

École polytechnique de Louvain

Learning Priorities in Scheduling Steel Production

Author: **Florian THOEN**

Supervisor: **Siegfried NIJSSEN**

Readers: **Gaël AGLIN, Xavier GILLARD**

Academic year 2022–2023

Master [120] in Data Science: Information technology

Contents

Contents	1
List of Figures	3
1 Introduction	5
2 Related Work	8
2.1 Structured Output Prediction	8
2.2 Structured output prediction algorithm	9
2.3 Prize-Collecting Traveling Salesman Problem	14
2.4 Hamming Distance	16
3 Problem Description	18
4 Problem Approach & Results	19
4.1 Implementation	19
4.2 Model Understanding	21
4.2.1 Hyper parameters	21
4.2.2 Results & Discussion	22
4.2.3 Instance Order Influence	24
4.2.4 Results & Discussions	25
4.2.5 Instance quality	27
4.2.6 Result & Discussions	27
4.3 Model Optimizing	30
4.3.1 Instance subset existence	30
4.3.2 Results & Discussions	31
4.3.3 Subset Property	33
4.3.4 Results & Discussions	33
4.3.5 New Instance Selection Method	35
4.3.6 Results & Discussions	36
4.3.7 Phi-based Selection Method	38
4.3.8 Results & Discussions	39
4.3.9 Weights Initialization Influence	43
4.3.10 Results & Discussions	43
5 Conclusion	47

List of Figures

1	Example of structured output prediction Problem	8
2	The schematic diagram of the stops grouping by zones	11
3	Example solution for the PCTSP, visiting customers 3 and 5	14
4	Arc Based Score with 10 epoch and $\delta = 0.001/0.005$	23
5	Arc Based Score with 10 epoch and $\delta = 0.01/0.05$	24
6	Arc Based Score with 10 epoch and $\delta = 0.001/0.005$ with shuffled instances	25
7	Arc Based Score with 10 epoch and $\delta = 0.01/0.05$ with shuffled instances	26
8	Impact on Score for each instances with $\delta = 0.001$	28
9	Impact on Score for each instances with $\delta = 0.005$	29
10	Arc-based Score depending on Training set size with different learning rate	32
11	Subset depending on learning rate	34
12	Subset depending on learning rate	34
13	Models Performances using Phi Selection Method	37
14	Models Performances using Phi Selection Method	37
15	With $lr = 0.001$	40
16	With $lr = 0.005$	41
17	Models Performances using several Phi Selection Methods	42
18	Weights Initialization influence on methods performances	44
19	Weights Initialization influence on methods performances	45

Acknowledgements

I would like to thank several people that help me during the preparation of this thesis.

First of all, I would like to thank Siegfried Nijssen for high good advice in the discussion that we had Doing our meetings. Thanks to his supervision, I was able to do the work in good conditions

I would also like to thank Gael Aglin for his great help during this thesis and for the wonderful work that he has done. Especially for the data generator and the initial implementation of the algorithm.

Thank to Xavier Gillard for taking the time to read my thesis and agreeing to be part of my jury.

Finally, thanks to my family for their support during these last years even if it was not easy every Day.

1 Introduction

Over the last decade, data has become crucial to business, particularly for decision-making and optimization. As a result, we have seen the advent of machine learning algorithms that use data as support. In this context, companies have realized the need to understand, analyze and use these algorithms on their collected data to make the best possible decisions. This need is driven by many problems that companies encounter daily. Among the problems frequently encountered, a category deals specifically with production planning problems.

This category contains problems called production order planning problems or also called scheduling problem . These problems can be frequently encountered in the metallurgy industry, where the order in which materials are produced is important. As a starting point, we were inspired by the case studied by PSI Metals, a supplier of production management solutions to the metallurgy industry.

The company wanted to study the data collected in the various factories belonging to its customers, particularly those associated with production lines manufacturing metal plates. On a production line, the aim is to manufacture metal plates with different properties. Properties like the thickness or temperature of each plate. The problem is that working with plates with different properties requires the machines on the production line to be set up differently, and this setting change can be costly.

In addition, the cost of this setting change is not fixed. In other words, it can vary according to the different settings the machines have to switch. As a result, it is necessary to choose the order in which the metal plates are sent to the production line. So, to optimize production lines, it is necessary to navigate between the different settings in a certain order to minimize production costs while still achieving a certain profit.

However, this is just one specific example of many within the order planning problems. We then had to find a way to formalize this problem conceptually. To do this, we decided to represent scheduling problems situations using a concept already used in the literature for steel production (A.Özgür et al. 2020 [1]): Prize-Collecting Traveling Salesman Problem (PCTSP). This concept artificially represents a situation corresponding to production planning problems in this industry.

PCTSP is a problem where the objective is to find the shortest path between different cities while collecting the rewards associated with each city visited. It combines minimizing the cost of the path and maximizing the rewards collected. In our case, the shortest route is the order in which our materials must be produced. The cities, therefore, represent the different materials, each associated with a profit generated when they are produced. These profits represent the rewards. The cost between each city is the cost of producing the material from the previously produced material. Since each material requires specific settings to be produced, switching between these different settings generates a certain cost. However, it should not be forgotten that this is only an artificial simulation, which does not correspond perfectly to reality because it simplifies some constraints.

Once the initial problem had been formalized, we had to choose the procedure to solve our PCTSPs to find the optimal solution automatically. As a reminder, the optimal solution is chosen with the constraints that must be met. Furthermore, it is always in the form of a sequence of elements representing an order. For example, in the problem studied by PSI, each element could represent a metal plate. Therefore, the arrangement of this sequence represents the order in which the different plates should be produced.

To do this, we combined two tools: a solver and a machine learning model. A solver is a tool from the field of constraint programming that seeks to solve a particular problem by respecting a set of constraints associated with that same problem. It aims to explore the space of possible solutions and provide the best solution satisfying the constraints imposed (F. Rossi et al. 2006 [2]). This tool is then used to find the optimal solution for each PCTSP assigned to it. We have combined this solver with a machine learning model to automate the resolution of PCTSPs. The aim is to ensure that this model can find the optimal solution using the PCTSP data it receives. As a reminder, the optimal solution is always a sequence. Therefore, the output of the model must follow a certain structure.

In addition, the learning algorithm needs to understand the relationships between the PCTSP data and their respective solutions in order to solve these problems effectively in the future. To do this, it will use the principle of weights. These are coefficients used to quantify the relative importance of features. By adjusting them correctly, the learning model will learn to extract useful information from these different features to produce accurate predictions on new data. The features represent

the information available about the PCTSP under consideration. Consequently, our model learns how to find the optimal solution using the weights that will have adjusted throughout its training.

The data set available to this model is then made up of instances. Each instance groups the information from a specific PCTSP. Each instance is an input that our model receives during training. The inputs contain matrices and the optimal solution. Each matrices represent the different costs incurred by changing the settings for each pair of materials. They will therefore be the features of our problem. Feature in machine learning are the measurable characteristics or properties of the data that are used to train and build models, enabling them to learn patterns and make predictions in various domains. Because of these data, we find ourselves in a supervised learning problem where the data relating to a specific situation represents the labeled data, and the optimal sequences represent the labels.

The learning model then receives the different instances as input, and it aims to find the weights that generalize the relationship between the characteristics and the solution of these instances. These weights are the coefficients that the solver assigns to each feature when looking for the solution of the PCTSP under consideration. In this way, we ensure that the learning algorithm can achieve the best possible performance for future planning problems.

We are therefore dealing with a structured prediction problem. In other words, using PCTSP data, we produce a prediction with a certain structure. In this case, a sequence represents the order in which the materials should be produced. In short, this algorithm will aim to learn, via weights, the preferences found in previously collected data to understand and solve future production planning problems.

However, even if we implement this algorithm, it requires some conditions to be potentially considered as the solution to this type of industry problem. In the professional world, labeled data are very costly to obtain, so the model can't have as much data as it wants. So, the objective is to build a high-performance model while using less data as possible during its training.

In conclusion, the objective of this work is to investigate whether it is possible to learn preferences using a machine learning algorithm while minimising the number of example needed.

2 Related Work

2.1 Structured Output Prediction

As described above, we are in a structured output prediction problem. This type of problem is one of the fundamental tools of machine learning. It appears in many practical areas, such as automatic language processing, computer vision, computational biology, and graph analysis, which is why the structured output prediction problem is a widely studied domain. It involves predicting a complex data structure. (S. Nowozin et al. 2011 [3]). Complex structure means a structured output with several dimensions. In contrast to simple output problems in which, for each instance considered, there is a single output of the same dimensions, structured output problems aim at predicting solutions that can be sequences, trees, or other types of data structures with different dimensions. These predicted outputs are subject to several constraints that they must respect.

Structured prediction:

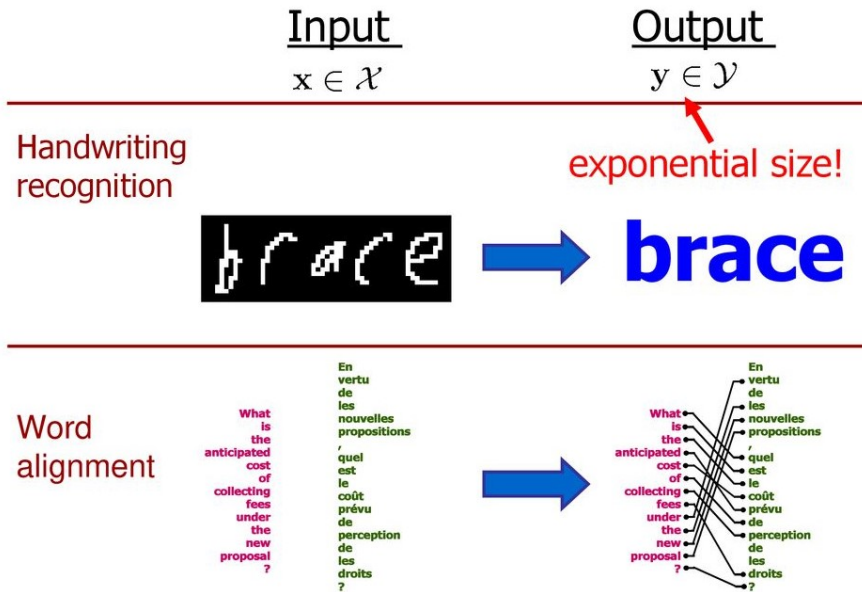


Figure 1: Example of structured output prediction Problem

For example, machine translation is a structured output problem where the objective is to find the translation of a sentence into another language. Therefore, this translation is a sequence of words where each word depends on the previous one. Because of the constraints that the structures must respect, this creates many

inference and learning challenges. Moreover, even if the constraints allow for simple construction of the solutions, finding the optimal one is still NP-hard. However, many approaches have been developed to solve these problems. First, we can talk about Hidden Markov Models (HMM). Hidden Markov models are probabilistic models that model the sequential dependencies between inputs and outputs. They perform well on problems such as speech recognition (P. N Garner et al., 2014 [4]).

Another approach has been to use deep learning to predict the structure of a protein (M. Torrisi et al. 2020 [5]). We can talk in particular about recurrent neural networks, a neural network architecture adapted to structured output problems because they consider sequential dependence. It takes into account the fact that the element in a sequence depends on the previous element. There are also algorithms specially designed for this type of problem. For example, the SEARN algorithm (H. Daumé et al. 2009 [6]). This algorithm transforms complex structured output prediction problems into several classification sub-problems. The aim is that any binary classifier can be combined with SEARN. Given a set of structured prediction examples, this algorithm represents the production of structured output as a search process over the decomposition y (which is the structured output) into $y_1, \dots, y_T$ and trains a cost-sensitive classifier to predict each of the y_t .

In our case, as described in the introduction, we have selected the method using a learning model using weights. Combined with a solver, this model can update its different weights during its training to learn the preferences ruling the solutions to our problem. We based ourselves on a scientific paper (R. Canoy et al. 2022 [7]) in which they developed a structured output prediction algorithm to learn the preferences.

