Empirical Hardness Models:
A case study on the Traveling Salesperson Problem

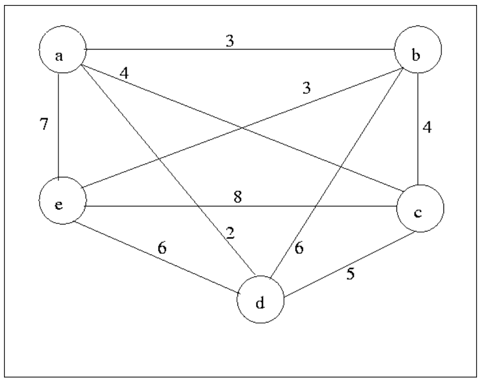
Introductions:

Empirical hardness models

There are quite a few interesting and important computational problems that are NP-hard.
This apparently suggests that these problems are extremely hard to solve for all instances but the smallest, but in practice the situation is often much better. The reason for that is that NP-hardness is only a worst-case notion. Thus, in practice there are many problems in which some instances turn out to be much easier to solve than worst-case instances of the same size.
This phenomenon led to a growing interest in understanding the "Empirical Hardness" (also called "typical-case complexity") of NP-hard problems.
The goal of this research method is to identify the factors of a problem that will dictate how hard some instances of that problem will be for a particular algorithm to solve in practice.
By doing so we hope to speed up the practical development of state of the art algorithms and also enrich the theoretical understanding we have on these problems, as these type of problems are often addressed by using heuristic methods and approximation algorithms.
In this project we tried to implement this research method on the "Traveling Salesperson problem".

Traveling Salesperson problem

The traveling salesperson problem (abbreviated: TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"
It is an NP-hard problem in combinatorial optimization,
The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization in particular and in computer science in general.
This problem has many applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. It is also used as a benchmark for many optimization methods.



There are many variations and many ways to model the TSP. In this project we case studied the classic variation and modelled it as a weighted graph (we represented it using an adjacency matrix), such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once.
In our case (classic variation):
- The graph is an undirected graph.
- The graph is a complete graph, meaning that each pair of vertices is connected by an edge.
- The intercity distances (weights) are positive integers from a bounded range.

The methodology of building Empirical Hardness Models:

The following methodology for predicting the running time of a given algorithm on individual instances of a problem such as TSP, where instances are drawn from some arbitrary distribution was first introduced by Leyton-Brown et al. 2002:

1) Select an algorithm of interest.
A black-box implementation is enough, no knowledge of the algorithm's internal workings in needed.

2) Select an instance distribution.
In practice, this may be achieved as a distribution over different instance generators, along with a distribution over each generator's parameters, including parameters that control the problem size.

3) Identify a set of features.
These features, used to characterize problem instances, must be quickly computable and distribution independent. Eliminate redundant or uninformative features.

4) Collect data.
Generate a desired number of instances by sampling from the distribution chosen in step 2.
For each problem instance, determine the running time of the algorithm selected in step 1, and compute all the features selected in step 3.
Split the data into a training set, a test set, and if sufficient data is available, a validation set.

5) Learn a model.
Based on the training set constructed in step 4, use a machine learning algorithm to learn a function mapping from the featured to a prediction of the algorithm's running time.
Evaluate the quality of this function on the test set.

