Predicting Completeness of Unstructured Shipping Addresses Using Ensemble Models

Vedang A. Waradpande
vedang.waradpande@razorpay.com
Razorpay Software Pvt. Ltd.
Bengaluru, Karnataka, India

Petchetti Vinay Surya Prakash
surya.kanna@razorpay.com
Razorpay Software Pvt. Ltd.
Bengaluru, Karnataka, India

Nikhil Jhaveri
nikhil.jhaveri@razorpay.com
Razorpay Software Pvt. Ltd.
Bengaluru, Karnataka, India

Shashank Agarwal
shashank734@gmail.com
Razorpay Software Pvt. Ltd.
Bengaluru, Karnataka, India

ABSTRACT

We present a novel solution to an inevitable issue faced by all current-day e-commerce companies in developing countries like India to a large extent. The issue concerns failed deliveries due to incomplete or bogus shipping addresses that is a major source of financial loss for these companies. Identifying the completeness of an address in India is a challenging task due to (i) lack of hierarchy in the address, (ii) names of the same places varying in different languages and dialects, (iii) structure of the address varying based on location, etc. We suggest a hybrid approach using a 1D ConvNet and Extreme Gradient Boosting (XGBoost) to solve this complex problem. The ConvNet learns character-level embeddings and captures the language semantics in the address, while the XGBoost model leverages our hand-crafted features specifically devised for this task using domain knowledge. We present novel methods for Address Parsing and detection of Monkey Typed addresses which we use to build our features. Our model learns and generalizes well, achieving an AUC value of **0.94** on an unseen test set, and outperforms various traditional and deep learning models. Finally, we demonstrate the practical value of our solution by creating a highly efficient and fast prediction service using our model, which is already deployed to serve real-time classification queries and has been found to be very useful by several online stores.

CCS CONCEPTS

• **Information systems → Document representation**; • **Applied computing → E-commerce infrastructure**.

KEYWORDS

Address Classification, Convolutional Neural Networks, XGBoost, E-Commerce, Text Classification, Machine Learning

ACM Reference Format:
Vedang A. Waradpande, Petchetti Vinay Surya Prakash, Nikhil Jhaveri, and Shashank Agarwal. 2021. Predicting Completeness of Unstructured Shipping Addresses Using Ensemble Models. In *In Proceedings of ACM SIGIR Workshop on eCommerce (SIGIR 2021 eCom).* ACM, New York, NY, USA, 9 pages.

SIGIR eCom'21, July 15, 2021, Virtual Event, Montreal, Canada

1 INTRODUCTION

In India, many orders placed in online stores cannot be delivered, and the items are sent back to the seller. This situation is known as Return To Origin (RTO) among e-commerce companies, and in this scenario, the company owning the store incurs corresponding costs. For e.g., delivery partners need to be paid for the attempt, and if the shipped goods are perishable, the cost of making them is an added loss. This incurred loss can be heavier if the order is a Cash on Delivery (COD) order, where the customer pays upon delivery. RTOs present a major issue to the e-commerce ecosystem in India, with nearly 30% [1, 2] of orders on an average resulting in RTOs across various store sizes, industries, and locations.

While RTOs can happen because of a variety of factors such as the fraudulent intent of the customer, rejection of impulsively ordered items, ground-level problems while shipping, etc., one of the prominent factors is that the shipping address provided by the customer is incomplete. This can mean that the address does not contain specific details that make delivery possible; for instance, a missing street number can render the shipping address incomplete. A shipping address can also be non-deliverable, i.e., it points to a place where deliveries cannot happen, like a park or bus stop. Finally, it can also be a non-existent address.

Predicting if a shipping address is incomplete is a daunting task, given the challenges that addresses in India present us with. Some of these are as follows:

- Customers generally don't follow a fixed structure when they mention the shipping address. So a given address can be written in many different ways, even by those that live at the address. For e.g. two addresses such as *"D-12, plot-5, Sunder Apartments, MG rd., near Jain temple, Dwarka, New Delhi"* and *"D-12, plot-5, Sector - 4, Dwarka, New Delhi - 110047"* point to the same location.
- Postal Index Numbers (PIN codes) are 6-digit codes used by the India Post and provide a way to assign addresses to small areas with some precision. However, customers are sometimes not aware of where one such area ends and another starts, and as a result, two addresses with different areas or pin codes may actually point to the same place.

- What constitutes a deliverable address varies by location. For e.g. in a metropolis like Mumbai, addresses would need to be very specific to be deliverable because of the high population density. However, in rural areas, deliveries can be made at a relatively less specific address because of the delivery person's familiarity with addresses and sparse population.
- Due to the fact that India is highly diverse linguistically, landmarks, streets, and even areas may be referred to by different names by various groups of people. For instance, crossroads are called *chowk* in the state of Maharashtra and *chouraha* in various North Indian states.
- The components of the address may be written in any order.
- Shipping addresses may be written in languages other than English.
- Common spelling mistakes just add to the complexity of the problem.

In practice, to verify the deliverability of user-entered addresses, e-commerce companies primarily rely on address validation (lookup in official databases [3]) and geocoding (assigning coordinates to address [4]) APIs. The lack of standardization in addresses of developing countries makes these tools very hard to use.

