Stat 447 Final Project Report

Jialong Li, Zekun Li

12/8/2022

Background

Federated Learning

Federated learning is a privacy-preserving technology to enable collaborative learning across different data providers without revealing user's data (Wu et al., 2022). Federated Learning is useful when companies want to generate a collaborative model, while they have some restrictions on how they can share their data.

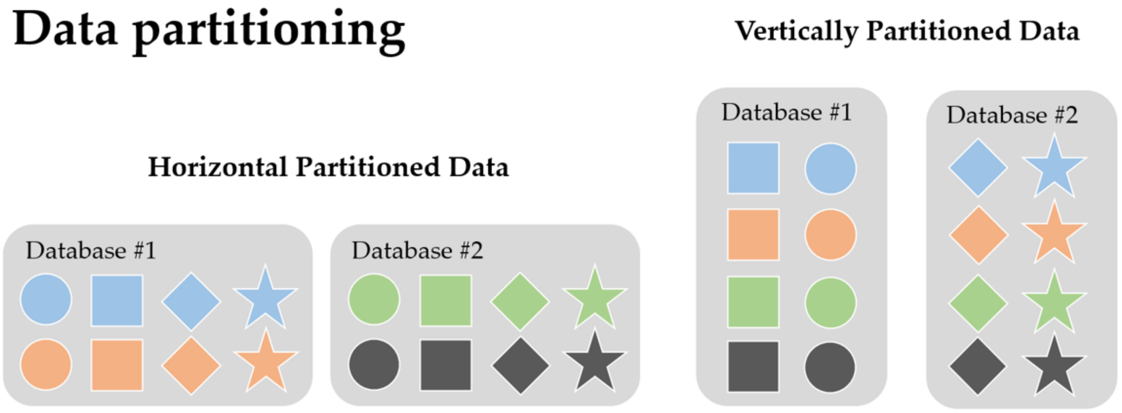


Figure 1: Data Partitioning Types

As shown in Figure 1. The feature and sample space of the data parties may not be identical. Based on how data is distributed, Federated Learning can be classified into three types: Horizontal Federated Learning, Vertical Federated Learning, and Transfer Federated Learning (Yang et al., 2019).

Federated Machine Learning: Concept and Applications

The first application of federated learning was to train a keyboard prediction model on Android platform, by Google in 2017. And Apple also utilizes the same method to improve Siri's voice recognition. In this case, they can use horizontal federated learning methods because their data providers have the same feature space across all devices.

Horizontal federated learning assumes homogeneous data. But in the real world, not every data provider focuses on one specific field, collecting identical data and only differing in the group of people it serves. Each data provider may collect different information in different ways. For example, if Google and Apple want to collaboratively train a model, their features they share will be quite different.

Vertical Federated Learning & Feature Alignment

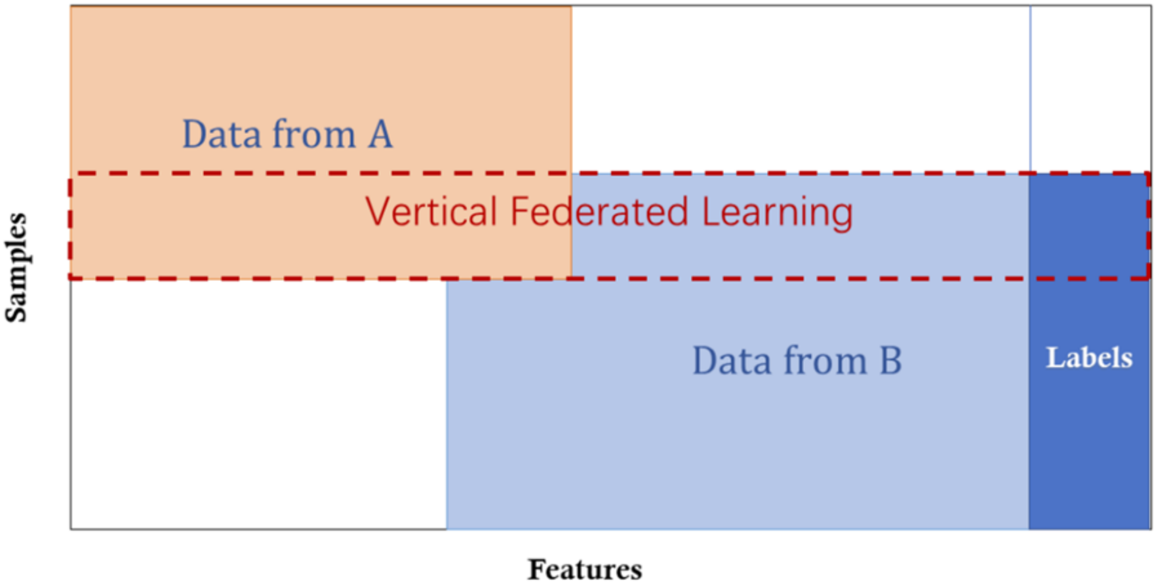


Figure 2: Vertical Federated Learning

In our project, vertical federated learning is the topic that we are interested in most. By doing so, companies can jointly train their data and get multi-industries domain insight. Unlike horizontal federated learning setup, the features in vertical federated learning are not one-to-one correspondence. We need to find where part of the features are the common feature from each dataset, then train the model using the matched (intersection) set. This process is called “feature alignment”. And if the match is successful, the vertically partitioned data will transform into horizontally partitioned data, so that we can run a horizontal federated learning method as usual to do the following jobs.