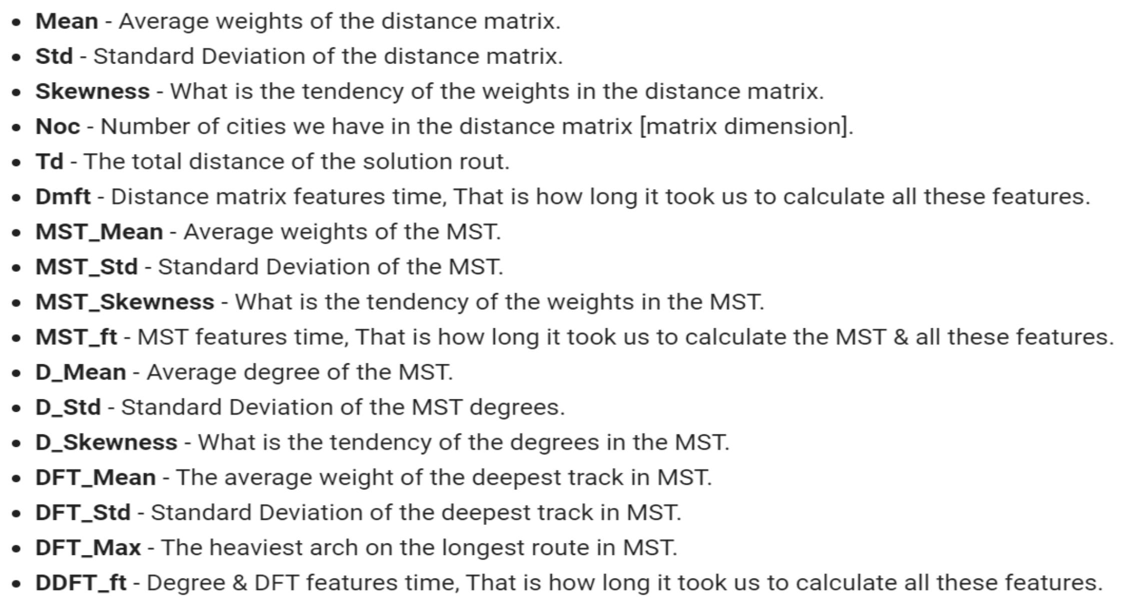
Building Empirical Hardness Models: TSP case study

1) Select an algorithm of interest.
There were many algorithms we could have chosen, Held-Karp or specific implementations of the branch and bound for example, but as the methodology above indicates, there is no to know the internal workings of the algorithm, so we decided to take a black box algorithm from google OR-Tools.

2) Select an instance distribution.
We selected uniform instance distribution to conduct our study. We first select uniformly the number of cities $|V|$ for the problem instance, in a range of 10 to 350.
Then we continue and select the weights for each of our $|V^2|$ edges for the problem instance, in a range of 1 to 1000.

3) Identify a set of features.
We had many features that we thought might be relevant to the empirical hardness of TSP. Starting with very basic ones such as number of cities we have, or the average weight of the distance matrix, and during this process we also came up with more complex ones.
We then examined our data and inferred how our models react to these features.
Finally, we filtered out the redundant ones, who seemed to have no contribution.

We were left with a list of 17 features:



- **Mean** - Average weights of the distance matrix.
- **Std** - Standard Deviation of the distance matrix.
- **Skewness** - What is the tendency of the weights in the distance matrix.
- **Noc** - Number of cities we have in the distance matrix [matrix dimension].
- **Td** - The total distance of the solution rout.
- **Dmft** - Distance matrix features time, That is how long it took us to calculate all these features.
- **MST_Mean** - Average weights of the MST.
- **MST_Std** - Standard Deviation of the MST.
- **MST_Skewness** - What is the tendency of the weights in the MST.
- **MST_ft** - MST features time, That is how long it took us to calculate the MST & all these features.
- **D_Mean** - Average degree of the MST.
- **D_Std** - Standard Deviation of the MST degrees.
- **D_Skewness** - What is the tendency of the degrees in the MST.
- **DFT_Mean** - The average weight of the deepest track in MST.
- **DFT_Std** - Standard Deviation of the deepest track in MST.
- **DFT_Max** - The heaviest arch on the longest route in MST.
- **DDFT_ft** - Degree & DFT features time, That is how long it took us to calculate all these features.

4) Collect data.
We generated a data set of 10,000 instances. Where in each instance, as we discussed before, the number of cities is ranging from 10 to 350, and the weight of each edge is ranging from 1 to

1000. All selected uniformly.
This data set was then partitioned to a train set and a test set, 70 percent and 30 percent respectively.
It is also worth mentioning that we have incorporated an outlier detection mechanism that helped us achieve a bit better results, after we cleaned the outliers from the data set.

