lecture_14.py

```python
from dataclasses import dataclass
import math
import torch
import torch.nn as nn
from torch.nn.functional import softmax
import numpy as np
import kenlm
import fasttext
import itertools
import mmh3
from bitarray import bitarray
from basic_util import count, repeat
from file_util import download_file
from execute_util import text, image, link
from lecture_util import article_link, named_link
from references import dolma

def main():
```

Last lecture: overview of datasets used for training language models

- Live service (GitHub) → dump/crawl (GH Archive) → processed data (The Stack)
- Processing: HTML to text, language/quality/toxicity filtering, deduplication

This lecture: deep dive into the mechanics

- Algorithms for filtering (e.g., classifiers)
- Applications of filtering (e.g., language, quality, toxicity)
- Deduplication (e.g., Bloom filters, MinHash, LSH)

```python
    filtering_algorithms()
    filtering_applications()
    deduplication()
```

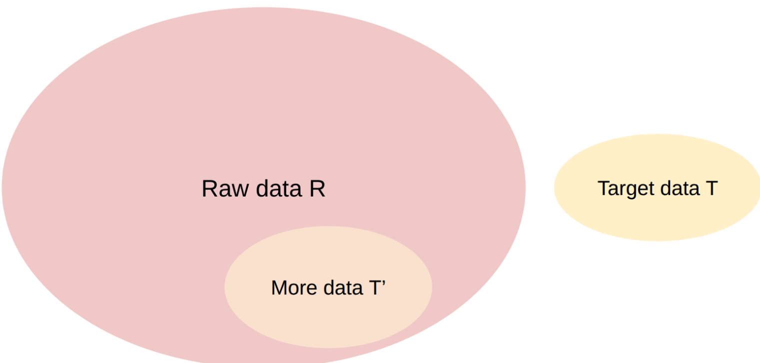
Summary

- Algorithmic tools: n-gram models (KenLM), classifiers (fastText), importance resampling (DSIR)
- Applications: language identification, quality filtering, toxicity filtering
- Deduplication: hashing scales to large datasets for fuzzy matching
- Now you have the tools (mechanics), just have to spend time with data (intuitions)

```python
def filtering_algorithms():
```

Algorithmic building block:

- Given some **target data** T and lots of **raw data** R, find subset T' of R similar to T.



Desiderata for filtering algorithm:

- Generalize from the target data (want T and T' to be different)
- Extremely fast (have to run it on R, which is huge)

```
    kenlm_main()         # Train n-gram model
    fasttext_main()      # Train a classifier
    dsir_main()          # Train bag of n-grams model, do importance resampling
    filtering_summary()
```

Survey paper on data selection [Albalak+ 2024]

```
def kenlm_main():
```

n-gram model with Kneser-Ney smoothing [article]

- KenLM: fast implementation originally for machine translation [code]
- Common language model used for data filtering
- Extremely simple / fast - just count and normalize

Concepts

Maximum likelihood estimation of n-gram language model:

- n = 3: p(in | the cat) = count(the cat in) / count(the cat)

Problem: sparse counts (count of many n-grams is 0 for large n)

Solution: Use Kneser-Ney smoothing to handle unseen n-grams [article]

- p(in | the cat) depends on p(in | cat) too

```
    # Download a KenLM language model
    model_url = "https://huggingface.co/edugp/kenlm/resolve/main/wikipedia/en.arpa.bin"
    model_path = "var/en.arpa.bin"
    download_file(model_url, model_path)
    model = kenlm.Model(model_path)

    # Use the language model
    def compute(content: str):
        # Hacky preprocessing
        content = "<s> " + content.replace(",", " ,").replace(".", " .") + " </s>"

        # log p(content)
        score = model.score(content)

        # Perplexity normalizes by number of tokens to avoid favoring short documents
        num_tokens = len(list(model.full_scores(content)))
        perplexity = math.exp(-score / num_tokens)

        return score, perplexity

    score, perplexity = compute("Stanford University was founded in 1885 by Leland and Jane Stanford as a tribute to the memory of their only child, Leland Stanford Jr.")  # @inspect score, @inspect perplexity
    score, perplexity = compute("If you believe that the course staff made an objective error in grading, you may submit a regrade request on Gradescope within 3 days after the grades are released.")  # @inspect score, @inspect perplexity
    score, perplexity = compute("asdf asdf asdf asdf asdf")  # @inspect score, @inspect perplexity
    score, perplexity = compute("the the the the the the the the the the the the the the the the")  # @inspect score, @inspect perplexity
```

