ALMA MATER STUDIORUM - UNIVERSITA DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica - Scienza e Ingegneria - DISI
Corso di Laurea Magistrale in Ingegneria Informatica

PROJECT WORK

on

DATA MINING

A solution for the LANL Earthquake Prediction challenge in Python

CANDIDATES
Valentina Protti
Giuseppe Tempesta

PROFESSOR
Prof. Claudio Sartori

Abstract

This project activity report is intended to explain the approach used in solving a Data Mining competition held on the *Kaggle* platform. In particular, the competition goes under the name "*LANL Earthquake Prediction*", and the major issue that the participants are asked to solve is to predict the time remaining before laboratory earthquakes occur, given real-time seismic data. The challenge is hosted by *Los Alamos National Laboratory* and has its ultimate goal in having the possibility to scale the results to the field, to be finally able to improve real earthquakes predictions.

The work hereby presented has its roots in Los Alamos' initial work, a first model built on laboratory experimental data. With reference to the initial data, the dataset provided for the challenge contains much more aperiodic occurrences of earthquakes, making it more realistic and comparable to real world occurrences.

The report will present the reader with an in-depth analysis of the problem and the provided data, followed by a first naïve approach to better understand the nature of the problem, and finally a comparison of the performances of various techniques for modeling the specific problem.

Contents

List of Figures

Chapter 1

Understanding the problem

Due to the huge impact of their consequences, the pursuit of forecasting earthquakes is one of the most important problems in Earth science. Studies that have been made so far focus on three key points: when, where and how large the event will be.

1.1 Previous studies

But how are these prediction achieved? Los Alamos National Laboratory has conducted a study on huge sets of laboratory experimental seismic data, showing the importance of the so called "slow earthquakes", which are still less understood. In their work [5], the researchers try to spark some light on the mechanics of slow-slip phenomena and their relationship with regular earthquakes, to which they seem to be precursors, through a complete systematic experimental study.

A second study, based on the results of laboratory experiments, takes advantages of Machine Learning techniques to predict the time to the next "labquake" by listening to the acoustic signal collected by specific laboratory sensors [10]. By using ML, even small seismic precursor magnitude can be detected, overcoming the limits of classic seismograph-based predicting systems. In particular, a Random Forest approach has been developed to predict the time remaining before the next failure, by averaging the predictions of 1,000 decision trees in each time window.

From each time window, a set of approximately 100 statistical features are computed, then selected recursively by usefulness, and lastly used to

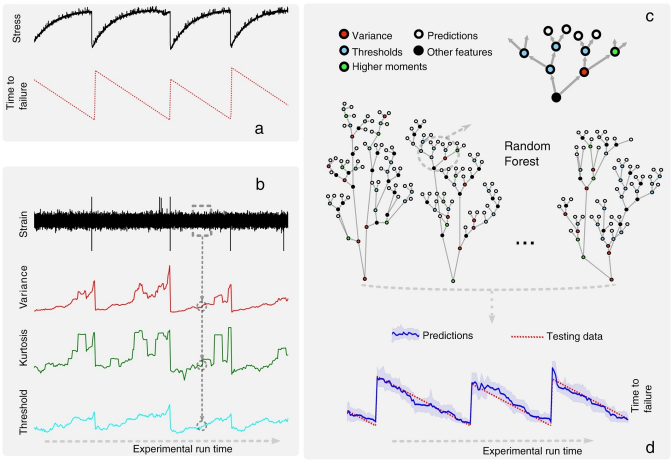

Figure 1.1: Random Forest (RF) approach for predicting time remaining before failure.

actually predict the time before the next earthquake. The results achieved through this study are quite accurate, even if it needs to be noted that a laboratory earthquake does not capture the physics of a complex, real-world earthquake.

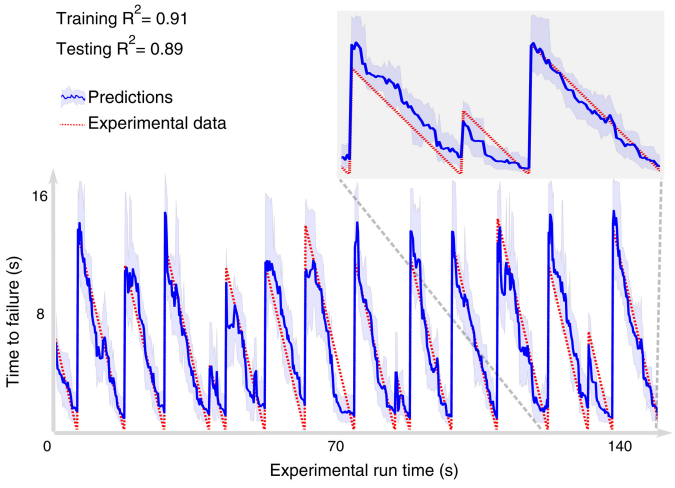

Figure 1.2: Time remaining before the next failure predicted by the Random Forest.

1.2 Understanding the data

With reference to the above mentioned studies, the dataset provided for the challenge contains more a-periodic occurrences of earthquake hazards, thus resembling more a real-world scenario. The data comes from a well-known experimental set-up used to study earthquake physics [1].

In particular, the dataset is made of two subsets:

- `train.csv` - A single, continuous training segment of experimental data (with 629.145.480 entries);
- `test` - A folder containing many small segments (`.csv`) of test data (2.624 segments of 150.000 entries each).