2.2 Structured output prediction algorithm

The Structured output prediction algorithm for learning preferences that we used is introduced in R.Canoy et al. 2022 [7]. In paper comes in a context where the trend of consumers using e-commerce sites is increasing exponentially, so the demand for delivery services is growing along with this trend. It has become relevant to study last-mile delivery because this last link in the distribution chain is the most complicated and costly step. One source of complexity is the many constantly changing, geographically-dispersed delivery locations. In addition, it is widespread in practice for route planners to suggest routes that are tailor-made to specific urban environments instead of recommending those provided by an off-the-shelf software.

The drivers themselves are also known to deviate from the plane’s optimal routes constantly (V.Ceikute et al. 2013 [8], Y.Li et al. 2018 [9]). Then, understanding the decision-making behaviors of the drivers is essential to offer routing plans that capture their preferences.

This structured output prediction algorithm was developed to optimize this last-mile delivery. A common approach was to use machine learning to learn these preferences through past decisions, for example, in recommender systems (Lu et al. 2015 [10]) or search engines (Joachims, 2002 [11]). However, classical machine learning can’t be used in this situation because the output has to be structured solutions (a route between delivery locations). So the approach used is a novel combination of two learning approaches. A fast probability estimations method that estimates the transition probabilities and that can be a reformulated search that any TSP solver can be used to find the maximum-likelihood TSP solution. As well as a structured output prediction method that can learn the weight of a linear preference function by gradient descent of a TSP solver before every weights update.

So the goal is to find the weights that characterize the drivers’ behavior during the previous deliveries to plan future deliveries in the best way. The weights are coefficients to quantify the relative importance of the difference features of a problem. In other terms, it allows the model to learn the relation between the urban environment of the day considered in the route chosen by the drivers. It is assumed that the driver’s preferences are influenced by his personal preferences and the distance between each stop that he tries to minimize. In this problem, the different addresses at which parcels will be dropped off are called stops. However, to facilitate the solution to the problem, the stops will be grouped by zones. That is, each zone will be composed of several stops according to their proximity as in figure 2.

The structured output prediction algorithm is used to learn the weights. Still, it needs a TSP solver, which uses the information in each example encountered during training to calculate the optimal route. This type of solver is used because the aim is to minimize the total delivery cost (route cost).

In this approach, two assumptions were necessary. Firstly, it was assumed that preferences could be captured as the costs required to get from one zone/stop to another. This assumption is standard as it is used in (R.Cnoy et al. 2019 [12], Mandi et al. 2021 [13], Chen et al. 2021 [14]) Therefore, these preferences could be

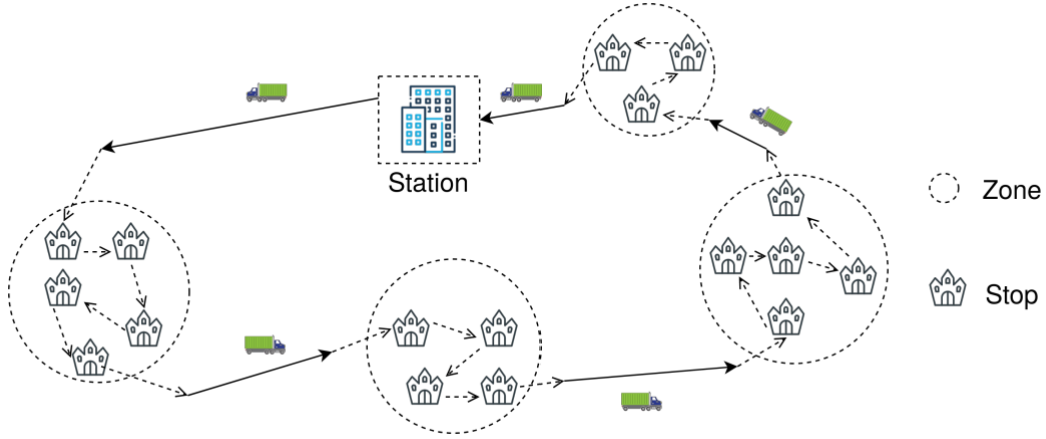


Figure 2: The schematic diagram of the stops grouping by zones

represented by transition matrices capturing these costs. Any TSP solver can then use these matrices. Data available are defined as follows:

- S : the set of delivery locations (or stops)
- $H = \{S_t, q^t, x^t\}_{t=1}^T$: the daily historical set of solutions with:
 - $S_t \subseteq S$: set of stops at day t
 - x^t : the route plan on day t - those plans are feasible solutions to the traveling salesman problem (TSP).
 - q^t : vector stating other relevant features of the data of day t
- $Z = \{Z_1, Z_2, \dots, Z_m\}$: zones such that $\bigcup_{k=1}^m Z_k = S$ and $Z_k \cap Z_{k'} = \emptyset$ for $k \neq k'$

The authors proposed an approach consisting of two parts. The first concern is the optimization of the visit to the different zones. The second part was optimizing the visit to the stops in each zone. Optimizing means minimizing the total cost between the different considered locations as possible. For our work, only the first part interests us. Indeed, in our case, as explained in the introduction, we can link the zones and the different calibration settings on the production lines.

This first part is composed of several steps. First, it is necessary to construct the matrices representing respectively the personal preferences of the driver and the distance between each zone using the available data. Then, the cost matrix is the new matrix resulting from the best-weighted combination of the two matrices that must be found using the structured output prediction algorithm. Once the cost

matrix is obtained, it is possible to use the TSP solver to obtain the optimal route. As a reminder, this route represents the order in which the different zones should be visited to optimize the path.

Given a set S of stops, with (asymmetric) costs c_{ij} associated with each (i, j) -pair of stops in the cost matrix, the TSP consists of determining a minimum cost circuit passing through each vertex once and only once (Miller et al. 1960 [15]; Laporte 1992 [16]). The programming formulation to solve the TSP is:

$$\text{TSP: } \min \sum_{i \neq j} c_{ij} x_{ij} \quad (1)$$

$$\text{subject to } \sum_{j \in S} x_{ij} = 1 \quad i \in S, i \neq j \quad (2)$$

$$\sum_{i \in S} x_{ij} = 1 \quad j \in S, j \neq i \quad (3)$$

$$\sum_{i, j \in V} x_{ij} \leq |V| - 1 \quad V \subset \{1, \dots, n\}, 2 \leq |V| \leq |V| - 2 \quad (4)$$

$$x_{ij} \in \{0, 1\} \quad i, j \in S, i \neq j \quad (5)$$

Where the variable vector x is defined as above. The objective function minimizes the sum of the costs over the arcs in the route. The first two constraints ensure that each stop is visited only once. The third constraint ensures that there are no subtours in the solution. Our approach does not rely on an integer programming formulation but works with any TSP solver.

By abuse of notation, we refer to $\text{TSP}([c_{ij}])$ as an anytime oracle that returns an optimal solution of (TSP) when the cost vector c is optimized, or the best one found in case a time limit is reached. Several exact methods and heuristics have been developed to solve this problem since the 60s, making it one of the most studied problems in operations and transportation research.

As the introduction describes, the model adjusts the weights using collected data. So finding the weights of this combination is like learning the preferences governing the problem data. Many approaches to the structure prediction problem have been

invented.

However, the authors of this article decided to base their approach on the seminal structured perceptron algorithm. As a reminder, a perceptron is a type of artificial neuron network node that operates by taking multiple inputs, applying weights to these inputs, summing them up, and sending through an activation function to produce an input (M.Minsky et al. 1969 [17]). They are assuming a given dataset $H = \{(u, x)\}$ of T training examples with u corresponding to an input structure and x the intended output structure, where $x \in X(u)$. $X(u)$ is the (implicit) space of all possible outputs that satisfy the structural constraints (e.g., the feasible solutions of the TSP defined implicitly by (2) - (5)). The structured perceptron algorithm requires defining a problem-specific representation function Φ that maps every valid (u, x) to a feature vector $\Phi(u, x)$. The feature vector's length determines the weight vector's length $\mathbf{w}$.

Algorithm 1 structured output prediction Algorithm

```

1 Initialize  $\mathbf{w}$ 
2 for  $e = 1$  to  $E$  do
3   forall  $(u, x) \in H$  do
4      $\hat{x} \leftarrow \arg \max_{\hat{x} \in X(u)} \mathbf{w}^T \Phi(u, \hat{x})$ 
5     if  $\hat{x} \neq x$  then
6        $\mathbf{w} \leftarrow \mathbf{w} + \delta(\Phi(u, x) - \Phi(u, \hat{x}))$ 
      end
    end
  end
end
7 return  $\mathbf{w}$ 

```

The weight vector is initialized on line 1, e.g., to constant values such as 0 or 1. In every epoch e , the algorithm iterates over the training examples in H (line 3). It first computes the predicted TSP $\hat{x}$ of the given instance for weight vector $\mathbf{w}$. If the expected output does not match the intended TSP x , then a small perceptron update to the weights $\mathbf{w}$ is performed, using the difference between the representation function $\Phi(u, x)$ of u with the accurate structured output x and $\Phi(u, \hat{x})$ of u with the predicted output $\hat{x}$. The δ is the learning rate: a small value that controls how large or small the weight updates are in each iteration. This procedure is repeated for E epochs.

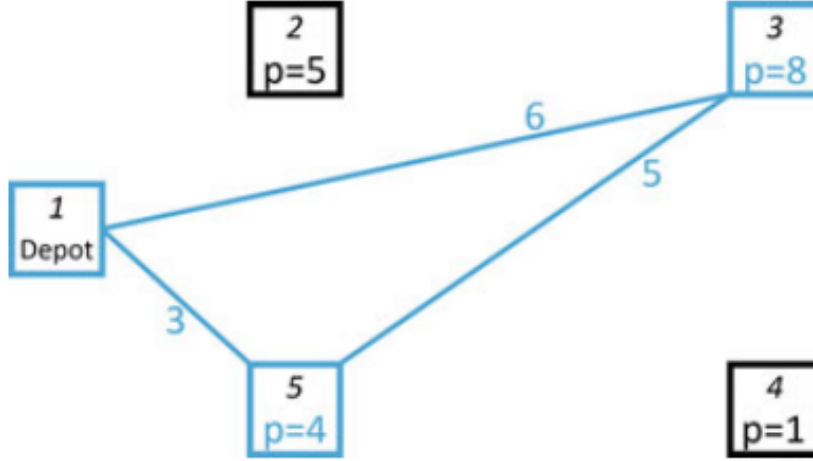


Figure 3: Example solution for the PCTSP, visiting customers 3 and 5

2.3 Prize-Collecting Traveling Salesman Problem

In the previous section, we presented the algorithm that we used. Nevertheless, as describes in introduction, the Prize-Collecting Traveling Salesman Problem (PCTSP) in order to artificialy simulate a scheduling production problem. In PCTSP, different cities to be visited each has the particularity of yielding a certain profit when we have visited them. This means that, in addition to minimizing the total distance we visited, the solution of a PCTSP considers ensuring a minimum profit (P. Vansteenwegen et al. 2019 [18]). The main difference between a PCTSP and a TSP is that the solution can only pass through some cities. It is considered satisfactory as long as the solution ensures a threshold profit and minimizes the cost of the traveled path.

We have therefore used this formulation to apply this variant in the structured perceptron algorithm. In our case, the shortest path is the order in which our materials must be produced. The cities, therefore, represent the different materials, each associated with a profit generated when they are produced. These profits represent the rewards. The cost between each city is the cost of producing the material from the previously produced material. For this work, we therefore used the mathematical definition proposed in the book where the Prize-Collecting Traveling Salesman problem is considered as follows: Consider a set of nodes $N = \{1, \dots, |N|\}$ where each node $i \in N$ is associated with the non-negative profit P_i . The start and end nodes are fixed to nodes 1 and $|N|$, respectively. Nodes 1 and $|N|$ can also represent a single depot when both are in the same location and have a profit. $P_i = 0$ is

associated with both nodes.

