Feature Engineering

Introduction

What is Feature Engineering?

Feature engineering is the process of selecting, manipulating, and transforming raw data into features that can be used in supervised learning. In order to make machine learning work well on new tasks, it might be necessary to design and train better features. As you may know, a “feature” is any measurable input that can be used in a predictive model — it could be the colour of an object or the sound of someone’s voice. Feature engineering, in simple terms, is **the act of converting raw observations into desired features using statistical or machine learning approaches.**

What is Generics?

Generics means **parameterized types**. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Scenario

You are a feature engineer in company XYZ and you are provided with 4 tables as shown below. Please develop an **extra simple** feature engineering application that implements **Java Generics** to transform the raw data provided into desired features.

Raw Data

To simplify the development, let’s assume that all **dates** used in this test are **integers** (the higher the value, the later the date).

Transaction Table

user_id	transaction_amount	date

user_id	transaction_amount	date
user2	null	3
user2	78.90	0
user2	82.10	1
user1	120.43	0
user1	234.80	2
user3	1780.00	1

Login Table

user_id	latitude	longitude	date
user1	-80.3	79.1	0
user2	95.6	-112.3	0
user2	80.4	165.4	2
user2	78.2	177.3	1
user3	-128.3	12.5	3
user3	3.5	23.7	0

Fraud Table

user_id	is_fraud	date
user1	true	0
user2	false	1

Fraud Scale Table

user_id	fraud_scale	date
user2	0.7	2
user3	0.3	3

Requirements

1. Your application **must** be written in **JAVA** language.

2. Your application **must** implement the **Generics** concept.
3. Your application must have **at least** 5 classes (exclude `Main` class). You could create more classes whenever you think it's necessary but the 5 classes listed below are a **MUST**.
4. Some people might be familiar with `instanceof` in JAVA and want to use it here. You are **not restricted** to use this in your application, but all of the following requirements should be able to do without using `instanceof` .
5. Create a **comparable generics class** named `Data` to represent all the raw data provided. Imagine an object of this class is a row in any table above, this class must include:
    a. A **private** variable named `user` of **String** type
    b. A **private** variable named `data` of **comparable generics** type
    c. A **private** variable named `date` of **int** type
    d. An all-argument constructor
    e. A **public** `compareTo` method that compares the data value of this object with the input data value and **returns an integer**. Says you define your generic type as `T`, then it should be something like `int compareTo(T other)`
    f. A **public** `compareTo` method that compares between this `Data` class object with another `Data` class object **based on** `date` variable (subtract another object `date` value from this object `date` value) and **return an integer**
    g. Any getter/setter and toString method(s) you think it's needed. You are **restricted** to add any additional variable(s) for this class.
6. Create a **generics abstract class** named `Feature` to represent all features we are going to create later. This abstract class must include:
    a. A **private** variable named `data` of **List of Data** type. Imagine this represents any table shown above.
    b. An all-argument constructor
    c. An **abstract** method named `dataCleaning` that **accepts no argument** and **returns this Feature (or its child class) object**

d. A **public concrete** method named `merge` that merges another Feature object's `data` into this Feature object's `data` based on a reference/condition and **returns this Feature (or its child class) object**.

   i. If I want to merge Fraud Scale Table data into Fraud Table, this method should convert fraud_scale value to boolean **(if less than 0.5 then false else true)** type and append into Fraud Table data

   ii. If I want to merge Fraud Table data into Fraud Scale Table, this method should convert is_fraud value to double/float **(if true then 1.0 else 0.0)** type and append into Fraud Scale Table data

   iii. Of course, this method should not only work for Fraud Table and Fraud Scale Table, it should work for any class that is a child class of `Feature` class

   iv. You have the freedom to decide what is/are the arguments you need for this function. The only restriction is that the **logic described above must be implemented in this method**, instead of doing it in the main method and passing in the result into this method.

e. A **public** method named `sortByDate` that **accepts no argument** and **returns** `data`. This method should sort data by its elements' `date` value

f. Any getter or setter method(s) you think it's needed. You are **restricted** to add any additional variable(s) for this class.

7. Create a **class** named `Transaction` which is a **child class** of `Feature` to represent Transaction Table feature. This class must includes:

   a. An all-argument constructor

   b. Implementation of `dataCleaning` method. This method should remove any `Data` object(s) with transaction_amount is null or less than 0 from `Feature.data` list

   c. A **public** method named `transformation` that transforms every transaction_amount value into deviation from mean at that value date and **returns this** `Transaction` **object**.

      i. Ex. if the row is of `date` 1, mean will be the average of transaction_amount of all rows of `date` <= 1 and subtract the mean value from this row value to get the deviation

ii. You have freedom to decide what is/are the arguments you need for this function. The only restriction is the **logic described above must be implemented in this method**, instead of doing it in the main method and passing in the result into this method.

