

THE LOST ALGORITHMS

Volume Two — Patterns and Decisions

By

MARTIN CARR

INDEPENDENT PUBLISHER

Copyright © 2026 by Martin Carr.

All rights reserved.

No part of this book may be reproduced, digitalized, or transmitted in any form without written permission from the publisher, except in the case of brief quotations embodied in critical reviews.

Publisher: Independent Publisher

ISBN Placeholder

TABLE OF CONTENTS

Preface: After the Labels	4
A Patent in Five Minutes	8
Part I — The Pattern: Finding Structure Without Labels	10
Sorting the Unsorted (Scalable k-Means)	11
Clusters of Any Shape (DBSCAN & OPTICS)	19
The Hidden Mixture (EM and Gaussian Mixtures)	27
Cutting the Graph (Spectral Clustering)	35
Climbing to the Peak (Mean Shift & Kernel Density Estimation)	43
Part II — The Sequence: Time, State, and Memory	51
The Filter That Tracks (Kalman Filters)	52
The Hidden Chain (Hidden Markov Models)	60
Part III — The Curve: Reasoning About What You Don't Know	69
Functions With Error Bars (Gaussian Processes)	70
The Spike in the Data (Anomaly Detection)	78
Part IV — The Choice: Learning From Consequences	86
The Slot Machines (Multi-Armed Bandits)	87
Learning to Act (Q-Learning & Temporal Difference)	95
Critic and Actor (Policy-Gradient Reinforcement Learning)	103
Part V — The Meaning: Language, Recommendation, and Compression	112
Words as Vectors (Word Embeddings)	113
The Bag of Words (Text Classification)	121

PREFACE: AFTER THE LABELS

Volume One of this series was a tour of the answer key. Every algorithm in it — the optimisers breeding toward a known fitness, the neural networks descending toward known targets, the classifiers trained on examples that someone had already sorted into categories — assumed that the right answer existed and was available. The machine's task was to generalise from answers it had been shown. That is supervised learning, and it is the comfortable, well-lit half of the field.

The world is not so generous, and this volume is about everything that happens once the answer key is taken away.

Consider how little of the data around you comes labelled. Nobody has sorted the photographs on a billion phones by the faces they contain. Nobody has divided a company's millions of customers into

the segments that actually exist in their behaviour. Nobody has told a drone its true position at each instant as it fights the wind — the truth is never observed directly, only inferred from noisy, partial sensors. Nobody has marked, among a day's millions of transactions, the handful that are fraud. And in the richest problems of all — a robot learning to walk, an agent learning to play, a system learning which action to take — there is no fixed answer to learn at all, only consequences that arrive after you act, rewards and penalties from which you must work out, for yourself, what good behaviour even means.

When no one will hand you the answer, the methods of Volume One fall silent. You cannot minimise an error you cannot measure; you cannot fit a label you do not have. Something different is needed — and the surprising, encouraging truth is that the something different was worked out decades ago, patented by the same kinds of institutions, and has now expired into the same public domain. Finding structure in unlabelled data; tracking hidden state through time; reasoning honestly about uncertainty; learning to act from reward alone; pulling meaning out of raw language: all of it has a history written in the patent record, and all of it is now free. Volume One made the case that patent expiry is one of technology's quiet engines — the force that gave the world generic medicine and the desktop 3D-printing boom — and that machine learning's own patent cliff had gone unharvested until this series. This volume gathers the second half of that harvest.

This volume tells that history in five movements. The Pattern is about clustering and density — making sense of data that nobody labelled, by discovering the groups and structure already latent in it. The Sequence is about time and hidden state — the Kalman filter that steadies every drone and spacecraft, and the hidden Markov models that infer what you cannot see from what you can. The Curve is about uncertainty itself — Gaussian processes that predict and tell you how sure they are, and the anomaly detectors that find the spike no one flagged. The Choice is about learning from consequences — the bandits behind every A/B test, and the reinforcement learning that learns to act from reward rather than instruction. And The Meaning is about language, taste, and compression — the word embeddings, recommenders, and quantisers that taught machines to handle the messy, human world of words and preferences.

Each chapter keeps the five-beat shape that ran through Volume One: the filing (the real, expired patent, with its assignee and citation count), the idea (built from zero, no prior knowledge assumed), the story (who built it and what it became), the code (lifted directly from the `patentml` library, fully runnable), and the inheritance (what this newly-free algorithm is good for now). And as before, every code listing is the genuine article — install the library, and the algorithm on the page is the algorithm in your hands.

You do not need to have read Volume One to read this one. Where a chapter leans on an idea from the first volume — gradient descent,

decision trees, the patent bargain itself — it gives you what you need in a sentence, and a brief primer on patents and the public domain follows this preface for readers starting here. But the two volumes are halves of one story, and they meet in the middle. Volume One asked: learn the answer I am showing you. Volume Two asks the harder and more interesting question: find the answer that no one knows.

CHAPTER 1

A PATENT IN FIVE MINUTES

For readers beginning with this volume, here is the whole premise in brief; Volume One's appendices give the fuller version.

A patent is a bargain. An inventor describes an invention in complete, working detail and publishes that description for anyone to read. In exchange, the government grants a temporary monopoly — for a US utility patent of the era this book covers, twenty years from the filing date — during which others may not make or sell the invention. The key word is publishes: a patent is the opposite of a secret. It is a permanent, public teaching document, and the monopoly is the fee the public pays to be taught.

When the twenty years end, the monopoly ends — completely and forever. The invention falls into the public domain, where anyone may build, use, sell, or teach it, with no permission and no payment.

A patent cannot be renewed back into force. Every patent in this book was filed long enough ago that this has already happened; you can verify any filing date yourself on Google Patents in about a minute.

One distinction matters. A patent is the opposite of a trade secret. A trade secret stays protected only while it stays hidden, possibly forever; a patent protects something published, for only twenty years. The institutions in this book chose, decades ago, to patent rather than to hide — buying two decades of monopoly, and guaranteeing that when those decades ran out, the knowledge would belong to everyone. That time is now spent. The teaching remains.

A final note, as in Volume One: the ideas in this book are public domain and belong to everyone; the code, the `patentml` library, is original work released by the author under the permissive MIT licence. The freedom to use the idea is a gift of time and the patent system. The freedom to use the code is a gift from the author. Take both, and let us begin where the labels run out.

PART I — THE PATTERN: FINDING STRUCTURE WITHOUT LABELS

Here the answer key vanishes. The five chapters of this part are about clustering: discovering the groups that already exist in data that no one has sorted. You are handed a heap of points — customers, documents, pixels, measurements — with no labels, no categories, no teacher, and asked the oldest question in unsupervised learning: what natural groups are hiding in here? Each chapter answers it differently — by carving space into regions, by following density, by assuming hidden bell curves, by cutting graphs, by climbing to peaks — and each was patented, decades ago, by a company trying to make sense of databases too large to eyeball. We begin with the workhorse that everyone reaches for first: k-means.

CHAPTER 2

SORTING THE UNSORTED (SCALABLE K-MEANS)

THE FILING

US Patent 6,012,058 — "Scalable system for K-means clustering of large databases"

Assignee: Microsoft Corporation. Filed 1998. Now expired; public domain.

The companion US Patent 6,092,072 (Lucent) covers the hierarchical two-level variant. Both now expired; public domain.

k-means is the most widely used clustering algorithm in the world, and it is usually the first one anybody learns. The basic idea is decades old, but Microsoft's patent addresses the problem that turns a textbook method into a real-world tool: scale. The naive algorithm wants the whole dataset in memory and sweeps over all of it

repeatedly; that breaks when the database has hundreds of millions of rows. The patent describes a scalable k-means that processes data in chunks and updates its clusters incrementally, so the full dataset never needs to sit in memory at once. That is the difference between a classroom exercise and something you can run on a corporate data warehouse — and it is exactly the concern that recurs across this part of the book, because clustering only earns its keep when the data is too big to sort by hand.

THE IDEA

You have a cloud of points and you believe they fall into k natural groups, but you do not know which point belongs to which group, nor where the groups are centred. k-means finds out by a beautifully simple back-and-forth that you could explain to a child.

Begin by guessing k centre points — call them centroids. Now repeat two steps until nothing changes:

1. Assign. Attach every data point to the nearest centroid. This carves the space into k groups — everyone belongs to whichever centre they are closest to.
2. Update. Move each centroid to the average position of all the points that were just assigned to it. The centre relocates to the middle of its flock.

That's it. Assign points to centres; move centres to the middle of their points; repeat. Each round, the centroids drift toward the dense heart of their groups, the assignments sharpen, and after a handful of iterations the whole thing stops moving — it has converged. The points are now sorted into k clusters, and you never told it a single label.

Two practical truths shape how k -means behaves. First, the starting guess matters: a bad initial scatter of centroids can converge to a poor grouping, so good implementations (including this one) use k -means++, a clever initialisation that spreads the starting centroids out, choosing each new one with probability proportional to its distance from the existing ones, which reliably gives better results. Second, you must choose k yourself — the algorithm will happily split data into however many groups you ask for, whether or not that many groups really exist, so picking the right k is a genuine part of the craft (the "elbow" in a plot of cluster tightness against k is the usual guide). The scalability that Microsoft's patent adds is the mini-batch trick: rather than re-examining every point each round, update the centroids from a random sample, which is far cheaper and converges to nearly the same place — the key to running k -means on data that does not fit in memory.

THE STORY

k -means dates to the 1950s and 60s — Stuart Lloyd's algorithm at

Bell Labs, and parallel work by others — making it one of the oldest algorithms in either volume of this book. It survived sixty years of competition to remain the default clustering method because it is fast, simple, scales well, and is good enough astonishingly often. When a data scientist says "let me just cluster this and see," they almost always mean k-means.

Microsoft's 1998 patent belongs to the moment when datasets outgrew memory and the practical challenge became not whether k-means worked but whether it could run at all on a database with hundreds of millions of records. The scalable, chunk-processing variant — read a batch, update the sufficient statistics, discard the batch — was the answer, and the same idea (mini-batch k-means) is still the standard way to cluster massive data today. The algorithm is everywhere: customer segmentation, image colour quantisation (reducing a photo to k representative colours — a direct cousin of the vector quantisation in this volume's final chapters), document grouping, anomaly preprocessing, feature learning. It is the unglamorous, reliable foundation of unsupervised learning.

THE CODE

The two steps are the whole algorithm, and `patentml` writes each as a small method. `Assignment` attaches each point to its nearest centroid:

```
def _assign(self, data):
```

```

        """Attach every point to the nearest centroid."""
        return [min(range(self.k), key=lambda j: euclidean(p,
self.centroids[j]))
                for p in data]

```

Update moves each centroid to the mean of its assigned points — the incremental, sum-and-count form the scalability patent calls for:

```

def _update_centroids(self, data, labels):
    """US6012058: move each centroid to the average of its points."""
    sums = [[0.0] * dim for _ in range(self.k)]
    counts = [0] * self.k
    for point, label in zip(data, labels):
        for d in range(dim):
            sums[label][d] += point[d]
        counts[label] += 1
    return [[s / counts[j] for s in sums[j]] if counts[j] else
self.centroids[j]
            for j in range(self.k)]

```

The fit loop alternates the two until the centroids stop moving — and the mini-batch option is right there, sampling a subset each round for scalability:

```

def fit(self, data):
    self.centroids = self._init_centroids(data)    # k-means++ start

```

```
for _ in range(self.max_iter):
    batch = random.sample(data, self.batch_size) if self.batch_size
else data
    labels = self._assign(batch)
    new_centers = self._update_centroids(batch, labels)
    shift = max(euclidean(o, n) for o, n in zip(self.centroids,
new_centers))
    self.centroids = new_centers
    if shift < self.tol:                # converged — centroids
stopped moving
        break
    self.labels = self._assign(data)
    return self
```

Using it is direct — you choose *k*, fit, and read off the cluster of any point:

```
from patentml import ScalableKMeans

km = ScalableKMeans(k=3, random_seed=1)
km.fit(points)
print("cluster assignments:", km.predict(points))
print("tightness (inertia):", km.inertia(points)) # lower = tighter
clusters
```

The inertia — the total squared distance from points to their

centroids — is your measure of how tight the clustering is, and plotting it against different values of k is how you find the natural number of groups. With three well-separated blobs and $k=3$, the centroids settle into their hearts in a few iterations, and every point is sorted, with no labels ever supplied.

THE INHERITANCE

Reach for k -means first, almost always, when you want to find groups in unlabelled numeric data. It is the default for a reason: fast, simple, scalable, and good enough for a vast range of real problems — customer segmentation, grouping documents or images, compressing colours, summarising data into representative prototypes. When someone hands you a pile of points and asks "what natural groups are in here?", k -means is the right first answer and frequently the last one you need.

Know its limits, because the rest of this part exists to address them: k -means assumes clusters are roughly round and similar in size (it carves space by distance to a centre), it needs you to choose k , and it has no concept of an outlier — every point gets assigned to some cluster, like it or not. The chapters ahead relax each of these assumptions in turn.

The public-domain, dependency-free form is a natural fit, because k -means is genuinely small — two steps and a convergence check —

and the mini-batch scalability means even the dependency-free version handles large data gracefully. You can cluster on an embedded device, segment data inside an application with no scientific-Python stack, or quantise an image's colours in a browser. Microsoft patented scalable k-means in 1998 to cluster databases too large for memory; the patent has expired; the most-used unsupervised algorithm in the world is now yours to read and run.

The next chapter takes on k-means' first great weakness — its assumption that clusters are round, and its refusal to recognise an outlier — with a method that follows density instead of distance, and that explicitly calls noise by its name.

CHAPTER 3

CLUSTERS OF ANY SHAPE (DBSCAN & OPTICS)

THE FILING

US Patent 6,049,797 — "Method, apparatus and programmed medium for clustering databases" by density-based detection with arbitrary cluster shape and explicit noise identification.

Assignee: Lucent Technologies. Filed 1998. 154 forward citations. Now expired; public domain.

The previous chapter's k-means has two stubborn blind spots: it assumes clusters are roughly round, and it forces every point into a cluster, with no notion that some points might just be noise.

DBSCAN — Density-Based Spatial Clustering of Applications with Noise — was designed to fix both, and Lucent's patent captures exactly what makes it special: it detects clusters of arbitrary shape, and it explicitly identifies noise points as outliers rather than

pretending they belong somewhere. Where k-means carves space into round territories around centres, DBSCAN follows the density of the data, tracing out clusters of whatever wandering shape they happen to take — long crescents, nested rings, tangled spirals — and leaving the sparse stragglers unassigned, named for what they are: noise.

THE IDEA

DBSCAN's governing intuition is that a cluster is a dense region of points, and clusters are separated by sparse regions. It makes this precise with two parameters and a simple notion of neighbourhood. You choose a radius, *eps*, and a count, *min_pts*. A point is a core point if it has at least *min_pts* neighbours within distance *eps* — that is, if it sits in a crowded place. The algorithm then grows clusters outward from core points: starting at any core point, it gathers all the points in its dense neighbourhood, and if any of those are themselves core points, it gathers their neighbourhoods too, chaining outward through the dense region until the density runs out. Everything reachable through this chain of dense neighbourhoods forms one cluster.

This chaining is what lets DBSCAN find clusters of any shape. k-means asks "which centre is this point closest to?" and so can only draw round territories. DBSCAN asks "is this point connected to that one through a path of dense neighbourhoods?" — and a path can

bend any way it likes. Two points at opposite ends of a long, curving filament belong to the same cluster as long as there is a continuous dense trail between them, even though they may be far apart and nowhere near any single centre. Round clusters, crescent clusters, spiral clusters, clusters wrapped around other clusters — DBSCAN handles them all, because it follows connectivity, not distance to a centroid.

And the points that are not densely connected to anything — the lonely stragglers in the sparse gaps — are labelled noise. This is a feature, not a failure. In the real world, data is full of outliers: measurement errors, rare events, genuine anomalies. k-means is forced to shove each one into the nearest cluster, distorting that cluster's shape and centre. DBSCAN simply sets them aside and says "these don't belong to any group" — which is both more honest and, often, exactly the outliers you were looking for. The two parameters also free you from k-means' most awkward demand: you do not specify the number of clusters in advance. DBSCAN discovers however many dense regions exist, whether that turns out to be two or two hundred. The companion algorithm OPTICS, also in this module and traceable to the same density-based lineage, extends the idea to handle clusters of varying density, which plain DBSCAN's single radius cannot.

THE STORY

DBSCAN was published in 1996 by Ester, Kriegel, Sander, and Xu, and it became one of the most influential and widely cited clustering algorithms ever devised — it later won a "test of time" award for its lasting impact, a recognition that its core idea had proven durable across two decades. Lucent's 1998 patent, cited over 150 times, is part of the same wave of late-1990s work on making clustering practical for large, messy, real-world databases, where the clean round blobs of textbooks give way to arbitrary shapes riddled with noise.

The reason DBSCAN endured is that real data is rarely shaped like k-means wants it to be. Geographic data, image segments, sensor readings, the structure of social and biological networks — these form clusters of irregular shape, of differing sizes, surrounded by genuine outliers. DBSCAN handles all of that gracefully and tells you about the outliers besides, which is why it remains a first-choice tool wherever cluster shape is unknown and noise is expected: spatial analysis, image processing, anomaly detection, and the exploratory clustering of any dataset whose structure you cannot assume in advance.

THE CODE

The main loop embodies the density rule directly: visit each point, check whether it is a core point (enough neighbours within ϵ), and if so, grow a cluster outward from it; otherwise mark it noise:

```
def fit(self, X):
    labels = [UNVISITED] * len(X)
    cluster_id = 0
    for i in range(len(X)):
        if labels[i] != UNVISITED:
            continue
        neighbours = self._region_query(X, i)      # everyone within
eps of point i
        if len(neighbours) < self.min_pts:
            labels[i] = NOISE                      # too sparse — it's an
outlier
        else:
            self._expand_cluster(X, labels, i, neighbours, cluster_id) #
grow from here
            cluster_id += 1
        self.labels_ = labels
    return self
```

The neighbourhood query is the heart of "density" — simply, who is within the radius:

```
def _region_query(self, X, idx):
    return [j for j in range(len(X)) if _euclidean(X[idx], X[j]) <=
self.eps]
```

And cluster expansion is the chaining: absorb a core point's

neighbours, and when any neighbour is itself a core point, absorb its neighbourhood too, rippling outward through the dense region until the density gives out. Using it requires no guess at the number of clusters — only a sense of scale and density:

```
from sklearn import DBSCAN

db = DBSCAN(eps=0.5, min_pts=5)
db.fit(points)

n_clusters = len(set(l for l in db.labels_ if l >= 0))
n_noise = sum(1 for l in db.labels_ if l == -1)
print(f"found {n_clusters} clusters and flagged {n_noise} noise
points")
```

Run this on data shaped like two interlocking crescents — the classic example where k-means fails badly, slicing each crescent in half — and DBSCAN traces each crescent correctly as a single cluster, following the dense curve all the way around, while marking any scattered stray points as noise. It discovered the number of clusters on its own, respected their shape, and refused to pretend the outliers belonged.