5) Learn a model.
While conducting our study, we tried 5 different regression models:

Random forest: or (random decision forests) are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

Ridge: is a popular type of regularized linear regression that includes an L2 penalty. This has the effect of shrinking the coefficients for those input variables that do not contribute much to the prediction task.

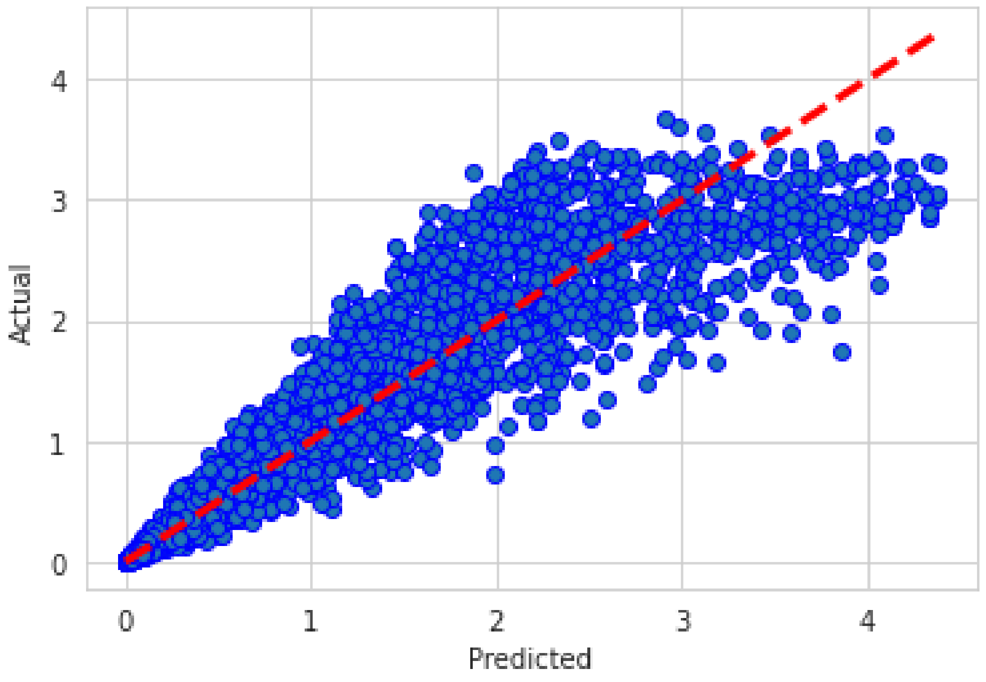
XGBoost: is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework. In prediction problems involving unstructured data (images, text, etc.) artificial neural networks tend to outperform all other algorithms or frameworks. However, when it comes to small-to-medium structured/tabular data, decision tree based algorithms are considered best-in-class right now. Please see the chart below for the evolution of tree-based algorithms over the years.

K-Neighbors: is a regressor based on the k-nearest neighbors algorithm.
Regression with scalar, multivariate or functional response.
The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

