Tutorial on the Nervousnet Challenge

About the data set:

The data provided represents energy consumption gathered from smart meters in Northern Ireland, from two types of consumers – households and small enterprises. Data is preprocessed and formatted in the following fashion:

Day ID, Time ID, Raw Value

Day ID comes in two ranges:

- The first range is [213, 240] and it represents one-month data during winter. More precisely, those are the 3rd, 4th, 5th and 6th week of a year.
- The second range is [388, 415] and it represents summer one-month data (weeks 28, 29, 30 and 31).

Time ID is identification of daily measurement, and it comes in the range [1, 48]. Data were collected every half an hour, hence measurement with ID 1 represents measurement taken at 00:30 during day *i*, and measurement with ID 48 of day *i* represents measurement taken at midnight between days *i* and *i+1*.

Raw Value is non-negative real number (of type double) and represents smart meter measurement.

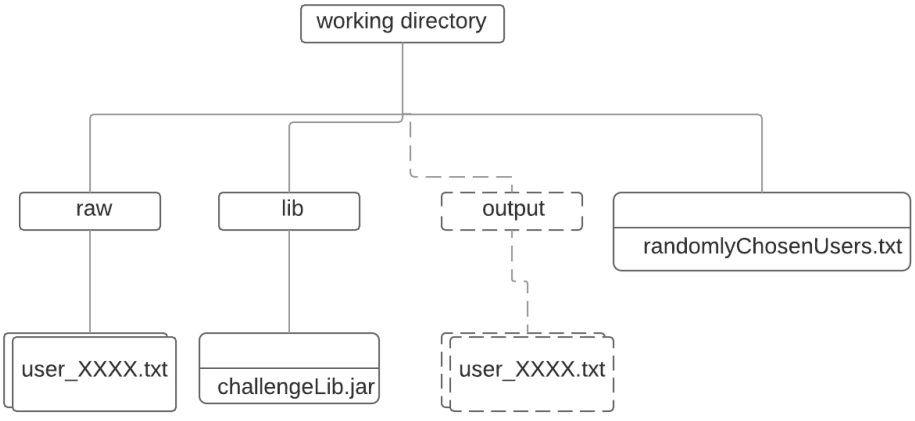
Data for every day within each of the ranges is consecutive and complete. In other words, there are no missing values in measurements for each of the days. For each of the weeks, data about all 7 days is provided.

A portion of 1000 users is randomly chosen from the data set in the time ranges described above. User ID is in range [1000, 7444], but note that data about only 1000 of them is provided. Each user-file contains complete data about all days from the ranges in the format mentioned. User IDs may not be consecutive. Users' data is given in separate files named as follows:

user_XXXX.txt

where XXXX represents user ID from the above mentioned range.

Directories hierarchy is given in the diagram below. All user-files are provided in the “raw” directory. The list of chosen users is provided in randomlyChosenUsers.txt file.



About the library:

For easier loading of the raw files and the dumping of the summarized data to disk, we have provided the “challenge.jar” Java library.

Class Loader:

Loader class is provided for loading raw files into memory. By default, it reads user-files from “raw” directory. For changing the path, you should use method

```
public void setPath (String rawPath);
```

The main method for obtaining raw data is

```
public HashMap <Integer,
                LinkedHashMap <Integer,
                LinkedHashMap <Integer, Double>>>
     exportRawValues()
          throws MissingDataException, MissingFileException;
```

The raw data in the hash map is organized as follows:

HashMap<***User ID***, LinkedHashMap<***Day ID***, LinkedHashMap<***Time ID***, ***Raw Value***>>>

LinkedHashMap is used to ensure data ordered in ascending order. Note that this method returns only reference to the hash map. If you want to keep raw values untouched, you can obtain a deep-clone copy of the hash map by invoking:

```
public HashMap <Integer,
                LinkedHashMap <Integer,
                LinkedHashMap <Integer, Double>>>
     exportClonedRawValues()
          throws MissingDataException, MissingFileException;
```

