundant features
- Need interpretability (identify important features)
- Want automatic feature selection

2.2.3 L2 Regularization (Ridge)

$$\text{Penalty}_{\text{L2}} = \sum_{i=1}^{n} w_i^2$$

Key Properties:

- Shrinks weights smoothly (no sparsity)
- Handles multicollinearity well
- Differentiable everywhere

When to Use:

- All features potentially useful
- Correlated features (multicollinearity)
- Want smooth weight distributions

2.2.4 Visual Comparison

Why L1 creates sparsity:

- L1 constraint region is diamond-shaped (sharp corners at axes)
- Solution likely touches a corner (one weight = 0)
- L2 constraint region is circular (no sharp corners)
- Solution smoothly shrinks all weights

Example

Concrete Example:
Dataset: 100 features, only 10 are actually useful

- **L1:** Might select 12-15 features (sparse), setting 85-88 weights to 0
- **L2:** Shrinks all 100 weights, but none exactly to 0
- **Elastic Net (L1 + L2):** Best of both - sparse selection + smooth shrinkage

2.3 Gradient Descent Variants

2.3.1 The Core Algorithm

$$w_{t+1} = w_t - \eta \nabla L(w_t)$$

Where:

- w_t = current weights
- η = learning rate
- ∇L = gradient of loss function

Type	Batch Size	Pros	Cons
Batch GD	All data	Stable, converges to optimum	Slow, memory intensive
Stochastic GD	1 sample	Fast, can escape local minima	Noisy, unstable
Mini-batch GD	32-512	**Best tradeoff**	Need to tune batch size

2.3.2 Three Variants

Industry Standard: Mini-batch with batch size 32-256 (depends on problem)

2.3.3 Advanced Optimizers

1. SGD with Momentum

- Adds "velocity" term: accumulates gradients over time
- Reduces oscillations, accelerates in consistent directions
- Hyperparameter: momentum β (typically 0.9)

2. Adam (Adaptive Moment Estimation) - Most Popular

- Combines momentum (first moment) + RMSprop (second moment)
- Adaptive learning rates per parameter
- Robust to learning rate choice
- Default for deep learning

3. RMSprop

- Adapts learning rate based on recent gradient magnitudes
- Good for non-stationary objectives (online learning)

Key Insight

Interview Question: "Which optimizer should we use?"
Answer Framework:

- **Default choice:** Adam (works well in 90% of cases)
- **When to use SGD + Momentum:**
  - Simple models (logistic regression, linear)
  - Want better generalization (Adam can overfit)
  - Have time to tune learning rate
- **When to use RMSprop:** Non-stationary problems, online learning

3 Evaluation Metrics: Deep Dive

3.1 Classification Metrics

3.1.1 Confusion Matrix Foundation

Everything starts from the confusion matrix:

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

3.1.2 Precision vs Recall

$$\text{Precision} = \frac{TP}{TP + FP} \quad \text{Recall} = \frac{TP}{TP + FN}$$

Example

Spam Detection Scenario:
100 emails: 10 are actual spam, 90 are legitimate
Model predicts: 15 emails as spam

- 8 correct spam detections (TP)
- 7 false alarms (FP) - legitimate emails marked as spam
- 2 missed spam emails (FN)

Metrics:

- Precision = $8/(8+7) = 53\%$ - "Of emails I marked spam, how many were actually spam?"
- Recall = $8/(8+2) = 80\%$ - "Of all actual spam, how many did I catch?"

Trade-off:

- Make threshold stricter → Higher precision, lower recall (fewer false alarms, but miss more spam)
- Make threshold looser → Higher recall, lower precision (catch more spam, but more false alarms)

3.1.3 When to Optimize for What

Optimize Precision when FP is costly:

- Spam detection (don't block real emails)
- Medical diagnosis (don't scare healthy patients)
- Fraud detection with manual review (don't waste investigator time)

Optimize Recall when FN is costly:

- Disease screening (don't miss sick patients)
- Airport security (don't miss threats)
- Fraud detection without review (catch all fraud)

Balance both with F1:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.1.4 AUC-ROC: The Gold Standard

What it measures: Probability that a random positive example is ranked higher than a random negative example.

Range: [0.5, 1.0]

- 0.5 = Random guessing
- 0.7-0.8 = Good model
- 0.8-0.9 = Excellent model
- 0.9+ = Outstanding (or data leakage!)

Why use it:

- Threshold-independent (unlike precision/recall)
- Works well with imbalanced data
- Measures ranking quality, not just binary classification
- Industry standard for CTR prediction

! Critical

When NOT to use AUC-ROC:

- **Highly imbalanced data (CTR ¡ 1%):** Use PR-AUC instead
- **When you care about a specific threshold:** Use precision/recall at that threshold
- **When calibrated probabilities matter:** Use log loss instead

3.1.5 Log Loss (Binary Cross-Entropy)

$$\text{LogLoss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

What it measures: Penalizes confident wrong predictions

When to use:

- Need calibrated probabilities (not just rankings)

- Used as training objective (not just evaluation)
- Example: Ad auctions (need actual probability for bid calculation)

Interpretation:

- Lower is better (0 = perfect)
- Heavily penalizes confident mistakes (predicting 0.99 when truth is 0)

3.2 Ranking Metrics

3.2.1 NDCG@K: The Industry Standard

Normalized Discounted Cumulative Gain at position K

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{2^{\text{rel}_i} - 1}{\log_2(i+1)}$$

$$\text{NDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}}$$

Where IDCG@K is the ideal (best possible) DCG@K.

