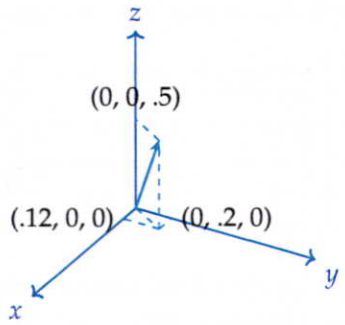


Figure 19: *Projecting a vector into the 3d space*

encode geography as some sort of numerical value so that the model can calculate them as inputs[14]. So, once we have a combination of vectors, we can compare it to other points. So in our case, each row of data tells us where to position each bird in relation to any other given bird based on the combination of features. And that's really what our numerical features allow us to do.

2.6 From Words to Vectors in Three Easy Pieces

In "Operating Systems: Three Easy Pieces", the authors write, "Like any system built by humans, good ideas accumulated in operating systems over time, as engineers learned what was important in their design." [3] Today's large language models were likewise built on hundreds of foundational ideas over the course of decades. There are, similarly, several fundamental concepts that make up the work of transforming words to numerical representations.

These show up over and over again, in every deep learning architecture and every NLP-related task[15]:

- **Encoding** - We need to represent our non-numerical, multimodal data as numbers so we can create models out of them. There are many different ways of doing this.
- **Vectors** - we need a way to store the data we have encoded and have the ability to perform mathematical functions in an optimized way on them. We store encodings as vectors, usually floating-point representations.
- **Lookup matrices** - Often times, the end-result we are looking for from encoding and embedding approaches is to give some approximation about the shape and format of our text, and we need to be able to quickly go from numerical to word representations across large chunks of text. So we use lookup tables, also known as hash

[14]There are some models, specifically decision trees, where you don't need to do text encoding because the tree learns the categorical variables out of the box, however implementations differ, for example the two most popular implementations, scikit-learn and XGBoost [1], can't.

[15]When we talk about tasks in NLP-based machine learning, we mean very specifically, what the machine learning problem is formulated to do. For example, we have the task of ranking, recommendation, translation, text summarization, and so on.

Table 2: *Tabular Input Data for Flutter Users*

bird_id	bird_posts	bird_geo	bird_likes
012	2	US	5
013	0	UK	4
056	57	NZ	70
612	0	UK	120

We start by selecting our model features and arranging them in tabular format. We can formulate this data as a table (which, if we look closely, is also a matrix) based on rows of the bird id and our bird features.

Tabular data is any structured data. For example, for a given Flutter user we have their user id, how many posts they've liked, how old the account is, and so on. This approach works well for what we consider traditional machine learning approaches which deal with tabular data. As a general rule, the creation of the correct formulation of input data is perhaps the heart of machine learning. I.e. if we have bad input, we will get bad output. So in all cases, we want to spend our time putting together our input dataset and engineering features very carefully.

These are all discrete features that we can feed into our model and learn weights from, and is fairly easy as long as we have numerical features. But, something important to note here is that, in our bird interaction data, we have both numerical and textual features (bird geography). So what do we do with these textual features? How do we compare "US" to "UK"?

The process of formatting data correctly to feed into a model is called **feature engineering**. When we have a single continuous, numerical feature, like "the age of the flit in days", it's easy to feed these features into a model. But, when we have textual data, we need to turn it into numerical representations so that we can compare these representations.

2.5 Numerical Feature Vectors

Within the context of working with text in machine learning, we represent features as numerical vectors. We can think of each row in our tabular feature data as a vector. And a collection of features, or our tabular representation, is a matrix. For example, in the vector for our first user, `[012, 2, 'US', 5]`, we can see that this particular value is represented by four features. When we create vectors, we can run mathematical computations over them and use them as inputs into ML models in the numerical form we require.

Mathematically, vectors are collections of coordinates that tell us where a given point is in space among many dimensions. For example, in two dimensions, we have a point $[2, 5]$, representing `bird_posts` and `bird_likes`.

In three dimensions, with three features including the bird id, we would have a vector

$$\begin{bmatrix} 12 & 2 & 5 \end{bmatrix} \tag{7}$$

which tells us where that user falls on all three axes.

But how do we represent "US" or "UK" in this space? Because modern models converge by performing operations on matrices [39], we need to

- **Candidate Generation** - First, we ingest data from the web app. This data goes into the initial piece, which hosts our first-pass model generating **candidate recommendations**. This is where collaborative filtering takes place, and we whittle our list of potential candidates down from millions to thousands or hundreds.
- **Ranking** - Finally, we need a way to order the filtered list of recommendations based on what we think the user will prefer the most, so the next stage is **ranking**, and then we serve them out in the timeline or the ML product interface we're working with.
- **Filtering** - Once we have a generated list of candidates, we want to continue to filter them, using business logic (i.e. we don't want to see NSFW content, or items that are not on sale, for example.). This is generally a heavily heuristic-based step.
- **Retrieval** - This is the piece where the web application usually hits a model endpoint to get the final list of items served to the user through the product UI.

Databases have become the fundamental tool in building backend infrastructure that performs data lookups. Embeddings have become similar building blocks in the creation of many modern search and recommendation product architectures. Embeddings are a type of **machine learning feature** — or model input data — that we use first as input into the feature engineering stage, and the first set of results that come from our candidate generation stage, that are then incorporated into downstream processing steps of ranking and retrieval to produce the final items the user sees.

2.4.2 Machine learning features

Now that we have a high-level conceptual view of how machine learning and recommender systems work, let's build towards a candidate generation model that will offer relevant flits.

Let's start by modeling a traditional machine learning problem and contrast it with our NLP problem. For example, let's say that one of our business problems is predicting whether a bird is likely to continue to stay on Flutter or to churn[13] — disengage and leave the platform.

When we predict churn, we have a given set of machine learning feature inputs for each user and a final binary output of 1 or 0 from the model, 1 if the bird is likely to churn, or 0 if the user is likely to stay on the platform.

We might have the following inputs:

- How many posts the bird has clicked through in the past month (we'll call this `bird_posts` in our input data)
- The geographical location of the bird from the browser headers (`bird_geo`)
- How many posts the bird has liked over the past month (`bird_likes`)

[13] An extremely common business problem to solve in almost every industry where either customer population or subscription based on revenues is important

closeness of users. Another common approach is using methods such **matrix factorization**, the process of representing users and items in a feature matrix made up of low-dimensional factor vectors, which in our case, are also known as embeddings, and learning those feature vectors through the process of minimizing a cost function. This process can be thought of as similar to Word2Vec [43], a deep learning model which we'll discuss in depth in this document. There are many different approaches to collaborative filtering, including matrix factorization and **factorization machines**.

- **Content filtering** - This approach uses metadata available about our items (for example in movies or music, the title, year released, genre, and so on) as initial or additional features input into models and work well when we don't have much information about user activity, although they are often used in combination with collaborative filtering approaches. Many embeddings architectures fall into this category since they help us model the textual features for our items.
- **Learn to Rank** - Learn to rank methods focus on ranking items in relation to each other based on a known set of preferred rankings and the error is the number of cases when pairs or lists of items are ranked incorrectly. Here, the problem is not presenting a single item, but a set of items and how they interplay. This step normally takes place after candidate generation, in a filtering step, because it's computationally expensive to rank extremely large lists.
- **Neural Recommendations** - The process of using neural networks to capture the same relationships that matrix factorization does without explicitly having to create a user/item matrix and based on the shape of the input data. This is where deep learning networks, and recently, large language models, come into play. Examples of deep learning architectures used for recommendation include Word2Vec and BERT, which we'll cover in this document, and convolutional and recurrent neural networks for sequential recommendation (such as is found in music playlists, for example). Deep learning allows us to better model content-based recommendations and give us representations of our items in an embedding space. [73]

Recommender systems have evolved their own unique architectures[12], and they usually include constructing a four-stage recommender system that's made up of several machine learning models, each of which perform a different machine learning task.

