


# GRAPH ML

→ Graph exists in non-euclidean space

Euclidean space deals with flat planes and non-curved spaces. Rules like parallel lines don't meet or sum of angles in a triangle is  $180^\circ$  are true in euclidean space.

Non-euclidean geometry is any kind of geometry that deals with curved surface or shape and rules of euclidean space do not apply

→ Graph Nodes do not have order e.g.  does not mean A comes before

→ Types of Graph Problems in ML / Applications of Graphs

→ Node classification

→ Binary classification:  
whether link exists b/w 2 nodes  
Training set:  
→ +ve: link exists b/w nodes  
→ -ve: no edge b/w nodes

→ Link Prediction (Recommendation)

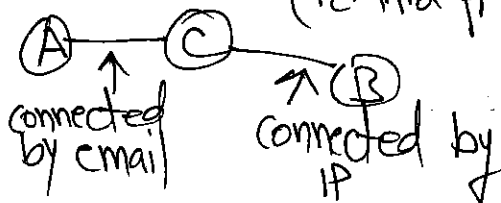
→ Graph classification (e.g. determining molecule types)

→ Community Detection

→ Most ML algos assume data is i.i.d hence tabular data. However, from tabular data, we can link examples/instances using some feature values and convert it into graph data. (i.e. i.i.d property is broken)

Example:

	IP	email
A		x
B	y	
C	y	x



Graph Data is not i.i.d.

→ Graph Schema Design is a key step for applied graph machine learning i.e. types of nodes (homogeneous or heterogeneous) node properties, how to link nodes (edges) if the data is tabular, kinds of edges etc.

↓  
same kind of nodes      diff kind of nodes

→ Real World Graphs are huge and dynamic in nature  
e.g. social n/w      ↑  
change frequently

→ Disadvantages / Cons of GNNs (including GraphSage)

i) GNNs are hard to parallelize: There are sequential steps during message-passing (one hop at a time).

ii) Over smoothing: The more  $n$ -hop neighbors the model is aggregating information from, the more smooth each node's final embedding will be. If you don't pay attention and aggregate all the info from a huge graph, all the node embeddings will look very similar.

## NODE EMBEDDINGS

## (GRAPH REPRESENTATION LEARNING)

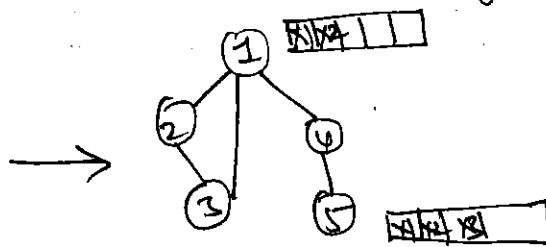
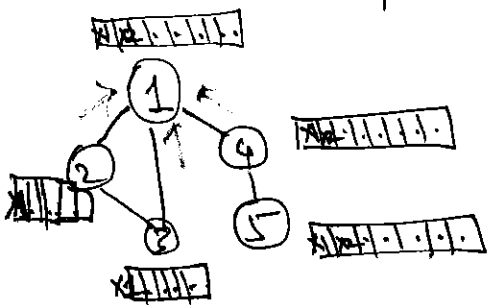
1. DEEPWALK
- Generate random walks starting from each node (seq. of nodes which capture the local structure) → like sentences
  - Use Word2Vec to learn representation of nodes using Skip-gram i.e. learn a neural network that predicts the prob. of word given its context (i.e. the words that appear before and after it)

2. Node2Vec
- Same as Deepwalk except how the random walk is generated
  - Uses 2 params that determines the combination of DFS and BFS exploration strategy

## LIMITATIONS OF ABOVE NODE EMBEDDINGS

1. Node type or Link/Edge type differentiation is not supported
2. Node features cannot be added to the algorithm

### 3. GCN (Graph Convolution N/w) & Message-Passing Layer



→ 1 is connected to 2, 3, 4  
 → 1<sup>st</sup> message passing happens between one-hop neighbors of 1 (i.e. except 5)

→ 5 is connected to 4 only, so after 1 hop aggregation or 1 layer of GCN, it knows about node 4 but nothing about node 1, 2 & 3. However, after 2<sup>nd</sup> round of message passing, node 5 will know about 2<sup>nd</sup> hop neighbors eg. 1, 2, 3

1 layer of GCN = one-hop neighbor aggregation of every node



→ K-Hop = K-layer GCN

→ Iteratively, we learn abt 1<sup>st</sup> hop neighbors, then 2<sup>nd</sup> hop neighbors and so on...