CCNet

[Wenzek+ 2019]

- Items are paragraphs of text
- Sort paragraphs by increasing perplexity
- Keep the top 1/3
- Was used in LLaMA

Summary: Kneser-Ney n-gram language models (with KenLM implementation) is fast but crude

```
def fasttext_main():
```

fastText classifier [Joulin+ 2016]

- Task: text classification (e.g., sentiment classification)
- Goal was to train a fast classifier for text classification
- They found it was as good as much slower neural network classifiers

Baseline: bag of words (not what they did)

```
L = 32                            # Length of input
V = 8192                          # Vocabulary size
K = 64                            # Number of classes
W = nn.Embedding(V, K)            # Embedding parameters (V x K)
x = torch.randint(V, (L,))        # Input tokens (L) - e.g., ["the", "cat", "in", "the", "hat"]
y = softmax(W(x).mean(dim=0))     # Output probabilities (K)
```

Problem: V*K parameters (could be huge)

fastText classifier: bag of word embeddings

```
H = 16                            # Hidden dimension
W = nn.Embedding(V, H)            # Embedding parameters (V x H)
U = nn.Linear(H, K)               # Head parameters (H x K)
y = softmax(U(W(x).mean(dim=0)))  # Output probabilities (K)
```

Only H*(V + K) parameters

Implementation:

- Parallelized, asynchronous SGD
- Learning rate: linear interpolation from [some number] to 0 [article]

Bag of n-grams

```
x = ["the cat", "cat in", "in the", "the hat"]  # @inspect x
```

Problem: number of bigrams can get large (and also be unbounded)

Solution: hashing trick

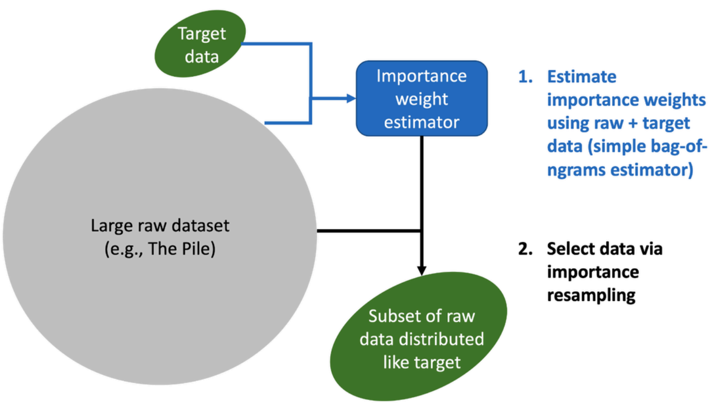
```
num_bins = 8  # In practice, 10M bins
hashed_x = [hash(bigram) % num_bins for bigram in x]  # @inspect hashed_x
```

- For quality filtering, we have K = 2 classes (good versus bad)
- In that case, fastText is just a linear classifier (H = K = 2)

In general, can use any classifier (e.g., BERT, Llama), it's just slower

```
def dsir_main():
```

Data Selection for Language Models via Importance Resampling (DSIR) [Xie+ 2023]




```
importance_sampling()
```

Setup:

- Target dataset D_p (small)
- Proposal (raw) dataset D_q (large)

Take 1:

- Fit target distribution p to D_p
- Fit proposal distribution q to D_q
- Do importance resampling with p, q, and raw samples D_q

Problem: target data D_p is too small to estimate a good model

Take 2: use hashed n-grams

```
training_text = "the cat in the hat"

# Hash the n-grams
num_bins = 4
def get_hashed_ngrams(text: str):
    ngrams = text.split(" ")  # Unigram for now
    return [hash(ngram) % num_bins for ngram in ngrams]

training_hashed_ngrams = get_hashed_ngrams(training_text)  # @inspect training_hashed_ngrams

# Learn unigram model
probs = [count(training_hashed_ngrams, x) / len(training_hashed_ngrams) for x in range(num_bins)]  # @inspect probs

# Evaluate probability of any sentence
hashed_ngrams = get_hashed_ngrams("the text")  # @inspect hashed_ngrams
prob = np.prod([probs[x] for x in hashed_ngrams])  # @inspect prob
```

