Machine Translation & Document Search

Locality Sensitive Hashing (LSH)

Learning Objectives

1. **Machine Translation Fundamentals**

   Understand translation as vector transformation

2. **Document Search Techniques**

   Master K-nearest neighbors and similarity search

3. **Locality Sensitive Hashing**

   Learn LSH for efficient approximate NN search

4. **Hash Functions & Hyperplanes**

   Random hyperplane hashing for cosine similarity

5. **Practical Implementation**

   Build document search with TF-IDF and LSH

KEY TOPICS

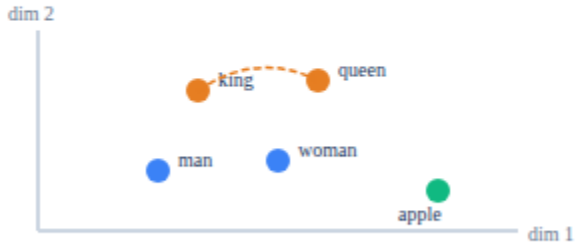
Word Embeddings | KNN Search | Hash Tables | LSH

Recap: Vector Space Models

Session 03 Key Concepts

- **TF-IDF Representation**
  Words as weighted vectors based on term frequency and document frequency
- **Cosine Similarity**
  Measure angle between vectors to find semantic similarity
- **Dimensionality Reduction (PCA)**
  Reduce dimensions while preserving variance and structure
- **Word Embeddings**
  Dense vector representations capturing semantic meaning

Vector Space Representation



TODAY'S EXTENSION

Building on vector representations to enable:

Translation
Cross-lingual mapping

Fast Search
Efficient similarity

PART I

Machine Translation

Translation as Vector Space Transformation

• Translation History • Cross-lingual Embeddings • Transformation Matrix • Learning to Translate

History of Machine Translation

1950s

Rule-Based MT

Georgetown experiment, linguistic rules, bilingual dictionaries

1990s

Statistical MT

IBM Models, phrase-based translation, parallel corpora

2014

Neural MT (Seq2Seq)

Encoder-decoder, RNN/LSTM, end-to-end learning

2017+

Transformer MT

Attention mechanism, BERT, GPT, state-of-the-art

Rule-Based Approach

Linguistic experts define grammar rules and translation patterns manually. Limited scalability.

Statistical Approach

Learn translation probabilities from parallel text corpora. P(target|source).

Vector Space Approach

Transform word embeddings between languages using learned linear mappings. Today's focus!

Translation as Vector Transformation

Core Insight

Word embeddings in different languages share similar **geometric structure**. Words with similar meanings occupy similar relative positions in their respective embedding spaces.

The Key Idea

Find a **transformation matrix R** that maps English embeddings to French embeddings:

$$X_{\text{English}} \cdot R \approx Y_{\text{French}}$$

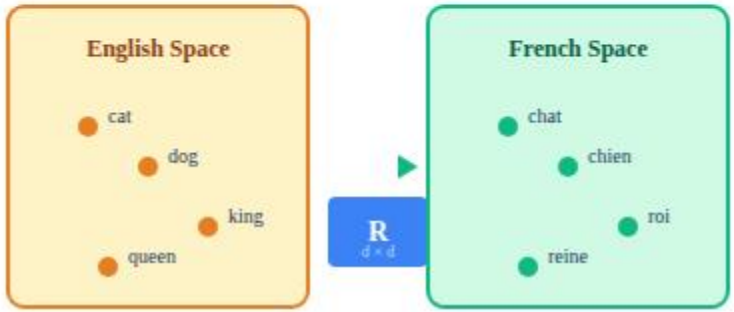
X: English embedding matrix

R: Transformation matrix

Y: French embedding matrix

Why it works: Both languages encode similar concepts (king, queen, man, woman) with similar relationships in their embedding spaces.

Vector Space Transformation



Key Insight: Similar Structure!

- king - queen ≈ roi - reine (gender relationship)
- cat - dog ≈ chat - chien (animal relationship)

Word Embeddings Across Languages

Monolingual Embeddings

Trained separately on each language corpus (Word2Vec, GloVe, fastText):

English
Word2Vec_{en}

French
Word2Vec_{fr}

The Challenge

Embeddings are trained independently, so they live in **different vector spaces**:

⚠️ You cannot directly compare English "cat" with French "chat" because their dimensions have different meanings!

The Solution: Alignment

Learn a transformation matrix **R** using a small bilingual dictionary to align the two spaces.

Before vs After Alignment

The Transformation Matrix R

Problem Setup

Given:

- $\mathbf{X} \in \mathbb{R}^{m \times d}$: English word embeddings
- $\mathbf{Y} \in \mathbb{R}^{m \times d}$: French word embeddings
- m = number of word pairs in dictionary
- d = embedding dimension (e.g., 300)

Find:

Matrix $\mathbf{R} \in \mathbb{R}^{d \times d}$ such that $\mathbf{XR} \approx \mathbf{Y}$

Matrix Dimensions

$$\underset{(m \times d)}{\mathbf{X}} \times \underset{(d \times d)}{\mathbf{R}} = \underset{(m \times d)}{\mathbf{Y}}$$

💡 Key Insight

R is a **linear transformation** that rotates and scales the English embedding space to align with the French space.