THE INHERITANCE

Reach for DBSCAN when you don't know the shape of your clusters,

you don't know how many there are, and you expect outliers — which describes a great deal of real exploratory data analysis. It is the natural complement to k-means: where k-means is fast and assumes round, equal clusters, DBSCAN is the tool for irregular structure and honest noise-handling. Spatial data, image regions, and any setting where "the clusters might be weird shapes and some points are just junk" make DBSCAN the right choice.

Its outlier-flagging is a genuine bonus that doubles as an anomaly detector: the points DBSCAN calls noise are precisely the ones that don't fit any dense pattern, which is often exactly what you are hunting — a theme this volume returns to in its chapter on anomaly detection. The practical craft is choosing `eps` and `min_pts`, which set the scale of "dense"; the OPTICS extension in the same module eases this by handling clusters of varying density.

The public-domain, dependency-free form is well suited to DBSCAN, because the algorithm is fundamentally simple — neighbourhood queries and a flood-fill through dense regions — and it carries no assumptions about cluster count or shape that you have to encode. You can run shape-agnostic, noise-aware clustering on a small device, inside an analysis tool with no scientific stack, or in a browser. Lucent patented density-based clustering with explicit noise detection in 1998 and earned over 150 citations as the idea proved its staying power; the patent has expired; the clustering method that follows density wherever it leads is now yours.

The next chapter takes a different tack again: instead of carving space or chasing density, it assumes the data was generated by a mixture of hidden bell curves and works backward to uncover them — soft, probabilistic clustering by expectation-maximisation.

CHAPTER 4

THE HIDDEN MIXTURE (EM AND GAUSSIAN MIXTURES)

THE FILING

US Patent 6,263,337 — "Scalable system for expectation maximization clustering of large databases"

Assignee: Microsoft Corporation. Filed 1998. 142 forward citations. Now expired; public domain.

The two clustering methods so far have made hard assignments: k-means puts each point in exactly one cluster, DBSCAN in exactly one cluster or none. But the world is often fuzzier than that. A customer might be 70% "bargain hunter" and 30% "brand loyalist." A data point near the boundary between two groups might genuinely, partly, belong to both. Expectation-Maximisation clustering with Gaussian mixtures embraces this fuzziness. It assumes the data was generated by a handful of overlapping bell-shaped distributions, and

it works backward to recover them — giving each point not a single label but a set of probabilities of belonging to each cluster.

Microsoft's patent, like the k-means one before it, is about making this scalable: processing data in chunks and compressing the statistics it needs so that the full dataset never has to live in memory at once.

THE IDEA

Imagine that your data was produced by a recipe: nature picked one of k "true" groups, each shaped like a bell curve (a Gaussian) with its own centre and spread, then drew a point from that group's bell. It did this many times, and handed you the resulting cloud of points — but threw away the record of which group each point came from. Your job is to reconstruct the recipe: find the k bell curves (their centres, their spreads, and how often each was chosen) that best explain the data you see. This is a Gaussian mixture model, and it is a genuinely different way of thinking about clustering — not "carve the space" but "what underlying distributions generated this?"

The difficulty is circular. If you knew which bell curve each point came from, estimating the curves would be easy (just measure the centre and spread of each group). And if you knew the curves, deciding which one each point probably came from would be easy. But you know neither. Expectation-Maximisation (EM) breaks the circle by alternating between the two, exactly like k-means but

softer:

1. E-step (Expectation). Given the current guess at the bell curves, compute for each point the probability that it came from each curve. These probabilities are called responsibilities — curve A is "responsible" for 80% of this point, curve B for 20%. Unlike k-means, no point is forced to choose; it is shared, in proportion, among the curves.

2. M-step (Maximisation). Given those soft responsibilities, re-estimate each bell curve — its centre is the responsibility-weighted average of all points, its spread the responsibility-weighted variance, its overall weight the total responsibility it claimed. Points that mostly belong to curve A pull curve A toward themselves, in proportion to how much they belong.

Repeat, and the curves slide and stretch to settle over the natural groups in the data, while the responsibilities sharpen. The quantity EM steadily improves is the log-likelihood — a measure of how well the current mixture explains the observed data — and it is guaranteed never to decrease, which is how you know the algorithm is converging. The result is both a soft clustering (each point's probabilities of membership) and a generative model (a recipe you could use to produce new, similar data). k-means, it turns out, is just the hard-edged special case of this: force every responsibility to be 0 or 1 and assume every bell curve is a perfect circle of the same size,

and EM collapses back into k-means.

THE STORY

Expectation-Maximisation was given its general formulation and its name by Dempster, Laird, and Rubin in 1977, in one of the most-cited papers in all of statistics. It is far more than a clustering method — EM is a general technique for fitting models with hidden variables, which is why its fingerprints are all over this book: the hidden Markov models of the next chapter are trained by a form of EM, and the "infer the hidden thing that generated the visible thing" pattern recurs throughout unsupervised learning. Gaussian-mixture clustering is its most intuitive application.

Microsoft's 1998 patent, like its k-means sibling, belongs to the late-1990s push to make these classic statistical methods survive contact with databases too large for memory. The scalable EM it describes reads the data in mini-batches and accumulates compact sufficient statistics — the running totals (how much responsibility, weighted sums, weighted squared sums) that are all the M-step actually needs — so the algorithm can fit a mixture to a dataset it never holds in full. Gaussian mixtures remain widely used wherever soft, probabilistic clustering is wanted: speaker and image modelling, density estimation, anomaly detection (a point the mixture explains poorly is a likely outlier), and as a building block in larger probabilistic systems. The soft, generative view of clustering

is one of the most powerful ideas in unsupervised learning, and it is now public domain.

THE CODE

The two steps are the algorithm, and `patentml` writes them plainly. The E-step computes each point's responsibilities — its probability of belonging to each Gaussian — working in log-space for numerical safety, and accumulates the log-likelihood that measures fit:

```
def _e_step(self, X, comps):
    """For each point, the probability it came from each Gaussian (its
    'responsibilities')."""
    resp = []
    ll_total = 0.0
    for x in X:
        log_probs = [math.log(max(c.weight, 1e-12)) + _log_prob(x, c)
        for c in comps]
        log_sum = _log_sum_exp(log_probs)    # how well the
        whole mixture explains x
        ll_total += log_sum
        probs = [math.exp(lp - log_sum) for lp in log_probs] #
        normalise to responsibilities
        resp.append(probs)
    return resp, ll_total / len(X)
```

The M-step (in the same module) then re-estimates each Gaussian's centre, spread, and weight from those responsibilities — every point contributing to every curve in proportion to how much it belongs. The fit loop alternates them until the log-likelihood stops improving, and runs several times from different starts to avoid poor local optima:

```
def fit(self, X):
    for trial in range(self.n_init):          # several restarts, keep the
best
        comps = self._init_components(X, rng) # k-means++ style
initial Gaussians
        ll_prev = float('-inf')
        for _ in range(self.max_iter):
            resp, ll = self._e_step(X, comps) # E-step: responsibilities
            comps = self._m_step(X, resp, comps) # M-step: re-fit the
Gaussians
            if abs(ll - ll_prev) < self.tol:  # converged — likelihood
stopped rising
                break
            ll_prev = ll
        return self
```

In use, the difference from k-means is the soft assignment — you can ask not just which cluster a point is in, but how strongly it belongs to each:

```
from patentml import EMClustering

gmm = EMClustering(n_clusters=3, seed=1)
gmm.fit(points)
```

Each component is a fitted bell curve: its centre, spread, and mixing weight.

```
for c in gmm.components:
    print(f"centre {[round(m, 2) for m in c.mean]} weight
{c.weight:.2f}")
```

Where two clusters overlap, EM does not draw a hard line through the contested region; it shares the boundary points between the curves in proportion to the evidence, and it hands you back the actual bell curves that best explain your data — a model, not just an assignment.

THE INHERITANCE

Reach for Gaussian-mixture EM when you want soft, probabilistic clustering — when a point can reasonably belong to more than one group and you want to know the proportions, rather than forcing a hard choice. Reach for it, too, when you want a generative model of your data: a mixture you can use to estimate the density anywhere, to score how typical or anomalous a new point is, or to generate synthetic examples. And reach for it when your clusters are not

round and equal-sized — because EM fits each Gaussian's own spread, it handles elongated and differently-sized clusters that plain k-means distorts.

The deeper inheritance is conceptual: EM is the master pattern for learning when something is hidden, and understanding it here — alternate between guessing the hidden assignments and refitting the model — is understanding the engine behind the hidden Markov models of the next chapter and much of probabilistic machine learning besides.

The public-domain, dependency-free form carries the same gift as the rest of this part: a soft, generative clustering method small enough to read in full — the E-step and M-step right there, the log-likelihood climbing — and to run anywhere, on a device or in a browser, with the scalability that lets it handle data larger than memory. Microsoft patented scalable EM clustering in 1998 and earned 142 citations; the patent has expired; the method that clusters by uncovering hidden bell curves is now yours.

The next chapter clusters in a way that looks nothing like the three so far: it builds a graph of which points are similar, and finds clusters in the mathematics of that graph's connectivity — spectral clustering.

CHAPTER 5

CUTTING THE GRAPH (SPECTRAL CLUSTERING)

THE FILING

US Patent 6,216,066 — "System and method for generating alerts through multi-variate data analysis", using Laplacian eigenspace embedding to find structure in high-dimensional data.

Assignee: General Electric. Filed 1998. 148 forward citations. Now expired; public domain.

The three clustering methods so far have worked in the original space of the data — measuring distances, following density, fitting bells. Spectral clustering does something stranger and, for certain problems, far more powerful: it throws away the original coordinates entirely, builds a graph of which points are similar to which, and then finds clusters in the mathematics of that graph's connectivity — specifically, in the eigenvectors of a matrix derived from it. GE's

patent, aimed at finding structure in high-dimensional, multi-variate data, uses exactly this Laplacian-eigenspace trick. The pay-off is that spectral clustering can separate groups that are hopelessly tangled in the original space — concentric rings, interlocking shapes — that defeat k-means completely, because it asks not "how far apart are these points?" but "how connected are they?"

THE IDEA

Forget coordinates for a moment and think of your data as a graph. Each point is a node. Between every pair of points you draw an edge whose weight reflects how similar they are — close points get a strong edge, distant points a weak one or none. A common choice is the RBF affinity, where the edge weight falls off exponentially with distance, so near neighbours are strongly linked and far points are effectively disconnected. Now your dataset is a web of connections, and the clustering question becomes a question about the web: how do you cut this graph into pieces, slicing as few strong edges as possible? A good cluster is a clump of nodes densely connected to each other but only weakly connected to the rest — and the best clustering is the one that severs only weak edges to separate the clumps.

Finding that best cut directly is computationally hard, but here is the beautiful trick at the heart of spectral methods. You build a matrix called the graph Laplacian from the affinities, and you compute its

eigenvectors — particular characteristic vectors that capture the graph's fundamental "modes," much as the resonant frequencies of a drum capture its shape (this acoustic analogy is why the method is called spectral). It turns out that the eigenvectors corresponding to the smallest eigenvalues of the Laplacian encode exactly the graph's cluster structure: they map each node to a new set of coordinates in which the tangled clusters of the original space become cleanly separated, roughly round blobs. Then — and this is the elegant punchline — you run ordinary k-means (Chapter 1) in that new eigenvector space, where it works perfectly, because the hard part has already been done by the graph.

So spectral clustering is a transformation followed by a simple clustering: turn the data into a similarity graph, use the graph's eigenvectors to re-embed the points into a space where clusters are obvious, and cluster there. The reason it succeeds where k-means fails is that two points on opposite ends of a long curving cluster are far apart in the original space (so k-means splits them) but strongly connected through a chain of similar neighbours in the graph (so spectral clustering keeps them together). It clusters by connectivity, like DBSCAN, but expresses that connectivity through the powerful and general language of linear algebra — which also lets it cluster things that are natively graphs, like social or biological networks, where there are no original coordinates at all.

THE STORY

Spectral clustering grew out of a confluence of ideas in the late 1990s and early 2000s — graph-partitioning methods from scientific computing, the "normalised cut" criterion from computer vision (Shi and Malik's influential image-segmentation work), and connections to random walks on graphs. It became a celebrated technique precisely because it could crack problems that stumped every distance-based method: data lying on curved manifolds, clusters wrapped around one another, and the clustering of network data where the relationships are the data. GE's 1998 patent, cited nearly 150 times, applies the Laplacian-eigenspace embedding to anomaly and structure detection in multi-variate industrial data — finding patterns in high-dimensional measurements where the structure is invisible to simpler methods.

The deeper significance of spectral clustering is that it was an early and influential member of a family now central to machine learning: methods that find a better representation of data before analysing it. The idea that you should first re-embed your data into a space where the structure becomes simple — and only then apply a basic algorithm — runs straight through to the manifold-learning and representation-learning techniques that underpin much of modern AI. Spectral clustering is also the bridge to graph machine learning, the increasingly important field of learning directly on networks. GE's patent has now expired, and the eigenvector trick that untangles the tangled is public domain.

THE CODE

The first step is to turn points into a similarity graph. The RBF affinity makes near points strongly linked and far points nearly disconnected:

```
def rbf_affinity(X, gamma=1.0):
    """Edge weight  $W[i, j] = \exp(-\gamma \|x_i - x_j\|^2)$ 
    strong, far! weak."""
    n = len(X)
    W = [[0.0] * n for _ in range(n)]
    for i in range(n):
        for j in range(i + 1, n):
            d2 = sum((X[i][k] - X[j][k]) ** 2 for k in range(len(X[0])))
            W[i][j] = W[j][i] = math.exp(-gamma * d2)
    return W
```

From the affinity graph you build the normalised Laplacian — the matrix whose eigenvectors encode the cluster structure:

```
def _normalised_laplacian(W):
    """ $L_{\text{sym}} = I - D^{-1/2} W D^{-1/2}$  — its smallest eigenvectors
    reveal the clusters."""
    d = [sum(W[i]) for i in range(len(W))]      # each node's total
    connectivity
    dsq = [1.0 / math.sqrt(max(di, 1e-12)) for di in d]
```

```
L = [[(1.0 if i == j else 0.0) - dsq[i] W[i][j] dsq[j]]
      for j in range(len(W))] for i in range(len(W))]
return L
```

The implementation then extracts the smallest eigenvectors (by power iteration on a deflated matrix — no external linear-algebra library needed, true to the dependency-free promise) and runs k-means in that eigenvector embedding. From the user's side, all of this is hidden behind a familiar interface:

```
from patentml import SpectralClustering

sc = SpectralClustering(n_clusters=2, affinity="rbf", gamma=1.0)
labels = sc.fit_predict(points)
print("cluster labels:", labels)
```

Hand this the classic "two concentric rings" dataset — an inner circle of points surrounded by an outer ring — and k-means fails utterly, slicing both rings down the middle into two half-moons, because the rings interleave in distance. Spectral clustering separates them perfectly: in the graph, each ring is a connected loop only weakly linked to the other, so the Laplacian's eigenvectors map the two rings to two tidy, separable blobs, and the final k-means step cleanly divides them. It solved by connectivity what no distance-based method could.

THE INHERITANCE

Reach for spectral clustering when your clusters are tangled in the original space — concentric, interlocking, curved, or lying on a twisted manifold — and distance-based methods like k-means visibly fail. It is the specialist for structure that connectivity reveals but distance hides. Reach for it, too, when your data is a graph — a social network, a citation network, a biological interaction network — because spectral clustering works natively on relationships and needs no coordinates at all.

The honest cost is scale: building and finding the eigenvectors of an affinity matrix over n points is expensive for very large n , so spectral clustering is at its best on small-to-medium datasets where its unique power justifies the cost (techniques exist to approximate it for larger data). And it shares k-means' need for you to choose the number of clusters.

The public-domain, dependency-free form is a small marvel here, because spectral clustering is usually assumed to require a heavyweight linear-algebra library for its eigenvectors — and `patentml` computes them in pure Python by power iteration, with no dependency at all. That makes a genuinely sophisticated, graph-and-eigenvector-based clustering method available where no scientific stack exists, and it lays the eigenvector trick bare for anyone wanting to understand one of machine learning's most

elegant ideas. GE patented the Laplacian-eigenspace approach in 1998 and earned nearly 150 citations; the patent has expired; the method that clusters by cutting graphs is now yours.

The final chapter of this part returns to the original space one more time, with a method that needs neither a chosen number of clusters nor an assumed shape — it simply lets every point climb uphill to the nearest peak of density, and groups together everything that climbs to the same summit.

CHAPTER 6

CLIMBING TO THE PEAK (MEAN SHIFT & KERNEL DENSITY ESTIMATION)

THE FILING

US Patent 6,317,707 — "Automatic clustering of tokens from a corpus for grammar acquisition", by mode-seeking density estimation in which each point shifts iteratively toward the weighted mean of its neighbours until it converges at a density peak.

Assignee: AT&T Corporation. Filed 1998. 269 forward citations.

Now expired; public domain.

This chapter's patent has the highest citation count in this part of the book — 269 — and it comes from AT&T's language research, where the problem was clustering words from a text corpus to help a machine learn grammar. The algorithm it rests on, mean shift, has a wonderfully physical intuition: imagine the data's density as a hilly landscape, with peaks where points are crowded and valleys where

they are sparse. Mean shift lets every point climb uphill to the nearest peak, and then declares that all the points which climbed to the same peak form a cluster. It needs no number of clusters specified in advance — the number of peaks decides that — and it makes no assumption about cluster shape. It simply follows the density to its summits.

THE IDEA

Underlying mean shift is the idea of kernel density estimation, which is worth understanding on its own because it appears throughout statistics. Given a scatter of points, how dense is the data at any particular location? KDE answers by placing a little smooth bump — a kernel, usually a small Gaussian — on top of every data point, and adding all the bumps together. Where points are crowded, the bumps pile up into a high ridge; where points are sparse, the summed bumps are low. The result is a smooth landscape of density over the whole space, estimated from the points alone. The width of each bump is the bandwidth, the one crucial parameter: a wide bandwidth gives a smooth landscape with few broad peaks, a narrow bandwidth a rugged landscape with many sharp ones.

