Data Cleaning and Variable Creation for Applications Data

Shreyash Kondakindi

September 21, 2025

1 Introduction

This report details the data preprocessing and feature engineering pipeline applied to a fraud detection applications dataset. The goal of this pipeline is to transform raw application data (containing personally identifiable information, dates, etc.) into a clean set of informative features suitable for modeling fraud risk. We describe each major step, including data loading, cleansing of spurious values, construction of composite identity keys, temporal feature calculations (e.g., recent application velocities and time since last occurrence), cross-entity aggregations, and final feature selection. Each operation is explained in terms of what is done, why it is needed to improve data integrity or model performance, and how it is implemented in code.

2 Data Loading and Initial Preprocessing

The raw applications data is first loaded into a pandas DataFrame. Key fields such as Social Security Number (SSN) are immediately converted to string type to preserve any leading zeros and to treat them as categorical identifiers rather than numeric values (e.g., an SSN `123456789` is converted to `"123456789"`). Date fields, including application date and date of birth (DOB), are parsed into datetime objects to enable time-based computations. Basic cleaning is performed on text fields: trimming whitespace, converting to a consistent case (e.g., lowercasing names and addresses) to ensure that identical values are recognized as such. As an initial feature, the applicant's age is derived from the DOB and the application date. This is computed by taking the difference in days between the application date and DOB and converting to years (e.g., via integer division by 365). The resulting age feature provides a numeric representation of DOB that is more directly useful for modeling (since age can have a non-linear relationship with fraud risk, whereas raw DOB is not model-friendly). By converting DOB to age, we both maintain privacy (DOB is sensitive PII) and create a more relevant feature. Additionally, obviously irrelevant or constant columns (if any exist in the raw data) are dropped. For example, if the dataset contained an ID field or any columns with the same value for all records, those would be removed at this stage to reduce noise. We ensure that after initial preprocessing, the dataset's fields are cleanly typed (numeric, categorical, dates as appropriate) and ready for more complex transformations.

3 Cleaning Frivolous PII Field Values

A critical cleaning step is to handle ”dummy” or placeholder values in PII fields that do not represent real, distinctive information. Such frivolous values are often used by applicants trying to hide or when information is not available, and if left unaddressed they could cause unrelated applications to appear falsely linked. We identify several fields and values for this treatment:

- **Social Security Number (SSN):** Instances of SSN given as `999999999` (or equivalently formatted `999-99-9999`) are dummy placeholders. These are replaced with a null or special value (e.g., an empty string or `"000000000"`) to indicate a missing or invalid SSN. This prevents multiple applications with SSN 999-99-9999 from being linked together as if they shared a legitimate SSN.
- **Phone Number:** Likewise, phone numbers given as `9999999999` (all 9's) are recognized as bogus. We replace these with a null or placeholder (e.g., `"0000000000"` or an explicit `"dummy_phone"` tag) to avoid spurious matches on phone number.
- **Street Address:** A known fake address pattern is `"123 MAIN ST68138"` (which appears to concatenate a fake street with a zip code). This and similar obviously fake addresses are replaced with a null or generic placeholder (such as `"UNKNOWN"` address). By doing so, we ensure that two different applicants who both put "123 MAIN ST68138" are not considered to live at the same real address for linking purposes.
- **Other PII fields:** We similarly check for trivial or filler values in other fields like email or name. For instance, an email of `"test@test.com"` or a last name `"LN"` might be placeholders. Any such occurrences are standardized to a null/placeholder to avoid misleading links.

These replacements are implemented using pandas operations, for example by applying conditional logic or using the `replace()` method for each field. After this cleaning, any PII field that contained a frivolous value is either null (which the model can treat as missing) or set to a harmless constant that is used consistently. This prevents the creation of features that erroneously tie together many records (for example, without cleaning, dozens of records might share the SSN `999-99-9999` and thus appear as the same person). By nullifying these dummy values, subsequent feature engineering steps will treat them as missing rather than genuine matches. To facilitate entity-based feature engineering, we create new composite keys by concatenating existing fields. These entity keys represent different granularities of identity and are used to link applications that share common personal information:

- **Full Name Key:** We combine the first name and last name into a single `fullname` key (e.g., `"John"` + `"Doe"` = `"JohnDoe"`). Prior to concatenation, names are standardized (trimmed and case-normalized) so that `"JOHN"` and `"John"` both map to `"john"` for consistency. The full name key helps capture cases where the same person (or two people with identical names) apply multiple times.
- **Name+DOB Key:** We further combine the full name with date of birth to create a `name_dob` key. For example, `"JohnDoe"` + `"1980-01-15"` might yield `"JohnDoe1980-01-15"`. This key is a more precise identifier of an individual, since two people with the same name but different birthdates will have different `name_dob` keys. It helps to distinguish common names and link records that truly refer to the same person.
- **Address Key:** We may also construct an address composite, such as concatenating street address with ZIP code (after standardizing the format). This yields an `address_zip` key representing a unique location. Similarly, one could form a `name_address` key if needed (combining an applicant's name with their address), which might capture household or family fraud patterns.
- **Other Keys:** Depending on the data, additional keys are constructed. For example, a phone-based key (combining phone area code with last name, or just using phone number alone as a key), or email-based keys (like email prefix + last name) can be derived. We also treat the SSN itself (cleaned as above) as an entity key on its own.

These composite keys are created using straightforward string concatenation in pandas (often via the `astype(str)` and string addition or using `df.apply` with a lambda). By generating these keys, we enable the model to consider whether two applications share a certain piece of identity information. For instance, the same `name_dob` appearing in two records strongly suggests the same real person, which could indicate repeat behavior. In the next steps, we will use these keys to derive features like counts and recent frequencies of occurrences.

4 Feature Engineering

Using the base fields and newly created keys, we engineer a comprehensive set of features that capture patterns of application behavior and linkages between applications. We describe each group of features below.

4.1 Days Since Last Occurrence

For each entity (each type of key or identifying field), we calculate the *days since last match* feature. This feature measures the recency of the previous application that shared this entity with the current application. It is computed by sorting the data chronologically by application date and, for each record and each key:

- Find the most recent prior application (if any) that has the same key value.
- Compute the difference in days between the current application's date and that prior application's date.

If no prior match is found (meaning this is the first time that particular value appears), we assign a default value (such as a large number or a special null indicator) to signify "no previous occurrence." For example, if an SSN `123-45-6789` was seen in an application on June 1 and then again on June 10, the second occurrence would have `ssn_days_since_last` = 9 days. This is implemented efficiently by grouping the DataFrame by each key (such as SSN, `name_dob`, phone, etc.), sorting within each group by date, and using a shift operation to access the previous date for subtraction. The rationale is that a short interval since a previous application with the same information might indicate rapid re-use of identity data, which can be a fraud signal. We create a separate `days_since_last` feature for each key type (e.g., `ssn_days_since_last`, `name_dob_days_since_last`, `phone_days_since_last`, etc.). Each of these quantifies the recency of activity for that identifier.

4.2 Recent Velocity Count Features

We also generate features that count how many times each entity has appeared in recent time windows prior to the current application—often referred to as *velocity* features. Specifically, for each key and for various look-back windows (0 days, 1 day, 3 days, 7 days, 14 days, and 30 days), we calculate the number of applications (including or up to the current one) that have the same key and occurred within that window ending on the current application's date:

- **0-day count (same-day count):** The number of other applications that occurred on the same day as the current one with the same key. For example, `ssn_count_0` for an application would be the count of applications with the same SSN on that exact date (excluding the current record itself, this would count other applications on that date).
- **1-day count:** The count of applications in the last 1 day (typically meaning the same day and the day before). `name_count_1` for instance would tell how many applications with the same full name occurred within the past 24 hours prior to this application.

- **3-day, 7-day, 14-day, 30-day counts:** Similarly, these features count occurrences within the last 3, 7, 14, or 30 days, respectively. For instance, `phone_count_7` gives the number of applications in the week leading up to (and including) the current application that share the same phone number.