Matrix Multiplication Visualization

Single Word Translation

$$\mathbf{x}_{cat} \cdot \mathbf{R} = \hat{\mathbf{y}}_{chat} \approx \mathbf{y}_{chat}$$

Find nearest neighbor to $\hat{\mathbf{y}}$ in French vocabulary

Loss Function for Translation

Objective: Minimize Error

We want **XR** to be as close as possible to **Y**. Use the **Frobenius norm** to measure the difference:

$$\text{Loss} = \|XR - Y\|_F^2$$

Frobenius Norm Explained

The Frobenius norm is the "length" of a matrix, computed as:

$$\|A\|_F = \sqrt{(\Sigma_i \Sigma_j a_{ij}^2)}$$

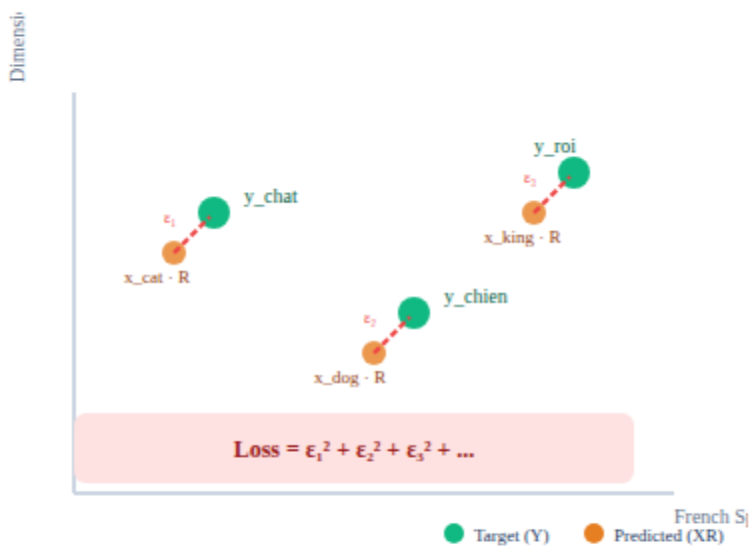
Sum of squared elements, then square root. Similar to Euclidean distance but for matrices.

Expanded Loss Formula

$$L(R) = \Sigma_{i=1}^{n} \|x_i R - y_i\|^2$$

Sum of squared distances for each word pair

Loss Visualization



Gradient Descent for Learning R

Computing the Gradient

To minimize the loss, we need the gradient with respect to R:

Loss function:

$$L = \|XR - Y\|_F^2$$

Gradient:

$$\nabla_R L = 2X^T(XR - Y)$$

Gradient Derivation

Step 1: Expand the squared norm

$$L = \text{tr}((XR - Y)^T(XR - Y))$$

Step 2: Apply matrix calculus

$$\partial L/\partial R = 2X^TXR - 2X^TY$$

Step 3: Simplify

$$\nabla_R L = 2X^T(XR - Y)$$

Update Rule

$$R \leftarrow R - \alpha \cdot \nabla_R L$$

α
Learning rate

$\nabla_R L$
Gradient direction

Training Algorithm

```
# Initialize R randomly
R = np.random.rand(d, d)

# Training loop
for i in range(iterations):
   # Compute gradient
   gradient = 2 * X.T @ (X @ R - Y)
   # Update R
   R = R - alpha * gradient

# Compute loss
loss = np.sum((X @ R - Y)**2)
```

K-Nearest Neighbors for Translation

Translation Process

1. **Get English embedding**
   $x = \text{embedding}["\text{cat}"]$
2. **Transform to French space**
   $\hat{y} = x \cdot R$
3. **Find nearest French word**
   $\text{argmax}_w \text{ cosine}(\hat{y}, \text{french}[w])$

K-Nearest Neighbors

Instead of finding just the closest word, find the **K closest words**:

Why K neighbors?

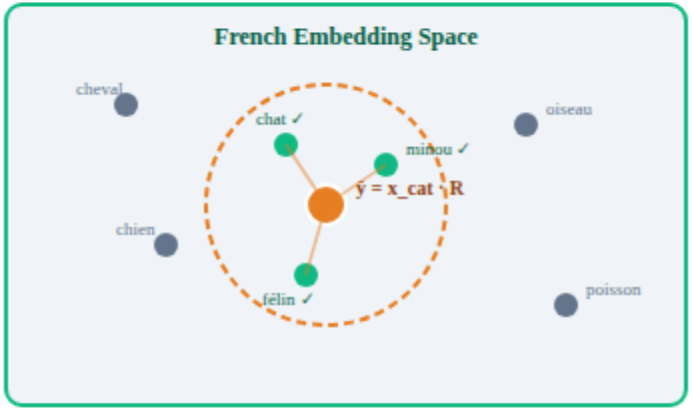
- Handle synonyms and multiple meanings
- Provide translation candidates for ranking
- Increase robustness to noise

⚠ The Problem

Finding K-nearest neighbors in high-dimensional space (d=300) with large vocabulary (V=100,000+) is **computationally expensive**!