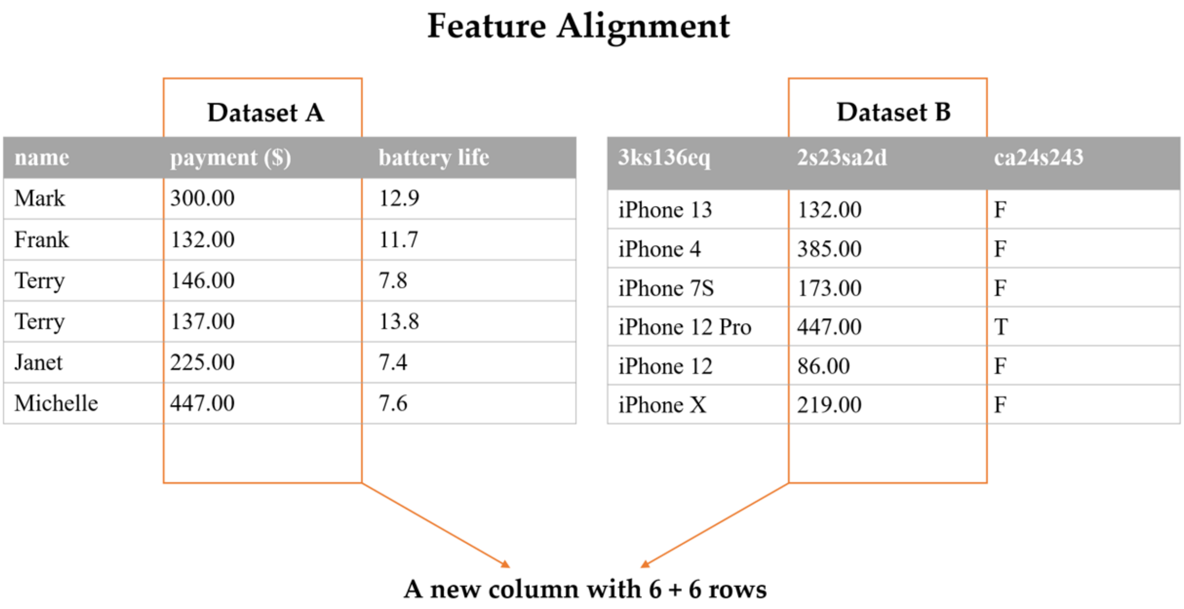
Here’s a naive example of feature Alignment. We always want to input the matched dataset to train the model.

Motivation: Why is Our Project Important?

Compared to Horizontal Federated Learning, Vertical Federated Learning is a newly emerging topic. There is limited Research on this topic.

However, in the real world, there are two main challenges. First, the data is highly heterogeneous (It’s a collaborative learning setup!). Second, companies (data providers) are under restriction on to what extent they can share their data (the dataset they provide is encrypted or sampled).

Therefore, we’d like to develop a package useful on Feature Alignment issues, especially for numeric matching. In this package, we can run our algorithm without knowing the feature name and we only have partial (sampled) data.



Feature Alignment

Dataset A

name	payment ($)	battery life
Mark	300.00	12.9
Frank	132.00	11.7
Terry	146.00	7.8
Terry	137.00	13.8
Janet	225.00	7.4
Michelle	447.00	7.6

Dataset B

3ks136eq	2s23sa2d	ca24s243
iPhone 13	132.00	F
iPhone 4	385.00	F
iPhone 7S	173.00	F
iPhone 12 Pro	447.00	T
iPhone 12	86.00	F
iPhone X	219.00	F

A new column with 6 + 6 rows

Figure 3: Feature Alignment Example

Project: an R Package for Feature Alignment

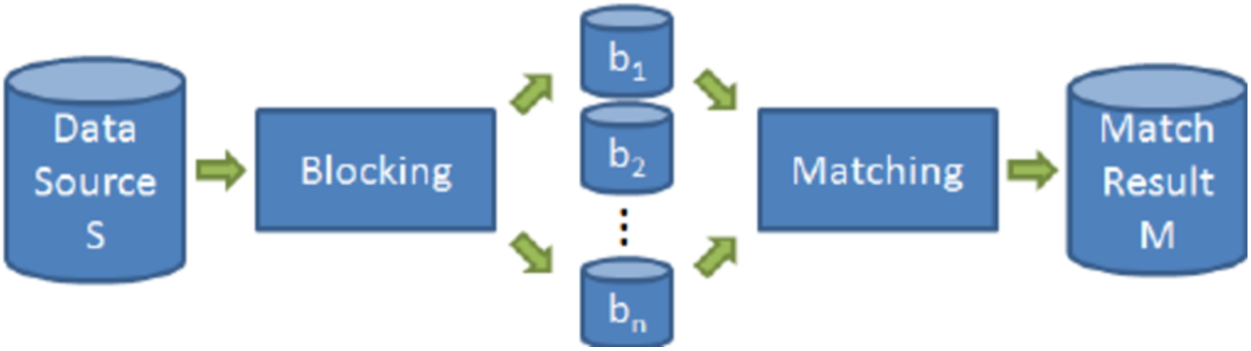


Figure 4: Work flow of feature alignment

Problem Set: Feature Alignment

This is a package to deal with Privacy Level 2 and 3 problems. In this case, either we have the same feature sample, but in a different order; or we have very different feature samples, and we need to find the overlapping parts.

