Meta Interview 2025-04-07

Preparation 2025-04-02

Machine Learning System Design – Educative

Introduction

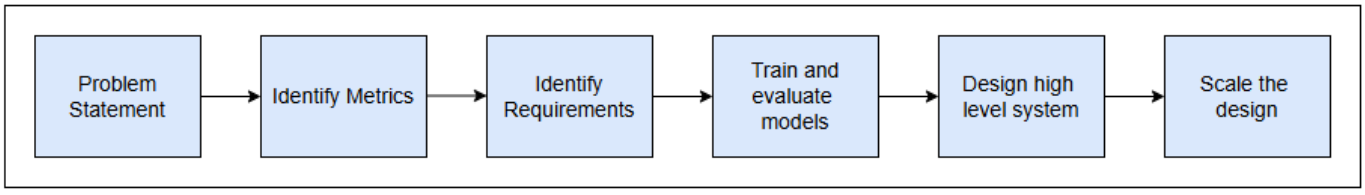
Learn how to approach Machine Learning System Design.

1. What should you expect in a machine learning interview?

- Most major companies, i.e. Facebook, LinkedIn, Google, Amazon, and Snapchat, expect Machine Learning engineers to have solid engineering foundations and hands-on Machine Learning experiences. This is why interviews for Machine Learning positions share similar components with interviews for traditional software engineering positions. The candidates go through a similar method of problem solving (Leetcode style), system design, knowledge of machine learning and machine learning system design.
- The standard development cycle of machine learning includes data collection, problem formulation, model creation, implementation of models, and enhancement of models. It is in the company's best interest throughout the interview to gather as much information as possible about the competence of applicants in these fields. There are plenty of resources on how to train machine learning models and how to deploy models with different tools. However, there are no common guidelines for approaching machine learning system design from end to end. This was one major reason for designing this course.

2. How will this course help you?

In this course, we will learn how to approach machine learning system design from a top-down view. It's important for candidates to realize the challenges early on and address them at a structural level. Here is one example of the thinking flow.



The 6 basic steps to approach Machine Learning System Design

The 6 basic steps to approach Machine Learning System Design

Problem statement

It's important to state the correct problems. It is the candidates job to understand the intention of the design and why it is being optimized. It's important to make the right assumptions and discuss them explicitly with interviewers. For example, in a LinkedIn feed design interview, the interviewer might ask broad questions:

Design LinkedIn Feed Ranking.

Asking questions is crucial to filling in any gaps and agreeing on goals. The candidate should begin by asking follow-up questions to clarify the problem statement. For example:

- Is the output of the feed in chronological order?
- How do we want to balance feeds versus sponsored ads, etc.?

If we are clear on the problem statement of designing a Feed Ranking system, we can then start talking about relevant metrics like user agreements.

Identify metrics

During the development phase, we need to quickly test model performance using offline metrics. You can start with the popular metrics like logloss and AUC for binary classification, or RMSE and MAPE for forecast.

Identify requirements

- Training requirements
    - There are many components required to train a model from end to end. These components include the data collection, feature engineering, feature selection, and loss function. For example, if we want to design a YouTube video recommendations model, it's natural that the user doesn't watch a lot of recommended videos. Because of this, we have a lot of negative examples. The question is asked:

How do we train models to handle an imbalance class?

Once we deploy models in production, we will have feedback in real time.

How do we monitor and make sure models don't go stale?

- Inference requirements

Once models are deployed, we want to run inference with low latency (<100ms) and scale our system to serve millions of users.

How do we design inference components to provide high availability and low latency?

Train and evaluate model

- There are usually three components: feature engineering, feature selection, and models. We will use all the modern techniques for each component.

- For example, in Rental Search Ranking, we will discuss if we should use ListingID as embedding features. In Estimate Food Delivery Time, we will discuss how to handle the latitude and longitude features efficiently.

Design high level system

In this stage, we need to think about the system components and how data flows through each of them. The goal of this section is to identify a minimal, viable design to demonstrate a working system. We need to explain why we decided to have these components and what their roles are.

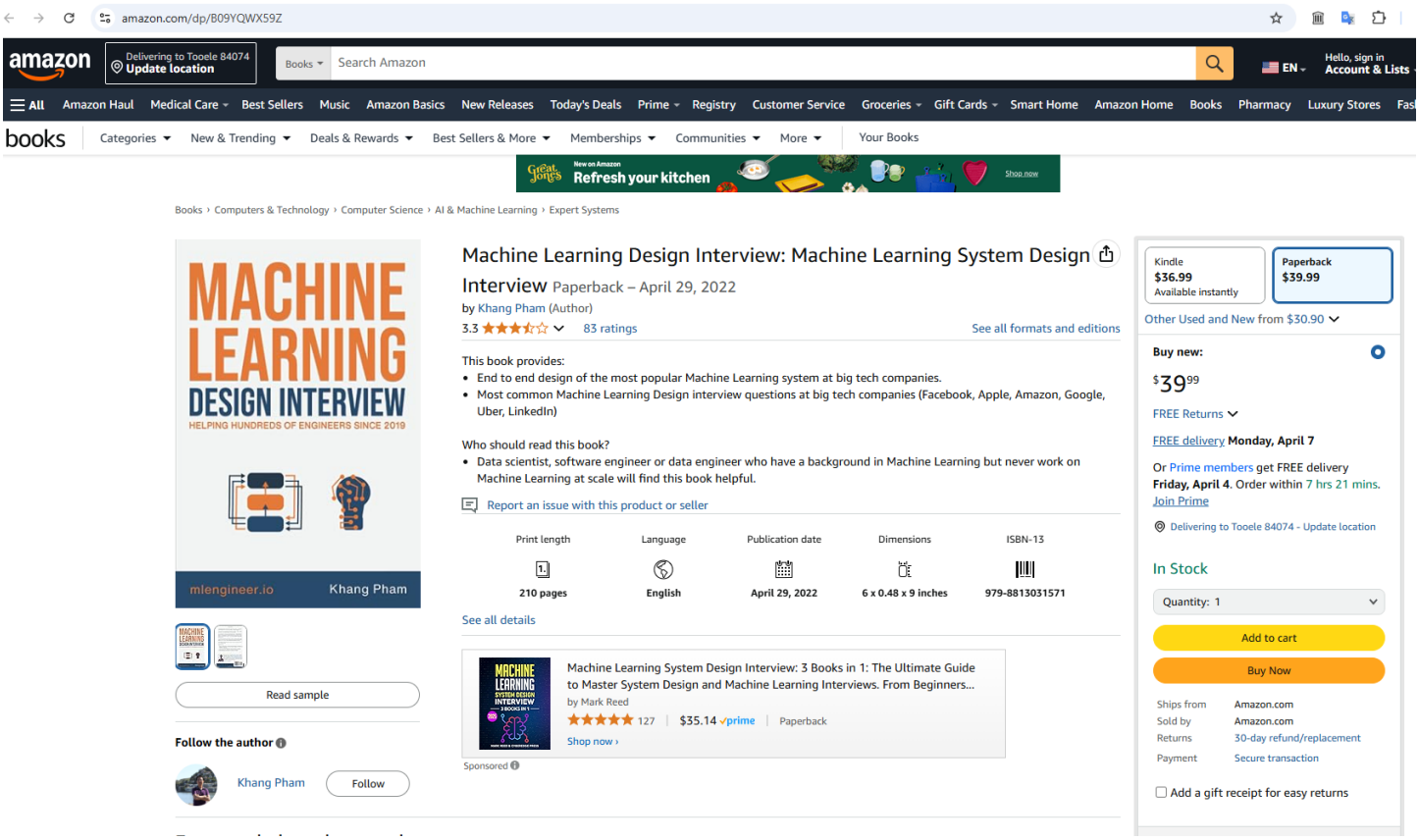
- For example, when designing Video Recommendation systems, we would need two separate components: the Video Candidate Generation Service and the Ranking Model Service.