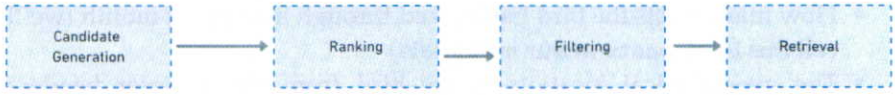


Figure 18: *Recommender systems as a machine learning problem*

[12]For a good survey on the similarities and difference between search and recommendations, read this great post on system design

improve through the process of gradient descent. We'll know because our loss should incrementally decrease in every training iteration.

We have finally trained our model. Now, we test the model's predictions on the 20 values that we've used as a hold-out set; i.e. the model has not seen these before and we can confidently assume that they won't influence the training data. We compare how many elements of the hold-out set the model was able to predict correctly to see what the model's accuracy was.

2.4.1 The Task of Recommendations

We just saw a simple example of machine learning as it relates to predicting continuous response variables. When our business question is, "What would be good content to show our users," we are facing the machine learning task for recommendation. Recommender systems are systems set up for **information retrieval**, a field closely related to NLP that's focused on finding relevant information in large collections of documents. The goal of information retrieval is to synthesize large collections of unstructured text documents. Within information retrieval, there are two complementary solutions in how we can offer users the correct content in our app: search, and recommendations.

Search is the problem of directed [17] information seeking, i.e. the user offers the system a specific query and would like a set of refined results. Search engines at this point are a well-established traditional solution in the space.

Recommendation is a problem where "man is the query." [58] Here, we don't know what the person is looking for exactly, but we would like to infer what they like, and recommend items based on their learned tastes and preferences.

The first industrial recommender systems were created to filter messages in email and newsgroups [22] at the Xerox Palo Alto Research Center based on a growing need to filter incoming information from the web. The most common recommender systems today are those at Netflix, YouTube, and other large-scale platforms that need a way to surface relevant content to users.

The goal of recommender systems is surface items that are relevant to the user. Within the framework of machine learning approaches for recommendation, the main machine learning task is to determine which items to show to a user in a given situation. [5]. There are several common ways to approach the recommendation problem.

- **Collaborative filtering** - The most common approach for creating recommendations is to formulate our data as a problem of finding missing user-item interactions in a given set of user-item interaction history. We start by collecting either explicit (ratings) data or **implicit** user interaction data like clicks, pageviews, or time spent on items, and compute. The simplest form of interactions are **neighborhood models**, where ratings are predicted initially by finding users similar to our given target user. We use similarity functions to compute the

differences between each point and the line, in other words to minimize the error, because it will mean that, at each point, our predicted y is as close to our actual y as we can get it, given the other points.

$$y = x_1\beta_1 + x_2\beta_2 + \varepsilon \tag{4}$$

The heart of machine learning is this training phase, which is the process of finding a combination of model instructions and data that accurately represent our real data, which, in supervised learning, we can validate by checking the correct "answers" from the test set.

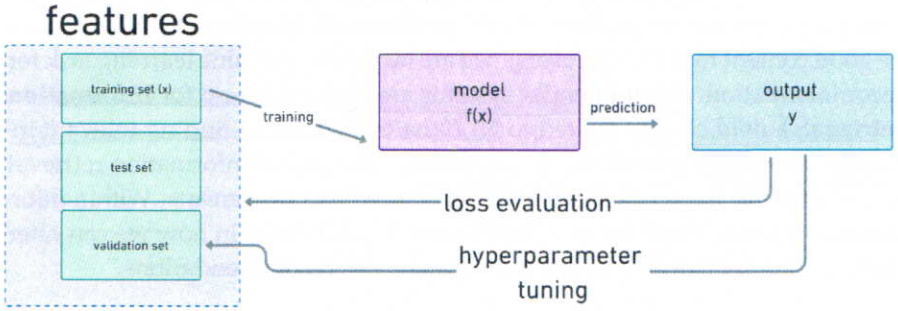


Figure 17: *The cycle of machine learning model development*

As the first round of training starts, we have our data. We **train** — or build — our model by initializing it with a set of inputs, X. These are from the training data. β_1 and β_2 are either initialized by setting to zero or initialized randomly (depending on the model, different approaches work best), and we calculate $\hat{y}$, our predicted value for the model. ϵ is derived from the data and the estimated coefficients once we get an output.

$$y = 2\beta_1 + 5\beta_2 + \varepsilon \tag{5}$$

How do we know our model is good? We initialize it with some set of values, weights, and we iterate on those weights, usually by minimizing a **cost function**. The cost function is a function that models the difference between our model's predicted value and the actual output for the training data. The first output may not be the most optimal, so we iterate over the model space many times, optimizing for the specific metric that will make the model as representative of reality as possible and minimize the difference between the actual and predicted values. So in our case, we compare $\hat{y}$ to y. The average squared difference between an observation's actual and predicted values is the cost, otherwise known as **MSE** - mean squared error.

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - (mx_i + b))^2 \tag{6}$$

We'd like to minimize this cost, and we do so with **gradient descent**. When we say that the model **learns**, we mean that we can learn what the correct inputs into a model are through an of iterative process where we feed the model data, evaluate the output, and to see if the predictions it generates

Table 1: *Tabular Input Data for Flutter Users*

bird_id	bird_posts	bird_likes
012	2	5
013	0	4
056	57	70
612	0	120

We'll need part of this data to train our model, part of it to test the accuracy of the model we've trained, and part to tune meta-aspects of our model. These are known as **hyperparameters**.

We take two parts of this data as holdout data that we don't feed into the model. The first part, the **test set**, we use to validate the final model on data it's never seen before. We use the second split, called the **validation set**, to check our hyperparameters during the model training phase. In the case of linear regression, there are no true hyperparameters, but we'll need to keep in mind that we will need to tune the model's metadata for more complicated models.

Let's assume we have 100 of these values. A usual accepted split is to use 80% of data for training and 20% for testing. The reasoning is we want our model to have access to as much data as possible so it learns a more accurate representation.

In general, our goal is to feed our input into the model, through a function that we pick, and get some predicted output, $f(X) \rightarrow y$.

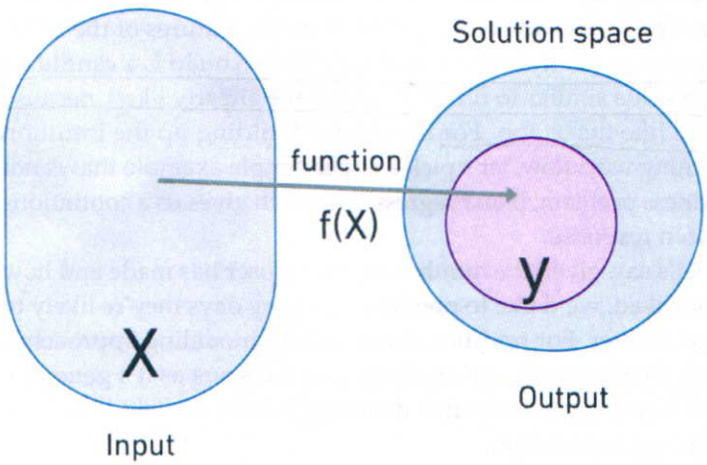


Figure 16: *How inputs map to outputs in ML functions [34]*

For our simple dataset, we can use the linear regression equation:

$$y = x_1\beta_1 + x_2\beta_2 + \varepsilon \tag{3}$$

This tells us that the output, y, can be predicted by two input variables, x_1 (bird posts) and x_2 (bird likes) with their given weights, β_1 and β_2, plus an error term ε, or the distance between each data point and the regression line generated by the equation. Our task is to find the smallest sum of squared