Loader class also provides a set of utility methods for iterating through the hash map.

```
public ArrayList<Integer> getSortedUsers()
```
→ returns User IDs sorted in ascending order

```
public ArrayList<Integer> getSortedDays(Integer user)
```
→ returns Day IDs sorted in ascending order, for the user specified

```
public Double getRawValue(Integer user, Integer day, Integer time)
```
→ returns raw value for the user ID, day ID and time ID specified

```
public ArrayList<Double> getSortedRawValues(Integer user, Integer day)
```
→ returns a list of raw values for the user ID and day ID specified, ordered in ascending order of corresponding time IDs.

Note that before all these utility methods, an `exportRawValues()` or an `exportClonedRawValues()` must be invoked, as these two methods actually load the raw files.

Class Dumper:

The dumper class formats output data and dumps them to disk. The default path for writing output data is directory “output” in your working directory. If directory does not exist, Dumper class creates it. **If data already exists, Dumper class overwrites existing content.**

The main class for dumping output data is the following:

```
public void dump(
            HashMap <Integer,
            LinkedHashMap <Integer,
            LinkedHashMap <Integer, Double>>> map);
```

As an argument, it uses the hash map organized the same way as the raw data hash map. The only difference is that, instead of raw values, it now should contain summarized values.

Before writing output data, it checks if there is any missing data. In the case some data is missing, dumping will be aborted. Data is not complete if it lacks information about any of the user IDs, day IDs or time IDs from the aforehand mentioned ranges. If data is complete, the method makes sure that all data is sorted in ascending order of their respective IDs, and then written to the disk.

The output data for each user is written in a separate file identified by corresponding user's ID. Formatting of the output data is organized in a similar fashion as raw data:

Day ID, Time ID, Summarized Value

If the hash map forwarded as argument contains any information labeled by identifiers that are out of their corresponding ranges, it will be ignored.

The path for output files can be changed by using

```
public void dump(
            String path,
            HashMap <Integer,
            LinkedHashMap <Integer,
            LinkedHashMap <Integer, Double>>> map);
```

Note that all directories on the path must be created manually in this case.

Dumper class provides one more public static method for creating and initializing new output hash map:

```
    public static HashMap <Integer,
                  LinkedHashMap <Integer,
                  LinkedHashMap <Integer, Double>>>
               initOutputMap();
```

It initializes hash map with user IDs and day IDs from the raw values. In the end, user should only provide one summarization value for each time ID for each of the user IDs and day IDs.

Class CalendarCalculator:

The `CalculatorClass` provides you with static utility methods for getting around the calendar based on Day IDs and week numbers.

`public static int` **`calculateWeek`**`(int day)`
→ returns week number based on day ID specified

`public static DAYS` **`calculateDayOfWeek`**`(int day)`
→ returns day of the week from the range `[DAYS.MONDAY, DAYS.SUNDAY]` , based on day ID specified

`public static int` **`calculateDay`**`(int week, DAYS day)`
→ returns day ID based on week number ad day of the week specified

Exceptions:

These utility methods throw some custom-made exceptions that should help in debugging process:

`IllegalHashMapArgumentException`
→ occurs when any of user IDs, day IDs or time IDs are out of their respective ranges

`MissingDataException`
→ occurs when data is not complete, during both, loading and dumping processes.

`MissingFileException`
→ occurs when a raw-values file is missing.

`NullArgumentException`
→ occurs when null pointer is used in a query for the hash map of class Loader.

