Hashing trick,
Random projections

Categorical data: 1-hot encoding

	item_type	seller_id
0	book	1234
1	phone	45
2	clothes	45
3	book	46
4	shoes	234
...	...	...

X := “item_type is” ‘book’, “seller” is ‘1234’ → $X \in \mathbb{R}^d$

- List all modalities of “item_type”
- Assign one index to each modality.
- Idem for other variables

```
{ 'shoes':   1,
  'phone':   2,
  'book':    3,
  'clothes': 4,
  ... }
```

0	type is “shoes”
0	type is “phone”
1	type is “book” ✓
0	type is “clothes
…	
0	seller is “45”
1	seller is “1234” ✓
0	seller is “5678”
…	

$\in \mathbb{R}^d$ with
d:= nb distinct “types” + nb distinct “seller”

Polynomial Kernel and « Crossfeatures»

“Quadratic Kernel”

Formally: $X.X^T \in \text{Matrices}^{d,d} \cong \mathbb{R}^{d^2}$

X^T: 0 0 1 0 …. 0 1 0 …

		X	
0	*type is “shoes”*	0	0 0 0 0 …. 0 0 0 …
0	*type is “phone”*	0	0 0 0 0 …. 0 0 0 …
1	*type is “book”* ✓	1	0 0 1 0 …. 0 1 0 …
0	*type is “clothes*	0	0 0 0 0 …. 0 0 0 …
…		…	…
0	*seller is “45”*	0	
1	*seller is “1234”* ✓	1	
0	*seller is “5678”*	0	
…		…	

- Type is “book” *and* seller is “1234”

“*Crossfeature*” between “type” and “seller”

$X \in \mathbb{R}^d$

Or use index:

```
('phone','45'):   1000,
('phone','46'):   1001,
('phone','47'):   1002,
 ...
('book','45'):   1013,
('book','46'):   1014,
 ...
```

0	*type is “shoes”*
0	*type is “phone”*
1	*type is “book”* ✓
0	*type is “clothes*
…	
0	*seller is “45”*
1	*seller is “1234”* ✓
0	*seller is “5678”*
…	
0	*“book” and “45”*
1	*“book” and “1234”* ✓
0	*“book” and “5678”*
…	

$\in \mathbb{R}^{d^2}$

Help I now have too many features !

- I have quite a few high cardinality categorical columns
  - Even building a vocabulary of possible values is a pain
- I want to use cross features and I have too many of them
- I want to limit the size of my model

Hash functions

a hash function:

- Maps complex data to integers
- Deterministic
- But « looks random »

```python
print(hash("a"))
print(hash("b"))
print(hash("hello"))
print(hash("this is a longer string"))
complex_object = ("some_structured_data", 123, 45)
print(hash( complex_object ))
print(hash("a"))
```

```
4622726560737668235
-6279647101590938825
2716929806936918215
-6349598305679374822
6867566546198076494
4622726560737668235
```

Hash(“a”)
Deterministic!

- Usages:
  - Pseudo random numbers
  - Partitioning data
  - Internal implementation of « dictionary »
  - *... surprisingly useful!*

```python
# Processing ~= 1% of items:
for item in myList:
    if hash(item) % 100 == 0:
        process(item)
# If the same item appears twice, it will be either processed twice, or not at all.
# If we run again, the same items will be selected, even if list is shuffled
```

The Hashing Trick

Building the index *without* looking at the data

- *Hash everything!*
- *Write "1" at the indexes of the hashes*

Chosen size d of the output vector

```
hash( ("type", "book")) %1000000
```

```
966886
```

"type" is "book",
"seller" is "1234"

Raw input

```
hash( ("seller", "1234")) %1000000
```

```
175421
```

```
hash( (("seller", "1234"), ("type", "book"))) %1000000
```

```
548128
```

Hash of the "crossfeature"

1 at index Hash(seller, 1234)

0
0
…
0
1
0
…
0
1
0
…
0
1
0
…

$X \in \mathbb{R}^d$ with
d:= 1000000
(*your* choice)

Collisions?

```
1  hash(( "seller", "626" ))%1000000
```

executed in 7ms, finished 15:19:59 2021-03-09

626786

```
1  hash(( "type", "1674" ))%1000000
```

executed in 4ms, finished 15:19:59 2021-03-09

626786

0
0
…
0
1
0
…

A "1" at index 626786 may either mean:

- ‘type is 1674’
- or ‘seller is 62’
- Or something else ? (more hashes colliding)

- When learning a model from hashed inputs (eg a linear model X.w), collisions constrain the learned model to set the **same weights to colliding features**.

