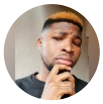
A Practical Guide to Feature Engineering in Python

Learn the underlying techniques and tools for effective Feature Engineering in Python



Rising Odegua Follow
Jan 8 · 16 min read

Source: Pixabay

Feature engineering is one of the most important skills needed in data science and machine learning. It has a major influence on the performance of machine learning models and even the quality of insights derived during exploratory data analysis (EDA).

In this article, we're going to learn some important techniques and tools that will help you properly extract, prepare, and engineer features from your dataset.

What you will learn in this article:

- What is feature engineering?
- How to handle missing values.
- How to handle categorical features.
- How to handle numerical/continuous features.
- Creating polynomial features.
- Normalization of features.
- Working with date/time features
- Working with latitudes and longitudes

Find the Jupyter notebook for this post here.

. . .

What is Feature Engineering?

Feature engineering is the process of using data's domain knowledge to create features that make machine learning algorithms work (Wikipedia). It's the act of extracting important features from raw data and transforming them into formats that are suitable for machine learning.

To perform feature engineering, a data scientist combines domain knowledge (knowledge about a specific field) with math and programming skills to transform or come up with new features that will help a machine learning model perform better.

Feature engineering is a practical area of machine learning and is one of the most important aspects of it. Below we highlight what a couple of industry experts have said about it:

> ...some machine learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features used.— Pedro Domingos, "A Few Useful Things to Know about Machine Learning"

. . .

> Coming up with features is difficult, time-consuming, requires expert knowledge. "Applied machine learning" is basically feature engineering.— Andrew Ng, Machine Learning and AI via Brain simulations

. . .

Preparing our Datasets

Now that we understand what feature engineering is, let's go straight into the practical aspect of this article. We'll use two datasets for this article. The first is the Loan Default Prediction dataset hosted on Zindi by Data Science Nigeria, and the second — also hosted on Zindi — is the Sendy Logistics dataset by Sendy.

You can find the descriptions of the dataset and the corresponding machine learning tasks in the links above. If you have cloned the **repo**, you'll have a folder of the datasets and the notebook used for this article and can follow along easily.

First, let's import some libraries and the datasets:

```
import pandas as pd
```

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')


#load loan datasets
loan_demographics = pd.read_csv('traindemographics.csv')
loan_prev = pd.read_csv('trainprevloans.csv')
loan_perf = pd.read_csv('trainperf.csv')
#load logistics dataset
sendy_data = pd.read_csv('sendy_logistics.csv')
```

feat1.py hosted with ❤️ by GitHub view raw

- We import *Pandas, NumPy, Seaborn, and Matplotlib* for basic data manipulation and visualization.
- We silence unnecessary warnings using the `filterwarnings` module, and finally we import the datasets.

We can see that the loan dataset has three tables. These tables are related to each other by the primary key (`customerid`).

Let's take a peek at our dataset and get a feel for what records are present.

```
loan_demographics.sample(3).T
```

	3269	2949	1780
customerid	8a858e725c3ae262015c44c29be41a95	8a858e845accb4f9015acce9b7b82583	8a858fcf5a192144015a1d262059287d
birthdate	1986-03-26 00:00:00.000000	1980-08-12 00:00:00.000000	1980-09-15 00:00:00.000000
bank_account_type	Savings	Savings	Savings
longitude_gps	3.48331	4.56953	7.42288
latitude_gps	6.63038	8.46734	9.06377
bank_name_clients	GT Bank	First Bank	EcoBank
bank_branch_clients	NaN	NaN	NaN
employment_status_clients	Permanent	NaN	Permanent
level_of_education_clients	NaN	NaN	NaN

```
loan_perf.sample(3).T
```

	1085	3765	1212
customerid	8a858f045bc9690c015bc9a82a3c19aa	8a858f265cfa1a64015d04610339125b	8a858fe35b8ba108015b8ba23e8e0143
systemloanid	301982538	301999941	301964785
loannumber	4	2	4
approveddate	2017-07-17 19:15:08.000000	2017-07-28 07:32:16.000000	2017-07-05 12:36:12.000000
creationdate	2017-07-17 18:15:03.000000	2017-07-28 06:32:10.000000	2017-07-05 11:36:07.000000
loanamount	20000	10000	20000
totaldue	24500	13000	24500
termdays	30	30	30
referredby	8a858fc55b2548dd015b28bb609670b2	8a858e935b496584015b496a6fde01f6	NaN
good_bad_flag	Good	Good	Bad

```
loan_prev.sample(3).T
```

	12584	3828	801
Unnamed: 0	12584	3828	801
customerid	8a858ee755a0c84a0155c428789463bc	8a858f4d5b809e68015b813974bc5538	8a858fa358d4a4d80158ee50ccb46401
systemloanid	301707454	301922810	301938851
loannumber	5	3	7
approveddate	2016-09-06 15:45:27.000000	2017-05-31 11:59:17.000000	2017-06-14 18:23:51.000000
creationdate	2016-09-06 14:45:20.000000	2017-05-31 10:59:11.000000	2017-06-14 17:22:45.000000
loanamount	20000	10000	30000
totaldue	21500	11500	34500
termdays	15	15	30
closeddate	2016-09-19 10:03:32.000000	2017-06-15 15:36:09.000000	2017-07-14 05:33:34.000000
referredby	NaN	8a858f725b49c0c0015b56d2ee6824d9	NaN
firstduedate	2016-09-21 00:00:00.000000	2017-06-15 00:00:00.000000	2017-07-14 00:00:00.000000
firstrepaiddate	2016-09-19 09:48:27.000000	2017-06-15 15:26:01.000000	2017-07-14 04:53:50.000000