Result: DSIR slightly better than heuristic classification (fastText) on the GLUE benchmark

	MNLI	QNLI	QQP	RTE	SST-2	MRPC	CoLA	STS-B	Avg
Random selection	$82.63_{0.41}$	$86.90_{0.28}$	$89.57_{0.30}$	$67.37_{1.69}$	$90.05_{0.41}$	$87.40_{1.08}$	$49.41_{3.67}$	$88.63_{0.11}$	80.25
Heuristic classification	$82.69_{0.17}$	$85.95_{0.79}$	$89.77_{0.32}$	$68.59_{1.75}$	$88.94_{0.98}$	$86.03_{0.93}$	$48.17_{3.19}$	$88.62_{0.22}$	79.85
Top-k Heuristic classfication	$83.34_{0.22}$	$88.62_{0.24}$	$89.89_{0.19}$	$70.04_{0.99}$	$91.15_{0.76}$	$86.37_{1.00}$	$53.02_{3.56}$	$89.30_{0.11}$	81.47
DSIR	$83.07_{0.29}$	$\mathbf{89.11}_{0.14}$	$89.80_{0.37}$	$\mathbf{75.09}_{2.76}$	$90.48_{0.57}$	$\mathbf{87.70}_{0.68}$	$\mathbf{54.00}_{1.34}$	$89.17_{0.13}$	**82.30**
Top-k DSIR	$\mathbf{83.39}_{0.06}$	$88.63_{0.38}$	$\mathbf{89.94}_{0.17}$	$72.49_{1.29}$	$\mathbf{91.01}_{0.79}$	$86.18_{1.12}$	$49.90_{1.10}$	$\mathbf{89.52}_{0.21}$	81.38

Comparison with fastText:

- Modeling distributions is a more principled approach capturing diversity
- Similar computation complexity
- Both can be improved by better modeling

```
def importance_sampling():
```

Setup:

- Target distribution p (want samples from here)
- Proposal distribution q (have samples from here)

```
vocabulary = [0, 1, 2, 3]
p = [0.1, 0.2, 0.3, 0.4]
q = [0.4, 0.3, 0.2, 0.1]

# 1. Sample from q
n = 100
samples = np.random.choice(vocabulary, p=q, size = n)  # @inspect samples
```

Samples (q): [0 2 0 2 1 1 0 1 0 1 0 0 0 0 1 1 0 3 1 1 1 1 1 1 3 1 0 2 0 1 3 0 2 1 1 2 0 0 0 3 2 1 1 0 1 1 0 3 2 0 2 0 1 0 1 2 2 2 0 0 0 0 2 1 0 2 0 1 3 0 0 0 0 0 0 0 1 1 2 2 2 1 0 1 1 0 1 0 1 0 1 1 3 3 0 1 0 0 2 0]

```
# 2. Compute weights over samples (w \propto p/q)
w = [p[x] / q[x] for x in samples]  # @inspect w
z = sum(w)  # @inspect z
w = [w_i / z for w_i in w]  # @inspect w

# 3. Resample
samples = np.random.choice(samples, p=w, size=n)  # @inspect samples
```

Resampled (p): [2 2 1 3 3 2 0 3 3 2 1 0 3 2 0 3 3 1 3 1 3 2 3 2 3 2 3 1 0 3 2 2 2 0 2 1 2 0 3 1 1 1 3 1 3 3 3 1 0 2 3 1 2 1 2 2 2 2 1 1 0 2 1 1 0 2 3 2 1 3 2 3 1 2 3 2 3 2 1 1 3 3 3 1 1 3 2 1 3 1 3 1 0 3 1 1 2 3 2 1]

```
def filtering_summary():
```

Implementations: KenLM, fastText, DSIR

General framework

Given target T and raw R, find subset of R similar to T
1. Estimate some model based on R and T and derive a scoring function
2. Keep examples in R based on their score

Instantiations of the framework

Generative model of T (KenLM):
1. score(x) = p_T(x)
2. Keep examples x with score(x) >= threshold (stochastically)

Discriminative classifier (fastText):
1. score(x) = p(T | x)
2. Keep examples x with score(x) >= threshold (stochastically)