Given the complexity of the problem and the massive number of variables, rule-based approaches generally fail at this task. Recently, Machine Learning algorithms have been harnessed to solve various associated problems, including address classification. However, to the best of our knowledge, there is no model which can successfully predict if a given shipping address is complete. In this paper, we propose an ensemble of an XGBoost model and a Convolutional Neural Network, for this task. Our key and novel contributions can be summarized as follows:

- We explain why datasets that give delivery information are not sufficient and introduce a dataset of 12,376 samples for this task.
- We propose an ensemble model for predicting if a shipping address is incomplete and show that it works better than several traditional and deep classification models. It also performs well on an unseen test set.
- We show that our approach is practically effective by building a real-time prediction service around the model and briefly describe its implementation.

This paper is structured as follows: Section 2 describes previous efforts to study addresses and solve other related problems. Section 3 provides an intuitive idea of address completeness and introduces some terminology. Section 4 explains the features we created, the architecture of the model, and some implementation details, while Section 5 discusses our experiments and their results. Finally, section 6 concludes and discusses future research directions and applications of our model.

2 RELATED WORK

Several problems related to addresses such as detecting monkey-typed addresses, geocoding, and various kinds of classification tasks have been explored to some extent individually in various papers.

In [5], Gevaers et al. define the last mile problem and highlight the challenges it poses such as increased overall cost, reduced delivery efficiency and environmental impact. Babu et al. [6] worked on classifying an address into a smaller sub-region accurately using various preprocessing techniques and models. Mangalgi and Lakshya et al. [7] further worked on the problem of address classification into sub-regions from a language-modeling perspective and experiment with traditional ML approaches, Bi-LSTMs [8], and the RoBERTa [9] model (which they finally propose). However, classifying addresses into sub-regions is not scalable for larger areas like countries without massive labeled data. Seng [10] worked on classifying Malaysian addresses according to property type (condominium, landed residential homes, and business premises). The proposed architecture in this paper is an LSTM model with an embedding layer. Kakkar and Babu [11] experimented with various techniques for clustering addresses and propose Leader clustering with word embeddings (*add2vec*), which can be used for the comparison of new addresses with existing ones. Kejriwal and Schekely [12] worked on creating embeddings from names of populated locations using DeepWalk, a network embedding algorithm.

Detecting the presence of monkey-typing in addresses is another problem we address in our paper. Previously, Babu et. al. [13] experimented with various ML models to detect monkey-typing, by creating two kinds of features: i. text features around which groups of characters are common and abnormal and ii. those which leverage the absence of vowels in the address. There has also been some work done on Address Parsing, another important component of our model, which involves separating the address into meaningful components [14–16].

We approach this problem from a text classification perspective, for which both traditional and deep learning models have been used extensively. Powerful and scalable traditional models such as XGBoost [17] and LightGBM [18] can have excellent performance on this task [19]. Deep learning models have the advantage of not requiring any feature engineering efforts and have thus been widely adapted for text classification, with Convolutional [20] and Recurrent [21] Neural Networks being the most popular architectures adapted for this task. While word-level models seem to be more common, character-level models have also been used [22].

To the best of our knowledge, this is the first work aiming to classify addresses according to their completeness by consolidating various components like Monkey Typing and Address Parsing. Our model and prediction service provides a fast and scalable solution for e-commerce companies to dynamically take decisions based on the address input by the user on checkout.

3 ADDRESS COMPLETENESS

In this section, we try to provide an intuitive understanding of address completeness and introduce terminology that we will use throughout the paper.

E-commerce companies generally store addresses in 6 components: Address Line 1, Address Line 2, Pin Code, City, State, and Country, and we mostly receive our data in this format. Hence, there may be no need to isolate components like cities from the address. However, we don't always expect this structure to exist. For instance, in case the online store only provides one Address Line, we assign the value "" to Address Line 2, and so on. We can also combine these components to form an address string, such

as *"G-43, Rajnath Apartments, Koramangala, Bengaluru, Karnataka, India"*.

We can attribute the completeness of an address to several aspects. For the above address, components such as *"Rajnath Apartments"* need to exist in the given city. Another aspect is the amount of detail in this address; if a delivery person goes to *"Koramangala"* would be they be able to find *"Rajnath Apartments"* without knowing the street number or some other landmark? Other factors also play a role here - the sequence of characters can tell us if this address is monkey-typed and the size of the mentioned city can tell us how specific the address needs to be for it to be deliverable. Broadly, we divide these aspects into three categories - (i) geographical context, (ii) word, entity, and address level information, and (iii) character-level relations.

During our feature creation process, we have looked at addresses as a sequential combination of the following units, and created features capturing information at each of these levels:

- **Words.** Each group of characters separated by a space is considered as a word in the address. In the above example, the words would be *"G-43"*, *"Rajnath"*, *"Apartments"*, *"Koramangala"* and *"Bengaluru"*.
- **Tokens.** Each set of words that represents a real-world entity such as an area, landmark, etc. In the above example the tokens would be *"G-43"*, *"Rajnath Apartments"*, *"Koramangala"* and *"Bengaluru"*.
- **Characters.** Each character in the address string.

4 MODEL

Our ensemble consists of an XGBoost model and a ConvNet, the probability predictions of which are given as input to a linear SVM, which gives a binary prediction (incomplete(1), complete(0)) as shown in figure 2. Subsections 4.1-4.3, describe the various kinds of features we created which are then used to train the XGBoost model and subsection 4.4 discusses the training and implementation details of the model.

4.1 Capturing Geographical Context

Locations in addresses have geographical relations between them. For instance, in the fictional address we considered, *"Koramangala"* is located inside of *"Bengaluru"*, and we need to inject this information into our model. We achieve this in two ways: (i) by creating a tree-based data structure that contains such relations captured from a large corpus of addresses, and (ii) by creating features that check the structure of the address in several ways.