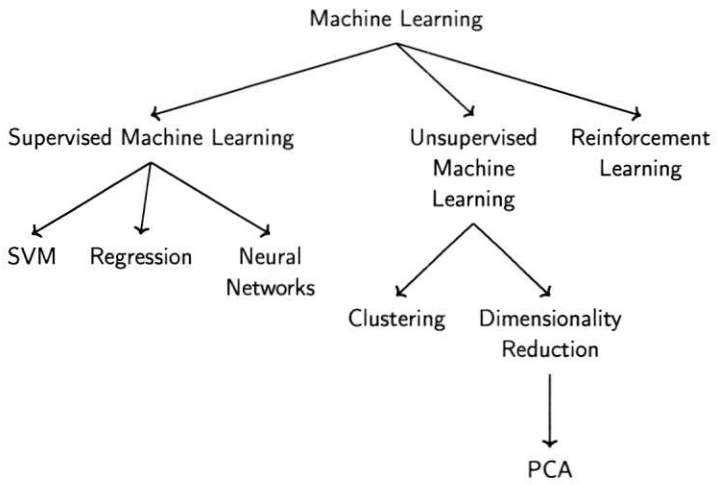


Figure 15: *Machine learning task solution space and model families*

2.4 Formulating a machine learning problem

As we saw in the last section, machine learning is a process that takes data as input to produce rules for how we should classify something or filter it or recommend it, depending on the task at hand. In any of these cases, for example, to generate a set of potential candidates, we need to construct a **model**.

A machine learning model is a set of instructions for generating a given output from data. The instructions are learned from the features of the input data itself. For Flutter, an example of a model we'd like to build is a **candidate generator** that picks flits similar to flits our birds have already liked, because we think users will like those, too. For the sake of building up the intuition for a machine learning workflow, let's pick a super-simple example that is not related to our business problem, linear regression, which gives us a continuous variable as output in response.

For example, let's say, given the number of posts a user has made and how many posts they've liked, we'd like to predict how many days they're likely to continue to stay on Flutter. For traditional **supervised** modeling approaches using tabular data, we start with our input data, or a **corpus** as it's generally known in machine learning problems that deal with text in the field known as **NLP** (natural language processing).

We're not doing NLP yet, though, so our input data may look something like this, where we have a UID (userid) and some attributes of that user, such as the number of times they've posted and number of posts they've liked. These are our machine learning **features**.

- **Model Serving** - Now that we have a model we like, we serve it to production, where it hits a web service, potentially cache, and our API where it then propagates to the front-end for the user to consume as part of our web app

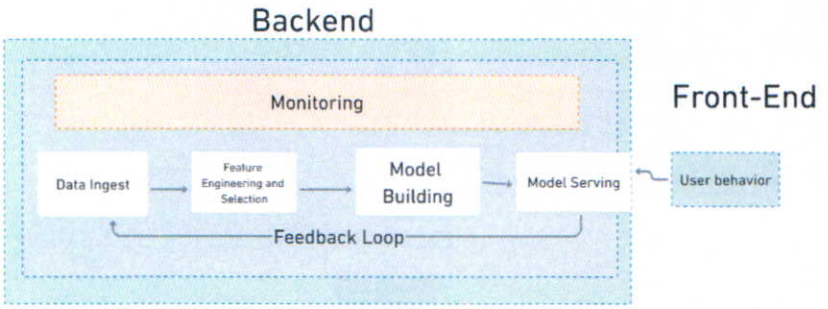


Figure 14: *CRUD app with ML*

Within machine learning, there are many approaches we can use to fit different tasks. Machine learning workflows that are most effective are formulated as solutions to both a specific business need and a machine learning **task**. Tasks can best be thought of as approaches to modeling within the categorized solution space. For example, learning a regression model is a specific case of a task. Others include clustering, machine translation, anomaly detection, similarity matching, or semantic search. The three highest-level types of ML tasks are **supervised**, where we have training data that can tell us whether the results the model predicted are correct according to some model of the world. The second is **unsupervised**, where there is not a single ground-truth answer. An example here is clustering of our customer base. A clustering model can detect patterns in your data but won't explicitly label what those patterns are. The third is **reinforcement learning** which is separate from these two categories and formulated as a game theory problem: we have an agent moving through an environment and we'd like to understand how to optimally move them through a given environment using explore-exploit techniques. We'll focus on supervised learning, with a look at unsupervised learning with PCA and Word2Vec.

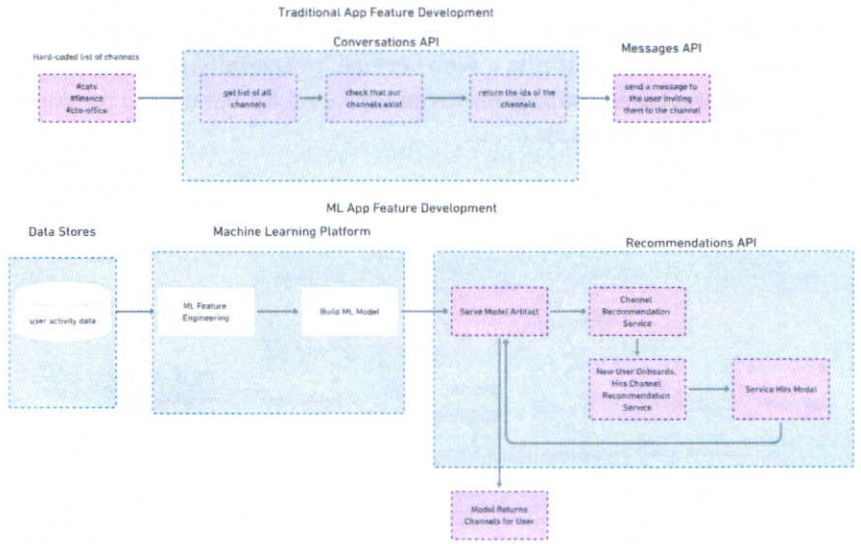


Figure 13: *Traditional versus ML architecture and infra*

2.3 Building a web app with machine learning

All machine learning systems can be examined through how they accomplish these four steps. When we build models, our key questions should be, "what kind of input do we have and how is it formatted", and "what do we get as a result." We'll be asking this for each of the approaches we look at. When we build a machine learning system, we start by processing data and finish by serving a learned model artifact.

The four components of a machine learning system are[11]:

- **Input data** - processing data from a database or streaming from a production application for use in modeling
- **Feature Engineering and Selection** - The process of examining the data and cleaning it to pick features. In this case, we mean features as attributes of any given element that we use as inputs into machine learning. Examples of features are: user name, geographic location, how many times they've clicked on a button for the past 5 days, and revenue. This piece always takes the longest in any given machine learning system, and is also known as finding **representations** [4] of the data that best fit the machine learning algorithm. This is where, in the new model architectures, we use embeddings as input.
- **Model Building** - We select the features that are important and train our model, iterating on different performance metrics over and over again until we have an acceptable model we can use. Embeddings are also the output of this step that we can use in other, downstream steps.

[11]There are infinitely many layers of horror in ML systems [37]. These are still the foundational components.

In short, the difference between programming and machine learning development is that we are not generating answers through business rules, but business rules through data. These rules are then re-incorporated into the application.

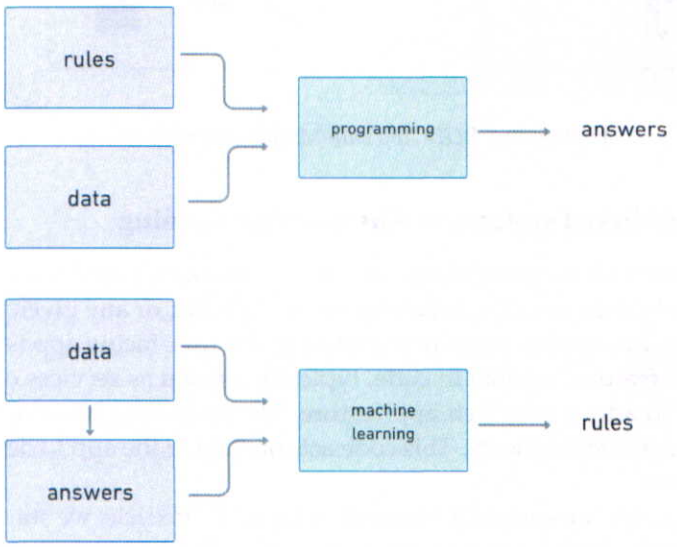


Figure 12: *Generating answers via machine learning. The top chart shows a classical programming approach with rules and data as inputs, while the bottom chart shows a machine learning approach with data and answers as inputs. [8]*

As an example, with Slack, for the channel recommendations product feature, we are not hard-coding a list of channels that need to be called from the organization's API. We are feeding in data about the organization's users (what other channels they've joined, how long they've been users, what channels the people they've interacted the most with Slack in), and building a model on that data that recommends a non-deterministic, personalized list of channels for each user that we then surface through the UI.

