DB Energie GPS comparator - Documentation

Mikoláš Fromm

02.08.2023

1 Introduction

This document describes the DB Energie GPS comparator application. The application is used to compare the GPS data from the DB Energie invoicing system with the data from the GPS system of the vehicle. The whole application is written in C# and is equipped with WinForms interface for easy use.

Motivation

My motivation for this topic is my part-time job in a company which operates locomotives in Germany (on Deutsche Bahn infrastructure). Deutsche Bahn (or DB from now on) regularly send energy invoices that full of mistakes where they claim the locomotive was spending energy in Germany, even though in reality the locomotive was outside Germany completely, for example in Hungary. Therefore is then needed to compare all timestamps from the DB invoices with the location data from the locomotive GPS system. This is a very time-consuming process, therefore I decided to automate it.

Goals

Major goal was to create an application capable of checking the correctness of the data from the invoice and separating the correct from the incorrect. Minor goals were also to automatically generate a refund request file and to automatically reassign all energy consumptions to all operators that were using the locomotive at the given time.

2 Application subdivision

The project is separated into the following sections:

- Wrappers
  - DB Energie wrapper
  - GPS wrapper
  - LokoUsage wrapper
- Data structures
- Data processing
  - Comparator
  - Exporter
- WinForm interface

2.1 Wrappers

Wrappers are used to communicate with or parse data from external systems. The main reason for this separation is to make the application more modular and easier to maintain. At the moment, none of the sources is offering any kind of API, therefore all the wrappers require uploading a .csv or .xslx file manually, but in case it changes, only the single wrapper will need to be modified and the rest of the application will remain the same.

All the wrappers define their own Interface, with which the core of the application communicates. This interface is then implemented by the wrapper itself.

2.1.1 DB Energie wrapper

```
public interface IDBE_wrapper
{
    void GetAllEntriesFromDBE();

    List<DbeEntry> Entries { get; }

    HashSet<LocoId> LocosIncluded { get; }

    public Dictionary<int, LocoId> LocoIdForGivenColumn { get; }
}
```

This wrapper currently has two implementations available:

1.
   ```
   // reading attachment for DBE invoice
   public class DBE_wrapper : IDBE_wrapper
   ```

2.
   ```
   // reading CSV export from BahnStromPortal
   public class DBE_abstimmung_wrapper : IDBE_wrapper
   ```

where the first one parses the input .csv file, which is attached to the DB Energie invoice and which has one 15 minutes long energy consumption timespan for each locomotive on separate row. It is also to notice that the invoice contains only the locomotives that were spending energy in Germany and that the entries in the attachement are sorted by the datetime, which is later used in evaluation.

The second implementation is used to parse the .csv file exported from the Bahn-StromPortal, which contains all the energy consumption timespans for all locomotives, even those that were not spending energy in Germany. The entries are also sorted by datetime, but each row contains entry for each locomotive. This information must be saved for later evaluation, therefore the $Dictionary < int, LocoId >$ is used, where the key is the column number and the value is the LocoId of the locomotive in that column.

The application is also prepared for a very big amount of data, therefore the wrapper is not saving all entries to the memory, but only saves the datetime span, locoId and the coordinates of the whole entry. This implies that the data file is read multiple times during the evaluation, but the user memory will be saved.

2.1.2 GPS wrapper

```
public interface IGPS_wrapper
{
    // indexed by each loco, containing sorted dates "from - to" in germany
    Dictionary<LocoId, List<DateSpan>> GetAllTimesInGermany(Dictionary<LocoId,
                                                            GpsLocoFilePath> gpsMapping);
    HashSet<LocoId> LocomotivesWithOutGPS { get; }
}
```

This wrapper is used to parse the GPS data from the locomotive GPS system. It uses output from PosiTrex GPS system, which is able to export all border-crossings which determines exactly when each locomotive has entered / left a specific country. PosiTrex is capable of exporting only a file per locomotive, therefore the implementation requires a filePath to the border-crossing file of each locomotive involved. Then it only reads the file line by line, where one line contains dateFrom, dateTo and the country of the activity, and filters out only the lines containing "Germany" as the country. The result is then saved in the $Dictionary < LocoId, List < DateSpan >>$, where the key is the LocoId and the value is the list of all the timespans when the locomotive was in Germany.