→ Local Feature Aggregation  $\approx$  Learnable CNN kernels

#### DRAWBACK OF GCNs:

- Require full graph for training (not suitable for dynamic graphs)
- Do not naturally generalize to unseen nodes

# GRAPHSAGE

↓ Aggregate  
Sample

→ Scalable way of Graph Machine Learning

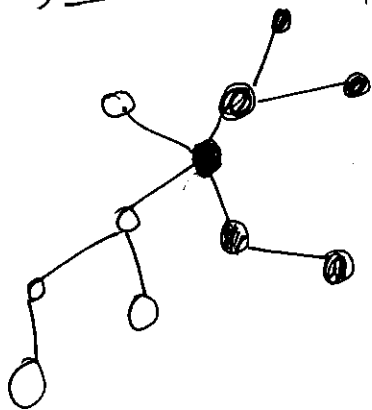
Adv:

→ trainable in minibatches using SGD  
ie. a subgraph can be used for training

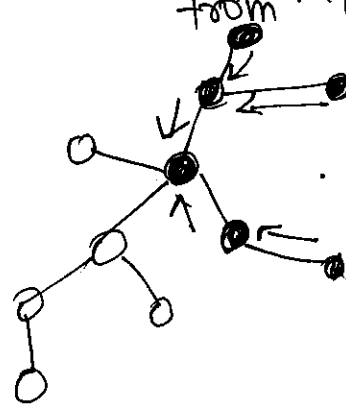
→ generates node embeddings on unseen nodes  
(for inference)

→ 3 Step process:

a) SAMPLE from k-hop neighborhood

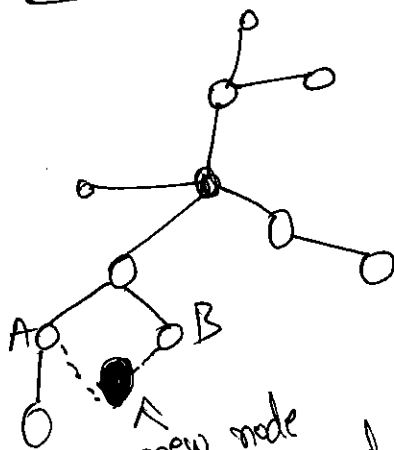


b) Aggregate feature information from neighbors

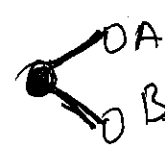


— Aggregator 1  
— Aggregator 2

c) Predict on unseen nodes

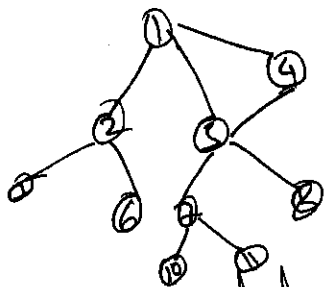


new node comes in and attaches to the other nodes in graph

=  = Aggregate Features from A & B to get embeddings on the unseen node

→ TRAINING USING A MINI-BATCH

i) Take a sub-graph from the big graph



ii) Build a computation graph for a sample of ~~nodes~~ <sup>connected</sup> sub-graph



mini-batch of computation graph  
that can be trained on a GPU

→ Similar to GCN,  $K$  hop =  $K$  layer of GNN  
Every hop = 1 layer of GNN for message passing  
Every node has an embedding at every layer based on msg passing

# DEPLOYMENT OF GRAPH SAGE (GNN)

→ 2 types of databases are needed  $\begin{cases} \text{Graph. e.g. Neptune, Neo4j} \\ \text{Relational e.g. Postgres} \end{cases}$

Graph DB :- stores relationship info. b/w nodes (e.g. adjacency matrix)  
- Has ability to insert and read subgraphs for training and inference

Relational DB :- stores node features to be used in GraphSage model

→ Model is built using DGL (Deep Graph library) which can use Tensorflow or PyTorch backend