Importance resampling (DSIR):
1. score(x) = p_T(x) / p_R(x)
2. Resample examples x with probability proportional to score(x)

```
def filtering_applications():
```

The same data filtering machinery can be used for different filtering tasks.

```
language_identification()
quality_filtering()
toxicity_filtering()
```

```
def language_identification():
```

Language identification: find text of a specific language (e.g., English)

Why not just go multilingual?
- Data: difficult to do curation / processing of high-quality data in any given language
- Compute: in computed-limited regime, less compute/tokens dedicated to any given language

Models differ on multilinguality:
- English was only 30% of BLOOM (was undertrained), English performance suffered [Laurençon+ 2023]
- Most frontier models (GPT-4, Claude, Gemini, Llama, Qwen) are heavily multilingual (sufficiently trained)

fastText language identification [article]
- Off-the-shelf classifier
- Supports 176 languages
- Trained on multilingual sites: Wikipedia, Tatoeba (translation site) and SETimes (Southeast European news)

Example: Dolma keeps pages with p(English) >= 0.5 [Soldaini+ 2024]

```
# Download the model
model_url = "https://dl.fbaipublicfiles.com/fasttext/supervised-models/lid.176.bin"
model_path = "var/lid.176.bin"
download_file(model_url, model_path)
model = fasttext.load_model(model_path)

# Make predictions
predictions = model.predict(["The quick brown fox jumps over the lazy dog."])  # English @inspect predictions
predictions = model.predict(["The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog."])  # Duplicate @inspect predictions
predictions = model.predict(["OMG that movie was 🔥🔥! So dope 😎🤘!"])  # Informal English @inspect predictions
predictions = model.predict(["Auf dem Wasser zu singen"])  # German @inspect predictions
predictions = model.predict(["The quadratic formula is $x = frac-bpmsqrtb^2 - 4ac2a.$"])  # Latex @inspect predictions
predictions = model.predict(["for (int i = 0; i < 10; i++)"])  # C++ @inspect predictions
```

```
predictions = model.predict(["Hello!"])  # English @inspect predictions
predictions = model.predict(["Bonjour!"])  # French @inspect predictions
predictions = model.predict(["Feliz Navidad / Próspero año y felicidad / I wanna wish you a Merry Christmas"])  # Spanish + English @inspect predictions
```

Caveats:
- Difficult for short sequences
- Difficult for low-resource languages
- Could accidentally filter out dialects of English
- Hard for similar languages (Malay and Indonesian)
- Ill-defined for code-switching (e.g., Spanish + English)

OpenMathText [Paster+ 2023]
- Goal: curate large corpus of mathematical text from CommonCrawl
- Use rules to filter (e.g., contains latex commands)
- KenLM trained on ProofPile, keep if perplexity < 15000
- Trained fastText classifier to predict mathematical writing, threshold is 0.17 if math, 0.8 if no math

Result: produced 14.7B tokens, used to train 1.4B models that do better than models trained on 20x data

```
def quality_filtering():
```

- Some deliberately do not use model-based filtering (C4, Gopher, RefinedWeb, FineWeb, Dolma)
- Some use model-based filtering (GPT-3, LLaMA, DCLM) [becoming the norm]

GPT-3 [Brown+ 2020]
- Positives: samples from {Wikipedia, WebText2, Books1, Books2}
- Negatives: samples from CommonCrawl

Train linear classifier based on word features [article]

Keep documents stochastically based on score

```
def keep_document(score: float) -> bool:
    return np.random.pareto(9) > 1 - score
```

** LLaMA/RedPajama** [Touvron+ 2023]
- Positives: samples from pages **referenced** by Wikipedia
- Negatives: samples from CommonCrawl
- Keep documents that are classified positive

phi-1 [Gunasekar+ 2023]

Philosophy: really high quality data (textbooks) to train a small model (1.5B)

Includes synthetic data from GPT 3.5 (later: GPT-4) and filtered data

```
R = "Python subset of the Stack"   # Raw data
prompt = "determine its educational value for a student whose goal is to learn basic coding concepts"
T = "Use GPT-4 with this prompt to classify 100K subset of R to get positive examples"
```

Train random forest classifier on T using output embedding from pretrained codegen model