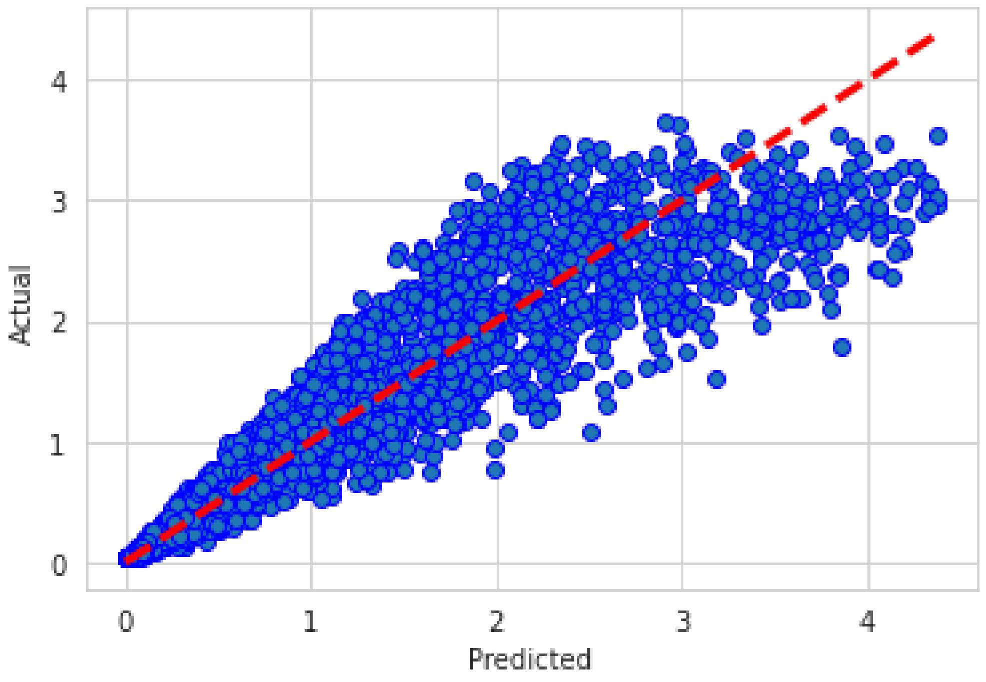
Linear Regression: is a regressor based on Ordinary least squares Linear Regression.
It fits a linear model with coefficients w = (w1, …, wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

Results:

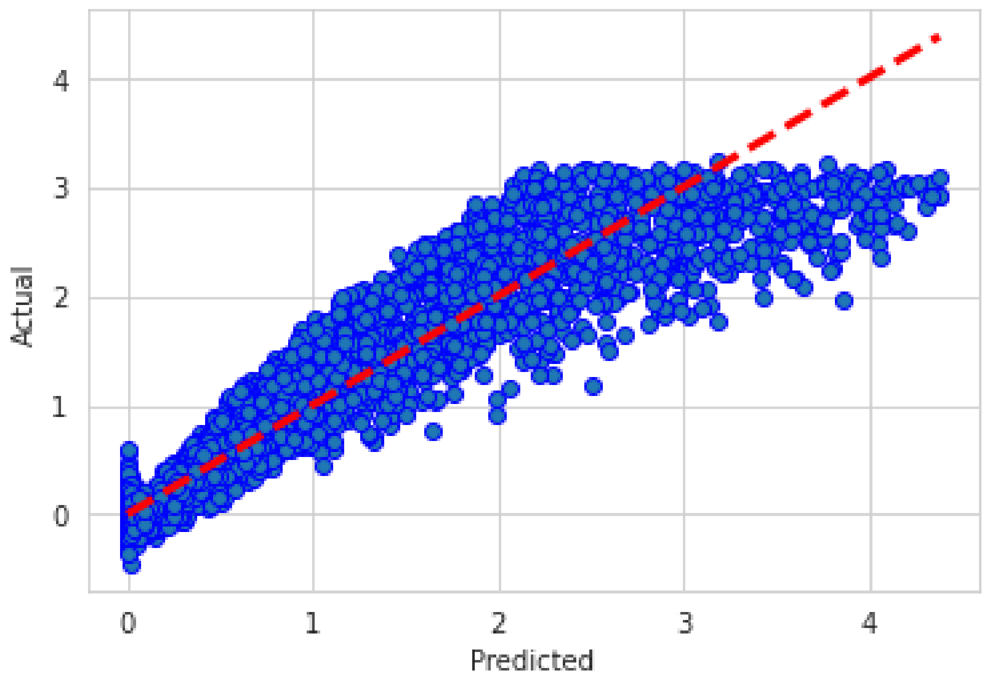
Random forest regressor:



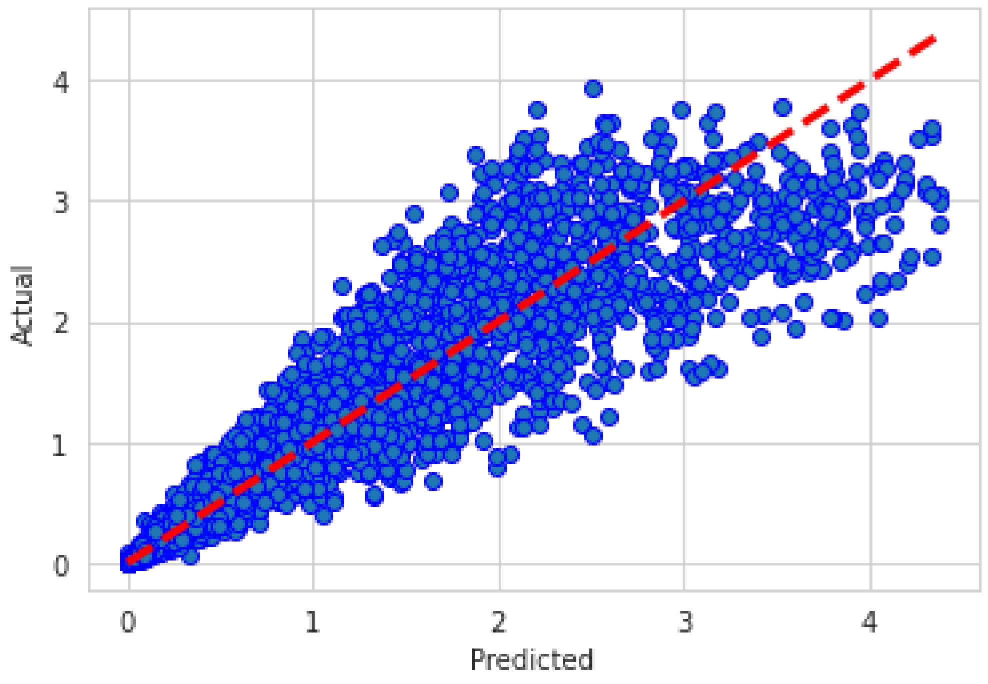
XGBoost regressor:



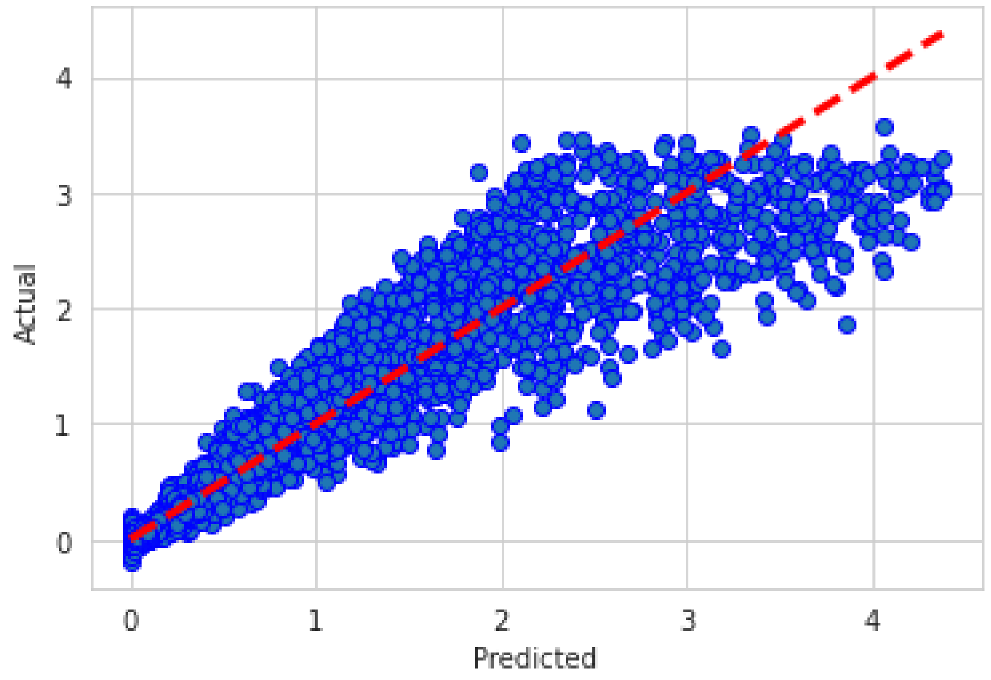
Ridge regressor:



K-Neighbors regressor:

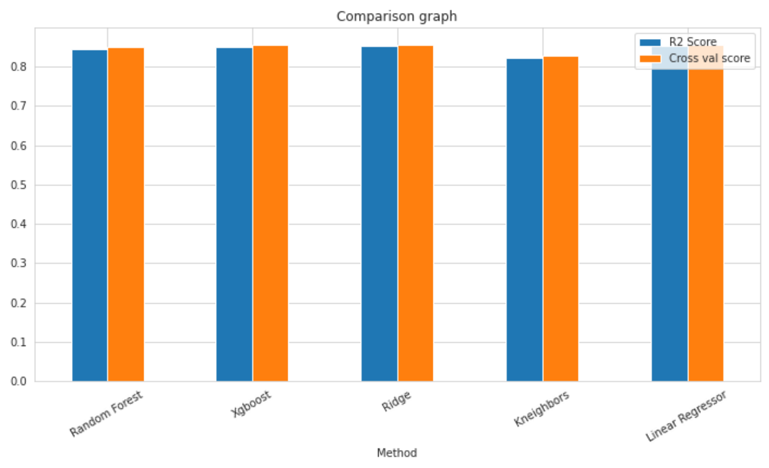


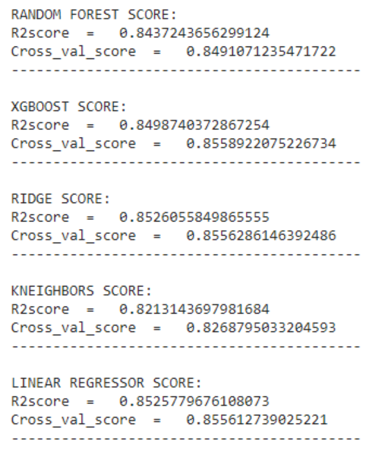
Linear regressor:



Notice that the predicted values we got from the Rigde and Linear Regressor models, for a few of the samples are negative, and when it comes to runtime it is impossible.
That occurs because both of these regressors are types of linear regressors, and linear regression does not respect the bounds of 0, it is linear always and everywhere.
Even if there are no such samples on the training set that indicate a negative predicted value.
One way to adress this issue is to use the natural logarithm of the predicted value, but we decided not to do so, for reasons of consistency.

In order to assess how the models will generalize to an independent data set (the test set) we used Cross-validation on our train set. And in order to assess the models performance on the test set, we used the coefficient of determination (R-squared). Here are the results:

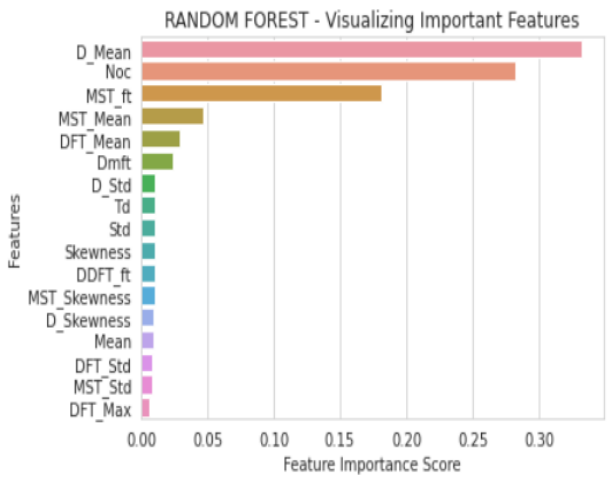


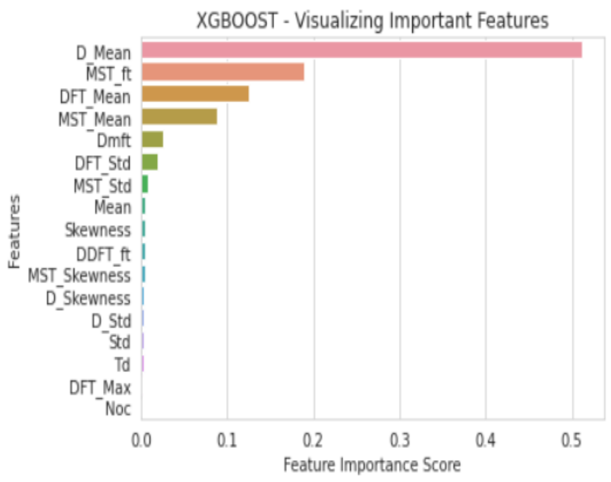


```
RANDOM FOREST SCORE:
R2score  =   0.8437243656299124
Cross_val_score  =   0.8491071235471722
------------------------------------------

XGBOOST SCORE:
R2score  =   0.8498740372867254
Cross_val_score  =   0.8558922075226734
------------------------------------------

RIDGE SCORE:
R2score  =   0.8526055849865555
Cross_val_score  =   0.8556286146392486
------------------------------------------

KNEIGHBORS SCORE:
R2score  =   0.8213143697981684
Cross_val_score  =   0.8268795033204593
------------------------------------------

LINEAR REGRESSOR SCORE:
R2score  =   0.8525779676108073
Cross_val_score  =   0.855612739025221
------------------------------------------
```

As you can see we have reached an R-squared score of a bit more than 85 percent in two of our models. Which is a very decent result compared to our starting point which was somewhere around 56-57 percent when we used a wide range of weights and large number of cities. (More on that in the discussion part)

Lastly, in two of the models (Random forest and XGBoost) we had the ability to use an internal functionality that enabled us to quanifty the importance of the various features we estimated and later used.
In the following figure, in the X-axis we can see the features we used, and in the Y-axis we can see the percentage score that each of there features received in the two models.



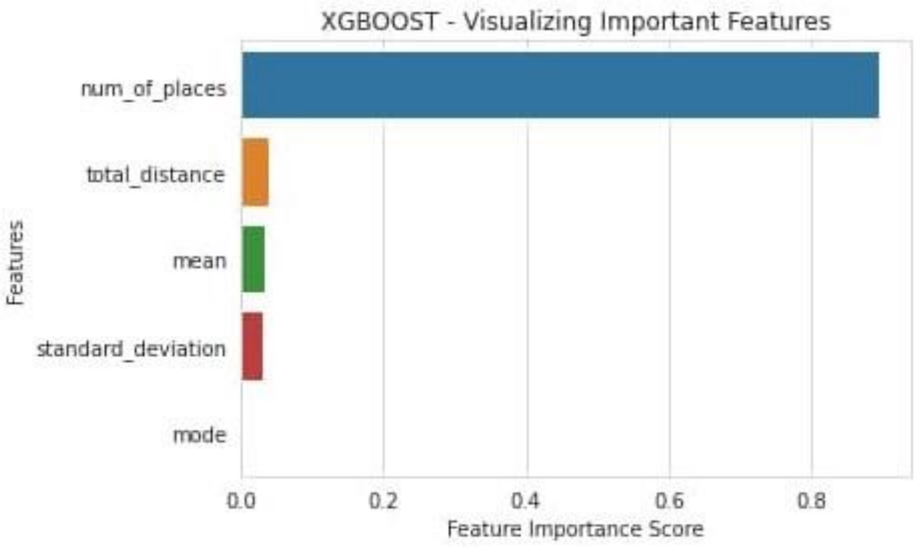


As you can see we have a relatively small set of 5-6 features that really set the tone for the predictive values: the average degree of the MST, the number of cities, the computation time of the MST and its features, the average weights of the MST, the average weight of the deepest track in the MST and the computation time of the features on the distance matrix.