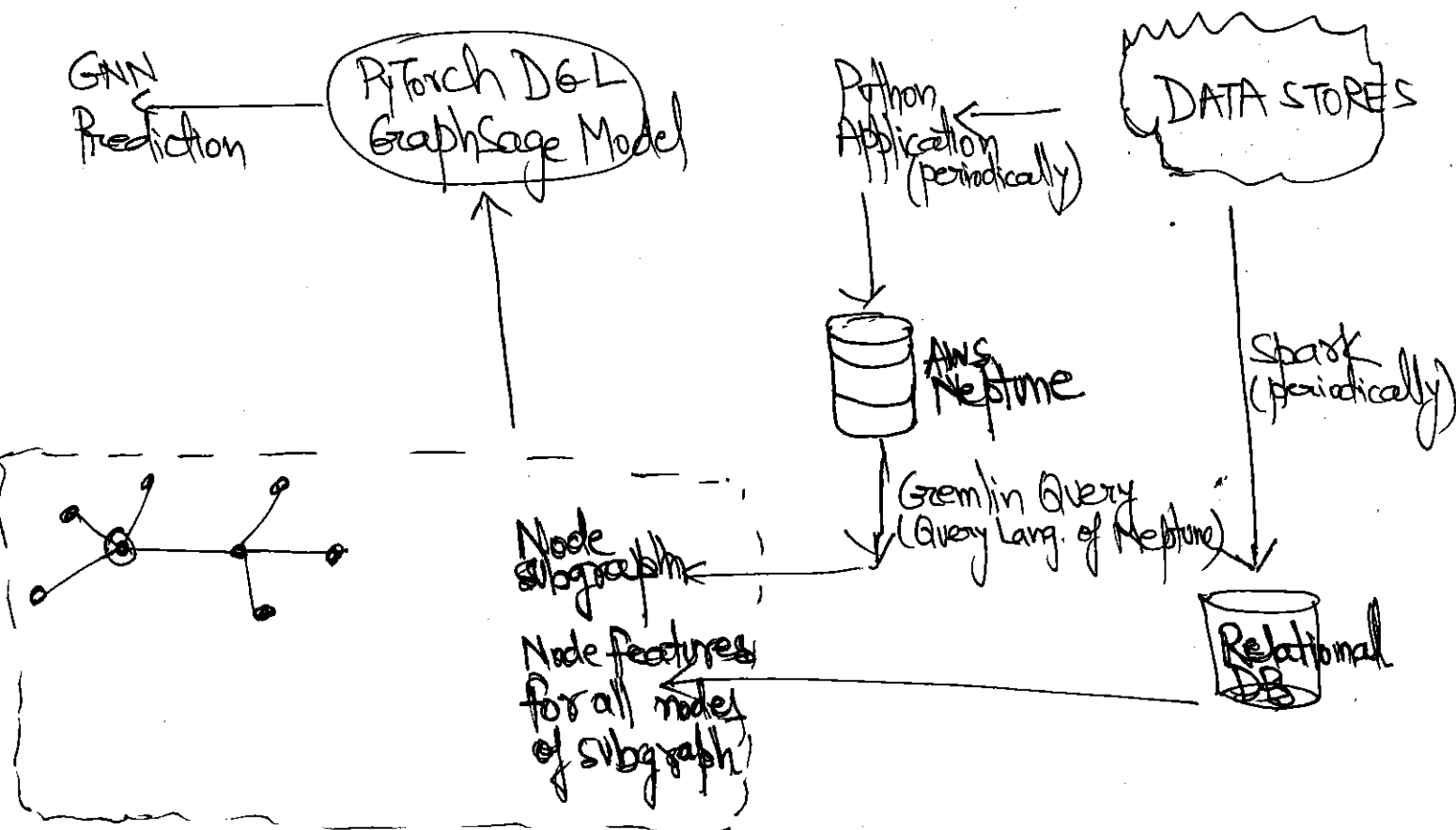
After training, the model is saved as PyTorch (.pth) object

Then the model is packaged as Docker Image

Deployed on Kubernetes infrastructure as a microservice using flask

→ Inference Steps:

- i) Client sends a POST request to endpoint /gnn\_prediction containing "USER\_ID" field as the main key in its JSON body
- ii) The request body starts to build the subgraph related to "USER\_ID" by calling the graph DB
- iii) The features related to "USER\_ID" ~~are~~ <sup>neighbors</sup> are fetched from relational DB and combined with the subgraph in a processed DGL subgraph ready to be used for prediction
- iv) GraphSage predicts using this DGL subgraph and response is returned to client



ARCHITECTURE FOR GRAPHSAGE DEPLOYMENT



# SEARCH SYSTEM DESIGN

→ Item Search  
(Product search)

## Ve Content Search

Vc Content Search Vs So  
(eg. Video search, web search etc)

## Vs Social Media Search



Represent them as vectors (e.g. embeddings)

Represent them as vectors (e.g. embeddings)  
Find similarity with query (e.g. cosine similarity, dot product)  
or 2 Tower Neural N/w

08 2 Tower Neural N/w

→ Personalized (logged in) vs Non-personalized (not logged in)

→ 3 Types of queries:

- Head : Tend to be short words/phrases  
Largest vol. of queries

- Lexical / keyword search

- Tail : Tend to be longer words or multi-word phrases } <sup>Search</sup>
- Smallest vol. of queries

-Toro: Everything in between } Lexical search + Semantic search

→ ONLINE METRICS:

1. CTR (may click irrelevant results as well)
2. Dwell time after click (e.g. 10 seconds dwell time after click)
3. Time to success
  - └ Min. no. of queries (to get desired result)
  - └ Relevant result at the top

→ OFFLINE METRICS:

- Candidate Generation Phase: Recall@K
- Ranking Phase: NDCG
- MAP
- Precision@K
- Top-K

→ Apart from query and item features, following could be used to enhance results:

- 1) User's previous session history : Ex. if search for "windows" then user's prev. search history can tell us whether the user is searching for real windows or windows PC
- 2) User profile : Profile about age, geography, gender, etc can help in refining search results
- 3) Item Age : To account for freshness
- 4) Item Popularity : To account for item popularity

# TYPO CORRECTION (AUTO CORRECT, SPELLING CORRECTION)

## - FUZZY MATCHING

1. Identify misspelled word: if not found in dictionary  $\rightarrow$  misspelled
2. Find ~~the~~ strings 'n' edit distance away
  - Edit types:
    - 1) Insert (add a letter)
    - 2) Delete (delete a letter)
    - 3) Replace (replace a letter with another letter)
    - 4) Switch (swap two nearby letters)
  - 'n' : number of edits (usually 1 to 3)
  - Edit distance could also be calculated using 'Levenshtein distance'  
e.g. AC  $\rightarrow$  CA (2 edit distance away using Levenshtein dist)
3. Filter candidate words:
  - After finding words 'n' edit distance away, find words which appear in "dictionary" of known words
4. Calc. Prob. of <sup>candidate</sup> ~~misspelled~~ word given misspelled word  
(e.g. a separate classifier trained on misspelled word and true word)

Example: whoosh library, also available in Elastic search

## AUTO TAGGING:

- Items/Content need to be auto tagged to aid in Keyword search
- OCR, Vision APIs, topic modeling can be used to extract tags  
(Pdf) (Image) (text content)  
(Trained on labelled data before)

## KEYWORD SEARCH

- User Query : → Type correction → Token/Keyword Extraction  
Stemming/Lemmatization
- Item/Content → Tagged, keywords extracted from description, title, etc.