The goal of the PCTSP is to determine a route that visits a subset of N , collecting at least a given profit P_{min} and minimizing the total travel cost. It is assumed that accumulated profits can be added, and each node can be visited at most once. The non-negative travel cost between nodes i and j is t_{ij} . The PCTSP can be formulated as an integer programming model with the following decision variables: $x_{ij} = 1$ if a visit to node i is followed by a visit to node j and 0 otherwise.

The variable u_i will be used in the subtour elimination constraints and allow us to determine the position of the visited nodes in the route. Constraints are :

$$\text{Minimize } \sum_{i=1}^{|N|-1} \sum_{j=2}^{|N|} t_{ij} x_{ij} \quad (6)$$

The objective function (6) is to minimize the total travel cost.

$$\sum_{j=2}^{|N|} x_{1j} = \sum_{i=1}^{|N|-1} x_{i|N|} = 1 \quad (7)$$

Constraints (7) ensure that the route starts from node 1 and ends on $|N|$.

$$\sum_{i=1}^{|N|-1} x_{ik} = \sum_{j=2}^{|N|} x_{kj} \leq 1; \forall k = 2, \dots, (|N| - 1) \quad (8)$$

Constraints (8) ensure the connectivity of the route and guarantee that each node is visited at most once

$$\sum_{i=1}^{|N|-1} \sum_{j=2}^{|N|} P_i x_{ij} \geq P_{min} \quad (9)$$

Constraint (9) sets a minimum value P_{min} to the profit collected.

$$2 \leq u_i \leq |N|; \forall i = 2, \dots, |N| \quad (10)$$

$$u_i - u_j + 1 \leq (|N| - 1)(1 - x_{ij}); \forall i, j = 2, \dots, |N| \quad (11)$$

The combination of constraints (10) and (11) prevent subtours.

2.4 Hamming Distance

Finally, we need a metric to measure the quality of the predictions of our model. Namely, we need to calculate the degree of similarity between two sequences. For this purpose, we have chosen the Hamming distance. This metric measures the distance between two sequences of symbols of the same length. The Hamming distance equals the number of positions from which the corresponding characters in the two sequences differ. In other words, it measures the number of substitutions needed to transform one series of symbols into another (R. Hamming, 1986 [19]). The formula is as follows:

$$dH(X, Y) = \sum_{i=1}^n (X_i \neq Y_i) \quad (12)$$

Where $dH(X, Y)$ is the Hamming distance between the sequences X and Y , sequences of the same length n . X_i and Y_i represent the symbols at the corresponding index i .

For example, you can consider the sequences

(a)1100010000

(b)1100010011

The distance between (a) and (b) equals 2. Indeed, transforming (b) into (a) requires two substitutions. This metric allows us to evaluate how far our predicted solution will be from the true answer.

However, even if the aiming distance offers a reliable way to compare two sequences, it is crucial to note its primary limitation. The arming distance can only be calculated between two sequences of equal length. This means that the attempt to compute the arming distance between two sequences of different lengths constitutes a violation of its underlying principle.

For example, trying to find the aiming distance between the sequences :

(c)1100010000

(d)110001001

is problematic because the sequence differs in length. In that case, an alternative must be found to adapt the computation or use a different distance, like the Levenshtein distance, which tolerates sequences of different lengths.

3 Problem Description

This work aims to use the previously presented perceptron algorithm on PCT-SPs. After implementing the machine learning algorithm, the objective is to observe if it's possible to obtain good precision while using not all the available data (a data subset). Then, we had to adapt the perceptron algorithm to work with our problem, a prize-collecting traveling salesman problem, not a traveling salesman problem.

We divided the main problem into three sub-problems.

- How do hyperparameters influence model precision and which values gave the best performance?
- Is there a subset of data that performs as well as when the model has access to the whole data set?
- Is there a relevant way to find this subset?

The first concerns question is about understanding the model and evaluating the influence of the different hyperparameters on the solution. Indeed, because we had to adapt the initial learning algorithm presented in the paper, it can react differently from the original one. Therefore, it is mandatory to understand how the hyperparameters work to make the best use of them.

The second question is to find if a subset exists, allowing the model to have good precision. We want to study if it's possible to learn the preference with a small amount, then we first need to prove the existence of a data subset on which our model performs well.

The last problem concerns the way to find this subset. Having proved that there is such a subset, we must find a feasible way in the industry world. For example, we cannot perform one complete model training to find this subset and then reuse it to retrain the model. This would be too costly in the industry world.

4 Problem Approach & Results

Before answering to the first question, we first need to implement the perceptron algorithm for it to work on our PCTSP instances.

4.1 Implementation

The first step is the implementation of the structured perceptron algorithm. We implement it in Python. The constraint programming part consists in modeling the prize-collecting traveling salesman problem using the CPMpy library [20]. We code this learning model so that it has several hyperparameters. Most of them intend to facilitate collecting results throughout the training process. Only a few of them concern the model itself:

- *epoch* E : represents the number of times the main procedure is repeated during learning.
- *size* : is the size of the instances the model uses to learn.
- *learning rate* δ : defines how much the model weights evolve over the iterations

Also, a few details are needed concerning the general structure of the learning algorithm. Concerning the weights, they are initialized randomly at the beginning of the learning process. While available data are visited according to a predefined order. In our case, we identify with numbers ranging from 1 to 200 by default because we have 200 examples. So, the instance identification depends on the number of examples you have. To obtain these instances we implement a generator that produces a sample of PCTSP instances. After instance identification, the algorithm visits each of them during the main procedure, starting with instance number one and ending with instance 200. Finally, we use the Hamming distance to evaluate the quality of the solutions. To do that, we compute the distance between the optimal solution present in the input data and the solution predicted by the model.

A problem is that the hamming distance is defined on two sequences of the same size and returns the number of positions where the characters are different [24]. However, our model may not predict a solution of the same size as the optimal sequence. This is because the solution of a PCTSP is allowed to only pass through some cities. Thus, when the model weights are still far from the actual weights, the model may predict a route satisfying the constraints that do not pass through all

cities while there may be better routes for the given situation. This route represents the best possible route with the weights available to the model when it considers this instance. Therefore, this happens most of the time in the first iterations because the model weights still need to converge to the real weights sufficiently. So, to resolve this problem, we modify the metric in order to be able to use it over all our model predictions.

This gave rise to Node-based hamming and Arc-based Hamming. For the first one, the idea is to do padding on the shortest string using a character that does not exist in any sequences. The padding we first two in the process of adding extra elements to the sequence in order to make them of equal length. Once the padding is done, the sequences have the same length. So we can use the hamming distance defined in Related Work (2.4) on them. The disadvantage of this approach is that it only captures common substrings once they are in the same position. For example, we consider the sequences :

(a) *ABCD*

(b) *CDAB*

If we use the node-based metric, the distance between (a) and (b) is equal to 4 because none of the elements present in the first sequence are also found in the other sequence at the same position. However, we can see that there are still subsequences AB and CD in both sequences. To overcome this problem, we propose a variant called Arc-based hamming.

The specificity of this metric is that it considers the transitions between the elements of a sequence rather than the positions of the elements themselves. It counts the transitions between two elements that are not common to the two sequences under consideration, thus allowing us to consider the character substrings common to the sequences while preserving the notion of order in the sequences. In the sequence, the order refers to the specific arrangement of elements within the sequence. With this modification, we can start analyzing the results produced by our learning algorithm.

4.2 Model Understanding

With the algorithm implemented, we try to answer the question about the hyper parameter influence on the model performance (Chapter 3). As explained earlier, understanding the learning algorithm requires understanding the hyper parameters' influence on the model's performance. To assess the quality of a model, we use a process that followed a specific pattern. First, we separate the data we have into two distinct sets:

- A training set consists of all instances the model had access to during its training. This represented the history of the data collected previously.
- A test set: This contains the instances that we use to evaluate the performance of our trained model.

After that, it was necessary to configure how the model was evaluated using these sets. The algorithm uses the training set to converge the model weights. The algorithm, therefore, went through each of the instances in this set one by one. It solves the PCTSP of the considered instance and updates its weights by recovering the found solution. After this update, the test set intervenes. Indeed, after each adjustment of weights, thus, at each iteration, the model is evaluated on the test instances. We take the model and use it to predict the solution for each example in the test set. Then we retrieve these solutions and compare them to the solutions associated with each of the instances considered. So we use Arc-based Hamming to determine how far the predicted solutions are from the real ones and average these obtained distances to compute the average performance of the model over the test set. Therefore, if the model goes through 100 iterations, it is evaluated 100 times. It allows us to get an overview of the evolution of the model's accuracy over the iterations. Once this was implemented, we can start the various tests related to the hyperparameters.

4.2.1 Hyper parameters

To obtain the best possible results, we start with the following reasoning. We set the instances to 10 (only 10 cities considered in PCTSPs) because the execution time of the model increases exponentially with the size of the PCTSPs considered. Then, we test different combinations of values for the number of epochs and the learning rate δ . To find these combinations, we set E (number of epochs) to 10.

Once the number of epochs is defined, we train the algorithm for four different learning rate values (0.01, 0.05, 0.001, 0.005). These learning rates are used during the entire research. The obtained results are represented by graphs 4 & 5.

4.2.2 Results & Discussion

The graphs 4 5 show the evolution of the model's accuracy as a function of the number of iterations performed. As you can see on the x-axis of each graph, it contains values ranging from 0 to 1600. Therefore, each value on the x-axis represents the number of iterations performed previously when we evaluate the model. In our case, we have 200 instances at our disposal. As explained above, we divide this data set into two groups. A training contains 160 examples and a test set contains 40 examples. As a reminder, the perceptron algorithm visits all the instances in the training set at each epoch. So one epoch generates 160 iterations. So, with 10 epochs, our model performs a total of 1600 iterations.

The accuracy of the models in Graph 4 starts at 50% and converge at a maximum of 98%. We can also see that the greater the number of iterations, the higher the score is. However, the accuracy increases slowly or even decreases after a certain number of iterations. After this threshold, the accuracy fluctuates between several values within a more or less wide range. Nevertheless, the values are above 80%. The important point is that the shape of the score's evolution differs according to the learning rate used. In this graph, we can see that the score for the value 0.005 reaches the threshold mentioned above after +/- 200 iterations. For the value 0.001, however, the model takes a little longer to learn, as it only reaches the threshold after +/- 800 iterations.

Despite this, the second learning rate is not necessarily worse. We can see that the learning rate achieved using the 0.001 value is much more stable than that of the model with the 0.005 value. This stability is reflected in the interval in which the oscillations encountered by the models fluctuate after reaching their respective thresholds. The range of values within which the score of the second model will oscillate is much smaller than that of the first model. As a result, the variations in the performance of the first model are more significant than that of the other. Moreover, the interval's size derived from the value 0.001 tends to get smaller with each iteration. In contrast, the interval for the first model tends to increase slightly with each iteration.

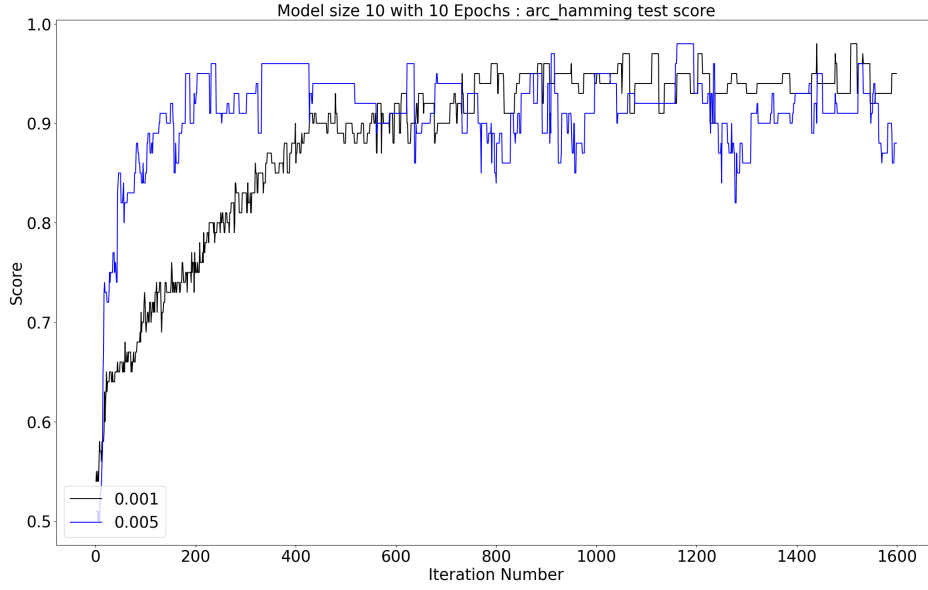


Figure 4: Arc Based Score with 10 epoch and $\delta = 0.001/0.005$

The second graph (5) is associated with the performance of models using learning rates equal to 0.01 and 0.05. About the general performance trend, we distinguish some similarities, but also some differences with the previous graph (4). Firstly, for each model, there is also a particular number of iterations above which the model's accuracies no longer progress as well as before. As in the first graph, the performance of the models exhibits oscillatory behavior above this threshold. We also observe that the two models converge at most towards an accuracy equal to 98%.