3.2.2 Why NDCG?

Advantages:

- **Graded relevance:** Can handle 0, 1, 2, 3 (not just binary)
- **Position-aware:** Top results matter more (logarithmic discount)
- **Normalized:** [0, 1] range, comparable across queries
- **Industry standard:** Used by Google, Amazon, LinkedIn, etc.

Example

Concrete Example:
Search query: "bluetooth headphones"
Top-5 results with relevance scores (0=irrelevant, 3=highly relevant):
DCG@5 = $7.0 + 1.89 + 0.5 + 0 + 1.16 = 10.55$
If ideal ordering was [3, 2, 2, 1, 0]: IDCG@5 = 11.7
NDCG@5 = 10.55/11.7 = 0.90 (pretty good!)

3.2.3 MRR: Mean Reciprocal Rank

$$\text{MRR} = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{\text{rank}_q}$$

When to use:

- **Navigational queries:** User looking for ONE specific result

- Example: "Facebook login", "Amazon", brand searches
- Only cares about position of FIRST relevant result

Example

Navigational Query Example:
Query: "Starbucks near me"
User wants: Nearest Starbucks location
If nearest location appears at position 3:

- MRR = 1/3 = 0.33
- User likely has to scroll past 2 irrelevant results
- Poor experience, even though result is present

For this query type, MRR is more meaningful than NDCG.

3.3 Online Metrics (A/B Testing)

3.3.1 Engagement Metrics

Click-Through Rate (CTR):

$$\text{CTR} = \frac{\text{Number of Clicks}}{\text{Number of Impressions}}$$

When it matters:

- Early funnel metric (interest)
- Correlation with relevance
- Can be gamed (clickbait)

Conversion Rate:

$$\text{CVR} = \frac{\text{Number of Conversions}}{\text{Number of Clicks}}$$

When it matters:

- Revenue generation
- True value metric
- Harder to game

3.3.2 Business Metrics

GMV (Gross Merchandise Value):

- Total dollar value of transactions
- Ultimate business metric for e-commerce

- May lag other metrics (need time to convert)

Revenue Per Search (RPS):

$$\text{RPS} = \frac{\text{Total Revenue}}{\text{Number of Searches}}$$

Key Insight

Metric Selection Framework for Interviews:
When asked "What metrics would you use?", structure answer as:

1. **Offline metrics:** For development (NDCG@10, AUC)
2. **Online guardrail metrics:** Must not degrade (latency, zero-result rate)
3. **Online engagement metrics:** Want to improve (CTR, add-to-cart)
4. **Business metrics:** Ultimate goal (GMV, revenue, retention)

Explain trade-offs between metrics and why you'd prioritize certain ones.

4 Learning to Rank (LTR): Complete Guide

4.1 Problem Formulation

Goal: Given query q and documents $D = \{d_1, d_2, ..., d_n\}$, produce optimal ranking π.

Input:

- Query features: query length, query type, user location
- Document features: title, category, price, rating
- Query-document features: BM25 score, embedding similarity

Output:

- Ranking: Ordered list of documents
- Scores: Relevance scores for each document

4.2 Pointwise Approach

4.2.1 Core Idea

Treat ranking as independent classification/regression for each document.

$$\text{score}(q, d) = f(\text{features}(q, d))$$

Predict relevance score for each document independently, then sort by score.

4.2.2 Methods

- **Regression:** Predict continuous relevance score
- **Classification:** Predict relevance category (0, 1, 2, 3)
- **Models:** Linear regression, logistic regression, neural networks

4.2.3 Pros and Cons

Pros:

- Simple to implement and understand
- Fast training (standard classification/regression)
- Works with small data
- Easy to interpret (feature weights)

Cons:

- Ignores relative order between documents
- Not optimized for ranking metrics (NDCG, MRR)
- Absolute scores may be miscalibrated

- Treats each document independently (no comparisons)

When to use:

- Cold start (limited training data)
- Simple baseline
- When absolute scores matter (not just order)

4.3 Pairwise Approach

4.3.1 Core Idea

Learn to compare pairs of documents: which should rank higher?

$$P(d_i \succ d_j|q) = \sigma(f(q, d_i) - f(q, d_j))$$

Train model to predict pairwise preferences.

4.3.2 Key Algorithm: RankNet

Neural network approach:

1. For each pair (d_i, d_j) where d_i should rank higher
2. Compute scores: $s_i = f(q, d_i)$, $s_j = f(q, d_j)$
3. Probability d_i ranks higher: $P_{ij} = \sigma(s_i - s_j)$
4. Loss: Binary cross-entropy on pair preferences

Gradient flows through pairs to optimize ranking order.