Brute force: O(V × d) for each query

KNN Translation Example



K=3 Results: 1. chat (0.95) | 2. minou (0.91) | 3. félin (0.87)

Cosine similarity scores

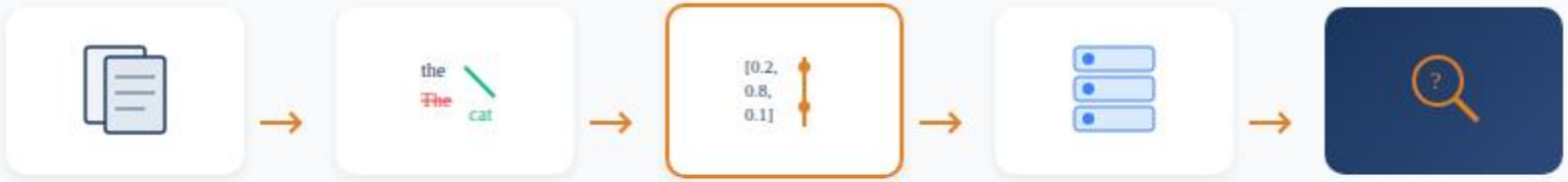
PART II

Document Search

Similarity-Based Information Retrieval

• Document Representation • Similarity Search • Nearest Neighbor Problem • Scalability Challenges

Document Search Pipeline



Vector Representations

TF-IDF
Sparse, high-dim (~10K)

Embeddings
Dense, lower-dim (~300)

Similarity Measures

- **Cosine:** $\cos(\theta) = (A \cdot B)/(\|A\|\|B\|)$
- **Euclidean:** $d = \sqrt{\Sigma(a_i - b_i)^2}$
- **Dot Product:** $A \cdot B = \Sigma a_i b_i$

The Search Problem

Given a query vector **q**, find the K documents with highest similarity:

$$\text{argmax}_k \, \text{sim}(q, d_k)$$

The Scalability Problem

Brute Force Nearest Neighbor

For each query, compute similarity with **every** document:

Time Complexity: **O(N × d)**

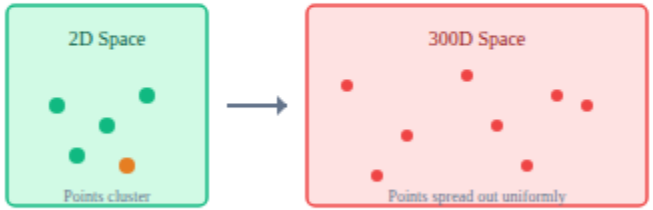
N = number of documents, d = dimension

1M	300	300M
documents	dimensions	operations/query!

Real-World Scale

Platform	Documents	Queries/sec
Google Search	100B+ pages	100K+
Spotify	100M+ songs	10K+
Netflix	15K+ titles	100K+
Wikipedia	60M+ articles	10K+

Curse of Dimensionality



In high dimensions, all points appear nearly equidistant!

Possible Solutions

1. **Tree-based (KD-tree, Ball tree)**
   Works poorly in high dimensions
2. **Locality Sensitive Hashing (LSH)**
   Approximate, sub-linear time! ✓
3. **Quantization (PQ, IVF)**
   Approximate, memory efficient

Approximate Nearest Neighbor (ANN)

Key Insight

We can **sacrifice a little accuracy** for **huge speed gains**!

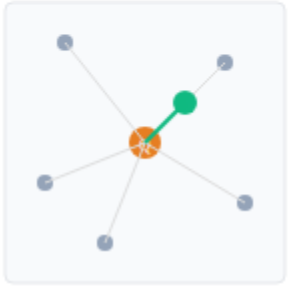
Exact NN	Approximate NN
O(N)	O(log N)
100% accurate	~95% accurate

When is "Good Enough" OK?

- ✓ Search engines: Top 10 results, not THE #1
- ✓ Recommendations: Similar items, not identical
- ✓ Duplicate detection: High recall more important
- ✗ Medical diagnosis: Need exact match

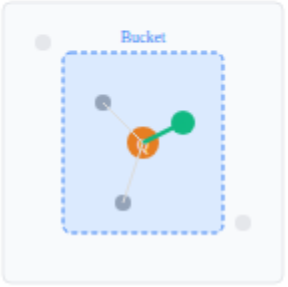
Exact vs Approximate

Exact Search



Check ALL points

Approximate (LSH)



Check only BUCKET

Solution

Locality Sensitive Hashing

Group similar items into "buckets" → Only search within bucket

PART III

Locality Sensitive Hashing

Similar items → Same bucket

⚡ Sub-linear Time

📊 High Dimensional

🎯 Approximate NN

What is Locality Sensitive Hashing?

Definition

Locality Sensitive Hashing is a hashing technique where **similar items** are hashed to the **same bucket** with high probability.