The Address Component Tree (ACT) is a tree-based data structure with each level (from top to bottom) representing state/union territories, city/district, and locality respectively, and is used primarily for validation on these 3 levels. For instance, it can be used to validate if the given city exists in the given state or if the given locality exists in the given city, and so on. To build this tree, we sampled over 2 million addresses on which successful deliveries have been completed and extracted state, city, and locality information. The top-level consists of 36 nodes, each of which corresponds to one of the 28 states or one of the 8 union territories in India.

Cities corresponding to each of these states and localities corresponding to each of the cities were captured from the large address corpus. There were several mistakes in the way the customers had entered the city in the city field. For instance, some entries featured more information apart from the city, others had alternate or older names for the city (a popular example being *"Bangalore"*, which was renamed to *"Bengaluru"* in 2014 but the two names are still used interchangeably) and yet others had common spelling mistakes (for e.g. *"Hyderbad"* instead of *"Hyderabad"*). We thus pruned the city nodes by using three major tools: phonetic matching, Levenshtein Distance, and frequency. While comparison we followed a normalization process, which included removing special characters and conversion of the city string to lowercase.

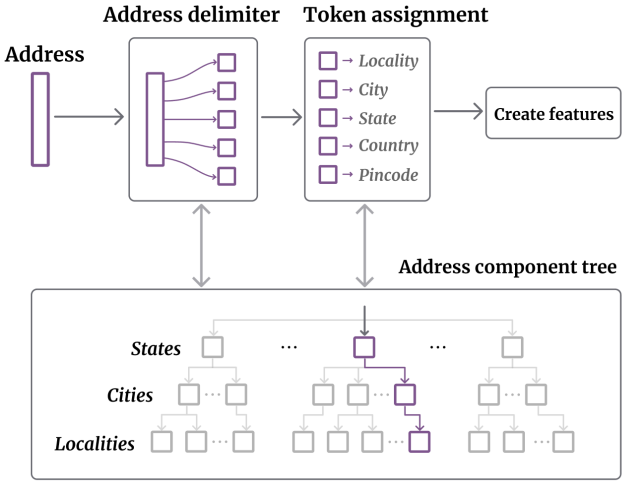

Figure 1: Use of Address Component Tree for extracting geographical information

Localities are not provided by the customer separately and we need to identify tokens that correspond to the locality in Address Lines 1 and 2. This is a challenging task since the order of tokens can be arbitrary in the address lines and most users don't delimit the components in their addresses using special characters such as commas. We thus build an Address Parser to separate localities from Address Lines 1 and 2. While extracting localities from the large corpus of addresses, we don't use the ACT in the parser and add the extracted localities to the ACT.

After the ACT is built, our Address Parser works in two steps. Figure 1 provides more clarity on this architecture.

(1) **Address Delimiter.** The goal of this module is to break the address into its constituent tokens by inserting commas in the right places. We first join Address Line 1 and Address Line 2 using a comma. We divide words that commonly occur in addresses into several categories based on how likely they are to occur: (i) just before a comma (e.g. *"apartment"*, *"road"*, etc.), (ii) just after a comma, (e.g. *"near"*, *"hno."*), (iii) both before and after commas (e.g. *"tower"*, *"block"*, etc.), (iv) neither before nor after a comma (e.g. *"of"*, *"no."*). We then assign two scores to each word, which denote how likely a comma is to lie before and after the word respectively. Using a combination of these scores and rules, we add commas in the address and hence break it into tokens.