These features are calculated by filtering the dataset by date range for each record and key. In practice, a rolling or expanding window method is used: as we iterate through applications in chronological order, we maintain counters of how many previous records with the same key fall within the relevant time window. This can be optimized with cumulative sums or by using date indexes to quickly slice the past X days of data for each key. Velocity features capture the idea of burstiness or recent frequency. A higher count in a short window (e.g., many applications with the same data in the past week) can signal suspicious behavior such as one person trying multiple times or sharing of identity information across applications. We generate such count features for all the important entity keys (SSN, full name, name+DOB, phone, address, etc.), each across the multiple time windows. For example, representative features include `ssn_count_3`, `name_dob_count_30`, `address_zip_count_7`, and so on.

4.3 Relative Velocity Ratios

In addition to raw counts, we compute relative velocity features, which are ratios comparing counts across different time windows. These ratios help normalize the raw counts and indicate acceleration or deceleration in activity. For instance, a feature like `count_0_by_3` (using a generic name here) would be defined as the count in the last 0 days divided by the count in the last 3 days for the same entity. Concretely, if an SSN has `ssn_count_0` = 2 (two other same-day applications) and `ssn_count_3` = 5 (five applications in the last 3 days), then `ssn_count_0_by_3` = 2/5 = 0.4. A higher ratio (close to 1) would mean most of the last 3 days' activities happened today (a spike of activity), whereas a lower ratio indicates the activity was more spread out. We create such ratio features for relevant window pairs, typically an immediate window to a slightly larger window. Examples include:

- `count_0_by_3`: Ratio of same-day count to three-day count.
- `count_3_by_7`: Ratio of three-day count to weekly count.
- `count_7_by_30`: Ratio of weekly count to monthly count.

These ratios are computed with care to avoid division by zero. In implementation, we ensure that if the denominator count is zero (meaning no occurrences in the longer window), the ratio is handled (for example, set to 0 if numerator is also zero, or flagged specially if numerator exists without denominator, though the latter case typically cannot happen if a shorter window count is nonzero the longer window count should also be nonzero). The relative velocity features help highlight surges in activity: for example, a high `name_dob_count_0_by_7` would signal that the person has submitted many applications today compared to the whole week.

4.4 Cross-Entity Unique Counts

Another group of features captures the diversity of cross-entity relationships, i.e., how many unique values of one identifier are associated with another. We generate features that quantify, for each record, the number of unique second-entity values seen per first-entity value (based on historical data up to that point). In simpler terms, for each pair of entity types (like SSN and address, name and phone, etc.), we ask:

- How many unique X's has this Y been associated with?

where X and Y are two different entity fields. For example:

- **Unique addresses per SSN:** For the SSN on the current application, how many distinct street addresses have we seen associated with that SSN across all applications? This becomes a feature, e.g., `ssn_unique_address_count`. A fraudster might use the same SSN with many different addresses, which would make this count high.
- **Unique SSNs per address:** Conversely, `address_unique_ssn_count` counts how many distinct SSNs have been used for the given address. A drop address (used for fraud) might have many people (many SSNs) linked to it.
- **Unique phones per name:** `name_unique_phone_count` would count how many different phone numbers have been used by applications with the same name.
- **Unique names per phone:** `phone_unique_name_count` counts how many different names shared this phone number.

And so on for all combinations deemed relevant. We implement this by iterating through pairs of keys using nested loops or a double `groupby`. For each entity type A and another entity type B, we do a groupby on A and aggregate the number of unique values of B in each group. That result is then brought back into the dataset: each record with a given A value receives the unique count of B associated with that A. In pandas, one approach is:

```
unique_counts = df.groupby(A)[B].nunique()
df['A_unique_B_count'] = df[A].map(unique_counts)
```

This assigns to each record the precomputed unique count for its A value. We repeat such calculations for each (A, B) pair. By doing so prior to model training (and ideally in a way that avoids future data leakage by using only past data up to the record's date when calculating the count), we get features that tell us, for example, if an identity element is reused across many others. These cross-entity features are powerful for fraud detection because they can unveil many-to-many relationships indicative of fraud rings or identity sharing. A legitimate user typically has one SSN used with maybe a couple of addresses over time; if we see one SSN with 10+ unique addresses, that stands out. Likewise, one address associated with dozens of SSNs is suspect. Representative features from this group include those mentioned above, as well as combinations like `ssn_unique_phone_count`, `name_dob_unique_device_count` (if device ID is a field), etc., covering all important pairs of identifiers in the data.