4.3.3 Extension: LambdaRank

Key insight: Not all pairs are equally important!
Pairs that would change NDCG more should have higher loss weight.

$$\lambda_{ij} = \frac{\partial \text{Loss}}{\partial s_i} \times |\Delta \text{NDCG}_{ij}|$$

Weight each pair by how much swapping would change NDCG.

4.3.4 Pros and Cons

Pros:

- Directly optimizes ranking order
- Better than pointwise for most use cases
- More training signal (quadratic pairs)
- Models relative preferences

Cons:

- More complex than pointwise
- Slower training (all pairs = $O(n^2)$)
- Still not directly optimizing NDCG
- May need pair sampling for efficiency

When to use:

- Moderate-sized datasets
- Need better ranking than pointwise
- Can't afford full listwise training

4.4 Listwise Approach

4.4.1 Core Idea

Optimize the entire list directly for ranking metrics.

Consider all documents together, not independently or pairwise.

4.4.2 LambdaMART: Production Standard

Combination of:

- **LambdaRank:** Gradient weighting by ΔNDCG
- **MART:** Multiple Additive Regression Trees (gradient boosting)

Implementation: XGBoost or LightGBM with `rank:ndcg` objective

```
import xgboost as xgb

# Configuration for ranking
params = {
    'objective': 'rank:ndcg',  # Listwise ranking
    'eval_metric': 'ndcg@10',  # Optimize NDCG@10
    'eta': 0.1,                # Learning rate
    'max_depth': 6,            # Tree depth
    'subsample': 0.8           # Row sampling
}

# Train
dtrain = xgb.DMatrix(X_train, label=y_train)
dtrain.set_group(group_sizes)  # Queries as groups

model = xgb.train(params, dtrain, num_boost_round=100)
```

4.4.3 Why LambdaMART Dominates

Advantages:

- Directly optimizes NDCG (or other ranking metrics)
- Best offline performance
- Handles non-linear feature interactions
- Robust to feature scales
- Feature importance built-in
- Industry battle-tested

Use cases:

- Production search ranking systems
- Large-scale recommendation systems
- Any problem with sufficient training data

Requirements:

- Sufficient data (thousands of queries)
- Graded relevance labels (0, 1, 2, 3)
- Grouping by query (queries as independent samples)

4.5 LTR Comparison Table

Approach	Unit	Pros	Cons	Use
Pointwise	Single doc	Simple, fast	Ignores order	Baseline
Pairwise	Doc pairs	Better ranking	Not direct NDCG	Medium data
Listwise	Full list	Best metrics	Complex, slow	Production

+ Action Item

Practice Exercise:
Explain to someone (or out loud) when you would use each LTR approach. Give concrete scenarios:

- Pointwise: Cold start with 100 labeled queries
- Pairwise: 1K queries, need better than baseline
- Listwise: 10K+ queries, production deployment

5 Model Selection for Search & Ads

5.1 Gradient Boosted Decision Trees (GBDT)

5.1.1 Why GBDT Dominates Ranking

Market Share: 80%+ of production ranking systems use XGBoost/LightGBM

Key Advantages:

1. **Handles mixed feature types** - numeric, categorical, sparse
2. **Captures non-linear interactions** automatically
3. **Robust to outliers** - tree splits are threshold-based
4. **Feature importance** built-in
5. **Fast training and inference**
6. **Less hyperparameter tuning** than neural networks
7. **Interpretable** - can visualize trees

5.1.2 How Gradient Boosting Works

Core Idea: Build trees sequentially, each correcting previous errors.

1. Start with initial prediction (mean or constant)
2. For each round $t = 1, 2, ..., T$:
   - Compute residuals (errors from previous prediction)
   - Fit tree to residuals (not original targets!)
   - Add tree prediction to ensemble
   - Update prediction: $F_t = F_{t-1} + \eta \cdot h_t$
3. Final prediction: sum of all trees

$$F(x) = \sum_{t=1}^{T} \eta \cdot h_t(x)$$

Where η is learning rate (shrinkage).

5.1.3 XGBoost Best Practices

Starting Configuration:

```
params = {
    'objective': 'rank:ndcg',     # For ranking
    'eta': 0.1,                   # Learning rate
    'max_depth': 6,               # Tree depth
    'min_child_weight': 1,        # Min samples per leaf
    'subsample': 0.8,             # Row sampling
```

```
    'colsample_bytree': 0.8,      # Column sampling
    'lambda': 1.0,                # L2 regularization
    'alpha': 0.0                  # L1 regularization
}
```

Tuning Process:

1. Fix `n_estimators=100`, tune `max_depth` and `min_child_weight`
2. Tune `subsample` and `colsample_bytree` (regularization)
3. Tune `lambda` (L2) if overfitting
4. Finally, tune `eta` and scale up `n_estimators`