Nevertheless, the accuracy of the models during the first iterations starts at around 70%. This is 20% higher than for the 0.001 and 0.005 learning rates. Another difference is the number of iterations needed to reach their respective thresholds and the size of the variations after these. For the 0.01 learning rate, we can see that its threshold is reached after ± 50 iterations. The second model reaches its point almost instantaneously. Moreover, the variations for these models (Graph 5) are more significant than those for the models represented in Graph 4. Note that the oscillations of the second model are greater than those of the first. Given the significant variations, we can again speak of the flagrant lack of 'stability' of the models in Graph 5.

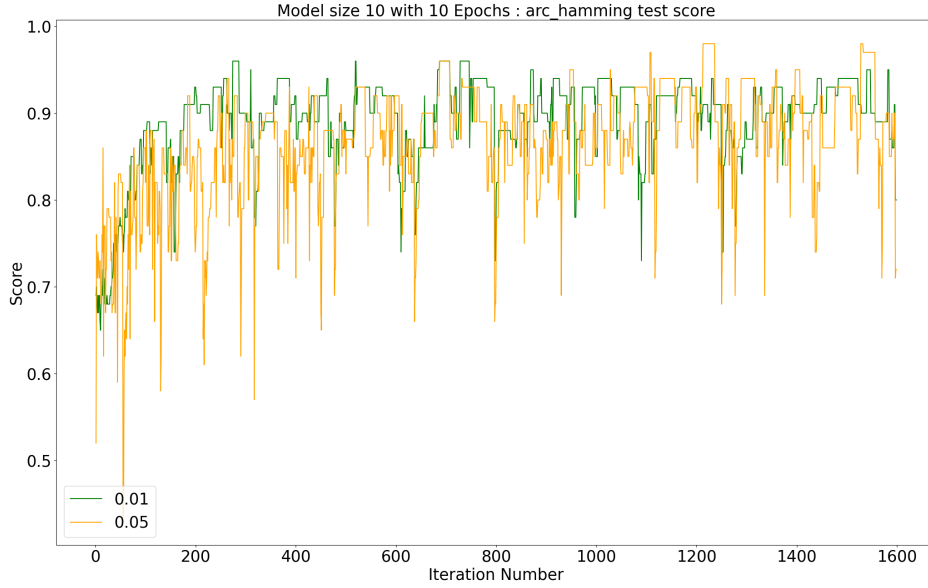


Figure 5: Arc Based Score with 10 epoch and $\delta = 0.01/0.05$

With this graph analysis, we propose two hypotheses:

- A model takes more than one epoch (160 iterations) to achieve maximum performance.
- The higher the learning rate, the less stable the model will be.

In our case, the value 0.001 provides a much more stable model than 0.05. Each model, as it learns, reaches a certain threshold. This is the point at which convergence becomes much more difficult. To compensate for this, using a reasonably small learning rate would probably be necessary, allowing slow but effective convergence in the long term.

4.2.3 Instance Order Influence

This section aims to see whether the order in which the instances are visited during model training influences the model's performance. Is there an order that gives better performance? We modify how the learning algorithm traverses the training set to answer this question. We ensure that the model crosses the set of instances in different random order for each epoch. With this modification, we can obtain results to observe the effect of the order in which the instances are traversed on the performance of our model. Finally, to test the validity of the hypotheses we

formulate earlier, we reuse the same values for the hyper-parameters as those we chose in the previous section. Result are displayed in graphs [6](#) & [7](#)

4.2.4 Results & Discussions

In graphs [6](#) and [7](#), we see that the accuracy of the models never reaches its maximum during the first epoch. The first hypothesis, therefore, seems to be valid.

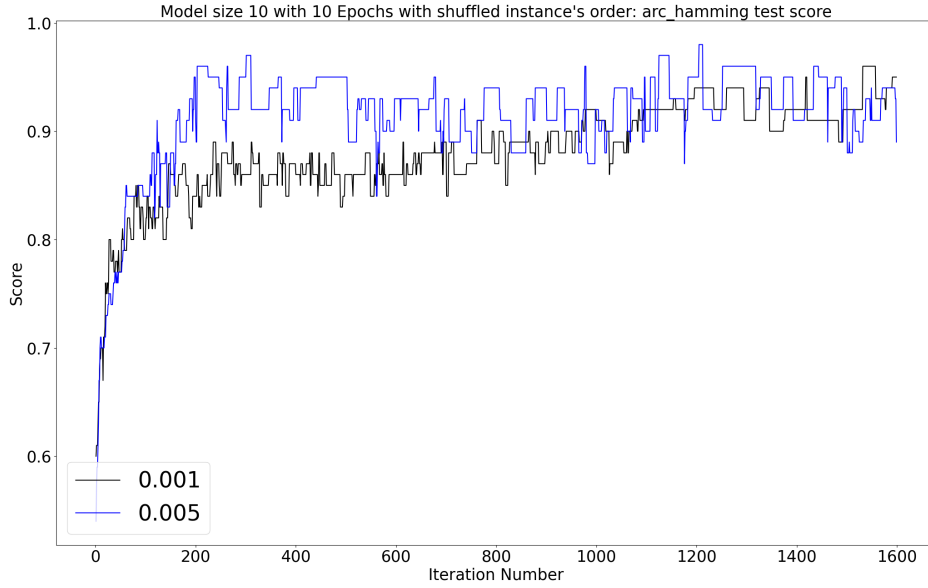


Figure 6: Arc Based Score with 10 epoch and $\delta = 0.001/0.005$ with shuffled instances

For the second hypothesis, we analyze the models associated with the values 0.001 and 0.005 (Graph [6](#)). Certain similarities are observable where the order of the instances is not random (Graph [4](#)). When the models are initialized, the scores start at around 50% and converge to a maximum of 98%. For each of the two models, there is also a certain number of iterations beyond which learning becomes much more difficult. Beyond this threshold, performance oscillates within a range of values. This threshold reached around 1100 iterations for 0.001 and 200 iterations for 0.005. In addition, the size of the variations also tends to reduce over time, reflecting a certain improvement in stability with each iteration. Finally, the variations for the 0.001 value are smaller than those for the other model. Therefore, as in the previous case, the 0.001 learning rate produces a more stable model.

However, some differences remain. The first concerns the general shape of the

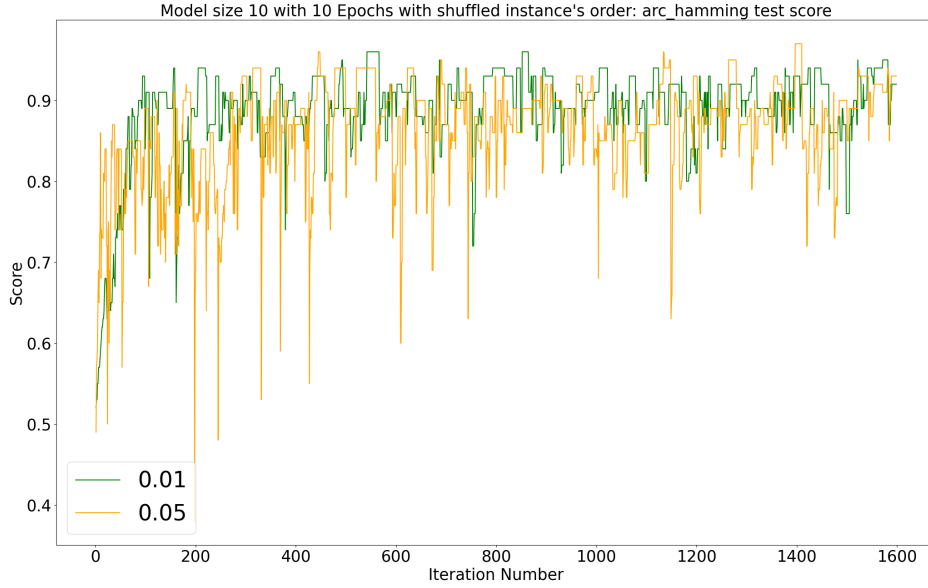


Figure 7: Arc Based Score with 10 epoch and $\delta = 0.01/0.05$ with shuffled instances

performance. By comparing the two graphs [4](#) and [6](#), it is easy to see that the learning process is not the same. In graph [4](#), the models reach their threshold pretty quickly. After that, the model scores improve very slowly. In graph [6](#), the learning is slower, so the difficult learning point is reached later in terms of iterations. This implies that the maximum performance of each model is reached later than in the case where the instances are not visited randomly. This is particularly noticeable when the learning rate is equal to 0.001. We then look at the results associated with the 0.01 and 0.05 values. The observations that we formulate above are also valid for graph [7](#), which also shows an increase in the instability of the model as the learning rate increases.

From these results, we conclude that a small learning rate (0.001/0.005) produces more stable and better models. That answers the question about the order of instances. The various graphs show that the order in which the instances are visited influences the algorithm’s learning. This is particularly true for small learning rates such as 0.001.

Nevertheless, it is possible to achieve the same performance (98% accuracy) as graph [4](#) if the number of epochs is high enough for the model to converge sufficiently. Therefore, we can say that the best configuration is to explore the instances randomly. This is justified by the fact that we want to create a model that is as

general as possible. If we test our model and it uses an arbitrarily defined order to traverse the instances, the results obtained are likely to be biased by this specific order. Therefore, We also set the number of epochs to 10, allowing the model that randomly traverses the instances to converge sufficiently toward its optimum.

With these results, we can confirm that the order in which the instances are visited influences the learning of our model but one question remains. What is the impact of the different cases on the learning of the model? Now that we understand the general behavior of the learning curve, the aim is to go deeper. This question also arises from the fact that the accuracy of the models shows more or less significant negative variations during learning. Consequently, it is legitimate to ask whether this is due to a general intrinsic property of PCTSPs or whether it is due to the lower quality of specific instances.

4.2.5 Instance quality

By quality, we mean each instance influences the learning of the model, i.e., we want to measure each instance’s impact on the scores when the model considers it. To do this, we first test two models with learning rates equal to 0.001 and 0.005, respectively because these values produce the best models. As explained above, from now on, we only consider algorithms that run through the instances in random order and with a fixed number of epochs (10). We then modify our code as follows. At each epoch, the algorithm stores the impact on the score for each instance. In other words, we compute the difference between the score obtained after updating the weights using the example in question and the score received when processing the previous instance. We plot the results in figures [8](#) and [9](#).

4.2.6 Result & Discussions

The x-axis contains the identifiers representing each of the 200 instances, while on the y-axis contains the impact of the corresponding instance on the score. In addition, for each of our instances, there are ten points. Given that our models have ten epochs, each instance is encountered ten times. A color code also represents the epoch number at which the impact on the score is recorded. The lighter the color of the dot, the higher the epoch at which the data is recorded. Conversely, the darker the color, the more likely it is recorded at an epoch close to initialization. In short, the impact on the score gives an idea of the improvement in precision after the algorithm has considered this instance.