Mean shift turns this density landscape into a clustering method by climbing it. Take any point and ask: in which direction does the density increase fastest? Remarkably, you do not need calculus to find out — the uphill direction is simply toward the weighted

average position of the neighbours, where nearby points count more (through the kernel) than distant ones. So you shift the point to that weighted-mean location. From the new spot, you compute the weighted mean again and shift once more. Each shift moves the point toward where its neighbourhood is densest, and repeating the shift walks it steadily uphill until it can climb no further — it has reached a mode, a peak of density, and stops. This is the "mean shift" the patent names: shift to the mean, again and again, until you converge on a summit.

Now do this for every point. Each one climbs to whichever peak sits above it. Points that climb to the same peak — that share a mode — belong to the same cluster. Because the number of clusters equals the number of peaks the points discover, you never specify it; the density decides. And because a peak can sit atop a cluster of any shape, mean shift makes no assumption about cluster geometry — it handles elongated, irregular groups as easily as round ones. The bandwidth is the only real dial, and it implicitly controls the number of clusters: smaller bandwidth, more peaks, more clusters. After every point has climbed, nearby summits (the same peak found from different starting points, landing fractionally apart) are merged, and each original point is labelled by the merged peak it reached.

THE STORY

Mean shift originated as a statistical procedure in the work of

Fukunaga and Hostetler in 1975, but it was rediscovered and popularised for machine learning and computer vision around the turn of the millennium, when Comaniciu and Meer showed it to be a powerful, general tool for clustering and especially for image segmentation — grouping the pixels of an image into coherent regions by clustering them in a combined space of colour and position. AT&T's 1998 patent, with its heavy citation count, applies the same mode-seeking idea to language, clustering words by the company they keep — an early instance of the distributional thinking that would later flower into the word embeddings of this volume's chapter on meaning.

Mean shift earned its lasting place because of what it does not require. In a field where most clustering methods make you commit upfront — k-means to a number of clusters, Gaussian mixtures to a number and a shape — mean shift asks only for a bandwidth and then discovers the rest from the data's own density. That made it a favourite for problems where the number and shape of clusters are genuinely unknown: image and video segmentation, object tracking (following a moving target by chasing the mode of its colour distribution frame to frame), and exploratory analysis. Its cost is computational — every point climbing through the density is more expensive than a few k-means sweeps — but when you cannot assume how many groups exist or what they look like, mean shift's willingness to let the data answer those questions itself is exactly what you want.

THE CODE

The climb is the heart of it: shift a point repeatedly toward the kernel-weighted mean of all points, until it stops moving. That weighted mean is the local uphill direction:

```
def _shift_to_mode(self, x):
    """Climb uphill: repeatedly move toward the kernel-weighted
    mean of the neighbours."""
    point = x[:]
    for _ in range(self.max_iter):
        weights = [self._kernel_fn(_euclidean(point, xi),
self.bandwidth)
                    for xi in self._X]          # near points weigh more
        total_w = sum(weights)
        new_point = [sum(weights[i] * self._X[i][j] for i in
range(len(self._X))) / total_w
                    for j in range(len(point))]  # the weighted mean = one
step uphill
        if _euclidean(new_point, point) < self.tol: # reached a peak —
stop climbing
            break
        point = new_point
    return point
```

Clustering is then: climb from every point, merge the peaks that

coincide, and label each point by the peak it reached:

```
def fit(self, X):
    modes = [self._shift_to_mode(x) for x in X]    # every point
    climbs to a peak
    centers = self._merge_modes(modes)            # peaks found in
    common place merge
    self.cluster_centers_ = centers
    self.labels_ = [min(range(len(centers)), key=lambda ci:
    _euclidean(x, centers[ci]))
                    for x in X]                  # label by nearest peak
    return self
```

Using it, you supply only a bandwidth — never a cluster count — and the algorithm reports how many groups it found:

```
from patentml import MeanShift, estimate_bandwidth

bw = estimate_bandwidth(points)    # a reasonable bandwidth,
inferred from the data
ms = MeanShift(bandwidth=bw)
ms.fit(points)

print("clusters discovered:", len(ms.cluster_centers_)) # the data
decided this number
print("cluster centres:", ms.cluster_centers_)
```

You never told it how many clusters to expect. Every point climbed the density landscape to a peak, the coincident peaks merged, and the number of distinct summits became the number of clusters — discovered, not dictated. The `estimate_bandwidth` helper even suggests a sensible bandwidth from the data's own scale, leaving you with essentially nothing to tune.

THE INHERITANCE

Reach for mean shift when you do not know how many clusters there are and refuse to guess — when the number of groups is itself something you want the data to tell you. That single property, discovering the cluster count from the density rather than requiring it as input, sets mean shift apart from almost every other method in this part and makes it a natural choice for genuinely exploratory analysis. Reach for it, too, when clusters may be irregularly shaped, since a density peak can crown a group of any geometry, and for the classic application of image segmentation, where it groups pixels into coherent regions without being told how many regions to expect.

Its companion, kernel density estimation, is a valuable tool in its own right whenever you need a smooth estimate of how dense data is at any location — for visualisation, for spotting outliers (low-density points), or as a building block in larger systems. Both are in the same module. The honest cost, as ever, is speed: the climb is more expensive than k-means, so mean shift suits small-to-medium data

where its assumption-free flexibility earns the price.

The public-domain, dependency-free form fits mean shift naturally, because the algorithm is conceptually and literally simple — repeatedly move toward a weighted mean — with no number of clusters to configure and no shape to assume. You can run mode-seeking clustering and kernel density estimation on a small device, inside an image tool with no scientific stack, or in a browser, and the readable climb makes the elegant "follow the density uphill" idea transparent. AT&T patented mode-seeking density clustering in 1998 and earned 269 citations applying it to language; the patent has expired; the method that lets every point climb to its peak is now yours.

That completes *The Pattern* — five ways to find structure in data that no one labelled, from k-means' fast round territories to mean shift's density summits. Part II changes the question from space to time: what do you do when your data arrives in sequence, and the truth you care about — a position, a hidden state — is never observed directly, but must be inferred from a stream of noisy clues?

PART II — THE SEQUENCE: TIME, STATE, AND MEMORY

Data that arrives in order carries a special difficulty: the thing you actually care about is often never observed directly. A drone's true position, a spoken word behind a noisy waveform, a machine's hidden condition behind its sensor readings — none of these are handed to you. You see only a stream of noisy, partial clues, and you must infer the hidden truth that produced them, updating your belief with every new observation. The two chapters of this part are the great classical answers to that problem. The Kalman filter tracks a continuous hidden state through noise; the hidden Markov model infers a sequence of discrete hidden states from a sequence of visible symbols. Both were patented for high-stakes real-world use, and both are now public domain. We begin with the algorithm that quietly steadies almost everything that moves.

CHAPTER 7

THE FILTER THAT TRACKS (KALMAN FILTERS)

THE FILING

US Patent 6,795,794 — "Method and apparatus" embodying Kalman and extended-Kalman state estimation.

Assignee: University of Illinois. Filed 2002. 127 forward citations.
Now expired; public domain.

The Kalman filter may be the most quietly consequential algorithm in either volume of this book. It is inside the device you used to navigate this morning, the aircraft and spacecraft that fly, the drones that hover, the robots that walk, and countless control systems that hold something steady against noise and disturbance. Its job is deceptively simple to state: given a stream of noisy measurements of something that is moving according to known rules, produce the best possible running estimate of its true state. The University of Illinois

patent covers the filter and its nonlinear extension; but the Kalman filter's real lineage runs back to the Apollo program, where it was used to guide spacecraft to the Moon. Few algorithms are at once so mathematically elegant and so embedded in the physical machinery of the world — and the version here is pure, dependency-free Python.

THE IDEA

You are tracking something — a vehicle's position, say — that changes over time according to rules you roughly know (it moves, with some velocity, perhaps with some acceleration). You cannot observe its true state directly; you only get measurements, and those measurements are noisy — every GPS fix, every sensor reading, jitters around the truth. Worse, your model of how the thing moves is also imperfect — winds gust, wheels slip. You have two sources of information, both flawed: a prediction from your motion model, and a measurement from your sensor. The Kalman filter is the optimal way to fuse them.

It works in a perpetual two-step rhythm, repeated at every tick of the clock:

1. Predict. Using your model of how the state evolves, project the current estimate forward in time. "If it was here, moving this fast, then a moment later it should be there." Crucially, the filter also

propagates its uncertainty — and because the motion model is imperfect, the uncertainty grows during prediction. The filter becomes less sure the longer it goes without a measurement.

2. Update. Now a measurement arrives. The filter compares it to what it predicted, and corrects its estimate toward the measurement — but only partly, by an amount called the Kalman gain. This is the genius of the method: the gain is computed from the relative uncertainties. If the sensor is very noisy and the prediction is confident, the filter trusts the prediction and barely moves toward the measurement. If the sensor is precise and the prediction shaky, it leans hard toward the measurement. It optimally weighs the two flawed sources by exactly how much each can be trusted, and in doing so its uncertainty shrinks — a measurement makes it more sure.

Predict (uncertainty grows), update (uncertainty shrinks), forever. The filter glides along, always holding the best possible estimate of a truth it can never see directly, automatically discounting noise and filling the gaps between measurements with its model. The whole thing is expressed in the language of matrices — a state transition matrix F for how the state evolves, an observation matrix H for how measurements relate to the state, and noise covariances Q and R that encode how much to distrust the model and the sensor respectively. When the system's dynamics are nonlinear — as most real motion is — the Extended Kalman Filter (also in this module) handles it by

linearising around the current estimate at each step, which is how the technique reaches into robotics, aerospace, and navigation.

THE STORY

The filter is named for Rudolf Kálmán, who published it in 1960, though the ideas have older roots. Its fame was sealed almost immediately and dramatically: NASA adopted it for the Apollo guidance computer, and the Kalman filter helped navigate the spacecraft that carried humans to the Moon and back. That origin story is not incidental — it captures exactly what the filter is for: estimating where you truly are when every measurement is noisy and the stakes are absolute. From there it spread into essentially every domain that involves tracking, navigation, or control. The satellite-navigation receiver in your pocket runs a Kalman filter to fuse satellite signals into a position. Aircraft and missiles use it for guidance. Robots use it to estimate their location and the state of the world (the famous SLAM problem — simultaneous localisation and mapping — leans on Kalman-family filters). Economists use it to track hidden economic states behind noisy indicators. The University of Illinois patent, cited 127 times, is part of the long industrial tail of refining and applying this foundational estimator.

What makes the Kalman filter beloved by engineers is that it is optimal (under its assumptions of linear dynamics and Gaussian noise, no estimator does better), recursive (it needs only the previous

estimate and the new measurement, not the whole history — so it runs forever in constant memory, perfect for real-time and embedded use), and principled about uncertainty (it always knows how sure it is). It is the gold standard against which every tracking method is measured.

THE CODE

The patentml Kalman filter is built from the matrices that define the system, and its two steps are exactly predict and update. The predict step propagates the state and grows the uncertainty by the process noise:

```
def predict(self, u=None, B=None):
    """Project the state forward; uncertainty grows by the process
    noise Q."""
    self.x = _mm(self.F, self.x)          # evolve the state
    estimate
    self.P = _add_m(_mm(_mm(self.F, self.P), _T(self.F)), self.Q) #
    uncertainty grows
    return self
```

The update step computes the Kalman gain from the relative uncertainties, corrects the estimate toward the measurement, and shrinks the uncertainty:

```

def update(self, z):
    """Fuse in a measurement z, weighting it against the prediction by
    the Kalman gain."""
    S = _add_m(_mm(_mm(self.H, self.P), _T(self.H)), self.R) #
    innovation covariance
    K = _mm(_mm(self.P, _T(self.H)), _cholesky_inv(S))      # the
    Kalman gain
    y = _sub_m(z, _mm(self.H, self.x))                      # measurement
    vs prediction
    self.x = _add_m(self.x, _mm(K, y))                      # correct
    toward measurement
    self.P = _mm(_sub_m(_eye(self.n), _mm(K, self.H)), self.P) #
    uncertainty shrinks
    return self

```

The filter method simply runs predict-then-update across a whole stream of observations. Tracking a noisy one-dimensional level — a constant truth buried in sensor noise — shows the filter at its simplest:

```
from patentml import KalmanFilter
```

A constant hidden value, measured noisily. $F=[[1]]$ (it stays put), $H=[[1]]$ (we measure it).

```
kf = KalmanFilter(F=[[1.0]], H=[[1.0]], Q=[[1e-4]], R=[[1.0]])
```

```
truth = 5.0
noisy = [truth + random.gauss(0, 1.0) for _ in range(50)] # 50 jittery
readings
estimates = kf.filter(noisy)

print(f"true value: {truth}")
print(f"final estimate: {estimates[-1][0]:.3f}") # close to 5.0, far
steadier than any reading
```

Each individual reading jitters a whole unit around the truth, but the filter, fusing them through its predict-update rhythm and trusting its confident model over the noisy sensor, settles to a steady estimate far closer to the true value than any single measurement. That smoothing — extracting a stable truth from a noisy stream — is what the Kalman filter does in every device that relies on it.

THE INHERITANCE

Reach for a Kalman filter whenever you must track a continuous hidden state from noisy measurements in real time — position, velocity, orientation, temperature, any quantity that evolves by known-ish rules and is measured imperfectly. It is the default, optimal answer for navigation, guidance, sensor fusion, and control, and the Extended Kalman Filter extends that reach to the nonlinear dynamics of real robots and vehicles. If your problem is "I have a stream of jittery sensor data and I need the best running estimate of

what's really going on," the Kalman filter is very likely the right tool, and has been for sixty years.

Its defining practical virtues make it ideal for exactly the constrained settings this book keeps returning to: it is recursive and constant-memory (it keeps only its current estimate and uncertainty, never the history), so it runs forever in a fixed footprint — precisely what an embedded controller or a real-time system needs. And it is principled about its own uncertainty, which lets downstream systems reason about how much to trust it.

The public-domain, dependency-free form is genuinely valuable here, because the Kalman filter lives overwhelmingly on small hardware — flight controllers, navigation chips, robot microcontrollers — where a heavyweight library is impossible and a compact, readable, pure implementation is exactly right. `patentml`'s filter is the matrices and the two steps, nothing more, ready to drop into a device or to study until the elegant logic of predict-and-update is clear. The University of Illinois patented its Kalman estimator in 2002; the algorithm guided Apollo to the Moon; the patent has expired; and the filter that steadies the moving world is now yours.

The next chapter turns from continuous state to discrete hidden state — inferring a sequence of unseen categories (the words behind speech, the condition behind sensor patterns) from a sequence of visible symbols, with the hidden Markov model.

CHAPTER 8

THE HIDDEN CHAIN (HIDDEN MARKOV MODELS)

THE FILING

US Patent 7,076,102 — "Video monitoring" using hierarchical hidden-Markov-model event learning and classification.

Assignee: Philips. Filed 2002. 116 forward citations. Now expired; public domain.

The companion US Patent 6,219,453 (AT&T, 1997) covers automatic correction via a hidden-Markov-model algorithm. Both now expired; public domain.

The Kalman filter of the last chapter tracked a continuous hidden state. The hidden Markov model tracks a discrete one — a hidden state that takes one of a finite set of values, and that hops from one to another over time, emitting a visible symbol at each step. Philips'

patent uses it to watch video and recognise events: the visible stream is what the camera sees, the hidden states are the underlying activities, and the model infers the activities from the pixels. This is the HMM's signature: a chain of hidden states you cannot see, throwing off a chain of observations you can, and a set of beautiful algorithms for reasoning from the visible back to the hidden. For two decades the HMM was the dominant technology behind speech recognition — the hidden chain behind every word a machine understood.

THE IDEA

Picture a process that is always in one of a few hidden states, and that ticks from state to state over time with certain probabilities — from "sunny" it usually stays sunny but sometimes turns rainy. This hopping is a Markov chain: where it goes next depends only on where it is now, not on its whole history. The catch is that you cannot observe the state directly. Instead, each state, when occupied, emits a visible observation with certain probabilities — on a rainy day your friend is more likely to carry an umbrella. You see the umbrellas; you must infer the weather. That gap — visible emissions from hidden states — is the "hidden" in hidden Markov model.

A discrete HMM is fully described by three sets of probabilities: how likely each state is to start the sequence; the transition probabilities of hopping from each state to each other; and the

emission probabilities of each state producing each observable symbol. With those in hand, three classic questions can be answered, and the elegance of the HMM is that each has an efficient algorithm:

How likely is this sequence? Given a model and a stream of observations, what is the probability the model would produce that stream? The forward algorithm computes it efficiently, summing over all the hidden paths that could have generated the observations without ever enumerating them one by one. This is how you score a sequence against a model — the basis of classification (which model fits this data best?) and anomaly detection (does any model fit it at all?).

What was the hidden sequence? Given the observations, what is the single most likely sequence of hidden states that produced them? The Viterbi algorithm finds it, threading the most probable path through the lattice of possible state sequences. This is the decoding step — recover the words from the sounds, the activities from the video, the true characters from the noisy ones.

How do I learn the model from data? Given only sequences of observations, with the states completely hidden and the probabilities unknown, can you learn the transition and emission probabilities? The Baum-Welch algorithm does it — and it is, at heart, the expectation-maximisation of Chapter 3 applied to sequences: alternate between guessing the hidden state assignments (using

forward-backward probabilities) and re-estimating the model from those guesses, climbing the likelihood until it converges. The same "infer the hidden, refit the model, repeat" pattern, now threaded through time.

All of this is done in log-space in any serious implementation, because multiplying thousands of small probabilities along a sequence underflows to zero otherwise — a numerical discipline the code makes visible.

THE STORY

The hidden Markov model rose to prominence through speech recognition, where from the 1970s through the 2000s it was the dominant paradigm — the hidden states modelling the phonemes and words, the observations modelling the acoustic signal, Viterbi decoding the most likely utterance, Baum-Welch training the models from transcribed speech. For a generation, when a machine understood a spoken word, an HMM was almost certainly inside. The same machinery spread to any problem with a hidden sequence behind a visible one: part-of-speech tagging and other sequence labelling in language, gene-finding in computational biology (the hidden states modelling genes and non-coding regions along a DNA strand), gesture and activity recognition, and — as in Philips' patent — event detection in video. AT&T's patent applies it to automatic error correction, decoding the intended sequence from a corrupted

one.