Select data from R that is classified positive by the classifier

Result on HumanEval:
- Train 1.3B LM on Python subset of The Stack (performance: 12.19% after 96K steps)
- Train 1.3B LM on new filtered subset (performance: 17.68% after 36K steps) - better!

```
@dataclass
class Example:
    text: str
    label: int


def toxicity_filtering():
    # WARNING: potentially offensive content below
```

Toxicity filtering in Dolma [Soldaini+ 2024]

Dataset: Jigsaw Toxic Comments dataset (2018) [dataset]

- Project goal: help people have better discussions online [article]
- Data: comments on Wikipedia talk page annotated with {toxic, severe_toxic, obscene, threat, insult, identity_hate}

Trained 2 fastText classifiers
- hate: positive = {unlabeled, obscene}, negative = all else
- NSFW: positive = {obscene}, negative = all else

```
    # Examples from the dataset: (obscene, text)
    train_examples = [
        Example(label=0, text="Are you threatening me for disputing neutrality? I know in your country it's quite common to bully your way through a discussion and push outcomes you want. But this is not Russia."),
        Example(label=1, text="Stupid peace of shit stop deleting my stuff asshole go die and fall in a hole go to hell!"),
    ]

    # Download model
    model_url = "https://dolma-artifacts.org/fasttext_models/jigsaw_fasttext_bigrams_20230515/jigsaw_fasttext_bigrams_nsfw_final.bin"
    model_path = "var/jigsaw_fasttext_bigrams_nsfw_final.bin"
    download_file(model_url, model_path)
    model = fasttext.load_model(model_path)

    # Make predictions
    predictions = model.predict([train_examples[0].text])  # @inspect predictions
    predictions = model.predict([train_examples[1].text])  # @inspect predictions
    predictions = model.predict(["I love strawberries"])  # @inspect predictions
    predictions = model.predict(["I hate strawberries"])  # @inspect predictions


def print_predict(model, content):
    """Run classifier `model` on `content` and print out the results."""
    predictions = model.predict([content])
    print(predictions)
    #labels, prob =
    #labels = ", ".join(labels)
    #text(f"{content} => {labels} {prob}")


def deduplication():
```

Two types of duplicates:
- Exact duplicates (mirror sites, GitHub forks) [Gutenberg mirrors]
- Near duplicates: same text differing by a few tokens

Examples of near duplicates:
- Terms of service and licenses [MIT license]
- Formulaic writing (copy/pasted or generated from a template)

Dataset	Example	Near-Duplicate Example
Wiki-40B	\n_START_ARTICLE_\nHum Award for Most Impactful Character \n_START_SECTION_\nWinners and nominees\n_START_PARAGRAPH_\nIn the list below, winners are listed first in the colored row, followed by the other nominees. [...]	\n_START_ARTICLE_\nHum Award for Best Actor in a Negative Role \n_START_SECTION_\nWinners and nominees\n_START_PARAGRAPH_\nIn the list below, winners are listed first in the colored row, followed by the other nominees. [...]
LM1B	I left for California in 1979 and tracked Cleveland 's changes on trips back to visit my sisters .	I left for California in 1979 , and tracked Cleveland 's changes on trips back to visit my sisters .
C4	Affordable and convenient holiday flights take off from your departure country, "Canada". From May 2019 to October 2019, Condor flights to your dream destination will be roughly 6 a week! Book your Halifax (YHZ) - Basel (BSL) flight now, and look forward to your "Switzerland" destination!	Affordable and convenient holiday flights take off from your departure country, "USA". From April 2019 to October 2019, Condor flights to your dream destination will be roughly 7 a week! Book your Maui Kahului (OGG) - Dubrovnik (DBV) flight now, and look forward to your "Croatia" destination!

- Minor formatting differences in copy/pasting

Product description repeated 61,036 times in C4

'"by combining fantastic ideas, interesting arrangements, and follow the current trends in the field of that make you more inspired and give artistic touches. We'd be honored if you can apply some or all of these design in your wedding. believe me, brilliant ideas would be perfect if it can be applied in real and make the people around you amazed!

[example page]

Deduplication training data makes language models better [Lee+ 2021]

- Train more efficiently (because have fewer tokens)
- Avoid memorization (can mitigate copyright, privacy concerns)

Design space:
1. What is an item (sentence, paragraph, document)?
2. How to match (exact match, existence of common subitem, fraction of common subitems)?
3. What action to take (remove all, remove all but one)?