4.5 Day-of-Week Fraud Risk

Some exploratory analysis of the data may reveal that the fraud rate varies by the day of week an application is submitted (for instance, perhaps weekend applications have a higher or lower fraud incidence). To capture this pattern without overfitting to noise, we create a feature representing the fraud risk associated with the application's day-of-week, using a smoothed estimate. First, we compute the observed fraud rate for each day of week (Monday through Sunday) from the historical data. Let y_{dow} denote the fraud rate for a particular day-of-week and $\bar{y}$ the overall fraud rate across all days. Because some days might have fewer observations (e.g., maybe fewer applications on Sunday), we apply a smoothing formula to pull extreme rates toward the overall average if the sample size is low. The smoothing is done using a logistic function based on the count of observations for that day. Specifically, for each day-of-week, we calculate a smoothed risk $y_{\text{dow\_smooth}}$ as:

$$y_{\text{dow\_smooth}} = \bar{y} + \frac{y_{\text{dow}} - \bar{y}}{1 + \exp\left(-\frac{\text{num}-\text{mid}}{c}\right)}$$

, where num is the number of applications on that day-of-week (in the training data), n_{mid} is a chosen midpoint count at which the adjustment is half, and c is a scaling factor controlling the steepness of the logistic function. This equation ensures that if num is small (few samples for that day), the denominator is large and $y_{\text{dow\_smooth}}$ stays closer to $\bar{y}$ (the global rate). If num is large, the fraction approaches $(y_{\text{dow}} - \bar{y})$ and the day-specific rate is trusted more. We then create a feature in the dataset for each application's day-of-week risk: the application date is converted to a weekday (0–6 or Monday–Sunday), and we map that to the smoothed fraud rate $y_{\text{dow\_smooth}}$ computed above. This gives a numeric feature (essentially an encoded day-of-week effect). It provides the model a sense of whether that particular day is relatively riskier or safer, while avoiding the high variance that a raw per-day fraud rate might introduce.

5 Feature Selection and Finalization

After creating the multitude of features above, we perform a final selection and sanity-check step to ensure all features are suitable for modeling. One filter we apply is dropping any feature that has very low variability or too few unique values. For example, if a feature ended up with ≤ 10 unique values across the entire dataset, it might be too coarse or not informative (it could even be practically constant if those values are not meaningful categories). Such features are candidates for removal. This threshold-based dropping was implemented by checking each feature's number of distinct values (e.g., using `df[n].nunique()`) and removing those that fall below the threshold. The rationale is to avoid including features that may not generalize or that might indicate data quirks (for instance, if a velocity count for a rarely used key only ever took a few values, it might not be reliable). At this stage, we also remove or exclude the raw PII columns from the model input, since their information is now captured in the engineered features. Fields like actual name strings, full address text, or SSN are dropped to prevent the model from trying to interpret raw identity data (which would be meaningless to a model and could also risk privacy leakage). Instead, only the aggregated or encoded features (counts, flags, ratios, etc.) and perhaps some coarse demographical data like age remain. The final dataset is then composed of the original non-PII informative fields (e.g., product type, application channel, etc. if any) plus all the newly engineered features. Originally, suppose the dataset had M original columns (including ID and label); we have now added N new feature columns. In our case, for example, the original data might have had around 12 relevant fields, and we engineered on the order of 80–100 additional features. This brings the total number of feature columns to roughly $M + N$ (not counting the target variable). We ensure that each of these features is numeric or a well-encoded categorical, with no high-cardinality raw strings. The dataset is now ready for modeling. Each row (application) is represented by a rich feature vector capturing temporal behavior (how recently and how frequently associated data has been seen), relational links (how many unique connections exist between this application's identifiers and others), and contextual risk factors (like day-of-week risk and age). These features can be used in a machine learning model to predict the likelihood of fraud for new applications. The data pipeline is also fully reproducible, meaning the same steps can be applied to future data to generate the features in a live scoring environment.