Early Stopping:

```
# Stop if validation metric doesn't improve for 20 rounds
model = xgb.train(
    params, dtrain,
    num_boost_round=1000,
    evals=[(dval, 'validation')],
    early_stopping_rounds=20
)
```

5.2 Deep Neural Networks for Ranking

5.2.1 When to Use DNNs

Advantages over GBDT:

- Better with high-cardinality categorical features (embeddings)
- Can leverage pre-trained representations (BERT, Word2Vec)
- End-to-end learning (features + model)
- Can share representations across tasks (multi-task learning)

Disadvantages:

- Need more data (10x-100x more)
- Harder to tune (many hyperparameters)
- Slower training
- Less interpretable
- Sensitive to feature scales

When to choose DNN:

- Very large datasets (millions of examples)
- Text/image features (need embeddings)
- Multi-task learning (CTR + CVR jointly)
- End-to-end learning important

5.2.2 Architecture: Wide & Deep

Google's production model for recommendations

```
Wide Component (Linear):
- Memorization of specific user-item interactions
- Cross-product features (user_id x item_id)
- Captures historical patterns

Deep Component (DNN):
- Generalization to unseen combinations
- Embedding layers for categorical features
- Dense layers to learn representations

Combined:
Output = sigmoid(Wide_output + Deep_output)
```

Why it works:

- Wide: Memorizes frequent patterns (precision)
- Deep: Generalizes to new combinations (recall)
- Together: Best of both worlds

5.2.3 Architecture: DeepFM

For CTR prediction (ads, recommendations)

```
FM Component:
- Factorization Machine
- Captures 2nd-order feature interactions
- All feature pairs

Deep Component:
- Same embeddings as FM
- MLP to learn higher-order interactions

Advantage: Shared embeddings reduce parameters
```

5.3 Two-Tower Models

5.3.1 Use Case: Candidate Generation

Problem: Retrieve top-K items from millions (too slow for deep ranking)
Solution: Pre-compute item embeddings, fast retrieval via ANN
Architecture:

```
Query Tower:              Item Tower:
User features -->         Item features -->
Dense(128) -->            Dense(128) -->
```

```
Dense(128) -->              Dense(128) -->
Query Embedding (64-dim)    Item Embedding (64-dim)

Similarity: dot(query_emb, item_emb)
```

Training:

- Positive pairs: (user, clicked_item)
- Negative sampling: (user, random_items)
- Loss: Contrastive loss or softmax

Serving:

1. Pre-compute all item embeddings offline
2. Index in vector DB (Faiss, ScaNN)
3. At query time: compute query embedding (1ms)
4. ANN search for top-1000 (5-10ms)
5. Pass to ranking model for top-100

Used by: YouTube, Pinterest, Spotify, Netflix

6 Feature Engineering for Search

6.1 Feature Categories

6.1.1 1. Query Features

Basic:

- Query length (characters, words, tokens)
- Query type (navigational, transactional, informational)
- Has numbers, special characters, etc.

Historical:

- Query frequency (head vs tail)
- Historical CTR for this query
- Average conversion rate
- Seasonality signals

6.1.2 2. Document Features

Content:

- Title, description text
- Category, brand
- Tags, attributes

Quality:

- User rating (average, count)
- Review count and sentiment
- Sales volume, popularity
- Age (days since published)

Business:

- Price, discount
- Inventory level
- Shipping speed
- Profit margin

6.1.3 3. Query-Document Features (Most Important!)

Text Matching:

- **BM25 score** (strongest signal for text relevance)
- TF-IDF similarity
- Exact match (title, brand, category)
- Edit distance (fuzzy matching)
- N-gram overlap (unigram, bigram)

Semantic:

- Embedding cosine similarity
- BERT score (if using neural ranking)

Behavioral:

- Historical CTR for (query, document) pair
- Conversion rate for pair
- Co-occurrence patterns

6.1.4 4. User Features

Demographics:

- Location (country, city, postal code)
- Language preference
- Device type (mobile, desktop)

Behavioral:

- Search history
- Purchase history
- Browse history
- User cohort (new, power user, dormant)

Session:

- Previous queries in session
- Clicks in session
- Time on page
- Scroll depth

6.1.5 5. Context Features

- Time of day, day of week
- Season, holidays
- Trending signals (surging interest)
- A/B test variant

6.2 Feature Engineering Techniques

6.2.1 Numerical Features

Normalization:

```
# Standard scaling
X_scaled = (X - mean) / std

# Min-max scaling
X_scaled = (X - min) / (max - min)
```

Log Transform: For skewed distributions

```
# For features like price, view count
X_log = np.log1p(X)  # log(1 + X), handles X=0
```

Binning: Convert continuous to categorical

```
# Price buckets
price_bucket = pd.cut(price,
    bins=[0, 10, 50, 100, np.inf],
    labels=['cheap', 'medium', 'expensive', 'premium'])
```

6.2.2 Categorical Features

One-Hot Encoding: For low cardinality

```
# For features with <50 unique values
category_encoded = pd.get_dummies(category)
```