Each entry of the training set has two fields:

- `acoustic_data` - the seismic signal [`int16`];
- `time_to_failure` - the time (in seconds) until the next laboratory earthquake [`float64`].

On the other hand, each segment from the test set folder is named after its `seg_id` and only has one field, the `acoustic_data`. While the training set is a single, continuous, big segment of experimental data, the test set is continuous within a single segment, but the set of files cannot be considered continuous; thus, the predictions can't be assumed to follow the same pattern of the training file.

The goal of the competition is to predict a single `time_to_failure` for each segment, corresponding to the time between the last row of the segment and the next laboratory earthquake. The results must be submitted on the Kaggle platform as a `.csv` file containing the predictions for each test segment, and the score is then obtained through the application of the *Mean Absolute Error* between the real time values and the predictions.

A first approach to better understand what the data represents is to plot it (or a part of it, given the prohibitive size of the training set). In picture 1.3 we can see 1% of the training data (obtained simply by sampling every 100 entries) [9].

The following (1.4) is instead the representation of the first 1% entries of the training dataset: even at a first glance we are able to note that the

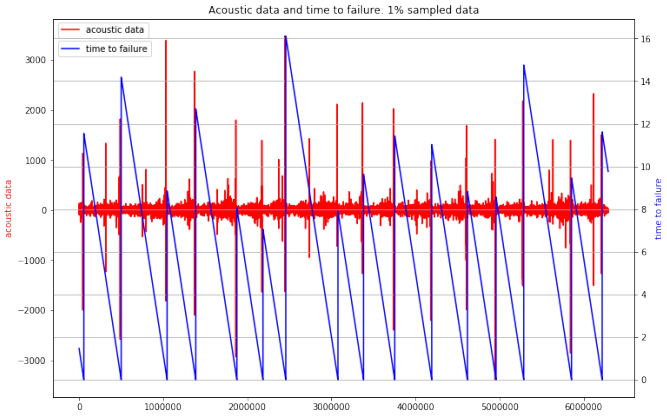

Figure 1.3: Plot of 1% sampling of the training data

failure ("*labquake*") occurs after some medium oscillations, a very large one and some other minor ones.

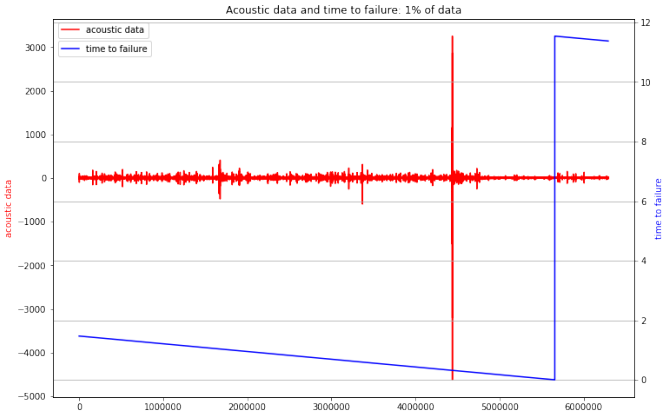

Figure 1.4: Plot of the first 1% of the training data

Before going into further details and taking a first step towards building the model from the training data, it's worth to also take a look at the structure of the test data. In picture 1.5 are represented four of the segments from the test folder [4].

Overall, what we can take away from this first dive into the datasets is:

- that the task of this Data Mining challenge will be in the regression spectrum, since the output falls in a continuous range rather than a set of discrete classes;

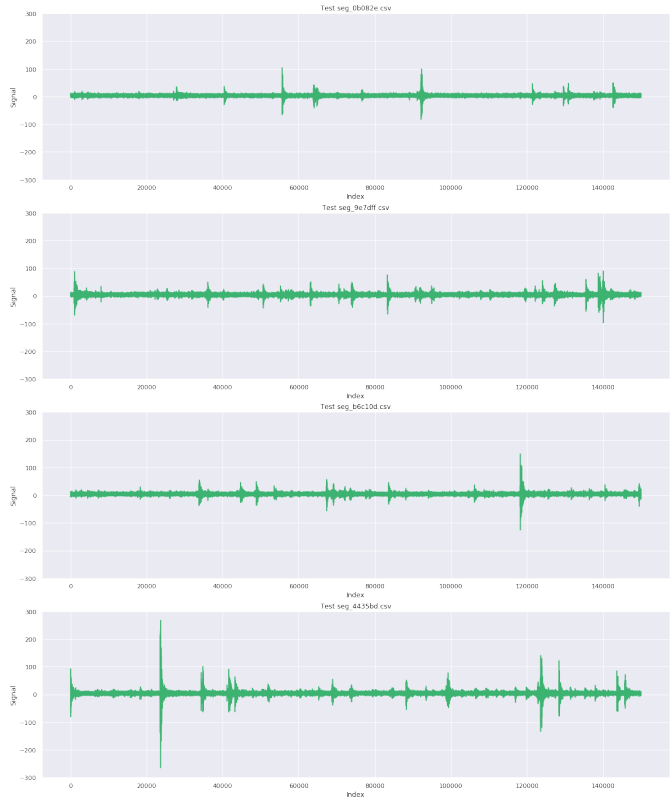

Figure 1.5: Plot of four segments of the test data

- that the dimension of the segments is not that big if compared to the very rare occurrences of laboratory earthquakes;
- that these failure events will appear very much like outliers, given the scarcity of representation and the intensity of the acoustic signal when compared to the other values.