Scale the design

In this stage, it's crucial to understand system bottlenecks and how to address these bottlenecks. You can start by identifying:

- Which components are likely to be overloaded?
- How can we scale the overloaded components?
- Is the system good enough to serve millions of users?
- How we would handle some components becoming unavailable, etc.
- You can also learn more about how companies scale there design here.

here points to https://rebrand.ly/mleio



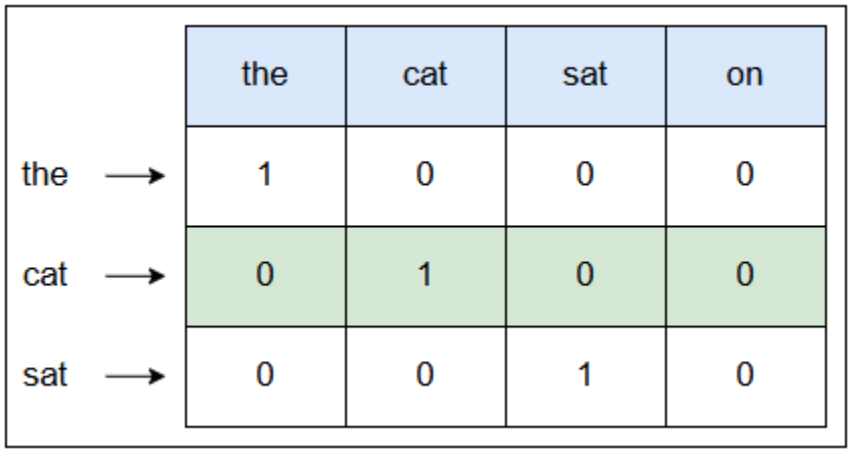
Feature Selection and Feature Engineering

Learn how tech companies like Facebook, Twitter, and Airbnb design their feature selection and feature engineering to serve billions of users.

1. One hot encoding

One hot encoding is a very common technique in feature engineering. It converts categorical variables into a one-hot numeric array.

- One hot encoding is very popular when you have to deal with categorical features that have medium cardinality.



One hot encoding example

Common problems

- Expansive computation and high memory consumption are major problems with one hot encoding. High numbers of values will create high-dimensional feature vectors. For example, if there are one million unique values in a column, it will produce feature vectors that have a dimensionality of one million.
- One hot encoding is not suitable for Natural Language Processing tasks. Microsoft Word's dictionary is usually large, and we can't use one hot encoding to represent each word as the vector is too big to store in memory.

Best practices

- Depending on the application, some levels/categories that are not important, can be grouped together in the "Other" class.
- Make sure that the pipeline can handle unseen data in the test set.

In Python, there are many ways to do one hot encoding, for example, `pandas.get_dummies` and `sklearn OneHotEncoder. pandas.get_dummies` does not "remember" the encoding during training, and if

testing data has new values, it can lead to inconsistent mapping. `OneHotEncoder` is a Scikitlearn Transformer; therefore, you can use it consistently during training and predicting.

One hot encoding in tech companies

- It's not practical to use one hot encoding to handle large cardinality features, i.e., features that have hundreds or thousands of unique values. Companies like Instacart and DoorDash use more advanced techniques to handle large cardinality features.

2. Feature hashing

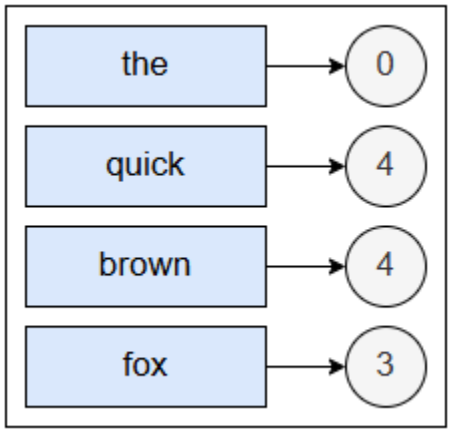
Feature hashing, called the hashing trick, converts text data or categorical attributes with high cardinalities into a feature vector of arbitrary dimensionality.

Benefits

- Feature hashing is very useful for features that have high cardinality with hundreds and thousands of unique values. Hashing trick is a way to reduce the increase in dimension and memory by allowing multiple values to be present/encoded as the same value.