Target Encoding: For high cardinality

```
# Replace category with mean target value
category_mean = df.groupby('category')['click'].mean()
df['category_encoded'] = df['category'].map(category_mean)
```

Embedding: For very high cardinality (DNN)

```
# Learn embedding for category_id
embedding_layer = Embedding(
    input_dim=num_categories,
    output_dim=16  # Embedding dimension
)
```

6.2.3 Feature Interactions

Explicit Cross Features:

```
# Create new feature from combinations
df['price_x_rating'] = df['price'] * df['rating']
df['query_in_title'] = (df['query'] in df['title'])
```

Let Model Learn: XGBoost, neural nets learn interactions automatically

6.3 Feature Selection

6.3.1 Why Feature Selection?

- Reduce overfitting (simpler model)
- Faster training and inference
- Improved interpretability
- Lower maintenance cost

6.3.2 Methods

1. Correlation-Based:

- Remove highly correlated features (redundant)
- Threshold: correlation > 0.95

2. Feature Importance (XGBoost):

```
# Get feature importance from trained model
importance = model.get_score(importance_type='gain')

# Keep top-K features
top_features = sorted(importance.items(),
    key=lambda x: x[1], reverse=True)[:100]
```

3. Ablation Study:

- Remove one feature at a time
- Measure impact on validation metric
- Drop if impact $<$ threshold

4. L1 Regularization:

- Use Lasso or L1-regularized model
- Automatically drives weak features to zero

Key Insight

Feature Engineering Interview Strategy:
When asked "What features would you use?", structure answer as:

1. **Start with must-haves:** Query-doc text match (BM25), category match
2. **Add quality signals:** Rating, sales, reviews
3. **Add behavioral:** Historical CTR (if available)
4. **Add user context:** Location, device, history
5. **Explain why each matters** for the specific problem
6. **Discuss feature engineering** (normalization, encoding)
7. **Mention feature selection** (importance, ablation)

7 ML System Design Framework

7.1 Overview: The 6-Step Framework

Use this framework for ANY "Design X system" question:

1. Problem Formulation (5 min)
2. Data Strategy (5-7 min)
3. Feature Engineering (7-10 min)
4. Model Selection (7-10 min)
5. Training & Evaluation (5-7 min)
6. Deployment & Monitoring (5-7 min)

Total time: 45 minutes, leaving 15 min for Q&A

7.2 Step 1: Problem Formulation

7.2.1 Questions to Ask

+ Action Item

Always clarify before designing:

1. What exactly are we optimizing? (relevance, engagement, revenue)
2. What's the scale? (QPS, users, items, latency)
3. What constraints exist? (latency budget, compute, cost)
4. What data is available? (logs, labels, user history)

7.2.2 Frame as ML Problem

Common patterns:

- **Search ranking:** Listwise ranking problem (optimize NDCG)
- **Recommendations:** Two-stage (retrieval + ranking)
- **Ad CTR:** Binary classification (predict click probability)
- **Feed ranking:** Multi-objective (relevance + diversity + freshness)

7.2.3 Define Success Metrics

Two levels:

1. **ML metrics** (offline): NDCG@10, AUC-ROC, precision@K
2. **Business metrics** (online): CTR, conversion, GMV, retention

Example

Example: E-commerce Search
Clarifying questions:

- "What's the latency requirement?" → 200ms p99
- "How many products?" → 10M SKUs
- "What's the query volume?" → 10K QPS peak

ML framing:

- Ranking problem: optimize NDCG@20
- Two-stage: Retrieve 1000 → Rank to 100

Metrics:

- Offline: NDCG@10, MRR (navigational queries)
- Online: CTR, add-to-cart rate, GMV per search

7.3 Step 2: Data Strategy

7.3.1 Data Sources

What data do we have?

- User logs (searches, clicks, purchases)
- Product catalog (title, category, price, inventory)
- User profiles (demographics, history)
- External data (trends, seasonality)

7.3.2 Label Generation

Options:

1. **Explicit labels:** Human raters (expensive, high quality)
2. **Implicit labels:** Clicks, purchases (free, biased)
3. **Hybrid:** Combine both

Label schema for ranking:

- 0 = Irrelevant (no click, bounce)
- 1 = Somewhat relevant (click, short dwell)
- 2 = Relevant (click, medium dwell, add to cart)
- 3 = Highly relevant (purchase)

7.3.3 Handling Biases

Position bias:

- Top results get more clicks (regardless of relevance)
- Solution: Inverse propensity weighting, randomization

Selection bias:

- Only see clicks on shown items (not all items)
- Solution: Exploration (occasionally show random items)

7.4 Step 3: Feature Engineering

Covered in previous section - reference that content
Key points to mention:

- Query features, doc features, query-doc features
- Text matching (BM25), semantic (embeddings)
- Behavioral (CTR, purchases)
- Normalization, encoding, interactions

7.5 Step 4: Model Selection

7.5.1 Propose Multiple Approaches

Always discuss 2-3 options with tradeoffs!

