- How to design and implement an MVP

After the examples above, you're might think real-time recommenders require specialized infra, deep learning models, and all that jazz. **I hope to convince you otherwise.**

We'll briefly go through how to design and build an MVP, focusing on what's commonly viewed as the bottleneck of real-time recommenders: *compute and serving*. In contrast, training is relatively easier and widely discussed.

To begin, I think it's useful to approach DS/ML systems in three broad strokes:

- **Requirements (or constraints)**: What does success look like? What can we not do?
- **Methodology**: How will we use data and code to achieve success?
- **Implementation**: What infrastructure is needed in production?

When defining requirements, we should start from the customer. How will real-time recommendations improve the customer experience, and in turn benefit the business? What business metrics are important? Goals and success metrics will vary based on the business and use case, and will not be defined here.

For our MVP, perhaps more important than requirements are constraints (i.e., how *not* to solve the problem). Here are some constraints for our real-time recommender:

- Latency: App responsiveness is key to user experience. Google found that taking an additional 500ms to generate search results reduced traffic by 20%. Amazon shared that 100ms additional latency reduces profit by 1%. Thus, for our MVP, we set a budget of **200ms latency** (excluding network latency from users to our server).
- Throughput: To assess our design's scalability, we set an expected throughput of **1,000 queries per second (QPS)**, or about 84 million queries a day. We probably won't need 1,000 QPS throughout the day as traffic fluctuates.
- Cost: To achieve a healthy ROI, cost should be a fraction of expected revenue. Assuming real-time recommendations reap an additional 100k monthly, we set an **infra budget of 10k monthly** (i.e., 10%).

We should also consider other aspects such as availability (aka redundancy), security, privacy, ethics, etc. Nonetheless, for our MVP, the first three constraints are technical and business showstoppers which we'll focus on.

To train item embeddings, we adopt the simple but effective word2vec approach, specifically, the skip-gram model. (This is also used by Instagram, Twitter, and Alibaba.) I've previously written about how to create embeddings via word2vec and DeepWalk and won't go into details here.

To generate candidates, we apply k-nearest neigbours (à la YouTube's implementation). However, exact kNN is slow and we don't really need the precision at this stage. Thus, we'll use *approximate* nearest neighbours (ANN) instead.

There are several open-sourced ANN implementations, such as Facebook's FAISS, Google's ScANN, and Hierarchical Navigable Small Word Graphs (hnswlib). We'll benchmark them on the recall/latency trade-off. To mimic production conditions, each query consists of a single embedding (i.e., batch size = 1). The graph below shows ScaNN outperforming the other two implementations. (FAISS, in particular, is optimized for batch queries.)

Benchmarking ANNs on recall vs latency; top-right is better.

To rank candidates, we start with a single-layer neural network (read: logistic regression) with cross features—simple, yet difficult to beat. Cross features are created by combining all possible pairs of features to capture their interactions. While this blows up the feature space, it's not a problem for logistic regression.

Beyond machine learning metrics (i.e., recall@k, NDCG, AUC), it's probably more important to consider business metrics (circling back to requirements). Different goals call for different metrics:

- If our goal is **increased engagement**, or to sell ads based on clicks, we'll want to consider absolute clicks, click-thru-rate, daily average users.
- If our goal is **customer acquisition**, we'll want to optimize for first sale (e.g., units sold, conversion), monthly average users, and be willing to take a hit on revenue.
- If our goal is **increased revenue**, we'll want to focus on revenue per session, average basket size, and customer lifetime value.

From experience, business stakeholders will usually have conflicting goals. Marketing wants to make the first sale (regardless of item price), customer experience wants to takedown poor quality products (even if they sell well), and commercial wants to maximize profit (by selling higher-priced items). Getting everyone to agree on key metrics and guardrails can be more difficult than improving our models.

Will our MVP require specialized infrastructure? Not necessarily. A cost-effective approach is to use EC2 instances that can scale horizontally with a load balancer in front. To further simplify things, we can just use AWS SageMaker.

SageMaker takes care of load-balancing on multiple instances (image source)

To assess latency and throughput, we have various options including `serverless-artillery` (AWS guide) and `locust`. Running several load tests showed that a SageMaker endpoint backed by 30 m5.xlarge instances was able to serve 1,200 queries per second without breaking a sweat. At this throughput, median latency was 25ms while the 99th percentile was 65ms. There were zero errors.

With regard to cost, m5.xlarge (16 gb RAM, 4 CPUs) instances in US West (Oregon) have an hourly rate of $0.269. Running 30 instances for 28 days works out to approximately 5.5k, well within our 10k budget. Using reserved instances and/or auto-scaling can help with lowering cost.

Our simple MVP deliberately excludes several considerations. For example, to let other services query our item embeddings, we might want to expose them as a separate service. This will require additional infra (e.g., DynamoDB) which will increase cost and ops burden. The additional service call (for item embeddings) will also add latency (10-30ms) though it can be minimized via a good network setup. Also, how can we expose our candidate generation and ranking services via generic APIs, so other users can mix-and-match as required? We'll want to consider these in the long-term roadmap.

Building a real-time recommender need not be hard

I hope this improves your understanding of real-time machine learning in the context of recommendation systems, and demonstrates that it's **not** an insurmountable challenge. Libraries (e.g., ScaNN) and managed services (e.g., AWS SageMaker) abstract away much of the nitty-gritty such as optimization, health checks, auto-scaling, recovery, etc. Building on them allows for effective, low-cost, real-time ML.