> *"If two points are close in the original space, they should hash to the same bucket with high probability."*

Traditional Hash vs LSH

Traditional Hash

- Similar inputs → Different outputs
- Goal: Minimize collisions
- Used for: Hash tables, security

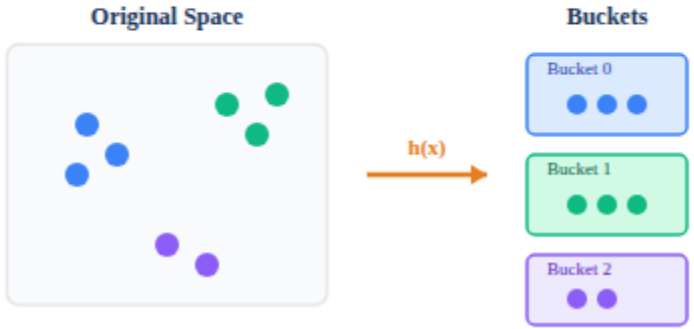
LSH

- Similar inputs → Same output
- Goal: Preserve similarity
- Used for: Nearest neighbor search

LSH Properties

1. **Locality:** Close points hash together
2. **Sensitivity:** Probability depends on distance
3. **Probabilistic:** Approximate, not exact

How LSH Works



Key Insight

Similar vectors → Same bucket → Fast search!

Only search within candidate bucket, not entire dataset

Random Hyperplane Hashing

Core Idea

Use **random hyperplanes** to partition the vector space. Each hyperplane divides space into two regions: positive (+) and negative (−).

$$h(v) = \text{sign}(v \cdot P) = 1 \text{ if } v \cdot P \geq 0 \text{ else } 0$$

P = random normal vector defining the hyperplane

Why It Works

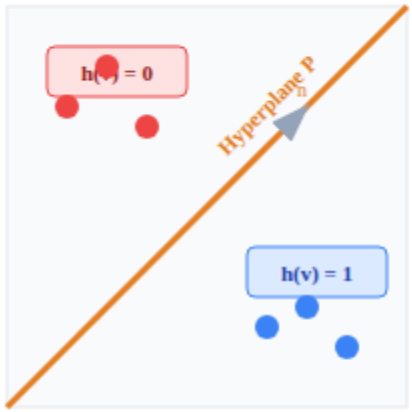
- **Similar vectors** are likely on the same side of the hyperplane
- **Dissimilar vectors** are likely separated by the hyperplane
- **Probability** of same hash ∝ cosine similarity!

Collision Probability

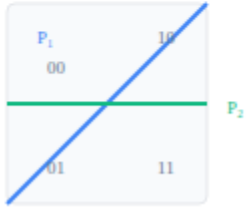
$$P(h(u) = h(v)) = 1 - \theta(u,v)/\pi$$

θ(u,v) = angle between vectors u and v

Hyperplane Partitioning



Multiple Planes



Hash Code

n planes → n bits

2^n buckets total

Example: 10 planes → 1024 buckets

Computing Hash Values

Step 1: Generate Random Planes

```
# Create n random hyperplanes (d dimensions)
n_planes = 10
d = 300 # embedding dimension
planes = np.random.randn(n_planes, d)
```

Step 2: Compute Dot Products

```
# For vector v, compute dot product with each plane
dot_products = np.dot(planes, v)
# Result: array of shape (n_planes,)
```

v = [0.5, -0.2, 0.8, ...] → dot_products = [1.2, -0.5, 0.3, -1.1, ...]

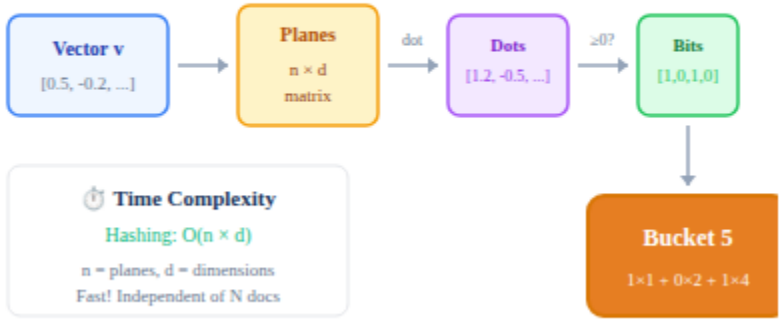
Step 3: Apply Sign Function

```
# Convert to binary hash
hash_bits = (dot_products >= 0).astype(int)
```

Complete Hash Function

```
def hash_vector(v, planes):
    """Compute LSH bucket for vector v"""
    # Dot products: (n_planes,)
    dots = np.dot(planes, v)
    # Binary hash
    h = (dots >= 0).astype(int)
    # Convert to bucket ID
    bucket = sum(h[i] * (2**i)
                 for i in range(len(h)))
    return bucket
```