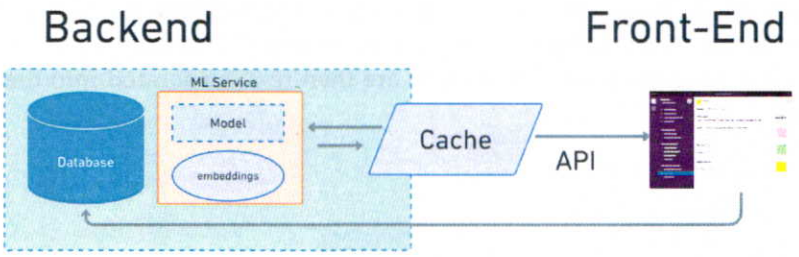


Figure 9: *CRUD App with Machine learning service*

2.2 Rules-based systems versus machine learning

To understand where embeddings fit into these systems, it first makes sense to understand where machine learning fits in at Flutter, or any given company, as a whole. In a typical consumer company, the user-facing app is made up of product features written in code, typically written as services or parts of services. To add a new web app feature, we write code based on a set of business logic requirements. This code acts on data in the app to develop our new feature.

In a typical data-centric software development lifecycle, we start with the business logic. For example, let's take the ability to post messages. We'd like users to be able to input text and emojis in their language of choice, have the messages sorted chronologically, and render correctly on web and mobile. These are the business requirements. We use the input data, in this case, user messages, and format them correctly and sort chronologically, at low latency, in the UI.

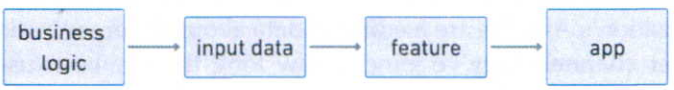


Figure 10: *A typical application development lifecycle*

Machine learning-based systems are typically also services in the backend of web applications. They are integrated into production workflows. But, they process data much differently. In these systems, we don't start with business logic. We start with input data that we use to build a model that will suggest the business logic for us. For more on the specifics of how to think about these data-centric engineering systems, see Kleppmann[35].

This requires thinking about application development slightly differently, and when we write an application that includes machine learning models as input, however, we're inverting the traditional app lifecycle. What we have instead, is data plus our desired outcome. The data is combined into a model, and it is this model which instead generates our business logic that builds features.

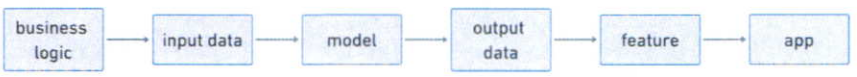


Figure 11: *ML Development lifecycle*

As an important note, **features** have many different definitions in machine learning and engineering. In this specific case, we mean collections of code that make up some front-end element, such as a button or a panel of recommendations. We'll refer to these as **product features**, in contrast with **machine learning features**, which are input data into machine learning models.

This application architecture is commonly known as **model-view-controller** pattern [20], or in common industry lingo, a **CRUD** app, named for the basic operations that its API allows to manage application state: create, read, update, and delete.

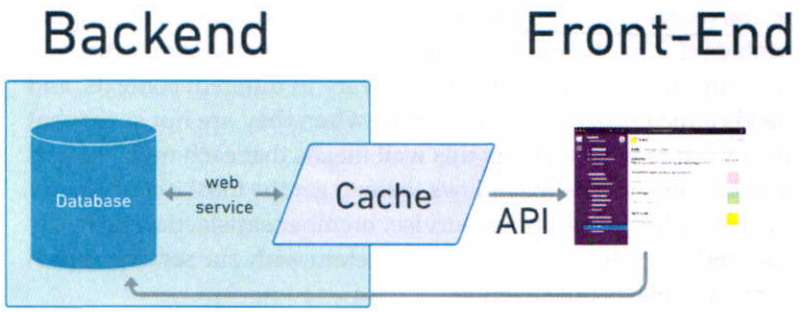


Figure 8: *Typical CRUD web app architecture*

When we think of structural components in the architectures of these applications, we might think first in terms of product features. In an application like Slack, for example, we have the ability to post and read messages, manage notifications, and add custom emojis. Each of these can be seen as an application feature. In order to create features, we have to combine common elements like databases, caches, and web services. All of this happens as the client talks to the API, which talks to the database to process data. At a more granular, program-specific level, we might think of foundational data structures like arrays or hash maps, and lower still, we might think about memory management and network topologies. These are all foundational elements of modern programming.

At the feature level, though, we see that it not only includes the typical CRUD operations, such as the ability to post and read Slack messages, but also elements that are more than operations that alter database state. Some features such as personalized channel suggestions, returning relevant results through search queries, and predicting Slack connection invites necessitates the use of machine learning.

items to the user, and information retrieval platforms like search engines, which must provide relevant answers to users upon keyword queries. There is a new category of hybrid applications involving question-and-answering in semantic search contexts that is arising as a result of work around the GPT series of models, but for the sake of simplicity, and because that landscape changes every week, we'll stick to understanding the fundamental underlying concepts.

In subscription-based platforms[9], there is clear business objective that's tied directly to the bottom line, as outlined in this 2015 paper [64] about Netflix's recsys:

> The main task of our recommender system at Netflix is to help our members discover content that they will watch and enjoy to maximize their long-term satisfaction. This is a challenging problem for many reasons, including that every person is unique, has a multitude of interests that can vary in different contexts, and needs a recommender system most when they are not sure what they want to watch. Doing this well means that each member gets a unique experience that allows them to get the most out of Netflix. As a monthly subscription service, member satisfaction is tightly coupled to a person's likelihood to retain with our service, which directly impacts our revenue.

Knowing this business context, and given that personalized content is more relevant and generally gets higher rates of engagement [30] than non-personalized forms of recommendation on online platforms,[10] how and why might we use embeddings in machine learning workflows in Flutter to show users flits that are interesting to them personally? We need to first understand how web apps work and where embeddings fit into them.

2.1 Building a web app

Most of the apps we use today — Spotify, Gmail, Reddit, Slack, and Flutter — are all designed based on the same foundational software engineering patterns. They are all apps available on web and mobile clients. They all have a front-end where the user interacts with the various **product features** of the applications, an API that connects the front-end to back-end elements, and a database that processes data and remembers state.

[9] In ad-based services, the line between retention and revenue is a bit murkier, and we have often what's known as a multi-stakeholder problem, where the actual optimized function is a balance between meeting the needs of the user and meeting the needs of the advertiser [75]. In real life, this can often result in a process of enshittification [15] of the platform that leads to extremely suboptimal end-user experiences. So, when we create Flutter, we have to be very careful to balance these concerns, and we'll also assume for the sake of simplification that Flutter is a Good service that loves us as users and wants us to be happy.

[10] For more, see this case study on personalized recommendations as well as the intro section of this paper which covers many personalization use-cases.

that is curated by the user. But we also would like to offer personalized, recommended flits so that the user finds interesting content on our platform that they might have not known about before.