Example

Example: Search Ranking
Approach 1: Traditional Ranking (Baseline)

- BM25 for text matching
- Boost by popularity, rating
- Pros: Fast, interpretable, no training needed
- Cons: No personalization, limited signals

Approach 2: GBDT Ranking (Recommended)

- LambdaMART (XGBoost)
- Features: BM25, category match, CTR, user history
- Optimize NDCG@10
- Pros: Best offline metrics, handles non-linearity, feature importance
- Cons: Need training data, retraining pipeline

Approach 3: Two-Stage with DNN (Advanced)

- Stage 1: Retrieve 10K via BM25 + vector search
- Stage 2: DNN ranking on top-1K
- Pros: Handles semantic search, end-to-end learning
- Cons: Complex, needs more data, higher latency

Recommendation: Start with Approach 2 (GBDT), iterate to 3 if needed.

7.5.2 Justify Your Choice

Always explain:

- Why this model for this problem?
- What are the trade-offs?
- What are alternatives and why not choose them?
- What would you do if requirements change?

7.6 Step 5: Training & Evaluation

7.6.1 Data Split

Time-based split (NOT random!):

```
Train: Days 1-60 (80%)
Val:   Days 61-75 (10%)
Test:  Days 76-90 (10%)
```

Why time-based?

- Prevents data leakage (future can't predict past)
- Realistic evaluation (model serves future data)
- Detects temporal drift

7.6.2 Training Frequency

Domain	Frequency	Reasoning
Search	Weekly	Patterns stable, query dist doesn't change fast
Ads	Daily	Campaigns change, CTR shifts quickly
Recommendations	2-3 days	Medium drift, new items arrive

7.6.3 Evaluation

Offline:

- Hold-out test set evaluation
- NDCG@10, MRR
- Per-query analysis (head vs tail)
- Error analysis (where does model fail?)

Online (A/B Test):

- Traffic: 5-10% treatment, 90-95% control
- Duration: 1-2 weeks (statistical significance)
- Metrics: CTR, conversion, GMV
- Guardrails: Latency p99, zero-result rate

7.7 Step 6: Deployment & Monitoring

7.7.1 Serving Architecture

Latency Budget Example (200ms total):

```
Retrieval:     50ms  (Elasticsearch query)
Feature Fetch: 30ms  (Redis lookup)
Model Infer:   80ms  (XGBoost scoring)
Hydration:     30ms  (DB fetch for top-K)
Network:       10ms
```

7.7.2 Optimization Strategies

Model Optimization:

- Quantization (float32 → int8)
- Pruning (remove weak trees/neurons)
- Distillation (train smaller model to mimic large one)

Serving Optimization:

- Feature store: Redis for low-latency (¡1ms) feature access
- Caching: Cache popular query results (80/20 rule)
- Batch inference: Process multiple queries together
- Two-stage ranking: Cheap first-pass, expensive rerank

7.7.3 Monitoring

Model Metrics:

- NDCG drop >2% → Alert
- Prediction distribution shift → Retrain

Business Metrics:

- CTR drop >5% → Investigate
- Conversion drop → Rollback

System Metrics:

- Latency p99 >500ms → Scale up
- Error rate >0.1% → Alert
- QPS spike → Auto-scale

Data Quality:

- Feature drift (distribution changes)
- Missing values spike
- Label quality degradation

8 ML Coding: Essential Implementations

8.1 Calculate AUC-ROC from Scratch

8.1.1 Understanding AUC-ROC

Definition: Area Under ROC Curve = Probability that a random positive is ranked higher than a random negative.

Intuition: For all positive-negative pairs, how often does the model rank the positive higher?

8.1.2 Implementation

```python
def auc_roc(y_true, y_scores):
    """
    Calculate AUC-ROC from scratch.

    Args:
        y_true: True binary labels (0 or 1)
        y_scores: Predicted scores (higher = more confident)

    Returns:
        AUC-ROC score [0.5, 1.0]
    """
    # Sort by scores descending
    sorted_indices = sorted(range(len(y_scores)),
                            key=lambda i: y_scores[i],
                            reverse=True)

    sorted_labels = [y_true[i] for i in sorted_indices]

    # Count correctly ranked pairs
    num_pos = sum(y_true)
    num_neg = len(y_true) - num_pos

    if num_pos == 0 or num_neg == 0:
        return 0.5  # Undefined, return random

    # For each positive, count negatives ranked below it
    correct_pairs = 0
    negatives_so_far = 0

    for label in sorted_labels:
        if label == 1:  # Positive
            # All negatives seen so far are ranked below
            correct_pairs += negatives_so_far
        else:  # Negative
            negatives_so_far += 1

    # AUC = fraction of correct pairs
    total_pairs = num_pos * num_neg
    auc = correct_pairs / total_pairs

    return auc

```

```
# Test
y_true = [1, 0, 1, 0, 1]
y_scores = [0.9, 0.3, 0.8, 0.2, 0.7]
print(f"AUC: {auc_roc(y_true, y_scores)}")  # Should be 1.0
```

