Title: Feature hashing

URL: https://en.wikipedia.org/wiki/Feature_hashing

PageID: 36126852

Categories: Category:Hashing, Category:Machine learning

In machine learning , feature hashing , also known as the hashing trick (by analogy to the kernel trick), is a fast and space-efficient way of vectorizing features , i.e. turning arbitrary features into indices in a vector or matrix. It works by applying a hash function to the features and using their hash values as indices directly (after a modulo operation), rather than looking the indices up in an associative array . In addition to its use for encoding non-numeric values, feature hashing can also be used for dimensionality reduction .

This trick is often attributed to Weinberger et al. (2009), but there exists a much earlier description of this method published by John Moody in 1989.

Motivation

Motivating example

In a typical document classification task, the input to the machine learning algorithm (both during learning and classification) is free text. From this, a bag of words (BOW) representation is constructed: the individual tokens are extracted and counted, and each distinct token in the training set defines a feature (independent variable) of each of the documents in both the training and test sets.

Machine learning algorithms, however, are typically defined in terms of numerical vectors. Therefore, the bags of words for a set of documents is regarded as a term-document matrix where each row is a single document, and each column is a single feature/word; the entry i , j in such a matrix captures the frequency (or weight) of the j 'th term of the vocabulary in document i . (An alternative convention swaps the rows and columns of the matrix, but this difference is immaterial.)

Typically, these vectors are extremely sparse —according to Zipf's law .

The common approach is to construct, at learning time or prior to that, a dictionary representation of the vocabulary of the training set, and use that to map words to indices. Hash tables and tries are common candidates for dictionary implementation. E.g., the three documents

John likes to watch movies.

Mary likes movies too.

John also likes football.

can be converted, using the dictionary

to the term-document matrix

(Punctuation was removed, as is usual in document classification and clustering.)

The problem with this process is that such dictionaries take up a large amount of storage space and grow in size as the training set grows. On the contrary, if the vocabulary is kept fixed and not increased with a growing training set, an adversary may try to invent new words or misspellings that are not in the stored vocabulary so as to circumvent a machine learned filter. To address this challenge, Yahoo! Research attempted to use feature hashing for their spam filters.

Note that the hashing trick isn't limited to text classification and similar tasks at the document level, but can be applied to any problem that involves large (perhaps unbounded) numbers of features.

Mathematical motivation

Mathematically, a token is an element t {\displaystyle t} in a finite (or countably infinite) set T {\displaystyle T} . Suppose we only need to process a finite corpus, then we can put all tokens appearing in the corpus into T {\displaystyle T} , meaning that T {\displaystyle T} is finite. However, suppose we want to process all possible words made of the English letters, then T {\displaystyle T} is countably infinite.

Most neural networks can only operate on real vector inputs, so we must construct a "dictionary" function φ : T → R n {\displaystyle \phi :T\to \mathbb {R} ^{n}} .

When T {\displaystyle T} is finite, of size | T | = m ≤ n {\displaystyle |T|=m\leq n} , then we can use one-hot encoding to map it into R n {\displaystyle \mathbb {R} ^{n}} . First, arbitrarily enumerate T = { t 1 , t 2 , . . , t m } {\displaystyle T=\{t_{1},t_{2},..,t_{m}\}} , then define φ (t i) = e i {\displaystyle \phi (t_{i})=e_{i}} . In other words, we assign a unique index i {\displaystyle i} to each token, then map the token with index i {\displaystyle i} to the unit basis vector e i {\displaystyle e_{i}} .

One-hot encoding is easy to interpret, but it requires one to maintain the arbitrary enumeration of T {\displaystyle T} . Given a token t ∈ T {\displaystyle t\in T} , to compute φ (t) {\displaystyle \phi (t)} , we must find out the index i {\displaystyle i} of the token t {\displaystyle t} . Thus, to implement φ {\displaystyle \phi } efficiently, we need a fast-to-compute bijection h : T → { 1 , . . . , m } {\displaystyle h:T\to \{1,...,m\}} , then we have φ (t) = e h (t) {\displaystyle \phi (t)=e_{h(t)}} .

In fact, we can relax the requirement slightly: It suffices to have a fast-to-compute injection h : T → { 1 , . . . , n } {\displaystyle h:T\to \{1,...,n\}} , then use φ (t) = e h (t) {\displaystyle \phi (t)=e_{h(t)}} .