→ Inverted Index <sup>e.g.</sup> (elastic search) (Index is usually from doc → terms here opposite)

Terms	Doc
Room	Doc1
Area	Doc9
Garden	Doc3 Doc8 Doc2

- BM25 : Term based ranking model which scores documents based on term frequency and document length

## SEMANTIC SEARCH : Identify Intent of Query

- Query Understanding (optional)
- Retrieval of Candidates (usually approx. KNN)
- Ranking (add more features and use classification models)

Semantic search aims to match a user's query with items, even though there are no word matches

### QUERY UNDERSTANDING

- Usually happens before search engine retrieves and ranks results

Components:

- Tokenization / Stemming / Lemmatization
- Query Correction (typo correction)
- Query expansion (synonyms, acronym expansion) } → Recall ↑
- Query reduction (removes terms) } → Precision ↑
- Entity Recognition / Query Scoping (apple : brand or fruit)
- Query Segmentation: divides search query into a seq. of semantic units (one or more tokens)  
e.g. "machine learning" framework  
"machine" "learning framework"

# TOKENIZATION FOR DNN:

- Unigram, bigram, char trigram
- Sub-word tokenization
  - Do not split the frequently used words into smaller subword
  - Split rare words into smaller meaningful words
- Eg. do not split 'boy' but split 'boys' into 'boy' and 's'

- Solution b/w. word tokenization & char. tokenization

<u>Cons</u>	<u>Pros</u>
<ul style="list-style-type: none"><li>- Vocab → large</li><li>- Sol<sup>n</sup>: limit vocab to few thousand words</li><li>- If limiting to few thousand words, lots of out of vocab words (OOV)</li></ul>	<ul style="list-style-type: none"><li>- limited Vocab: fixed # of chars</li><li>- Very few oov</li></ul>
	<u>Cons</u>
	<ul style="list-style-type: none"><li>- Char does not have any meaning or info as word does</li></ul>

- Used in several SOTA models (e.g. BERT)
- Example algo: wordpiece, sentencepiece

- Typo CORRECTION: Subword tokenization + Char ngrams make the model robust to typos and noise

- OOV TOKEN ISSUE: Sol<sup>n</sup>: 1) Subword tokenization + char ngrams  
2) Hashing: Each oov token is hashed into a bucket, the bucket is assigned an embedding and treated like a special token  
(con: hash collisions)

[↶ Back to posts](#)

# Feature Engineering for Recommendation Systems — Part 1

**Nikhil Garg**

6 min. read | July 29, 2022

The way ML features are typically written for NLP, vision, and some other domains is very different from the way they are written for recommendation systems. In this series, we'll examine the most common ways of engineering features for recommendation systems.

In the first part (i.e., this post), we will look at counter and rate-based features and a few best practices related to writing them. In the second part, we will look at a few more techniques and best practices for writing good counter/rate based features. And in the third and final part, we will look at more advanced kinds of features that work well for large neural networks. But first, let's address a beginner FAQ right away.

**FAQ: I thought recommendation systems were all about collaborative filtering, which doesn't need ML features. So, what features are we talking about here?**

**Answer:** as described in this post, recommendation systems have many distinct stages. Collaborative filtering is one of the many algorithms for powering (a subset of) retrieval. Irrespective of the choice of retrieval algorithms, candidates obtained from retrieval still need to be ranked by another ML model — often called the ranking model. This post is about the features used in ranking ML models.



# What are counter and rate-based features?

The golden principle of any modern personalization system (including recommendations) is that learning directly from people's behaviors works a lot better than learning from their stated preferences. And one of the best and simplest ways of featurizing behavioral patterns is through counter and rate-based features.

A counter feature, well, is just that — a feature that counts something, typically some sort of user behavior. For instance, the number of times a user has clicked a product of some category encodes their behavioral affinity towards that category and is a counter feature.

A rate feature is a near cousin of counter feature — it's simply the ratio of two related counters. As an example, in the above example of user-category affinity, a feature that may capture the user behavior even better is the fraction of times the user clicks on a product of some category upon viewing it.

Whenever I start on ANY recommendation problem, I always start with a few counter / rate features along with a basic GBDT model. This baseline of counter/rate features with GBDT takes a model very, very far. In fact, these features are responsible for a large part of model performance for even the most sophisticated ranking/recommendation systems in FAANG and other top tech companies.

Okay, so hopefully, you're convinced that you should write counter / rate based features. Let's now examine some techniques and best practices for using them.