Visual Example



Building the Hash Table

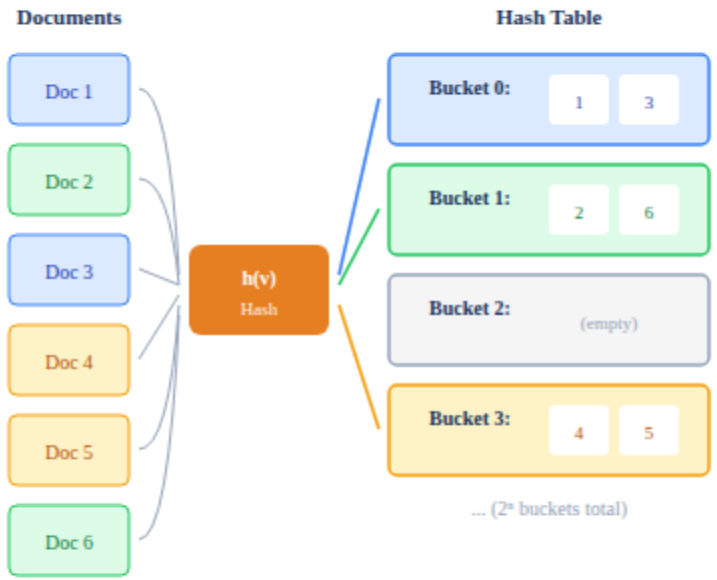
Indexing Phase (One-time)

1. **Generate random planes**
   Create n random vectors for hashing
2. **Hash all documents**
   Compute bucket ID for each document vector
3. **Store in buckets**
   Group documents by their hash values

Python Implementation

```python
def build_lsh_index(docs, planes):
    """Build hash table from documents"""
    hash_table = defaultdict(list)
    for doc_id, doc_vec in docs:
        bucket = hash_vector(doc_vec, planes)
        hash_table[bucket].append(doc_id)
    return hash_table
```

Hash Table Structure



Querying with LSH

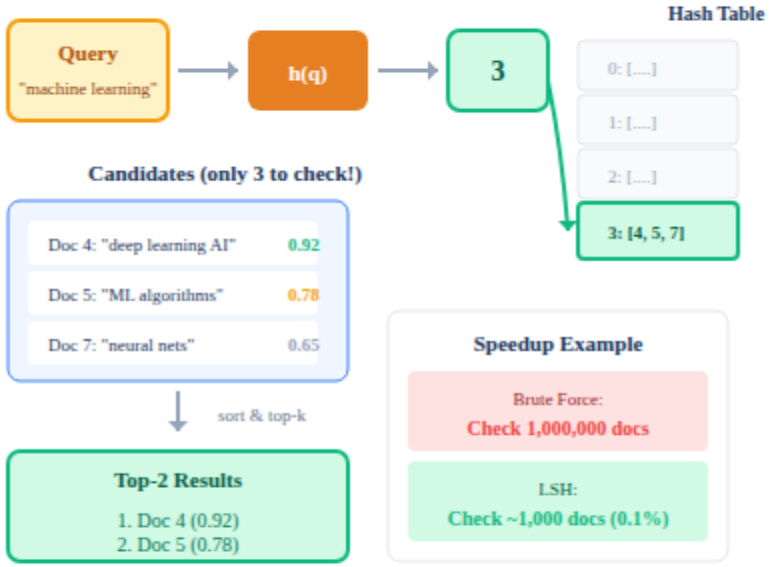
Query Phase (Each Search)

1. **Hash the query**
   bucket_id = h(query_vector)
2. **Retrieve candidates**
   candidates = hash_table[bucket_id]
3. **Compute exact similarity**
   Only for candidates, not all documents!
4. **Return top-K results**
   Sort candidates by similarity score

Query Implementation

```
def query_lsh(q, hash_table, planes, k):
    """Find k-nearest neighbors"""
    # Step 1: Hash query
    bucket = hash_vector(q, planes)
    # Step 2: Get candidates
    cands = hash_table[bucket]
    # Step 3: Compute similarities
    sims = [(cosine(q, c), c) for c in cands]
    # Step 4: Return top-k
    return sorted(sims, reverse=True)[:k]
```

Query Process Visualization



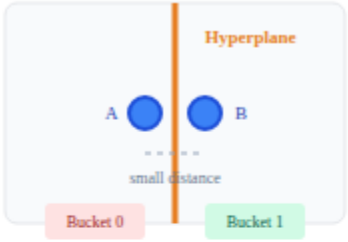
The Problem with Single Hash Table

False Negatives Problem

Similar vectors may be hashed to **different buckets** due to random hyperplane placement. This means we might miss true nearest neighbors!

Example: Two very similar documents might end up in different buckets if a hyperplane happens to fall between them.

Why This Happens



Problem:

A and B are similar but in different buckets!

If query = A,

we miss B as a candidate

Impact on Search Quality

~60%

Typical recall with single table (10 planes)

40%

True neighbors missed!

Solution: Multiple Hash Tables

Use **L different hash tables**, each with its own set of random hyperplanes. This increases the probability that similar items share at least one bucket.

Union of candidates from all tables

→ Higher chance of finding true neighbors!