When a dataset is vertically partitioned across organisations the problem arises of how to identify corresponding entities. The ideal solution would be joining datasets by common unique IDs; however, across organisations this is rarely feasible. In a non-privacy-preserving context, matching can be based on shared personal identifiers, such as name, address, gender, and date of birth, that are effectively used as weak IDs. However, in our setting, weak identifiers are considered private to each party so that not included in the dataset.

As mentioned before, the two most important steps in entity resolution are blocking and comprising. Since feature alignment is a horizontal version of entity resolution, we can also adapt the idea to develop our method.

According to our experience, previous research, such as Dedupe, RecordLinkage, and other packages only support entity resolution instead of feature alignment. And their work has good results in string matching, but low performance in numeric matching issues, or even does not support numeric matching at all. In light of that, I think it would be a good idea to develop an R package dedicated to feature alignment, especially for numeric matching.

Why is Our Project Good?

1. Blocking -> not only good for Feature Alignment, every data related job need this, especially for Data Engineers who need to deal with dirty datasets.
2. Comparison -> choose the most accurate metric to test our data, and it's robust and 100% automatically.
3. Matching -> Automatic. Easy to use.

Methodology

In order to make an educated guess on which feature in the first dataset corresponds to which feature in the second dataset, we need to find a way to quantify the "similarity" of the two features.

For numerical values, we can investigate the distributions' similarity (statistical distance between distributions) to see if the two features from two datasets are the same feature.

Numerical Distribution Metrics to Explore

From a more naive perspective, we have the t-test comparing the means of two groups. Intuitively, the mean of numerical features conveys information about the distribution. Accordingly, we may assume that two samples with similar mean are more likely to come from the same population than otherwise. We experimented with this approach to analyze its accuracy in feature alignment. However, since the t-test is a parametric test of difference, it makes some assumptions about our random variable. For example, it assumes that the data is approximately normally distributed, which will not be the case for most real-life situations. Although we relied on CLT to normalize our data, the matching result is nonetheless unsatisfactory.

To achieve satisfactory matching accuracy, we must explore more advanced metrics that consider both the distribution's shape and support. After a thorough literature review, we determined that the statistical test we ought to conduct is most likely non-parametric.

1. Earth Mover's distance (aks Wasserstein distance)

EMD-based similarity analysis (EMDSA) is an important and effective tool in many multimedia information retrieval and pattern recognition applications. However, the computational cost of EMD is super-cubic to the number of the "bins" given an arbitrary "D". According to our experiment, both the computation is expensive and the result is not satisfactory.

$$\mathbf{EMD}(P, Q) = \inf_{\gamma \in \prod(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

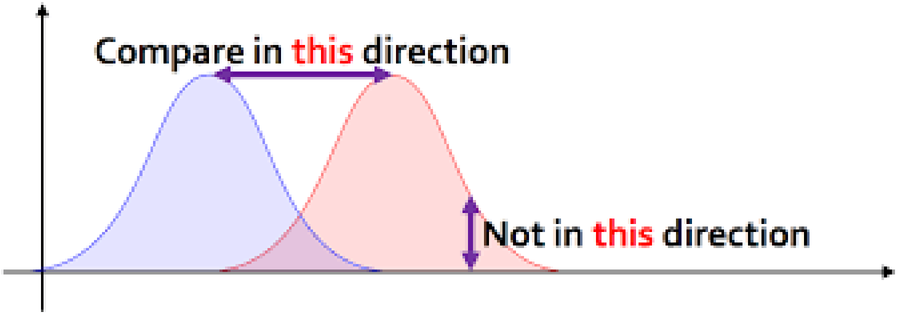


Figure 5: EMD graphic explanation

2. Kolmogorov–Smirnov test (K-S Test)

Kolmogorov–Smirnov test essentially answers the question, "What is the probability that these two sets of samples were drawn from the same (but unknown) probability distribution?" which is intuitively remarkably relevant to our numerical feature alignment problem.
The Kolmogorov–Smirnov test is a non-parametric goodness-of-fit test. It can be used to determine whether two distributions differ, or whether an underlying probability distribution differs from a hypothetical distribution. It can also be used to determine whether two samples come from the same underlying population. Suppose that the first sample has size m with an observed cumulative distribution function of *F(x)* and that the second sample has size n with an observed cumulative distribution function of *G(x)*. Define

$$\mathbf{D}_{m,n} = \max_x |\mathbf{F}(x) - \mathbf{G}(x)|$$

where F and G are the observed cumulative probability distribution functions of the two samples, and D is the maximum distance in cumulative distribution functions. We use two sample K-S tests to test for differences in the shape of two sample distributions. Notice that the K-S test compares the overall shape, not specifically central tendency, dispersion, or some other parameters. The usage of K-S tests does require univariate continuous distribution, but since our numerical features are continuous and univariate, the requirement is satisfied.

3. Cramér–von Mises distance

Similar to K-S, the Scramer-von Mises criterion is also used for judging the goodness of fit of a cumulative distribution function. The statistic is calculated as

$$\omega^2 = \int_{-\infty}^{\infty} [F_n(x) - F^*(x)]^2 \mathrm{d}F^*(x)$$

Which is basically the squared sum of deviations of weighted empirical distribution function (our sample) to theoretical distribution function (theoretical CDF). However, this criterion does not apply to the feature alignment problem because we do not have the theoretical CDF.

4. Kullback–Leibler divergence

For distributions P and Q of a continous random variable, relative entropy is defined to be the integral:

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x)\log(\frac{p(x)}{q(x)})dx$$

Where p and $\mathbf{q}$ denote the probability densities of P and Q.