- Here, same weight for «seller is 626» and «type is 1674»

Hashing Trick: Why?

Warning: When deploying your model to another machine, **make sure that the "hash" implementation is the same on every machine**. *For example, python internal "hash" function changes when restarting the kernel.*

- Simple
  - No need to precompute indexes
  - No need to care *too much* about « new » modalities
- Fixed size model
  - **Known memory footprint** → no surprise crash
- Collisions are not so bad
  - Collision between rare modalities is OK
  - Collisions between frequent modalities is unlikely
  - Also, redondant features may help the model to recover from collisions (eg lots of « crossfeatures »)
- Common use cases
  - (Generalized) linear model with many crossfeatures
  - Input layer of neural networks

Avoiding Collisions?

```
hash(( "seller", "626" ))%100000000
```

12626786

```
hash(( "type", "1674" ))%100000000
```

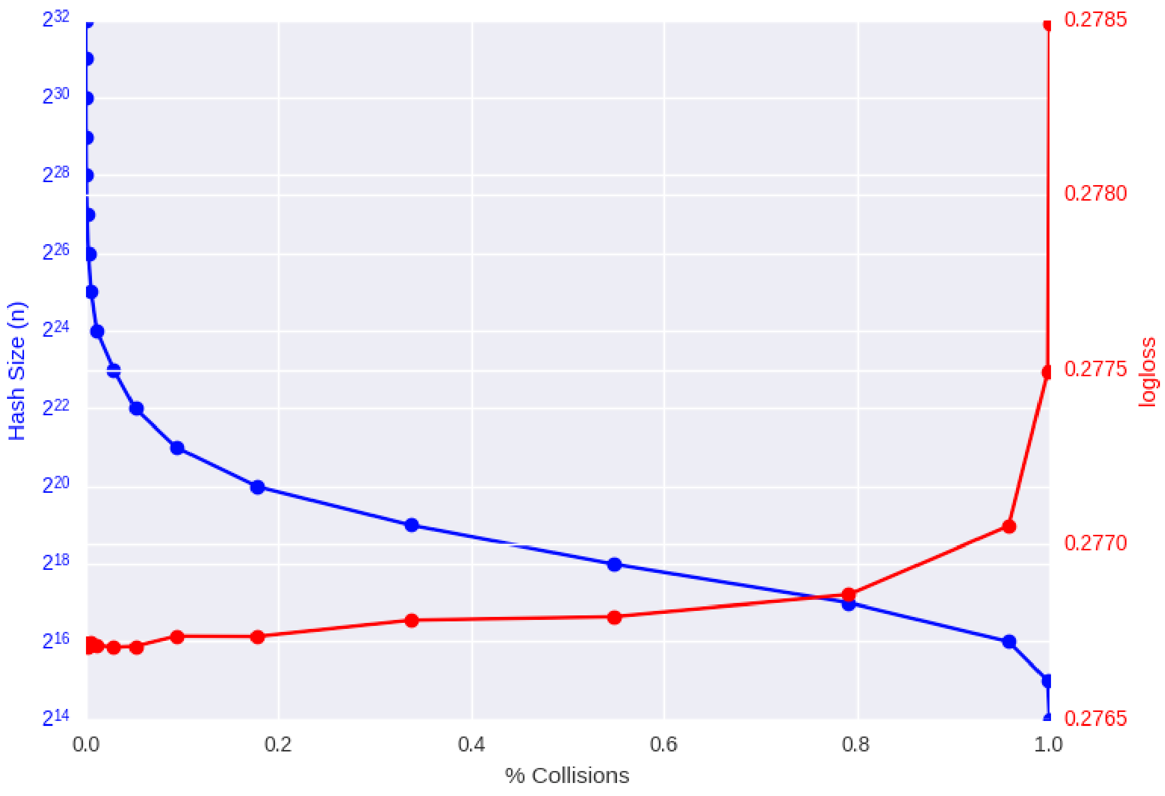
86626786

- Increase n ➔ less collisions.
- Difficult to fully avoid: « birthday paradox »
  - K disinct hashed modalities => $O(K^2)$ pairs of modalities
  - Each single pair has a probability 1/n of colliding together
  - Overall, $O(K^2/n)$ expected collisions
- But increasing n increases the size of the model and the learning time
- Test for best tradeoff!!

- Other options:
  - Keep index of «common» modalities and hash only « rare » modalities
  - Several hashes per modality (more collisions, but more redundancy to recover from those collisions)
- But try « Ostrich » algorithm first!!

The hashing trick

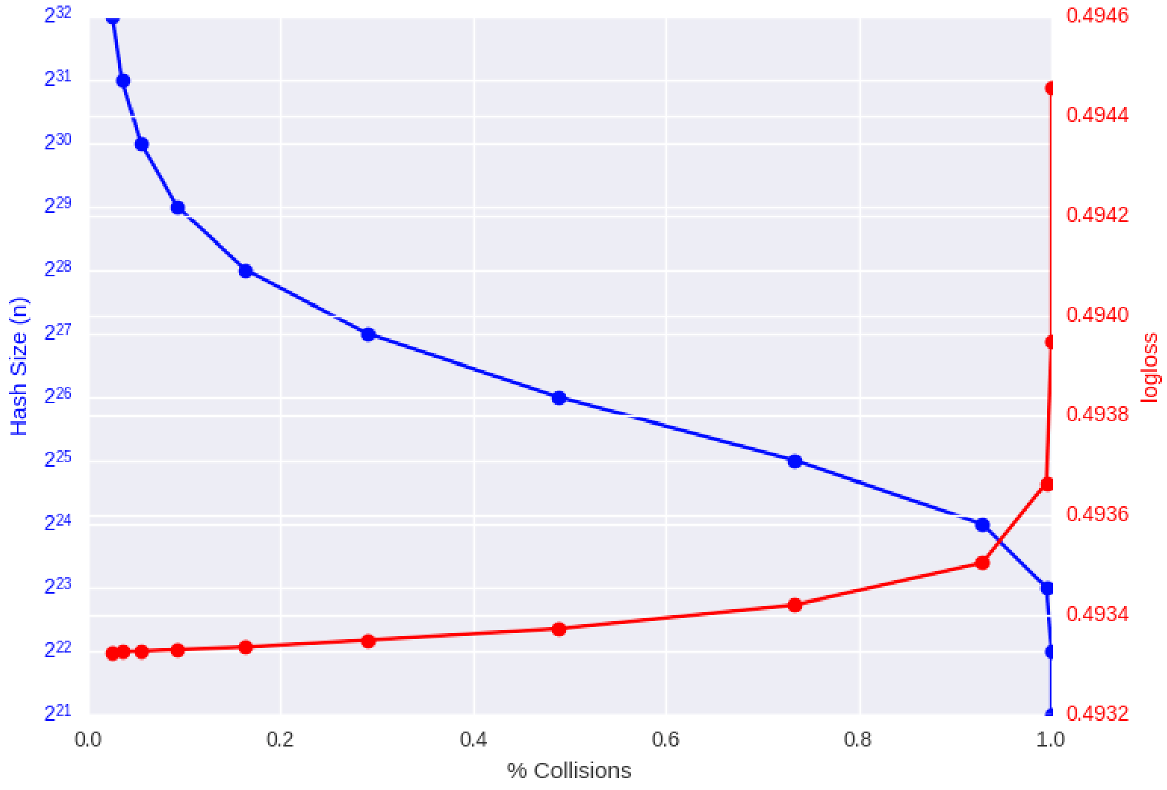
https://booking.ai/dont-be-tricked-by-the-hashing-trick-192a6aae3087