```
sendy_data.sample(3).T
```

	18657	15483	13137
Order No	Order_No_14236	Order_No_21083	Order_No_11540

User Id	User_Id_3565	User_Id_496	User_Id_1049
Vehicle Type	Bike	Bike	Bike
Platform Type	3	3	3
Personal or Business	Business	Business	Personal
Placement - Day of Month	9	17	5
Placement - Weekday (Mo = 1)	6	4	2
Placement - Time	8:31:33 AM	3:45:26 PM	10:01:34 AM
Confirmation - Day of Month	9	17	5
Confirmation - Weekday (Mo = 1)	6	4	2
Confirmation - Time	8:31:46 AM	3:47:48 PM	10:02:36 AM
Arrival at Pickup - Day of Month	9	17	5
Arrival at Pickup - Weekday (Mo = 1)	6	4	2
Arrival at Pickup - Time	8:40:51 AM	3:58:54 PM	10:12:05 AM
Pickup - Day of Month	9	17	5
Pickup - Weekday (Mo = 1)	6	4	2
Pickup - Time	8:42:23 AM	4:03:13 PM	10:29:52 AM
Arrival at Destination - Day of Month	9	17	5
Arrival at Destination - Weekday (Mo = 1)	6	4	2
Arrival at Destination - Time	9:29:30 AM	4:29:31 PM	10:47:18 AM
Distance (KM)	26	9	8
Temperature	20.2	24	21.1
Precipitation in millimeters	NaN	NaN	2.1
Pickup Lat	-1.28447	-1.26269	-1.30033
Pickup Long	36.7866	36.7827	36.7975
Destination Lat	-1.40102	-1.29802	-1.25869
Destination Long	36.9458	36.7886	36.8053
Rider Id	Rider_Id_326	Rider_Id_279	Rider_Id_267
Time from Pickup to Arrival	2827	1578	1046

We now have an overview of our datasets. From this, we can see that the loan dataset contains mainly three types of features (Numerical, Categorical and Date features), while the logistic dataset contains four types of features (Numerical, Categorical, Date, and Geo features).

With this in the back of our mind, let's do some feature engineering.

> ***Note:*** *The techniques and things we do here do not follow a fixed order and may not always apply to your dataset. The intention is to show what is possible, leaving you to choose methods at your discretion or as your use case requires.*

. . .

. . .

How to handle missing values

Missing values are values that are not recorded during data collection. They are mostly not provided, left out due to errors, or too difficult to measure. Missing values may be very important to models, and as such there exist numerous ways and techniques to handle them. Let's go over some of these techniques.

The choice of technique is dependent on the type of features you have. For categorical features, you can do things like:

- **Mode filling:** Fill missing values with the most popular/frequent/modal class.
- **Temporal filling (forward or backward fill)**: Fill missing values with the preceding value (top-down) or with the succeeding value (bottom-up).
- **Encoding and fill:** In this method, you can encode the values using different strategies, and then fill with either the mean, mode, or the median.

In the loan demographic dataset, we have three categorical features (`bank_branch_clients` , `employment_status_clients` , `level_of_education_clients`) with missing values. Let's try mode filling using the `employment_status_clients` feature:

```
#check for missing values
loan_demographics.isna().sum()
```

```
customerid                      0
birthdate                       0
bank_account_type               0
longitude_gps                   0
latitude_gps                    0
bank_name_clients               0
bank_branch_clients          4295
employment_status_clients     648
level_of_education_clients   3759
dtype: int64
```

```
loan_demographics['employment_status_clients'].value_counts()
```

```
Permanent        3146
Self-Employed     348
Student           142
Unemployed         57
Retired             4
Contract            1
Name: employment_status_clients, dtype: int64
```

From the output of the `value_counts` above, we can see the classes present in the `employment_status_clients` feature and their corresponding frequencies. We see that the `Permanent` class is more popular. We can use this value to fill all missing values in the `employment_status_clients` feature, as shown below.

```
nt)
yment_status_clients'] = loan_demographics['employment_status_clients'].fillna(value='Permanen
```

feat2.py hosted with ❤️ by GitHub view raw

For numerical features, we can also do things like:

- Filling with mean, mode, or median.
- Temporal filling (backward or forward filling).
- Use machine learning models: Train a machine learning model to learn the most appropriate fill values.

To demonstrate the process of filling numerical values, we will use the Sendy logistics dataset, as it has two numerical features (`Temperature` and `Precipitation` in millimeters) that contain missing values.

```
mean_df = round(sendy_data['Temperature'].mean())
mode_df = round(sendy_data['Temperature'].mode()[0])
median_df = round(sendy_data['Temperature'].median())

#Fill with mean
print("Filling with mean value of {}".format(mean_df))
sendy_data['Temperature'] = sendy_data['Temperature'].fillna(mean_df)

#Fill with mode
print("Filling with modal value of {}".format(mode_df))
sendy_data['Temperature'] = sendy_data['Temperature'].fillna(mode_df)

#Fill with median
print("Filling with median value of {}".format(median_df))
sendy_data['Temperature'] = sendy_data['Temperature'].fillna(median_df)
```

feat3.py hosted with ❤️ by GitHub view raw