For example, if instance 167 has a score of 0.05, it means that instance 167 has allowed the model to improve its accuracy by 5%. Another point to note is that each instance may not have ten points. This means that several impacts for this example have the same value. In our case, when this happens, the missing points are equal to zero. They are, therefore, all stacked on the line representing the value zero.

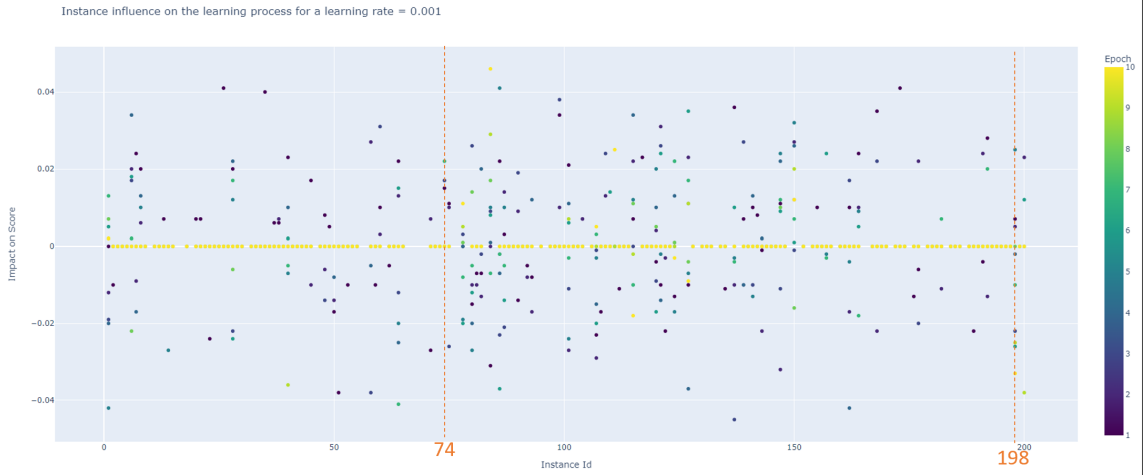


Figure 8: Impact on Score for each instances with $\delta = 0.001$

We first analyze the general shape of the graphs [8](#) and [9](#). For the value 0.001, the impact of each instance is between -0.045 and 0.046. For the second model, these values are between -0.08 and 0.066. The range for the second model is greater because a learning rate of 0.005 leads to greater variations in the learning curve. Secondly, it is easy to see that most very light-colored dots have a null impact. This phenomenon reflects the convergence process. As they are light-colored, these data are collected during fairly high iterations. The model has, therefore, almost finished converging toward its maximum accuracy.

We then look in detail at the instances in graphs [8](#) and [9](#). We omit the yellow dots from our analysis. Given that they symbolize the data collected towards the end of the model learning process, they cannot give us any precise information about the quality of the examples. Following this analysis, we distinguish three instances categories.

The first concerns the so-called "high quality" instances. This category refers to instances that had only a positive or null impact on the score. This applies to both models. For example, instance 74 (graph [8](#)) has three positive values (0.015, 0.017, 0.025), and the others are equal to zero. Consequently, this example never has a

negative influence on the accuracy of our model. Moreover, this observation is also valid for graph [9](#), with points with positive values (0.023, 0.029) and the remainder equal to zero. Note that this is not the only example. Instances 78, 155, and 172 are also in this category.

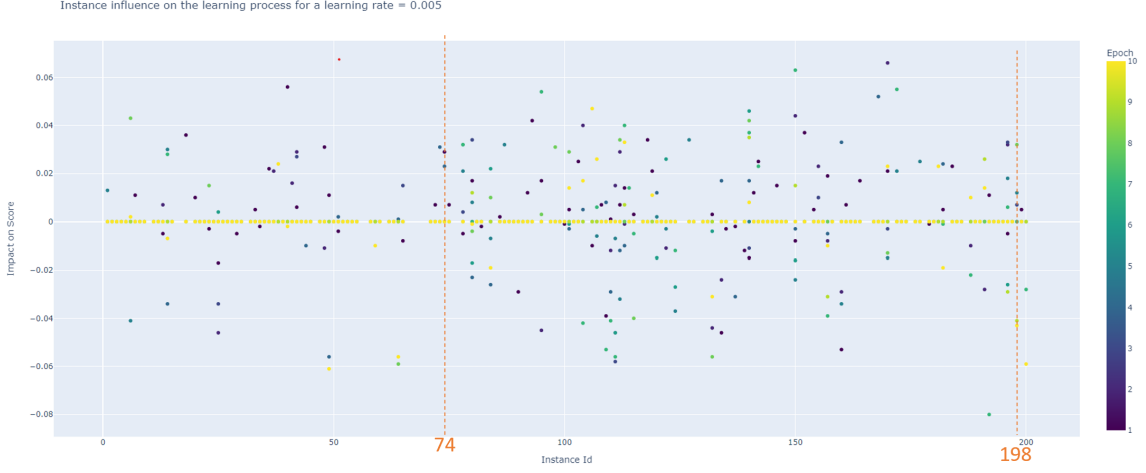


Figure 9: Impact on Score for each instances with $\delta = 0.005$

The second category refers to "medium quality" data. In other words, all the instances have both a positive or zero impact and a negative impact on the accuracy of the models. Most of the time, the number of positive impacts equals the number of negative ones. Instance 198, in both models ([Figure 8](#), [9](#)), generates a positive impact each time but also a negative one when it is considered. The majority of instances are in this category.

The last category logically constitutes the group of "low quality" instances. This category, therefore, contains instances that, most of the time, reduce our models' quality. In other words, these instances have, at most, one fairly weak positive impact. The rest of the values associated with them are less than or equal to zero.

From these conclusions, we can confirm that instances have different qualities. This implies that a sub-group of instances is potentially more helpful than the rest. However, this is a hypothesis. We need to verify this subset's existence.

4.3 Model Optimizing

Now that we understand the hyper parameter's influence on our model performance, we try to answer the problem about the instances subset existence (3).

4.3.1 Instance subset existence

We use the data collected during the previous analysis to prove the existence of this sub-group of examples that could potentially deliver good performance. As a reminder, when we aim to understand the quality of the instances, so we collect data to study the impact of each instance and display results in graphs 8 and 9. We also collect data for learning rates 0.01 and 0.05.

For each learning rate, we create a ranking of the instances. We, therefore, have four different rankings. The instances are ranked according to their average impact on the score. For example, for each instance in Graph 8 associated with a learning rate of 0.001, we average the values for the instance in question and rank them in descending order according to their average values. As a result, the instance in the first place has the most significant average impact. While the last one has the worst average impact.

We also modify the algorithm to take on an additional hyper-parameter called the "filter percentage." After making the changes, the prediction model works as follows.

Once the hyper-parameters and parameters are initialized, the algorithm begins by accessing the ranking associated with the assigned learning rate. It then selects a certain number of instances, starting with the best. This number is defined by the percentage assigned to the "filter percentage" hyper-parameter. This sub-group of examples then becomes the new training set that the algorithm used during its training. Assuming that the percentage chosen is 2%, the model's training set would contain the examples in the top 2% of the best instances in the chosen ranking. Note that resizing the training set also modifies our models' total number of iterations. For the 2% example, it means that the algorithm selects 2% of the 160 examples available to the model for training. So three instances. As a result, the model performs thirty iterations (3 x 10 epochs).

To prove the existence of a subset of more useful instances than the others, we again test four models with learning rates of 0.001, 0.005, 0.01, and 0.05, re-

spectively. For each of these, a series of different percentages are assigned to the "filter percentage" hyper-parameter. This approach aims to observe whether the new training also produced satisfactory performance. In short, to analyze if it was possible not to use all the data available.

4.3.2 Results & Discussions

Results are in graph [10](#), which shows the score of the different models as a function of the size of the training set they used. The size is expressed as a percentage of the data set available in the training set. For example, the value 0.1 on the x-axis means that the model uses a training set with a size equivalent to 10% of the original set. In our case, this represents a size of 16 examples ($160 / 10$). We compute the score on the x-axis as the average of the scores obtained by the model over 5% of its iterations. In other words, for a model with 1000 iterations, the score is calculated by averaging the scores obtained over the last 50 iterations.

In Graph [10](#), we can first see that the best performance is delivered by the learning rate 0.005 with a size of 80% and a score of 0.98. This accuracy is closely followed by 0.001 for the same training set size with a score of 0.97. While the worst is produced by the 0.001 training rate having a size of 2% with a score of 0.38. Unsurprisingly, the 0.001 and 0.005 values deliver the best performances. Indeed, given that they are already the values that gave the best models when we use all the data available. Consequently, the closer the size of the training set is to the original training set, the more likely it is that 0.001 and 0.005 give the best scores. This statement is also confirmed at 60% and above. On graph [10](#), it is easy to see that from 0.6, these models perform best.

Another interesting observation is that using a subset of instances produces better results for certain learning rates than using the whole data set. For example, the model with a learning rate of 0.005 (orange in Graph [10](#)). We can see that it obtains a score of 0.98 when it uses 80% of the data. When it uses 100% of the data, it obtains a score of 0.95. This phenomenon can also be observed for the value 0.05 (red in a graph [10](#)) when it obtains a score of 0.95 at 40% and 0.87 at 100%. From these results, we already can say that there is a first indication about a subset of instances that offer better performance. However, this is insufficient, as the aim is to use as few instances as possible. Consequently, we particularly observe in the results provided by the small percentages, below 20%.

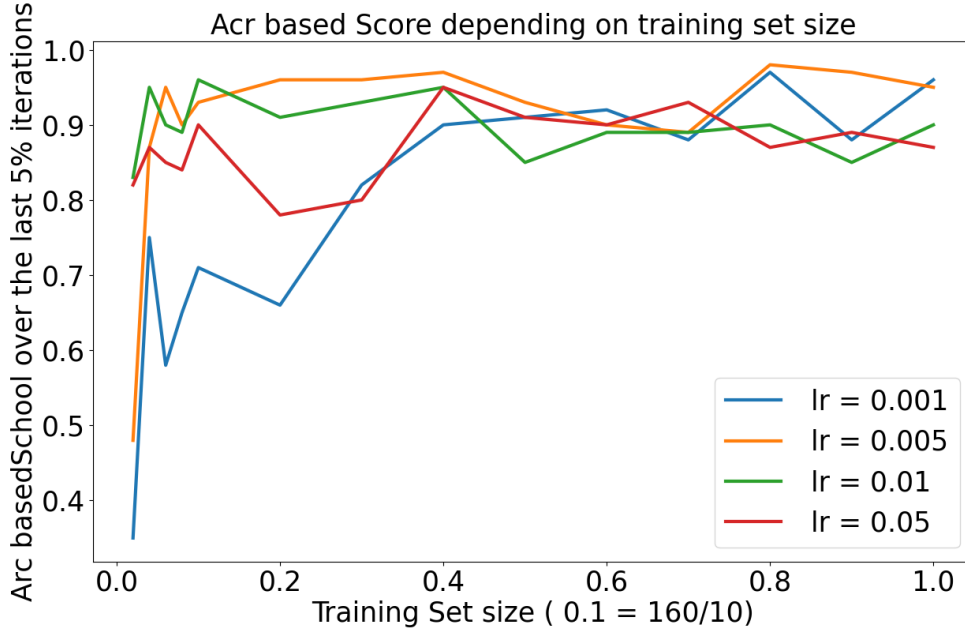


Figure 10: Arc-based Score depending on Training set size with different learning rate

The first thing to note is that of the four models, 0.001 provides a much lower accuracy than the other models. This is due to the small number of iterations associated with the small percentage. Indeed, the update of the weights at each iteration is small due to the learning rate equal to 0.001. Conversely, the updates are larger for the values 0.005, 0.01, and 0.05. Consequently, the first model inevitably needs more iterations than the others to converge toward a satisfactory result.