It might also happen that the locomotive has faulty or no GPS system installed and therefore the GPS data is not available. In this case, the locomotive is considered to be in Germany for the whole time and the entry is saved in the $HashSet < LocoId > LocomotivesWithOutGPS$ to later notify the user about this fact.

2.1.3 LokoUsage wrapper

```
public interface ILokoUsage_wrapper
{
    // indexed by locomotive, containing "from - to" and customer name for each time span.
    Dictionary<LocoId, List<CustomerDateSpan>> GetAllCustomers(IEnumerable<LocoId> locomotives);

    IList<string> CustomerNames { get; }
}
```

This wrapper parses an input file, which holds for every locomotive and for every day (that might be split into more pieces) a customer name, who was using the locomotive at the time. It is a .xlsx file where each locomotive is on a single sheet, named by its LocoId. This wrapper is invoked after a DB Energie input file is read, therefore only the contained locomotives are parsed. The result is then saved in the $Dictionary < LocoId, List < CustomerDateSpan >>$, where the key is the LocoId and the value is the list of all the timespans when the locomotive was used by a specific customer. *CustomerDateSpan* is almost like a *DateSpan*, but it also contains the customer name.

Asides from the data, the wrapper also saves all the customer names in the $IList < string > CustomerNames$ to later generate a file for each customer.

2.2 Data structures

The application uses the following data structures instead of the primitive types in order to maximize the readability of the code and to make the code more self-explanatory.

```
public struct LocoId {
    public string shortId;
    public string longId; }

public record struct SheetIndex {
    public int row;
    public int column; }

public struct GpsLocoFilePath {
    public string path; }

public enum CheckMethod {
    None,
    InvoiceCheck,
    PreCheck }

public enum DistributionResult {
    NoError,
    IncludingErrors }

public record struct CustomerDateSpan {
    public DateSpan dateSpan;
    public string customerName; }

public record struct DbeEntry {
    public DateTime date;
    public LocoId id;
    public SheetIndex sheet_index; }

public record struct DateSpan {
    public DateTime startDate;
    public DateTime endDate; }
```

2.3 Data processing

This section contains the core of the application, where the data from the wrappers is processed and evaluated. Since more than one evaluation method is available, the application uses the *CheckMethod* enum to determine which method should be used.

2.3.1 Comparator

```
public static class Comparator {
    // supportive method for parsing the locomotive id from the DBE entry
    public static LocoId GetLocoId(string locomotive) {...}

    // main unit deciding whether the entry is correct or not
    public static EvaluationResults EvaluateResults(
        IEnumerable<LocoId> locomotives_in_germany
        IEnumerable<DbeEntry> dbe_entries,
        Dictionary<LocoId, List<DateSpan>> real_loco_dates_in_germany) {...}

    // Invoice check with operators split
    public static void MakeCompareWork(
        Dictionary<LocoId, GpsLocoFilePath> gpsMapping,
        IEnumerable<DbeEntry> dbeEntries,
        HashSet<LocoId> locosInGermany,
        IGPS_wrapper gpsWrapper,
        ILokoUsage_wrapper lokoUsageWrapper,
        IExporter exporter,
        double price = 0,
        bool splitCustomers = false) {...}

    // Invoice check without operators split
    public static void MakeCompareWork(
        Dictionary<LocoId, GpsLocoFilePath> gpsMapping,
        IEnumerable<DbeEntry> dbeEntries,
        HashSet<LocoId> locosInGermany,
        IGPS_wrapper gpsWrapper,
        IExporter exporter) {...}

    // Pre-check without operators split
    public static void MakeCompareWork(
        Dictionary<LocoId, GpsLocoFilePath> gpsMapping,
        IEnumerable<DbeEntry> dbeEntries,
        Dictionary<int, LocoId> locoIdForGivenColumn,
        HashSet<LocoId> locosInGermany,
        IGPS_wrapper gpsWrapper,
        IExporter exporter) {...}
}
```

The *EvaluateResults* method is the main unit of the application, where the data from the DBE invoice is compared with the data from the GPS system. For the successful evaluation, it is required that both *dbe_entries* and *real_loco_dates_in_germany* are date-time sorted. As long as the dbe_entry is in the range of any real stay of the locomotive in Germany, the entry is considered to be correct. If the entry is outside the range, it tries to move to the next range and if it is not even in that range, it is considered to be incorrect.