In practice, there is no simple way to construct an efficient injection h : T → { 1 , . . . , n } {\displaystyle h:T\to \{1,...,n\}} . However, we do not need a strict injection, but only an approximate injection. That is, when t ≠ t ′ {\displaystyle t\neq t'} , we should probably have h (t) ≠ h (t ′) {\displaystyle h(t)\neq h(t')} , so that probably φ (t) ≠ φ (t ′) {\displaystyle \phi (t)\neq \phi (t')} .

At this point, we have just specified that h {\displaystyle h} should be a hashing function. Thus we reach the idea of feature hashing.

Algorithms

Feature hashing (Weinberger et al. 2009)

The basic feature hashing algorithm presented in (Weinberger et al. 2009) is defined as follows.

First, one specifies two hash functions: the kernel hash h : T → { 1 , 2 , . . . , n } {\displaystyle h:T\to \{1,2,...,n\}} , and the sign hash ζ : T → { − 1 , + 1 } {\displaystyle \zeta :T\to \{-1,+1\}} . Next, one defines the feature hashing function: φ : T → R n , φ (t) = ζ (t) e h (t) {\displaystyle \phi :T\to \mathbb {R} ^{n},\quad \phi (t)=\zeta (t)e_{h(t)}} Finally, extend this feature hashing function to strings of tokens by φ : T ∗ → R n , φ (t 1 , . . . , t k) = ∑ j = 1 k φ (t j) {\displaystyle \phi :T^{*}\to \mathbb {R} ^{n},\quad \phi (t_{1},...,t_{k})=\sum _{j=1}^{k}\phi (t_{j})} where T ∗ {\displaystyle T^{*}} is the set of all finite strings consisting of tokens in T {\displaystyle T} .

Equivalently, φ (t 1 , . . . , t k) = ∑ j = 1 k ζ (t j) e h (t j) = ∑ i = 1 n (∑ j : h (t j) = i ζ (t j)) e i {\displaystyle \phi (t_{1},...,t_{k})=\sum _{j=1}^{k}\zeta (t_{j})e_{h(t_{j})}=\sum _{i=1}^{n}\left(\sum _{j:h(t_{j})=i}\zeta (t_{j})\right)e_{i}}

Geometric properties

We want to say something about the geometric property of φ {\displaystyle \phi } , but T {\displaystyle T} , by itself, is just a set of tokens, we cannot impose a geometric structure on it except the discrete topology, which is generated by the discrete metric . To make it nicer, we lift it to T → R T {\displaystyle T\to \mathbb {R} ^{T}} , and lift φ {\displaystyle \phi } from φ : T → R n {\displaystyle \phi :T\to \mathbb {R} ^{n}} to φ : R T → R n {\displaystyle \phi :\mathbb {R} ^{T}\to \mathbb {R} ^{n}} by linear extension: φ ((x t) t ∈ T) = ∑ t ∈ T x t ζ (t) e h (t) = ∑ i = 1 n (∑ t : h (t) = i x t ζ (t)) e i {\displaystyle \phi ((x_{t})_{t\in T})=\sum _{t\in T}x_{t}\zeta (t)e_{h(t)}=\sum _{i=1}^{n}\left(\sum _{t:h(t)=i}x_{t}\zeta (t)\right)e_{i}} There is an infinite sum there, which must be handled at once. There are essentially only two ways to handle infinities. One may impose a metric, then take its completion , to allow well-behaved infinite sums, or one may demand that nothing is actually infinite, only potentially so . Here, we go for the potential-infinity way, by restricting R T

{\displaystyle \mathbb {R} ^{T}} to contain only vectors with finite support : ∀ (x t) t ∈ T ∈ R T {\displaystyle \forall (x_{t})_{t\in T}\in \mathbb {R} ^{T}} , only finitely many entries of (x t) t ∈ T {\displaystyle (x_{t})_{t\in T}} are nonzero.