8.1.3 Time Complexity

- Sorting: $O(n \log n)$
- Counting pairs: $O(n)$
- Total: $O(n \log n)$

8.2 Calculate NDCG@K

8.2.1 Implementation

```
import numpy as np

def dcg_at_k(relevances, k):
    """
    Calculate DCG@K.

    Args:
        relevances: Relevance scores in ranked order
        k: Cutoff position

    Returns:
        DCG@K score
    """
    relevances = np.array(relevances)[:k]
    if len(relevances) == 0:
        return 0.0

    # DCG = sum((2^rel - 1) / log2(pos + 1))
    positions = np.arange(1, len(relevances) + 1)
    dcg = np.sum((2**relevances - 1) / np.log2(positions + 1))

    return dcg

def ndcg_at_k(y_true, y_scores, k):
    """
    Calculate NDCG@K.

    Args:
        y_true: True relevance scores
        y_scores: Predicted scores
        k: Cutoff position

    Returns:
        NDCG@K score [0, 1]
    """
    # Sort by predicted scores (descending)
```

```
    sorted_indices = np.argsort(y_scores)[::-1]
    sorted_relevances = np.array(y_true)[sorted_indices]

    # Actual DCG
    dcg = dcg_at_k(sorted_relevances, k)

    # Ideal DCG (sort by true relevances)
    ideal_relevances = sorted(y_true, reverse=True)
    idcg = dcg_at_k(ideal_relevances, k)

    if idcg == 0:
        return 0.0

    return dcg / idcg

# Test
y_true = [3, 2, 1, 0, 2]      # True relevances
y_scores = [0.9, 0.7, 0.5, 0.3, 0.8]  # Predictions
print(f"NDCG@5: {ndcg_at_k(y_true, y_scores, 5)}")
```

8.3 Logistic Regression with Gradient Descent

```
import numpy as np

def sigmoid(z):
    """Sigmoid activation function."""
    return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

def logistic_regression(X, y, learning_rate=0.01, epochs=1000):
    """
    Train logistic regression with gradient descent.

    Args:
        X: Features (n_samples, n_features)
        y: Binary labels (n_samples,)
        learning_rate: Step size
        epochs: Number of iterations

    Returns:
        weights, bias
    """
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)
    bias = 0

    for epoch in range(epochs):
        # Forward pass
        linear = np.dot(X, weights) + bias
        predictions = sigmoid(linear)

        # Compute gradients
        dw = (1/n_samples) * np.dot(X.T, (predictions - y))
        db = (1/n_samples) * np.sum(predictions - y)
```

```python

        # Update parameters
        weights -= learning_rate * dw
        bias -= learning_rate * db

        # Log loss (optional monitoring)
        if epoch % 100 == 0:
            loss = -np.mean(y * np.log(predictions + 1e-15) +
                            (1 - y) * np.log(1 - predictions + 1e-15))
            print(f"Epoch {epoch}, Loss: {loss:.4f}")

    return weights, bias

# Test
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y = np.array([0, 0, 1, 1])
weights, bias = logistic_regression(X, y)
print(f"Weights: {weights}, Bias: {bias}")
```

+ Action Item

Coding Practice Checklist:

- ☐ Implement AUC-ROC from scratch (no sklearn)
- ☐ Implement NDCG@K from scratch
- ☐ Implement logistic regression with gradient descent
- ☐ Implement K-Means clustering
- ☐ Calculate precision, recall, F1 manually
- ☐ Implement train/test split (time-based)

Practice on: LeetCode, HackerRank, or implement in Jupyter notebook

9 Common Interview Questions & Answers

9.1 Conceptual Questions

9.1.1 Q: Explain pointwise, pairwise, and listwise LTR. When to use each?

Answer:

Pointwise:

- Treats each document independently
- Predicts relevance score for each doc
- Use when: Cold start, baseline, small data
- Example: Logistic regression for relevance classification

Pairwise:

- Learns to compare pairs: which should rank higher?
- Optimizes pairwise ranking preferences
- Use when: Moderate data, need better than baseline
- Example: RankNet, LambdaRank

Listwise:

- Optimizes entire list for ranking metrics (NDCG)
- Considers all documents together
- Use when: Production system, lots of data, want best performance
- Example: LambdaMART (XGBoost with rank:ndcg)

Recommendation: Start with pointwise baseline, move to listwise for production.

9.1.2 Q: How do you handle cold start for new items/users?

Answer:

New Items:

1. **Content-based features:** Use item metadata (category, brand, price)
2. **Popularity bias:** Show to fraction of traffic, measure engagement
3. **Exploration:** Random boosting (epsilon-greedy)
4. **Transfer learning:** Use similar items' patterns

New Users:

1. **Global popularity:** Show trending/popular items
2. **Demographic features:** Use location, device
3. **Quick profiling:** Ask preferences upfront
4. **Rapid learning:** Update user model after each interaction

Example: Netflix's "New User Experience" - asks genre preferences immediately.