Booking dataset

The hashing trick

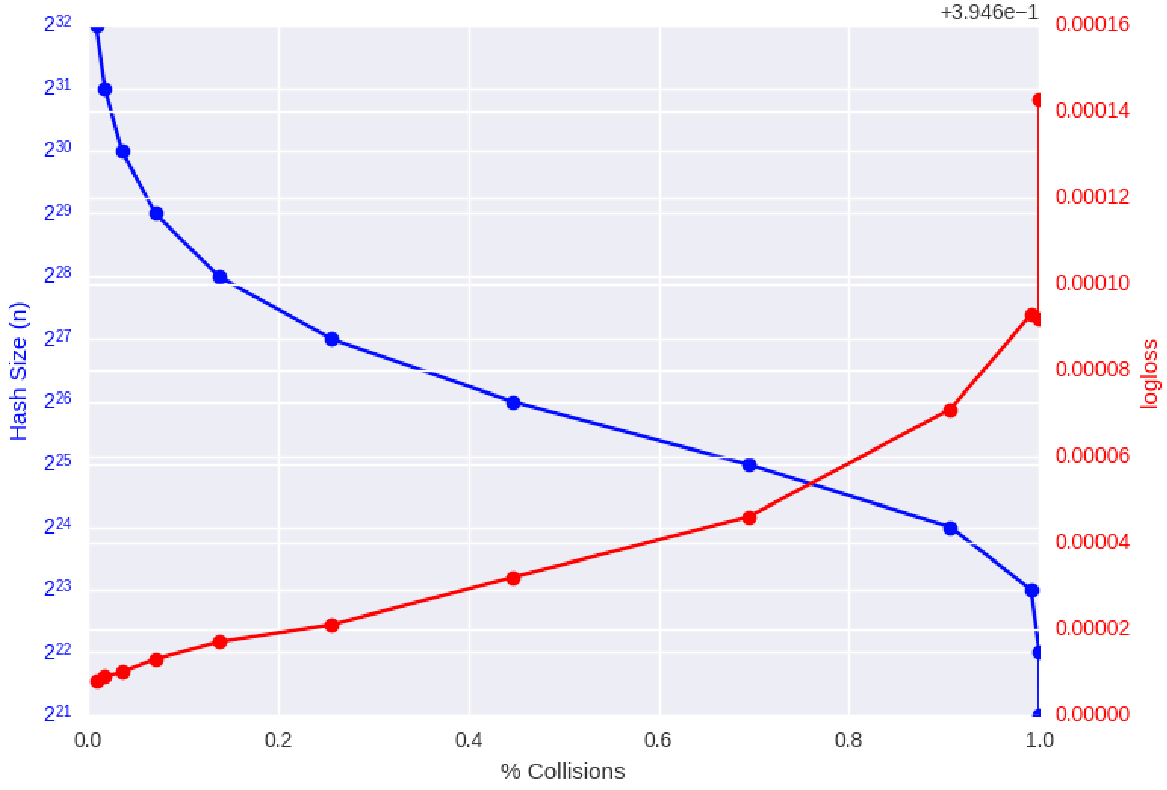
https://booking.ai/dont-be-tricked-by-the-hashing-trick-192a6aae3087