The HMM's reign in speech was eventually ended by the deep recurrent and transformer networks of the modern era, which now dominate large-scale sequence tasks. But the HMM remains the right tool for a large class of problems, and an essential one to understand: it is interpretable (the states and transitions often mean something you can read), it trains from modest data, it runs cheaply, and its three algorithms are among the most elegant in machine learning. Where you have a discrete hidden process behind a discrete observable one — and especially where data is limited and interpretability matters — the HMM is alive and well. Philips' video-event patent has now expired into the public domain.

THE CODE

The forward algorithm is the workhorse — scoring a sequence by summing over hidden paths, in log-space for numerical safety:

```
def forward(self, obs):
    """Log P(observations | model), summed over all hidden state
    paths."""
    alpha = [[LOG_ZERO] * self.N for _ in range(len(obs))]
    for i in range(self.N):
        # start probabilities
        alpha[0][i] = math.log(self.pi[i]) + math.log(self.B[i][obs[0]] +
1e-300)
```

```

for t in range(1, len(obs)):
    for j in range(self.N):
        log_sum = LOG_ZERO
        for i in range(self.N):                # sum over previous
states
            log_sum = log_add(log_sum, alpha[t-1][i] +
math.log(self.A[i][j] + 1e-300))
            alpha[t][j] = log_sum + math.log(self.B[j][obs[t]] + 1e-300)
# emit this observation
        log_prob = LOG_ZERO
        for i in range(self.N):
            log_prob = log_add(log_prob, alpha[-1][i])
        return alpha, log_prob

```

Viterbi finds the single most likely hidden path — the same lattice, but taking the best predecessor at each step instead of summing, and remembering the choice so it can be traced back:

```

def viterbi(self, obs):
    """US7076102: the most likely hidden-state sequence behind the
observations."""
    delta = [[LOG_ZERO] * self.N for _ in range(len(obs))]
    psi = [[0] * self.N for _ in range(len(obs))]
    ...
    for t in range(1, len(obs)):
        for j in range(self.N):

```

```

        best_val, best_i = max((delta[t-1][i] + math.log(self.A[i][j] +
1e-300), i)
                                for i in range(self.N))    # best path into state
j
        delta[t][j] = best_val + math.log(self.B[j][obs[t]] + 1e-300)
        psi[t][j] = best_i                                # remember it, to
backtrack
        # ... backtrack through psi to recover the full state sequence

```

And Baum-Welch learns the whole model from raw observation sequences — the sequence-flavoured EM of Chapter 3. Using an HMM ties the three together: train on examples, score new sequences, decode the hidden states:

```

from patentml import HiddenMarkovModel

hmm = HiddenMarkovModel(n_states=2, n_symbols=3,
random_seed=1)
hmm.baum_welch(training_sequences, n_iter=20)    # learn the
model from data

_, score = hmm.forward(new_sequence)            # how well does the
model fit?
states = hmm.viterbi(new_sequence)              # the most likely
hidden path
print("log-likelihood:", round(score, 2))

```

```
print("inferred hidden states:", states)
```

From sequences of nothing but visible symbols, Baum-Welch recovers a model of the hidden process; forward then scores how well any new sequence fits it; and Viterbi reconstructs the hidden chain that most likely produced it — the activities behind the video, the words behind the sounds, the genes behind the DNA.

THE INHERITANCE

Reach for a hidden Markov model when you have a discrete hidden process behind a discrete observable sequence and you want to infer the hidden from the visible — sequence labelling, event and gesture recognition, biological sequence analysis, fault and condition monitoring, or any setting where the data is a stream of symbols generated by states you cannot see directly. Reach for it especially when data is limited and interpretability matters, because an HMM trains from modest examples and its states and transitions are often human-readable — you can look at the learned model and understand what it discovered, which the deep sequence models that supplanted it in large-scale speech cannot offer.

The HMM is also one of the great teaching algorithms, and worth understanding for its own sake: the forward algorithm's trick of summing over exponentially many paths in linear time, Viterbi's dynamic-programming search for the best path, and Baum-Welch's

threading of EM through a sequence are foundational ideas that recur across machine learning. Knowing them changes how you see every sequence problem.

The public-domain, dependency-free form suits the HMM well, because the model is small — three probability tables — and its algorithms are pure, readable loops. You can run sequence inference on an embedded device monitoring a stream of events, label sequences inside an application with no machine-learning stack, or study the three algorithms in full to learn the foundations of sequence modelling. Philips patented HMM event-learning for video in 2002 and earned 116 citations; the technique drove a generation of speech recognition; the patent has expired; and the hidden chain is now yours to train and decode.

That completes *The Sequence*. We have tracked continuous hidden state through noise with the Kalman filter, and inferred discrete hidden state from symbols with the HMM. Part III turns from what is hidden to how sure we are — making uncertainty itself the object of study, with models that predict and report their own confidence, and detectors that find the rare anomaly no one labelled.

PART III — THE CURVE: REASONING ABOUT WHAT YOU DON'T KNOW

Every model is uncertain, but most hide it. They hand you a prediction — a number, a label — and say nothing about how much to trust it. The two chapters of this part make uncertainty the point. The Gaussian process predicts a value and tells you, honestly, how sure it is — wide error bars where it has no data, tight ones where it does. The anomaly detector inverts the question entirely: instead of asking "what is this?", it asks "is this even normal?", flagging the rare point that fits no pattern. Both are about knowing the limits of what you know — the most honest, and in high-stakes settings the most important, thing a model can do.

CHAPTER 9

FUNCTIONS WITH ERROR BARS (GAUSSIAN PROCESSES)

THE FILING

US Patent 6,931,384 — "System and method providing utility-based decision making about communications", resting on probabilistic inference over function spaces with uncertainty quantification.

Assignee: Microsoft Corporation. Filed 2001. 258 forward citations. Now expired; public domain.

Most of the predictive models in these two volumes give you an answer and stop. A Gaussian process gives you an answer and its uncertainty — not a single predicted value but a whole probability distribution over possible values, narrow where the model is confident and wide where it is guessing. Microsoft's heavily-cited patent uses exactly this property: utility-based decision-making needs to know how uncertain a prediction is, because a confident

prediction and a wild guess should drive very different decisions.

The Gaussian process is the most principled tool in machine learning for prediction-with-honest-uncertainty, and it has a second remarkable property: it works beautifully with very little data, which is precisely when knowing your uncertainty matters most.

THE IDEA

Ordinary regression fits a single function to your data — a line, a curve — and predicts with it. A Gaussian process does something more ambitious: instead of committing to one function, it reasons about a distribution over all plausible functions that could fit the data, and at any input it reports the average of those functions (its prediction) together with how much they disagree (its uncertainty). Where your data points pin the functions down, they all agree and the uncertainty is tight. Far from any data, the plausible functions fan out wildly, and the uncertainty balloons. The Gaussian process knows when it is extrapolating into the unknown, and says so.

The machinery that makes this work is the kernel — the same idea from the support vector machine in Volume One, here playing a different role. The kernel defines how similar two input points are, and therefore how strongly their function values should be correlated: nearby points should have similar outputs, distant points need not. Feed the kernel your data and it produces, by an exact and elegant piece of linear algebra, the posterior — the predictive

distribution at any new point, as a mean and a variance. The mean is a weighted combination of your observed values, weighted by kernel-similarity to the query point. The variance starts from the kernel's prior uncertainty and shrinks by however much your nearby data constrains it. The choice of kernel encodes your assumptions about the function's character: the smooth RBF kernel for gently varying functions, the Matérn for rougher ones, the periodic kernel for repeating patterns — and the library offers all of these.

Two properties make Gaussian processes special among regression methods. First, calibrated uncertainty: the error bars are not an afterthought but fall directly out of the mathematics, and they behave correctly — confident near data, uncertain far from it. Second, data efficiency: because a Gaussian process reasons about whole functions rather than fitting many parameters, it produces sensible predictions and honest uncertainty from a mere handful of points, where a neural network would have nothing to learn from. The cost is computational — the exact method involves inverting a matrix the size of your dataset, which becomes expensive for large data — so Gaussian processes are at their best on small-to-medium problems where their uncertainty and data-efficiency are worth the price.

THE STORY

Gaussian processes have deep roots in statistics and geostatistics — the technique of "kriging," used to interpolate mineral concentrations

across a landscape from sparse boreholes, is a Gaussian process by another name, and the error bars there had real economic meaning. The method was brought fully into machine learning in the 1990s and 2000s, with Rasmussen and Williams' work giving it a definitive modern treatment. Microsoft's 2001 patent, cited 258 times, sits in this period, applying probabilistic inference with uncertainty to decision-making — a natural fit, because rational decisions require knowing not just what you predict but how much to trust it.

The Gaussian process found its killer application in Bayesian optimisation — and here the two volumes of this book join hands. Recall the estimation-of-distribution optimiser from Volume One, which learned a model of where good solutions lived. Bayesian optimisation is its most powerful modern form, and the model it learns is very often a Gaussian process: to tune the expensive hyperparameters of a machine-learning model, you fit a Gaussian process to the trials you have run, and use its uncertainty to decide where to try next — exploring where the GP is unsure, exploiting where it predicts good results. The honest error bars are the whole engine of the decision. This is now the standard method for automated machine-learning tuning, experimental design, and optimising any costly black-box function. The Gaussian process is also the gold standard wherever calibrated uncertainty is non-negotiable: scientific modelling from sparse measurements, safety-critical prediction, and active learning. Microsoft's patent has now expired into the public domain.

THE CODE

The kernel defines similarity, and `patentml` offers a family of them; the RBF kernel for smooth functions is the default:

```
def rbf_kernel(length_scale=1.0, variance=1.0):
    """Nearby points strongly correlated, distant points
independent."""
    def k(a, b):
        sq = sum((x - y) ** 2 for x, y in zip(a, b))
        return variance * math.exp(-0.5 * sq / (length_scale ** 2))
    return k
```

Fitting builds the kernel matrix over the training points and factorises it (by Cholesky decomposition — pure Python, no linear-algebra dependency). Prediction then returns, for each query point, both a mean and a standard deviation — the prediction and its honest uncertainty:

```
def predict(self, X_star, return_std=False):
    means, stds = [], []
    for xs in X_star:
        k_star = [self.kernel(xs, xi) for xi in self._X]      # similarity
        # to training points
        mean = sum(k_star[i] * self._alpha[i] for i in
range(len(self._X)))
```

```
means.append(mean * self._y_std + self._y_mean)
if return_std:
    # variance starts from the prior and shrinks by what nearby
data constrains
    v = _solve_lower(self._L, k_star)
    var = self.kernel(xs, xs) - sum(vi * vi for vi in v)
    stds.append(math.sqrt(max(var, 0.0)) * self._y_std)
return means, (stds if return_std else None)
```

That `return_std=True` is the whole point of the method — you get error bars, not just a number:

```
from patentml import GaussianProcessRegressor, rbf_kernel

gp =
GaussianProcessRegressor(kernel=rbf_kernel(length_scale=1.0),
noise_var=1e-3)
gp.fit(X_train, y_train)           # just a handful of points
is fine

means, stds = gp.predict(X_test, return_std=True)
for m, s in zip(means, stds):
    print(f"prediction {m:.2f} ± {s:.2f}")      # tight ± near data,
wide ± far from it
```

Trained on a few points and asked to predict across a range, the

Gaussian process returns tight error bars in the neighbourhoods where it has data and visibly wider ones in the gaps and beyond the edges — a model that tells you where it is guessing. That $\pm$ on every prediction is what no ordinary regressor provides and what makes the Gaussian process the tool of choice when uncertainty must be taken seriously.

THE INHERITANCE

Reach for a Gaussian process when you need to know how uncertain a prediction is, not just what it is — in scientific modelling from sparse data, in safety-critical systems where overconfidence is dangerous, in any setting where a downstream decision should depend on the model's confidence. And reach for it when data is scarce, because the Gaussian process produces sensible predictions and calibrated uncertainty from a handful of points, exactly the regime where most other methods have too little to learn from.

Its standout application — and the place this book's two volumes meet — is Bayesian optimisation: using the Gaussian process's uncertainty to decide where to sample next when each evaluation is expensive, the now-standard method for tuning models and optimising costly black-box functions. Understanding the Gaussian process is understanding the engine inside automated machine learning. The honest limit is scale: the exact method grows costly with dataset size, so it suits small-to-medium problems where its

unique strengths pay off.

The public-domain, dependency-free form is a quiet triumph here, because Gaussian processes are normally assumed to require a serious numerical-computing stack — yet `patentml` implements the exact posterior, Cholesky factorisation and all, in pure Python. That puts principled, uncertainty-aware regression within reach on a device or in a browser, and lays bare the elegant linear algebra of the posterior for anyone wanting to truly understand how a model can know what it does not know. Microsoft patented uncertainty-based probabilistic inference in 2001 and earned 258 citations; the patent has expired; the model that predicts with honest error bars is now yours.

The next chapter takes uncertainty to its sharpest point: instead of predicting a value with error bars, it asks the binary question that underlies fraud detection, fault monitoring, and security — is this point normal at all, or is it an anomaly?

CHAPTER 10

THE SPIKE IN THE DATA (ANOMALY DETECTION)

THE FILING

US Patent 7,225,343 — "System and methods for adaptive model generation for detecting intrusions in computer systems", scoring anomalies by unsupervised random partitioning.

Assignee: Columbia University. Filed 2003. 159 forward citations. Now expired; public domain.

Every method so far in this volume has asked "what kind of thing is this?" — which cluster, which hidden state, which predicted value. Anomaly detection asks a sharper and stranger question: "is this thing normal at all?" Columbia's patent was born of computer security — detecting intrusions, the rare malicious events hidden in a flood of ordinary activity — and that is the archetype of the whole field. You have masses of normal data and a vanishing few

anomalies; the anomalies are too rare and too varied to learn as a "class," so you cannot treat this as ordinary classification. Instead you must learn the shape of normal and flag whatever falls outside it. The patent's particular insight, embodied in the isolation forest, is a beautifully counterintuitive way to do this — and it has become one of the most popular anomaly detectors in the world.

THE IDEA

The naive approach to anomaly detection is to model normality densely — fit the data's distribution and flag low-probability points (the Gaussian mixtures of Chapter 3 and the density estimation of Chapter 5 can both be used this way). The isolation forest turns the problem inside out with a brilliant observation: anomalies are easy to isolate. If you repeatedly chop the data with random splits — pick a feature at random, pick a split value at random, divide, and recurse — a normal point, buried in a dense crowd of similar points, takes many splits to isolate into its own region. An anomaly, sitting alone out in sparse space, gets cut off from everything else in just a few splits. The number of random cuts needed to isolate a point is therefore a measure of how anomalous it is: few cuts means anomaly, many cuts means normal.

So you build a forest of random "isolation trees," each grown by recursive random splitting on a small sample of the data. To score a point, you drop it down each tree and measure its path length — how

deep it falls before being isolated. Average that path length across the forest, normalise it (the implementation uses an exact correction, $c(n)$, for the expected path length in a random tree), and you have an anomaly score: short average paths score near 1 (anomalous), long paths score well below 0.5 (normal). The method is elegant because it never has to model what normal looks like in detail — it exploits the single structural fact that outliers are, by definition, easier to separate from the crowd. It is fast (random splits are cheap), it scales well, it handles many dimensions gracefully, and it makes almost no assumptions about the data's distribution.

The companion method in this chapter, the one-class SVM (also in the module), takes the more classical route: it learns a tight boundary around the normal data — a frontier enclosing "normal" — and flags anything that falls outside it. Both are unsupervised: they learn from data that is mostly or entirely normal, with no need for labelled examples of the anomalies, which is exactly right because in real anomaly problems you rarely have labelled anomalies to learn from — that is the whole difficulty.

THE STORY

Anomaly detection has a long history across statistics (outlier detection), engineering (fault and novelty detection), and security (intrusion detection), unified by the common structure of "lots of normal, few weird, find the weird." Columbia's 2003 patent comes

straight from the intrusion-detection world, where the problem is acute: attacks are rare, constantly mutating, and hidden in enormous volumes of legitimate traffic, so you cannot enumerate them — you must model normal behaviour and flag deviations. The isolation forest itself was introduced by Liu, Ting, and Zhou in 2008, and its inside-out logic — isolate rather than model — made it an immediate favourite, fast and effective where density-based methods struggled in high dimensions.

The field has only grown in importance. Fraud detection in finance and payments, fault and failure prediction in industrial machinery (a sensor pattern that has never been seen before), network and security monitoring, quality control in manufacturing, health monitoring from wearables, and — bringing this book full circle — detecting when a machine-learning system itself is receiving inputs unlike anything in its training data, a key safety concern for deployed AI. Across all of these, the structure is the same and the isolation forest is a standard tool. Columbia's patent has now expired into the public domain, and the inside-out trick — that the anomaly is the point you can isolate fastest — is free.

THE CODE

The whole isolation-forest idea hinges on path length, and the recursion that measures it is short. A point falls through random splits until it is isolated; the depth at which that happens is its path

length:

```
def _path_length(node, x, depth):
    if node.feats == -1 or node.left is None:      # isolated (or a tiny
leaf)
        return depth + _c(node.size)              # depth reached, plus a
correction
    if x[node.feats] < node.thresh:
        return _path_length(node.left, x, depth + 1)
    else:
        return _path_length(node.right, x, depth + 1)
```

Each tree is grown by random splits — a random feature, a random threshold — which is what makes anomalies fall out quickly:

```
feat = rng.randint(0, d - 1)          # a random feature
thresh = rng.uniform(vmin, vmax)       # a random split value
left_idx = [i for i in idxs if X[i][feat] < thresh]
right_idx = [i for i in idxs if X[i][feat] >= thresh]
```

The score averages path length across the forest and turns it into an anomaly score near 1 (anomalous) or below 0.5 (normal):

```
def score_samples(self, X):
    cn = _c(self._n_fit)
    scores = []
```

```
for x in X:
    mean_path = sum(_path_length(t, x, 0) for t in self._trees) /
self.n_estimators
    scores.append(2 * (-mean_path / (cn + 1e-10))
score != 1) anomaly
return scores
```

Using it, you train on data that is mostly normal and then score new points — high scores are the anomalies:

```
from sklearn import IsolationForest

iso = IsolationForest(n_estimators=100, random_seed=1)
iso.fit(normal_data)                # learn the shape of
"normal"

scores = iso.score_samples(new_points)    # higher = more
anomalous

flags = iso.predict(new_points)          # +1 normal, -1
anomaly

print("anomaly scores:", [round(s, 2) for s in scores])
```

A point planted far out in sparse space gets isolated by the random splits in just a cut or two, so its average path length is short and its score spikes toward 1 — flagged — while the dense mass of normal points take many cuts to isolate and score low. The forest found the

outlier without ever being shown an example of one, exploiting nothing but the fact that the strange point was easy to cut away from the crowd.