As others participants to the challenge have noticed, it may be also relevant to note that the test set doesn't contain *any* earthquake: thus, it may be worth considering not including the few failure occurrences that can be found in the training set, to avoid the model trying to match the data to these much higher peaks when fed with the test set.

Chapter 2

A first, naïve model

A first, very simple approach to building the model for the purpose of this competition is given directly by the promoters of the challenge [7].

Before seeing what the *Python kernel* looks like, it's worth to dig a bit deeper on how the data is prepared for the task. In fact, taking the dataset "as it is", it's easy to notice that it has just one feature (`acoustic_data`) that can be used to compute the regression task of predicting `time_to_failure` on the test set.

For this reason, data needs to be prepared: the obvious choice is to divide the training set into chunks of 150.000 rows (the size of each segment of the test set; this is not the only choice available), and for each of them compute some features representing the data; in this first basic solution we will extract the mean, standard deviation, maximum and minimum. The resulting dataset will contain an entry for each portion of the initial dataset, with one column for each computed feature (in this case 4), and another dataset with just the original `time_to_failure` associated with the last row of the chunk (similarly to the test segments).

2.1 Basic Feature Benchmark

```
1 # This Python 3 environment comes with many helpful analytics libraries installed
2 # It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-python
3 # For example, here's several helpful packages to load in
4
5 import numpy as np # linear algebra
6 import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
7
8 # Input data files are available in the "../input/" directory.
```

```
# For example, running this (by clicking run or pressing Shift+Enter) will list the files in the
      input directory

import os
print(os.listdir("../input"))

# Any results you write to the current directory are saved as output.
```

```
import matplotlib.pyplot as plt
from tqdm import tqdm
from sklearn.preprocessing import StandardScaler
from sklearn.svm import NuSVR
from sklearn.metrics import mean_absolute_error
```

```
train = pd.read_csv('../input/train.csv', dtype={'acoustic_data': np.int16, 'time_to_failure':
     np.float64})
```

After these preliminary operations of including the necessary libraries and loading the dataset, we are able to get into the data preparation as previously described. In the following snippet, `X_train` is the dataset containing the segments' 4 computed features, while `Y_train` contains the associated `time_to_failure`.

```
# Create a training file with simple derived features

rows = 150_000
segments = int(np.floor(train.shape[0] / rows))

X_train = pd.DataFrame(index=range(segments), dtype=np.float64,
                       columns=['ave', 'std', 'max', 'min'])
y_train = pd.DataFrame(index=range(segments), dtype=np.float64,
                       columns=['time_to_failure'])

for segment in tqdm(range(segments)):
    seg = train.iloc[segment*rows:segment*rows+rows]
    x = seg['acoustic_data'].values
    y = seg['time_to_failure'].values[-1]

    y_train.loc[segment, 'time_to_failure'] = y

    X_train.loc[segment, 'ave'] = x.mean()
    X_train.loc[segment, 'std'] = x.std()
    X_train.loc[segment, 'max'] = x.max()
    X_train.loc[segment, 'min'] = x.min()
```

The kernel's authors' choice for modeling the solution is to use *Support Vector Regression*. The `scikit-learn` Python library implementation of SVR recommends explicitly that the data is scaled, since Support Vector Machine algorithms are not scale invariant.

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
```

```
svm = NuSVR()
svm.fit(X_train_scaled, y_train.values.flatten())
y_pred = svm.predict(X_train_scaled)
```

The results predicted are then compared with the actual training data, by computing the *Mean Absolute Error*, and finally the test data is prepared and fed to the model and results are printed to the `submission.csv` file. As expected, this simple model has a score of 2,314 on the training set (the score on the test set can only be computed by the competition's promoters), which denotes a really bad performance 2.1.

```
48 score = mean_absolute_error(y_train.values.flatten(), y_pred)
49 print(f'Score: {score:0.3f}')
```

```
50 submission = pd.read_csv('../input/sample_submission.csv', index_col='seg_id')
51 X_test = pd.DataFrame(columns=X_train.columns, dtype=np.float64, index=submission.index)
```

```
52 for seg_id in X_test.index:
53     seg = pd.read_csv('../input/test/' + seg_id + '.csv')
54
55     x = seg['acoustic_data'].values
56
57     X_test.loc[seg_id, 'ave'] = x.mean()
58     X_test.loc[seg_id, 'std'] = x.std()
59     X_test.loc[seg_id, 'max'] = x.max()
60     X_test.loc[seg_id, 'min'] = x.min()
```