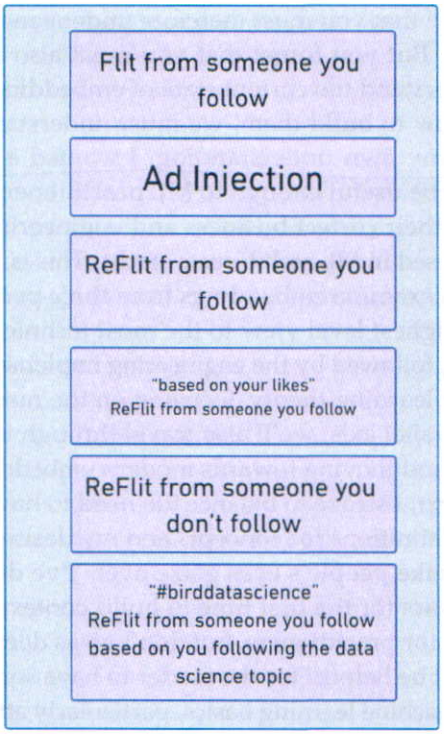


Figure 7: *Flutter's content timeline in a social feed with a blend of organic followed content, advertising, and recommendations.*

How do we solve the problem of what to show in the timeline here so that our users find the content relevant and interesting, and balance the needs of our advertisers and business partners?

In many cases, we can approach engineering solutions without involving machine learning. In fact, we should definitely start without it [76] because machine learning adds a tremendous amount of complexity to our working application [57]. In the case of the Flutter home feed, though, machine learning forms a business-critical function part of the product offering. From the business product perspective, the objective is to offer Flutter's users content that is relevant[8], interesting, and novel so they continue to use the platform. If we do not build discovery and personalization into our content-centric product, Flutter users will not be able to discover more content to consume and will disengage from the platform.

This is the case for many content-based businesses, all of which have feed-like surface areas for recommendations, including Netflix, Pinterest, Spotify, and Reddit. It also covers e-commerce platforms, which must surface relevant

[8]The specific definition of a relevant item in the recommendations space varies and is under intense academic and industry debate, but generally it means an item that is of interest to the user

are already NLP experts, and surface-level marketing spam blurbs for people looking to buy embeddings-based tech, and that neither of these overlap in what they cover.

In Systems Thinking, Donella Meadows writes, "You think that because you understand 'one' that you must therefore understand 'two' because one and one make two. But you forget that you must also understand 'and.'" [45] In order to understand the current state of embedding architectures and be able to decide how to build them, we must understand how they came to be. In building my own understanding, I wanted a resource that was technical enough to be useful enough to ML practitioners, but one that also put embeddings in their correct business and engineering contexts as they become more often used in ML architecture stacks. This is, hopefully, that text.

In this text, we'll examine embeddings from three perspectives, working our way from the highest level view to the most technical. We'll start with the business context, followed by the engineering implementation, and finally look at the machine learning theory, focusing on the nuts and bolts of how they work. On a parallel axis, we'll also travel through time, surveying the earliest approaches and moving towards modern embedding approaches.

In writing this text, I strove to balance the need to have precise technical and mathematical definitions for concepts and my desire to stay away from explanations that make people's eyes glaze over. I've defined all technical jargon when it appears for the first time to build context. I include code as a frame of reference for practitioners, but don't go as deep as a code tutorial would[7]. So, it would be helpful for the reader to have some familiarity with programming and machine learning basics, particularly after the sections that discuss business context. But, ultimately the goal is to educate anyone who is willing to sit through this, regardless of level of technical understanding.

It's worth also mentioning what this text does not try to be: it does not try to explain the latest advancements in GPT and generative models, it does not try to explain transformers in their entirety, and it does not try to cover all of the exploding field of vector databases and semantic search. I've tried my best to keep it simple and focus on really understanding the core concept of embeddings.

2 Recommendation as a business problem

Let's step back and look at the larger context with a concrete example before diving into implementation details. Let's build a social media network, **Flutter**, the premier social network for all things with wings. Flutter is a web and mobile app where birds can post short snippets of text, videos, images, and sounds, to let other birds, insects and bats in the area know what's up. Its business model is based on targeted advertising, and its app architecture includes a "home" feed based on birds that you follow, made up of small pieces of multimedia content called **"flits"**, which can be either text, videos, or photos. The home feed itself is by default in reverse chronological order

[7] In other words, I wanted to straddle the "explanation" and "reference" quadrants of the Diátaxis framework

sional matrix containing multiple levels of embeddings, and that's because in BERT's embedding representation, we have 13 different layers. One embedding layer is computed for each layer of the neural network. Each level represents a different view of our given **token** — or simply a sequence of characters. We can get the final embedding by pooling several layers, details we'll get into as we work our way up to understanding embeddings generated using BERT.

When we create an embedding for a word, sentence, or image that represents the artifact in the multidimensional space, we can do any number of things with this embedding. For example, for tasks that focus on content understanding in machine learning, we are often interested in comparing two given items to see how similar they are. Projecting text as a vector allows us to do so with mathematical rigor and compare words in a shared embedding space.

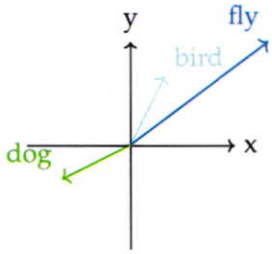


Figure 5: *Projecting words into a shared embedding space*

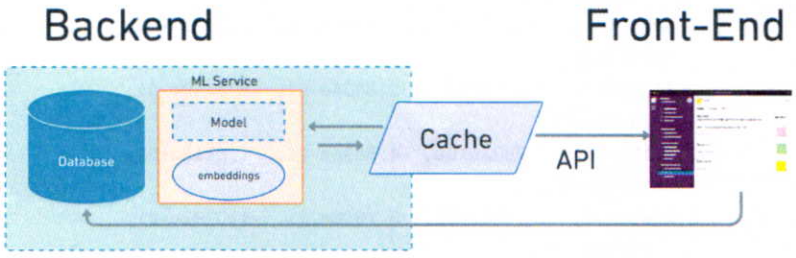


Figure 6: *Embeddings in the context of an application.*

Engineering systems based on embeddings can be computationally expensive to build and maintain [61]. The need to create, store, and manage embeddings has also recently resulted in the explosion of an entire ecosystem of related products. For example, the recent rise in the development of vector databases to facilitate production-ready use of nearest neighbors semantic queries in machine learning systems[5], and the rise of embeddings as a service[6].

As such, it's important to understand their context both as end-consumers, product management teams, and as developers who work with them. But in my deep-dive into the embeddings reference material, I found that there are two types of resources: very deeply technical academic papers, for people who

[5]For a survey of the vector database space today, refer to this article

[6]Embeddings now are a key differentiator in pricing between on-demand ML services

```
import torch
from transformers import BertTokenizer, BertModel

# Load pre-trained model tokenizer (vocabulary)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

text = """Hold fast to dreams, for if dreams die, life is a broken-winged bird that cannot fly."""

# Tokenize the sentence with the BERT tokenizer.
tokenized_text = tokenizer.tokenize(text)

# Print out the tokens.
print (tokenized_text)

['[CLS]', 'hold', 'fast', 'to', 'dreams', ',', 'for', 'if', 'dreams', 'die', ',', 'life', 'is', 'a', 'broken', '-', 'winged', 'bird', 'that', 'cannot', 'fly', '.', '[SEP]']

# BERT code truncated to show the final output, an embedding

[tensor([-3.0241e-01, -1.5066e+00, -9.6222e-01,  1.7986e-01, -2.7384e+00,
                 -1.6749e-01,  7.4106e-01,  1.9655e+00,  4.9202e-01,
                 -2.0871e+00,
                 -5.8469e-01,  1.5016e+00,  8.2666e-01,  8.7033e-01,
                  8.5101e-01,
                  5.5919e-01, -1.4336e+00,  2.4679e+00,  1.3920e+00,
                 -3.9291e-01,
                 -1.2054e+00,  1.4637e+00,  1.9681e+00,  3.6572e-01,
                  3.1503e+00,
                 -4.4693e-01, -1.1637e+00,  2.8804e-01, -8.3749e-01,
                  1.5026e+00,
                 -2.1318e+00,  1.9633e+00, -4.5096e-01, -1.8215e+00,
                  3.2744e+00,
                  5.2591e-01,  1.0686e+00,  3.7893e-01, -1.0792e-01,
                  5.1342e-01,
                 -1.0443e+00,  1.7513e+00,  1.3895e-01, -6.6757e-01,
                 -4.8434e-01,
                 -2.1621e+00, -1.5593e+01,  1.5249e+00,  1.6911e+00,
                 -1.2916e+00,
                  1.2339e+00, -3.6064e-01, -9.6036e-01,  1.3226e+00,
                  1.6427e+00,
                  1.4588e+00, -1.8806e+00,  6.3620e-01,  1.1713e+00,
                  1.1050e+00, ...
                 2.1277e+00])
```