Probability Analysis

Let **p** = probability that two similar vectors hash to same bucket in one table

Example: $p \approx 0.6$ for cosine similarity > 0.8

Single Table

$P(\text{miss}) = 1-p$

= 40%

L Tables

$P(\text{miss}) = (1-p)^L$

→ decreases exponentially!

Multiple Hash Tables Implementation

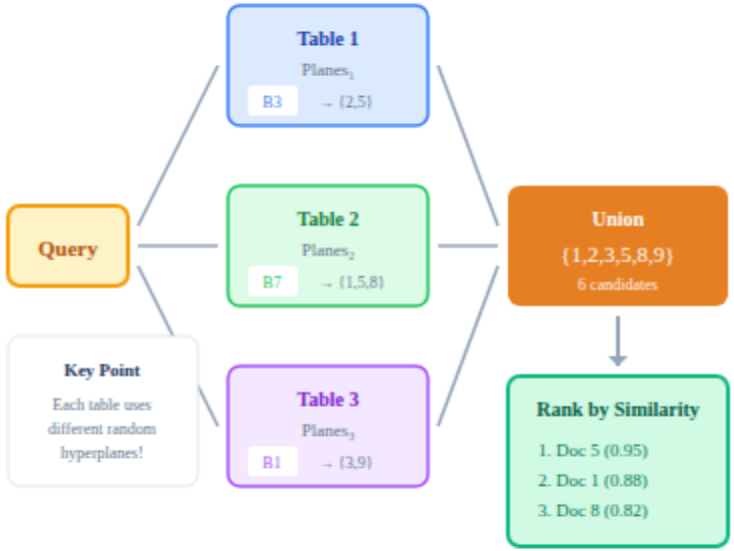
Building L Hash Tables

```
class MultiTableLSH:
  def __init__(self, n_tables, n_planes, dim):
    self.L = n_tables # number of tables
    self.n = n_planes # planes per table
    # L sets of random planes
    self.planes = [
      np.random.randn(n_planes, dim)
      for _ in range(n_tables)
    ]
    # L hash tables
    self.tables = [defaultdict(list)
          for _ in range(n_tables)]
```

Indexing Documents

```
def index(self, doc_id, doc_vec):
  """Add document to all L tables"""
  for i in range(self.L):
    # Hash with table i's planes
    bucket = hash_vector(doc_vec,
             self.planes[i])
    self.tables[i][bucket].append(doc_id)
```

Multi-Table Query Flow



Approximate vs Exact Nearest Neighbors

Exact NN (Brute Force)

- ✓ 100% accuracy
- ✗ O(N × d) per query
- ✗ Not scalable

Approximate NN (LSH)

- ≈ ~95-99% recall
- ✓ Sub-linear time
- ✓ Highly scalable

Key Metrics

Recall@K

Recall = |LSH results ∩ True NN| / K

Fraction of true nearest neighbors found by LSH

QPS (Queries Per Second)

QPS = Number of queries / Time (seconds)

How many queries can be processed per second

Performance Comparison

Tuning LSH Parameters

Key Parameters

n — **Number of Planes per Table**

More planes → More buckets (2^n) → Smaller buckets → Fewer candidates

L — **Number of Hash Tables**

More tables → Higher recall → More candidates from union

Trade-offs

↑ n (more planes)

- ✓ Faster (fewer candidates)
- ✗ Lower recall (miss neighbors)

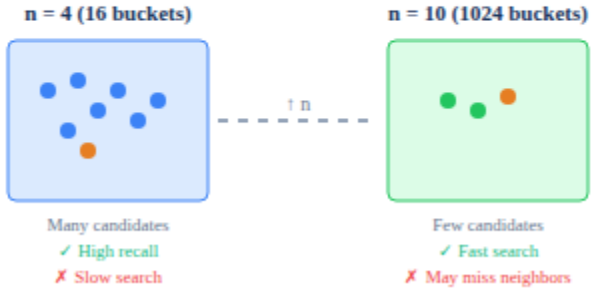
↑ L (more tables)

- ✓ Higher recall
- ✗ More memory & candidates

⚖ Finding the Balance

Tune n and L together to meet recall & speed targets

Effect of n (Planes per Table)



Expected Bucket Size

$$E[\text{bucket size}] \approx N / 2^n$$

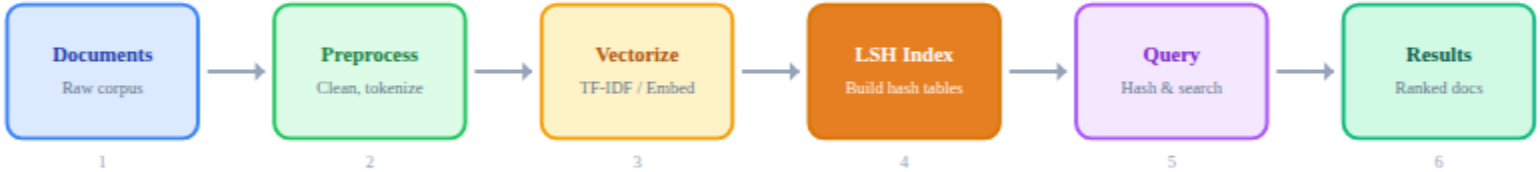
N = total documents, n = number of planes