Key challenge:
- Deduplication is fundamentally about comparing items to other items
- Need linear time algorithms to scale

```
hash_functions()

exact_deduplication()
bloom_filter()

jaccard_minhash()
locality_sensitive_hashing()


def hash_functions():
```

- Hash function h maps item to a hash value (integer or string)
- Hash value much smaller than item
- Hash collision: h(x) = h(y) for x ≠ y

Tradeoff between efficiency and collision resistance [article]
- Cryptographic hash functions (SHA-256): collision resistant, slow (used in bitcoin)
- DJB2, MurmurHash, CityHash: not collision resistant, fast (used for hash tables)

We will use MurmurHash:

```
h = mmh3.hash("hello")  # @inspect h


def exact_deduplication():
```

Simple example
1. Item: string
2. How to match: exact match
3. Action: remove all but one

```
# Original items
items = ["Hello!", "hello", "hello there", "hello", "hi", "bye"]  # @inspect items

# Compute hash -> list of items with that hash
hash_items = itertools.groupby(sorted(items, key=mmh3.hash), key=mmh3.hash)

# Keep one item from each group
deduped_items = [next(group) for h, group in hash_items]  # @inspect deduped_items
```

- Pro: simple, clear semantics, high precision
- Con: does not deduplicate near duplicates
- This code is written in a MapReduce way, can easily parallelize and scale

C4 [Raffel+ 2019]
1. Item: 3-sentence spans
2. How to match: use exact match
3. Action: remove all but one

Warning: when a 3-sentence span is removed from the middle of a document, the resulting document might not be coherent

```
def bloom_filter():
```

Goal: efficient, approximate data structure for testing set membership

Features of Bloom filters

- Memory efficient
- Can update, but can't delete
- If return 'no', definitely 'no'
- If return 'yes', most likely 'yes', but small probability of 'no'
- Can drive the false positive rate down exponentially with more time/compute

```
    items = ["the", "cat", "in", "the", "hat"]
    non_items = ["what", "who", "why", "when", "where", "which", "how"]
```

First, make the range of hash function small (small number of bins).

```
    m = 8  # Number of bins
    table = build_table(items, m)
    for item in items:
        assert query_table(table, item, m) == 1
    result = {item: query_table(table, item, m) for item in non_items}  # @inspect result
    num_mistakes = count(result.values(), True)  # @inspect num_mistakes
    false_positive_rate = num_mistakes / (len(items) + num_mistakes)  # @inspect false_positive_rate
```

Problem: false positives for small bins

Naive solution: increase the number of bins

Error probability is O(1/num_bins), decreases polynomially with memory

Better solution: use more hash functions

```
    k = 2  # Number of hash functions
    table = build_table_k(items, m, k)
    for item in items:
        assert query_table_k(table, item, m, k) == 1
    result = {item: query_table_k(table, item, m, k) for item in non_items}  # @inspect result
    num_mistakes = count(result.values(), 1)  # @inspect num_mistakes
    false_positive_rate = num_mistakes / (len(items) + num_mistakes)  # @inspect false_positive_rate
```

Reduced the false positive rate!

```
    false_positive_rate_analysis()


def false_positive_rate_analysis():
```

Assume independence of hash functions and items [article]

```
    m = 1000   # Number of bins
    k = 10     # Number of hash functions
    n = 100    # Number of items we're inserting
```

Consider a test input (not in the set) that would hash into a given test bin (say, i).

Now consider putting items into the Bloom filter and seeing if it hits i.

```
    # Insert one item, ask if the test bin B(i) = 1?
    # B: [0 0 1 0 0 0 0 0 0 0] - have to miss 1 time
    f = 1 / m                              # P[B(i) = 1 after 1 insertion with 1 hash function]  # @inspect f
    # B: [0 0 1 0 0 1 0 1 0 0] - have to miss k times
    f = 1 - (1 - 1 / m) ** k               # P[B(i) = 1 after 1 insertion with k hash functions]  # @inspect f

    # Insert n items, ask if the test bin B(i) = 1?
    # Have to miss k*n times
    f = 1 - (1 - 1 / m) ** (k * n)         # P[B(i) = 1 after n insertions for 1 hash function]  # @inspect f
    # Get k chances to miss (since test input is hashed k times too)
    f = f ** k                             # P[B(i) = 1 after n insertions for k hash functions]  # @inspect f
```