```
61 X_test_scaled = scaler.transform(X_test)
62 submission['time_to_failure'] = svm.predict(X_test_scaled)
63 submission.to_csv('submission.csv')
```

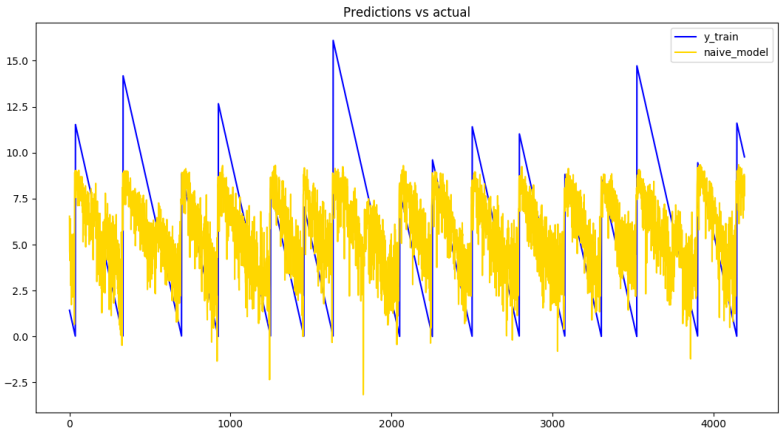

Figure 2.1: Naïve model, `time_to_failure` predictions vs real values.

Chapter 3

More advanced models comparison and issues

Starting from the basic model presented in the previous section, for the purpose of this project activity we theorized and tested various approaches to solving the problem.

3.1 First model: adding more features with Linear Regression

In order to make a step forward in the development of the regression model, our first attempt was to simply check out the performances obtained by a simple model with a few more features than the basic ones. In the attempt to study the impact of different features on the results, our choice for the model was the simple `Linear Regression` (implemented in `scikit-learn`), applied on a set of 12 features.

```
35 # Create a training file with simple derived features
36
37 rows = 150000
38 segments = int( np.floor(train.shape[0]) / rows)
39
40 X_train = pd.DataFrame(index=range(segments), dtype=np.float64,
41                        columns=['ave', 'std', 'max', 'min', 'mad', 'kurt', 'skew', 'median', 'q01',
                                'q05', 'q95', 'q99'])
42
43 y_train = pd.DataFrame(index=range(segments), dtype=np.float64,
44                        columns=['time_to_failure'])
45
46 for segment in tqdm(range(segments)):
47     seg = train.iloc[segment*rows:segment*rows+rows]
48
49
50     x = seg['acoustic_data']
```

```
51      y = seg['time_to_failure'].values[-1]
52
53      y_train.loc[segment, 'time_to_failure'] = y
54
55      X_train.loc[segment, 'ave'] = x.mean()
56      X_train.loc[segment, 'std'] = x.std()
57      X_train.loc[segment, 'max'] = x.max()
58      X_train.loc[segment, 'min'] = x.min()
59      X_train.loc[segment, 'mad'] = x.mad()
60      X_train.loc[segment, 'kurt'] = kurtosis(x)
61      X_train.loc[segment, 'skew'] = skew(x)
62      X_train.loc[segment, 'median'] = x.median()
63      X_train.loc[segment, 'q01'] = np.quantile(x, 0.01)
64      X_train.loc[segment, 'q05'] = np.quantile(x, 0.05)
65      X_train.loc[segment, 'q95'] = np.quantile(x, 0.95)
66      X_train.loc[segment, 'q99'] = np.quantile(x, 0.99)
```

In particular, we chose quite a standard set of features to add to the first four: `MAD` returns the Mean Absolute Deviation on the values (its accuracy is closely related to the Mean Squared Error, or `MSE`); *kurtosis* is a measure of the "tailedness" (or the shape) of the probability distribution of a real-valued random variable, calculated as the fourth standardized moment; *skewness* is a measure of the asymmetry of the probability distribution of a real-valued random variable, calculated as the third standardized moment; the *median* is the value separating the higher half from the lower half of the data; the *q-th quantiles* are cut points dividing the range of a probability distribution into intervals with the same probability: x is a q-th quantile for a variable X if $Pr[X < x] \leq q$.

The computed score of the so constructed model improves, even if slightly, the results of the naïve model, with a value of 2,251.

At this point, out of curiosity we took a look at the predicted `time_to_failure` resulting from the test data, and we noticed that there was a considerable number of negative values, evidently wrong (it should be remembered that they represent the time between the current segment and the next laboratory earthquake, which cannot be negative quantities).

Based on that observation, and given that the nature of the data is approximately symmetrical (see figure 1.4), we though about introducing a whole new set of features generated by the same computational functions applied on the absolute values of the dataset.

```
39  X_train = pd.DataFrame(index=range(segments), dtype=np.float64,
40                         columns=['ave', 'std', 'max', 'min', 'mad', 'kurt', 'skew', 'median', 'q01',
                                   'q05', 'q95', 'q99', 'abs_mean', 'abs_std', 'abs_max', 'abs_min', '
                                  abs_mad', 'abs_kurt', 'abs_skew', 'abs_median', 'abs_q01', 'abs_q05',
                                  'abs_q95', 'abs_q99'])
```