THE INHERITANCE

Reach for anomaly detection whenever your problem has the shape "lots of normal, a few weird, find the weird"* and you cannot* treat it as ordinary classification because the anomalies are too rare, too varied, or simply unlabelled. Fraud, intrusion, fault prediction, quality control, novelty detection, monitoring deployed systems for inputs unlike their training data — all share this structure, and all call for learning the shape of normal rather than the catalogue of abnormal. The isolation forest is an excellent default: fast, scalable, effective in many dimensions, and almost assumption-free; the one-class SVM is the classical alternative when you prefer to learn an explicit boundary around normality.

The defining virtue is that these methods are unsupervised — they need only normal (or mostly-normal) data, no labelled anomalies, which is exactly the situation real anomaly problems present, since if you already had a good catalogue of anomalies you would not have an anomaly problem. This also connects back to the clustering of this volume's first part: DBSCAN's noise points and a Gaussian mixture's low-likelihood points are anomaly signals too, and the isolation forest is the specialist built for the job.

The public-domain, dependency-free form suits anomaly detection particularly well because so much of it happens at the edge — on the device, in the security appliance, in the embedded monitor watching a sensor stream — where flagging the abnormal in real time, without a heavyweight stack or a round-trip to a server, is exactly what is needed. An isolation forest is a handful of small random trees and a path-length count, light enough for any of those settings. Columbia patented adaptive anomaly detection for intrusion in 2003 and earned 159 citations; the patent has expired; the detector that finds the strange by isolating it is now yours.

That completes The Curve — reasoning about what you do not know, whether as honest error bars or as the flagged outlier. Part IV makes the boldest move in this volume: it removes even the data. Instead of learning from examples, the algorithms of reinforcement learning learn from consequences — acting, observing the reward, and discovering good behaviour with no one ever showing them the right answer.

PART IV — THE CHOICE: LEARNING FROM CONSEQUENCES

Here even the dataset disappears. The algorithms of this part are not shown examples at all — they learn by acting, observing what happens, and adjusting. There is no answer key, not even an unlabelled pile of data; there is only a world that responds to what you do with reward or regret, and the problem of working out, from those consequences alone, how to act well. This is reinforcement learning, and it begins with the purest version of the dilemma at its heart — the tension between trying new things and exploiting what already works — captured in the multi-armed bandit. From there we build to agents that learn to act across time, and to the actor-critic methods that drive modern reinforcement learning.

CHAPTER 11

THE SLOT MACHINES (MULTI-ARMED BANDITS)

THE FILING

US Patent 6,981,040 — a machine-learning system "selecting actions based on maximizing expected reward."

Assignee: Utopy, Inc. Filed 2000. 919 forward citations. Now expired; public domain.

With over nine hundred citations, this is one of the most influential patents in the entire book — second only to the voting-ensemble patent of Volume One. It addresses the most elemental problem in all of decision-making under uncertainty, the explore-exploit dilemma, dressed in the metaphor that gives the field its name. Imagine a row of slot machines — "one-armed bandits" — each paying out at a different, unknown rate. You have a limited number of pulls. Every pull forces a choice: play the machine that has paid best so far

(exploit your knowledge), or try a machine you've barely played, which might be better (explore for more knowledge). Play it too safe and you may pour your pulls into a mediocre machine while a jackpot sits untried; explore too much and you squander pulls on losers. The bandit problem is the mathematics of getting this balance right, and Utopy's heavily-cited patent — selecting actions to maximise expected reward — is a cornerstone of an idea that now silently governs much of the digital world.

THE IDEA

You face k options ("arms"), each returning rewards from an unknown distribution. At each step you pick one arm, receive a reward, and update what you know. Your goal is to maximise total reward over many steps — which means you must learn which arms are good while using that knowledge, simultaneously. The quantity you are implicitly minimising is regret: the gap between the reward you actually earned and the reward you would have earned had you known the best arm from the start. Several beautiful strategies manage the explore-exploit trade-off, and they form a ladder of sophistication:

Epsilon-greedy is the simple baseline: most of the time, pull the arm with the best average reward so far (exploit); but with a small probability ϵ , pull a random arm (explore). It works, and it is the honest starting point — but it explores blindly, wasting some

pulls on arms already known to be bad, and it never stops exploring unless you decay epsilon over time.

UCB1 — Upper Confidence Bound — is cleverer and needs no tuning at all. For each arm it computes not just the average reward but an optimism bonus that is large for arms tried few times and shrinks as an arm is played more. It then pulls the arm with the highest average-plus-bonus. The effect is "optimism in the face of uncertainty": an arm gets explored because you are uncertain about it, and the exploration tapers off naturally as certainty grows. The bonus, scaling as the square root of $\log(t)/n$, gives UCB1 strong theoretical guarantees on regret with zero parameters to set.

Thompson Sampling is the elegant Bayesian approach, and often the best in practice. For each arm it maintains a probability distribution over that arm's true reward rate — its belief, updated with every observation. To choose, it samples one value from each arm's belief distribution and pulls the arm with the highest sample. Arms it is confident are good get picked often; arms it is uncertain about get picked sometimes, in exact proportion to the probability they might be best. Exploration emerges naturally from the uncertainty in the beliefs, with no explicit exploration parameter at all. It is probability matching, and it is beautiful. The library also includes EXP3 for adversarial settings where rewards are not even stationary, and LinUCB for contextual bandits, where each decision comes with side information.

THE STORY

The multi-armed bandit was studied as a serious mathematical problem from the mid-twentieth century, originally motivated by — strikingly — the ethics of clinical trials, where the explore-exploit dilemma has life-or-death stakes: every patient assigned to an inferior treatment arm so that you can learn is a real cost, and the bandit framework is how you balance learning against the welfare of the patients in front of you. That framing has returned to prominence as adaptive clinical trials gain acceptance.

But the bandit's explosion in importance came with the internet. Utopy's 2000 patent, cited over nine hundred times, sits at the dawn of the era in which bandit algorithms became the silent machinery of the web. Every time a website runs an A/B test that adaptively shifts traffic toward the better-performing variant, that is a bandit. Every recommendation that balances showing you proven favourites against surfacing something new is a bandit. Ad selection, news article placement, hyperparameter search, dynamic pricing — all are bandit problems, choosing among options to maximise reward while learning which options are best. The contextual variant, which uses information about the user or situation to choose, is the backbone of personalised recommendation at scale. Few algorithms in this book are so directly responsible for shaping daily digital experience, and its foundational patent has now expired.

THE CODE

The three core strategies differ only in how they choose, and `patentml` makes the contrast vivid. Epsilon-greedy: exploit the best, but sometimes roll the dice:

```
def select(self):
    if random.random() < self.epsilon:
        return random.randint(0, self.n - 1)    # explore: a random
arm
    return max(range(self.n), key=lambda i: self.values[i]) # exploit:
the best so far
```

UCB1: pull the arm with the best average plus an optimism bonus that shrinks with experience — no parameters:

```
def select(self):
    for i in range(self.n):
        if self.counts[i] == 0:
            return i                # try each arm once first
    log_t = math.log(self.t)
    return max(range(self.n),
                key=lambda i: self.values[i] + math.sqrt(2 * log_t /
self.counts[i]))
    #                average reward + optimism bonus (big when n_i
is small)
```

Thompson Sampling: keep a belief about each arm, sample from it, and pick the best sample:

```
def select(self):
    samples = [self._beta_sample(self.alpha[i], self.beta_p[i]) for i in
range(self.n)]
    return max(range(self.n), key=lambda i: samples[i]) #
exploration emerges from the sampling
```

```
def update(self, arm, reward):
    if reward > 0.5:
        self.alpha[arm] += 1                # a success sharpens belief
upward
    else:
        self.beta_p[arm] += 1                # a failure, downward
```

Run any of them against arms with hidden payout rates and watch it home in on the best:

```
from patentml import ThompsonSampling

bandit = ThompsonSampling(n_arms=3)
true_rates = [0.3, 0.5, 0.8]                # arm 2 is best — but the
bandit doesn't know
for _ in range(1000):
    arm = bandit.select()
```

```
reward = 1.0 if random.random() < true_rates[arm] else 0.0
bandit.update(arm, reward)

print("times each arm was chosen:", bandit.alpha)  # heavily
concentrated on arm 2
```

Knowing nothing at the start, the bandit tries all three arms, watches the rewards, and steadily concentrates its pulls on the best one — exploring just enough to be confident, exploiting as soon as it is. That self-balancing, learned purely from consequences, is what drives the A/B tests and recommendations you meet every day.

THE INHERITANCE

Reach for a multi-armed bandit whenever you must repeatedly choose among options to maximise reward, while learning which options are best from the outcomes — A/B testing, recommendation, ad and content selection, dynamic pricing, hyperparameter search, adaptive experiments. Wherever the static approach is "run a fixed experiment, then act," the bandit offers something better: act and learn at the same time, shifting toward what works as the evidence accumulates, and wasting far fewer trials on inferior options. For online decision-making, the bandit is very often the right framework, and Thompson Sampling or UCB1 the right algorithm.

The bandit is also the conceptual gateway to all of reinforcement

learning. Its explore-exploit dilemma is the seed of the harder problems in the rest of this part — the only thing the bandit lacks is state and consequences over time, which the next chapters add. Understanding the bandit is understanding the core tension that every reinforcement learner must resolve.

The public-domain, dependency-free form is perfectly suited to bandits, because each algorithm is just a few counters or beliefs and a selection rule — tiny, fast, and trivially embeddable. You can run adaptive selection inside an application, on a device, or at the edge of a web service with no machine-learning stack at all, making real-time choices that improve themselves. Utopy patented reward-maximising action selection in 2000 and earned over nine hundred citations as the idea became the web's silent decision engine; the patent has expired; the mathematics of the wise choice is now yours.

The next chapter adds what the bandit lacks — a world that changes in response to your actions, where a choice now affects the choices available later — and meets the algorithm that learns to act across time: Q-learning.

CHAPTER 12

LEARNING TO ACT (Q-LEARNING & TEMPORAL DIFFERENCE)

THE FILING

US Patent 6,985,172 — "Model-based incident detection" using temporal-difference learning.

Assignee: Southwest Research Institute. Filed 2002. 145 forward citations. Now expired; public domain.

The companion US Patent 6,353,814 (Michigan State University, 1997) covers a "developmental learning machine" using reward and punishment over a Markov process. Both now expired; public domain.

The bandit of the last chapter chose among options with no notion of state — each pull was independent, the world unchanged by your choice. The real world is not like that. When you act, the situation

changes, and your next choices depend on where your last choice left you. A move in a game, a turn on a road, a control adjustment on a machine — each alters the state and shapes what comes next, and a reward may arrive only much later, long after the action that earned it. Q-learning is the breakthrough that learns to act well in such worlds, where consequences ripple forward through time. Southwest Research Institute's patent applies the temporal-difference update at its core to detecting incidents in video, but the update rule it names is one of the most important equations in all of reinforcement learning.

THE IDEA

Model the world as states and actions: you are in some state, you take an action, you receive a reward, and you land in a new state. Your aim is not to maximise the immediate reward but the total reward over time — and that is what makes this hard, because the best action now may yield little immediate reward while setting up a large reward later. A good chess move may sacrifice a piece to win the game twenty moves on. To act well, you must somehow account for future consequences, not just present ones.

Q-learning does this by learning a value for each state-action pair, written $Q(s, a)$: an estimate of the total future reward you can expect if you take action a in state s and then act well thereafter. If you knew the true Q values, acting optimally would be trivial — in any state, just take the action with the highest Q . The miracle is that you

can learn these values from experience, without any model of how the world works, through the temporal-difference update:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Unpack it, because it is the heart of the field. After taking action a in state s , you get reward r and land in state s' . Your estimate of what $Q(s, a)$ should have been: the reward you actually got, r , plus the discounted value of the best you can do from the new state, $\gamma \max_{a'} Q(s', a')$. The gap between this (the "target") and your old estimate is the temporal-difference error — your surprise. You nudge $Q(s, a)$ toward the target by the learning rate (the learning rate) of that error. The discount factor, γ , controls how much you care about the future versus the present: near 1 and the agent is far-sighted, planning for distant rewards; near 0 and it is myopic. The deep trick is bootstrapping — each value estimate is updated using other value estimates, so that reward information propagates backward through the chain of states over many episodes. A reward at the end of a maze gradually pulls up the values of the states leading to it, until the agent has carved a path of high values from start to goal. And because it must still balance trying new actions against taking the best known one, Q-learning uses the explore-exploit strategies of the last chapter — typically epsilon-greedy — to choose actions while learning.

THE STORY

Q-learning was introduced by Chris Watkins in 1989, and its great theoretical virtue made it famous: it was proven to converge to the optimal action-values even though it is model-free — the agent never needs to know the rules of the world, only to experience it. It learns purely from the stream of states, actions, and rewards, which is exactly what you want when the world is too complex to model. Temporal-difference learning, the broader idea, came from Richard Sutton and is one of the foundational contributions to reinforcement learning; Sutton and Andrew Barto's textbook made these ideas the canon of the field. Southwest Research's patent applies temporal-difference learning to a concrete real-world stream — detecting incidents from video over time — while Michigan State's "developmental learning machine" frames the same reward-and-punishment learning in terms of an agent developing competence.

For years, tabular Q-learning — storing a value for every state-action pair in a table — worked beautifully on problems small enough to enumerate, and it remains the clearest way to understand reinforcement learning. Its limitation is scale: you cannot keep a table for the states of a video game screen or a robot's sensors, because there are astronomically many. The resolution, which ignited the modern era, was to approximate the Q-function with a neural network instead of a table — Deep Q-Networks — which in 2015 famously learned to play dozens of Atari games at human level from raw pixels, and which descend directly from the tabular update

in this chapter. The same temporal-difference idea, scaled by deep learning, sits inside the systems that mastered Go and beyond. The patent has expired; the equation that taught machines to act across time is public domain.

THE CODE

The entire Q-learning algorithm is the temporal-difference update, and patentml writes it almost exactly as the equation reads:

```
def update(self, state, action, reward, next_state, done=False):
    """US 6 985 172:  $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s, a') - Q(s, a)]$ """
    q_current = self._get_q(state, action)
    if done:
        q_target = reward # terminal: only the
        final reward
    else:
        q_next = max(self._get_q(next_state, a) for a in self.actions)
        q_target = reward + self.gamma * q_next # reward now
        + discounted best future
    td_error = q_target - q_current # the "surprise"
    self._set_q(state, action, q_current + self.alpha * td_error) #
    nudge toward the target
```

Action selection is the explore-exploit choice from the last chapter

— mostly take the best known action, occasionally try something new:

```
def choose_action(self, state):
    if random.random() < self.epsilon:
        return random.choice(self.actions)          # explore
    q_vals = {a: self._get_q(state, a) for a in self.actions}
    return max(q_vals, key=q_vals.get)              # exploit the
best-valued action
```

A full learning loop is the agent living through episodes, updating its values from each transition:

```
from patentml import QLearner

agent = QLearner(actions=["up", "down", "left", "right"], alpha=0.1,
gamma=0.99)

for episode in range(500):
    state = env.reset()
    done = False
    while not done:
        action = agent.choose_action(state)
        next_state, reward, done = env.step(action)
        agent.update(state, action, reward, next_state, done) # learn
from the consequence
```

```
state = next_state
agent.decay_epsilon()           # explore less as it
learns
```

Over hundreds of episodes, reward earned at a goal propagates backward through the temporal-difference update, raising the values of the states and actions that lead there, until the agent's epsilon-greedy policy reliably steers it to the goal — a path discovered entirely from trial, error, and delayed reward, with no one ever showing it the way.

THE INHERITANCE

Reach for Q-learning when your problem is sequential decision-making under delayed reward — an agent acting in a world where actions change the state and the payoff may come much later. Game-playing, robot control, navigation, resource management, dialogue, trading, any setting framed as "states, actions, rewards over time" is reinforcement-learning territory, and tabular Q-learning is the right tool whenever the state space is small enough to enumerate — and the right teacher always, because it makes the core ideas concrete. When the state space is too large, the same temporal-difference update, with a neural network standing in for the table, scales to the hardest problems in the field.

The conceptual inheritance is enormous: the temporal-difference

update in this chapter is, with refinements, the engine inside the reinforcement-learning systems that mastered Atari, Go, and modern control — and, increasingly, inside the methods used to align large language models from human feedback. To understand this one equation is to hold the seed of all of it.

The public-domain, dependency-free form makes tabular Q-learning an ideal way to learn reinforcement learning hands-on — the value table and the update right there, nothing hidden — and a genuinely usable controller for small-state problems on modest hardware, embedded agents that adapt to their environment with no framework at all. Southwest Research patented temporal-difference incident detection in 2002 and earned 145 citations; Watkins and Sutton's ideas beneath it are foundational; the patent has expired; and the rule that learns to act across time is yours.

The final chapter of this part addresses Q-learning's main limit — that it learns values and reads off a policy, which struggles with large or continuous action spaces — by learning a policy directly, with a critic to guide it: the actor-critic methods that drive modern reinforcement learning.

CHAPTER 13

CRITIC AND ACTOR (POLICY-GRADIENT REINFORCEMENT LEARNING)

THE FILING

US Patent 6,985,172 — the same Southwest Research Institute patent of the last chapter, here in its aspect covering concurrent value-function and policy optimisation, where a critic baseline reduces the variance of policy-gradient estimates. Filed 2002. 145 forward citations. Now expired; public domain.

This is the same filing as Chapter 11, seen from its other face — a nice illustration of how a single patent can anticipate two related techniques. Q-learning learned the value of each action and then acted greedily on those values. That works wonderfully when actions are few and discrete, but it falters when there are very many actions, or a continuous range of them (a steering angle, a motor torque, any real-valued control), because you cannot take the maximum over an

infinite set of actions, and small changes in value can flip the chosen action abruptly. Actor-critic methods solve this by learning the policy directly — a function that outputs action probabilities — while a critic learns values to guide the actor's improvement. This pairing is the workhorse of modern reinforcement learning, and its variance-reduction trick, the critic baseline, is exactly what the patent describes.

THE IDEA

Actor-critic, as the name promises, has two cooperating parts:

The actor is the policy — it looks at the current state and outputs a probability for each action (for continuous actions, the parameters of a distribution to sample from). The agent acts by sampling from these probabilities. Crucially, the actor is adjusted not by reading off values but by policy gradient: you nudge the actor's parameters in the direction that makes good actions more likely and bad actions less likely. If an action led to better-than-expected reward, increase its probability; if worse, decrease it. This direct approach handles continuous and high-dimensional action spaces naturally, because the actor simply outputs a distribution rather than needing to maximise over all actions.