```
Filling with mean value of 23
Filling with modal value of 25.0
Filling with median value of 24
```

- First, we calculate the value of the chosen fill method, then we use the Pandas `fillna` function to automatically find and replace missing values with the calculated value.

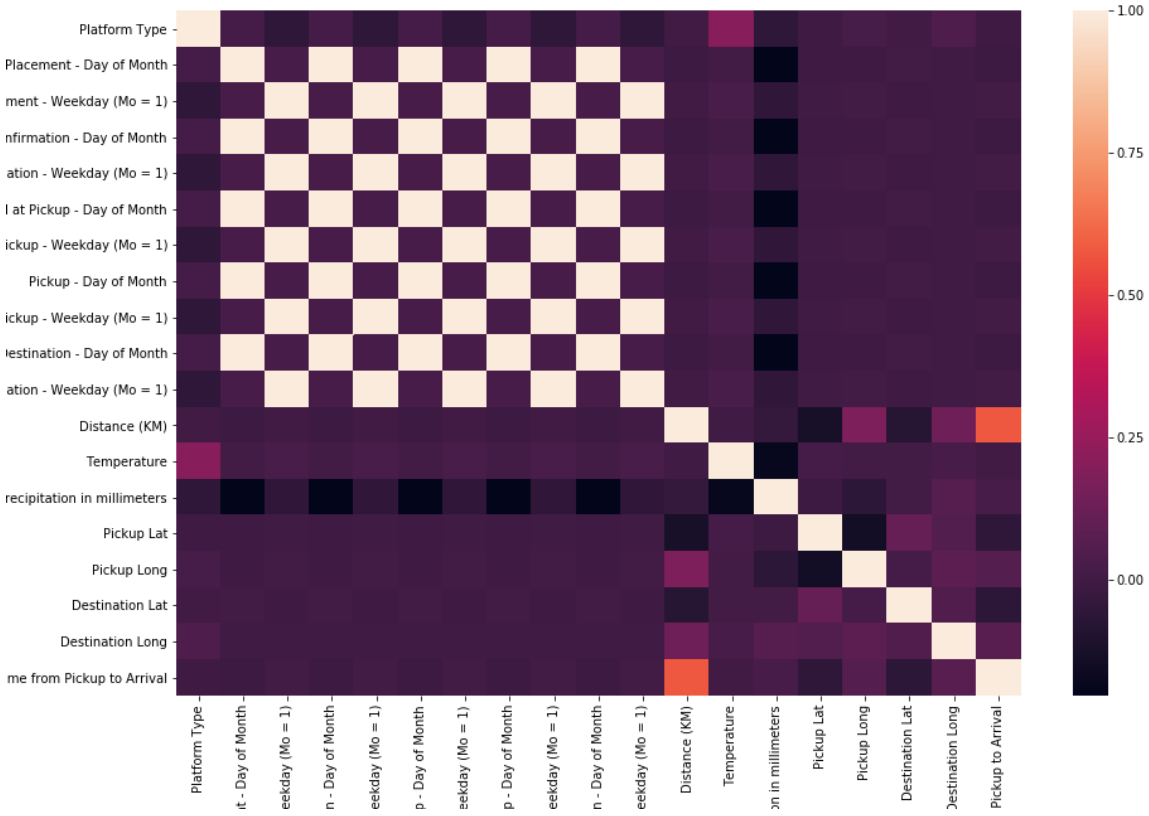
> ***Note:*** *You should only use one method (mean, mode, or median fill) at any given time. Also, the calculated mode in Pandas always returns the modal value and its index. This means we can have more than one returned modal value. We choose the first one by indexing with [0].*

Use modeling to fill missing values

To demonstrate filling with modeling, we'll use the feature `Precipitation in millimeters`) in the Sendy dataset. But first, we need to select features that correlate with it. That is, feature(s) that can help predict `Precipitation in millimeters` .

The Seaborn heatmap plot can help us decide. We demonstrate this below.

```
#Plot heatmap of feature correlation
plt.figure(figsize = (15,10))
sns.heatmap(sendy_data.corr())
```



From the heatmap plot above, we can see that the majority of the features do not really correlate with `Precipitation in millimeters`. We can make use of the last three features (`Destination Lat`, `Destination Long` and `Time from Pickup to Arrival`), as these show little correlation. Let's demonstrate this below:

```
near_model import LinearRegression

ession()

ecipitation in millimeters', 'Destination Lat', 'Destination Long', 'Time from Pickup to Arriv
_data[to_train]
```

```
with missing values and no missing values as test and train set respectively.
df[temp_df['Precipitation in millimeters'].notnull()].drop(columns='Precipitation in millimete
df[temp_df['Precipitation in millimeters'].notnull()]['Precipitation in millimeters']
f[temp_df['Precipitation in millimeters'].isnull()].drop(columns='Precipitation in millimeters

inear model to the dataset
 y_train)
ct(x_test)

ues
pred, 5))

g
cipitation in millimeters'][sendy_data['Precipitation in millimeters'].isnull()] = pred
```

feat4.py hosted with ❤ by GitHub view raw

- First, we import and use a simple linear regression model.
- We save the correlated features as observed from the heatmap to a list called `to_train`.
- We create a train-test dataset from these features, where the train dataset contains no missing values and the test dataset contains the missing values we want to fill.
- Finally, we fit the model, make predictions on the test set, and use the predictions to fill in the missing values.

Alternatively, you can use the `IterativeImputer` function in the `sklearn.experimental` module of the sklearn library to automatically fill missing values. We demonstrate this below:

```
#get the index of missing so we can some of the values used for filling
missing_indx = list(sendy_data['Temperature'][sendy_data['Temperature'].isna()].index)
```

feat5.py hosted with ❤ by GitHub view raw

First, we save the index of the missing values in a variable so we can observe the fill values calculated by the `IterativeImputer` function.

```
# explicitly require this experimental feature
```

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.ensemble import RandomForestRegressor

# Run the imputer with a simple Random Forest estimator
imp = IterativeImputer(RandomForestRegressor(n_estimators=5), max_iter=5, random_state=1
to_train = ['Temperature', 'Destination Lat', 'Destination Long','Time from Pickup to Ar

#perform filling
sendy_data[to_train] = pd.DataFrame(imp.fit_transform(sendy_data[to_train]), columns=to_

#display some of the filled values
sendy_data['Temperature'][missing_indx].head(10)
```

feat6.py hosted with ❤️ by GitHub view raw

```
: 2     23.70
  8     26.06
  14    21.66
  15    20.42
  16    21.46
  32    23.84
  42    22.32
  46    27.18
  49    22.34
  53    21.36
  Name: Temperature, dtype: float64
```

- First, we explicitly require and enable the `IterativeImputer` function. This must be done before you can use it.
- Next, we import a RandomForest model and create an imputer object by passing the RandomForest model.
- Next, we specify the list of correlated features as a Python list.
- Finally, we fit transform and convert the result to a Pandas dataframe.

. . .

The future of machine learning is on the edge.

. . .

How to handle categorical features

Categorical features are features that can take on values from a limited set. For example, the relative hotness of a place/thing (hot, hotter, hottest) or star ratings for an application (1,2,3,4,5). In regards to our dataset, features like `level_of_education_clients` in the `loan_demographics` dataset is a categorical feature containing classes like `Secondary`, `Graduate`, `Post-Graduate`, and `Primary`.

Machine learning models cannot work with categorical features the way they are. These features must be converted to numerical forms before they can be used. The process of converting categorical features to numerical form is called *encoding*.

There are numerous types of encoding, and the choice of which kind to use is mostly dependent on the categorical type. So first, let's understand the different categorical feature types.

Types of categorical features

1. **Ordinal Categorical Features:** Ordinal categorical features have a natural ordered category. That is, one class is higher than another. For example, star ratings (1,2,3,4,5), where class 5 is a higher rating than 4/3/2/1.