## 1. Choosing the appropriate splits

Typically you'd want to write lots of counter features corresponding to various kinds of splits. Here is an example of related but different counter features for a social media recommendation system:

While you should choose any good splits, you should also be careful about not choosing splits that are too sparse. For instance, if one of your split is the click rate of user on content from a publisher on a given topic between 11am-12 noon, most likely almost all the splits are empty, and this feature is too sparse. Super sparse features like this don't help the model much<sup>[1]</sup> and in fact, come at a huge computational/storage cost. How much sparsity is too much is hard to spell out, but my rule of thumb is to not have  $O(N^2)$  splits where  $N$  is the number of large-volume entities. For instance, a feature with (# of user, # of products) is potentially fine since the number of products in most e-commerce engines is small (though even this feature might not work at Amazon scale). A similar feature with (# of user, # of authors) is probably fine in the media world. But feature with (# of user, # of posts) might be too sparse even at a smaller scale.

### 3. Don't forget to encode context

Most users have different behaviors based on their own context. For instance, they may be more interested in short byte sized headlines in mornings and longer-form content later in the evening. Or even more dynamic things like how much time they have already spent on your app since morning may affect their behavior.

These contextual signals are very powerful for model to make sense of the user behavior and make a huge difference in the quality. But the right way to encode them is once again via counter/rate features. So in addition to throwing the time of the day as a categorical feature, you might also want to add a feature that measures the user activity (say CTR) of user across all content at this time of the day.

Context features don't have to be limited to capturing the preferences of a single user. They can also be used to encode broad preferences across many users. For instance, one powerful feature for any video streaming recommendation engine is — CTR of all users who are on WIFI on any video. This feature captures the idea that users who are on WIFI (vs, say mobile data) are much more likely to watch full videos.

In the above example of time of day CTR, should we create one feature per hour of the day? For instance, what is the CTR of the user between 12-1am, 1-2am, ....10-11pm, 11-12am? A much smarter idea is to only throw in one feature, i.e., CTR of the user at the **current** time of the day. That is, if the feature is evaluated at 5pm, use the user CTR between 5-6pm and if the same feature is evaluated at 8am, use the user CTR between 8-9am.

In fact, this idea is pretty universal when writing counter based features and is one area where beginners get stuck. As another example, say we want to create a counter feature capturing the user's affinity towards the author of a post. We roughly know that we want to capture the CTR of this user on the post authors. But there are thousands of authors in the system — should we create thousands of features, one for each author?

Once again, this idea of a dynamic match comes to the rescue. The feature as evaluated on (user, post) will be defined as follows: what is the CTR of the user on the content written by the author of this post. And so the same feature will lookup the user's CTR on different authors for different candidate posts. [2]

## Conclusion

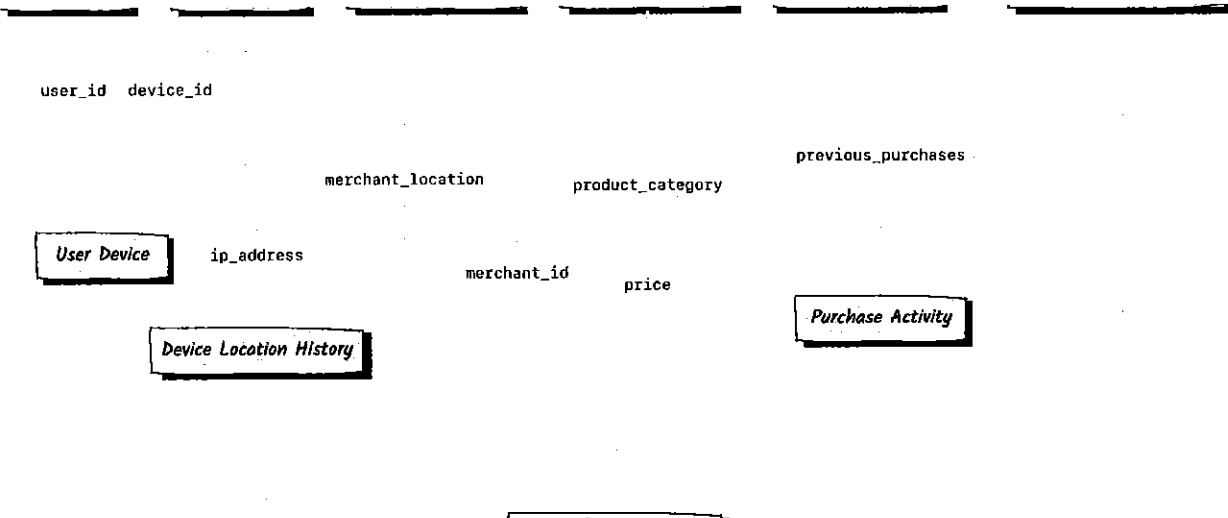
Unlike many other kinds of ML, most of the feature engineering for any recommendation or personalization system is based on counter or rate features. In this post, we reviewed some techniques and the best practices of encoding useful signals as counter and rate features. In the second part of this series published here, we will discuss a few more techniques for encoding counter / rate features. And in the third part of series, we will discuss how feature engineering is done for more advanced neural network-based recommendation systems — stay tuned

1. These super sparse features can be useful if you're training a large deep neural network with lots of data. But they aren't useful for regular-sized data with GBDTs. ↩



for an (user, author) pairs, but the actual feature has some more logic that determines which of these it is going to look up. ↩

# Other articles you might enjoy



## Feature Engineering for Fraud Detection

Nick Parsons | 9 min. read | December 07, 2022

User	Candidate	Action	Context	Aggregation	Window
uid gender <u>age</u> location cohort ...	id <u>publisher</u> category picture quality ...	like share <u>click</u> long click ...	time of day day of week mobile device wifi vs 4g ...	count <u>rate</u>	1 hour 1 day 3 days <u>7 days</u> 28 days

## Feature Engineering for Recommendation Systems — Part 2



- Users infrequently searching for the same information once they've found it.
- Users often using different queries (synonyms, slightly different levels of detail, etc.) to search for the same information, whether it is different users searching for the same information, or the same user returning at a different time to find information they searched for in the past.