Criteo dataset

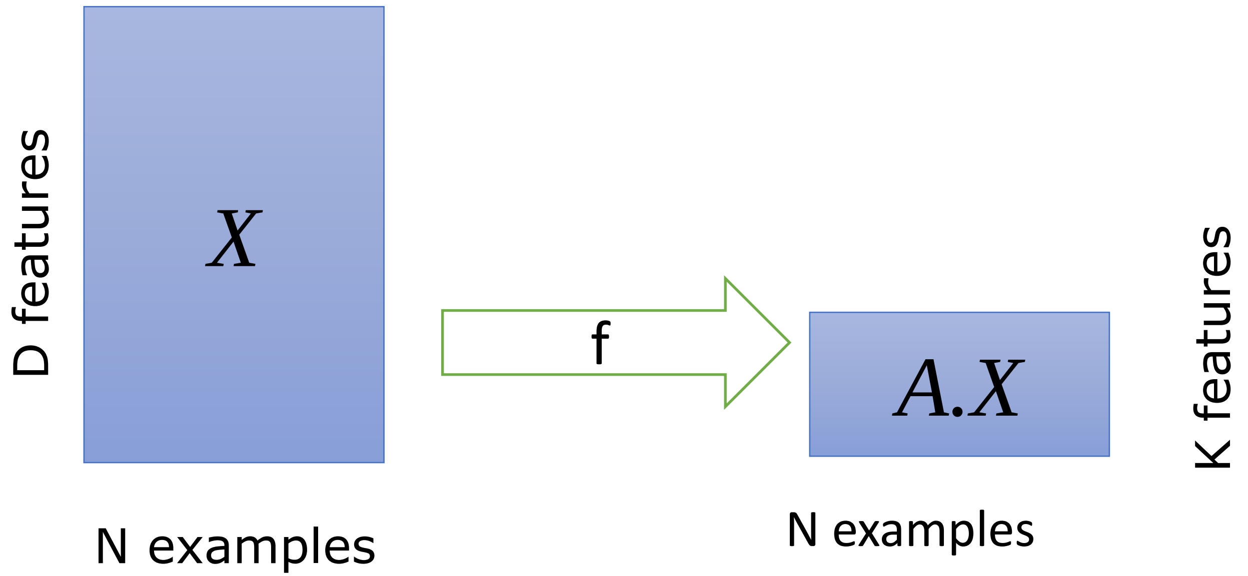
The hashing trick

https://booking.ai/dont-be-tricked-by-the-hashing-trick-192a6aae3087



Avazu dataset

Dimensionality reduction



Reducing the dimensions of input data before applying ML algo.

Ideally:

- K << D
- (approximately) preserving distances between examples:

  $d(X_i, X_j) \approx d(f(X_i), f(X_j))$
- f linear. $f: X_i \to A.X_i$. (easy to compute)

Classical method: PCA

- Rows of A := first K eigenvectors of $X \cdot X^T$
- Good preservation of distances ✓
- $O(ND^2)$ or $O(N^2D)$ ✗
- Data dependent

  (can we avoid it ?)