2. **Non-Ordinal Categorical Features:** This type of feature has no specific order. That is, no class is higher than the other. One example would be type of food (rice, pasta, macaroni, spaghetti). Rice is not in some weird way higher than pasta/macaroni/spaghetti, right?

What encoding scheme to use and when to use it

Manual Encoding of Ordinal Feature: If the classes in a categorical feature are ordinal, and the unique values are small, then you can manually assign labels that have some form of ordering. Let's demonstrate this below using the `level_of_education_clients` feature in the `loan demographic` dataset. This feature has some form of ordinality as the class `Post-Graduate` is higher than `Graduate` / `Secondary` / `Primary` classes.

```
loan_demographics['level_of_education_clients'].unique()
```

```
array([nan, 'Secondary', 'Graduate', 'Post-Graduate', 'Primary'],
      dtype=object)
```

```
#use a simple map function
map_education = {"Primary" : 1, "Secondary": 2, "Graduate": 3, "Post-Graduate": 4}

loan_demographics['level_of_education_clients'] = loan_demographics['level_of_education_c
loan_demographics['level_of_education_clients'].value_counts()
```

feat7.py hosted with ❤ by GitHub view raw

```
: 3.0    420
  2.0     89
  4.0     68
  1.0     10
  Name: level_of_education_clients, dtype: int64
```

- First, we create a dictionary mapping classes to their labels. Here, the highest class (`Post-Graduate`) is assigned the highest number.
- Next, we use the Pandas `map` function to find and replace every class with its corresponding label.

To perform automated encoding, we'll use an efficient library called ***categorical_encoders***. This library offers numerous encoding schemes out of the box and has first-hand support for Pandas dataframes.

To install the library, you can use pip as follows:

```
pip install category_encoders

          or

conda install -c conda-forge category_encoders
```

Label Encoding: If you have a large number of classes in a categorical feature, you can use label encoding. Label encoding assigns a unique label (integer number) to a specific class. We demonstrate this using two features (`bank_name_clients` and `bank_branch_clients`) with large numbers of unique classes, 18 and 45 respectively.

```
#Check the number of unique classes
cat_cols = loan_demographics.select_dtypes(include='object').columns
for col in cat_cols:
    print("Number of classes in {}".format(col))
    print(loan_demographics[col].nunique())
    print('--------------------------')
```

feat8.py hosted with ❤️ by GitHub view raw

```
Number of classes in customerid
4334
--------------------------
Number of classes in birthdate
3297
--------------------------
Number of classes in bank_account_type
3
--------------------------
Number of classes in bank_name_clients
18
--------------------------
Number of classes in bank_branch_clients
45
--------------------------
Number of classes in employment_status_clients
6
--------------------------
```

The `OrdinalEncoder` function in the categorical_encoders library can be used to label encode, as shown below:

```
import category_encoders as ce

#Label encoding
```

```
cat_cols = ['bank_name_clients', 'bank_branch_clients']
encoder = ce.OrdinalEncoder(cols=cat_cols)
loan_demographics = encoder.fit_transform(loan_demographics)
```

feat9.py hosted with ❤ by GitHub view raw

	0	1	2	3	4
customerid	8a858e135cb22031015cbafc76964ebd	8a858e275c7ea5ec015c82482d7c3996	8a858e5b5bd99460015bdc95cd485634	8a858efd5ca70688015cabd1f1e94b55	8a858e785acd3412015acd48f4920d04
birthdate	1973-10-10 00:00:00.000000	1986-01-21 00:00:00.000000	1987-04-01 00:00:00.000000	1991-07-19 00:00:00.000000	1982-11-22 00:00:00.000000
bank_account_type	Savings	Savings	Savings	Savings	Savings
longitude_gps	3.31922	3.3256	5.7461	3.36285	8.45533
latitude_gps	6.5286	7.1194	5.56317	6.64249	11.9714
bank_name_clients	1	2	3	1	1
bank_branch_clients	1	1	1	1	1
employment_status_clients	NaN	Permanent	NaN	Permanent	Permanent
level_of_education_clients	NaN	NaN	NaN	NaN	NaN

- First, we save the categorical columns we want to encode into a list.
- Next, we create the encoder object.
- Finally, we fit-transform the dataset.

One-Hot Encoding: One-hot encoding uses binary values to represent classes. It creates a feature per category, and can quickly become inefficient as the number of classes in the categorical feature increases. We demonstrate how to use this below:

```
cats = ['bank_account_type','level_of_education_clients']
one_hot_enc = ce.OneHotEncoder(cols=cats)
loan_demographics = one_hot_enc.fit_transform(loan_demographics)
loan_demographics.head().T
```

feat10.py hosted with ❤ by GitHub view raw

	0	1	2	3	4
customerid	8a858e135cb22031015cbafc76964ebd	8a858e275c7ea5ec015c82482d7c3996	8a858e5b5bd99460015bdc95cd485634	8a858efd5ca70688015cabd1f1e94b55	8a858e785acd3412015acd48f4920d04
birthdate	1973-10-10 00:00:00.000000	1986-01-21 00:00:00.000000	1987-04-01 00:00:00.000000	1991-07-19 00:00:00.000000	1982-11-22 00:00:00.000000
bank_account_type_1	1	1	1	1	1
bank_account_type_2	0	0	0	0	0
bank_account_type_3	0	0	0	0	0
longitude_gps	3.31922	3.3256	5.7461	3.36285	8.45533
latitude_gps	6.5286	7.1194	5.56317	6.64249	11.9714
bank_name_clients	1	2	3	1	1
bank_branch_clients	1	1	1	1	1
employment_status_clients	NaN	Permanent	NaN	Permanent	Permanent
level_of_education_clients_1	1	1	1	1	1
level_of_education_clients_2	0	0	0	0	0
level_of_education_clients_3	0	0	0	0	0
level_of_education_clients_4	0	0	0	0	0
level_of_education_clients_5	0	0	0	0	0