PART IV

Practical Applications

Real-World Use Cases of LSH & Document Search

Complete Document Search Pipeline



Python Implementation

```
# 1. Load and preprocess
docs = load_documents("corpus.txt")
clean = [preprocess(d) for d in docs]

# 2. Vectorize with TF-IDF
vectorizer = TfidfVectorizer()
vectors = vectorizer.fit_transform(clean)

# 3. Build LSH index
lsh = MultiTableLSH(n_tables=10,
n_planes=15,
dim=vectors.shape[1])
for i, vec in enumerate(vectors):
lsh.index(i, vec)

# 4. Query
query = "machine learning NLP"
q_vec = vectorizer.transform([query])
results = lsh.query(q_vec, k=10)
```

Vectorization Options

TF-IDF
Sparse, interpretable, fast

Word2Vec / GloVe
Dense, semantic similarity

BERT / Sentence-BERT
Contextual, state-of-the-art

Application: Recommendation Systems

Content-Based Recommendation with LSH

Item Representation

🎬 Movies
Genre, director, actors, plot keywords, year

🎵 Music
Genre, tempo, key, artist, audio features

🛒 Products
Category, description, price range, brand

📄 Articles
Topic, TF-IDF, entities, sentiment

Why LSH for Recommendations?

- ✓ **Real-time Recommendations**
  Sub-millisecond lookups for live systems
- ✓ **Millions of Items**
  Scale to massive catalogs efficiently
- ✓ **Cold Start Solution**
  Find similar items for new products
- ✓ **Dynamic Updates**
  Add new items without full reindex

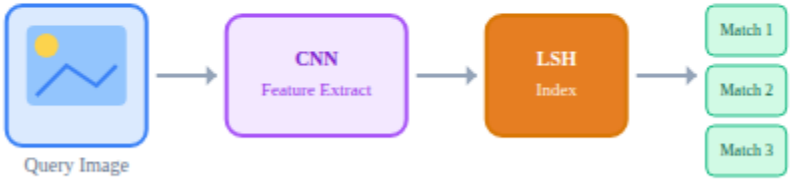
📊 Performance Example

Netflix-like catalog
10,000 movies

Brute Force	~100ms
LSH (L=10)	~2ms
Speedup	50×

Application: Image Search & Reverse Lookup

Image → Vector → LSH



Use Cases

- Google Reverse Image
- Pinterest Visual Search
- E-commerce "Shop Look"
- Copyright Detection

Feature Extraction Methods

Deep Learning (CNN)

ResNet, VGG, EfficientNet → 512-2048d vectors

✓ Best for semantic similarity

Traditional Features

SIFT, ORB, Color Histograms

⚡ Faster, less accurate

Perceptual Hashing

pHash, dHash → 64-256 bit vectors

📋 Good for near-duplicate detection

Python Example

```
from
torchvision import models

# Load pretrained CNN
resnet = models.resnet50(pretrained=True)
resnet.fc = Identity() # Remove classifier

# Extract features (2048-dim)
def
get_features(img):
   tensor = preprocess(img)
   return resnet(tensor).numpy()

# Build LSH index
lsh = LSH(dim=2048, n_planes=20)
for img in image_database:
   lsh.index(get_features(img))
```

Lab Practice

Building a Document Search System with LSH

Lab Overview: Document Search System

🎯 Learning Objectives

- ✓ Implement TF-IDF vectorization from scratch
- ✓ Build LSH index with random hyperplanes
- ✓ Compare exact vs approximate search performance
- ✓ Tune LSH parameters (n_planes, n_tables)
- ✓ Build simple search interface

📊 Dataset

AG News Dataset

News articles from 4 categories

120K	4
train documents	categories

⏰ **Time:** ~60 minutes | 📝 **Deliverable:** Jupyter Notebook

1 Data Preprocessing

Load data, clean text, tokenize, remove stopwords

2 TF-IDF Vectorization

Compute TF-IDF vectors for all documents

3 LSH Implementation

Build hash tables with random hyperplanes

4 Search & Evaluation

Query system, compare with brute force, measure recall

★ Bonus: Parameter Tuning

Experiment with different n_planes and n_tables values

1 Task 1: Data Preprocessing

📋 Steps

1.1 Load AG News Dataset
Use pandas or datasets library

1.2 Text Cleaning
Lowercase, remove punctuation, numbers

1.3 Tokenization
Split text into tokens/words

1.4 Remove Stopwords
Filter common words (the, is, at...)

💡 Tips

- Use `re.sub()` for regex cleaning
- NLTK provides stopwords list
- Start with 10K docs for testing

💻 Starter Code

```
# Load data
from
datasets import load_dataset
dataset = load_dataset("ag_news")
docs = dataset['train']['text'][:10000]

# Preprocessing function
import
re
from
nltk.corpus import stopwords

def preprocess(text):
    # Lowercase
    text = text.lower()
    # Remove non-alphabetic
    text = re.sub(r'[^a-z\s]', '', text)
    # Tokenize
    tokens = text.split()
    # Remove stopwords
    stops = set(stopwords.words('english'))
    tokens = [t for t in tokens
          if t not in stops]
    return tokens

# Process all documents
processed = [preprocess(d) for d in docs]
```