For personalized search, some of the most common kinds of features include:

- ✓ **NLP-Type Features** — Features that understand the semantic content of the query. These might come from SOTA models (such as BERT), FastText (either directly fed or the dot product of the query), or document embeddings.
- ✓ **Simple Content-Based Features** — Simple features that give “basic metrics” for the documents, like the character or word count, presence of stop words, the term frequency-inverse document frequency (TF-IDF) of the query terms in the document, etc.
- ✓ **Counter-Based Engagement Features** — User/documentation engagement features, such as number of clicks the document has received, number of long clicks the document has received (clicks which had a view time > x seconds), bounce rate, etc.
- ✓ **Reputation Features** — Features representing how reputable of a source the document comes from. These might include the author of the document, the number of inbound links, the page rank, etc.
- **Context Features** — Features representing the context for how or when the query happened, such as the time of day or day of the week when the query was performed, the user's location and/or IP address, the device the query was made from, whether the query was made on mobile or web, etc.

## Challenges in Feature Engineering for Search

```

{
  "my-index-000001" : {
    "settings" : {
      "index" : {
        "number_of_replicas": "1",
        "number_of_shards": "1",
        "creation_date": "1474389951325",
        "uuid": "n6gzFZTgS664GUfx0Xrpjw",
        "version": {
          "created": ...
        },
        "routing": {
          "allocation": {
            "include": {
              "_tier_preference": "data_content"
            }
          }
        },
        "provided_name" : "my-index-000001"
      }
    }
  }
}

```

By default `flat_settings` is set to `false`.

## Fuzziness

Some queries and APIs support parameters to allow inexact fuzzy matching, using the `fuzziness` parameter.

When querying text or keyword fields, fuzziness is interpreted as a Levenshtein Edit Distance — the number of one character changes that need to be made to one string to make it the same as another string.

The `fuzziness` parameter can be specified as:

`0, 1, 2` The maximum allowed Levenshtein Edit Distance (or number of edits)

`AUTO`

Generates an edit distance based on the length of the term. Low and high distance arguments may be optionally provided `AUTO: [low], [high]`. If not specified, the default values are 3 and 6, equivalent to `AUTO: 3, 6` that make for lengths:

`0..2`

Must match exactly

`3..5`

One edit allowed

`>5`

Two edits allowed

`AUTO` should generally be the preferred value for `fuzziness`.

[↩ Back to posts](#)

# Feature Engineering for Fraud Detection

**Nick Parsons Nikhil Garg**

9 min. read | December 07, 2022

## Introduction

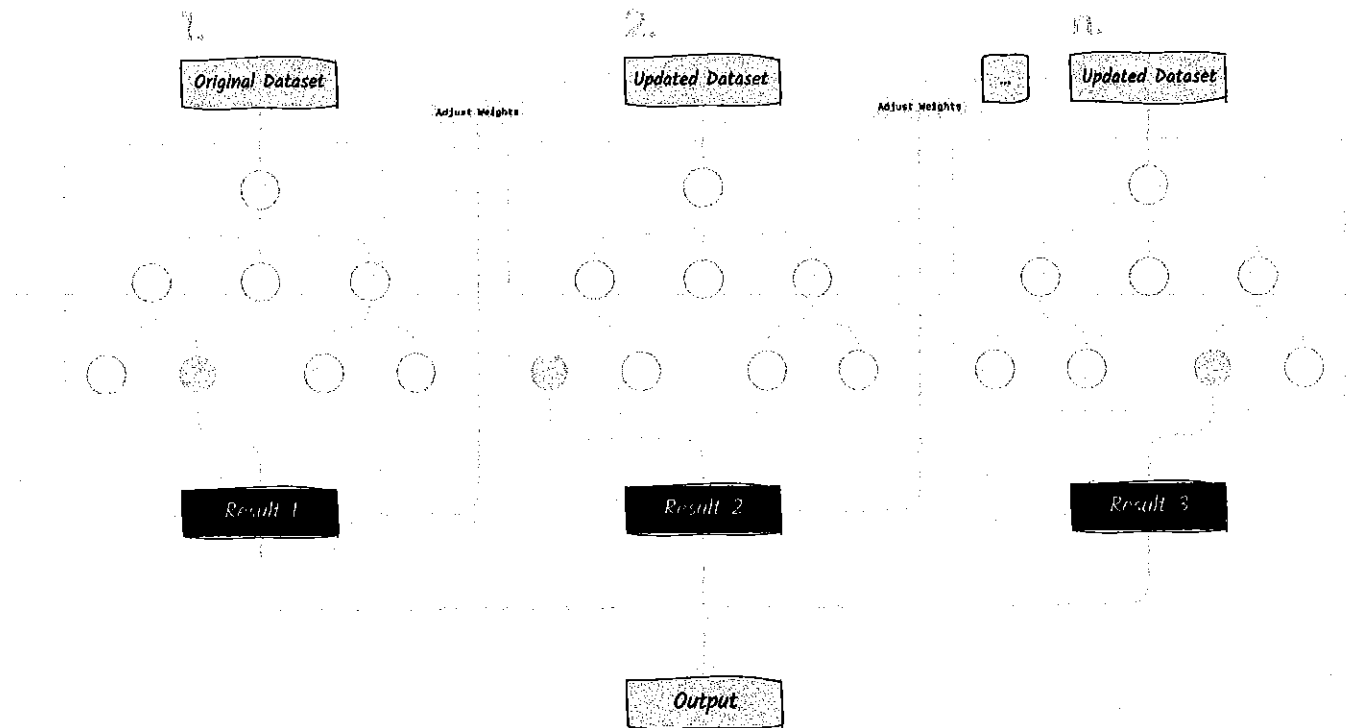
Fraud detection is critical in keeping remediating fraud and services safe and functional. First and foremost, it helps to protect businesses and individuals from financial loss. By identifying potential instances of fraud, companies can take steps to prevent fraudulent activity from occurring, which can save them a significant amount of money. Fraud detection also saves users a lot of headaches and instills trust in them when they know these protections are in place.