Figure 4: *Analyzing Embeddings with BERT. See full notebook source*

We can see that this embedding is a PyTorch tensor object, a multidimen-

What do embeddings actually look like? Here is one single embedding, also called a **vector**, in three **dimensions**. We can think of this as a representation of a single element in our dataset. For example, this hypothetical embedding represents a single word "fly", in three dimensions. Generally, we represent individual embeddings as row vectors.

$$\begin{bmatrix} 1 & 4 & 9 \end{bmatrix} \tag{1}$$

And here is a **tensor**, also known as a **matrix**[3], which is a multidimensional combination of vector representations of multiple elements. For example, this could be the representation of "fly", and "bird."

$$\begin{bmatrix} 1 & 4 & 9 \\ 4 & 5 & 6 \end{bmatrix} \tag{2}$$

These embeddings are the output of the process of **learning** embeddings, which we do by passing raw input data into a machine learning model. We transform that multidimensional input data by compressing it, through the algorithms we discuss in this paper, into a lower-dimensional space. The result is a set of vectors in an **embedding space.**

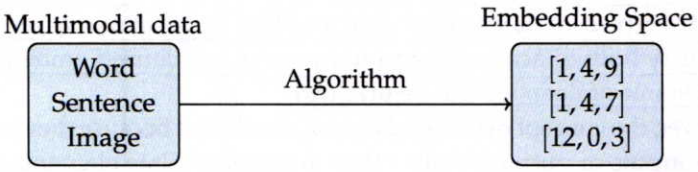


Figure 3: *The process of embedding.*

We often talk about item embeddings being in X dimensions, ranging anywhere from 100 to 1000, with diminishing returns in usefulness somewhere beyond 200-300 in the context of using them for machine learning problems[4]. This means that each item (image, song, word, etc) is represented by a vector of length X, where each value is a coordinate in an X-dimensional space.

We just made up an embedding for "bird", but let's take a look at what a real one for the word "hold" would look like in the quote, as generated by the BERT deep learning model,

> "Hold fast to dreams, for if dreams die, life is a broken-winged bird that cannot fly." — Langston Hughes

We've highlighted this quote because we'll be working with this sentence as our input example throughout this text.

[3]The difference between a matrix and a tensor is that it's a matrix if you're doing linear algebra and a tensor if you're an AI researcher.

[4]Embedding size is tunable as a hyperparameter but so far there have only been a few papers on optimal embedding size, with most of the size of embeddings set through magic and guesswork

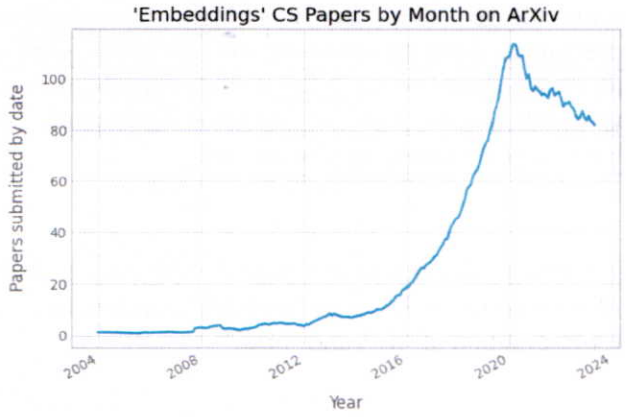


Figure 2: *Embeddings papers in arXiv by month. It's interesting to note the decline in frequency of embeddings-specific papers, possibly in tandem with the rise of deep learning architectures like GPT source*

Building and expanding on the concepts in Word2Vec, the Transformer [66] architecture, with its self-attention mechanism, a much more specialized case of calculating context around a given word, has become the de-facto way to learn representations of growing multimodal vocabularies, and its rise in popularity both in academia and in industry has caused embeddings to become a staple of deep learning workflows.

However, the concept of embeddings can be elusive because they're neither data flow inputs or output results - they are intermediate elements that live within machine learning services to refine models. So it's helpful to define them explicitly from the beginning.

As a general definition, embeddings are data that has been transformed into n-dimensional matrices for use in deep learning computations. The process of embedding (as a verb):

- *Transforms* multimodal input into representations that are easier to perform intensive computation on, in the form of **vectors**, tensors, or graphs [51]. For the purpose of machine learning, we can think of vectors as a list (or array) of numbers.
- *Compresses* input information for use in a machine learning **task** — the type of methods available to us in machine learning to solve specific problems — such as summarizing a document or identifying tags or labels for social media posts or performing **semantic search** on a large text corpus. The process of compression changes variable feature dimensions into fixed inputs, allowing them to be passed efficiently into downstream components of machine learning systems.
- *Creates an embedding space* that is specific to the data the embeddings were trained on but that, in the case of deep learning representations, can also generalize to other tasks and domains through **transfer learning** — the ability to switch contexts — which is one of the reasons embeddings have exploded in popularity across machine learning applications

1 Introduction

Implementing deep learning models has become an increasingly important machine learning strategy[1] for companies looking to build data-driven products. In order to build and power deep learning models, companies collect and feed hundreds of millions of terabytes of multimodal[2] data into deep learning models. As a result, **embeddings** — deep learning models' internal representations of their input data — are quickly becoming a critical component of building machine learning systems.

For example, they make up a significant part of Spotify's item recommender systems [27], YouTube video recommendations of what to watch [11], and Pinterest's visual search [31]. Even if they are not explicitly presented to the user through recommendation system UIs, embeddings are also used internally at places like Netflix to make content decisions around which shows to develop based on user preference popularity.

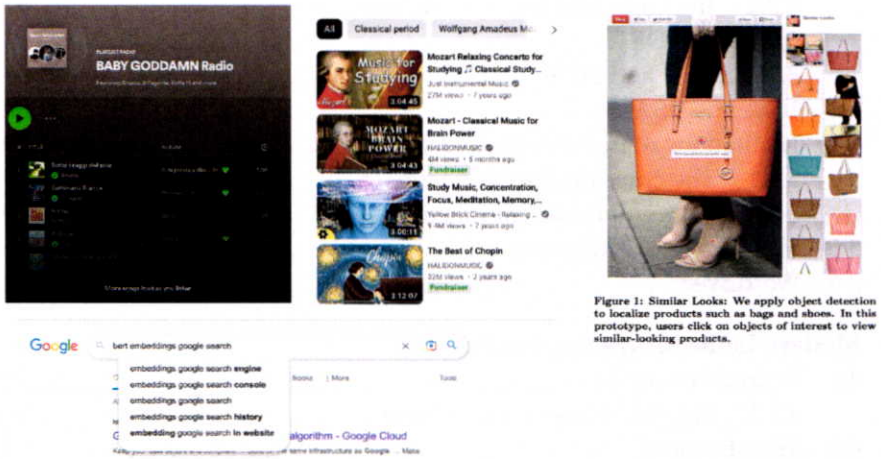


Figure 1: Similar Looks: We apply object detection to localize products such as bags and shoes. In this prototype, users click on objects of interest to view similar-looking products.