Define an inner product on R T {\displaystyle \mathbb {R} ^{T}} in the obvious way: ■ e t , e t ′ ■ = { 1 , if t = t ′ , 0 , else. ■ x , x ′ ■ = ∑ t , t ′ ∈ T x t x t ′ ■ e t , e t ′ ■ {\displaystyle \langle e_{t},e_{t'}\rangle ={\begin{cases}1,{\text{ if }}t=t',\\0,{\text{ else.}}\end{cases}}\quad \langle x,x'\rangle =\sum _{t,t'\in T}x_{t}x_{t'}\langle e_{t},e_{t'}\rangle } As a side note, if T {\displaystyle T} is infinite, then the inner product space R T {\displaystyle \mathbb {R} ^{T}} is not complete . Taking its completion would get us to a Hilbert space , which allows well-behaved infinite sums.

Now we have an inner product space, with enough structure to describe the geometry of the feature hashing function φ : R T → R n {\displaystyle \phi :\mathbb {R} ^{T}\to \mathbb {R} ^{n}} .

First, we can see why h {\displaystyle h} is called a " kernel hash ": it allows us to define a kernel K : T × T → R {\displaystyle K:T\times T\to \mathbb {R} } by K (t , t ′) = ■ e h (t) , e h (t ′) ■ {\displaystyle K(t,t')=\langle e_{h(t)},e_{h(t')}\rangle } In the language of the "kernel trick", K {\displaystyle K} is the kernel generated by the "feature map" ϕ : T → R n , ϕ (t) = e h (t) {\displaystyle \varphi :T\to \mathbb {R} ^{n},\quad \varphi (t)=e_{h(t)}} Note that this is not the feature map we were using, which is φ (t) = ζ (t) e h (t) {\displaystyle \phi (t)=\zeta (t)e_{h(t)}} . In fact, we have been using another kernel K ζ : T × T → R {\displaystyle K_{\zeta }:T\times T\to \mathbb {R} } , defined by K ζ (t , t ′) = ■ ζ (t) e h (t) , ζ (t ′) e h (t ′) ■ {\displaystyle K_{\zeta }(t,t')=\langle \zeta (t)e_{h(t)},\zeta (t')e_{h(t')}\rangle } The benefit of augmenting the kernel hash h {\displaystyle h} with the binary hash ζ {\displaystyle \zeta } is the following theorem, which states that φ {\displaystyle \phi } is an isometry "on average".

Theorem (intuitively stated) — If the binary hash ζ {\displaystyle \zeta } is unbiased (meaning that it takes value − 1 , + 1 {\displaystyle -1,+1} with equal probability), then φ : R T → R n {\displaystyle \phi :\mathbb {R} ^{T}\to \mathbb {R} ^{n}} is an isometry in expectation: E [■ φ (x) , φ (x ′) ■] = ■ x , x ′ ■ . {\displaystyle \mathbb {E} [\langle \phi (x),\phi (x')\rangle]=\langle x,x'\rangle .}

By linearity of expectation, E [■ φ (x) , φ (x ′) ■] = ∑ t , t ′ ∈ T (x t x t ′ ′) ⋅ E [ζ (t) ζ (t ′)] ⋅ ■ e h (t) , e h (t ′) ■ {\displaystyle \mathbb {E} [\langle \phi (x),\phi (x')\rangle]=\sum _{t,t'\in T}(x_{t}x'_{t'})\cdot \mathbb {E} [\zeta (t)\zeta (t')]\cdot \langle e_{h(t)},e_{h(t')}\rangle } Now, E [ζ (t) ζ (t ′)] = { 1 if t = t ′ 0 if t ≠ t ′ {\displaystyle \mathbb {E} [\zeta (t)\zeta (t')]={\begin{cases}1\quad {\text{ if }}t=t'\\0\quad {\text{ if }}t\neq t'\\\end{cases}}} , since we assumed ζ {\displaystyle \zeta } is unbiased. So we continue E [■ φ (x) , φ (x ′) ■] = ∑ t ∈ T (x t x t ′) ■ e h (t) , e h (t) ■ = ■ x , x ′ ■ {\displaystyle \mathbb {E} [\langle \phi (x),\phi (x')\rangle]=\sum _{t\in T}(x_{t}x'_{t})\langle e_{h(t)},e_{h(t)}\rangle =\langle x,x'\rangle }

The above statement and proof interprets the binary hash function ζ {\displaystyle \zeta } not as a deterministic function of type T → { − 1 , + 1 } {\displaystyle T\to \{-1,+1\}} , but as a random binary vector { − 1 , + 1 } T {\displaystyle \{-1,+1\}^{T}} with unbiased entries, meaning that P r (ζ (t) = + 1) = P r (ζ (t) = − 1) = 1 2 {\displaystyle Pr(\zeta (t)=+1)=Pr(\zeta (t)=-1)={\frac {1}{2}}} for any t ∈ T {\displaystyle t\in T} .