```
66      [...]
```

```
67      X_train.loc[segment, 'abs_mean'] = x.abs().mean()
68      X_train.loc[segment, 'abs_std'] = x.abs().std()
69      X_train.loc[segment, 'abs_max'] = x.abs().max()
70      X_train.loc[segment, 'abs_min'] = x.abs().min()
71      X_train.loc[segment, 'abs_mad'] = x.abs().mad()
72      X_train.loc[segment, 'abs_kurt'] = kurtosis(x.abs())
73      X_train.loc[segment, 'abs_skew'] = skew(x.abs())
74      X_train.loc[segment, 'abs_median'] = x.abs().median()
75      X_train.loc[segment, 'abs_q01'] = np.quantile(x.abs(), 0.01)
76      X_train.loc[segment, 'abs_q05'] = np.quantile(x.abs(), 0.05)
77      X_train.loc[segment, 'abs_q95'] = np.quantile(x.abs(), 0.95)
78      X_train.loc[segment, 'abs_q99'] = np.quantile(x.abs(), 0.99)
```

The results obtained in terms of score and predictions using this set of 24 values entailed another slight improvement, giving a value of 2,097 for the mean absolute error (our score), and fewer negative predictions in `submission.csv`.

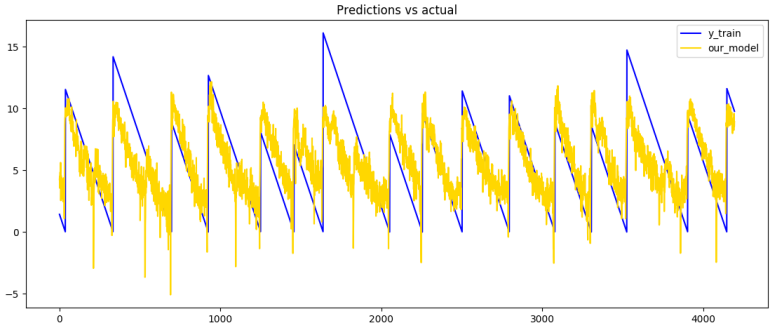

Figure 3.1: Linear Regression, 24 features, `time_to_failure` predictions vs real values.

After submitting our results to Kaggle's platform, the score of this kernel calculated by the system was **1,660**. Other attempts to improve this solution by adding more features were unsatisfactory.

3.2 Second model: Support Vector Regression on more features

After applying linear regression to an expanded set of features, the following choice was for us to merge the intuition of using more than the basic features while relying on the Support Vector Regression as for the naïve solution. The same 24 features described in 3.1 were used, while our exploration in this type of model was mainly focused on the choice of the kernel function (chosen through a parameter defined in the implementation of `NuSVR`

inside `scikit-learn`). It must also be remembered that, as stated in the documentation, this model needs data to be previously scaled (otherwise performances decrease considerably).

The choice of the kernel function in this case is crucial: a *kernel function* makes it possible to transform data from a n-dimensional space (in our case, defined by the set of computed features) into another space with reduced dimensions, in order to find a more clear dividing margin between classes ("*kernel trick*"). There are of course different choices for the kernel function: radial basis function, linear, polynomial, just to name some of them.

Results of changing the `kernel` parameter were the following, in terms of score on the training set:

- `rbf` or *radial basis function*, default: 2,1038;
- `linear`: 2,1638 (also, quite a few negative results for the submission file);
- `poly` or *polynomial*: 2,5109861 (even more negative results).

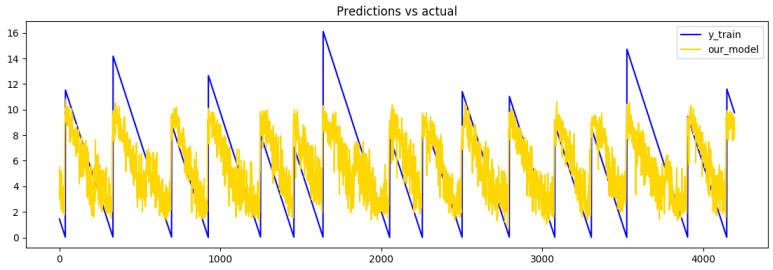

Figure 3.2: Support Vector Regression with `rbf`, 24 features, `time_to_failure` predictions vs real values.

After submitting our results to Kaggle's platform, the score of this kernel calculated by the system was **1,609**.

3.3 Introducing preliminary feature selection

Due to the high dimensionality of the dataset, the computations made on it take several minutes to be completed. In order to boost performances and with the goal to improve the accuracy of the models, our next attempt was to introduce a preemptive feature selection.

Feature selection can be seen as a pre-processing step, that works by selecting the best features on the basis of some specified tests.

`Scikit-learn` exposes some choices for this purpose, among which `SelectKBest` and `SelectPercentile` (the former removing all but the k highest scoring features, the latter keeping the specified percentage of highest scoring features). The scoring functions available for regression purposes are `f_regression` (being able to capture linear correlations between features) and `mutual_info_regression` (which underlines some more particular correlations). Our choices for the solution were to use `SelectKBest` with `mutual_info_regression` to select the best 10 features, to be tested afterwards with our most promising model up until now, `NuSVR` 3.2.

The 10 selected features were the ones in image 3.3.

```
100%|██████████| 4194/4194 [02:58<00:00, 22.4lit/s]
                ave                 std  ...  abs_q95  abs_q99
0  4.884113333333334  5.101106130606765  ...     12.0     20.0
1  4.725766666666667  6.588823781946801  ...     12.0     24.0
2  4.906393333333333  6.967397033524431  ...     14.0     30.0
3  4.902240000000000  6.922305187180113  ...     13.0     26.0
4  4.908720000000000  7.301110189758817  ...     13.0     32.0