Optimal value of k (given fixed m / n ratio) [results in f ~ 0.5]

```
    k = math.log(2) * m / n  # @inspect k
```

Resulting false positive rate (improved)

```
    f = 0.5 ** k  # @inspect f
```

Tradeoff between compute (k), memory (m), and false positive rate (f) [lecture notes]

Example: Dolma

- Set false positive rate to 1e-15
- Perform on items = paragraphs

```
def build_table(items: list[str], num_bins: int):
    """Build a Bloom filter table of size `num_bins`, inserting `items` into it."""
    table = bitarray(num_bins)  # @inspect table
    for item in items:
        h = mmh3.hash(item) % num_bins  # @inspect item, @inspect h
        table[h] = 1  # @inspect table
    return table


def build_table_k(items: list[str], num_bins: int, k: int):
    """Build a Bloom filter table of size `num_bins`, inserting `items` into it.
    Use `k` hash functions."""
    table = bitarray(num_bins)  # @inspect table
    for item in items:
        # For each of the k functions
        for seed in range(k):
            h = mmh3.hash(item, seed) % num_bins  # @inspect item, @inspect h, @inspect seed
            table[h] = 1  # @inspect table
    return table


def query_table(table: bitarray, item: str, num_bins: int, seed: int = 0):
    """Return whether `item` is in the `table`."""
    h = mmh3.hash(item, seed) % num_bins
    return table[h]


def query_table_k(table: bitarray, item: str, num_bins: int, k: int):
    """Return 1 if table set to 1 for all `k` hash functions."""
    return int(all(
        query_table(table, item, num_bins, seed)
        for seed in range(k)
    ))


def jaccard_minhash():
```

Let's now look at approximate set membership.
First we need a similarity measure.

Jaccard similarity

Definition: Jaccard(A, B) = |A intersect B| / |A union B|

```
A = {"1", "2", "3", "4"}
B = {"1", "2", "3", "5"}

def compute_jaccard(A, B):
    intersection = len(A & B)  # @inspect intersection
    union = len(A | B)  # @inspect union
    return intersection / union
jaccard = compute_jaccard(A, B)  # @inspect jaccard
```

Definition: two documents are **near duplicates** if their Jaccard similarity >= threshold

Algorithmic challenge: find near duplicates in linear time

MinHash

MinHash: a random hash function h so that Pr[h(A) = h(B)] = Jaccard(A, B)

Normally, you want different items to hash to different hashes
...but here, you want collision probability to depend on similarity

```
def minhash(S: set[str], seed: int):
    return min(mmh3.hash(x, seed) for x in S)
```

Characteristic matrix representation:

```
item | A | B
1    | 1 | 1
2    | 1 | 1
3    | 1 | 1
4    | 1 | 0
5    | 0 | 1
```

Random hash function induces a permutation over items
Look at which item is first in A and which item is first in B.
Each item has the same probability as being first (min)

- If 1, 2, 3 is first, then first in A = first in B.
- If 4, 5 is first, then first in A ≠ first in B.

```
# Verify MinHash approximates Jaccard as advertised
n = 100  # Generate this many random hash functions
matches = [minhash(A, seed) == minhash(B, seed) for seed in range(n)]
estimated_jaccard = count(matches, True) / len(matches)  # @inspect estimated_jaccard
assert abs(estimated_jaccard - jaccard) < 0.01
```

Now we can hash our items, but a collision doesn't tell us Jaccard(A, B) > threshold.

```
def locality_sensitive_hashing():
```

Locality sensitive hashing (LSH) [book chapter]

Suppose we hash examples just one MinHash function
P[A and B collide] = Jaccard(A, B)
On average, more similar items will collide, but very stochastic...

Goal: have A and B collide if Jaccard(A, B) > threshold
We have to somehow sharpen the probabilities...