Feature hashing example

- First, you chose the dimensionality of your feature vectors. Then, using a hash function, you convert all values of your categorical attribute (or all tokens in your collection of documents) into a number. Then you convert this number into an index of your feature vector. The process is illustrated in the diagram below.



An illustration of the hashing trick for desired dimensionality of 5 for the originality of K of values of an attributes

- Let's illustrate what it would look like to convert the text "The quick brown fox" into a feature vector. The values for each word in the phrase are:

```
the = 5
quick = 4
brown = 4
fox = 3
```

- Let define a hash function, *h*, that takes a string as input and outputs a non-negative integer. Let the desired dimensionality be 5. By applying the hash function to each word and applying the modulo of 5 to obtain the index of the word, we get:

```
h(the) mod 5 = 0
h(quick) mod 5 = 4
h(brown) mod 5 = 4
h(fox) mod 5 = 3
```

- In this example:
    - `h(the) mod 5 = 0` means that we have one word in dimension **0** of the feature vector.
    - `h(quick) mod 5 = 4` and `h(brown) mod 5 = 4` means that we have two words in dimension **4** of the feature vector.
    - `h(fox) mod 5 = 3` means that we have one word in dimension **3** of the feature vector.
    - As you can see, that there are no words in dimensions 1 or 2 of the vector, so we keep them as 0.
- Finally, we have the feature vector as: `[1, 0, 0, 1, 2]`.
- As you can see, there is a collision between words “quick” and “brown.” They are both represented by dimension 4. The lower the desired dimensionality, the higher the chances of collision. To reduce the probability of collision, we can increase the desired dimensions. This is the trade-off between speed and quality of learning.

Commonly used hash functions are MurmurHash3, Jenkins, CityHash, and MD5.

https://en.wikipedia.org/wiki/MurmurHash

From Wikipedia, the free encyclopedia
MurmurHash is a non-cryptographic hash function suitable for general hash-based lookup.[1][2][3] It was created by Austin Appleby in 2008[4] and, as of 8 January 2016,[5] is hosted on GitHub along with its test suite named SMHasher. It also exists in a number of variants,[6] all of which have been released into the public domain. The name comes from two basic operations, multiply (MU) and rotate (R), used in its inner loop.

Unlike cryptographic hash functions, it is not specifically designed to be difficult to reverse by an adversary, making it unsuitable for cryptographic purposes.

Variants
MurmurHash1
The original MurmurHash was created as an attempt to make a faster function than Lookup3.[7] Although successful, it had not been tested thoroughly and was not capable of providing 64-bit hashes as in Lookup3. Its design would be later built upon in MurmurHash2, combining a multiplicative hash (similar to the Fowler–Noll–Vo hash function) with an Xorshift.

MurmurHash2

MurmurHash2[8] yields a 32- or 64-bit value. It comes in multiple variants, including some that allow incremental hashing and aligned or neutral versions.

MurmurHash2 (32-bit, x86)—The original version; contains a flaw that weakens collision in some cases.[9]
MurmurHash2A (32-bit, x86)—A fixed variant using Merkle–Damgård construction. Slightly slower.
CMurmurHash2A (32-bit, x86)—MurmurHash2A, but works incrementally.
MurmurHashNeutral2 (32-bit, x86)—Slower, but endian- and alignment-neutral.
MurmurHashAligned2 (32-bit, x86)—Slower, but does aligned reads (safer on some platforms).
MurmurHash64A (64-bit, x64)—The original 64-bit version. Optimized for 64-bit arithmetic.
MurmurHash64B (64-bit, x86)—A 64-bit version optimized for 32-bit platforms. It is not a true 64-bit hash due to insufficient mixing of the stripes.[10]
The person who originally found the flaw[clarification needed] in MurmurHash2 created an unofficial 160-bit version of MurmurHash2 called MurmurHash2_160.[11]

MurmurHash3
The current version, completed April 3, 2011, is MurmurHash3,[12][13] which yields a 32-bit or 128-bit hash value. When using 128-bits, the x86 and x64 versions do not produce the same values, as the algorithms are optimized for their respective platforms. MurmurHash3 was released alongside SMHasher, a hash function test suite.

Implementations
The canonical implementation is in C++, but there are efficient ports for a variety of popular languages, including Python,[14] C,[15] Go,[16] C#,[13][17] D,[18] Lua, Perl,[19] Ruby,[20] Rust,[21] PHP,[22][23] Common Lisp,[24] Haskell,[25] Elm,[26] Clojure,[27] Scala,[28] Java,[29][30] Erlang,[31] Swift,[32] Object Pascal,[33] Kotlin,[34] JavaScript,[35] and OCaml.[36]

It has been adopted into a number of open-source projects, most notably libstdc++ (ver 4.6), nginx (ver 1.0.1),[37] Rubinius,[38] libmemcached (the C driver for Memcached),[39] npm (nodejs package manager),[40] maatkit,[41] Hadoop,[1] Kyoto Cabinet,[42] Cassandra,[43][44] Solr,[45] vowpal wabbit,[46] Elasticsearch,[47] Guava,[48] Kafka,[49] and RedHat Virtual Data Optimizer (VDO).[50]

Vulnerabilities
Hash functions can be vulnerable to collision attacks, where a user can choose input data in such a way so as to intentionally cause hash collisions. Jean-Philippe Aumasson and Daniel J. Bernstein were able to show that even implementations of MurmurHash using a randomized seed are vulnerable to so-called HashDoS attacks.[51] With the use of differential cryptanalysis, they were able to generate inputs that would lead to a hash collision. The authors of the attack recommend using their own SipHash instead.

Algorithm
algorithm Murmur3_32 is

```
    // Note: In this version, all arithmetic is performed with unsigned 32-bit integers.
    //       In the case of overflow, the result is reduced modulo 232.
    input: key, len, seed

    c1 ← 0xcc9e2d51
    c2 ← 0x1b873593
    r1 ← 15
    r2 ← 13
    m ← 5
    n ← 0xe6546b64

    hash ← seed

    for each fourByteChunk of key do
        k ← fourByteChunk

        k ← k × c1
        k ← k ROL r1
        k ← k × c2

        hash ← hash XOR k
        hash ← hash ROL r2
        hash ← (hash × m) + n

    with any remainingBytesInKey do
        remainingBytes ← SwapToLittleEndian(remainingBytesInKey)
        // Note: Endian swapping is only necessary on big-endian machines.
        //       The purpose is to place the meaningful digits towards the low end of the value,
        //       so that these digits have the greatest potential to affect the low range digits
        //       in the subsequent multiplication.  Consider that locating the meaningful digits
        //       in the high range would produce a greater effect upon the high digits of the
        //       multiplication, and notably, that such high digits are likely to be discarded
        //       by the modulo arithmetic under overflow.  We don't want that.

        remainingBytes ← remainingBytes × c1
        remainingBytes ← remainingBytes ROL r1
        remainingBytes ← remainingBytes × c2

        hash ← hash XOR remainingBytes

    hash ← hash XOR len

    hash ← hash XOR (hash >> 16)
```

```
    hash ← hash × 0x85ebca6b
    hash ← hash XOR (hash >> 13)
    hash ← hash × 0xc2b2ae35
    hash ← hash XOR (hash >> 16)
```

A sample C implementation follows (for little-endian CPUs):

```
static inline uint32_t murmur_32_scramble(uint32_t k) {
    k *= 0xcc9e2d51;
    k = (k << 15) | (k >> 17);
    k *= 0x1b873593;
    return k;
}
uint32_t murmur3_32(const uint8_t* key, size_t len, uint32_t seed)
{
        uint32_t h = seed;
    uint32_t k;
    /* Read in groups of 4. */
    for (size_t i = len >> 2; i; i--) {
        // Here is a source of differing results across endiannesses.
        // A swap here has no effects on hash properties though.
        memcpy(&k, key, sizeof(uint32_t));
        key += sizeof(uint32_t);
        h ^= murmur_32_scramble(k);
        h = (h << 13) | (h >> 19);
        h = h * 5 + 0xe6546b64;
    }
    /* Read the rest. */
    k = 0;
    for (size_t i = len & 3; i; i--) {
        k <<= 8;
        k |= key[i - 1];
    }
    // A swap is *not* necessary here because the preceding loop already
    // places the low bytes in the low places according to whatever endianness
    // we use. Swaps only apply when the memory is copied in a chunk.
    h ^= murmur_32_scramble(k);
    /* Finalize. */
        h ^= len;
        h ^= h >> 16;
        h *= 0x85ebca6b;
        h ^= h >> 13;
        h *= 0xc2b2ae35;
        h ^= h >> 16;
        return h;
}
```

Tests

Test string	Seed value	Hash value (hexadecimal)	Hash value (decimal)
	0x00000000	0x00000000	0
	0x00000001	0x514E28B7	1,364,076,727
	0xffffffff	0x81F16F39	2,180,083,513
test	0x00000000	0xba6bd213	3,127,628,307
test	0x9747b28c	0x704b81dc	1,883,996,636
Hello, world!	0x00000000	0xc0363e43	3,224,780,355
Hello, world!	0x9747b28c	0x24884CBA	612,912,314
The quick brown fox jumps over the lazy dog	0x00000000	0x2e4ff723	776,992,547
The quick brown fox jumps over the lazy dog	0x9747b28c	0x2FA826CD	799,549,133

https://en.wikipedia.org/wiki/Jenkins_hash_function

Jenkins hash function

From Wikipedia, the free encyclopedia

The **Jenkins hash functions** are a family of non-cryptographic hash functions for multi-byte keys designed by Bob Jenkins. The first one was formally published in 1997.

The hash functions [edit]

one_at_a_time [edit]

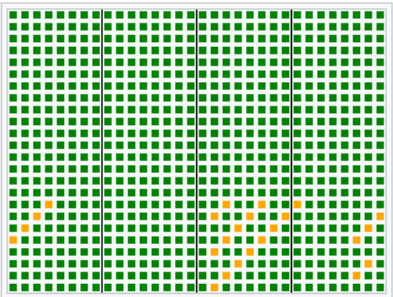
Jenkins's **one_at_a_time** hash is adapted here from a WWW page by Bob Jenkins,[1] which is an expanded version of his *Dr. Dobb's* article.[2] It was originally created to fulfill certain requirements described by Colin Plumb, a cryptographer, but was ultimately not put to use.[1]

```
uint32_t jenkins_one_at_a_time_hash(const uint8_t* key, size_t length) {
  size_t i = 0;
  uint32_t hash = 0;
  while (i != length) {
    hash += key[i++];
    hash += hash << 10;
    hash ^= hash >> 6;
  }
  hash += hash << 3;
  hash ^= hash >> 11;
  hash += hash << 15;
  return hash;
}
```

Sample hash values for **one_at_a_time** hash function.

```
one_at_a_time("a", 1)
0xca2e9442
one_at_a_time("The quick brown fox jumps over the lazy dog", 43)
0x519e91f5
```

The avalanche behavior of this hash is shown on the right.



Avalanche behavior of Jenkins One-at-a-time hash over 3-byte keys

Each of the 24 rows corresponds to a single bit in the 3-byte input key, and each of the 32 columns corresponds to a bit in the output hash. Colors are chosen by how well the input key bit affects the given output hash bit: a green square indicates good mixing behavior, a yellow square weak mixing behavior, and red would indicate no mixing. Only a few bits in the last byte of the input key are weakly mixed to a minority of bits in the output hash.