Hash Encoding: Hash encoding or feature hashing is a fast and space-efficient way of encoding features. It's very efficient for categorical features with large numbers of classes. A hash encoder works by applying a hash function to the features. We demonstrate how to use this below.

```
cat_cols = ['bank_name_clients', 'bank_branch_clients']
hash_enc = ce.HashingEncoder(cols=cat_cols, n_components=10)
loan_demographics = hash_enc.fit_transform(loan_demographics)
loan_demographics.head()
```

feat11.py hosted with ❤️ by GitHub view raw

	0	1	2	3	4
col_0	0	0	0	0	0
col_1	0	0	0	0	0
col_2	1	1	1	1	1
col_3	0	0	0	0	0
col_4	0	0	0	0	0
col_5	0	0	0	0	0
col_6	0	0	1	0	0
col_7	0	0	0	0	0
col_8	1	1	0	1	1
col_9	0	0	0	0	0
customerid	8a858e135cb22031015cbafc76964ebd	8a858e275c7ea5ec015c82482d7c3996	8a858e5b5bd99460015bdc95cd485634	8a858efd5ca70688015cabd1f1e94b55	8a858e785acd3412015acd48f4920d04
birthdate	1973-10-10 00:00:00.000000	1986-01-21 00:00:00.000000	1987-04-01 00:00:00.000000	1991-07-19 00:00:00.000000	1982-11-22 00:00:00.000000
bank_account_type	Savings	Savings	Savings	Savings	Savings
longitude_gps	3.31922	3.3256	5.7461	3.36285	8.45533
latitude_gps	6.5286	7.1194	5.56317	6.64249	11.9714
employment_status_clients	NaN	Permanent	NaN	Permanent	Permanent
level_of_education_clients	NaN	NaN	NaN	NaN	NaN

- First, we specify the features we want to hash encode.
- Next, we create a hash encoder object and specify the length of the hash vector to be used.
- Finally, we fit-transform the dataset.

Target Encoding: In target encoding, we calculate the average of the target value by a specific category and replace that categorical feature with the result. Target encoding helps preserve useful properties of the feature and can sometimes help improve classification models—however, it can sometimes lead to severe overfitting.

To demonstrate target encoding, we'll use the loan performance dataset (`loan_perf`). The target of interest in this dataset is the `good_bad_flag` feature, and the task is to

predict if a customer will repay a loan or not. The feature `good_bad_flag` is represented as a categorical feature, so we first convert it to numerical form, as shown below:

```
map_target = {"Good": 0, "Bad": 1}
loan_perf['good_bad_flag'] = loan_perf['good_bad_flag'].map(map_target)
```

feat12.py hosted with ❤ by GitHub view raw

Next, we'll target encode the feature `loannumber`. This feature is numeric in nature, but can be treated as a categorical feature because it has a limited number of classes.

```
target_enc = ce.TargetEncoder(cols=['loannumber'])
loan_perf = target_enc.fit_transform(X=loan_perf, y=loan_perf['good_bad_flag'])
loan_perf.head().T
```

feat13.py hosted with ❤ by GitHub view raw

	0	1	2	3	4
customerid	8a2a81a74ce8c05d014cfb32a0da1049	8a85886e54beabf90154c0a29ae757c0	8a8588f35438fe12015444567666018e	8a85890754145ace015429211b513e16	8a858970548359cc0154883481981866
systemloanid	301994762	301965204	301966580	301999343	301962360
loannumber	0.201835	0.249103	0.162698	0.267164	0.166667
approveddate	2017-07-25 08:22:56.000000	2017-07-05 17:04:41.000000	2017-07-06 14:52:57.000000	2017-07-27 19:00:41.000000	2017-07-03 23:42:45.000000
creationdate	2017-07-25 07:22:47.000000	2017-07-05 16:04:18.000000	2017-07-06 13:52:51.000000	2017-07-27 18:00:35.000000	2017-07-03 22:42:39.000000
loanamount	30000	15000	20000	10000	40000
totaldue	34500	17250	22250	11500	44000
termdays	30	30	15	15	30
referredby	NaN	NaN	NaN	NaN	NaN
good_bad_flag	0	0	0	0	0

- First, we create a target encoder object and pass the column(s) we want to encode.
- Next, we fit-transform the dataset by passing both the features and the target of interest.

There are many more encoding schemes (binary encoders, count Encoders, leave one out encoders, CatBoost encoders etc.) that you can try out for your use cases. A good place to learn about them is in the official documentation of the categorical_encoder library.

How to handle numerical/continuous features

Numerical/Continuous features are the most common type of features found in datasets. They can represent values from a given range. For example, the price of a product, the

temperature of a place, or coordinates on a map.

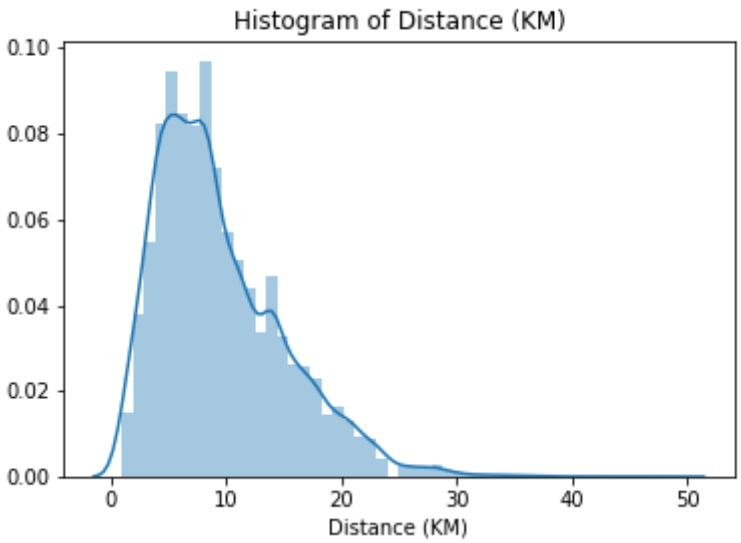
Feature engineering on numerical data mostly depends on domain knowledge. Some of the things we can do here are:

Log Transformation: Log transformation helps to center (or in statistical terms normally distribute) data. This strategy can help most machine learning models perform better.

> ***Note**: If you log transform the target feature, always take the exponent at the end of the analysis when interpreting the result.*

Log transformations are mostly performed on skewed features. Features can either be left or right skewed. Skewness can be easily checked by visualization. To demonstrate log transformation, we will use the `Distance (KM)` feature in the Sendy dataset, as this feature is right skewed.