Example:

```
try {
	HashMap<Integer, LinkedHashMap<Integer, LinkedHashMap<Integer, Double>>>
		rawMap = loader.exportRawValues();

	HashMap<Integer, LinkedHashMap<Integer, LinkedHashMap<Integer, Double>>>
		outputMap = Dumper.initOutputMap();

	for(Integer user : loader.getSortedUsers()) {
		for(Integer day : loader.getSortedDays(user)) {
			try {
				ArrayList<Double> rawValues =
					loader.getSortedRawValues(user, day);
				ArrayList<Double> summarizedValues =
					this.summarize(rawValues);

				for(int time = Tags.minTime, i = 0; time <=
						Tags.maxTime; time++, i++) {
					outputMap.get(user).get(day).put(time,
						summarizedValues.get(i));
				}

			} catch(Exception e) {
				e.printStackTrace();
			}
		}
	}

	dumper.dump(outputMap);

} catch (IllegalHashMapArgumentException e) {
	e.printStackTrace();
} catch (NullArgumentException e) {
	e.printStackTrace();
} catch (MissingDataException e) {
	e.printStackTrace();
} catch (MissingFileException e) {
	e.printStackTrace();
}
```

In the example given, first we load the raw values into memory and obtain the reference to the hash map with all raw values by using `loader.exportRawValues()`. Then, hash map for output data is created and initialized by invoking `Dumper.initOutputMap()`.

We process data by:

- iterating through the user IDs (`loader.getSortedUsers()`) and day IDs (`loader.getSortedDays(user)`) and obtaining all measurements for specific day (`loader.getSortedRawValues(user, day)`)
- applying summarization algorithm to the raw values, and receiving `summarizedValues` as a result
- filling in the output hash map only with time ID and summarized values (`outputMap.get(user).get(day).put(time, summarizedValues.get(i))`).

In the end, we write output data using `dumper.dump(outputMap)`.

The implementation of the summarizing algorithm is given below.

```
private ArrayList<Double> summarize
      (ArrayList<Double> raws) throws Exception {
            FastVector atts = new FastVector();
            atts.addElement(new Attribute("SmartMeter"));

            Instances kmeans = new Instances("kmeans", atts, 0);
            for(Double raw : raws) {
                  double vals[] = new double[kmeans.numAttributes()];
                  vals[0] = raw;
                  kmeans.add(new Instance(1.0, vals));
            }

            SimpleKMeans kMeansAlgo = new SimpleKMeans();
            kMeansAlgo.setSeed(10);
            kMeansAlgo.setPreserveInstancesOrder(true);
            kMeansAlgo.setNumClusters(numOfClusters);
            kMeansAlgo.buildClusterer(kmeans);

            int[] assignments = kMeansAlgo.getAssignments();
            Instances centroids = kMeansAlgo.getClusterCentroids();

            ArrayList<Double> centroidList = new ArrayList<Double>();
            for(int i = 0; i < assignments.length; i++) {
                  double rawValue = kmeans.instance(i).value(0);
                  double centroidValue =
                        centroids.instance(assignments[i]).value(0);
                  centroidList.add(centroidValue);
            }

            return centroidList;
      }
```

The method uses WEKA[1] implementation of the k-Means clustering algorithm, as an example of summarizing data. The method first creates the `Instances` object and feeds it with raw values. `SimpleKMeans` applies k-Means algorithm on the raw data from the `Instances` object. Furthermore, it matches each raw value to the summarized (here called cluster centroid) value with the closest Euclidean distance – array `assignments` stores identification of cluster centroid value for a corresponding raw value. In the end, a list containing centroid values is returned. Centroid values are sorted in the same order as their corresponding raw values provided in a list forwarded as the argument.