The critic is a value function — it estimates how good each state is, much as Q-learning estimated values, and it learns by the same

temporal-difference idea from the last chapter. But its job here is different: the critic provides the baseline that tells the actor whether an action turned out better or worse than expected. This is the key to making policy gradients work in practice. The raw policy-gradient signal — "this action led to total reward R , so adjust toward it by R " — is extremely noisy, because total reward swings wildly from episode to episode, and learning from such a noisy signal is painfully slow. The critic fixes this. Instead of "this action got reward R ," the actor learns from the advantage — R minus what the critic expected — which measures whether the action was better or worse than the typical outcome from that state. Subtracting the critic's baseline dramatically reduces the variance of the learning signal without biasing it, and that variance reduction is precisely what the patent claims: "a critic baseline reduces variance in policy gradient estimates." The advantage-based version is called A2C, Advantage Actor-Critic, and it is the form implemented here, alongside a lightweight PPO-style variant that adds a safeguard against changing the policy too drastically in any one step.

So the two parts dance: the actor proposes actions and is refined to favour those the critic judges better than average; the critic watches outcomes and refines its judgement of what "average" is. Actor and critic improve together, each making the other's job possible.

THE STORY

The policy-gradient idea — directly optimising a parameterised policy by gradient ascent on expected reward — has roots in the REINFORCE algorithm of the early 1990s. Its great weakness was variance: the learning signal was so noisy that training was slow and unstable. The actor-critic architecture, pairing the policy with a learned value baseline, was the fix that made policy-gradient methods practical, and it has been refined steadily ever since. Southwest Research's 2002 patent, in describing concurrent policy and value optimisation with a variance-reducing critic, captures this architecture at an early and influential moment.

The pay-off has been enormous, because actor-critic methods are what make reinforcement learning work in the continuous-control settings that Q-learning cannot handle — robotics, locomotion, dexterous manipulation, autonomous vehicles, where actions are real-valued and high-dimensional. Modern descendants — A2C/A3C, and especially PPO (Proximal Policy Optimisation) — are among the most widely used reinforcement-learning algorithms in existence. PPO in particular has become a default workhorse: it trained agents to superhuman play in complex video games, it drives much of robotics research, and — bringing this book to a strikingly contemporary point — it is central to reinforcement learning from human feedback, the technique used to align large language models with human preferences. When a modern AI assistant is tuned to be helpful and harmless, an actor-critic policy-gradient method is very often doing the tuning. The lineage runs from this expired patent's

"actor and critic" straight into the training of the systems defining the current era.

THE CODE

The actor is a policy that turns a state into action probabilities via softmax — small, linear, and readable:

```
class LinearActor:
    def logits(self, state):
        return [sum(self.W[a][s] * state[s] for s in range(self.n_states))
+ self.b[a]
                for a in range(self.n_actions)]
    def probs(self, state):
        return _softmax(self.logits(state))      # a probability for each
action
    def select_action(self, state, rng):
        probs = self.probs(state)
        # sample an action from the policy's distribution
```

The critic is a value function estimating how good a state is, and the advantage — outcome minus the critic's expectation — is the signal that drives the actor. The A2C update computes that advantage and pushes the actor toward better-than-expected actions while training the critic toward the observed returns:

```

For a transition: advantage = (reward +  $\gamma \cdot V(\text{next\_state})$  - critic.value(state)) (0 if done else 1)

```

```

advantage = td_target - critic.value(state)      # better or worse than expected?

```

```

actor.update(state, action, advantage, lr_actor)  # make good actions more likely

```

```

critic.update(state, td_target, lr_critic)       # sharpen the value estimate

```

That single quantity, advantage, is the whole reason the critic exists: it replaces the wild raw return with a centred, low-variance signal. Training an A2C agent is the familiar episode loop, with actor and critic improving in tandem:

```

from patentml import A2C

```

```

agent = A2C(n_states=4, n_actions=2, seed=1)

```

```

for episode in range(1000):

```

```

    state = env.reset()

```

```

    done = False

```

```

    while not done:

```

```

        action = agent.select_action(state)

```

```

        next_state, reward, done = env.step(action)

```

```

        agent.learn(state, action, reward, next_state, done)  # actor +

```

critic step together

```
state = next_state
```

The actor starts with a random policy and the critic with no idea of value; episode by episode, the critic learns what good states are worth, the advantage signal steers the actor toward actions that beat the critic's expectations, and a competent policy emerges — learned, as always in this part, purely from acting and observing the consequences, with the critic's baseline keeping the learning stable enough to succeed.

THE INHERITANCE

Reach for actor-critic methods when your reinforcement-learning problem has large or continuous action spaces — robotics and continuous control above all, where Q-learning's "take the best action" is impossible because the actions form an unbounded continuum. By learning a policy directly, the actor handles real-valued, high-dimensional actions naturally, and the critic's baseline makes the learning stable and efficient enough to work in practice. For modern, scaled reinforcement learning, actor-critic and its descendants — A2C, PPO — are very often the methods of choice.

The contemporary inheritance could hardly be more current: PPO, a direct descendant of the architecture in this chapter, is a standard tool

for training game-playing agents, robots, and — through reinforcement learning from human feedback — the large language models at the centre of today's AI. The "actor and critic" the patent named in 2002 is, in refined form, part of how the current generation of AI systems is aligned with human intent. Understanding the advantage signal and the actor-critic dance is understanding a load-bearing piece of the modern field.

The public-domain, dependency-free form makes these often-intimidating methods legible — the actor's policy, the critic's value, and the advantage that couples them, all in plain, readable Python — and usable for small continuous-control problems on modest hardware. It is the clearest possible way to learn the architecture behind modern reinforcement learning before scaling it up with deep networks. Southwest Research patented concurrent policy-and-value optimisation with a variance-reducing critic in 2002 and earned 145 citations; the patent has expired; the actor-and-critic at the heart of modern reinforcement learning is now yours.

That completes *The Choice* — learning to act from consequences alone, from the single-step bandit through value-based Q-learning to policy-based actor-critic. Part V, the last, returns from action to meaning: how machines came to handle the messy, human world of language, taste, and the compression of too much information — word embeddings, recommenders, and the codebooks that shrink

data to its essence.

PART V — THE MEANING: LANGUAGE, RECOMMENDATION, AND COMPRESSION

The final movement of this volume is about the messiest data of all: the human world of words, taste, and overwhelming abundance. How does a machine come to treat words as things it can compute with? How does it learn your preferences well enough to recommend what you have never seen? How does it shrink a flood of data down to its essence without losing what matters? The five chapters here answer those questions — word embeddings that turn language into geometry, the text classifier that reads, the collaborative filter that recommends, the matrix factorisations that compress, and the vector quantisers that build codebooks. Each is a patent from the late-1990s and 2000s, expired and free, and several of them turn out to be ancestors of the most modern AI of all. We begin with the idea that taught machines the meaning of words.

CHAPTER 14

WORDS AS VECTORS (WORD EMBEDDINGS)

THE FILING

US Patent 6,556,983 — "Methods and apparatus for finding semantic information, such as usage logs, using pattern lattice", covering latent semantic structure in token co-occurrence. Assignee: Microsoft Corporation. Filed 2000. 645 forward citations. Now expired; public domain.

This is the surprise of the whole series. The idea of representing words as vectors — points in a high-dimensional space, arranged so that words with similar meanings sit close together — is associated in most people's minds with word2vec, the famous 2013 work that made "king" man + woman "H queen" a viral demo of machine-learned meaning. But the underlying idea, that a word's meaning can be captured from the statistics of the company it keeps,

is older, and the patent record shows the major players staking claims on latent-semantic structure in word co-occurrence years before the famous papers. Microsoft's patent, cited 645 times, is part of that earlier wave. The technique it points to — turning words into geometry — is one of the foundational ideas of modern natural-language processing, and a direct ancestor of the large language models of today.

THE IDEA

Computers handle numbers; language is symbols. The first problem in machine learning over text is how to represent a word as something you can compute with. The crude answer is to give each word its own slot — "cat" is slot 5,012, "dog" is slot 8,317 — but this is useless for meaning, because in that scheme "cat" and "dog" are exactly as different from each other as "cat" and "democracy." There is no notion that some words are similar. Word embeddings fix this by representing each word as a dense vector of real numbers — a point in, say, 100-dimensional space — arranged so that the geometry encodes meaning. Similar words land near each other; the directions between words capture relationships.

How can a machine possibly learn such an arrangement, with no one to define what words mean? Through the distributional hypothesis, one of the deepest ideas in linguistics: a word is characterised by the company it keeps. Words that appear in similar contexts tend to have

similar meanings. "Cat" and "dog" both occur near "pet," "fur," "vet," "feed" — and that statistical overlap is enough to discover that they are similar, without any dictionary. The machine reads an enormous body of text, tallies which words co-occur with which, and arranges the vectors so that words with similar co-occurrence patterns are placed close together. Meaning falls out of statistics.

Two main routes lead to the same destination, and the library implements both. The count-based route (the GloVe-style approach) builds a giant table of how often each word co-occurs with each other word, transforms those counts into a measure of association called pointwise mutual information (PMI) — which captures whether two words co-occur more than chance would predict — and then factorises that matrix down to dense, low-dimensional vectors (using the kind of matrix factorisation that is the subject of this part's chapter on SVD). The predictive route (word2vec's skip-gram with negative sampling) instead trains a simple model to predict, for each word, which words appear in its context, and reads the learned weights as the embeddings. Remarkably, the two routes are mathematically related and yield similar geometry. The result, by either path, is a space where the famous analogies hold — where the `vector arithmetic king - man + woman really does` because the embedding has captured the relational structure of meaning as directions in space.

THE STORY

The distributional hypothesis dates to linguists of the 1950s (Firth's "you shall know a word by the company it keeps"), and statistical approaches to word meaning — latent semantic analysis, which factorised word-document matrices — were developed through the 1990s, the era of Microsoft's patent. The watershed moment in the public imagination came in 2013, when word2vec, from a team at Google, produced fast, high-quality embeddings at scale and showed off the arithmetic of analogies, igniting an explosion of work on representing meaning as geometry. GloVe, the count-based method, followed shortly. Suddenly, machines had a useful representation of words — one that captured similarity and relationship — and it became the standard first layer of nearly every natural-language system: feed text in, convert each word to its embedding vector, and let the rest of the model work in that meaningful geometric space.

The deeper significance is that word embeddings were the first great success of representation learning for language — the idea that a machine could learn a useful representation of meaning from raw text alone, rather than being handed hand-built features. That idea is the seed of everything that followed. The large language models defining the current era of AI begin, at their input, by embedding words (or word-pieces) as vectors, and their entire achievement is, in a sense, a vastly more sophisticated and context-sensitive version of the same move this chapter describes: turn language into geometry, then compute. Microsoft's patent on latent semantic structure in co-occurrence has now expired, and the foundational technique of

computational meaning is public domain.

THE CODE

Everything starts with co-occurrence — tallying which words appear near which, weighted so that closer neighbours count for more:

```
def build_cooccurrence(sentences, vocab_size, window=5):
    """Count how often each word appears near each other word
    (closer = heavier)."""
    cooc = defaultdict(float)
    for sent in sentences:
        for pos, word in enumerate(sent):
            for offset in range(1, window + 1):
                if pos + offset < len(sent):
                    ctx = sent[pos + offset]
                    weight = 1.0 / offset      # nearer context words
                    weigh more
                    cooc[(word, ctx)] += weight
                    cooc[(ctx, word)] += weight
    return cooc
```

From those counts, the PMI transform turns raw co-occurrence into a measure of genuine association (controlling for how common each word is), and factorising the PMI matrix yields the dense embedding vectors. The library wraps this into a `PMIEmbeddings` model and a

word2vec-style Word2Vec model; both expose the geometry that makes embeddings useful:

```
from patentml import PMIEmbeddings, tokenise

sentences, w2i, vocab = tokenise(corpus_text)
model = PMIEmbeddings(dim=50, window=5)
model.fit(sentences, vocab_size=len(vocab))
```

Words with similar meaning are near each other in the learned space:

```
print(model.most_similar("king", w2i, vocab, topn=5))
```

And relationships are directions: king " man + wo
"queen"

```
print(model.analogy("king", "man", "woman", w2i, vocab))
```

Trained on enough text, the model places "king" and "queen" near each other and "cat" and "dog" near each other, with no dictionary and no labels — only the statistics of which words keep which company. And the analogy query, doing literal vector arithmetic in the learned space, recovers "queen," because the embedding has captured the male-to-female relationship as a consistent geometric direction. Meaning, made computable.

THE INHERITANCE

Reach for word embeddings whenever you need to turn words into a

representation a machine can compute with that respects meaning — as the input layer to almost any text-processing system, for measuring semantic similarity between words or documents, for clustering or classifying text, or for any task where "these two words mean similar things" must be something the machine can act on. Pre-trained embeddings turned countless language tasks from hard to tractable by handing models a meaningful starting representation, and the technique remains foundational even in the age of large language models, which are built on a more elaborate version of the same idea.

The deeper inheritance is conceptual: word embeddings are the clearest, most beautiful demonstration of the distributional hypothesis and of representation learning — meaning emerging from raw statistics, with no supervision. Understanding how co-occurrence becomes geometry, and geometry becomes analogy, is understanding the first principle on which modern language AI is built.

The public-domain, dependency-free form is especially valuable here, because embeddings are usually trained with heavyweight tooling on enormous corpora — yet the core algorithm, co-occurrence to PMI to factorisation, is small enough to read and to run on a modest corpus in pure Python. That makes it both a superb way to understand how machines learn the meaning of words, and a practical way to build compact, domain-specific embeddings where a

giant pre-trained model would be overkill or impossible — on a device, in an application, with no dependencies. Microsoft patented latent semantic structure in co-occurrence in 2000 and earned 645 citations; the patent has expired; the technique that turned language into geometry is now yours.

The next chapter puts embeddings and their simpler cousins to work on the oldest practical language task — reading a document and deciding what it is about — with the text classifier that powers spam filters, sentiment analysis, and topic sorting.

CHAPTER 15

THE BAG OF WORDS (TEXT CLASSIFICATION)

THE FILING

US Patent 6,192,360 — "Methods and apparatus for classifying text and for building a text classifier" using TF-IDF features and a naive Bayes classifier.

Assignee: Microsoft Corporation. Filed 1998. 364 forward citations. Now expired; public domain.

The previous chapter turned words into rich geometric vectors. This chapter steps back to the simpler, older, and astonishingly effective technique that solved most practical text problems first — and still solves many of them best. Text classification is the task of reading a document and assigning it a category: spam or not, positive or negative, sports or politics or finance. Microsoft's patent, cited 364 times, captures the classic recipe that dominated the field for years

and remains a superb default: represent each document as a bag of words weighted by TF-IDF, and classify it with naive Bayes. We met naive Bayes in Volume One as the engine of the spam filter; here it is paired with the TF-IDF representation that makes it shine on the full range of text-classification problems.

THE IDEA

The first challenge is representation: how do you turn a document — an ordered sequence of words of varying length — into a fixed bundle of numbers a classifier can digest? The bag of words model makes a bold simplification: throw away the order entirely and represent the document just by which words it contains and how often. "The cat sat on the mat" becomes a tally: the×2, cat×1, sat×1, and so on. This seems to discard a great deal — and it does — but for deciding the topic or category of a document, the words present matter far more than their arrangement, and the bag of words captures the essential signal with radical simplicity.

Raw counts, though, are misleading, because common words dominate. "The" appears in every document and tells you nothing; a rare, distinctive word tells you a great deal. TF-IDF — Term Frequency × Inverse Document Frequency — corrects for this. Term frequency rewards words that appear often in this document; inverse document frequency discounts words that appear in many documents across the whole collection. Multiply them, and each word gets a

weight that is high only if it is frequent here but rare overall — exactly the words that distinguish this document from the rest. "The" gets a near-zero weight everywhere; a word like "touchdown" gets a high weight in the sports articles where it appears and is absent elsewhere. TF-IDF is one of the most useful and enduring ideas in all of information retrieval, and it is the R in the relevance ranking behind classic search engines.

With documents represented as TF-IDF-weighted bags of words, naive Bayes (from Volume One) classifies them: it learns how characteristic each word is of each category, and scores a new document by combining the evidence of all its words, under the "naive" assumption that the words contribute independently. The assumption is false — words are correlated — but naive Bayes works remarkably well on text anyway, and it is fast, tiny, interpretable, and needs little training data. The combination — TF-IDF features into naive Bayes — is the classic text-classification pipeline that Microsoft's patent describes, and it is still the right first thing to try on any new text problem.

THE STORY

Text classification is one of the oldest applied problems in machine learning, driven from the start by real and pressing needs: sorting documents, routing email, filtering spam, organising the explosion of digital text. The TF-IDF representation has roots in information

retrieval research of the 1970s (Karen Spärck Jones's work on inverse document frequency is foundational), and the marriage of TF-IDF with naive Bayes became the workhorse of text classification through the 1990s and 2000s — the era of Microsoft's patent. It powered spam filters, news categorisation, sentiment analysis, and the topic-sorting behind early search and recommendation.

What is striking is how durable this simple recipe has proven. The deep-learning revolution brought powerful new methods to text — the embeddings of the last chapter, recurrent and transformer networks — and for the hardest language tasks those methods now dominate. But for straightforward classification on modest data, TF-IDF plus naive Bayes (or a linear classifier) remains shockingly competitive, often within a hair of far more complex models at a tiny fraction of the cost and complexity. It is the honest baseline that every fancy text model must beat, and frequently the pragmatic production choice when the problem is simple and the data is limited. Microsoft's patent on the TF-IDF-and-naive-Bayes text classifier has now expired into the public domain.