Solution: use n hash functions
Break up into b bands of r hash functions each (n = b * r)

```
n = 12      # Number of hash functions
b = 3       # Number of bands
r = 4       # Number of hash functions per band
```

Hash functions:

```
h1 h2 h3 h4  |  h5 h6 h7 h8  |  h9 h10 h11 h12
```

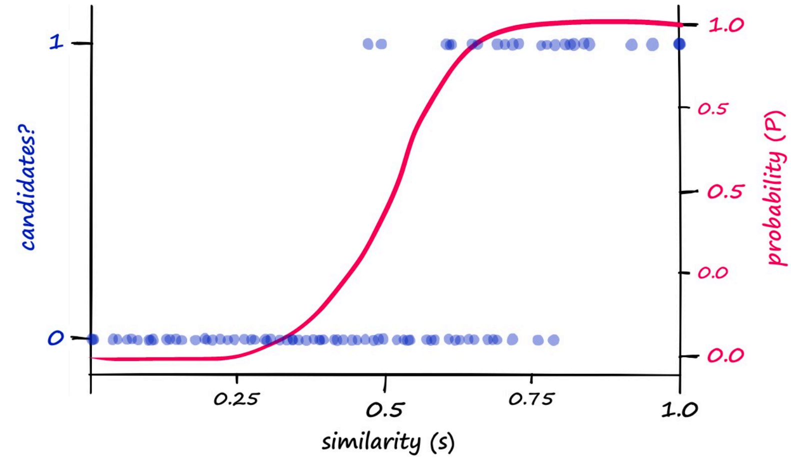
Key: A and B collide if for *some* band, *all* its hash functions return same value
As we will see, the and-or structure of the bands sharpens the threshold

Given Jaccard(A, B), what is the probability that A and B collide?

```
def get_prob_collision(sim, b, r):  # @inspect sim, @inspect b, @inspect r
    prob_match = sim ** r                        # Probability that a fixed band matches  @inspect prob_match
    prob_collision = 1 - (1 - prob_match) ** b   # Probability that some band matches  @inspect prob_collision
    return prob_collision
```

Example

```
prob_collision = get_prob_collision(sim=0.8, b=5, r=10)  # @inspect prob_collision
```



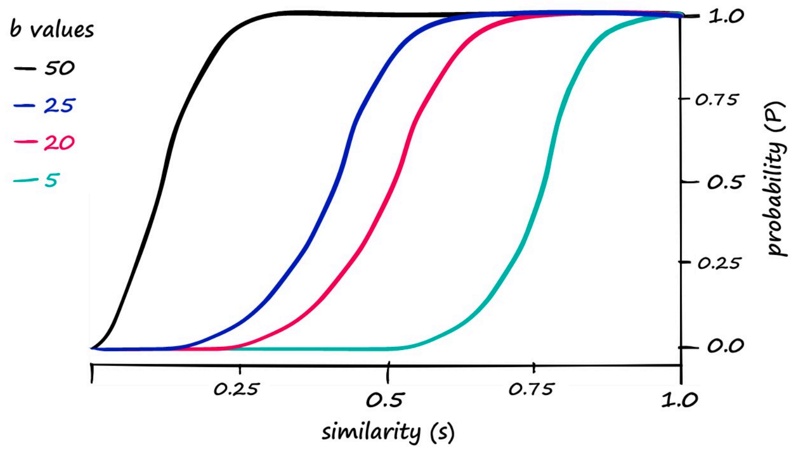

```
sims = [0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.98]
probs = {sim: get_prob_collision(sim=sim, b=10, r=10) for sim in sims}  # @inspect probs
```

Increasing r sharpens the threshold and moves the curve to the right (harder to match)

```
probs = {sim: get_prob_collision(sim=sim, b=10, r=20) for sim in sims}  # @inspect probs
```

Increasing b moves the curve to the left (easier to match)

```
probs = {sim: get_prob_collision(sim=sim, b=20, r=20) for sim in sims}  # @inspect probs
```



Example setting [Lee+ 2021]: n = 9000, b = 20, r = 450

```
b = 20
r = 450
```

What is the threshold (where the phase transition happens)?

```
threshold = (1 / b) ** (1 / r)  # @inspect threshold
```

Probability that a fixed band matches:

```
prob_match = (1 / b)  # @inspect prob_match
```

Probability that A and B collide (≈ 1-1/e):

```
prob_collision = 1 - (1 - 1 / b) ** b  #  @inspect prob_collision


if __name__ == "__main__":
    main()
```