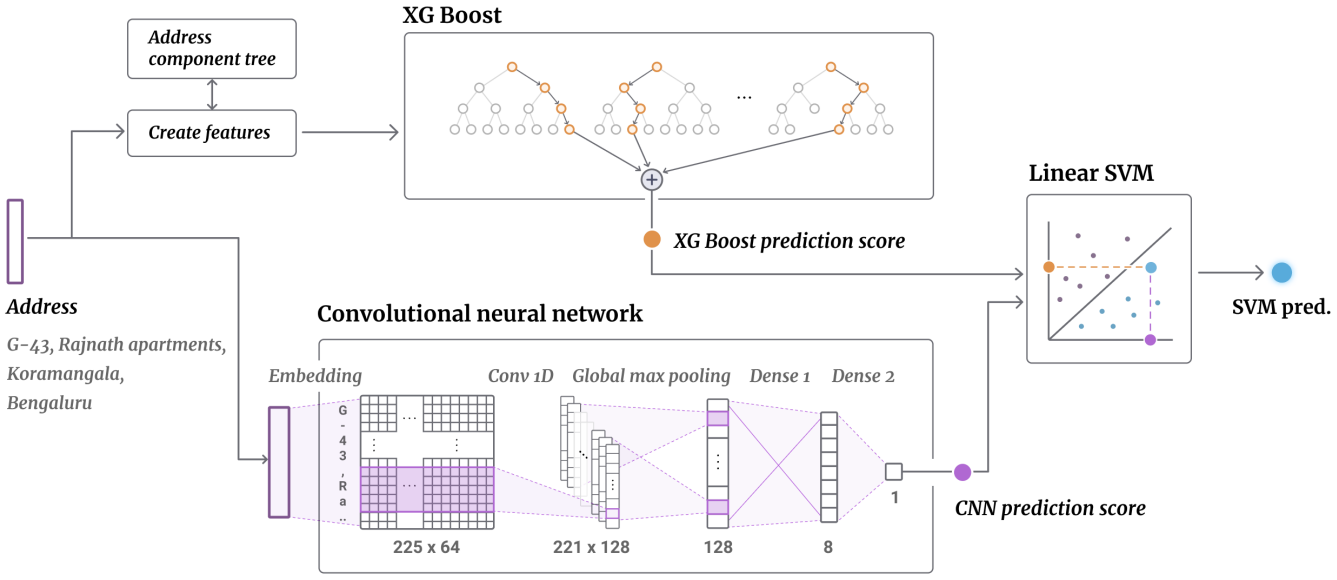

Figure 2: Ensemble model for classification of addresses

(2) **Token assignment.** We then assign predefined labels to these tokens; there are 12 labels which include *"unit"*, *"block"*, *"society"*, *"street"*, *"landmark"*, *"place_of_interest"*, *"locality"*, *"pin_code"*, *"city"*, *"state"*, *"country"* and *"unassigned"*. While pin code, city, and state are provided by the user separately, these are still included in the labels since many users still provide these in the address lines. From our large corpus of addresses, we first isolate common words for each token. For instance, words like *"heights"* and *"colony"* are likely to appear at the *"society"* level. Using these word groups and the ACT, we then assign probability scores that denote how likely they are to be assigned to each of the labels and greedily assign them to the most likely label. Multiple tokens may be assigned to the same label. For tokens still unassigned (because of a tie or zero scores), we adopt a hierarchical approach and check the labels nearby tokens have been assigned to, and assign a label accordingly. All tokens with no labels are then labeled *"unassigned"*.

We use the Address Parser and Address Component Tree for creating features involving various validations and checks (see subsection 4.2). Apart from this, we also created features such as population density, city tier, and percentage of urban vs rural population in the area surrounding the address on the pin code, city, and state level.

4.2 Information on Various Levels

We create several features for capturing information on 3 levels: words, tokens, and address.

4.2.1 Word level features. Basic word-level features such as the mean and median length of words in the address, the total number of words with length less than a threshold (= 3), and the number of words with digits in them were experimentally found to work well. Apart from these, we use the concept of a word score, based on how many words frequently found in our larger corpus of addresses are found in the given address. Another associated feature is the number of frequent words occurring in the given address.

4.2.2 Token level features. Since tokens represent real entities, the features created on this level are based on validations performed on the assignment created by the Address Parser. A basic type of validation is to check which labels have tokens assigned to them. Other useful validations include checking if the various components of the address have valid information present in them. For instance, pin codes in India always have 6 digits with the first digit being non-zero and any pin codes not following this format can thus be said to be invalid. We also used features based on the likelihood of both ordered and unordered bi-grams in our address. The likelihood is calculated using our larger corpus of addresses.

4.2.3 Address level features. Features created on the address level include certain properties about the address and the results of various validations performed on it.

- **Address properties.** The absence of any digits in Address Lines 1 & 2 heavily indicates that the address may not be deliverable. The few exceptions to this include prominent establishments or landmarks, especially in rural areas. Some indicators of a good address were found to be a relatively high number of words in the address and a high number of delimiters such as commas provided by the user to differentiate between tokens in the address lines. A higher number of vowels and symbols also contributes positively towards an address being complete. Some online stores also provide the user with an option to mark the given address as a home or an office address; these properties are also useful for determining the completeness of the address.

- **Address validations.** Based on our parsed address, we create three different scores for each address based on how informative, correct and complete they were respectively. These scores, which range from 0 to 1, are based on prior knowledge and observation of addresses from these perspectives. One of the key validations we performed was based on the hypothesis that addresses in rural (tier-3) areas need not be too specific for a delivery to happen. We used the city tier to check if an address is rural and the parsed address to check what information is provided by the user.

4.3 Monkey Typing

Monkey-typed addresses are random and often repetitive sequences of characters which the user inputs in the address fields to get to the next screen. These kinds of addresses can be safely assumed to be not deliverable. Overall, this feature creation module aims to detect two patterns of Monkey Typing made from a computer keyboard: (i) random, non-recurring characters typed with one or 2 hands and (ii) recurring patterns as are sometimes seen in monkey-typed text. The outputs from algorithm 1 are provided as input to our model. Apart from these, our CNN model also learns character-level relations which indicate Monkey Typing, since these are a subset of the incomplete samples. Considering the complex structure of this module, we don't fully elaborate on all utility functions but explain their function and output.

The function Has-Monkey-Typing returns two Boolean values, h, r denoting the presence of one or two hand random monkey typing and recurring pattern monkey typing respectively. These are then added to the features of the respective sample.

Algorithm 1 Address has Monkey Typing

```
function Has-Monkey-Typing(s)                          ▷ s: sentence
    s ← convert all characters in s to lowercase
    s' ← replace all characters except a − z with space ("")
    T = s'.split(" + ")          ▷ T: list of all words separated by whitespace in s
    h = false                    ▷ h: is one or two hand MT present
    r = false                    ▷ r: is recurring pattern MT present

    for i = 0 → T.length do
        if Has-One-Hand-Mt(T[i]) or Has-Two-Hand-Mt(T[i]) then
            h = true
            break

    r ← Has-Recurring-Pattern-Mt(s)

    return h, r
```

4.3.1 One-Hand Monkey Typing. To detect one-hand monkey typing, we consider 3 different checks given in the following functions. We declare an address monkey typed only if all three conditions are satisfied.