String Features Matching

The other significant data type in a dataset is String, which in R language it is called Character. In order to match features of this type, we also have to come up with an indicator of “similarity”. However, unlike for numerical values, we could consider their distribution; for String features matching, there was no straightforward or standard approach. Indeed, there are a variety of methods for string similarity comparison. For instance, we have Levenshtein distance, Jaro-Winkler Distance, Hamming Distance, etc. However, all of them is to compare the similarity between two strings rather than the similarity between two string columns. There is no easy way for them to be immediately applicable to our string feature alignment problem. Hence we have to invent our own method.

Inspired by information retrieval and text mining, each word is assigned a different coordinate, and a document is represented by the vector of the number of occurrences of each word in the document; we devised a method for string feature alignment.

If two features coming from two datasets are the same feature, what are some common characteristics those string values exhibit? Some examples of string columns are Name, address, and website URL. If two features are both Names, they probably have a similar character frequency distribution. The same logic goes for address and URL. Moreover, if two features significantly differ in their character frequencies, it would be unlike for the two unknown features to correspond to the same real-world entity.

Firstly, we may count the number of occurrences of each character in a string feature column. By doing this for every appropriate string feature, we obtain a dataframe consisting of all the information for each character of our interest. Now the frequency information is stored in a data frame, and we would like to compute their mutual similarities. Like always, we need an indicator. After thorough research, we determine that cosine similarity, although conceptually naive, is the best approach.

Cosine Similarity

In data analysis, cosine similarity is a measure of similarity between two sequences of numbers. Two vectors in an inner product space. Cosine similarity is defined as the cosine of the angle between them.

The cosine of two non-zero vectors canbe derived by using the Euclidean dot product formula:

$$\text{cosine similarity} = S_C(A, B) := cos(\theta) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

In our case, we are matching string features from two datasets. Assuming that the same string feature would generally have similar character frequency, we count the occurrence of each character in a feature column to build a vector, and do this for each string feature in the two datasets. The two with the highest cosine similarity are likely to be the same feature.

Function Design

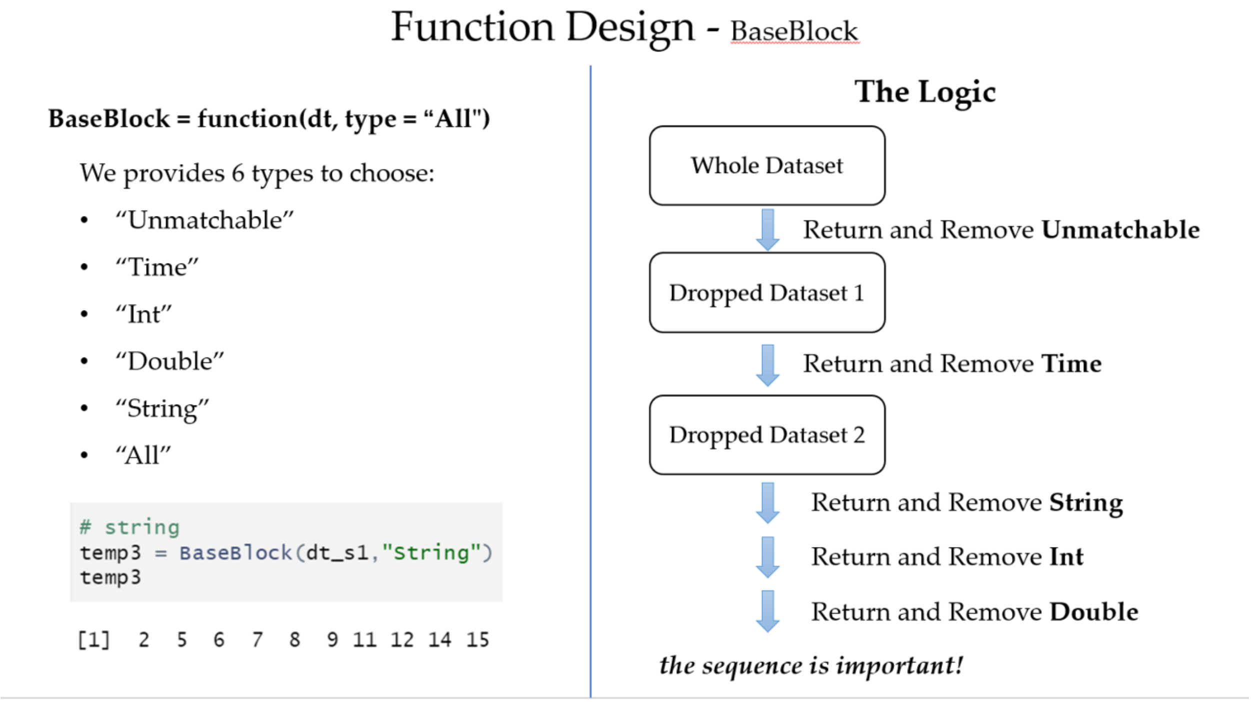
Blocking Functions

BaseBlock

The idea of this function:

Different types of data, except for int and double, are basically impossible to match together.