[5 rows x 24 columns]
['std', 'mad', 'q01', 'q05', 'q95', 'q99', 'abs_mean', 'abs_mad',
 'abs_q95', 'abs_q99']
```

Figure 3.3: 10 features selected from `SelectKBest`.

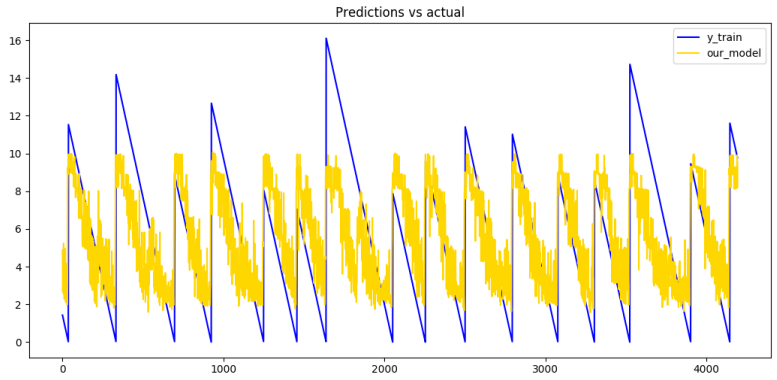

Figure 3.4: Support Vector Regression with `rbf`, 10 selected features, `time_to_failure` predictions vs real values.

The score obtained on the training set with just the above mentioned features was 2,1138, which would seem not to improve the one obtained in 3.2 without feature selection; after submitting our results to Kaggle's platform though, the score of this kernel calculated by the system was **1,588**.

3.4 Third model: Neural Network Regression

Artificial Neural Networks are computing systems inspired by biological neural networks, that learn to perform tasks by taking examples as input, generally without any task-specific rule. In particular, a MultiLayer Perceptron (MLP) is made at least of 3 layers: input, output and one or more hidden layers (neurons that use non-linear activation functions); thanks to the complexity, MLP can distinguish non-linearly separable data.

Implementation of MLP Regressor in `scikit-learn` has a huge set of parameters to define in order to be able to refine the model. Our exploration focused on some of them:

- `hidden_layer_sizes`, number of neurons in each hidden layer;
- `max_iter`, maximum number of iterations;
- `solver`; the default is `adam`, a stochastic gradient-based optimizer that works well on large datasets, but other options are `sgd`, a stochastic gradient descent, and `lbfgs`, from the family of quasi-Newton methods;
- `warm_start`, a boolean option that enables or disables initialization of the next call with the previous solution.

Various attempts on so-constructed models showed that scaling is highly recommended for performance reasons.

Picture 3.5 is a synthetic representation of the different parameters settings we had the chance to test. Other than the described parameters, we also diversified the number of features used as input for the models; the results shown are computed on the training set by us, and on a portion of the test set by submitting the solutions to Kaggle's platform. What can be inferred from them is that the solution seem to work well on a medium to large-sized set of features (24 to 60) and on a smaller number of hidden layer neurons.

features	hidden_layer_size	solver	warm_start	SCORE train set	SCORE test set (13% on Kaggle)
10 (selected)	200	Adam	False	2.0756	-
10 (selected)	2000	Adam	False	2.0894	-
24	400	Adam	True	1.7248	1.887
24	10.000	Adam	True	1.0086	2.271
60	200	Adam	False	1.9211	1.598
60	2000	Adam	True	1.8528	1.680

Figure 3.5: Attempts and results on different parameters settings.

Aside from these tries, the most successful attempt with a score of 2,0015 on the training set had the following settings:

- number of features: 24;
- `hidden_layer_sizes`: 1000;
- `max_iter`: 2500;
- `solver`: sgd;
- `warm_start`: false.

Other settings in the following snippet are to print progress messages (`verbose`), to set the tolerance of the loss score between consecutive iterations (`tol`) and not to shuffle samples between iterations (`shuffle`).

```
14  from sklearn.neural_network import MLPRegressor
```

```
109 model = MLPRegressor(verbose=True, random_state=10, tol=0.000001, max_iter=2500,
        hidden_layer_sizes=1000, shuffle=False, solver='sgd')
```

After submitting our results to Kaggle's platform, the score of this kernel calculated by the system was **1,540**.

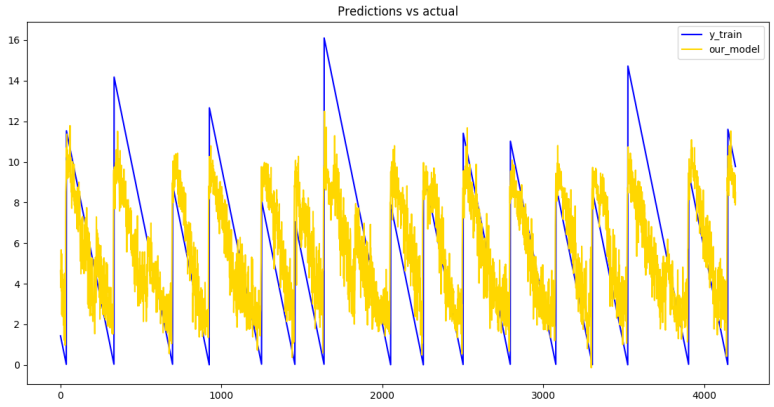

Figure 3.6: Best MLP Regression, `time_to_failure` predictions vs real values.

3.5 The problem of overfitting: Gaussian Processes and Random Forests

Gaussian Processes are a versatile supervised learning method that aims at interpolating the observations and predicting results with some confidence intervals. Similarly to SVR, a kernel function must be specified.

Introducing the regression model based on GPs on our dataset, the results obtained were surprising to say the least: the precision score calculated on the training set was in the order of magnitude of e-10 (basically, the error didn't exist).