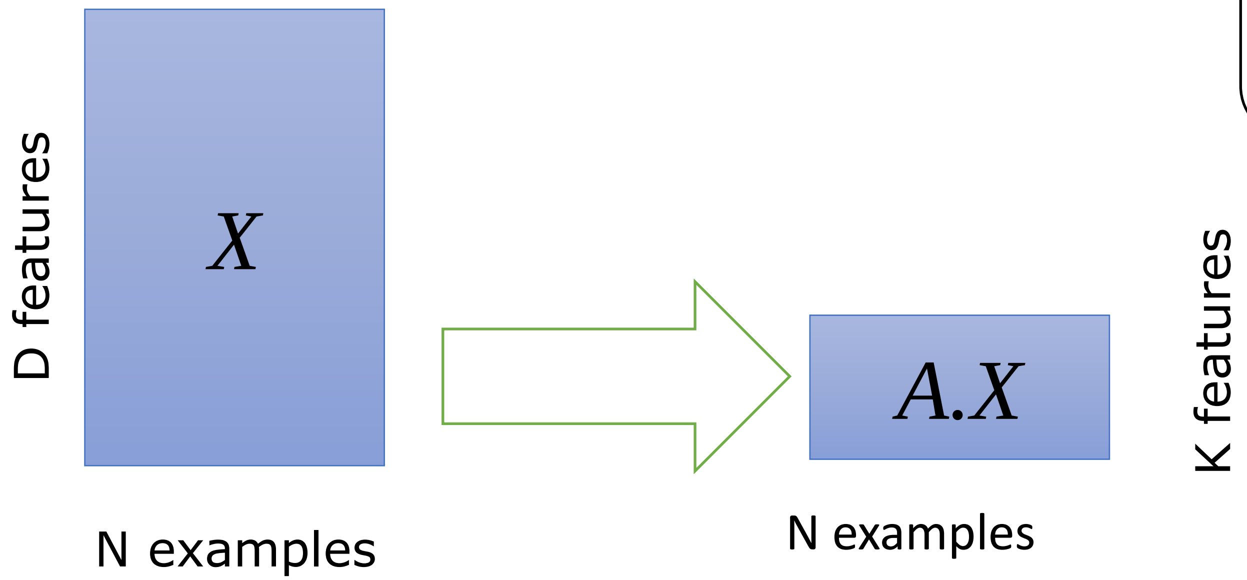
Random projections

This is hashing trick!! (if X is binary)

- Still requires large K (> 10^6) to work well

What if…

- I Choose $A_{ij} \in \{0;1\}$
- Independent columns, with exactly one 1 per column, placed uniformly at random ?



Instead, “Gaussian” random projections:

A_{ik} iid ~ Gaussian(0,1) * $1/\sqrt{K}$

- Is it likely to get $d(A.X_i , A.X_j) \approx d(X_i , X_j)$?
- For all pairs i,j simultaneously ??

- A is a matrix of size (K,D)
- Crazy idea: **Choosing A *randomly***!?

Random projections

Checking that "$|A.z| \approx |z|$" *(with high proba)*
Set $z := X_i - X_j$ to get
$d(X_i, X_j) \approx d(A.X_i, A X_j)$

Let $z \in \mathbb{R}^D$ and $|\ |$ the Euclidian norm.
Let's compare $|A.z|^2$ and $|z^2|$.
$|A.z|^2 = \sum_k (A_k . z)^2$

A sum of independent Gaussian variables is Gaussian:
$A_k .z = \text{sum } A_{k,j} * z_j$.
$A_{k,j} * z_j$ are independent and $\sim N(0, |z_j|/\sqrt{K})$

Lemma:

- $A_k .z$ are iid
- $A_k .z \sim N(0,1) * |z| / \sqrt{K}$

Thus:

- $E((A_k .z)^2) = |z|^2 / K$
- $E(|A.z|^2) = |z|^2$
- $\mathrm{Var}(|A.z|^2) = 2 |z|^4 / K$

(Variance of a chi²)
O(1/K): With large K, low variance.
Large probability of getting $|A.z| \approx |z|$

Random projections

Formalizing
"$d(X_i, X_j) \approx d(A.X_i, AX_j)$"

Let $\varepsilon > 0$. We would like, for all pair (i,j):

$(1-\varepsilon) . d(X_i , X_j) < d(A.X_i , A.X_j) < (1+\varepsilon).d(X_i , X_j)$

No 100% garanty that inequalities are true.
Instead, we want them to hold (for all i,j)
with a probability $> 1 - \delta$

Let $\delta > 0$ « *accepted Probability of failing* »

Theorem (Johnson–Lindenstrauss):
if $K > O(\log(n/\delta) / \varepsilon^2)$, the probability that all inequalities above hold is higher than $1- \delta$

Proof idea: $O(n^2)$ pairs → Let us get Proba of failing for one pair smaller than δ/n^2
Use concentration of $|Az|^2$ around its mean.

- Does not depend on D!
- O() constant is around 10

- Example:
  N=1B, $\varepsilon = 0.1$ ➔ $K \approx 20000$
- In practice, K significantly smaller is often good enough.

Random projections

Typical values

$n = O(1B)$; $d = O(1M)$ $\Rightarrow$ $k = O(10K)$

Memory: storing A: $O(DK) \approx 40GB$

- Maybe ok?
- Use only a few bits per entry?
- Store only the seeds?!

Computing A.X: $O(ndk)$ ✗

- But X is usually sparse ✓

Many variants

- Replace A_{ij} ~ Gaussian by A_{ij} in {+/-1}
  - 1 bit per entry
- "Fast Johnson-Lindenstrauss transform"
  - A is the product of sparse random matrices and a structured matrix allowing fast computation
  - $O(n.d.\log(d) + nk^2)$ ✓