–Output example–:



Function Design - BaseBlock

BaseBlock = function(dt, type = “All")

We provides 6 types to choose:

- “Unmatchable”
- “Time”
- “Int”
- “Double”
- “String”
- “All”

```
# string
temp3 = BaseBlock(dt_s1,"String")
temp3
```

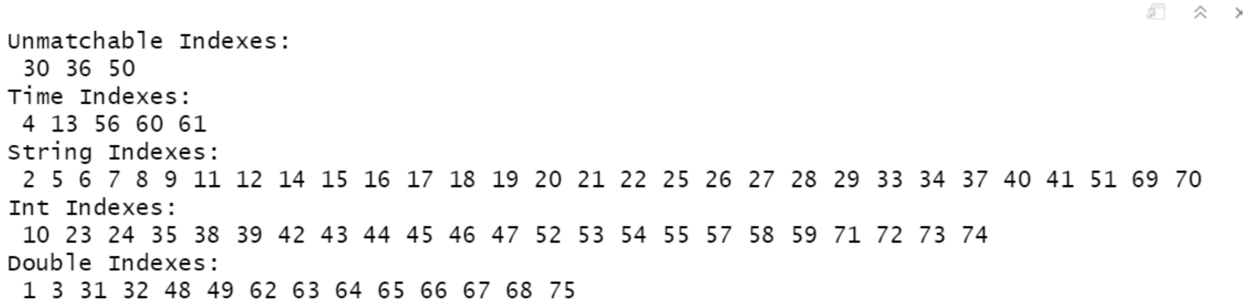
```
[1]  2  5  6  7  8  9 11 12 14 15
```

The Logic

Whole Dataset
Return and Remove **Unmatchable**
Dropped Dataset 1
Return and Remove **Time**
Dropped Dataset 2
Return and Remove **String**
Return and Remove **Int**
Return and Remove **Double**

the sequence is important!

Figure 6: BaseBlock



```
Unmatchable Indexes:
 30 36 50
Time Indexes:
 4 13 56 60 61
String Indexes:
 2 5 6 7 8 9 11 12 14 15 16 17 18 19 20 21 22 25 26 27 28 29 33 34 37 40 41 51 69 70
Int Indexes:
 10 23 24 35 38 39 42 43 44 45 46 47 52 53 54 55 57 58 59 71 72 73 74
Double Indexes:
 1 3 31 32 48 49 62 63 64 65 66 67 68 75
```

Figure 7: BaseBlock output sample

AdvancedBlock

The idea of this function:

In int, double, and string feature sample, some of the features are binary or categorical. Binary features, categorical features and Neither binary nor categorical features should be divided into separate pools and then matching.

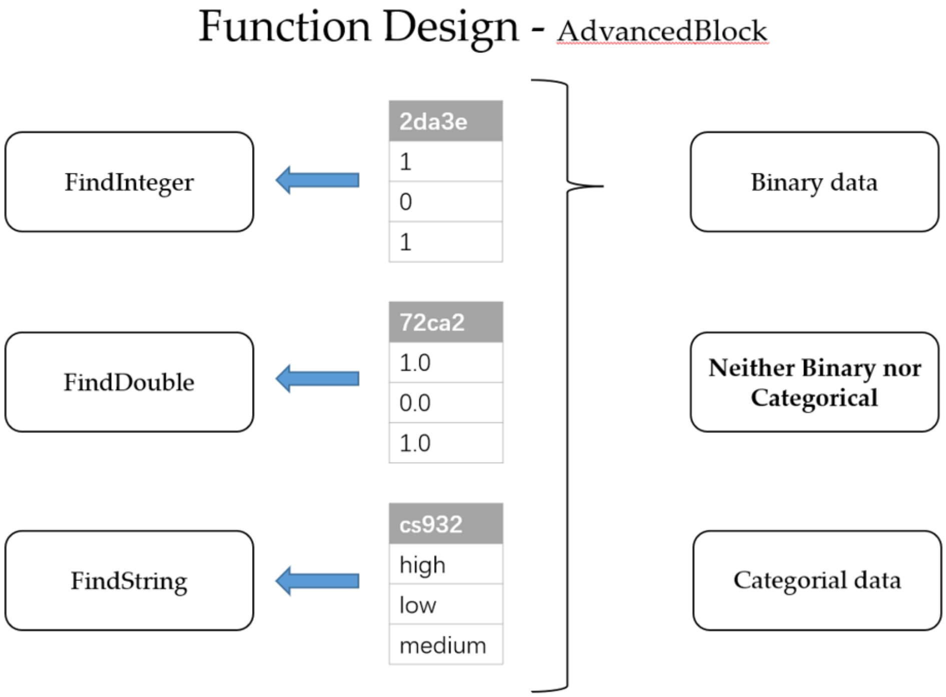


Figure 8: AdvancedBlock

Output example:

Matching Functions

visual_inspect_returns_ks_test_result()

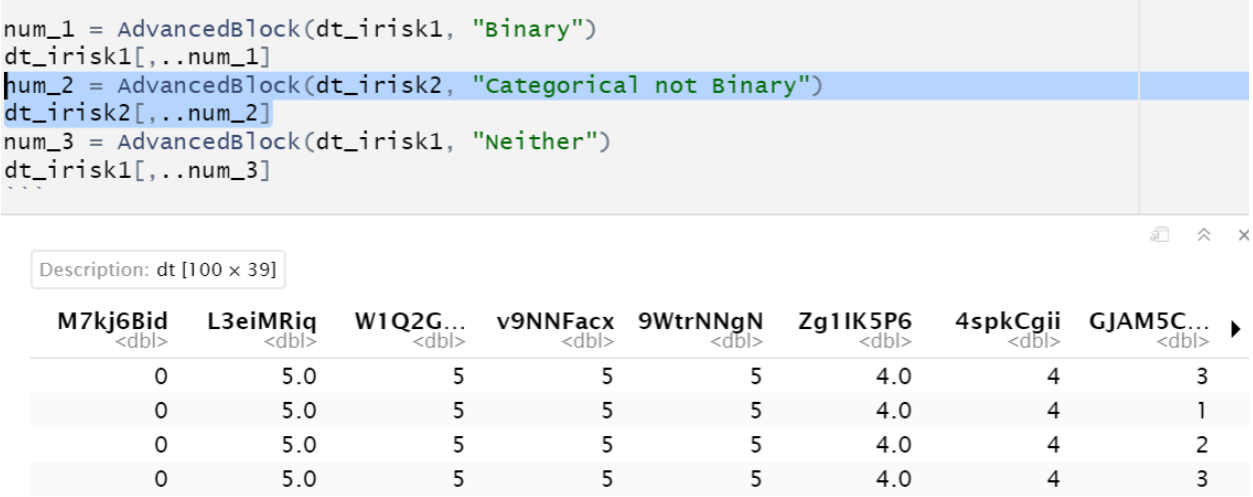
This function takes two numerical vectors and one optional argument input_sample_size; outputs a graph of their density plots annd returns the result of KS test.