There is a wide variety of applications under the broader umbrella of fraud/risk detection — credit card transaction fraud, payment fraud, identity fraud, account takeover, etc. All of these types of fraud or risk can be broadly categorized into two types — first-party fraud (i.e., fraudulent users trying to trick a bank) or third-party fraud (i.e., a bad actor who is trying to compromise the account of a legitimate user). Most of these applications have a similar structure when it comes to ML and feature engineering, which we will explore in this post. First, we will briefly describe how the modeling problem is formulated and how the labels are obtained. And finally, we will do a deep dive into feature engineering for fraud.

## Modeling



GBDTs work by training a series of decision trees on the data, where each tree is built to correct the mistakes made by the previous trees in the sequence.



In the case of fraud detection, the features input into the GBDTs might include information about the transaction, the customer involved, and any other relevant financial or behavioral data, and whether training data includes labels that indicate if a transaction is fraudulent or not.

The GBDT algorithm uses this information to identify patterns and relationships, even complex non-linear relationships, that are indicative of fraudulent activity. By training the model on a large dataset of known fraudulent and non-fraudulent transactions, the GBDT algorithm can learn to make highly accurate predictions about the likelihood of a given transaction being fraudulent.

Fraud detection often involves working with large and highly unbalanced datasets, where the number of fraudulent transactions is relatively small compared to the number of non-fraudulent



- Oversampling the fraudulent transactions or undersampling the legitimate transactions — this can balance out the dataset, but can also lead the model to believe that fraud is more frequent than it is.
- Changing the metrics used to evaluate the model — you can use metrics like precision, rather than accuracy, or use the F1 score to evaluate how well the model is performing.
- Using algorithms like ensemble methods or cost-sensitive learning — Both of these algorithms can help improve the performance of the machine learning model on the minority class. GBDT (which itself is an ensemble) is a great algorithm for fraud detection because it is powerful and flexible, and relatively easy to use and interpret, but some other ensemble methods are particularly effective at handling imbalanced datasets, and cost-sensitive learning is particularly adept at considering the costs associated with false positives and false negatives.

(scale\_pos\_wt hyperparam)  
in xgboost

## Labels

Since the problem is formulated as supervised learning, ground truth labels are needed to train the model — in this case, whether transactions as fraudulent or non-fraudulent.

In some problems, e.g., credit card transaction fraud, if a transaction was incorrectly classified as non-fraudulent and permitted to happen, the end user may dispute the charge. That can serve as good quality labels. But in general, obtaining high-quality labels in a timely fashion is a huge problem for fraud detection, unlike recommendation systems where what the user clicked on can easily serve as good-quality labels.

As a result, typically, humans often need to be involved to manually provide the ground truth labels along with a bunch of automated augmentation/correction mechanisms.

Another challenge with labels is that there is a delay in receiving labels. There is typically a 2-3 week delay in processing fraudulent activity on an account. In some problems, like account takeovers, malicious actors may wait months before starting to use the accounts to say post



to get 70% of all labels which means models have to be trained on several months old data at any point.

## Feature Engineering

In order to create effective features for fraud detection, it is necessary to understand the underlying data and the problem that the model is trying to solve (first or third-party fraud, as well as the types of fraud therein). This often involves a combination of domain expertise, data exploration, and experimentation to identify the most relevant and informative features to use as input (often accomplished through exploratory data analysis, or “EDA”). Once these features have been identified, they may need to be transformed or combined in order to make them more useful for training a machine learning model (also included in EDA). This process of feature engineering is crucial for achieving good performance in fraud detection and other machine learning tasks.