2 Task 2: TF-IDF Vectorization

📐 Formulas

$$TF\text{-}IDF(t, d) = TF(t, d) \times IDF(t)$$

Term Frequency (TF)

$TF(t, d) = count(t \text{ in } d) / |d|$

Inverse Document Frequency (IDF)

$IDF(t) = \log(N / df(t))$

🎯 Options

A **From Scratch**
Build vocabulary, compute TF, IDF manually

B **Use sklearn**
TfidfVectorizer for quick implementation

⚠️ **Note:** Normalize vectors for cosine similarity! Use L2 norm.

💻 Implementation

```
# Option A: From scratch
from
collections import Counter
import
numpy as np

def build_vocab(docs):
   vocab = set()
   for doc in docs:
      vocab.update(doc)
   return {w: i for i, w in enumerate(vocab)}

def compute_tfidf(docs, vocab):
   N = len(docs)
   df = Counter()
   for doc in docs:
      df.update(set(doc))

   # Compute IDF
   idf = {w: np.log(N/(df[w]+1))
         for w in vocab}
   return idf


# Option B: sklearn (recommended)
from
sklearn.feature_extraction.text import
   TfidfVectorizer

vectorizer = TfidfVectorizer(
   max_features=5000,
   stop_words='english')
X = vectorizer.fit_transform(docs)
vectors = X.toarray() # (N, 5000)
```

3 Task 3: LSH Implementation

🏗️ Class Structure

```
class
LSH:
    def __init__(self, n_planes, n_tables):
        # Initialize random planes

    def _hash(self, vector, planes):
        # Hash vector to bucket ID

    def index(self, vectors, ids):
        # Add vectors to hash tables

    def query(self, vector, k):
        # Find k nearest neighbors
```

🔢 Hash Function

```
def
_hash(self, vector, planes):
    # Dot product with planes
    projections = np.dot(planes, vector)
    # Sign function → binary
    hash_bits = (projections >= 0).astype(int)
    # Binary to integer
    bucket_id = sum([b*(2**i)
        for i, b in enumerate(hash_bits)])
    return bucket_id
```

💻 Complete Implementation

```
class
LSH:
    def __init__(self, dim, n_planes=10, n_tables=5):
        self.n_planes = n_planes
        self.n_tables = n_tables
        # Random planes for each table
        self.planes = [
            np.random.randn(n_planes, dim)
            for _ in range(n_tables)]
        self.tables = [{} for _ in range(n_tables)]
        self.vectors = {}

    def index(self, vectors, ids):
        for vec, idx in zip(vectors, ids):
            self.vectors[idx] = vec
            for t in range(self.n_tables):
                bucket = self._hash(vec, self.planes[t])
                if bucket not in self.tables[t]:
                    self.tables[t][bucket] = []
                self.tables[t][bucket].append(idx)

    def query(self, vector, k=10):
        candidates = set()
        for t in range(self.n_tables):
            bucket = self._hash(vector, self.planes[t])
            if bucket in self.tables[t]:
                candidates.update(self.tables[t][bucket])
        # Rank by cosine similarity
        scores = []
        for idx in candidates:
            sim = cosine_sim(vector, self.vectors[idx])
            scores.append((idx, sim))
        return sorted(scores, key=lambda x: -x[1])[:k]
```

4 Task 4: Search & Evaluation

📊 Evaluation Metrics

Recall@K

How many of the true top-K are found?

$$\text{Recall} = |\text{LSH\_results} \cap \text{True\_NN}| \;/\; K$$

Query Time

Average time per query (ms)

Speedup

LSH time vs Brute-force time

🎯 Expected Results

~90%	10-50×
Recall@10	Speedup

💻 Evaluation Code

```
import
time

def brute_force_knn(query, vectors, k):
   """Exact nearest neighbors"""
   sims = [(i, cosine_sim(query, v))
         for i, v in enumerate(vectors)]
   return sorted(sims, key=lambda x: -x[1])[:k]

def evaluate(lsh, vectors, n_queries=100, k=10):
   recalls = []
   lsh_times, bf_times = [], []

   for _ in range(n_queries):
      q = random.choice(vectors)

      # LSH search
      t0 = time.time()
      lsh_res = lsh.query(q, k)
      lsh_times.append(time.time() - t0)

      # Brute force
      t0 = time.time()
      bf_res = brute_force_knn(q, vectors, k)
      bf_times.append(time.time() - t0)

      # Compute recall
      lsh_ids = set([x[0] for x in lsh_res])
      bf_ids = set([x[0] for x in bf_res])
      recalls.append(len(lsh_ids & bf_ids) / k)

   print(f"Recall: {np.mean(recalls):.2%}")
   print(f"Speedup: {np.mean(bf_times)/np.mean(lsh_times):.1f}x")
```