- Is-Msd-Based: We assume that unique keystrokes with one hand should be localized to a certain region of the keyboard, and hence the mean square distance between unique keystrokes should not be too high. The mean squared (Euclidean) distance between keystrokes, D_{msd} is calculated using Γ, a pre-defined map between keys and "co-ordinates", treating the keyboard as a 2D plane. The co-ordinates of the keys are $'q' : (0.0, 0.0)$, $'w' : (1.0, 0.0)$, $'a' : (0.2, 1.0)$ and so on.
- Has-N-Consecutive-Components: We posit that random keystrokes seldom have coherent syllables attached together by vowels, as is the norm in English. If words have long blocks of consecutive consonants, there is a higher chance of them being monkey typed.
- Is-Direction-Based: Here, we assume that the user will typically not change "direction" much while typing. The direction in question is captured by the change in x and y coordinates in consecutive keystrokes.

Algorithm 2 Address has One Hand Monkey Typing

```
function Has-One-Hand-Mt(w)                            ▷ w: word
    if Is-Direction-Based(w) and Is-Msd-Based(w) and Has-N-Consecutive-Consonants(w) then
        return true
```

4.3.2 Two-Hand Monkey Typing. We assume that the user enters random characters through each hand. In the function Has-Two-Hand-Mt, we try to assign each entered character to the left or right hand, based on the direction change, the mean squared distance between consecutive keystrokes, and the hand assigned to the last character. This is captured in the function Get-Prev-And-Curr-Hand, which returns the hand assigned to the previous and current keystroke. Finally, we check if the unique keystrokes assigned to each hand follow the random pattern by calling the procedure Has-One-Hand-Mt. The parameter $\tau_{min}(= 8)$ indicates the minimum number of characters needed in a word to consider it for 2-hand monkey typing. $\tau_l(= 4)$ and $\tau_r(= 4)$ are corresponding length thresholds on the subsequences assigned to the left and right hand.

4.3.3 Recurring Pattern Monkey Typing. Users commonly display a tendency to repeat patterns of characters while monkey typing. We detect this in the function Has-Recurring-Pattern-Mt by extracting substrings with a window size w, which is varied as $w \in \{1, 2, ..., \lfloor s.length/2 \rfloor\}$. Finally, their frequency is checked, and if the frequency of any substring is greater than a threshold $f_{min}(= 3)$, we flag the address for displaying recurrent pattern monkey typing. The threshold $\tau(= 4)$ indicates the minimum indicates the minimum number of characters present in the sentence for it to be considered for Recurring pattern monkey typing.

4.4 Learning and Implementation

We explain the process for training the model in this subsection.

4.4.1 Feature Selection. We had originally created over 250 features for training the XGBoost model. To select important features, we first remove columns that had a very high correlation with each other (> 0.95), keeping only one of two such columns based on domain knowledge or model importance. We iteratively train an XGBoost model and remove features with very low or zero importance. We use 114 features for training our final XGBoost model.

Algorithm 3 Address has Two-Hand Monkey Typing

function HAS-TWO-HAND-MT(w, τ_{min}, τ_l, τ_r)
  if $w < \tau_{min}$ or w contains special characters **then**
    return $false$

  Initialize map M_{hand} with keys l, r and corresponding values as empty lists and initialize h_l as *None*
  for $i = 0 \rightarrow w.length - 1$ **do**
    $h_{prev}, h_{curr} \leftarrow$ GET-PREV-AND-CURR-HAND($w[i], w[i+1], h_l$)

    if $i = 0$ **then**
      Add $w[i]$ to $M_{hand}[h_{prev}]$
    Add $w[i+1]$ to $M_{hand}[h_{curr}]$
    $h_l \leftarrow h_{curr}$

  if $M_{hand}['l'].length < \tau_l$ or $M_{hand}['r'].length < \tau_r$ **then**
    return $false$

  $h_{left} \leftarrow$ remove all consecutive multiple occurences of every character with a single character in $M_{hand}['l']$
  $h_{right} \leftarrow$ remove all consecutive multiple occurences of every character with a single character in $M_{hand}['r']$

  if HAS-ONE-HAND-MT(h_{left}) and HAS-ONE-HAND-MT(h_{right}) **then**
    return $true$

  return $false$

Algorithm 4 Address has Recurring Pattern Monkey Typing

function HAS-RECURRING-PATTERN-MT(s, w, f_{min}, τ) ▷ s: sentence, w: window size, f_{min}: min freq. for repeated patterns, τ: min length threshold
  $s' \leftarrow$ convert all characters in s to lowercase
  $s'' \leftarrow$ remove all non alphabetical characters from s'
  if $s''.length <= \tau$ **then**
    return $false$

  Let σ be an empty list
  for $i = 1 \rightarrow w$ **do**
    for $j = 1 \rightarrow s''.length - 1$ by w **do**
      Add substring from indices $min(i + j, s''.length)$ to $min(i + j + w, s''.length)$ to σ

  $\sigma_f \leftarrow$ select substrings λ from σ such that $\lambda.length = w$
  Let M_{freq} be a map with keys as substrings and values as their frequencies in σ_f

  if $M_{freq}[\lambda] > f_{min}$ for any λ in $M_{freq}.keys$ **then**
    return $true$

  return $false$

4.4.2 XGBoost optimization. We use XGBoost in the supervised setting to learn from all features created from the addresses in subsections 4.1-4.3. Let $D = \{(\mathbf{x}_i, y_i)\}, \mathbf{x}_i \in \mathbb{R}^m, y_i \in \{0, 1\}$ represent the training set with n samples, each of which contains features $\mathbf{x}_i$ and binary labels y_i denoting complete/not complete. Once trained, the XGBoost algorithm predicts the output for new data samples by aggregating the output from each of the K tree structures. Let $f_k(x)$ be the function represented by the k^{th} tree structure and $\mathcal{F}$ be the space of regression trees, CART. Then,