The output of the evaluation is the *EvaluationResults* object, which contains information about error entries, which are grouped together, and about each locomotive stays, which is then used to generate the output files.

Above that all is the main function *MakeCompareWork* which is gathering all necessary data from the wrappers and then invokes the *EvaluateResults* method. The method is overloaded to support all the evaluation methods available. Because the *EvaluateResults* function does not split the entries by operators, the *MakeCompareWork* method is also overloaded to support splitting the entries by operators. This is done by the *IExporter* interface, which is overloaded to generate more type of outputs, one of which is the output file for each operator.

2.3.2 Exporter

```
public interface IExporter
{
    // export the evaluation and split the entries by operators
    void ExportAndFillTemplate(EvaluationResults evaluationResults,
                               Dictionary<LocoId, List<CustomerDateSpan>> customerDateTimes,
                               IEnumerable<string> customerNames,
                               double price);

    // export the evaluation without splitting the entries by operators
    void ExportAndFillTemplate(EvaluationResults evaluationResults);

    // export the pre-check evaluation without splitting the entries by operators
    void ExportAndFillTemplate(EvaluationResults evaluation,
                               Dictionary<int, LocoId> LocoIdForGivenColumn);

    // add the output directory given by user from the WinForm interface
    IExporter AddOutputDir(string outputDir);
}
```

The *Exporter* is filling prepared templates from the *templates* folder with the data from the evaluation. Because the customer should get the most of the information possible, the template is filled with all the rows from the source file that have been considered correct earlier. This is also the reason, why *SheetIndexes* are stored at the beginning of the evaluation.

Independently on the customer-split, a refund file, containing all the dates of the incorrect entries, is generated. This file is in different format than the template for users. It contains fullId of the locomotive, Id of the virtual energy card of the curernt operator, dateFrom and dateTo of the incorrect entry and the position of the locomotive (Interior or Exterior - relative to the Germany).

All of these files are then saved in the output directory, which is given by the user in the WinForm interface.

2.4 WinForm interface

The WinForm interface is used to make the application more user-friendly. It is also used to gather all the necessary data from the user, such as the input files, output directory, price of the energy, etc. The interface is also used to invoke the *MakeCompareWork* method from the *Comparator* class.

User is first prompted to choose the input file, which should be checked. Both types of checking require the input file to be a .csv file. Therefore, file-type filter is set in the FileDialog. The user must then choose one of the two checking methods by clicking on the appropriate checkBox. The first one is the invoice check, which is used to check the correctness of the DB Energie invoice. The second one is the pre-check, which is used to check the correctness of the data from the BahnStromPortal. Both checks require to choose the GPS data file for each locomotive, which are then used to compare the data from the invoice with the data from the GPS system.

When the input file is chosen, the application scans the file completely to find all contained locomotives and generates for each locomotive one fileDialog button, allowing the user to match the GPS file with the locomotive easily.

When checking the DB Energie invoice and also choosing to make the customer split, the user is also prompted to choose the file containing the customer leases, which serves as a source of datetimes when the locomotive was used by a specific customer. This file is then parsed by the *LokoUsageWrapper* and the result is used to split the entries by operators.

Lastly, when choosing to make a customer split, it is also required to enter a price of the energy, which is then used to calculate the amount of money that should be reinvoiced to each operator.

User also has a possibility to choose the output directory, where all the output files will be saved. If the user does not choose any output directory, the files will be saved in the fixed *results* directory in the root of the project directory.