THE CODE

The TF-IDF vectoriser learns the vocabulary and the document frequencies, then transforms documents into weighted bags of words. The `patentml` text pipeline ties the vectoriser to the classifier; here is

the whole pipeline's interface, with the vectoriser learning which words distinguish documents:

```
class TextPipeline:
    def __init__(self, max_features=5000, min_df=2, alpha=1.0):
        self.vectoriser = TFIDFVectoriser(max_features, min_df) #
        bag of words ! TF-IDF weights
        self.classifier = NaiveBayesClassifier(alpha)           # the
        classic text classifier
    def fit(self, docs, labels):
        X = self.vectoriser.fit(docs)      # learn vocabulary + IDF,
        weight each document
        self.classifier.fit(X, labels)     # learn how characteristic each
        word is of each class
        return self
    def predict(self, docs):
        return self.classifier.predict(self.vectoriser.transform(docs))
```

Using it is a two-line affair — fit on labelled documents, predict on new ones:

```
from patentml import TextPipeline

docs = ["great film loved it", "wonderful and moving", "boring and
dull", "awful waste of time"]
labels = ["pos", "pos", "neg", "neg"]
```

```
pipe = TextPipeline()
pipe.fit(docs, labels)

print(pipe.predict(["a wonderful moving film"]))
print(pipe.predict(["dull and boring waste"]))
```

The pipeline tokenised each document into a bag of words, weighted the words by TF-IDF so that distinctive terms like "wonderful" and "boring" counted heavily while filler counted for nothing, learned from four tiny examples how each word leans, and correctly classified new sentences it had never seen — fast, transparent, and from almost no data. That efficiency on small data is exactly why the recipe endures.

THE INHERITANCE

Reach for the TF-IDF-and-naive-Bayes text classifier first, on any text-classification problem — spam, sentiment, topic, intent, routing. It is the honest baseline that sets the bar, it trains from modest data, it runs instantly, its decisions are interpretable (you can see which words drove a classification), and it is often nearly as good as anything fancier on straightforward problems. Before reaching for embeddings or a large language model, you should know what this simple recipe achieves on your problem — frequently it is enough, and reaching for heavier machinery would be wasted effort.

TF-IDF itself is a tool worth carrying beyond classification: it is the classic way to measure the relevance of words to documents, the backbone of keyword extraction and traditional search ranking, and a useful feature representation for clustering and similarity. The bag-of-words idea, for all its crudeness, remains a remarkably strong representation whenever word presence matters more than word order.

The public-domain, dependency-free form fits text classification naturally, because the whole pipeline — tokenise, weight by TF-IDF, score by naive Bayes — is small, fast, and free of heavy dependencies. You can classify text inside an application, on a device, in a browser extension, or in any locked-down setting with no machine-learning stack, and the readable implementation makes the enduring TF-IDF and naive-Bayes ideas transparent. Microsoft patented the TF-IDF text classifier in 1998 and earned 364 citations; the patent has expired; the simple recipe that reads and sorts text is now yours.

The next chapter turns from classifying content to predicting preference — the collaborative filter that recommends what you will like by learning from the tastes of people like you.

CHAPTER 16

PEOPLE LIKE YOU (COLLABORATIVE FILTERING)

THE FILING

US Patent 6,496,816 — "Collaborative filtering with mixtures of Bayesian networks"

Assignee: Microsoft Corporation. Filed 1998. 128 forward citations. Now expired; public domain.

Few machine-learning techniques have shaped daily life as directly as collaborative filtering — the engine behind "customers who bought this also bought," "recommended for you," and the suggestion queues of every major shopping, streaming, and social platform. Its premise is one of the most quietly powerful ideas in all of applied machine learning, and it requires no understanding of the items at all. It does not need to know what a film is about or what a product does. It needs only the pattern of who liked what, and from

that alone it can predict what you will like — by finding people whose tastes resemble yours and recommending what they enjoyed that you have not yet seen. Microsoft's patent frames this with a mixture-of-Bayesian-networks model; the library implements both that probabilistic approach and the classic neighbourhood method.

THE IDEA

Imagine a giant table: rows are users, columns are items, and each filled-in cell is a rating — how much that user liked that item. The table is enormously sparse, because each user has rated only a tiny fraction of all items. The recommendation problem is to fill in the blanks: predict how much a user would like an item they have not yet rated, so you can recommend the items they would rate highest. Collaborative filtering does this by exploiting the patterns of agreement among users, and there are two classic approaches.

The memory-based (or neighbourhood) approach is the most intuitive. To predict how much you would like an item, find the users most similar to you — those whose past ratings most agree with yours — and look at how they rated that item. If people who share your taste loved it, you probably will too. Similarity is measured between users' rating patterns, typically by cosine similarity, which compares the direction of two users' rating vectors regardless of how harshly or generously each tends to rate. The prediction is then a similarity-weighted average of your neighbours' ratings: the more a

neighbour resembles you, the more their opinion counts. This is, in essence, the k-nearest-neighbours idea from Volume One, applied to taste — "people like you also liked this."

The model-based approach, which Microsoft's patent represents, instead builds a compact model of the rating patterns. The mixture-of-Bayesian-networks version assumes there are a few hidden types of user — clusters of taste — and that each user is a soft blend of these types, with each type having characteristic preferences over items. Learning the model (by the expectation-maximisation idea from this volume's clustering chapter) discovers the latent taste-types and each user's membership in them, and predictions follow from the model. A closely related and hugely influential model-based idea is matrix factorisation — decomposing the giant sparse rating table into compact "taste vectors" for users and items, so that a user's predicted rating for an item is just the alignment between their vectors. (This connects directly to the next chapter's subject; the famous Netflix Prize was won largely with matrix-factorisation collaborative filtering.) Model-based methods scale better and generalise more gracefully than neighbourhood methods, at the cost of a training step.

The deep magic, in both approaches, is that collaborative filtering is content-blind. It never examines the items themselves — it works purely from the structure of who agreed with whom. This is its great strength (it works for any kind of item, even ones too subtle to

describe in features) and the source of its famous weakness, the cold-start problem: a brand-new user or item, with no ratings yet, has no pattern to exploit, and must be handled by other means until the ratings accumulate.

THE STORY

Collaborative filtering emerged in the early 1990s, with pioneering systems at Xerox PARC (filtering email) and the GroupLens project (recommending Usenet news), and it grew up alongside electronic commerce, which gave it both its data and its purpose. As online shopping and, later, streaming exploded, the recommendation engine became a core piece of business infrastructure — by some estimates responsible for a large fraction of what people buy and watch on the major platforms. Microsoft's 1998 patent sits at the start of this era, with a sophisticated probabilistic take on the problem.

The defining public moment came with the Netflix Prize (2006–2009), a million-dollar competition to improve Netflix's recommendations, which galvanised the field and established matrix factorisation as the dominant technique — compact latent taste-vectors for users and items, learned from the sparse ratings, often combined into the ensembles of Volume One's voting chapter. The lessons of that competition shaped a decade of industrial recommender systems. Modern recommendation has since grown more elaborate — deep-learning recommenders, hybrid systems that

blend collaborative signals with item content and context — but the collaborative-filtering core, predicting preference from the pattern of who-liked-what, remains central. Microsoft's patent has now expired into the public domain.

THE CODE

The neighbourhood method rests on measuring how similar two users' tastes are, by the cosine of their rating vectors over the items they have both rated:

```
def _cosine(a, b):
    """Similarity of two users by the items they have both rated."""
    keys = set(a) & set(b)           # items both users rated
    if not keys:
        return 0.0
    dot = sum(a[k] * b[k] for k in keys)
    norm_a = math.sqrt(sum(v * v for v in a.values()))
    norm_b = math.sqrt(sum(v * v for v in b.values()))
    return dot / (norm_a * norm_b) if norm_a and norm_b else 0.0
```

The memory-based filter learns from a list of (user, item, rating) interactions and predicts an unseen rating as the similarity-weighted average of the most similar users' ratings:

```
from patentml import MemoryCF
```

```
interactions = [  
    (1, 101, 5.0), (1, 102, 3.0),  
    (2, 101, 4.0), (2, 103, 5.0),    # user 2's tastes overlap user 1's  
    (3, 102, 2.0), (3, 103, 4.0),  
]  
cf = MemoryCF(k_neighbours=2)  
cf.fit(interactions)
```

Predict how much user 1 would like item 103, from users similar to user 1:

```
print("predicted rating:", round(cf.predict(1, 103), 2))  
print("top recommendations for user 1:", cf.recommend(1, n=2))
```

The filter never looked at what items 101, 102, or 103 are — it knew only the ratings. To predict user 1's rating for item 103, it found that user 2 shares user 1's taste (they agree on item 101) and leaned on user 2's enthusiasm for item 103. From the bare pattern of who-liked-what, it produced a recommendation — the content-blind magic that drives the suggestion queues you see every day. The library's BayesianCF offers the model-based alternative, discovering latent taste-types and predicting from them.

THE INHERITANCE

Reach for collaborative filtering whenever you have interaction data — ratings, purchases, clicks, plays — and want to recommend items

to users, and especially when the items are hard to describe in features but the patterns of preference are rich. Its content-blindness is a genuine superpower: it works for any kind of item, surfacing non-obvious recommendations ("people who liked this obscure thing also liked that other obscure thing") that a content-based system, reasoning only from item features, would never find. For the core recommendation problem, collaborative filtering is the foundational approach, and matrix factorisation its most powerful classic form.

Mind the cold-start problem — new users and items with no history need a fallback (popularity, content features, or a few solicited preferences) until their ratings accumulate — and remember that the model-based and matrix-factorisation variants scale and generalise better than raw neighbourhood methods for large catalogues. The connection to the next chapter is worth holding: matrix factorisation, the winning recommender technique, is the same low-rank decomposition idea that the SVD chapter is about.

The public-domain, dependency-free form makes a real recommendation engine available with no infrastructure at all — the similarity computation and the weighted average are a handful of lines, small enough to run inside an application, on a device, or in a service with no machine-learning stack. It is also the clearest way to understand the idea that quietly shapes so much of what people buy and watch. Microsoft patented collaborative filtering with Bayesian mixtures in 1998 and earned 128 citations; the patent has expired;

the engine of "recommended for you" is now yours.

The next chapter goes underneath recommendation to the mathematical idea that powers its most successful form — and powers dimensionality reduction across all of machine learning: the matrix factorisation that finds the few dimensions that truly matter.

CHAPTER 17

SHRINKING THE MATRIX (RANDOMISED SVD & PCA)

THE FILING

US Patent 6,807,536 — "Methods and systems for computing singular value decompositions of matrices", using randomised sampling to make the computation tractable for high-dimensional data.

Assignee: Microsoft Corporation. Filed 2001. 170 forward citations. Now expired; public domain.

The companion US Patent 6,529,891 (Microsoft, 1998, 124 cites) covers Bayesian model selection — choosing how many dimensions to keep. Both now expired; public domain.

This chapter is about one of the most useful and unifying ideas in all of machine learning: dimensionality reduction, the art of taking

high-dimensional data — thousands of features per example — and finding the few underlying dimensions that actually carry the information. The mathematical engine behind it is the singular value decomposition (SVD) and its close relative principal component analysis (PCA), and Microsoft's patent addresses the practical problem of computing it efficiently for large, high-dimensional data through randomisation. This is not a niche technique; it is connective tissue. The matrix factorisation that won the Netflix Prize in the last chapter, the factorisation that produces word embeddings two chapters ago, the compression of images and signals, the de-noising of data, the visualisation of high-dimensional datasets — all are SVD and PCA at work.

THE IDEA

High-dimensional data is usually far simpler than it looks. A dataset with a thousand features rarely has a thousand independent directions of variation; the features are correlated, redundant, entangled. The true structure — the part that actually varies and matters — often lives in a much smaller number of dimensions. Dimensionality reduction finds those dimensions.

Principal component analysis is the most intuitive version. Imagine your data as a cloud of points in high-dimensional space. PCA finds the directions along which the cloud is most spread out — the directions of greatest variance. The first principal component is the

single direction capturing the most variation; the second is the next-most, at right angles to the first; and so on. Usually, a handful of these components capture the overwhelming majority of the variation in the data, and the rest is mostly noise. By keeping only the top few components and projecting the data onto them, you compress a thousand-dimensional dataset down to, say, ten dimensions while preserving almost all of its meaningful structure. You have found the few axes that matter and discarded the redundant rest.

The singular value decomposition is the deeper, more general engine underneath. It factorises any matrix into three pieces that reveal its essential structure: a set of fundamental patterns, a set of weights (the singular values) saying how important each pattern is, and how those patterns combine to reconstruct the data. Keeping only the patterns with the largest singular values gives the best possible low-rank approximation — the closest simpler matrix to your original — which is exactly what compression, de-noising, and latent-factor discovery all need. PCA is essentially SVD applied to centred data. And the same factorisation, applied to a user-item rating table, is the matrix-factorisation collaborative filtering of the last chapter (the singular vectors become the latent taste-vectors); applied to a word co-occurrence matrix, it produces the word embeddings of three chapters ago. One factorisation, many faces.

The catch the patent addresses is cost. Computing an exact SVD of a

huge matrix is expensive. Microsoft's contribution is randomisation: by randomly sampling or projecting the matrix first, you can compute an approximate SVD that captures the dominant structure at a fraction of the cost — and since you only want the top few components anyway, the approximation loses almost nothing. Randomised SVD made dimensionality reduction practical for the large, high-dimensional data of the modern era, and the patentml implementation computes it by power iteration in pure Python, with no linear-algebra library at all.

THE STORY

The singular value decomposition is a cornerstone of linear algebra with a long mathematical history, but its central role in data analysis grew through the twentieth century — PCA dates to Pearson and Hotelling in the early 1900s — and exploded with the rise of high-dimensional data. Latent semantic analysis applied SVD to text in the 1990s; image compression and signal processing leaned on low-rank approximation; and, as the last chapter recounted, matrix factorisation became the heart of recommender systems. Microsoft's 2001 patent on randomised SVD belongs to the moment when datasets grew large enough that the classical exact algorithms became a bottleneck, and randomisation was the key that kept the technique practical at scale — an idea that has only grown in importance and is now standard in large-scale numerical computing.

Dimensionality reduction endures because the problems it solves are perennial. Compression: store and transmit data with far fewer numbers. De-noising: the signal lives in the dominant components, the noise in the discarded ones, so reconstructing from the top components cleans the data. Visualisation: project high-dimensional data down to two or three dimensions to actually see its structure. Feature extraction: turn redundant raw features into a compact, informative set before feeding a model. Latent-factor discovery: uncover the hidden dimensions — taste-types, topics, modes of variation — that generate the data. Even in the deep-learning era, where neural networks learn their own representations, SVD and PCA remain everywhere as preprocessing, analysis, and compression tools. Microsoft's randomised-SVD patent has now expired into the public domain.

THE CODE

The `patentml` randomised SVD finds the dominant directions by power iteration — repeatedly multiplying through the matrix to amplify its strongest patterns — with no external numerical library. The building blocks are plain vector operations:

```
def _mat_vec(A, v):  
    return [_dot(row, v) for row in A]          # the core operation,  
repeated to find structure
```

The `randomised_svd` and `pca` functions wrap this into the full decomposition. From the user's side, PCA reduces high-dimensional data to its most informative few dimensions in a couple of lines:

```
from patentml import randomised_pca, svd_transform
```

Reduce 50-dimensional data to its 2 most informative directions

```
components, mean = randomised_pca(X, n_components=2)
```

```
X_reduced = svd_transform(X, components, mean)
```

```
for original, compressed in zip(X[:3], X_reduced[:3]):
```

```
    print(f"{len(original)}-D -> {len(compressed)}-D : {[round(v, 2)  
for v in compressed]}")
```

Fifty correlated features collapse to two coordinates that capture most of the data's real variation — a compression that throws away redundancy and noise while preserving structure. Those two dimensions can be plotted to see the data, fed to a downstream model as compact features, or used to reconstruct a de-noised approximation of the original. The same `randomised_svd` underneath is what factorises a rating table into taste-vectors or a co-occurrence table into embeddings — the one decomposition wearing its many hats.

THE INHERITANCE

Reach for PCA and SVD whenever you have high-dimensional data and suspect its real structure is lower-dimensional — which is almost always. Use them to compress data into fewer numbers, to de-noise by keeping only the dominant components, to visualise high-dimensional datasets by projecting to two or three dimensions, to extract features before modelling, and to discover latent factors — the hidden dimensions behind ratings, words, or measurements. Dimensionality reduction is one of the most broadly useful tools in the entire field, a near-universal preprocessing and analysis step, and PCA is very often the first thing to do with an unfamiliar high-dimensional dataset just to see what is there.

The unifying inheritance is the deepest lesson of this chapter: the same low-rank factorisation powers collaborative filtering's taste-vectors, word embeddings' meaning-geometry, image and signal compression, and latent-topic discovery. To understand SVD is to see the common mathematical heart beneath techniques that look unrelated on the surface. Microsoft's companion patent on Bayesian model selection even addresses the natural follow-on question — how many dimensions to keep — by penalising complexity, the same Occam's-razor discipline that recurred throughout Volume One.

The public-domain, dependency-free form is a genuine achievement here, because SVD and PCA are the techniques most assumed to require a heavyweight numerical library — yet `patentml` computes a

randomised SVD by power iteration in pure Python. That puts dimensionality reduction within reach on a device or in a browser with no scientific stack, and lays bare the power-iteration machinery for anyone wanting to understand the factorisation at the heart of so much of machine learning. Microsoft patented randomised SVD in 2001 and earned 170 citations; the patent has expired; the engine that finds the few dimensions that matter is now yours.

The final chapter of this part takes compression in a different direction — not finding continuous dimensions but building a small vocabulary of representative prototypes to which any data point can be snapped: vector quantisation, the codebook idea that compresses signals and underlies the quantised search at the frontier of modern AI.

CHAPTER 18

THE CODEBOOK (VECTOR QUANTISATION)

THE FILING

US Patent 6,421,467 — "Adaptive vector quantization/quantizer", refining a codebook so that each codeword minimises the average distortion over the vectors assigned to it.

Assignee: Texas Tech University. Filed 1999. 101 forward citations. Now expired; public domain.

The companion US Patent 5,719,950 (3M, 1995, 445 cites) covers classification by clustering. Both now expired; public domain.

The last chapter compressed data by finding continuous dimensions. Vector quantisation compresses in a complementary way: it builds a small vocabulary of representative prototype vectors — a codebook — and then represents any data point not by its full coordinates but

by the single nearest codeword in that vocabulary. Instead of storing a vector, you store the index of the prototype it most resembles. It is the vector generalisation of rounding: just as you might round 3.7 to the nearest integer 4, vector quantisation rounds a whole vector to the nearest codeword. Texas Tech's patent describes the adaptive refinement of such a codebook. The idea is decades old, central to the compression of audio, images, and speech — and, in a striking modern turn, it is the same idea now powering the quantised vector search at the heart of today's AI retrieval systems.