One of the challenges with feature engineering for fraud in particular is that, like with labels, much of the most useful features need to be handwritten. There are some features that can be used from the raw data, but, more often than not, the most useful features are ones that combine several different pieces of data, like the amount and the location and time of day (e.g., someone is likely to spend more money on dinner in a high cost-of-living area than they are to spend at a coffee shop in a low cost-of-living area) and/or splitting the values for a specific piece of data (like splitting the date from the time or the dollars from the cents of a transaction amount).

Some of the most common patterns of features for fraud detection include:

- Affinity features built on top of rolling counter-based features — how many times an event of some kind happened in rolling windows (e.g., how many times a user has made an ATM withdrawal in this city in the last 6 months or the average transaction amount for a user with a particular credit card).
- Velocity features — how quickly events happen (e.g., the ratio of the purchases the user made in the last hour to the average number of transactions they made per hour in the last



- Reputation features — some “reputation” score for various things like the email domain of the user, the IP address the activity is coming from, the vendor the purchase was made from, etc. Some of these are, again, using counters of past behavior.
- External API-based features — e.g., the credit score of the user or location of an IP, etc.
- Relatively static profile features — e.g., the zip code from which the request originated or the age of the user’s account, etc.

As mentioned above, determining which of these features is most useful involves a lot of industry knowledge and experimentation (since the normal log and wait process would require you to wait months for results, due to the nature of the data being received and the rate at which it is received). When doing experimentation via exploratory data analysis (EDA), the engineer uses different statistics and visualizations of the data to find the best data points and combinations by looking for patterns, anomalies, and relationships between data (which can also inform the engineer that only one piece of data from the relationship needs to be used).

Note that even though fraud models are trained on very old training data, realtime ML still gives you a massive leg up in fraud detection because of its adeptness with long-tail, as referenced in this post.

## Enjoying the article?

Join 1000+ others on our newsletter mailing list and get content direct to your inbox!

Submit

Feature engineering for fraud detection can be challenging for a number of reasons. As was mentioned before, fraud is a complex and dynamic problem that can take many different forms, which makes it difficult to identify the specific patterns and relationships that are indicative of fraudulent activity.

Fraud detection often requires working with sensitive financial data, which may have strict privacy and security requirements. This can make it difficult to obtain the data that is needed for feature engineering and can also limit the types of transformations and manipulations that can be performed on the data. When working with sensitive data, your best chances of success are to:

- Ensure data is properly secured — to protect against unauthorized access, implement security measures such as encryption, access controls, and monitoring.
- Consider which data is actually necessary — during the feature engineering stage, using tactics such as exploratory data analysis (EDA) helps you to see if any of the features in consideration have relationships with others, which can allow you to select the less sensitive metric for the model that will be put into production.
- Anonymize the data as much as possible — your use of sensitive data is more likely to be allowed to continue if proper protections are put in place. The first layer of ensuring this is making sure only the people who should have access to the data do, as mentioned in the first bullet. It is also a good practice to anonymize data within your system, in case of situations such as compromised credentials being utilized by bad actors.

Another challenge is that, as previously mentioned, labels are often delayed, so training data consists of very old examples (sometimes several months old). This means that, if you were to employ the popular practice of logging and waiting, you would need to wait months to test a new feature. This is why a practice such as EDA is often used. Another “issue” with labels being delayed is that point-in-time data feature reconstruction needs to cover a long window (often times a year) to get the necessary data volume. This also increases the risk of feature code

Earlier, we mentioned that it is especially common to create features for fraud detection models by combining or splitting the features you receive from the raw data. This often means that you need more data sources to get enough data of the types you need, which increases the need to hit external APIs. Sometimes, this comes in the form of batched data dumps (like the credit scores for a large number of people as of a certain date) that need to be “merged” with live requests (like to a credit bureau) in a lambda-like architecture. With live requests, it is also important to consider the latency constraints mandated by the system or use case. In most cases with fraud detection, it is very sensitive to lag, since a bad actor can do a lot of damage in even a few seconds. This means that features must be only a few seconds old, if that. In addition, serving latencies of the models using those features need to be incredibly low; often, a serving latency of around 200ms end-to-end (including feature extraction and model scoring) is required to make a decision as to whether a transaction is fraud in an acceptable amount of time.

Another reason that working with external APIs (e.g., hitting credit bureau APIs for credit scores) is hard is that it can be difficult to create correct point-in-time values; since the data is external, you don't always have easy access to historical data or the details you need from it, meaning you need to ensure you record all of the right information when you first access the data. For example, we mentioned that the IP address is a feature that can be used to predict fraud; one issue with this is that the location of an IP changes over time, so you need to know the location of the IP at the time of training, which mandates careful managing of the data obtained from external APIs, so that you have the details you need when you need them in the future.

## Final Thoughts

Feature engineering is a complex artform, and the precision and thoughtfulness that needs to go into it when working with sensitive data and high stakes, like in the case of fraud detection, further complicates its planning and implementation. While there are a good number of things to account for when performing feature engineering for fraud detection, this article helps to highlight many of the challenges, technologies, and strategies that can be employed throughout this process to give you a leg up when going through the process.