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i),\ f_k \in \mathcal{F} \tag{1}$$

The gradient tree boosting algorithm behind XGBoost minimizes a regularized objective function:

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \tag{2}$$
$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

Tree structures are chosen in a greedy manner. Let $\hat{y}_i^{(t)}$ represent the prediction of the i-th instance at the t-th iteration. We choose the tree structure f_t which best minimizes the loss mentioned in Eq. 2.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t) \tag{3}$$

The native Python package of XGBoost mentioned in [17] also allows parallel tree training and is highly scalable, which is useful for our purposes. We use the H2O AutoML platform [23] for hyperparameter search for our XGBoost model. Our final XGBoost model is trained for 124 rounds with the learning objective as *"binary:logistic"*. The learning rate (*"eta"*) is set to 0.3, the maximum depth for each tree (*"max_depth"*) is set to 20. Except for parameters *"subsample"* = 0.6, *"min_child_weight"* = 10, *"colsample_bytree"* = 0.8, and *"colsample_bylevel"* = 0.8, all other parameters are set to default.

4.4.3 ConvNet optimization. We use a character-level CNN instead of a word-level model. The primary motivation behind this is that several "words" such as those representing last-mile information (e.g. *"a-703"*) are likely to be extremely infrequent across addresses. The character-level model also performed better experimentally. Characters in addresses have temporal relations with each other given the inherent hierarchy of tokens we expect in the address. For instance, house numbers usually occur at the start of the address, and the city name should likely occur towards the end.

The input string to the ConvNet is a concatenation of Address Lines 1 and 2, padded to 225 characters with trailing 0s where address length is less than 225 characters. We use an embedding layer to learn character representations. In terms of notation and function, we closely follow [20] and use Tensorflow [24] for implementation.

We present the working of our 1D convolution and global max-pooling layers. Let $\mathbf{x}_i \in \mathbb{R}^k$ be a k-dimensional vector corresponding to a character in an address of length n. The address can be represented as a matrix with concatenated ([]) character vectors.

Model	Validation		Test		Training Time (secs)
	Precision	AUC	Precision	AUC	
Logistic Regression	0.482 ± 0.0405	0.737 ± 0.0152	0.447 ± 0.0158	0.733 ± 0.0136	1.65
Bi-LSTM	0.741 ± 0.0290	0.921 ± 0.0041	0.734 ± 0.0207	0.919 ± 0.0042	1074.20
CNN	0.784 ± 0.0275	0.928 ± 0.0044	0.804 ± 0.0202	0.928 ± 0.0046	40.75
XGBoost	0.836 ± 0.0162	0.939 ± 0.0053	0.804 ± 0.0150	0.931 ± 0.0033	11.35
CNN + XGBoost Ens.	**0.841 ± 0.0293**	**0.944 ± 0.0039**	**0.833 ± 0.0274**	**0.942 ± 0.0048**	**298.89**

Table 1: Performance with 95% confidence intervals of baselines and proposed model; cutoff fixed where recall = 0.6.

$$\mathbf{x}_{1:n} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n] \tag{4}$$

Let $\mathbf{x}_{i:i+j}$ denote the concatenation of characters $[\mathbf{x}_i, ..., \mathbf{x}_{i+j}]$. The convolution operation is performed by applying the (jth) filter $\mathbf{w}_j \in \mathbb{R}^{hk}$ to a window of h words:

$$c_{i,j} = \sigma(\mathbf{w}_j \cdot \mathbf{x}_{i:i+h-1} + b) \tag{5}$$

where $b \in \mathbb{R}$ is the bias term and σ is a non-linear activation function; in our case the Rectified Linear Unit [25], $ReLU$.

$$ReLU(x) = x \text{ if } x > 0 \text{ else } 0 \tag{6}$$

Applying m filters will produce m feature maps $\mathbf{z} = [c_1, c_2, ..., c_m]$, where $c_{i,j}$ is the jth element of the ith vector c_i. The global max pooling layer will select the maximum element from each of these feature maps, thus resulting in a single vector of size m. Our 1D-Conv layer contains 128 filters of size 5. This is followed by a Dense layer with a dropout rate of 0.5 and 8 neurons, each with a $ReLU(.)$ activation function. The final Dense layer with only 1 neuron has a sigmoid activation function given the binary classification task.

We use the Adam Optimizer [26] and binary cross-entropy loss function to optimize our model. The weights are initialized using the GlorotUniform initializer [27]. We train the model for 15 epochs with an Early Stopping mechanism by monitoring the validation loss with a patience of 2 epochs. The samples themselves are divided into mini-batches of 15 while training the model. The values for all parameters have been finalized after rigorous experimentation.