find_best_match_based_on_KS_test()

Returns the matching result for numerical features. This function is the master function that combines the results of many helper functions. Helper functions include: find_best_index(), BaseBlock().

find_best_match_character_features()

Returns the matching result for string features. This function is the master function that combines the results of many helper functions. Helper functions include: get_occurrence_num(), get_occurrence_matrix_as_df(), calculate_cosine(), FindString().



~~~
num_1 = AdvancedBlock(dt_irisk1, "Binary")
dt_irisk1[,..num_1]
num_2 = AdvancedBlock(dt_irisk2, "Categorical not Binary")
dt_irisk2[,..num_2]
num_3 = AdvancedBlock(dt_irisk1, "Neither")
dt_irisk1[,..num_3]
```
~~~

Description: dt [100 × 39]

M7kj6Bid	L3eiMRiq	W1Q2G...	v9NNFacx	9WtrNNgN	Zg1IK5P6	4spkCgii	GJAM5C...
<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
0	5.0	5	5	5	4.0	4	3
0	5.0	5	5	5	4.0	4	1
0	5.0	5	5	5	4.0	4	2
0	5.0	5	5	5	4.0	4	3

Figure 9: AdvancedBlock output example

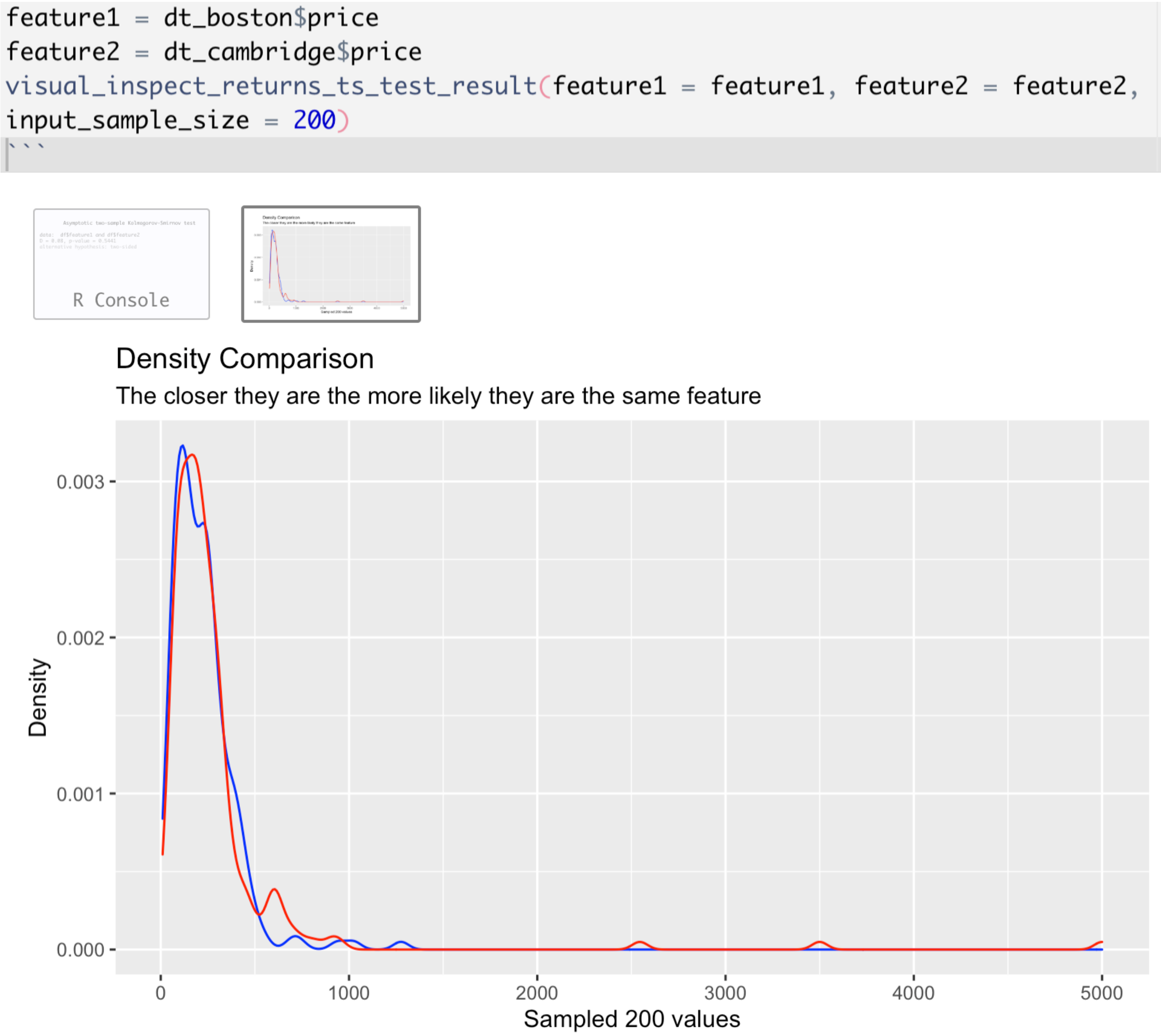
auto_matching()

This function returns the ultimate feature-wise matching result for two input datasets. User may customize the function by specifying the optional parameters (sample_size_numeric, sample_size_string, and characters_to_match) to increase matching accuracy.