1 - http://www.cs.waikato.ac.nz/ml/weka/downloading.html

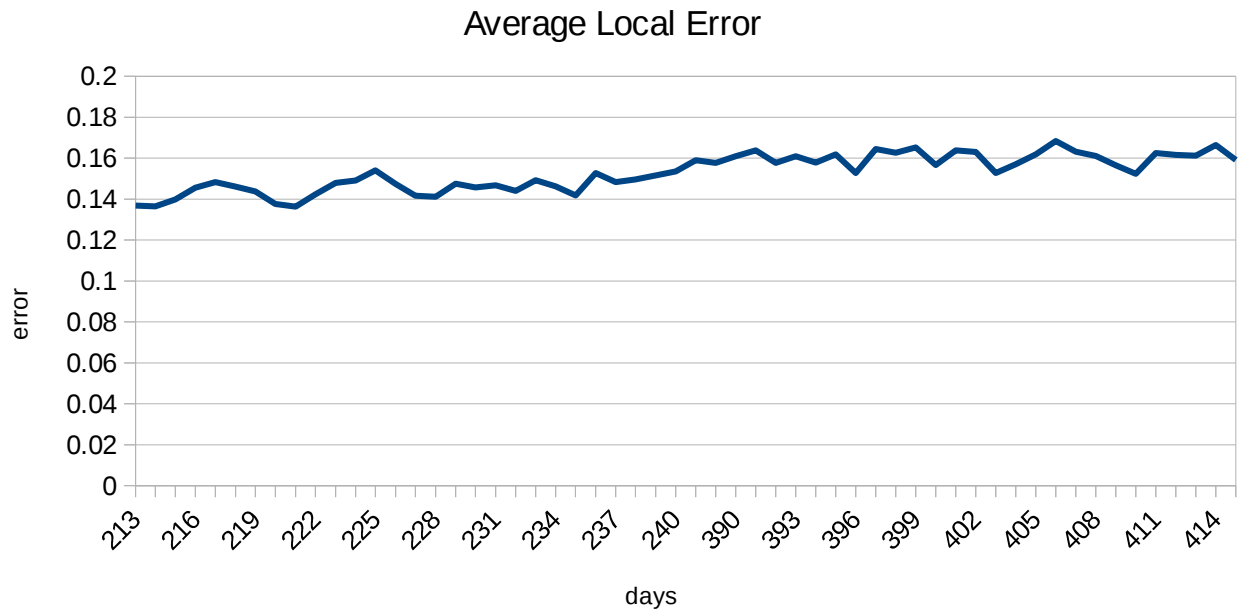
Evaluation:

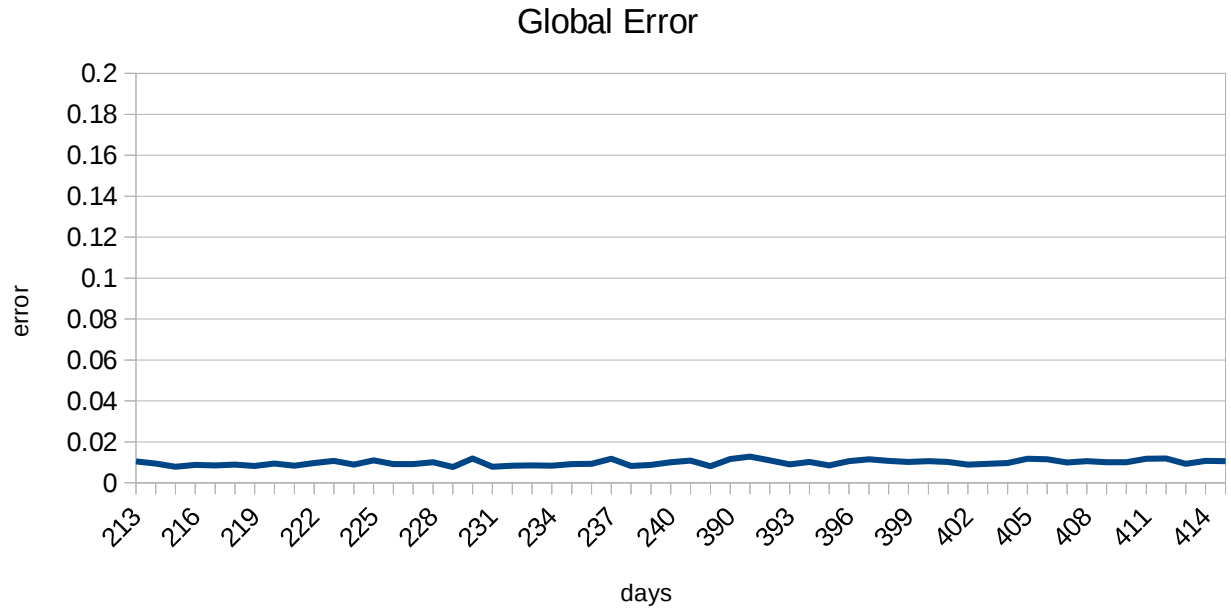
In order to evaluate performance of the summarization algorithm, we calculate the following metrics:

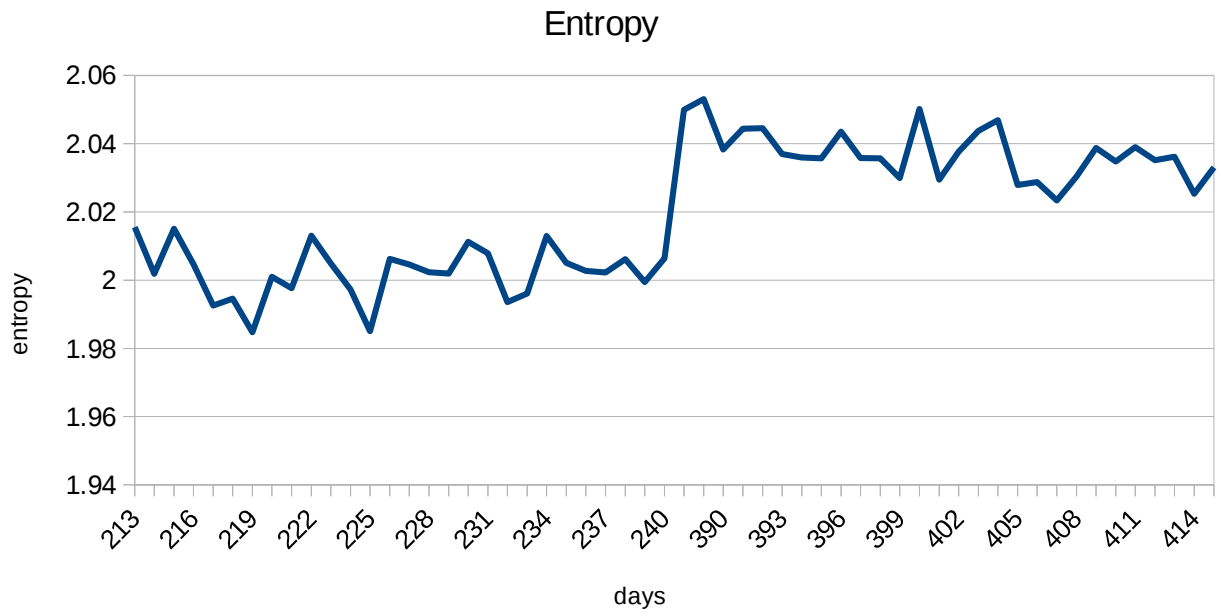
- *Average local error* → relative error between raw and summarized value, averaged across all the users in certain time point
- *Global error* → relative error between summation of raw and summarized values of all the users in certain time point.
- *Entropy* → entropy of summarized values per day
- *Diversity* → rate of change of summarized values per day