Figure 1: *Left to right: Products that use embeddings used to generate recommended items: Spotify Radio, YouTube Video recommendations, visual recommendations at Pinterest, BERT Embeddings in suggested Google search results*

The usage of embeddings to generate compressed, context-specific representations of content exploded in popularity after the publication of Google's Word2Vec paper [47].

[1] Check out the machine learning industrial view Matt Turck puts together every year, which has exploded in size.

[2] Multimodal means a variety of data usually including text, video, audio, and more recently as shown in Meta's ImageBind, depth, thermal, and IMU.

Contents

Abstract

Over the past decade, embeddings — numerical representations of machine learning features used as input to deep learning models — have become a foundational data structure in industrial machine learning systems. TF-IDF, PCA, and one-hot encoding have always been key tools in machine learning systems as ways to compress and make sense of large amounts of textual data. However, traditional approaches were limited in the amount of context they could reason about with increasing amounts of data. As the volume, velocity, and variety of data captured by modern applications has exploded, creating approaches specifically tailored to scale has become increasingly important.

Google's Word2Vec paper made an important step in moving from simple statistical representations to semantic meaning of words. The subsequent rise of the Transformer architecture and transfer learning, as well as the latest surge in generative methods has enabled the growth of embeddings as a foundational machine learning data structure. This survey paper aims to provide a deep dive into what embeddings are, their history, and usage patterns in industry.

Colophon

This paper is typeset with LaTeX. The cover art is Kandinsky's "Circles in a Circle" , 1923. ChatGPT was used to generate some of the figures.

Code, LaTeX, and Website

The latest version of the paper and code examples are available here. The website for this project is here.

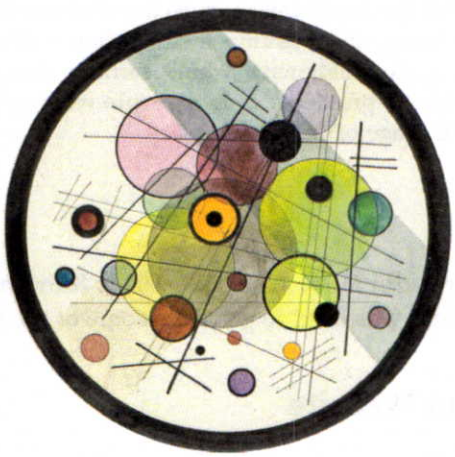
About the Author

Vicki Boykis is a machine learning engineer. Her website is vickiboykis.com and her semantic search side project is viberary.pizza.

Acknowledgements

I'm grateful to everyone who has graciously offered technical feedback but especially to Nicola Barbieri, Peter Baumgartner, Luca Belli, James Kirk, and Ravi Mody. All remaining errors, typos, and bad jokes are mine. Thank you to Dan for your patience, encouragement, for parenting while I was in the latent space, and for once casually asking, "How do you generate these 'embeddings', anyway?"

License



What are embeddings

Vicki Boykis

Embeddings Notes

1 → Introduction

embeddings: deep learning models' internal representation of their input data. Started with Word2Vec [important paper for me to read].

Building on the concepts in Word2Vec, the transformer architecture, with its self attention mechanism, has become the default way to learn representations of growing multimodal vocabularies → reconciles understanding as embeddings

embeddings are data that has been transformed into n-dimensional matrices for use in deep learning computations. [these are intermediate elements]

Process of embedding

1) Transforms multimodal input into representations that are easier to perform intensive computations on, in the form of vectors, tensors, or graphs.

2) Compresses input information for use in a machine learning task
   → the types of methods available to us in ML to solve specific problems.

   The process of compression changes variable dimensions into fixed inputs, allowing them to pared efficiently with downstream components of ML systems.

3) Creates an embedding space that is specific to the data the embeddings were trained on but that, in the case of deep learning representations, can also generalize to other tasks and domains through transfer learning — the ability to switch contexts.
   → like the equivalent of a journey transfer concretely?

Embeddings Visually

basically a vector [row vector] for an individual embedding

row vector [1 4 5] can be one word.

multidimensional collection of vector representations of multiple elements

$\begin{bmatrix} 1 & 4 & 9 \\ 4 & 5 & 6 \end{bmatrix}$ — tensor/matrix representation of fly/bird

pass raw input [text/image] into an ML model to get these.

Word
Sentence
Image

→ Algo → [1, 4, 9] [1, 4, 7] [12, 0, 3]

Multimodal Data — Embedding Space

We often talk about these embeddings being in X dimensions, ranging anywhere from 100 to 1000. Small dimensions returns in usefulness somewhere beyond 200-300 in the context of ML. (2)

Bert

can tokenize and return back tensor objects that represent the embeddings for a part of the text

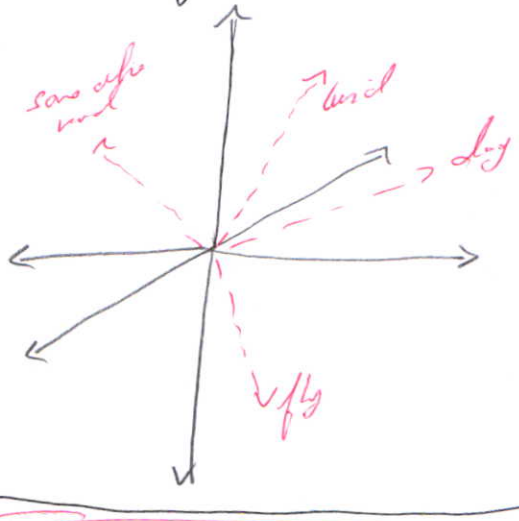
(1) tokenize the text
(2) return tensorized form
(3) hit embedding algo

* embedding is a python tensor object
n dimensional matrix containing multiple levels of embeddings → BERT embedding representation has 13 different layers.

one embedding layer is copied for each layer of the neural network

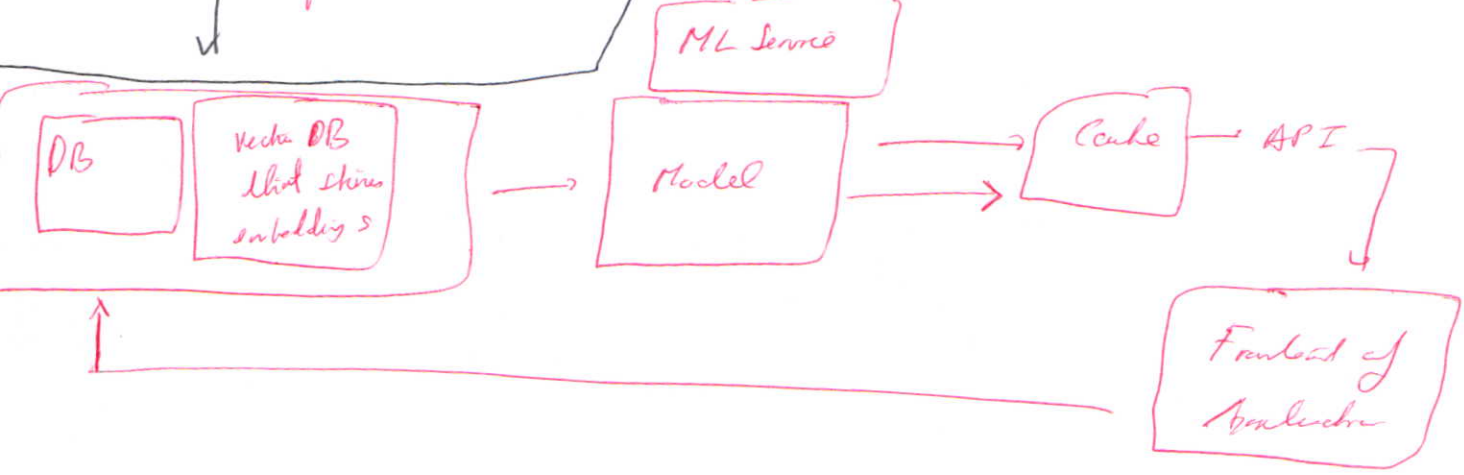
more clarity will be achieved once we fully embed BERT

When we create an embedding for a word, sentence, or image that represent the artifact in multidimensional space, we can graph text as a vector → graphing vectors in this form allows us to do mathematical comparisons repeatedly.