THE IDEA

Suppose you have a stream of vectors — chunks of an audio signal, blocks of an image, feature vectors of any kind — and you want to compress them. Storing every vector in full is expensive. Vector quantisation observes that the vectors are not spread uniformly; they cluster. So you choose a modest number of representative prototypes — the codebook — positioned at the centres of those clusters, and you approximate each incoming vector by the nearest codeword. Now, instead of transmitting or storing the full vector, you store only its codeword's index. If your codebook has 256 entries, any vector — however high-dimensional — collapses to a single byte: the index of its nearest prototype. Decompression is a lookup: replace each index with its codeword. The compression can be enormous; the cost is some distortion, the gap between each original vector and the codeword that stands in for it, and the whole art is designing a

codebook that minimises that distortion.

How do you build a good codebook? This is exactly the clustering problem from this volume's first part — the codewords are cluster centres, and you want them placed so that every data vector is close to its nearest one. The classic method, the LBG algorithm (Linde–Buzo–Gray) implemented here, is a clever, structured form of this. It starts with a single codeword (the global average of all the data) and then repeatedly splits: each codeword is perturbed into two nearby ones, and a round of k-means-style refinement (assign every vector to its nearest codeword, then move each codeword to the average of its assigned vectors) settles them into place. Split, refine, split, refine — doubling the codebook each time — until it reaches the target size. This growing-by-splitting approach gives better coverage than scattering codewords randomly, building a codebook that adapts to the data's structure, exactly as Texas Tech's patent describes: each codeword adaptively updated to minimise the average distortion over the vectors assigned to it.

The relationship to the rest of the book is worth naming. Vector quantisation is clustering used for compression — k-means' codebook of centroids, put to work as a compression vocabulary. It is also the discrete cousin of the dimensionality reduction in the last chapter: where SVD compresses by finding continuous low-dimensional coordinates, vector quantisation compresses by snapping to a discrete set of prototypes. The two are often combined

— reduce dimensions, then quantise — and that combination is precisely what powers the modern systems at the end of this story.

THE STORY

Vector quantisation was a major topic in signal processing and data compression from the 1980s, with the LBG algorithm (1980) as its workhorse. It became fundamental to the compression of speech, audio, and images: representing blocks of a signal by codebook indices is a powerful lossy-compression strategy, and vector quantisation appeared in speech codecs, image-compression schemes, and early video standards. Texas Tech's 1999 patent on adaptive quantisation belongs to this rich tradition of squeezing signals down to a vocabulary of prototypes. The codebook idea also underlies the "bag of visual words" approach in computer vision, where image patches are quantised to a visual vocabulary, and it connects to the self-organising maps and competitive-learning methods of neural-network history.

Then came an unexpected and very modern second act. The AI systems of today depend on vector search — finding, among millions or billions of high-dimensional embedding vectors (the word and document embeddings of earlier chapters), the ones nearest to a query. Doing this exactly is expensive, and the dominant solution is product quantisation, a sophisticated descendant of vector quantisation that splits each vector into pieces and quantises each

piece against its own small codebook, achieving massive compression and lightning-fast approximate search. This is the technology inside the vector databases that let large language models retrieve relevant context — the "retrieval" in retrieval-augmented generation. The codebook idea that compressed 1980s telephone speech is, in refined form, what lets a modern AI system search a billion embeddings in milliseconds. The `patentml` library includes a `ProductQuantiser` alongside the classic LBG, and Texas Tech's adaptive-quantisation patent has now expired into the public domain.

THE CODE

The LBG algorithm grows a codebook by splitting and refining. The refinement step is pure clustering — assign each vector to its nearest codeword, then move each codeword to the centre of its assigned vectors:

```
def _centroid(vectors):  
    n, d = len(vectors), len(vectors[0])  
    return [sum(v[j] for v in vectors) / n for j in range(d)] # the heart  
of refinement
```

The full LBG fit starts from one codeword and doubles the codebook by perturbing-and-refining until it reaches the target size. Using it, you learn a codebook from data and then quantise any vector to its nearest codeword's index:

```

from patentml import LBG

lbg = LBG(n_codewords=16, seed=1)
lbg.fit(training_vectors)          # learn 16 representative prototypes

code = lbg.encode(some_vector)    # !' a single
massive compression
recon = lbg.decode(code)          # !' the code
the original
print(f"vector compressed to index {code}; reconstruction error
{lbg.distortion([some_vector]):.3f}")

```

A high-dimensional vector collapses to a single small integer — the index of its nearest prototype — and is reconstructed, approximately, by looking that index up in the codebook. The codebook, learned by splitting and refining, sits at the centres of the data's natural clusters so that the reconstruction error stays small. That trade — a tiny, controllable distortion for an enormous reduction in size — is the essence of vector quantisation, whether it is compressing a voice call or, in its product-quantisation form, packing a billion AI embeddings into a searchable index.

THE INHERITANCE

Reach for vector quantisation when you need to compress vectors to a small vocabulary of prototypes — lossy compression of signals,

images, audio, or feature vectors; turning continuous data into discrete symbols (a codebook index) for downstream processing; or building the "bag of prototypes" representations used in vision and beyond. Whenever "represent each item by its nearest of a few hundred prototypes" is an acceptable approximation, vector quantisation delivers dramatic compression with controllable, well-understood distortion.

Its most consequential modern inheritance is approximate nearest-neighbour search at scale — the product-quantisation descendants that power the vector databases behind today's AI retrieval systems. Understanding the codebook idea here, where a vector becomes an index into a learned vocabulary, is understanding the seed of the technology that lets modern AI search billions of embeddings in real time. And the connection to clustering and to the previous chapter's dimensionality reduction completes a circle: vector quantisation is clustering put to work as compression, the discrete partner of SVD's continuous reduction, and the two are routinely combined.

The public-domain, dependency-free form suits vector quantisation naturally, because the codebook is a small list of prototypes and encoding is a nearest-neighbour lookup — light enough to run inside a codec on a device, an embedded compressor, or a browser, with no heavy dependencies. The readable LBG implementation also makes the elegant split-and-refine codebook design transparent, and the

product quantiser shows how the same idea scales to modern vector search. Texas Tech patented adaptive vector quantisation in 1999 and earned 101 citations; the patent has expired; the codebook idea — from 1980s speech to today's billion-vector search — is now yours.

That completes *The Meaning*, and with it the five movements of Volume Two. We have turned words into geometry, read and sorted text, recommended by taste, compressed by finding the dimensions that matter, and built codebooks of representative prototypes. One chapter remains: a capstone that assembles the unsupervised and decision-making parts of both volumes into a single working system, and closes the series.

PART VI — PUTTING IT TOGETHER

. . .

CHAPTER 19

A SYSTEM THAT FINDS ITS OWN STRUCTURE

Volume One closed by building a supervised pipeline — features in, a known answer out — and then noting the assumption beneath it: that someone had already provided the answers. Volume Two has spent its whole length working without that luxury. This capstone assembles the unsupervised and decision-making parts of both volumes into a single system that takes in raw, unlabelled data and produces structure, insight, and action — with no answer key anywhere. Every component is drawn from an expired patent; the whole thing runs in pure Python with no dependencies.

THE ANATOMY OF AN UNSUPERVISED SYSTEM

When no one hands you labels, a working system still has a natural shape, and we have met every part of it.

First, reduce and represent. Raw data is high-dimensional and noisy. Before anything else, compress it to the dimensions that matter — the PCA and SVD of Chapter 16 — both to remove redundancy and noise and to make everything downstream faster and clearer. If the data is text, the representation step is the embeddings of Chapter 13 or the TF-IDF of Chapter 14, turning words into vectors first.

Second, find the structure. With the data in a clean, compact representation, discover the groups within it — the clustering of Chapters 1 through 5. Choose the method by what you know: k-means when you expect round groups and can guess their number, DBSCAN or mean shift when you cannot, a Gaussian mixture when you want soft, probabilistic membership.

Third, watch for the abnormal. Not every point belongs to a clean group. The anomaly detection of Chapter 9 flags the points that fit no pattern — the outliers, the novelties, the errors — which are often exactly what you care about, and which the clustering step would otherwise distort.

Fourth, model the sequence, if there is one. If your data unfolds over time, the Kalman filter and hidden Markov model of Chapters 6 and 7 track the hidden state behind the stream.

Fifth, decide and act. Finally, turn understanding into action under uncertainty — the bandits and reinforcement learners of Chapters 10

through 12 — choosing what to do next and learning from the consequences, closing the loop.

BUILDING IT

Here is an unsupervised pipeline assembled entirely from public-domain parts: reduce the dimensions, cluster the structure, and flag the anomalies — all without a single label.

```
from patentml import (StandardScaler, randomised_pca,  
svd_transform,
```

```
ScalableKMeans, IsolationForest)
```

1. Represent — scale, then reduce to the dimensions that carry the signal

```
scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(raw_data)
```

```
components, mean = randomised_pca(X_scaled, n_components=5)
```

```
# Chapter 16
```

```
X_reduced = svd_transform(X_scaled, components, mean)
```

2. Find the structure — no labels, no supervision

```
clusterer = ScalableKMeans(k=4, random_seed=1) #
```

```
Chapter 1
```

```
clusterer.fit(X_reduced)
```

```
groups = clusterer.predict(X_reduced)
```

3. Watch for the abnormal — points that fit no group

```
detector = IsolationForest(n_estimators=100, random_seed=1) #
```

Chapter 9

```
detector.fit(X_reduced)
```

```
anomaly_scores = detector.score_samples(X_reduced)
```

```
for i in range(len(raw_data)):
```

```
    flag = " <-- ANOMALY" if anomaly_scores[i] > 0.6 else ""
```

```
    print(f"point {i}: group {groups[i]}{flag}")
```

Read what happened. Raw, unlabelled, high-dimensional data was scaled and compressed to its five most informative dimensions; the points were sorted into four discovered groups that no one defined; and the points belonging to no real group were flagged as anomalies. Nobody labelled a single example. The system found the structure itself, and told you both what is normal — the groups — and what is not — the anomalies. Add the sequence models of Part II if the data flows over time, and the decision-makers of Part IV to act on what you find, and you have the skeleton of a real unsupervised intelligence system, every bone of it from a patent that has expired into the public domain. No frameworks. No dependencies. No permission required.

THE SERIES COMPLETE

Step back and see the whole of what these two volumes have built.

Volume One taught machines to predict from examples — to breed solutions, to learn by gradient, to classify with the great workhorses of applied machine learning. Volume Two taught them to work where no examples are given — to find structure in unlabelled data, to infer hidden state through time, to reason about uncertainty, to act from consequences, and to handle the human world of language and meaning. Together they span the working core of the field: supervised and unsupervised, prediction and decision, the labelled and the unlabelled.

And every one of them — every optimiser, every network, every classifier, clusterer, filter, recommender, and learner across both volumes — came from a patent that has expired into the public domain. The institutions that built modern machine learning chose, decades ago, to patent rather than to hide. They bought their twenty years of monopoly, and in exchange they told the world, in permanent and exhaustive detail, exactly how each technique works. The monopolies are spent. The teaching remains, and it belongs to everyone now — to you. The `patentml` library is one person's attempt to make that inheritance not just legally free but practically free: readable, runnable, dependency-free, ready to drop into a device or a browser or a classroom.

The final chapter steps back from the algorithms to make the case for why this matters — why, in an age of vast and heavy AI, there remains a deep and growing value in machine learning that is small,

transparent, and free.

The final chapter follows: Where Free Beats Heavy.

CHAPTER 20

WHERE FREE BEATS HEAVY

We have reached the end of the algorithms. This last chapter is not about a patent; it is about a thesis that has run quietly beneath every page of both volumes, and deserves to be stated plainly now that the evidence is in. The thesis is this: in an age of enormous, heavy, opaque artificial intelligence, there is a deep and growing value in machine learning that is small, transparent, and free — and the public-domain inheritance these books have toured is exactly that.

The dominant story of AI in this era is a story of scale. Models with hundreds of billions of parameters, trained on data centres full of specialised chips, served from the cloud, accessed through an API, owned by a handful of large organisations. That story is real and its achievements are genuine. But it is not the whole story, and treating it as if it were leaves enormous and important territory unserved. There are places the heavy machinery cannot go, problems it is the

wrong tool for, and values it cannot honour. In those places, free beats heavy.

The edge. Vast amounts of computing happen on small, disconnected, resource-constrained hardware — microcontrollers in sensors and appliances, flight controllers in drones, chips in medical and industrial devices, anything that must run without a network and within a tiny power and memory budget. A model that needs a data centre is useless here. A Kalman filter, a decision tree, a naive Bayes classifier, a k-means clusterer, a small reinforcement learner — these run in kilobytes, on a microcontroller, in real time, forever. The dependency-free algorithms in these volumes are not a compromise for the edge; for the edge, they are frequently the right and only answer. The Kalman filter steadied Apollo on the computing power of a pocket calculator. It still steadies the drone in your hand.

The locked-down and the air-gapped. A great deal of serious computing happens in environments that forbid the heavy stack — regulated industries wary of supply-chain risk, secure systems with no internet, corporate machines where you cannot install a 200-megabyte library with a hundred transitive dependencies, classified or critical infrastructure that must be auditable line by line. A machine-learning library that is pure standard-library Python — no compiled extensions, no dependency tree, no network calls, readable in an afternoon — is not just convenient in these settings; it is sometimes the only thing that can be permitted at all. Provenance

matters here too: every algorithm in these books traces to a documented, expired, public-domain patent, which is a cleaner pedigree than most code can claim.

The browser and the classroom. Pure-Python, dependency-free code runs in the browser through Pyodide, putting real machine learning into a web page with no server and no cost. And for learning — for the student, the curious engineer, the practitioner who wants to truly understand rather than merely call — there is nothing like code you can read in full. Every algorithm in these volumes can be opened, read end to end, instrumented with a print statement anywhere, broken, fixed, and understood. The heavy frameworks hide their workings inside compiled kernels; these files hide nothing. To learn how a thing works, you must be able to see it work, and that is a value no amount of scale can provide.

The simply-sufficient. And then there is the largest territory of all: the enormous number of real problems that do not need the heavy machinery. A spam filter, a recommendation, a customer segmentation, a fraud flag, a forecast on a spreadsheet of features — for a vast range of practical problems, a gradient-boosted tree or a naive Bayes classifier or a k-means clustering is not a poor substitute for a giant model; it is as good or better, at a thousandth of the cost and complexity, with results you can explain. Reaching for a billion-parameter model to solve a problem that a decision tree handles is not sophistication; it is waste. Knowing when not to reach

for the heavy tool is one of the marks of a real practitioner, and these volumes are, among other things, a catalogue of when the simple tool is the right one.

None of this is an argument against scale. The large models are extraordinary and they will keep advancing. It is an argument for completeness — for remembering that the field is wider than its frontier, that most problems are not frontier problems, and that the foundations matter as much as the heights. The algorithms in these two volumes are those foundations. They are what the frontier was built on. The gradient descent that trains the largest model is the gradient descent of Volume One, Chapter 10. The embeddings at a language model's input are the word vectors of Volume Two, Chapter 13. The vector search behind its retrieval is the quantisation of Chapter 17. The reinforcement learning that aligns it is the actor-critic of Chapter 12. To understand these foundations is to understand the frontier, and to possess them — freely, readably, runnably — is to be able to build, learn, and reason without permission.

That is the final inheritance. The institutions that built modern machine learning patented its foundations and, in doing so, published them forever. The monopolies have expired. What remains is a vast, free, public-domain library of the ideas that made the field — and a chance, for anyone willing to read, to own them. These two volumes have been a guided tour of that inheritance, on the long morning after

the vault unlocked. The code is installed with a single line. The ideas are yours. Go and build something.

APPENDIX A: THE PATENTS OF VOLUME TWO

Every algorithm in this volume traces to an expired United States patent, readable in full on Google Patents. Citation counts are forward citations — later patents citing this one as prior art.

Chapter	Patent	Assignee	Filed	Cites	Subject
---	---	---	---	---	---
1	US6012058	Microsoft	1998	—	Scalable k-means clustering
2	US6049797	Lucent	1998	154	DBSCAN density clustering + noise
3	US6263337	Microsoft	1998	142	Scalable EM / Gaussian mixtures
4	US6216066	General Electric	1998	148	Spectral clustering (Laplacian)
5	US6317707	AT&T	1998	269	Mean shift + kernel density

| 6 | US6795794 | University of Illinois | 2002 | 127 | Kalman +
Extended Kalman filter |
| 7 | US7076102 | Philips | 2002 | 116 | Hidden Markov model event
learning |
| 8 | US6931384 | Microsoft | 2001 | 258 | Gaussian process
regression |
| 9 | US7225343 | Columbia University | 2003 | 159 | Isolation forest /
anomaly detection |
| 10 | US6981040 | Utopy | 2000 | 919 | Multi-armed bandits |
| 11 | US6985172 | Southwest Research | 2002 | 145 | Q-learning /
temporal difference |
| 12 | US6985172 | Southwest Research | 2002 | 145 | Actor-critic
policy gradient |
| 13 | US6556983 | Microsoft | 2000 | 645 | Word embeddings /
co-occurrence |
| 14 | US6192360 | Microsoft | 1998 | 364 | TF-IDF + naive Bayes
text classifier |
| 15 | US6496816 | Microsoft | 1998 | 128 | Collaborative filtering |
| 16 | US6807536 | Microsoft | 2001 | 170 | Randomised SVD / PCA |
| 17 | US6421467 | Texas Tech | 1999 | 101 | Vector quantisation /
codebooks |

APPENDIX B: A CITATION MAP

A rough family tree of influence across both volumes. The most-cited patents — SmartTouch's voting ensemble (1,182), Utopy's bandits (919), Microsoft's online classifier (896) and word embeddings (645) — are the roots from which much later work grew. Ideas recur and connect: the EM of Volume Two's clustering chapter trains the hidden Markov models of its sequence chapter; the matrix factorisation of the SVD chapter powers both the collaborative filtering and the word embeddings; the k-means of the clustering chapter becomes the codebook of vector quantisation; the gradient descent of Volume One trains every network in both. The field is a web, not a list, and the patent record traces its threads.

APPENDIX C: RUNNING THE CODE

Install once — `pip install patentml` — and every listing in this volume is importable and runnable with no other dependency. Each algorithm is a single self-contained pure-Python file; find the one behind a chapter by the module named in its code listings. Because the whole library is standard-library Python, it runs unmodified on a server, a Raspberry Pi, or — via Pyodide — in a browser, and can be trimmed for a microcontroller. Run any module directly to see its built-in worked example.

CHAPTER 21

INDEX OF ALGORITHMS · INDEX OF COMPANIES AND INVENTORS

(Compiled at typesetting.)

End of Volume Two — Patterns and Decisions.

End of The Lost Algorithms.

The institutions that built modern machine learning patented its foundations and, in doing so, published them forever. The monopolies have expired. The ideas are yours.