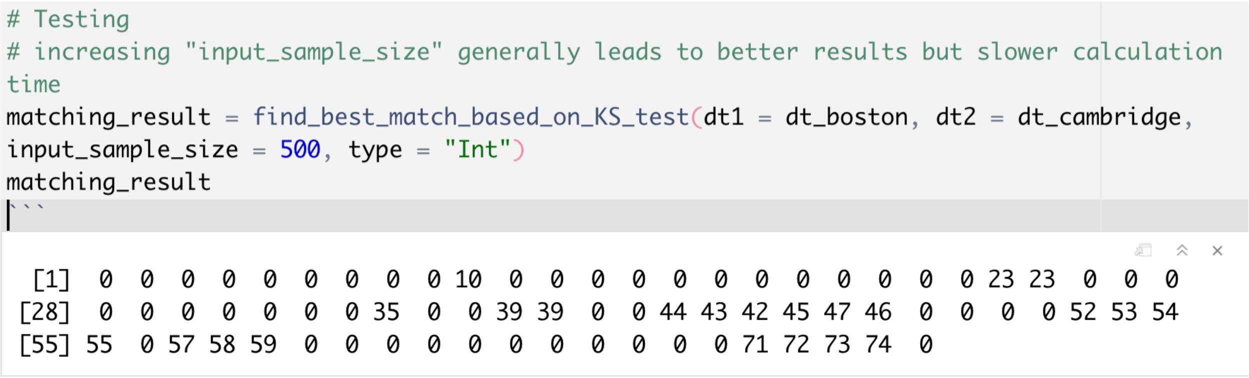
Reference

Wu, Z., Li, Q., & He, B. A Coupled Design of Exploiting Record Similarity for Practical Vertical Federated Learning. In Advances in Neural Information Processing Systems.

Yang, Q., Liu, Y., Chen, T., & Tong, Y. (2019). Federated machine learning: Concept and applications. ACM Transactions on Intelligent Systems and Technology (TIST), 10(2), 1-19.

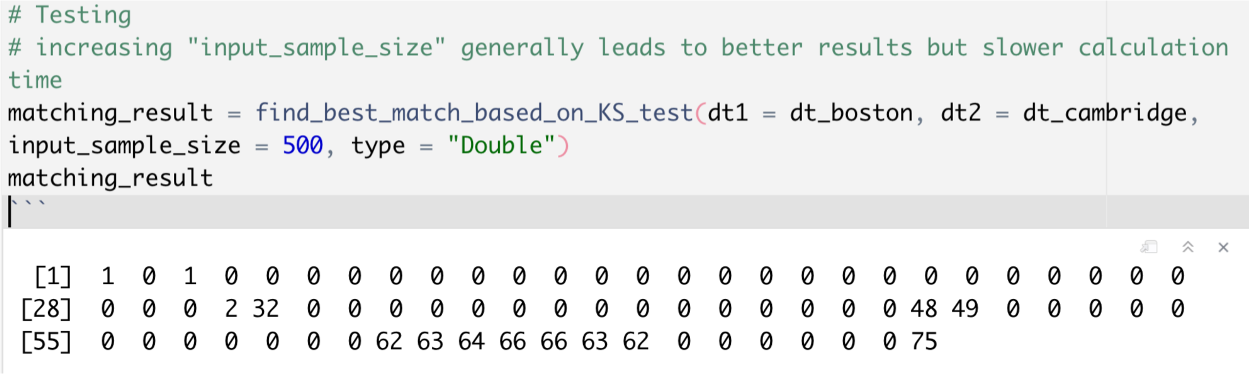


```
feature1 = dt_boston$price
feature2 = dt_cambridge$price
visual_inspect_returns_ts_test_result(feature1 = feature1, feature2 = feature2,
input_sample_size = 200)
```
```

```
           Asymptotic two-sample Kolmogorov-Smirnov test

data:  df$feature1 and df$feature2
D = 0.08, p-value = 0.5441
alternative hypothesis: two-sided
```

R Console
Density Comparison
The closer they are the more likely they are the same feature
Density
0.003
0.002
0.001
0.000
0
1000
2000
3000
4000
5000
Sampled 200 values

Figure 10: visual_inspect_returns_ks_test_result - Usage



```
# Testing
# increasing "input_sample_size" generally leads to better results but slower calculation
time
matching_result = find_best_match_based_on_KS_test(dt1 = dt_boston, dt2 = dt_cambridge,
input_sample_size = 500, type = "Int")
matching_result
```
```