Then, as mentioned above, the other three models provide good results for small percentages. Their convergence is faster because the learning rates are higher. Nevertheless, although the values were 0.01 and 0.05, we show in the previous sections that the models linked to these values are unstable. The model for 0.005, on the other hand, performs well. In particular, for 4% and 6% where it obtains a score of 0.87 and 0.95, respectively. This shows that a group of instances can achieve fairly similar or even better accuracy than using all the examples.

With the different previous analyses, we can confirm the existence of a subset of more useful instances. However, the models are subject to certain conditions regarding their learning rate to optimize their accuracy. If the percentage of data used is tiny, the learning rate must be large enough to allow quick convergence.

In our case, 0.005 produce a stable model using only 6% of the available data and obtaining an accuracy of 95% on the test set. Nevertheless, this conclusion raises some questions. We prove the existence of a subset of examples is more useful than others, but we have no information about its properties. We use four different learning rates to prove the existence of a subset. However, we do not know whether the groups of examples depend on the learning rate used or whether they are the same regardless of the learning rate. We have tried to address this problem in the next chapter.

4.3.3 Subset Proprety

For this experiment, we focus only on models with a learning rate of 0.001 and 0.005 (Graphs [11](#) and [12](#)). There are two lines on each graph, each corresponding to a different model. The one whose legend associates it with a BIS learning rate means the model uses the unique ranking to select a training set. Conversely, the other model uses a ranking based on this learning rate. The unique ranking for each instance is equal to the average instance score throughout the different learning rate rankings. With these results, we would be able to analyze if the instance subset is changing depending on the model's learning rate.

4.3.4 Results & Discussions

The scores for the model with a learning rate of 0.001 (Graph [11](#)) show that using the unique ranking performs less well for percentages below 70. Above this value, both models tend towards the same efficiency. The higher the percentage, the greater the chance of selecting the best instances specific to the learning rate under consideration. Consequently, increasing the percentage reduces the impact of the main ranking compared with the rankings associated with the different learning rates. Nevertheless, the data for percentages below 20% give the most interesting results. So, with these observations, the model that does not use the unique ranking obtains better performances.

In a graph [12](#), the same phenomenon is also valid for the value 0.005. The shape of the data broadly follows what is shown in the previous graph. The difference between the two models appears mainly for small percentages and tends to diminish as these increase. One additional difference with Graph [11](#) is that the scores of the models converge at a size of 70% while the models for the value 0.005 (Graph [12](#))

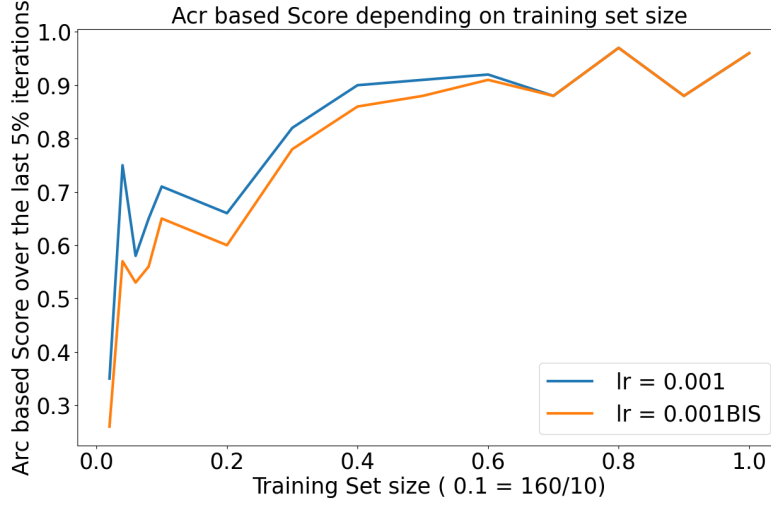


Figure 11: Subset depending on learning rate

converge earlier (50%). This is due to the difference in size between the two learning rates. Given that 0.005 leads to faster convergence, we can say that the model needs less data to produce a good result than for the value 0.001.

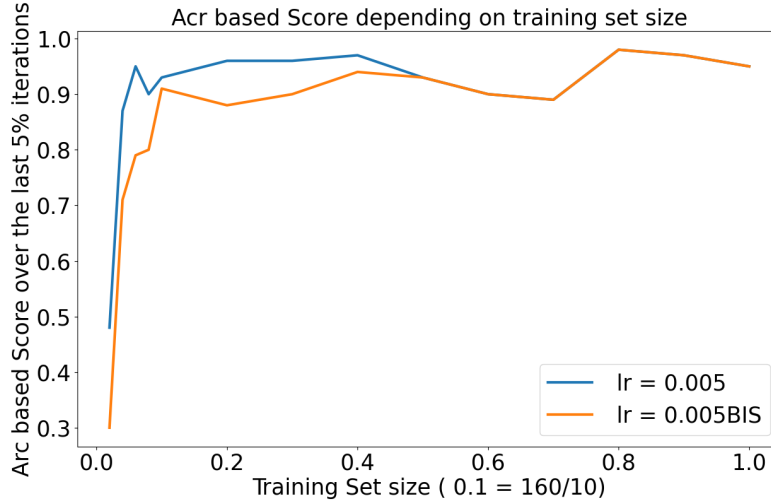


Figure 12: Subset depending on learning rate

From these observations, we can state that the best instances depend on the learning rate assigned to the algorithm. However, in the various experiments we have carried out so far, an underlying problem remains. This comes from the rankings we use to select the best examples. These rankings are obtained by training the models completely and then ranking them according to the average scores of the instances.

In the industry, we cannot afford to train a model to select the best instances and then use these examples to re-train the model because the whole process would be too costly. Consequently, the rankings represent metrics we do not normally access. Then, the objective is to optimize the algorithm so that it has its training set at the start of its learning process. In this way, we reduce the cost of the algorithm by combining the selection of the best instances and the learning of the model in a single training session.

4.3.5 New Instance Selection Method

this chapter and the order till the end has as the objective to answer the question about the best way to find this instance subset (3).

To begin with, we consider how the algorithm converges toward the real weights. In line 13, the algorithm uses the difference ($\Delta \Phi$) between two feature vectors (Φ). The first is created using the optimal solution for the instance under consideration. While the other is created using the prediction given by our PCTSP solver, for instance, under consideration. Therefore, the difference of feature vector computes how far the solution given by our PCTSP solver, using the current weights of our model, is from the optimal solution. This is why this value is used to update the weights of the learning model. We, therefore, choose to use this difference when selecting the best instances. We assume that the best instances have the computed difference closest to 0. Conversely, the worst instances have the highest values (positive or negative) for this metric.

To achieve this, we modify the algorithm so that it reduces the size of its training set fairly quickly. This method would be pointless if the instances are selected too late, as the model already performs many iterations. However, it was also important not to select the instances too early, given the random initialization of the weights at the start of training. Indeed, if we perform this selection during the first epoch, the difference between the feature vectors would be biased. Because of the random initialization of the weights, they can be very far from or very close to the real weights. So, if they don't converge to a minimum, the difference of each instance computed during the first epoch is biased.

Algorithm 2 Structured Prediction Algorithm

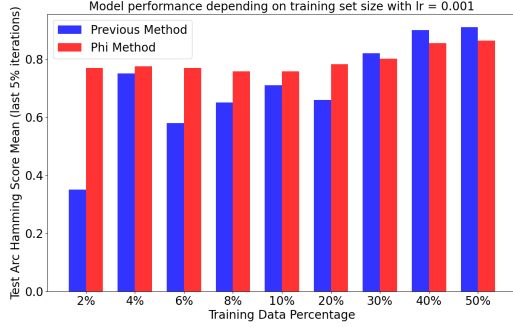
```
8 Initialize  $\mathbf{w}$ 
9 for  $e = 1$  to  $E$  do
10   forall  $(u, x) \in H$  do
11      $\hat{x} \leftarrow \arg \max_{\hat{x} \in X(u)} \mathbf{w}^T \Phi(u, \hat{x})$ 
12     if  $\hat{x} \neq x$  then
13        $\mathbf{w} \leftarrow \mathbf{w} + \delta(\Phi(u, x) - \Phi(u, \hat{x}))$ 
14     end
15   end
16 end
17 return  $\mathbf{w}$ 
```

Assuming we have a random initialization that generates weights far from the real weights. The whole point is, therefore, to allow the model to converge sufficiently towards a certain point at which learning becomes more difficult. This is the threshold we discussed in the experiments in chapter 4.2.2. Once this point is reached, we compute the the difference between feature vectors of each instance and then ranked them in descending order. After that, the model modifies its training set with this new ranking according to the percentage that is assigned to it. In addition, each ranking is different depending on the model's learning rate. We set it up this way because, as demonstrated in previous experiments, the subsets differ according to the learning rate.

4.3.6 Results & Discussions

We limit our experiments exclusively to small percentages because the objective is to learn preference sees with the smallest training set possible. Therefore, studying the results with percentages higher than 50% would not make much sense. We test models using this selection method with the four learning rates we have used until now (Graph 13). The new models select instances considering remarks on convergence made earlier. Consequently, the first epoch is carried out normally to allow convergence until learning becomes difficult. Then, the second and third epochs are also carried out with all the data. However, the difference between the feature vectors of each instance is stored during these epochs. When these epochs are completed, the algorithm computes each instance's average $\Delta \Phi$. This establishes a ranking of the algorithm used to select instances according to their assigned percentage. Once the selection had been made, the rest of the training was carried

out using this new training set.



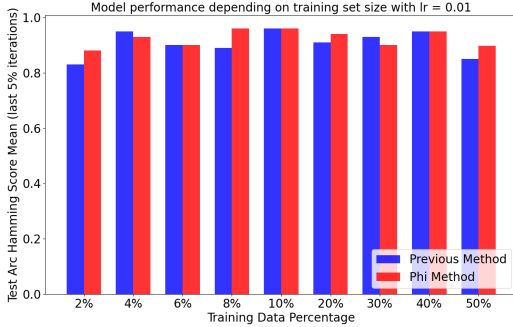
(a) With $lr = 0.001$



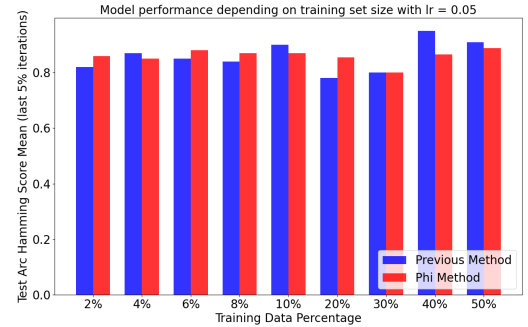
(b) With $lr = 0.005$

Figure 13: Models Performances using Phi Selection Method

We show the results in figures 13 and 14. First, we can see that for the learning rate 0.001 (Graph 13a), the new selection method produces better results than the old one. However, this difference tends to diminish as the number of examples selected increases. The difference in performance is particularly striking for 2%. Therefore, using the Phi provides a better selection than previous experiments. This observation is repeated for the 0.005 learning rate (Graph 13b), where the new selection method also generates better results than the old one for small percentages.



(a) With $lr = 0.01$



(b) With $lr = 0.05$

Figure 14: Models Performances using Phi Selection Method

However, this phenomenon is not systematic for larger learning rates (0.01 14a and 0.05 14b). Once again, this is explained by the updates between each iteration, which are greater than when using lower learning rates. As a result, even if the best instances are not selected, the threshold will be reached fairly quickly. This is true regardless of the method used.

Regarding overall performance, we can say that the best scores are achieved when using the value 0.005. The effectiveness of this selection method is undeniable for small percentages. From this observation, we can conclude that it is possible to reduce the costs generated by selecting the best instances, and the feature vectors selection method to build subsets is a better method. We have therefore decided to explore this avenue further in the next chapter.

4.3.7 Phi-based Selection Method

To further develop the selection of the best instances using the Φ vectors. In the previous experiments, the newly implemented method used three complete epochs before sorting into its training set. By complete epochs, we mean performed on the entire training on the original training set. Furthermore, we aim to reduce the number of iterations performed during the training of our model as much as possible. For this reason, we implement three other methods using feature vectors that perform fewer iterations than the first method.