Standard implementations of the Perl programming language prior to version 5.28 included Jenkins's one-at-a-time hash or a hardened variant of it, which was used by default.[3][4]

lookup2 [edit]

The **lookup2** function was an interim successor to one-at-a-time. It is the function referred to as "My Hash" in the 1997 Dr. Dobbs journal article, though it has been obsoleted by subsequent functions that Jenkins has released. Applications of this hash function are found in:

- the SPIN model checker, for probabilistic error detection. In a paper about this program, researchers Dillinger and Manolios note that lookup2 is "a popular choice among implementers of hash tables and Bloom filters". They study lookup2 and a simple extension of it that produces 96-bit rather than 32-bit hash values.[5]
- The Netfilter firewall component of Linux,[6] where it replaced an earlier hash function that was too sensitive to collisions. The resulting system, however, was shown to still be sensitive to hash flooding attacks, even when the Jenkins hash is randomized using a secret key.[7]
- The program that solved the game of kalah used the Jenkins hash function, instead of the Zobrist hashing technique more commonly used for this type of problem; the reasons for this choice were the speed of Jenkins' function on the small representations of kalah boards, as well as the fact that the basic rule of kalah can radically alter the board, negating the benefit of Zobrist's incremental computation of hash functions.[8]

lookup3 [edit]

The **lookup3** function consumes input in 12 byte (96 bit) chunks.[9] It may be appropriate when speed is more important than simplicity. Note, though, that any speed improvement from the use of this hash is only likely to be useful for large keys, and that the increased complexity may also have speed consequences such as preventing an optimizing compiler from inlining the hash function.

The lookup3 function was incorporated into Hierarchical Data Format 5 as a checksum for internal data structures based on its relative strength and speed in comparison to CRC32 and Fletcher32. [10]

The avalanche behavior of this hash is shown on the

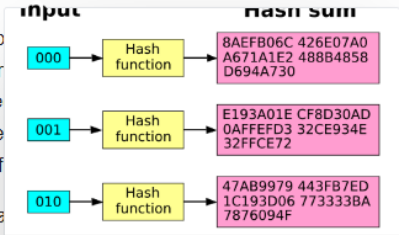
Each o
columr
bit affe
square
byte of

Standa
Jenkin

looku

The lo
journal
found i

- the
that lookup2 is a popular choice among impleme



In cryptography, the **avalanche effect** is the desirable property of cryptographic algorithms, typically block ciphers and cryptographic hash functions, wherein if an input is changed slightly, the output changes significantly. In the case of high-quality block ciphers, such a small change in either the key or the pla

SpookyHash [edit]

In 2011 Jenkins released a new 128-bit hash function called SpookyHash.[11] SpookyHash is significantly faster than lookup3.

Example for V2 (little-endian x64):

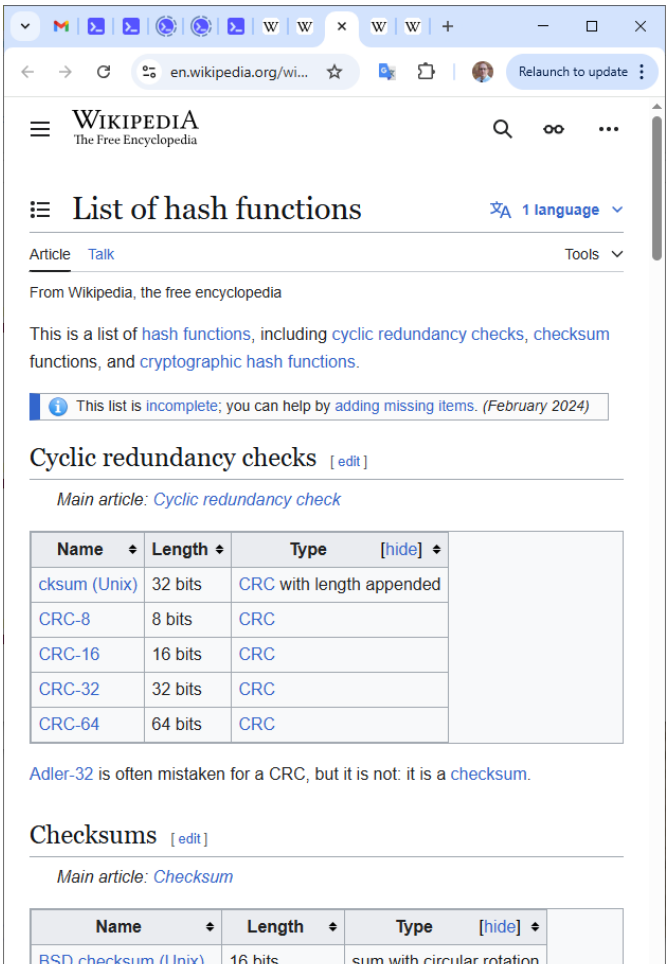
The short method for less than 192 bytes (43 bytes):

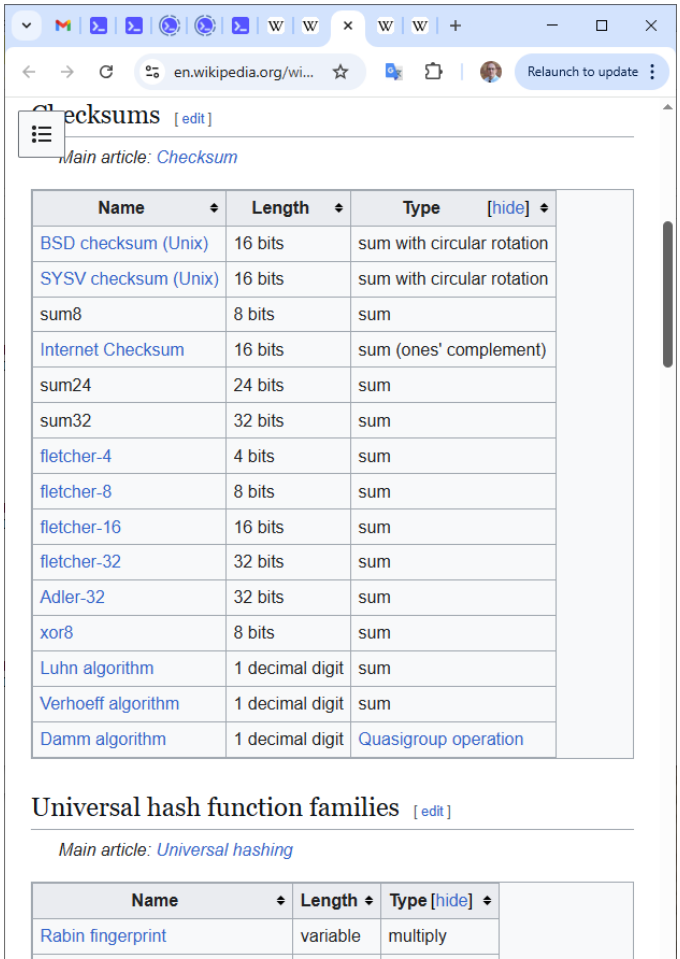
```
Hash128("The quick brown fox jumps over the lazy dog")
2b12e846aa0693c71d367e742407341b
```

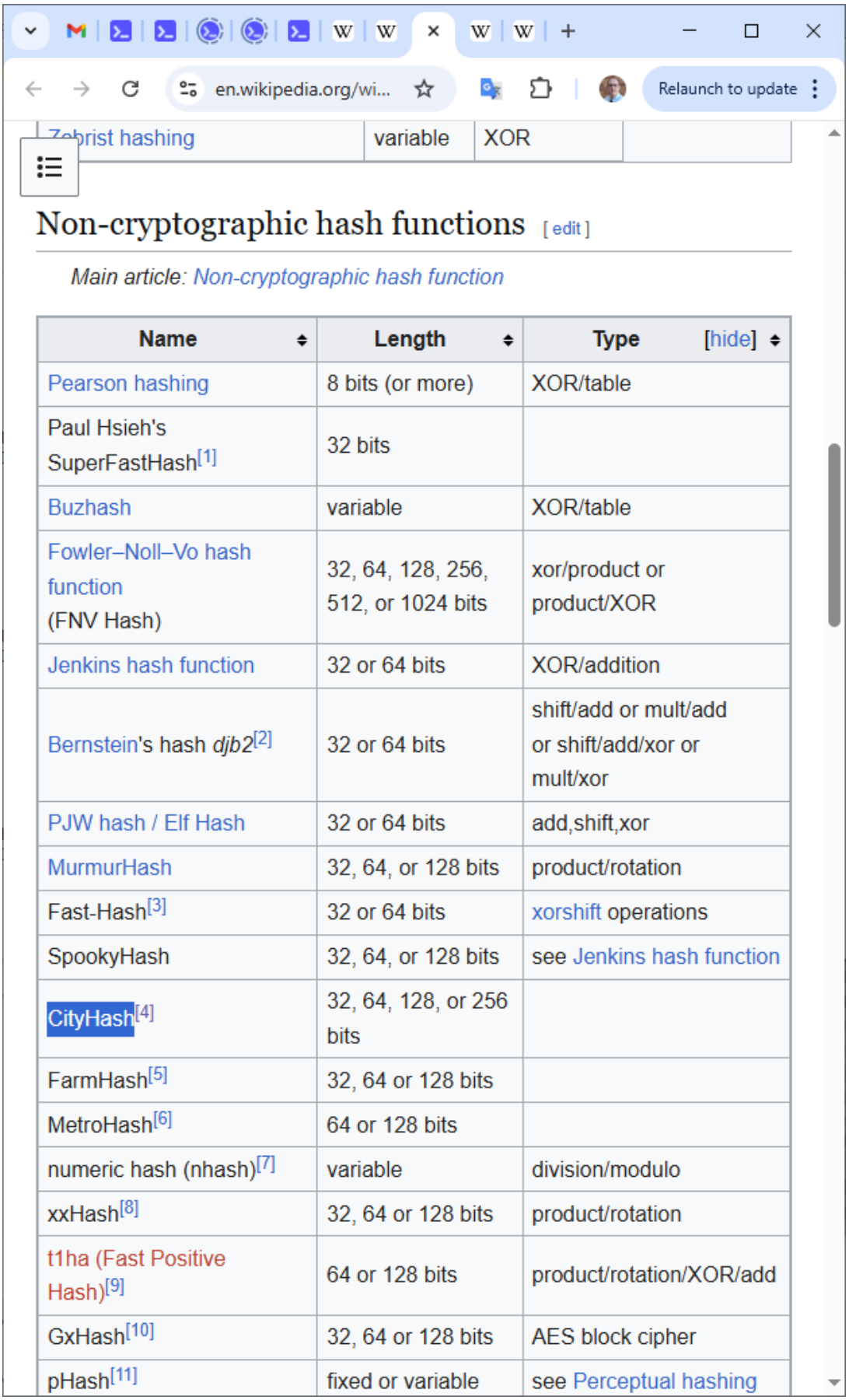
The standard method for more than 191 bytes (219 bytes):

```
Hash128("The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog")
f1b71c6ac5af39e7b69363a60dd29c49
```

https://en.wikipedia.org/wiki/List_of_hash_functions







Zobrist hashing	variable	XOR	

Non-cryptographic hash functions [edit]

Main article: Non-cryptographic hash function

Name	Length	Type [hide]
Pearson hashing	8 bits (or more)	XOR/table
Paul Hsieh's SuperFastHash[1]	32 bits	
Buzhash	variable	XOR/table
Fowler–Noll–Vo hash function (FNV Hash)	32, 64, 128, 256, 512, or 1024 bits	xor/product or product/XOR
Jenkins hash function	32 or 64 bits	XOR/addition
Bernstein's hash *djb2*[2]	32 or 64 bits	shift/add or mult/add or shift/add/xor or mult/xor
PJW hash / Elf Hash	32 or 64 bits	add,shift,xor
MurmurHash	32, 64, or 128 bits	product/rotation
Fast-Hash[3]	32 or 64 bits	xorshift operations
SpookyHash	32, 64, or 128 bits	see Jenkins hash function
CityHash[4]	32, 64, 128, or 256 bits	
FarmHash[5]	32, 64 or 128 bits	
MetroHash[6]	64 or 128 bits	
numeric hash (nhash)[7]	variable	division/modulo
xxHash[8]	32, 64 or 128 bits	product/rotation
t1ha (Fast Positive Hash)[9]	64 or 128 bits	product/rotation/XOR/add
GxHash[10]	32, 64 or 128 bits	AES block cipher
pHash[11]	fixed or variable	see Perceptual hashing

https://en.wikipedia.org/wiki/List_of_hash_functions#cite_note-4

https://github.com/google/cityhash

CityHash, a family of hash functions for strings.

Introduction

CityHash provides hash functions for strings. The functions mix the input bits thoroughly but are not suitable for cryptography. See "Hash Quality," below, for details on how CityHash was tested and so on.

We provide reference implementations in C++, with a friendly MIT license.

CityHash32() returns a 32-bit hash.

CityHash64() and similar return a 64-bit hash.

CityHash128() and similar return a 128-bit hash and are tuned for strings of at least a few hundred bytes. Depending on your compiler and hardware, it's likely faster than CityHash64() on sufficiently long strings. It's slower than necessary on shorter strings, but we expect that case to be relatively unimportant.

CityHashCrc128() and similar are variants of CityHash128() that depend on _mm_crc32_u64(), an intrinsic that compiles to a CRC32 instruction on some CPUs. However, none of the functions we provide are CRCs.

CityHashCrc256() is a variant of CityHashCrc128() that also depends on _mm_crc32_u64(). It returns a 256-bit hash.

All members of the CityHash family were designed with heavy reliance on previous work by Austin Appleby, Bob Jenkins, and others. For example, CityHash32 has many similarities with Murmur3a.

Performance on long strings: 64-bit CPUs

We are most excited by the performance of CityHash64() and its variants on short strings, but long strings are interesting as well.

CityHash is intended to be fast, under the constraint that it hash very well. For CPUs with the CRC32 instruction, CRC is speedy, but CRC wasn't designed as a hash function and shouldn't be used as one. CityHashCrc128() is not a CRC, but it uses the CRC32 machinery.

On a single core of a 2.67GHz Intel Xeon X5550, CityHashCrc256 peaks at about 5 to 5.5 bytes/cycle. The other CityHashCrc functions are wrappers around CityHashCrc256 and should have similar performance on long strings. (CityHashCrc256 in v1.0.3 was even faster, but we decided it wasn't as thorough as it should be.) CityHash128 peaks at about 4.3 bytes/cycle. The fastest Murmur variant on that hardware, Murmur3F, peaks at about 2.4 bytes/cycle. We expect the peak speed of CityHash128 to dominate CityHash64, which is aimed more toward short strings or use in hash tables.

For long strings, a new function by Bob Jenkins, SpookyHash, is just slightly slower than CityHash128 on Intel x86-64 CPUs, but noticeably faster on AMD x86-64 CPUs. For hashing long strings on AMD CPUs and/or CPUs without the CRC instruction, SpookyHash may be just as good or better than any of the CityHash variants.

Performance on short strings: 64-bit CPUs

For short strings, e.g., most hash table keys, CityHash64 is faster than CityHash128, and probably faster than all the aforementioned functions, depending on the mix of string lengths. Here are a few results from that same hardware, where we (unrealistically) tested a single string length over and over again:

Hash	Results
CityHash64 v1.0.3	7ns for 1 byte, or 6ns for 8 bytes, or 9ns for 64 bytes
Murmur2 (64-bit)	6ns for 1 byte, or 6ns for 8 bytes, or 15ns for 64 bytes
Murmur3F	14ns for 1 byte, or 15ns for 8 bytes, or 23ns for 64 bytes

We don't have CityHash64 benchmarks results for v1.1, but we expect the numbers to be similar.

Performance: 32-bit CPUs

CityHash32 is the newest variant of CityHash. It is intended for
32-bit hardware in general but has been mostly tested on x86. Our benchmarks
suggest that Murmur3 is the nearest competitor to CityHash32 on x86.
We don't know of anything faster that has comparable quality. The speed rankings
in our testing: CityHash32 > Murmur3f > Murmur3a (for long strings), and
CityHash32 > Murmur3a > Murmur3f (for short strings).

Installation

We provide reference implementations of several CityHash functions, written
in C++. The build system is based on autoconf. It defaults the C++
compiler flags to "-g -O2", which is probably slower than -O3 if you are
using gcc. YMMV.

On systems with gcc, we generally recommend:

```
./configure
make all check CXXFLAGS="-g -O3"
sudo make install
```

Or, if your system has the CRC32 instruction, and you want to build everything:

```
./configure --enable-sse4.2
make all check CXXFLAGS="-g -O3 -msse4.2"
sudo make install
```

Note that our build system doesn't try to determine the appropriate compiler
flag for enabling SSE4.2. For gcc it is "-msse4.2". The --enable-sse4.2
flag to the configure script controls whether citycrc.h is installed when
you "make install." In general, picking the right compiler flags can be
tricky, and may depend on your compiler, your hardware, and even how you
plan to use the library.

For generic information about how to configure this software, please try:

```
./configure --help
```

Failing that, please work from city.cc and city*.h, as they contain all the
necessary code.

Usage

The above installation instructions will produce a single library. It will
contain CityHash32(), CityHash64(), and CityHash128(), and their variants,
and possibly CityHashCrc128(), CityHashCrc128WithSeed(), and
CityHashCrc256(). The functions with Crc in the name are declared in
citycrc.h; the rest are declared in city.h.

Limitations

1) CityHash32 is intended for little-endian 32-bit code, and everything else in
the current version of CityHash is intended for little-endian 64-bit CPUs.

All functions that don't use the CRC32 instruction should work in
little-endian 32-bit or 64-bit code. CityHash should work on big-endian CPUs
as well, but we haven't tested that very thoroughly yet.

2) CityHash is fairly complex. As a result of its complexity, it may not
perform as expected on some compilers. For example, preliminary reports
suggest that some Microsoft compilers compile CityHash to assembly that's
10-20% slower than it could be.

Hash Quality

We like to test hash functions with SMHasher, among other things.
SMHasher isn't perfect, but it seems to find almost any significant flaw.
SMHasher is available at http://code.google.com/p/smhasher/

```
SMHasher is designed to pass a 32-bit seed to the hash functions it tests.
No CityHash function is designed to work that way, so we adapt as follows:
For our functions that accept a seed, we use the given seed directly (padded
with zeroes); for our functions that don't accept a seed, we hash the
concatenation of the given seed and the input string.

The CityHash functions have the following flaws according to SMHasher:

(1) CityHash64: none

(2) CityHash64WithSeed: none

(3) CityHash64WithSeeds: did not test

(4) CityHash128: none

(5) CityHash128WithSeed: none

(6) CityHashCrc128: none

(7) CityHashCrc128WithSeed: none

(8) CityHashCrc256: none

(9) CityHash32: none

Some minor flaws in 32-bit and 64-bit functions are harmless, as we
expect the primary use of these functions will be in hash tables.  We
may have gone slightly overboard in trying to please SMHasher and other
similar tests, but we don't want anyone to choose a different hash function
because of some minor issue reported by a quality test.


For more information
====================

http://code.google.com/p/cityhash/

cityhash-discuss@googlegroups.com

Please feel free to send us comments, questions, bug reports, or patches.
```

https://en.wikipedia.org/wiki/MD5

Feature hashing in tech companies

- Feature hashing is popular in many tech companies like Booking, Facebook, Yahoo, Yandex, Avazu and Criteo.
- One problem with hashing is collision. If the hash size is too small, more collisions will happen and negatively affect model performance. On the other hand, the larger the hash size, the more it will consume memory.
- Collisions also affect model performance. With high collisions, a model won't be able to differentiate coefficients between feature values. For example, the coefficient for "User login/ User logout" might end up the same, which makes no sense.

Depending on application, you can choose the number of bits for feature hashing that provide the correct balance between model accuracy and computing cost during model training. For example, by increasing hash size we can improve performance, but the training time will increase as well as memory consumption.

3. Crossed feature

- Crossed features, or **conjunction**, between two categorical variables of cardinality c1 and c2 is just another categorical variable of cardinality c1 × c2. If c1 and c2 are large, the conjunction feature has high cardinality, and the use of the hashing trick is even more critical in this case. Crossed feature is usually used with a hashing trick to reduce high dimensions.
- As an example, suppose we have Uber pick-up data with latitude and longitude stored in the database, and we want to predict demand at a certain location. If we just use the feature latitude for learning, the model might learn that a city block at a particular latitude is more likely to have a higher demand than others. This is similar for the feature longitude. However, a feature cross of longitude by latitude would represent a well-defined city block. Consequently, the model will learn more accurately.

Uber Pickups Map in 2015 (source Uber)

Read more about different techniques in handling latitude/longitude here: Haversine distance, Manhattan distance.

Crossed feature in tech companies

This technique is commonly applied at many tech companies. LinkedIn uses crossed features between user location and user job title for their Job recommendation model. Airbnb also uses cross features for their Search Ranking model.

4. Embedding

Feature embedding is an emerging technique that aims to transform features from the original space into a new space to support effective machine learning. The purpose of embedding is to capture semantic meaning of features; for example, similar features will be close to each other in the embedding vector space.

Benefits

- Both one hot encoding and feature hashing can represent features in another dimension. However, these representations do not usually preserve the semantic meaning of each feature. For example, in the **Word2Vector** representation, embedding words into dense multidimensional representation helps to improve the prediction of the next words significantly.

Press+to interact

A 4-dimensional embedding

- As an example, each word can be represented as a d dimension vector, and we can train our supervised model. We then use the output of one of the fully connected layers of the neural network model as

embeddings on the input object. For example, cat embedding can be represented as a [1.2, -0.1, 4.3, 3.2] vector.

How to generate/learn embedding vector?

- For popular deep learning frameworks like TensorFlow, you need to define the dimension of embedding and network architecture. Once defined, the network can learn embedding automatically. For example:
- # Embed a 1,000 word vocabulary into 5
- dimensions.
- embedding_layer =
- tf.keras.layers.Embedding(1000, 5)

Embedding in tech companies

This technique is commonly applied at many tech companies.

- Twitter uses Embedding for UserIDs and cases like recommendations, nearest neighbor searches, and transfer learning.
- Doordash uses Store Embedding (store2vec) to personalize the store feed. Similar to word2vec, each store is one word and each sentence is one user session. Then, to generate vectors for a consumer, we sum the vectors for each store they ordered from in the past 6 months or a total of 100 orders. Then, the distance between a store and a consumer is determined by taking the cosine distance between the store's vector and the consumer's vector.
- Similarly, Instagram uses account embedding to recommend relevant content (photos, videos, and Stories) to users.

The embedding dimensionality is usually determined experimentally or from experience. In TensorFlow documentation, they recommend: d=D4*d*=4*D*. Where D*D* is the number of categories. Another way is to treat d*d* as a hyperparameter, and we can tune on a downstream task.

In large scale production, embedding features are usually pre-computed and stored in key/value storage to reduce inference latency.

5. Numeric features

Normalization

- For numeric features, normalization can be done to make the mean equal 0, and the values be in the range [-1, 1]. There are some cases where we want to normalize data between the range [0, 1].

v=v−min_of_vmax_of_v−min_of_v*v*=*max_of_v*−*min_of_v**v*−*min_of_v*

where,

v*v* is feature value,

min_of_v*min_of_v* is a minimum of feature value,

max_of_vmax_of_v is a maximum of feature value

Standardization

- If features distribution resembles a normal distribution, then we can apply a standardized transformation.

v=v−mean_of_vstd_of_v$v=\frac{v-mean_of_v}{std_of_v}$

where,

vv is feature value,

mean_of_v$mean_of_v$ is a mean of feature value,

std_of_vstd_of_v is the standard deviation of feature value

- If feature distribution resembles power laws, then we can transform it by using the formula:

log(1+v1+median_of_v)$log(\frac{1+v}{1+median\_of\_v})$

In practice, normalization can cause an issue as the values of min and max are usually outliers. One possible solution is “clipping”, where we choose a “reasonable” value for min and max. You can also learn more about how companies apply feature engineering here.

Feature selection and Feature engineering quiz

1.

We have a table with columns UserID, CountryID, CityID, zip code, and age. Which of the following feature engineering is suitable to represent the data in a Machine Learning algorithm?

A.

Apply one hot encoding for all columns

B.

Apply Entity Embedding for CountryID and CityID; one hot Encoding of UserID and zipcode, and normalization of age.

C.

Apply Entity Embedding for CountryID, CityID, UserID, and zip code, and normalization of age.

Font

F

f

font

f