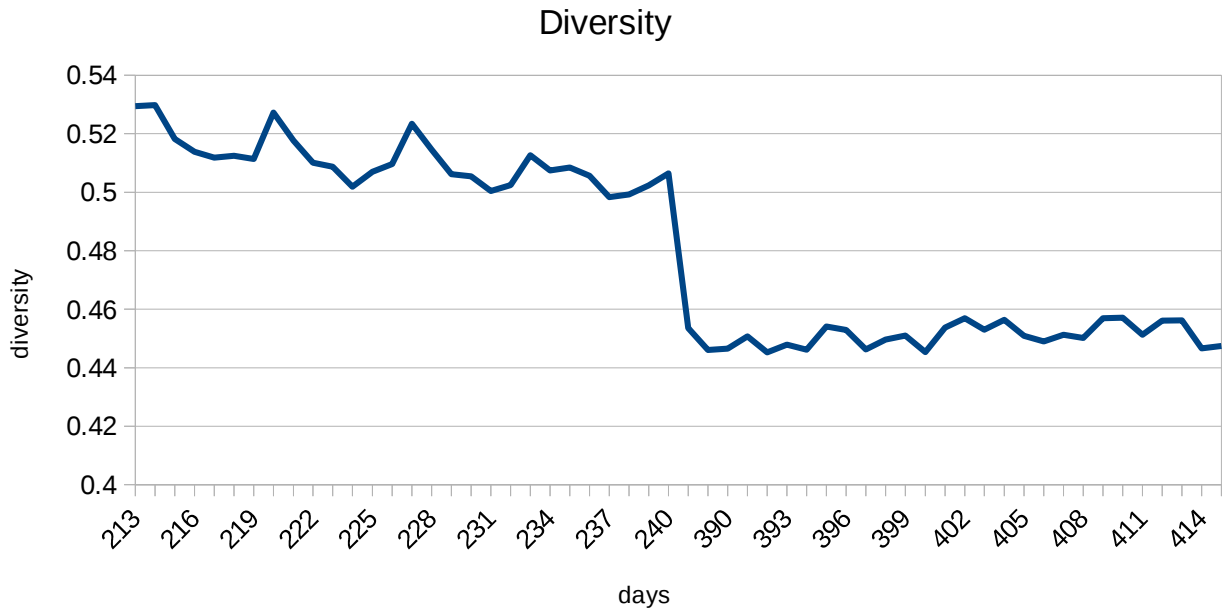
The challenge of the summarization algorithm is in finding equilibrium between two opposing requirements – preserving data privacy and maximizing accuracy of data analytics. Privacy preservation is maximized when entropy and diversity are minimized. On the other hand, data analytics accuracy is maximized when global and average local errors are minimized.

In the example above, the errors, entropy and diversity mainly depend on number of clusters chosen for the k-Means algorithm. The evaluation results of WEKA's implementation of the k-Means algorithm with 5 clusters are given below.




Global Error
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
error
213
216
219
222
225
228
231
234
237
240
390
393
396
399
402
405
408
411
414
days


Entropy
2.06
2.04
2.02
2
1.98
1.96
1.94
entropy
213
216
219
222
225
228
231
234
237
240
390
393
396
399
402
405
408
411
414
days


Diversity
0.54
0.52
0.5
0.48
0.46
0.44
0.42
0.4
diversity
213
216
219
222
225
228
231
234
237
240
390
393
396
399
402
405
408
411
414
days

In the plots above, each metric's dependence of time (days, in this case) is illustrated. The interesting characteristic of the data set provided is that average diversity in summer days is lower compared to winter days. We relate this to several factors, e.g. consumers are taking vacations during summer, some consumers use electricity for additional heating during winter etc.

These metrics will be used to evaluate your algorithm, and will determine your ranking on the leading board. For more info about the leading board, see section The Leadingboard. For more info about how to submit the outcome of your algorithm, check section Output submission.

Web Platform

Our hackathon web platform enables downloading all necessary data for the hackathon. Also, each team must upload the output of its algorithm using this platform. The platform also contains leading board – 4 different boards that rank the performance of your algorithm among other teams' algorithms, each for different metrics (for more information about metrics, see ***Evaluation*** section).

In the beginning of the hackathon, you and your team must sign in on our web platform. The URL (http://hackathon.inn.ac/) will be available during the hackathon.

Output submission

In order to evaluate performance of your algorithm, your algorithm's output should be uploaded to the web platform. If you and your team used the library provided, class Dumper will create output directory with all the necessary files.