This is a good intuitive picture, though not rigorous. For a rigorous statement and proof, see

Pseudocode implementation

Instead of maintaining a dictionary, a feature vectorizer that uses the hashing trick can build a vector of a pre-defined length by applying a hash function h to the features (e.g., words), then using the hash values directly as feature indices and updating the resulting vector at those indices. Here, we assume that feature actually means feature vector.

Thus, if our feature vector is ["cat","dog","cat"] and hash function is h (x f) = 1 {\displaystyle h(x_{f})=1} if x f {\displaystyle x_{f}} is "cat" and 2 {\displaystyle 2} if x f {\displaystyle x_{f}} is "dog". Let us take the output feature vector dimension (N) to be 4. Then output x will be [0,2,1,0].

It has been suggested that a second, single-bit output hash function ξ be used to determine the sign of the update value, to counter the effect of hash collisions . If such a hash function is used, the algorithm becomes

The above pseudocode actually converts each sample into a vector. An optimized version would instead only generate a stream of (h , ζ) {\displaystyle (h,\zeta)} pairs and let the learning and prediction algorithms consume such streams; a linear model can then be implemented as a single hash table representing the coefficient vector.

Extensions and variations

Learned feature hashing

Feature hashing generally suffers from hash collision, which means that there exist pairs of different tokens with the same hash: t ≠ t ′ , φ (t) = φ (t ′) = v {\displaystyle t\neq t',\phi (t)=\phi (t')=v} . A machine learning model trained on feature-hashed words would then have difficulty distinguishing t {\displaystyle t} and t ′ {\displaystyle t'} , essentially because v {\displaystyle v} is polysemic .

If t ′ {\displaystyle t'} is rare, then performance degradation is small, as the model could always just ignore the rare case, and pretend all v {\displaystyle v} means t {\displaystyle t} . However, if both are common, then the degradation can be serious.

To handle this, one can train supervised hashing functions that avoids mapping common tokens to the same feature vectors.

Applications and practical performance

Ganchev and Dredze showed that in text classification applications with random hash functions and several tens of thousands of columns in the output vectors, feature hashing need not have an adverse effect on classification performance, even without the signed hash function.

Weinberger et al. (2009) applied their version of feature hashing to multi-task learning , and in particular, spam filtering , where the input features are pairs (user, feature) so that a single parameter vector captured per-user spam filters as well as a global filter for several hundred thousand users, and found that the accuracy of the filter went up.

Chen et al. (2015) combined the idea of feature hashing and sparse matrix to construct "virtual matrices": large matrices with small storage requirements. The idea is to treat a matrix M ∈ R n × n {\displaystyle M\in \mathbb {R} ^{n\times n}} as a dictionary, with keys in n × n {\displaystyle n\times n} , and values in R {\displaystyle \mathbb {R} } . Then, as usual in hashed dictionaries, one can use a hash function h : N × N → m {\displaystyle h:\mathbb {N} \times \mathbb {N} \to m} , and thus represent a matrix as a vector in R m {\displaystyle \mathbb {R} ^{m}} , no matter how big n {\displaystyle n} is. With virtual matrices, they constructed HashedNets , which are large neural networks taking only small amounts of storage.

Implementations

Implementations of the hashing trick are present in:

Apache Mahout

Gensim

scikit-learn

sofia-ml

Vowpal Wabbit

Apache Spark

R

TensorFlow

Dask-ML

See also

Bloom filter – Data structure for approximate set membership

Count–min sketch – Probabilistic data structure in computer science

Heaps' law – Heuristic for distinct words in a document

Locality-sensitive hashing – Algorithmic technique using hashing

MinHash – Data mining technique

References

External links

Hashing Representations for Machine Learning on John Langford's website

What is the "hashing trick"? - MetaOptimize Q+A