```
 [1]  0  0  0  0  0  0  0  0  0 10  0  0  0  0  0  0  0  0  0  0  0  0 23 23  0  0  0
[28]  0  0  0  0  0  0  0 35  0  0 39 39  0  0 44 43 42 45 47 46  0  0  0  0 52 53 54
[55] 55  0 57 58 59  0  0  0  0  0  0  0  0  0  0  0 71 72 73 74  0
```

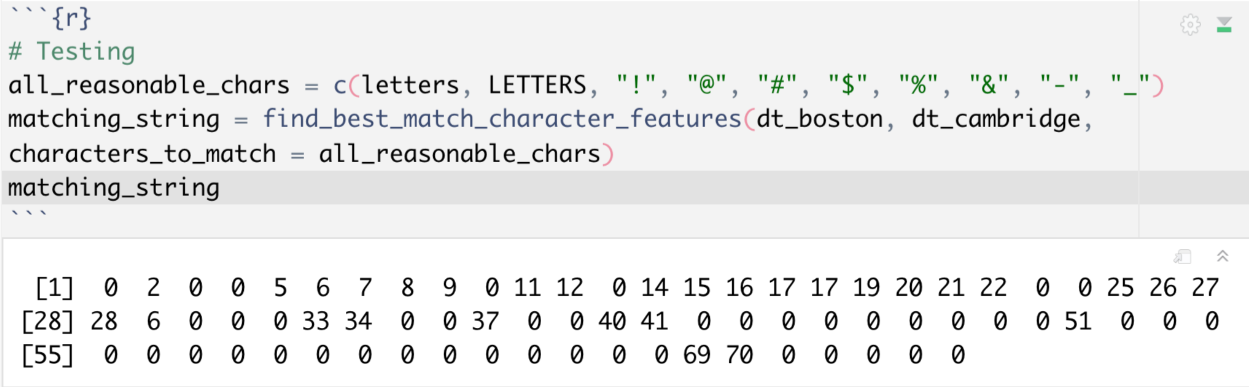
Figure 11: find_best_match_based_on_KS_test - Usage int



```
# Testing
# increasing "input_sample_size" generally leads to better results but slower calculation
time
matching_result = find_best_match_based_on_KS_test(dt1 = dt_boston, dt2 = dt_cambridge,
input_sample_size = 500, type = "Double")
matching_result
```

```
 [1]  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
[28]  0  0  0  2 32  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 48 49  0  0  0  0  0
[55]  0  0  0  0  0  0  0 62 63 64 66 66 63 62  0  0  0  0  0  0 75
```

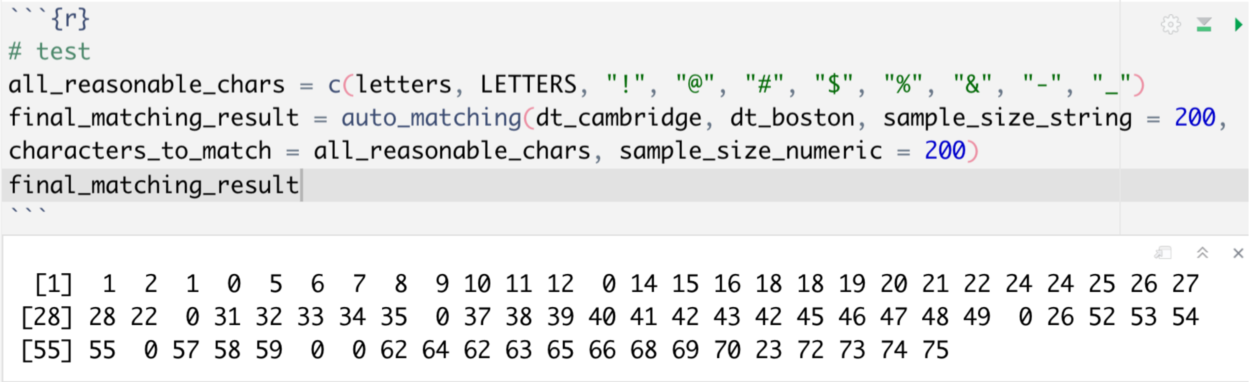
Figure 12: find_best_match_based_on_KS_test - Usage double



~~~
```{r}
# Testing
all_reasonable_chars = c(letters, LETTERS, "!", "@", "#", "$", "%", "&", "-", "_")
matching_string = find_best_match_character_features(dt_boston, dt_cambridge,
characters_to_match = all_reasonable_chars)
matching_string
```
~~~

```
 [1]  0  2  0  0  5  6  7  8  9  0 11 12  0 14 15 16 17 17 19 20 21 22  0  0 25 26 27
[28] 28  6  0  0  0 33 34  0  0 37  0  0 40 41  0  0  0  0  0  0  0  0  0 51  0  0  0
[55]  0  0  0  0  0  0  0  0  0  0  0  0  0  0 69 70  0  0  0  0  0
```

Figure 13: find_best_match_character_features - Usage



~~~
```{r}
# test
all_reasonable_chars = c(letters, LETTERS, "!", "@", "#", "$", "%", "&", "-", "_")
final_matching_result = auto_matching(dt_cambridge, dt_boston, sample_size_string = 200,
characters_to_match = all_reasonable_chars, sample_size_numeric = 200)
final_matching_result
```
~~~

```
 [1]  1  2  1  0  5  6  7  8  9 10 11 12  0 14 15 16 18 18 19 20 21 22 24 24 25 26 27
[28] 28 22  0 31 32 33 34 35  0 37 38 39 40 41 42 43 42 45 46 47 48 49  0 26 52 53 54
[55] 55  0 57 58 59  0  0 62 64 62 63 65 66 68 69 70 23 72 73 74 75
```

Figure 14: auto_matching - Usage