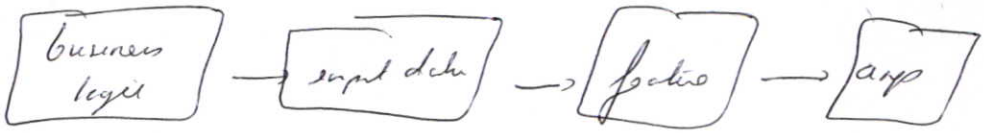
Building Engineering Systems with Embedding

(1) Recommendation as the first business problem
Rough Org Architecture Plan

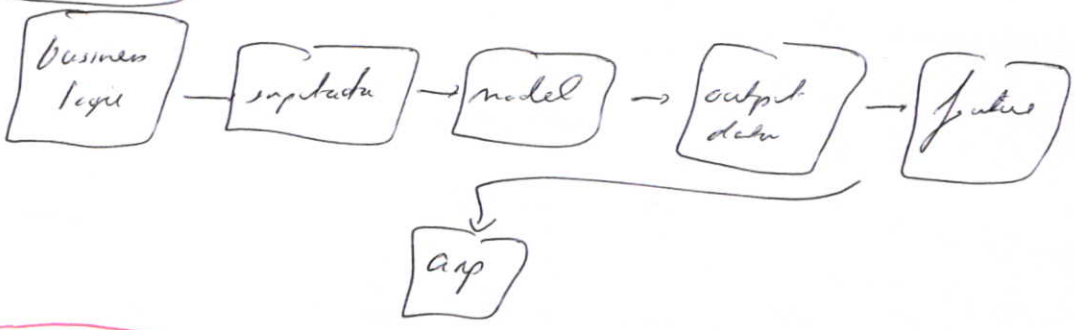


Rules Based Systems versus machine learning

typical app



ML App



for ML systems we start with input data that we use to build a model that will recreate the business logic for us

Klepmann for system design might be a good resource

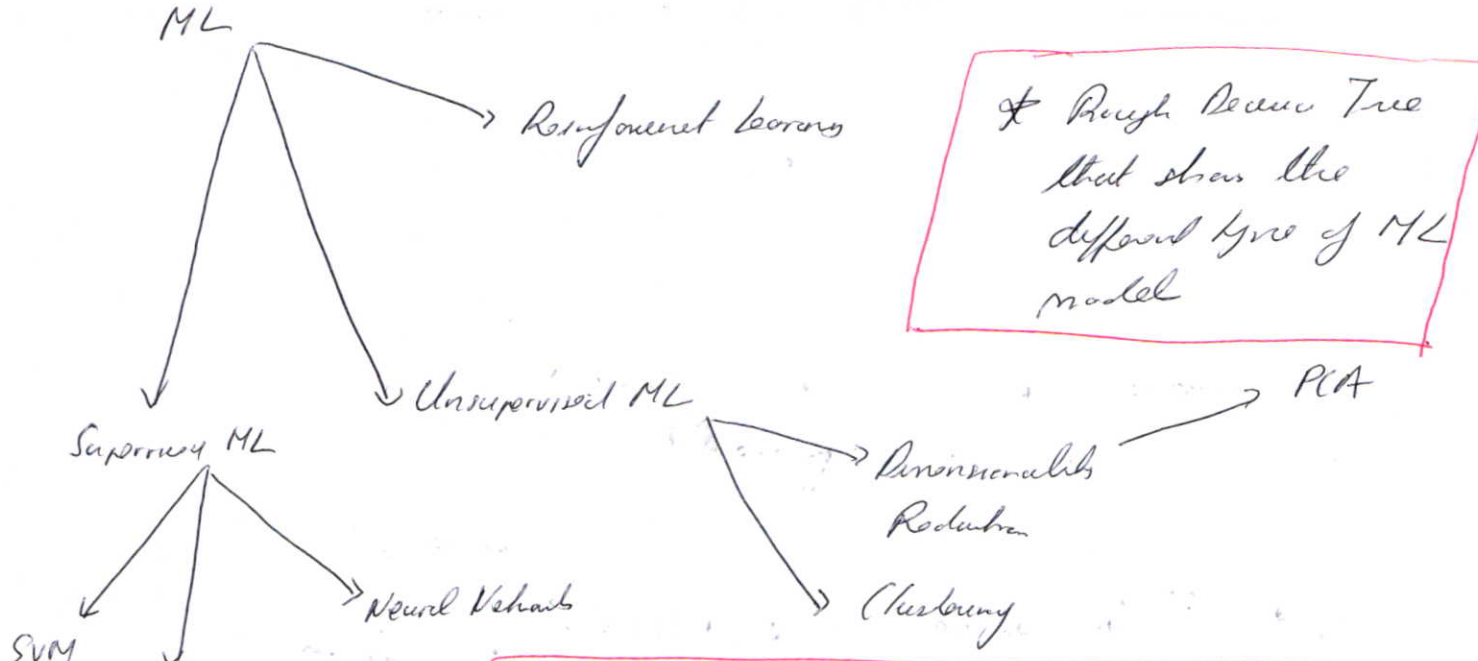
What we have in ML is data plus our desired outcome. We use algorithms to essentially generate the rules/business logic that we use to build features. → rules get reincorporated into the application

programming generates answers ~~to the~~ through business rules instead

2.3 Building a web app with machine learning

4 components

1. Input data: processing data from a database or streaming from a production application for use in modelling
2. Feature Engineering and Selection: The process of examining the data and cleaning it to pick features. In this case, we mean features as attributes of any given element that we use as input into ML. also known as finding representation of the data that best fit the ML algorithm. [this is where we use embeddings as an input
3. Model Building - We select features that are important and train our model
4. Model Serving - serve to production [app, etc]



* Rough Decision Tree that shows the different type of ML model

Formulating a ML problem 24

Ok, they are using the twitter clone example. Since ML is a process that takes data as an input to produce rules for how we should classify something or filter it or recommend it, depending on the task.

The model does something. In this case say we want the model that picks posts that are to be recommended to the user.

↓

Say we build a super simple model like a linear regression that predicts posts based on 2 features

↓

we can train a basic data, say our model has two features

posts
posts they liked

↓

we need part of the data to train our model, part to test accuracy and part to fine tune meta parameters of the model

↳ hyperparameters

train → test → validation

trains [we train an model by evaluating it with a set of inputs]

↓ can be done by minimizing a cost function

2.4.1 The Task of Recommendation

system set up for information retrieval (IR)

Search / Recommendation are two sides of the same coin

goal is to surface items that are relevant to the user

collaborative filtering

→ formulate our data as a problem of finding missing user-item interactions in a given set of user-item interaction history.

→ we start by collecting explicit data like ratings as explicit data from users

→ similarity from one neighborhood models

→ content filtering

Candidate generation → ① Input data from a web app. This data goes into the critical piece, which looks over first-pass model generating candidate recommendations [collaborative filtering step]

Ranking → ② ranks the filtered list of recommendations based on what we think the user will prefer the most

Filtering → ③ Hierarchical + Rule Logic Based filter

Retrieval → ④ final list Retrieved.

2.5 Numerical Feature Vectors

→ we represent features as numerical vectors. We can think of each row in our tabular feature data as a vector.

each row of tabular data can be a vector

[012, 2, 5, US]

how do we represent US as a vector?

↳ this is where Word2Vec comes in.

1) Encoding → we need to represent our non-numerical, multimodal data as numbers so we can create models out of them.

2) these encodings as vectors

3) Lookup matrices / hash tables [also known as attention] to help us map between the word and the numbers