To submit an entry, follow these steps:

1. ***The "output" directory must be compressed in ZIP format***. Make sure that "output" directory itself is compressed, and not all files separately. The directory structure that zip file should have is shown in the following example:

```
output.zip
     |-- output
          |-- user_1015.txt
          |-- user_1018.txt
          |-- user_1021.txt
          |-- ...
          |-- user_7429.txt
          |-- user_7432.txt
          |-- user_7434.txt
```

2. This zip file needs to be uploaded through the submission route (http://hackathon.inn.ac/submission/) in order to be analyzed and ranked. The submission page is given in the *Illustration 1*.

Each team can upload zip file multiple times. The previously uploaded version will be deleted, and only the most recent one will be kept on the server. The last version will also be used for ranking your team on the leading board. Your submission history can be reviewed on the page shown in the *Illustration 2*.

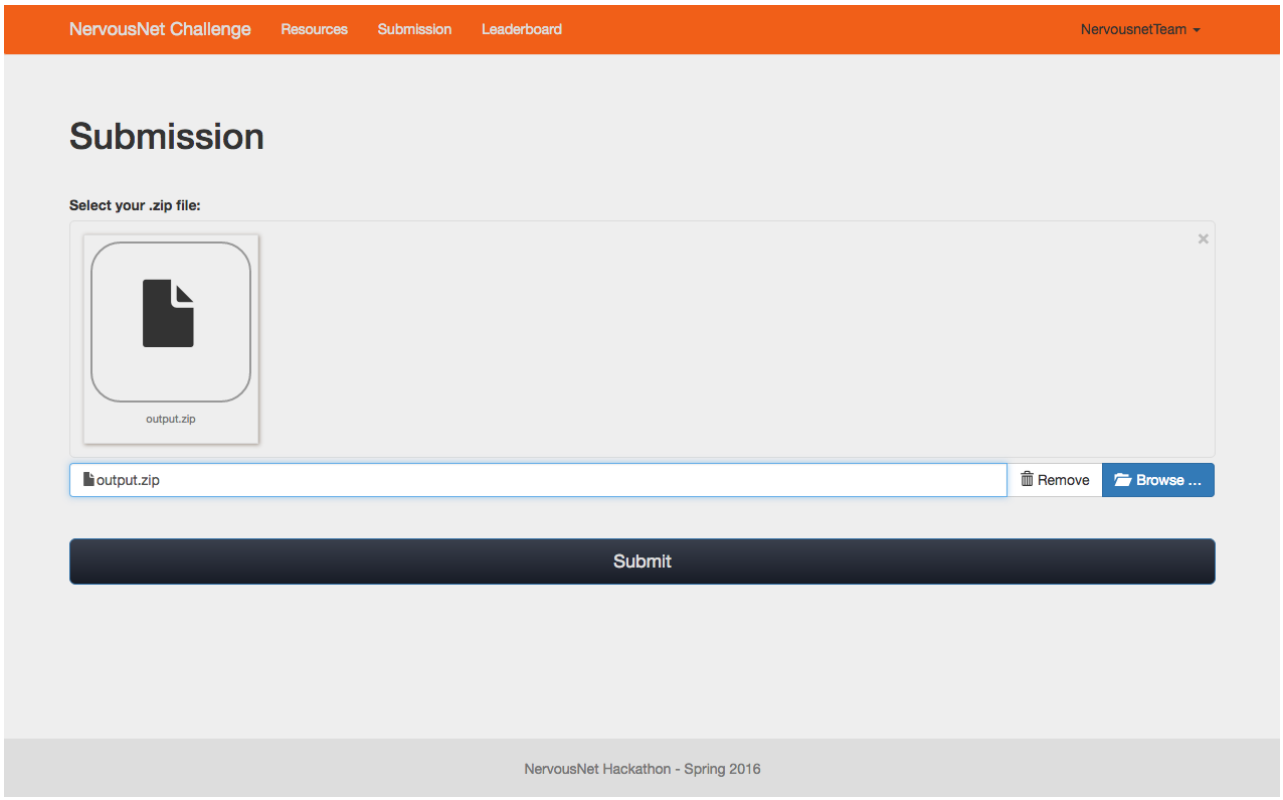


Illustration 1: Submission Page

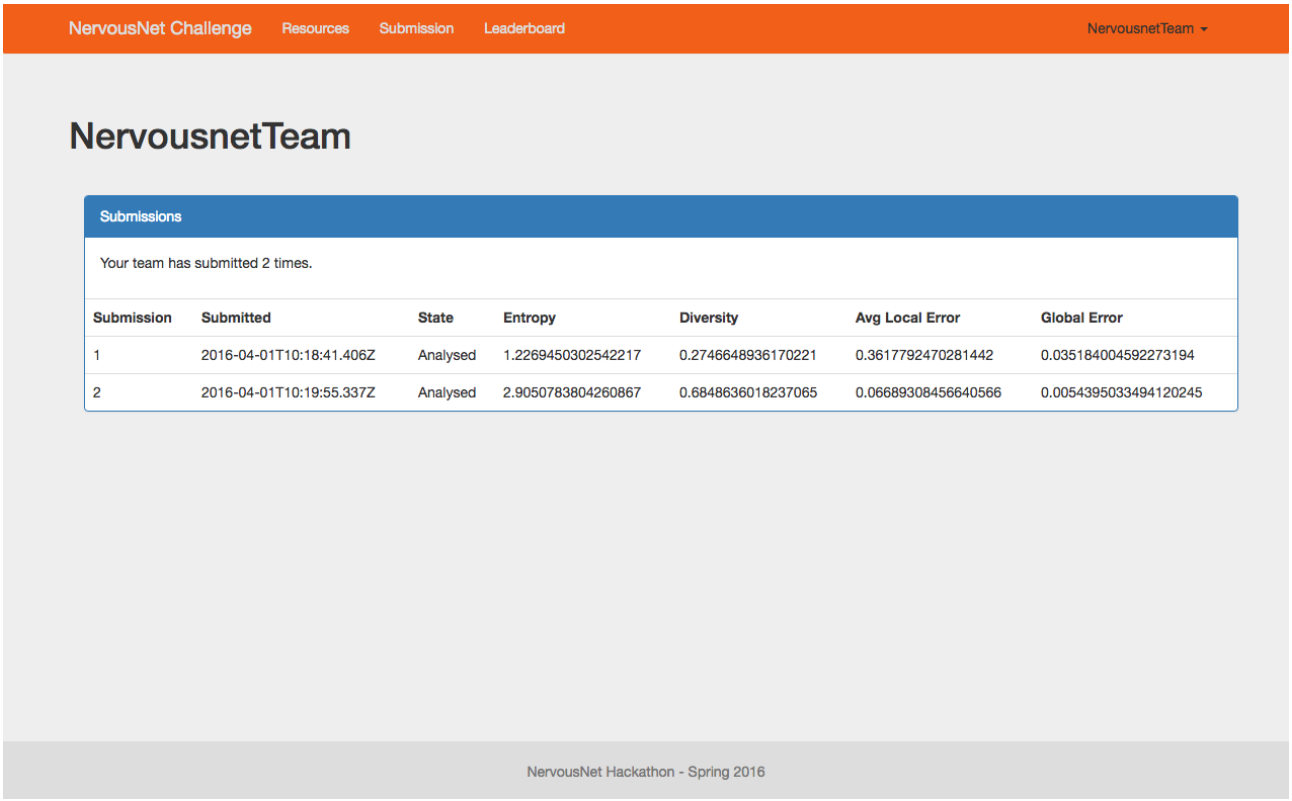


Illustration 2: Submission History

The Leaderboard

The leaderboard can be found on the URL ***http://hackathon.inn.ac/leaderboard/***. The preview of the leaderboard is given in the *Illustration 3*.

After the submission, your output files will be analyzed, and your team will be ranked on the leaderboard accordingly. If you are not satisfied with performance of your algorithm, you are allowed to upgrade it, and to upload new version of the algorithm to the server.

… to be continued