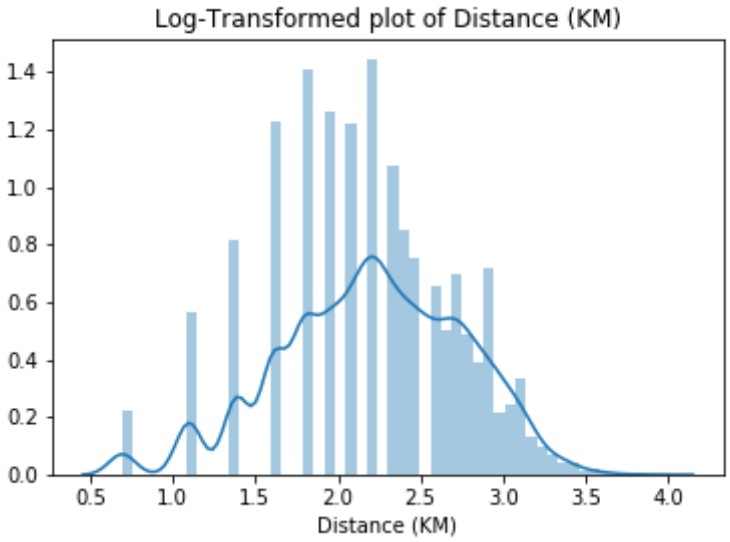
```
sns.distplot(sendy_data['Distance (KM)'])
plt.title("Histogram of Distance (KM)")
plt.show()
```




```
sendy_data['Distance (KM)'] =  np.log1p(sendy_data['Distance (KM)'])
sns.distplot(sendy_data['Distance (KM)'])
plt.title("Log-Transformed plot of Distance (KM)")
```

```
plt.show()
```

feat14.py hosted with ❤️ by GitHub view raw



We log transform by taking the logarithm of all the instances. We use the efficient NumPy implementation, which adds 1 to every value before taking the logarithm. This helps us to avoid taking log of zero.

Using Domain Knowledge: If you have domain expertise or have someone with domain expertise on your team, you can come up with new features that can greatly help your machine learning models. We demonstrate this by creating some new features from the loan dataset, as shown below.

1. **Interest elapsed:** Interest elapsed is a feature we create from the difference between the `totaldue` and the `loanamount`.

```
loan_prev['interest_elapsed'] = loan_prev['totaldue'] -
loan_prev['loanamount']
```

2. **Loan count:** We can calculate the total number of loans collected by a customer by aggregating loan numbers.

```
#Groupby customer id and calculate their total loans taken
loannumber_count = loan_prev.groupby(by='customerid').agg(['count'])[['loannumber']].rese
```

```
#merge back to dataset on customer_id
loan_prev = loan_prev.merge(right=loannumber_count, how='left', on='customerid')
loan_prev.head()
```

feat15.py hosted with ❤ by GitHub view raw

	0	1	2	3	4
Unnamed: 0	0	1	2	3	4
customerid	8a2a81a74ce8c05d014cfb32a0da1049	8a2a81a74ce8c05d014cfb32a0da1049	8a2a81a74ce8c05d014cfb32a0da1049	8a8588f35438fe12015444567666018e	8a85890754145ace015429211b513e16
systemloanid	301682320	301883808	301831714	301861541	301941754
loannumber	2	9	8	5	2
approveddate	2016-08-15 18:22:40.000000	2017-04-28 18:39:07.000000	2017-03-05 10:56:25.000000	2017-04-09 18:25:55.000000	2017-06-17 09:29:57.000000
creationdate	2016-08-15 17:22:32.000000	2017-04-28 17:38:53.000000	2017-03-05 09:56:19.000000	2017-04-09 17:25:42.000000	2017-06-17 08:29:50.000000
loanamount	10000	10000	20000	10000	10000
totaldue	13000	13000	23800	11500	11500
termdays	30	30	30	15	15
closeddate	2016-09-01 16:06:48.000000	2017-05-28 14:44:49.000000	2017-04-26 22:18:56.000000	2017-04-24 01:35:52.000000	2017-07-14 21:18:43.000000
referredby	NaN	NaN	NaN	NaN	NaN
firstduedate	2016-09-14 00:00:00.000000	2017-05-30 00:00:00.000000	2017-04-04 00:00:00.000000	2017-04-24 00:00:00.000000	2017-07-03 00:00:00.000000
firstrepaiddate	2016-09-01 15:51:43.000000	2017-05-26 00:00:00.000000	2017-04-26 22:03:47.000000	2017-04-24 00:48:43.000000	2017-07-14 21:08:35.000000
(loannumber, count)	11	11	11	6	2

3. **Speed:** From Physics, we know that speed is *Distance per unit of Time*— therefore, we can create a new feature (`Speed`) in the Sendy dataset from the features `Distance (KM)` and `Time from Pickup to Arrival`.

```
#create feature speed in sendy dataset
sendy_data['speed'] = sendy_data['Distance (KM)'] / sendy_data['Time from Pickup to Arriv
sendy_data.head().T
```

feat16.py hosted with ❤ by GitHub view raw

Distance (KM)	4	16	3	9	9
Temperature	20.4	26.4	NaN	19.2	15.4
Precipitation in millimeters	NaN	NaN	NaN	NaN	NaN
Pickup Lat	-1.31775	-1.35145	-1.30828	-1.2813	-1.2666
Pickup Long	36.8304	36.8993	36.8434	36.8324	36.7921
Destination Lat	-1.30041	-1.295	-1.30092	-1.25715	-1.29504
Destination Long	36.8297	36.8144	36.8282	36.7951	36.8098
Rider Id	Rider_Id_432	Rider_Id_856	Rider_Id_155	Rider_Id_855	Rider_Id_770
Time from Pickup to Arrival	745	1993	455	1341	1214
speed	0.00536913	0.0080281	0.00659341	0.00671141	0.00741351

Polynomial (Cross) Features

Polynomial features create interactions among features. They help to capture relationships among independent variables and can help decrease the bias of a machine learning model, as long as it's not contributing to massive overfitting.

We can create polynomial/cross features manually by simply adding, multiplying, or dividing features with each other. In this article, we're going to use the Polynomial Feature module present in the sklearn library.