8. Create a **class** named `Login` which is a **child class** of `Feature` to represent Login Table feature. This class must includes:
   a. An all-argument constructor
   b. Implementation of `dataCleaning` method. This method should remove any `Data` object(s) with latitude is not within [-90, 90] or longitude is not within [-180, 180] from `Feature.data` list.
9. Create a **generics class** named `Fraudulent` which is a **child class** of `Feature` to represent both Fraud Table and Fraud Scale Table features. This class must includes:
   a. An all-argument constructor
   b. Implementation of `dataCleaning` method. This method should remove the `Data` object(s) from `Feature.data` list if the object's:
      i. Fraud Table: is_fraud is null
      ii. Fraud Scale Table: fraud_scale is null or not within [0, 1]
   c. A **public** method named `transformation` that **returns this** `Fraudulent` **object** after it does the following:
      i. Fraud Table: if is_fraud is true then true else false (basically it means no transformation is needed for this table)
      ii. Fraud Scale Table: if fraud_scale < 0.5 then 0.0 else 1.0
      iii. You have freedom to decide what is/are the arguments you need for this function. The only restriction is the **logic described above must be implemented in this method**, instead of doing it in the main method and passing in the result into this method.
10. Create a **public static** method in your `Main` class named `pointInTimeJoin` that **accepts a list of** `Feature` **objects (features) and a single** `Feature` **object (label)**. This method should perform point in time join for all the features input based on the

input label and **return a list of list of values of all the features and the label**. For every label value, point in time join will search for features' value of the latest date that is before the label date. For visual explanation, see Appendix A.

11. In your `main` method, you should be able to do all of the followings:

```
// initialize 'transactions' (Transaction object) first
// with all data shown in Transaction Table above
transactions = ...

// this ONE statement should be able to compile and execute
// you are not allowed to split this into multiple statements
transactions.dataCleaning().transformation();

//print out all data in transactions


// initialize 'logins' (Login object) first
// with all data shown in Login Table above
logins = ...

// this statement should be able to compile and execute
logins.dataCleaning();

//print out all data in logins


// initialize 'fraudulent' and 'scaleFraudulent' (Fraudulent objects) first
// with all data shown in Fraud Table and Fraud Scale Table above respectively
fraudulent = ...
scaleFraudulent = ...

// this ONE statement should be able to compile and execute
// you are not allowed to split this into multiple statements
// notice that I do not fix the arguments of transformation and merge methods
// before this for you thus the ... part is depending on how you define your methods
fraudulent.dataCleaning().transformation(...)
  .merge(scaleFraudulent.dataCleaning().transformation(...), ...);

// print out all data in fraudulent


// initialize 'fraudulentCopy' and 'scaleFraudulentCopy' (Fraudulent objects) first
// with all data shown in Fraud Table and Fraud Scale Table above respectively
fraudulentCopy = ...
scaleFraudulentCopy = ...

// this ONE statement should be able to compile and execute
// you are not allowed to split this into multiple statements
// notice that I do not fix the arguments of transformation and merge methods
// before this for you thus the ... part is depending on how you define your methods
```

```
scaleFraudulentCopy.dataCleaning()
  .merge(fraudulentCopy.dataCleaning(), ..)
  .transformation(...);

//print out all data in scaleFraudulentCopy


// Do point in time join for
// transactions and logins features
// and passing in fraudulent as label
pointInTimeJoin([transactions, logins], fraudulent)
// print out the result of pointInTimeJoin()
```

Appendix A

user_id	transaction_amount	date
user1	120.43	0
user2	78.90	0
user2	82.10	1
user3	1780.00	1
user1	234.80	2
user2	null	3

There are 2 data with date <= 1, thus we take the latest which is 82.10

user_id	is_fraud	date
user1	true	0
user2	false	1

user_id	latitude	longitude	date
user1	-80.3	79.1	0
user2	95.6	-112.3	0
user3	3.5	23.7	0
user2	78.2	177.3	1
user2	80.4	165.4	2
user3	-128.3	12.5	3

Same case here, we take the latest value which is (78.2, 177.3)

The final result of point in time join should be

transaction_amount_deviation	latitude	longitude	is_fraud
0.00	-80.3	79.1	true
1.60	78.2	177.3	false
1.60	80.4	165.4	true
0.00	3.5	23.7	false