```
14  from sklearn.gaussian_process import GaussianProcessRegressor

101 model = GaussianProcessRegressor()
```

Indeed, the graph in 3.7 shows that the predicted values perfectly overlap the real values of `time_to_failure` in the training set (notice that both have been plotted).

However, when submitting our results to Kaggle's platform tempted by the above described performances, the score of this kernel was **2,792**, showing that the built model was indeed overfitted to the training data and performed very badly on the test data.

Almost the same conclusion can be drawn for the *Random Forest Regressor* (and its even more random evolution, *Estremely Randomized Trees*

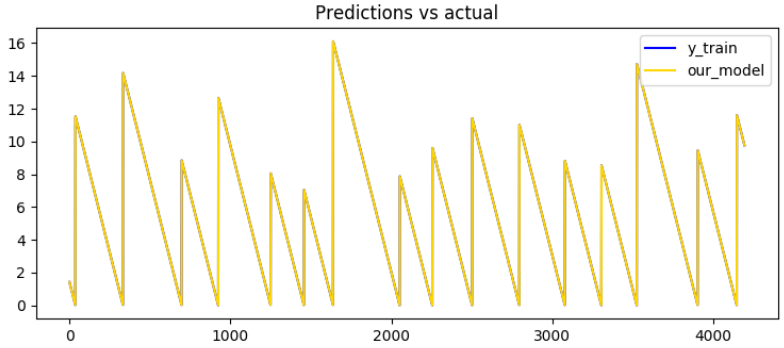

Figure 3.7: Gaussian Process Regression, 24 features, `time_to_failure` predictions vs real values.

Regressor). These are ensemble methods based on decision trees, whose purpose is to combine predictions of several estimators, doing so by building them independently, then averaging the predictions and obtaining a single base estimator with reduced variance and improved robustness.

The mechanism followed by Random Forests is to build a tree by splitting nodes according to the best split among a random subset of features; this randomness gives the model a decrease in variance with respect to a single non-random tree, that instead chooses the split out of all features. In addition, in Extremely Randomized Trees randomness is used also to select the thresholds that generate the splits.

Implementation of `RandomForestRegressor` available in `scikit-learn` takes several parameters as input, among which `n_estimators`, that enables to specify the number of trees in the forest (the larger, the better, but the more intense will be the computation), and `criterion`, the function measuring the quality of a split (the default is `MSE`, which fits well to our situation).