We will create polynomial features from the `loannumber`, `totaldue`, and `termdays` features in the loan previous dataset, as shown below.

```
#Use Sklearn Polynomial Features
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures()
to_cross = ['loannumber', 'totaldue', 'termdays']
crossed_feats = poly.fit_transform(loan_prev[to_cross].values)

#Convert to Pandas DataFrame and merge to original dataset
crossed_feats = pd.DataFrame(crossed_feats)
loan_prev = pd.concat([loan_prev, crossed_feats], axis=1)

loan_prev.head().T
```

feat17.py hosted with ❤️ by GitHub view raw

	0	1	2	3	4
Unnamed: 0	0	1	2	3	4
customerid	8a2a81a74ce8c05d014cfb32a0da1049	8a2a81a74ce8c05d014cfb32a0da1049	8a2a81a74ce8c05d014cfb32a0da1049	8a8588f35438fe12015444567666018e	8a85890754145ace015429211b513e16
systemloanid	301682320	301883808	301831714	301861541	301941754
loannumber	2	9	8	5	2
approveddate	2016-08-15 18:22:40.000000	2017-04-28 18:39:07.000000	2017-03-05 10:56:25.000000	2017-04-09 18:25:55.000000	2017-06-17 09:29:57.000000
creationdate	2016-08-15 17:22:32.000000	2017-04-28 17:38:53.000000	2017-03-05 09:56:19.000000	2017-04-09 17:25:42.000000	2017-06-17 08:29:50.000000
loanamount	10000	10000	20000	10000	10000
totaldue	13000	13000	23800	11500	11500
termdays	30	30	30	15	15
closeddate	2016-09-01 16:06:48.000000	2017-05-28 14:44:49.000000	2017-04-26 22:18:56.000000	2017-04-24 01:35:52.000000	2017-07-14 21:18:43.000000
referredby	NaN	NaN	NaN	NaN	NaN
firstduedate	2016-09-14 00:00:00.000000	2017-05-30 00:00:00.000000	2017-04-04 00:00:00.000000	2017-04-24 00:00:00.000000	2017-07-03 00:00:00.000000
firstrepaiddate	2016-09-01 15:51:43.000000	2017-05-26 00:00:00.000000	2017-04-26 22:03:47.000000	2017-04-24 00:48:43.000000	2017-07-14 21:08:35.000000
(loannumber, count)	11	11	11	6	2
0	1	1	1	1	1
1	2	9	8	5	2
2	13000	13000	23800	11500	11500
3	30	30	30	15	15
4	4	81	64	25	4
5	26000	117000	190400	57500	23000
6	60	270	240	75	30
7	1.69e+08	1.69e+08	5.6644e+08	1.3225e+08	1.3225e+08
8	390000	390000	714000	172500	172500

9	900	900	900	225	225

- First, we import the `PolynomialFeatures` function from sklearn
- Next, we create an object from it. Here, we can specify the degree of interaction (defaults to 2).
- Next, we specify the features we want to cross (defaults to all).
- Next, we perform crossing using the `fit-transform` method.
- `PolynomialFeatures` returns a NumPy array object, so we convert this to a Pandas dataframe and then merge with the original dataset (`loanprev`).

Normalization of Features

Normalization helps change the values of numeric features to a common scale, without distorting differences in the range of values or losing information. Normalization is very important for distance-based models like KNNs, and it also helps speed up training in neural networks.

Some normalization functions available in sklearn include:

1. `StandardScaler` : Standardize features by subtracting the mean and scaling to unit variance.
2. `RobustScaler` : Scale features using statistics that are robust to outliers.
3. `MinMaxScaler` : Normalize features by scaling each feature to a specified range (range depends on you!).

> ***Note: You should never fit your scaler to the test/validation set.*** *This can cause leakages. Also, scalers in sklearn are not robust to missing values, which means you should always fill missing values before attempting to use the scalers.*

```
from sklearn.preprocessing import StandardScaler

feats = ['loannumber', 'totaldue', 'termdays']
sc = StandardScaler()
sc_data = sc.fit_transform(loan_prev[feats])
```

```
6  sc_data
```

feat18.py hosted with ❤️ by GitHub view raw

```
array([[-0.67377132, -0.62877649,  0.30213166],
       [ 1.48047238, -0.62877649,  0.30213166],
       [ 1.17272328,  0.40432506,  0.30213166],
       ...,
       [-0.05827312, -0.62877649,  0.30213166],
       [-0.98152042, -0.62877649,  0.30213166],
       [-0.67377132, -0.62877649,  0.30213166]])
```

```
array([[-0.25      ,  0.        ,  0.        ],
       [ 1.5       ,  0.        ,  0.        ],
       [ 1.25      ,  0.83076923,  0.        ],
       ...,
       [ 0.25      ,  0.        ,  0.        ],
       [-0.5       ,  0.        ,  0.        ],
       [-0.25      ,  0.        ,  0.        ]])
```

```
array([[0.08      , 0.29543697, 0.4       ],
       [0.64      , 0.29543697, 0.4       ],
       [0.56      , 0.6295437 , 0.4       ],
       ...,
       [0.24      , 0.29543697, 0.4       ],
       [0.        , 0.29543697, 0.4       ],
       [0.08      , 0.29543697, 0.4       ]])
```

Working with date features

Date features are a popular type of feature present in many datasets. These features are temporal in nature and require specific feature extraction techniques. There are numerous things we can do with temporal features, some of which we briefly demonstrate below.

Time Elapsed: Time elapsed is the difference in time between two dates. We demonstrate this below by calculating the seconds elapsed between the `approveddate` and `creationdate` features in the loan performance dataset.

Extract Date Features: We demonstrate below how to extract features like days, weeks, hours, seconds, etc:

> *You can check out other properties you can extract from a Pandas date feature **here**.*

Period of the day: We can extract the period of the day (morning, afternoon, evenings) from a date feature by manually mapping the hours of a day to the period. We demonstrate this below.

> ***Note:** Numerous time functions, including the ones used here, have already been implemented in the timeseries module of the datasist library. You can easily call and use these functions in just one line of code. Learn more about the datasist library **here**.*

Working with latitudes and longitudes

Geo-based features are a class of features present in range of datasets. These features contain records about the geographical location of a place/point in space. Features like `Longitudes`, `Latitudes`, and `Address` are geo-features that need to be engineered.

There are numerous things that we can do with latitude and longitude features. We can use libraries like Geojson or Geopy to convert these numerical values to physical addresses on a map.