4.4.4 Ensembling. We train both our XGBoost model and ConvNet on our training set with binary labels (complete/not complete). We perform k-fold cross validation to evaluate their performance as well as form the inputs for the combining model, Linear SVM [28]. Let α be an address from the training set, $\mathbf{x}_{feat}$ be its featurized representation and $\mathbf{x}_{emb}$ be its input to the ConvNet. Let, $x_{xgb} = XGBoost(\mathbf{x}_{feat})$ and $x_{cnn} = ConvNet(\mathbf{x}_{emb})$, then the input feature vector to the linear SVM is

$$\mathbf{x}_{svm} = [x_{xgb}, x_{cnn}] \tag{7}$$

where x_{xgb} and x_{cnn} are the out-of-fold cross validation predictions (probability values) of the XGBoost and ConvNet models respectively. The linear SVM model is trained on such pairs of probability values to predict labels $y \in \{0, 1\}$.

4.4.5 Implementation. We build a real-time prediction service using our model, which is now used in our product, *Thirdwatch* [29]. We use Apache Flink [30, 31] as our execution engine since it provides stream processing, high parallelization, and fault-tolerant dataflows. Features and character-level representations for a new address are calculated in parallel at run-time. We observed a low average latency of 202 ms and Apache Flink's parallelization and Asynchronous I/O also allow us to process multiple requests to the API at the same time. We also wish to highlight that we chose a rule-based approach for certain components like Monkey Typing and Address Parsing to avoid sequential model calls which may reduce the latency of the prediction service.

5 EXPERIMENTS

5.1 Dataset

Several client companies shared their data with us for training our models. This data consists of order details including shipping address and the final delivery status of individual items in the order such as *"rto"*, *"in_transit"*, *"canceled"*, *"fulfilled"*, *"lost"*, and so on. We initially experimented with this data for predicting address completeness but this was incorrect because there can be other reasons for an order to result in an RTO than an incomplete address, and an incomplete address does not necessarily result in an RTO. To overcome this problem, we sampled a dataset of 12,376 shipping addresses from this data and manually labeled each address as complete (0) or incomplete (1). These addresses were sampled over three different periods of time, in a stratified way such that the total number of addresses selected for each tier (1, 2, and 3) is the same. We split this dataset by assigning 80% of the addresses to the training set and 20% to the testing set. The ratio of addresses in each tier is kept constant in each of the training, validation (wherever used), and test datasets.

The labeling was carried out in two phases. First, three individuals (not the authors) were asked to label the addresses. The process involves finding a match on Google Maps [32] as close as possible to the current address. The individuals then try to check if enough information is present to take someone from the matched address to the shipping location. In certain difficult cases, they take a majority vote and rely on domain knowledge. Second, the authors review the labels in random order. To avoid biasing the data, the authors do not change the labels unilaterally but ask for a review in case they feel a particular label is erroneous.

5.2 Setup

We use classification models such as Logistic Regression (LR), Naive Bayes (NB), Multi-Layer Perceptron (MLP), XGBoost, ConvNet, and Bidirectional LSTM as baselines for comparison. The ConvNet and Bi-LSTM are trained with an embedding layer and all other models utilize our hand-crafted features. For all models except the proposed

Tier	Precision	Recall	AUC	Sample Addresses Marked Incomplete (1) & Complete (0) by Model
tier-1	0.79	0.60	0.94	*Incomplete* 1. Bamnoli, Delhi - 110077 2. elcot sez sholinganallur, 600119, chennai
				Complete 1. flat 211, prem's sai brindavan apartments, annapurna enclave, hyderabad - 500050 2. 159 Othaivadai street Kodambakkam, Chennai - 600024
tier-2	0.88	0.60	0.94	*Incomplete* 1. 121, thrissur - 680601 2. Visram Sadan, Rajkot - 360001
				Complete 1. Kalyani, Ponnurunni road vyttila, Ernakulam,Cochin - 682019 2. vaatsalya school, near head post off., vishwakarma colony, ajmer-305624
tier-3	0.89	0.60	0.91	*Incomplete* 1. rasra Kotwari ballia, Kotwari-221712 2. holenarasipura hassan, holenarasipura - 573211
				Complete 1. cell city mobile elambulasserri,mannarkkad-678582 2. hno 267 laxette varca salcete goa varca Church, Varca - 403721

Table 2: Tier-wise performance of ensemble model on test data

model, we train the model on the full training set containing 80% of all addresses, validate using 5-fold cross-validation, and test on the testing set. Hyperparameter optimization is achieved using Grid Search for LR and NB and the H2O AutoML interface for XGBoost.

5.3 Results

We present our top-performing models (and LR as a baseline) in table 1. We used the metrics precision, recall, and area under the receiver operating characteristic (AUC) for evaluating each of the models. The learning capability and translation to unseen datasets are evaluated using AUC. We don't use accuracy since there is a class imbalance in the data we use, with only 15.5% of addresses in both the train and test set being non-deliverable. Our model outperforms all other classifiers with an AUC value of 0.94 in both validation and testing phases. It should be noted that the XGBoost model performance comaparable to our model on the validation set with a far lesser training time. However, in production, we train our models at an interval of a few days and so we choose the model which translates better to the test set.

The consumption of the prediction from our service is subject to manual review and decision from the online store, with each review costing them some amount of money. Low false positives, and hence, high precision while maintaining good recall is important. To compare precision for these models, we fix the cutoff such that the recall is set to 0.6 for each of the classifiers. For all models, we report the mean precision, recall, AUC and training time for 100 trials with various weight initializations and random folds for cross-validation and training, as well as their 95% confidence intervals, calculated using the bootstrap method [33] (also over 100 trials).