9.1.3 Q: How do you detect and handle training/serving skew?

Answer:

Causes:

- Features change between train and serve time
- Example: Inventory (in-stock → out-of-stock)
- Time lag: Training on yesterday's data, serving today

Detection:

- Monitor feature distributions (train vs serve)
- Compare prediction distributions
- Track model performance over time

Solutions:

1. **Point-in-time features:** Train with historical features (as they were)
2. **Bucketize fast-moving features:** Inventory: [0, 1-10, 10-100, 100+]
3. **Exclude problematic features:** If can't fix, don't use
4. **Online learning:** Update model frequently

9.2 System Design Questions

9.2.1 Q: Design a product search ranking system for e-commerce.

Answer Framework:

1. Problem Formulation:

- Optimize NDCG@20 (graded relevance)
- Latency: 200ms p99
- Scale: 10M products, 10K QPS

2. Data:

- Logs: searches, clicks, purchases
- Product catalog: title, category, price, rating
- Labels: Implicit (clicks → rel=1, purchase → rel=3)

3. Features:

- Query-doc: BM25, category match, brand match
- Doc quality: Rating, sales, review count
- Behavioral: Historical CTR, conversion rate

4. Model:

- Two-stage architecture
- Stage 1: BM25 retrieval → 1000 candidates
- Stage 2: LambdaMART ranking → top-100

5. Training:

- Weekly retraining
- Time-based split: Train (60d), Val (10d), Test (10d)
- Optimize NDCG@10

6. Serving:

- ES for retrieval (50ms)
- Redis for features (30ms)
- XGBoost inference (80ms)
- A/B test: CTR, conversion, GMV

9.3 Trade-Off Questions

9.3.1 Q: XGBoost vs Neural Networks for ranking?

Answer:

Dimension	**XGBoost**	**Neural Network**
Data needed	10K-100K samples	100K-1M+ samples
Training time	Minutes to hours	Hours to days
Interpretability	High (feature importance)	Low (black box)
Feature engineering	Manual (but important)	Automatic (embeddings)
Categorical features	Needs encoding	Embeddings (better)
Performance	Excellent (80% use case)	Best (with enough data)

Recommendation:

- Start with XGBoost (faster iteration, good enough)
- Move to NN if: Very large data, text/image features, multi-task learning

10 Final Preparation Checklist

10.1 Knowledge Checklist

+ Action Item

Before your interview, ensure you can:
Fundamentals:

- ☐ Explain bias-variance tradeoff with examples
- ☐ Compare L1 vs L2 regularization
- ☐ Describe gradient descent variants
- ☐ Explain overfitting and solutions

Metrics:

- ☐ Calculate precision, recall, F1 by hand
- ☐ Explain AUC-ROC and when to use
- ☐ Explain NDCG@K formula and intuition
- ☐ Compare MRR vs NDCG vs MAP

Learning to Rank:

- ☐ Explain pointwise, pairwise, listwise LTR
- ☐ Describe LambdaMART algorithm
- ☐ Discuss when to use each approach

Models:

- ☐ Explain why XGBoost is effective
- ☐ Describe Wide & Deep architecture
- ☐ Explain two-tower models for retrieval
- ☐ Compare GBDT vs neural networks

System Design:

- ☐ Memorize 6-step framework
- ☐ Can design search ranking end-to-end
- ☐ Can design recommendation system
- ☐ Can design CTR prediction for ads

Coding:

- ☐ Implement AUC-ROC from scratch
- ☐ Implement NDCG@K from scratch
- ☐ Implement logistic regression

10.2 Day Before Interview

1. **Review** this guide (2 hours)
   - Focus on sections you marked as weak
   - Re-read key insights and examples
2. **Practice** explaining concepts out loud (1 hour)
   - Record yourself
   - Explain to a friend/family
   - Use a rubber duck!
3. **Mock interview** (1 hour)
   - "Design a search ranking system"
   - Use 6-step framework
   - Time yourself (45 min)
4. **Prepare questions** for interviewer
   - About Faire's ML stack
   - About team structure
   - About interesting ML problems
5. **Rest** and get good sleep!

10.3 Interview Day

+ Action Item

Morning of interview:

1. Quick review of framework (30 min)
2. Practice one coding problem (15 min)
3. Review your past ML projects (20 min)
4. Prepare to connect your experience to questions

10.4 During Interview

! Critical

Interview Best Practices:

- **Clarify first:** Ask questions before designing
- **Structure your answer:** Use frameworks (6-step, STAR)
- **Think out loud:** Verbalize your reasoning
- **Draw diagrams:** Visualize architectures
- **Discuss trade-offs:** Every decision has pros/cons
- **Give concrete numbers:** "NDCG 0.75", "latency 100ms", not "good" or "fast"
- **Connect to experience:** "At company X, we did Y and saw Z% improvement"
- **Be honest:** Say "I don't know, but here's how I'd find out"

You're ready for your Faire ML interview!

Trust your preparation and experience.
Good luck!