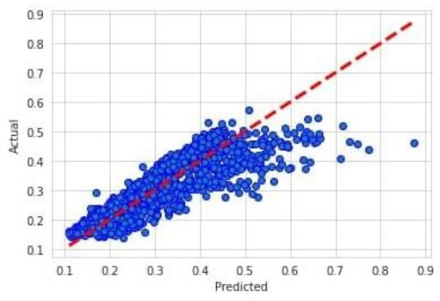
One interesting phenomenon that can be observed is that the number of cities feature (Noc) seems to have no effect when we use XGBoost as our model, while in other models the Noc feature is one of the most prominent features.
Our assumption is that the other top features correlate a lot with the Noc feature, but probably have more input about the difficulty of the problem. So they act as proxies to get a sense of the number of cities, and therefore the XGBoost model don’t see a necessity to have Noc as an additional feature. But again, this is only our assumption, and this has not been proven.

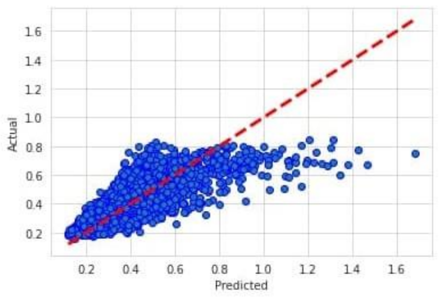
Discussion:

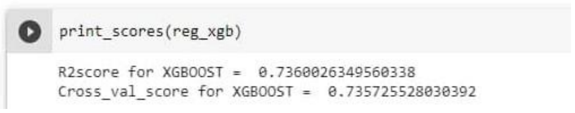
In this project we tried to use the metohdology of empirical hardness models and apply it on the traveling salesperson problem.
When we started to work we divided the data we generated into 9 sets:
We had 3 categories for number of cities range: 10-40, 40-80, 80-160, and 3 categories for weight range: 1-10, 1-100, 1-1000.
We started with some very basic features:



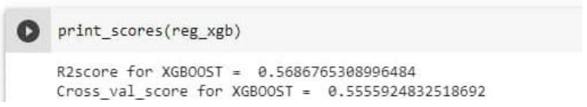
very quickly we understood that we need to find more significant features because the ones we had did not deliver, as we saw clearly from the accuracies our models provided.
We noticed that as we increase the weight range, and by doing so adding a degree of difficulty to our problem instances, our accuracy decreased.
In the following figure you can see the results we got from XGBoost model for a range of 40-80 cities. On the left side we used 1-10 weight range, while on the right side we used 1-1000 weight range.







```
print_scores(reg_xgb)

R2score for XGBOOST =  0.7360026349560338
Cross_val_score for XGBOOST =  0.735725528030392
```



```
print_scores(reg_xgb)

R2score for XGBOOST =  0.5686765308996484
Cross_val_score for XGBOOST =  0.555924832518692
```

The next step for us was getting back to the “drawing board” and try to come up with better features. We went and did a lot of reading on the traveling salesperson problem and looked for other mathematical characteristics that can be translated to better features that can be computed relatively easy and can add consistent accuracy to our models.
Then we started an iterative process of adding new features, testing their effect on the accuracy, and finally filtering those that seemed to have no added value for us.
One connection that we found, and that was very helpful during this process, was the connection between the minimum spanning tree (MST) of a graph and the TSP.
Recall that this connection helps us because the computation time of the MST problem is polynomial.
This connection is also used for solution approximations, most notably the MST-DFS and Christofides algorithms.
There are many lines of similarity between both problems because when we are solving the TSP we look for an Hamiltonian cycle that has the minimal weight sum, and when we solve the MST problem we look for a tree that has the minimal weight sum.
We believe that means that quite a few edges would probably be present in the intersection set between the TSP path edge set and the MST edge set.
So it comes as no surprise that features extracted from the MST problem can help us better predict the runtime of TSP.

Regarding future work, the obvious work to get done is to try and find even better features than the ones we found that would increase our accuracy.
We can also try and expand this case study to other versions of the problem (there are quite a few) and examine how different features affect different versions of this problem.
Maybe some features are “universal” and can help us get a better prediction for all version of this problem and others are not.

In conclusion, in this project we performed a case study on the traveling salesperson problem using a general methodology, in order to better understand the empirical hardness of the problem. Although the work on this project consisted of factors that were new to us, such as: reading and understanding of a scientific paper, implementing ideas from it, and doing a small-scale research. Thus, quite hard and lengthy.