Firstly, we start from the operation of the first method, which uses the information gathered during epochs 1 and 2 to sort the instances. An interesting variant of this method would be using the information gathered in Epoch 1 to carry out several experiments for selection. In this way, this method only uses two complete epochs before reducing its training set. We name this method "Phi Epoch 1". The first method is called "Phi Epoch 1-2".

Secondly, we also want to change how we perform the selection. In the two previous methods, the best instances were ranked during epoch 1 (or 1-2). There was an inequality in how each difference between the feature vectors of the different examples are calculated. This was because the weights change between each instance. Consequently, it is possible that the fact that the instances are evaluated with slightly different weights led to a particular bias in the data collected. To test this theory, we create a third method called "Phi Weights," which works as follows:

- Performs epoch 0 and stores the instance with the smallest $\Delta \Phi$.
- Finds the weights used for this instance
- Performs epoch one by evaluating each instance with the weights retrieved in the previous step and sorts according to the information gathered.

- After reducing the training set, performs the rest of the training normally using the reduced set.

Using the same weights to evaluate all the instances, we can put all the examples equally before calculating their $\Delta \Phi$. An additional question remains.

Until now, we always use methods based on a certain logic. Here, on the fact that the best instances have a difference between their feature vectors close to zero. However, we haven't yet empirically proved the effectiveness of this compared to basic logic. For this reason, we have created a fourth method called "Random Choice."

This method works quite simply. Immediately after the random initialization of the model weights, the algorithm randomly selects the instances until it constitutes the subset according to the percentage allocated to it. Selection is, therefore, independent of the learning rate associated with the model. The performance of this method allows us to see whether the logic used in the first three methods was relevant. Indeed, if, in the end, a purely random selection based on no heuristics is more efficient than our other methods, we could conclude that it is not relevant to continue the optimization using our logic based on feature vectors.

4.3.8 Results & Discussions

For these experiments, we collect the results from four categories of models (Figures [15](#), [16](#), [17](#))—one for each learning rate tested. Then, for each category, we have four different models. Each model, therefore, uses a different selection method. Finally, we reuse the way we tested the effectiveness of the methods in the experiments in the previous chapter. The x-axis shows contains the percentage of data used to sort instances within the different methods. The y-axis shows the average Arc-based score computed over the last 5% of iterations.

Concerning the value 0.001 (Graph [15](#)), the best accuracies are obtained by the fourth method for percentages equal to or greater than 20%. This method also produces the worst performance, but this time for 2%. We also observe that the random method performs better than the others for large percentages. Given that the instances are selected randomly, increasing the number of instances selected increases the chances of recovering those with the best performance. In addition, these results seem to demonstrate that the methods using feature vectors do not seem relevant when the percentages exceed a certain order of magnitude. Nevertheless, to verify

this hypothesis, we analyze whether it is also valid for the other categories of models. The aim is to check whether this phenomenon is not a special case associated with the 0.001 learning rate.

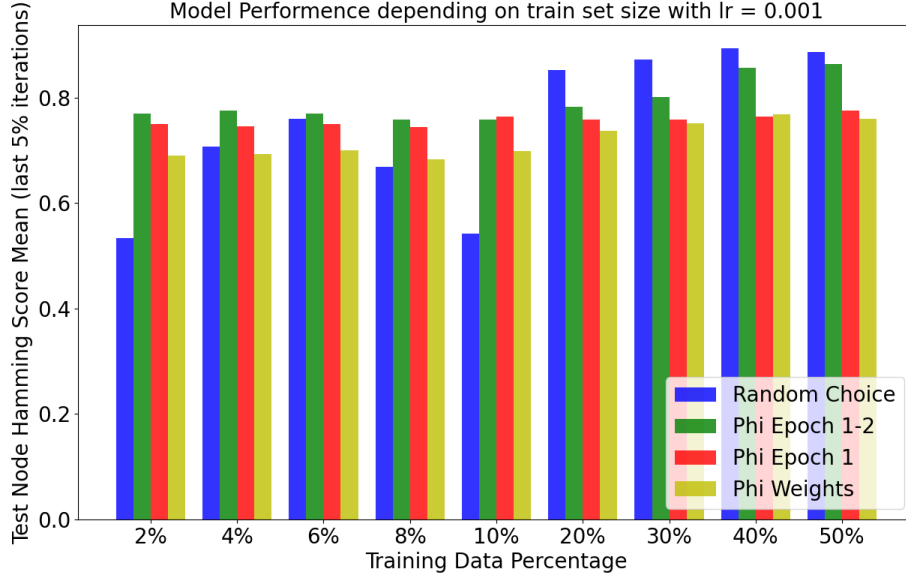


Figure 15: With $lr = 0.001$

However, contrary to this hypothesis, other methods perform better than random selection for percentages below 20%. The best of these methods is "Phi Epoch 1-2", which achieves an accuracy of ± 0.78 for percentages ranging from 2% to 10%. For these small sizes ($<20\%$), apart from random selection, it can also be seen that the ranking of the best methods between the different percentages remains similar. "Phi Epoch 1-2" performs best, followed by "Phi Epoch 1" and "Phi Weights," which still achieves the lowest accuracy.

For the 0.005 learning rate (Graph [16](#)), we observed results broadly similar to those previously analyzed. Firstly, the best performance is provided by "Phi Weights" for 20%. Whereas the random method obtains the worst accuracy when used for 2%. The similarities are that the random method performs worst for 2%, 4%, and 6%. Despite this, certain differences remain. The first concerns the 8% and 10% sizes. In contrast to the 0.001 learning rate, random selection achieves fairly high accuracy, catching up with the other methods. We can also see that the ratio between the scores of the first three methods changes for these sizes. That is, "Phi Epoch 1" followed by "Phi Epoch 1-2" and "Phi Weights."

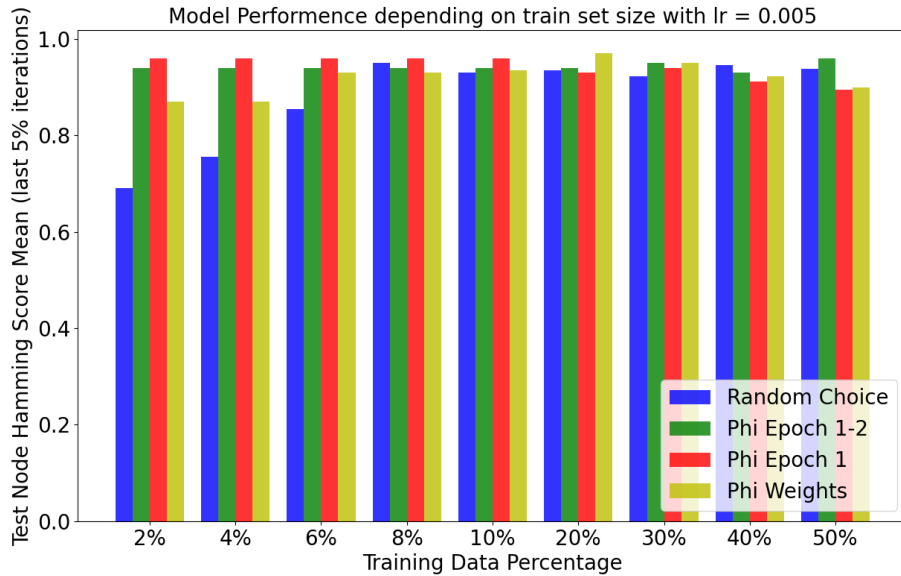


Figure 16: With $lr = 0.005$

As far as the higher percentages are concerned, even if this method is not the best, it still provides an almost equivalent accuracy to that of the other methods. This is because this method, unlike the others, does not perform a complete Epoch. It can therefore be seen that, once again, it is the most appropriate method for large percentages.

In Graph [17](#), for the other two learning rates, random selection obtains a better score for the small percentages than for the 0.001 and 0.005 learning rates. Furthermore, this method does not seem to follow any fixed rules for the different percentages and performs as well as or better than the other methods. In some cases, its performance drops. For example, for 10% in figure [17b](#), its performance is below 0.8. Whereas for 8%, it obtains a performance of almost 0.9. We also can observe the same kind of behavior for the learning rate 0.01 (Graph [17a](#)). For this value, the best performance is obtained via the random method for 30%. Whereas for learning rate 0.005 (Graph [17b](#)), the best score is associated with the "Phi Epoch 1" method for a size equivalent to 20% of the initial training set.

However, this seems to be an anomaly as it is the only time the "Phi Epoch 1" method produces the best score for large percentages. Consequently, we decide not to take it into account at the risk of biasing our analysis. In addition, the mixed results associated with 0.01 and 0.05 are much more likely than the others to produce results that may appear disordered. Because of the learning rates are quite large, we conclude that models need very few iterations to converge toward the threshold at

which training becomes difficult. Once this threshold is reached, because the weight updates at the end of each iteration are fairly large, the model is much more likely not to converge toward the solution. In other words, we say that it risks going back and forth between predictions on either side of the solution without ever converging toward it.

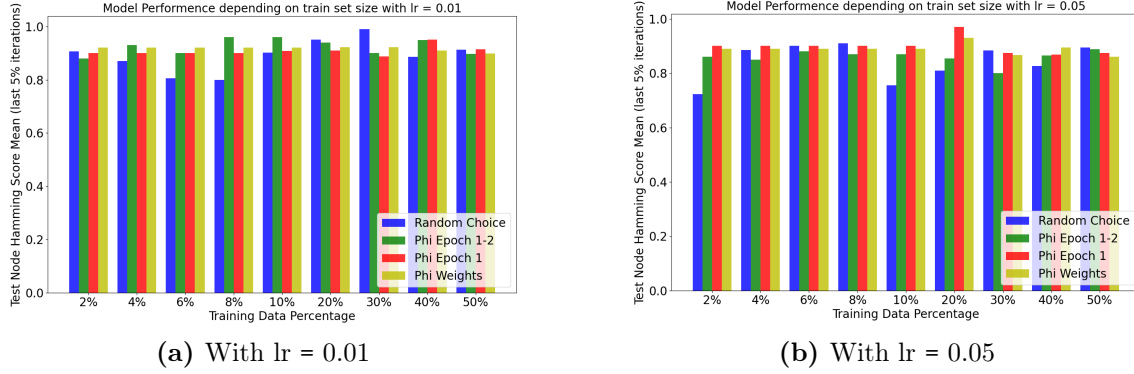


Figure 17: Models Performances using several Phi Selection Methods

From these analyses, we confirm several things. First, it has enabled us to prove that there is a method for reducing the number of examples used by our learning model during training. However, this method is subject to some conditions. The first is that to be better than a simple random selection, the method performance depends on the learning rate used. In our case, we observe that when the learning rate exceeded 0.005, random selection produces equivalent results to the other methods. The second condition is that the reduction we want to achieve is large enough. For the values 0.001 and 0.005 (Graph [15](#) & [16](#)), the methods using feature vectors were only relevant for small percentages.

Despite the experiments in this chapter, one final factor could call into question our various conclusions. Since the beginning of this work, we use the perceptron algorithm, which learns the data preferences using weights. So, until now, the weights have been initialized randomly. The problem is that this random initialization can potentially bias our results.

In an example where the model initializes the weights, which by chance, happens to be close to the actual weights. In this case, the model automatically produces better results than it does. It only needs a few iterations to converge towards the threshold at which learning is difficult and then select the instances. Conversely, if the weights are initialized as far as possible from the real weights, our methods

may provide poor scores. This is because the model needs more iterations to converge. Therefore it is possible that, in reality, our methods using feature vectors do not allow sufficient convergence before selecting instances. This results in a poor selection. If this happens, then it's also possible that our methods are less effective than random selection, which would make them pointless. We tried to verify this hypothesis in the next chapter.

4.3.9 Weights Initialization Influence

As explained above, we then have to look at the potential influence of how the weights in our model are initialized. The aim is to visualize the effect that the weights had on the performance of our methods when they are initialized as far as possible from the real weights. To do this, we carry out the same type of experiment as in the previous chapter, changing how the model weights are initialized.