```
14 from sklearn.ensemble import RandomForestRegressor
```

```
115 model = RandomForestRegressor(n_estimators=10000)
116 model.fit(X_train, y_train.values.flatten())
```

Results obtained by computing the score on the available data were very encouraging: only 0,8146 for the training set while using 1000 estimators, and down to 0,7785 with 10000. After submitting our results to Kaggle's platform, though, the score of this kernel (the best case obtained) calculated by the system was **1,687**. Even if much better than GPs' results on test data, the score does not reflect the improvement obtained on the training set,

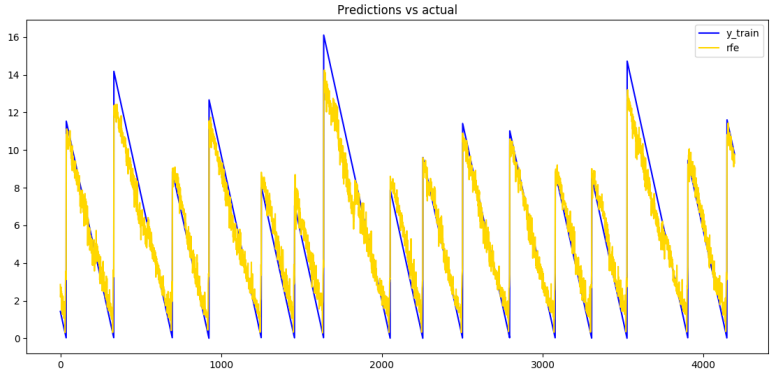

Figure 3.8: Random Forest Regression, 24 features, 10.000 iterations, `time_to_failure` predictions vs real values.

showing also in this case that these more advanced models can be very precise in the building phase, but are prone to overfitting. A further development of this solution could perhaps be obtained through a better, more in-depth knowledge of the setting parameters, which was beyond the purpose of this exploring project activity.

Chapter 4

Infrastructure and tools

Given the nature of the data, even the easiest computation would be really heavy. The sole task of loading `training.csv` file takes around 3 minutes, and the used RAM amount is about 11 GBs. For this reason, the personal computers physically at our disposal were not enough.

4.1 Microsoft Azure Virtual Machines

In order to support heavy computations, we signed a subscription on the Microsoft Azure Platform [8], that provides students with an initial credit of 100$, and therefore the possibility to access (some of) their virtual machines.

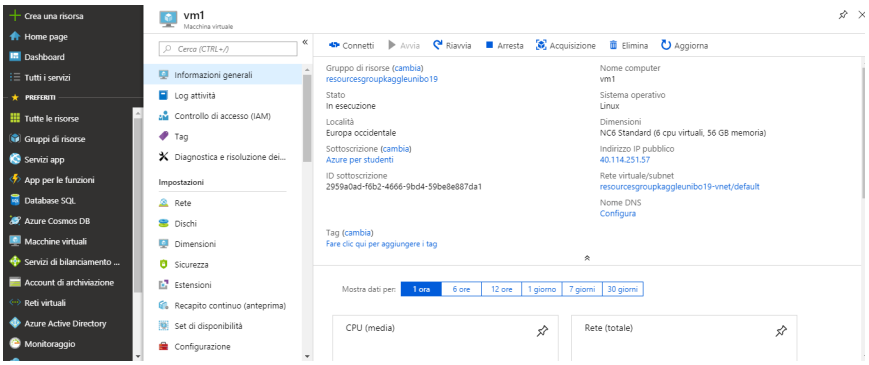

Figure 4.1: Azure Platform screenshot.

According to the limitations given to student accounts, we instantiated the machine with the maximum of cores and GBs of RAM possible, which happened to be the NC6 Standard machine (with 6 virtual CPUs and 56 GBs RAM). Python was already installed on it, and we only had to install the

other libraries (like `scikit-learn`) in order for our scripts to run. Access to the machine was made possible through SSH connection established through the *PuTTY* client installed on our laptops [2]. Graphic visualization of the plots presented in the report were made available through *Xming* [3].

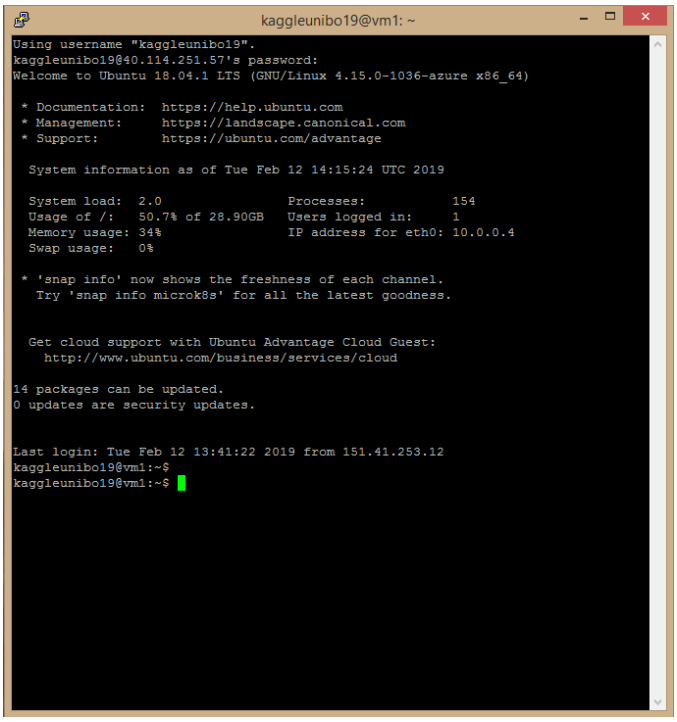

Figure 4.2: Command line with SSH access to the VM.

The whole collection of `.py` scripts explored and the various results obtained have been saved and stored in a repository, available on GitHub [6].

Chapter 5

Conclusions

For this Data Mining project activity we were presented with a simulated seismic data mining problem and the task to predict the time remaining until the next laboratory earthquake; starting from raw data, we pre-processed it by computing aggregated representative data to be fed to a model of our choice. We tested different models from the most used Python library, `scikit-learn`, starting from the most simple ones up until neural networks, and we encountered various issues that we tried to dig deep into with the intention of understanding them and drawing our conclusions. All things considered, it looks like more complex models tend to overfit to the training data, giving a good score on the training set and a much worse one on the test set, while simpler models seem to be more general, giving unsatisfactory results on the training set but performing much better on the test set.

Further development of the presented models could be accomplished by tweaking and refining the parameters settings, a task that we did not pursue due to time restrictions, knowledge boundaries, and to our common decision to focus more on exploring different models and collecting the responses.

Bibliography

[1] LANL Earthquake Prediction. 2019. URL: https://www.kaggle.com/c/LANL-Earthquake-Prediction.

[2] PuTTY. 2019. URL: https://putty.org/.

[3] Xming. 2019. URL: http://www.straightrunning.com/XmingNotes/.

[4] Allunia. Shaking Earth. 2019. URL: https://www.kaggle.com/allunia/shaking-earth.

[5] J. R. Leeman et al. Laboratory observations of slow earthquakes and the spectrum of tectonic fault slip modes. nat. commun. 7:11104 doi: 10.1038/ncomms11104. Technical report. 2016. URL: https://www.nature.com/articles/ncomms11104.

[6] Tempesta Giuseppe and Protti Valentina. GitHub solution repository. 2019. URL: https://github.com/GiusTemp/kaggleunibo.

[7] inversion. Basic Feature Benchmark. 2019. URL: https://www.kaggle.com/inversion/basic-feature-benchmark.

[8] Microsoft. Azure Portal. 2019. URL: https://portal.azure.com/.

[9] Gabriel Preda. LANL Earthquake EDA and Prediction. 2019. URL: https://www.kaggle.com/gpreda/lanl-earthquake-eda-and-prediction.

[10] Bertrand Rouet-Leduc, Claudia Hulbert, Nicholas Lubbers, Kipton Barros, Colin J. Humphreys, and Paul A. Johnson. Machine Learning Predicts Laboratory Earthquakes. Technical report. 2017. URL: https://doi.org/10.1002/2017GL074677.