But these methods are slow and don't really scale to a large number of features. In this article, we'll bypass these methods in favor of demonstrating simpler and quicker ways to extract features from longitudes and latitudes.

The techniques shown below are culled from this amazing kernel on Kaggle by Beluga.

1. **Manhattan distance:** The Manhattan distance is the sum of the horizontal and vertical distance between two points. Let's demonstrate this below using the Sendy dataset:

- First, we write a function to calculate the Manhattan distance. This is implemented in base NumPy.
- Next, we used the function created to calculate the Manhattan distance between `Pickup` and `Destination`.

2. **Haversine distance:** The Haversine distance is the great-circle distance between two points on a sphere, given their longitudes and latitudes. It's very important in navigation.

- First, we write a function to calculate the Harversine distance. This is also implemented in base NumPy.
- Next, we use the function to calculate the Harversine distance between `Pickup` and `Destination`.

3. **Bearing**: The bearing is the compass direction used to travel from a starting point, and must be within the range 0 to 360.

- First, we create the `bearing` function in NumPy.
- Next, we use the function to calculate the bearing between `Pickup` and `Destination`.

4. **Center point:** We can calculate the mid-point between two points from their latitudes and longitudes. This can be done by adding the points and dividing the result by 2.

- First, we calculate the center latitude by adding `Pickup Latitude` with `Destination Latitude`, and then dividing the result by 2. We do the same for the `Pickup Longitude` and `Destination Longitude`.

> ***Note:*** *Geo functions like Manhattan, Harversine, and bearing distances are already implemented in the datasist library. You can easily call and use them in just one line of code. Learn more about the datasist library* ***here****.*

And we draw the curtain here...

Feature engineering is essential and is often the difference between a good machine learning model and the best machine learning model.

In this post, we have learned about some of the techniques and tools for performing feature engineering. We started by defining feature engineering, then looked at ways for handling missing values.

Next, we explored some encoding techniques for categorical features, and then, various ways for handling numerical features, where we specifically talked about log

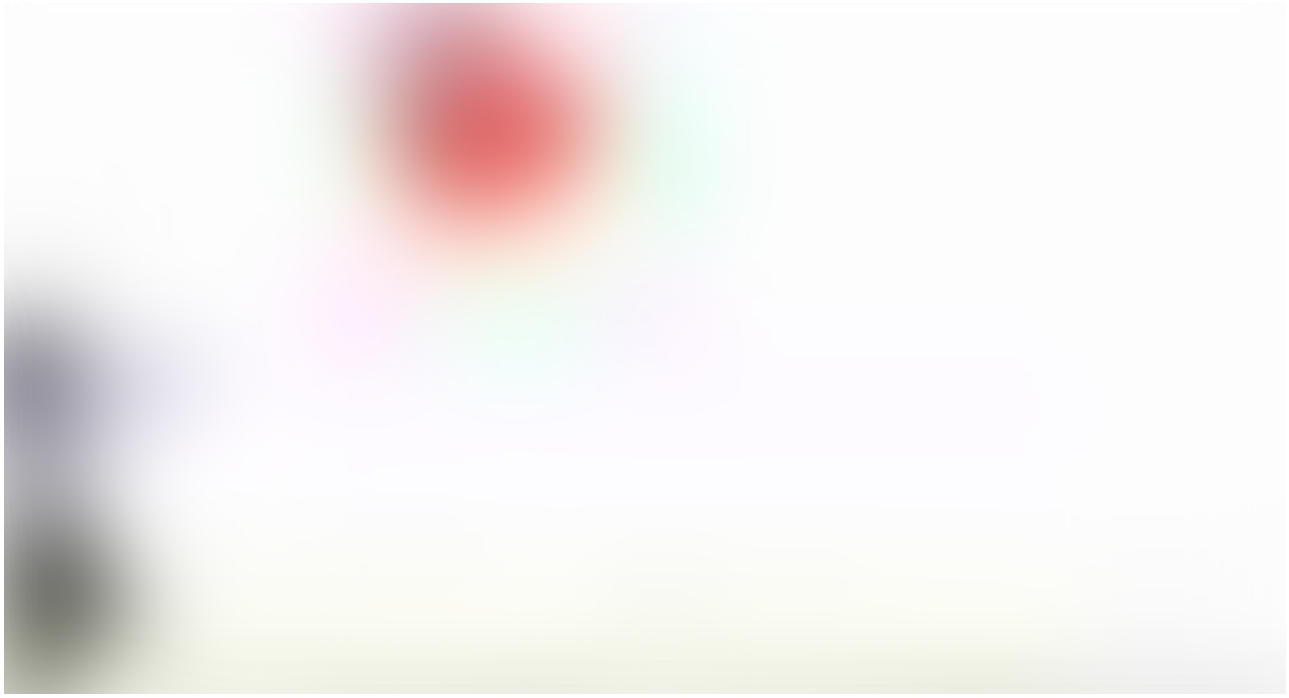
transformations, polynomial/cross features, and the use of domain expertise in creating new features.

Then, we looked at some normalization strategies available in sklearn, how to work with date features, and finally, how to handle geo features like latitude and longitude.

This has been a really long post, but I hope you've learned a lot and will use some (or many!) of the techniques explored here in your next project.

If you have any questions, suggestions, or feedback, don't hesitate to use the comment section below. I'll see you soon, happy analysis!

Link to full Notebook with explanations and codes on GitHub



*Connect with me on **Twitter.***

*Connect with me on **LinkedIn.***

. . .

*Editor's Note: **Heartbeat** is a contributor-driven online publication and community dedicated to exploring the emerging intersection of mobile app development and machine*

learning. We're committed to supporting and inspiring developers and engineers from all walks of life.

Editorially independent, Heartbeat is sponsored and published by ***Fritz AI****, the machine learning platform that helps developers teach devices to see, hear, sense, and think. We pay our contributors, and we don't sell ads.*

If you'd like to contribute, head on over to our ***call for contributors****. You can also sign up to receive our weekly newsletters (****Deep Learning Weekly*** *and the* ***Fritz AI Newsletter****), join us on* ***Slack****, and follow Fritz AI on* ***Twitter*** *for all the latest in mobile machine learning.*

Thanks to Austin Kodra.

Machine Learning Feature Engineering Python Heartbeat Data Science For Ml