Given that the level of detail in addresses required for delivery changes drastically between city tiers, we also report the metrics for each of the tiers in table 2. This provides us with one level of filter for checking the quality of predictions given by the model. We further provide random examples of addresses that have been marked incomplete by the model for each city tier. It is clear that the model performs better for city tier 2 and 3 as compared to tier 1. This may be due to the fact that tier 1 addresses require more detail to be deliverable as opposed to tiers 2 and 3, which constitute a majority of the data.

6 CONCLUSIONS AND FUTURE WORK

We introduce a novel XGBoost + CNN ensemble approach for predicting if an Indian shipping address is complete enough for delivery. A linear SVM combines the probability scores from the CNN and XGBoost models to provide the final prediction. We sample and label a dataset of 12,376 addresses with equal addresses across city tiers for our experiments, which show that our model outperforms several classification models and generalizes well to an unseen test set. We further implement and test a scalable real-time prediction service which proves that our approach is practically sound.

One of the limitations of this approach is the small size of the dataset. This approach also combines novel solutions to several research problems such as Address Parsing and detection of Monkey Typing. With advancements in these, our approach can be improved. Another simple improvement can be replacing the Address Component Tree with a knowledge graph and find related tokens by graph search. A further change can be providing confidence scores for each address being incomplete.

Even with these limitations, however, our model can serve several use-cases. For instance, with our prediction service, action can be taken even before the shipping process for the product has begun, such as: i. Decide dynamically whether to display the cash-on-delivery option to the user based on the address they input, ii. Schedule messages or automated call systems to confirm (and update) the address of users, etc. Further research of unstructured addresses in developing countries like India can help build more robust systems which reduce the chances of problems such as Return to Origin.

REFERENCES

[1] Abenaya Gunasekaran. How cash on delivery propels indian e-commerce - razorpay thirdwatch. https://razorpay.com/blog/cash-on-delivery-e-commerce-india/, Oct 2020.

[2] Chhavi Tyagi. Catch me if you can: For ecommerce fraudsters it may be the end of the game. *The Economic Times*, Dec 2017.

[3] Kevin Conti. White paper: The future of address validation. Technical report, Pitney Bowes.

[4] Claudio Owusu, Yu Lan, Minrui Zheng, Wenwu Tang, and Eric Delmelle. Geocoding fundamentals and associated challenges. *Geospatial data science techniques and applications*, pages 41–62, 2017.

[5] Roel Gevaers, Eddy Van de Voorde, and Thierry Vanelslander. Characteristics and typology of last-mile logistics from an innovation perspective in an urban context. *City Distribution and Urban Freight Transport: Multiple Perspectives, Edward Elgar Publishing*, pages 56–71, 2011.

[6] T Ravindra Babu, Abhranil Chatterjee, Shivram Khandeparker, A Vamsi Subhash, and Sawan Gupta. Geographical address classification without using geolocation coordinates. In *Proceedings of the 9th Workshop on Geographic Information Retrieval*, pages 1–10, 2015.

[7] Shreyas Mangalgi, Lakshya Kumar, and Ravindra Babu Tallamraju. Deep contextual embeddings for address classification in e-commerce. *CoRR*, abs/2007.03020, 2020.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[10] Loke Kar Seng. A two-stage text-based approach to postal delivery address classification using long short-term memory neural networks.

[11] Vishal Kakkar and T Ravindra Babu. Address clustering for e-commerce applications. In *eCOM@ SIGIR*, 2018.

[12] Mayank Kejriwal and Pedro Szekely. Neural embeddings for populated geonames locations. In *International Semantic Web Conference*, pages 139–146. Springer, 2017.

[13] T Ravindra Babu and Vishal Kakkar. Address fraud: Monkey typed address classification for e-commerce applications. In *eCOM@ SIGIR*, 2017.

[14] Hao Li, Wei Lu, Pengjun Xie, and Linlin Li. Neural chinese address parsing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3421–3431, 2019.

[15] Minlue Wang, Valeriia Haberland, Amos Yeo, Andrew Martin, John Howroyd, and J Mark Bishop. A probabilistic address parser using conditional random fields and stochastic regular grammar. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 225–232. IEEE, 2016.

[16] Nosheen Abid, Adnan ul Hasan, and Faisal Shafait. Deepparse: A trainable postal address parser. In *2018 Digital Image Computing: Techniques and Applications (DICTA)*, pages 1–8. IEEE, 2018.

[17] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[18] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30:3146–3154, 2017.

[19] Qian Li, Hao Peng, Jianxin Li, Congying Xia, Renyu Yang, Lichao Sun, Philip S. Yu, and Lifang He. A survey on text classification: From shallow to deep learning. *CoRR*, abs/2008.00364, 2020.

[20] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[21] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. Recurrent neural network for text classification with multi-task learning. *CoRR*, abs/1605.05101, 2016.

[22] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

[23] Erin LeDell and Sebastien Poirier. H2O AutoML: Scalable automatic machine learning. *7th ICML Workshop on Automated Machine Learning (AutoML)*, July 2020.

[24] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[25] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[26] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[27] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[28] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[29] Razorpay thirdwatch. https://razorpay.com/thirdwatch/.

[30] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[31] Shashank Agarwal. Data science at scale using apache flink. https://medium.com/razorpay-unfiltered/data-science-at-scale-using-apache-flink-982cb18848b, 2019.

[32] Google maps. https://maps.google.com/.

[33] Bradley Efron and Robert Tibshirani. Improvements on cross-validation: the 632+ bootstrap method. *Journal of the American Statistical Association*, 92(438):548–560, 1997.