The weights are no longer initialized randomly for these new experiments but followed a specific process. In our case, the weights are previously initialized randomly by taking a value between 0 and 100. We use this interval to have models where the weights are initialized as far as possible from the real weights. As a result, each weight in the model was initially set to a value equal to the furthest limit of the corresponding real weight. For example, if one of the real weights is 23, the related model weights are initialized to 100. Conversely, if the real weight is 57, the model weights are initialized to 0. Using these results, we can see the effect of this type of initialization on model performance. We then compare these results with those obtained in the previous chapter to determine whether the conclusions drawn from the earlier experiments are relevant.

4.3.10 Results & Discussions

We, therefore, collect the results from the experiments explained above and put them on the same graph. We then compare them with the data from a random initialization of the weights. As a reminder, the data from a random initialization of the weights are those we collected in chapter [4.3.8](#).

Once again, we separate the data according to the learning rate assigned to their model. There are four different colors to represent the different methods used. However, we add another feature to distinguish between random and arbitrary initialization. Points with a round shape are results obtained from models with random

initialization of their weight. On the other hand, the points with a square shape are data obtained from models with a weight initialization as described in [4.3.9](#). Note that we focus mainly on the 0.001 and 0.005 learning rates.

Firstly, for the 0.001 learning rate (Graph [18](#)), we can see that the "Phi Epoch1-2" method obtains the best accuracy for small percentages. This is true regardless of how the weights are initialized. Regarding the weights, we can also observe that "Phi Epoch1-2" does not vary greatly in accuracy depending on how they are initialized. At least for small sizes. We can also conclude that this method is closely followed by "Phi Epoch1", Furthermore, the ratio between the two methods remains equivalent regardless of comparison, with "Phi Epoch1-2" remaining significantly better.

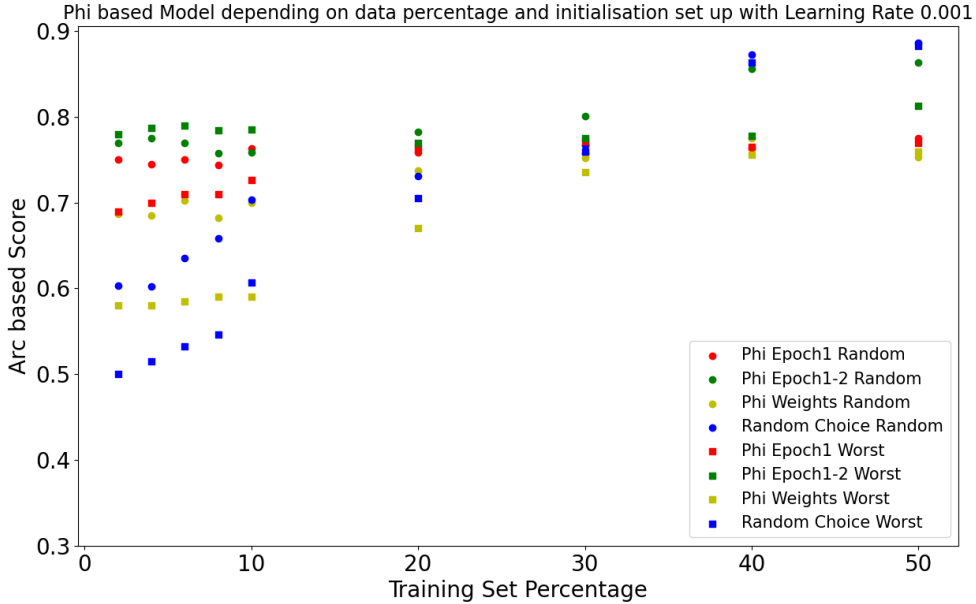


Figure 18: Weights Initialization influence on methods performances

However, the color difference between these methods diminishes as the percentages increase. Unlike the first method, "Phi Epoch1" appears to be more sensitive to the initialization in which the weights are in. Especially for small sizes. On the other hand, the score obtained when the weights are in the worst possible way is quite lower than the score obtained when they are not.

The differences in behavior between these methods seem to come from the number of iterations they perform before sorting their training set. "Phi Epoch1-2" is, therefore, less sensitive to weights initialized in the worst way because it allows

more iterations than "Phi Epoch1" regarding the model of the weight to converge sufficiently for sorting to be as good as possible. Despite this, worst initialized methods achieve significantly better accuracy for small sizes than the random selection method. As for the "Phi Weights" method, it doesn't seem very conclusive, as we can see that its accuracy is very sensitive to how the weights are initialized. It was also less effective than random selection for certain percentages.

For value 0.005 (Graph 19). Firstly, we observe that the accuracy obtained with this learning rate is all higher than the previous one. This observation is particularly significant for small sizes, where the best scores are around 0.9, regardless of how the weights are initialized. Whereas for the previous value, the best scores for these percentages are just under 0.8. The interesting point is that the methods are the same regardless of how the weights are initialized. We explain this difference in behavior with the previous case by the fact that 0.005 is larger than 0.001. This difference in size means that the model in Graph 19 can update its weights more frequently at the end of each iteration. The models, therefore, converge faster and require fewer iterations before selecting the optimal instances. Nevertheless, it is still observable that our methods perform better than random selection. Particularly for small sizes.

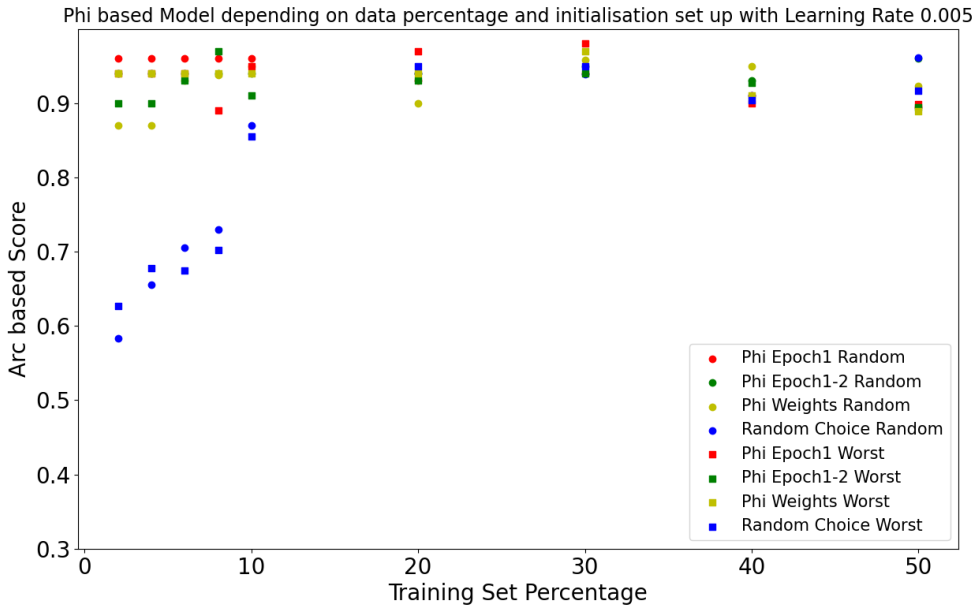


Figure 19: Weights Initialization influence on methods performances

Another difference is that the "Phi Weights" method gives better results than in the previous case. For large percentages, we also found that the random method

no longer offered the best accuracy except for 50% and that all the methods are less sensitive to how their initialized weights than the case associated with the 0.001 learning rate. We also explain this effect by the difference in magnitude between 0.005 and 0.001.

From these different observations, we learn several things. Firstly, even if how the weights are initialized influences our methods, they still perform better than a simple random selection. Particularly for small percentages where the difference is quite significant. However, we can see that the effectiveness of our methods will depend on the learning rate. Particularly when the number of instances to be selected increases. As we have seen with 0.001 (Graph [18](#)), random selection performs better for 40% and 50%. Nevertheless, it is possible to cancel this condition by increasing the learning rate, as in Figure [19](#) associated with 0.005.

5 Conclusion

Based on the results obtained during this work, we answered the various questions posed in the description of our main problem. Firstly, we show that certain values for the hyper-parameters give better performance. In particular, a learning rate of 0.005 accompanied by a sufficient number of epochs allows the model to converge sufficiently.

From the results obtained during the experiments, we conclude that there is a subset of instances that is more useful than others during the training of our model. Furthermore, the latest experiments prove that we can use this subset to learn preferences within the data accurately.

In conclusion, we prove that learning preferences in a production planning problem with a small amount of data is possible.

However, following this work, further experiments could be carried out. In this work, we use PCTSP instances with a size of 10. It could be interesting to test whether the observations we have made are also valid for instances with other sizes (20, 30, etc...) and observe the influence of the change in size on the performance of our model. A second idea would be to find other methods that don't use feature vectors to find the subset of instances and compare the results with our data. A final experiment could be to create a metric that can be applied directly to the instances, and that calculates a score giving the quality of each instance. In this way, it would not be necessary to visit all the instances at least once before selecting.

References

- [1] A.Özgür, Y.Uygun, and M-T.Hütt, *A review of planning and scheduling methods for hot rolling millq in steel production*. Computers & Industrial Engineering, 2020.
- [2] F. Rossi and P. van Beek, *Handbook of Constraint Programming*. Elsevier, 2006.
- [3] S. Nowozin and C. Lampert, *Structured Learning and Prediction in Computer Vision*. Foundations and Trends, 2011.
- [4] P. Garner, D. Imseng, and T. Meyer, *Automatic Speech Recognition and Translation of a Swiss German Dialect: Walliserdeutsch*. Idiap Research Institute, 2014.
- [5] M. Torrisi, G. Pollastri, and Q. Le, *Deep Learning methods in protein structure prediciton*. Elsevier, 2020.
- [6] H. Daumé, J. Langford, and D. Marcu, *Search-based structured prediction*. Springer Science, 2009.
- [7] R. Canoy, V. Bucarey, Y. Molenbruch, M. Mulamba, J. Mandi, and T. Guns, *Probability estimation and structured output prediction for learning preferences in the last mile delivery*. arXiv, 2022.
- [8] V.Ceikute and C.S.Jensen, *Routine service quality-local driver behaviour versus routing services*. 14th International Conference on Mobile Data Management, 2013.
- [9] Y.Li and W.Phillips, *Learning from route plan deviation in last mile delivery*. Massachussets Institute of Technology, 2018.
- [10] J.Li, D.Wu, and M.Mao, *Recommender System Application Developments*. Decision Support Systems 74, 2015.
- [11] T.Joachims, *optimising search engines using click through data*. In proceedings of the 8th ACM SIGKDD international conference on knowledge discovery and data mining, 2002.
- [12] R.Canoy and T.Guns, *Vehicle routing by learning from historical solutions*. International Conference on Principles and Practice of Constrained Programming, Springer, 2019.

- [13] J.Mandi, R.Canoy, V.Bucarey, and T.Guns, *Data-driven VRP a neural network model to learn Hayden preferences for VRP*. in 27th International Conference on Principle and Practice of Constrained Programming(CP2021), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
- [14] L. ans Y.Chen ans A.Langevin, *An inverse optimization approach for capacitated vehicle routing problem*. European Journal of Operational Research, 2021.
- [15] C. E. Miller, A. W. Tucker, and R. A. Zemlin, *Integer programming formulation of traveling salesman problems*. Journal of the ACM (JACM), 1960.
- [16] G. Laporte, *The traveling salesman problem: An overview of exact and approximate algorithms*. European Journal of Operational Research, 1992.
- [17] M. ans S.Papert, *Perceptrons : An Introduction to Computational Geometry*. MIT Press, 1969.
- [18] P. Vansteenwegen and A. Gunawan, *Orienteering Problems : Models and Algorithms for Vehicle Routing Problems with Profits*. Springer, 2019.
- [19] R. Hamming, *Coding and Information Theory : Second Edition*. Prentice-Hall, Inc., 1986.
- [20] T.Guns, *CPMpy:Constraint Programming and Modelling in Python*. <https://cpmpy.readthedocs.io/en/latest/>, 2021.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl