Online Shortest Path with Machine Learned Advice

Supervisor:
Prof. Flavio Chierichetti

Student:
Rocco Lo Conte

SAPIENZA
UNIVERSITÀ DI ROMA

SAPIENZA
UNIVERSITÀ DI ROMA

Online Shortest Path with Machine Learned Advice

FACULTY OF INGEGNERIA DELL'INFORMAZIONE, INFORMATICA E STATISTICA

DEPARTMENT OF INFORMATICA

DEGREE IN COMPUTER SCIENCE

Supervisor:
Prof. Flavio Chierichetti

Student:
Rocco Lo Conte, 1946073

Academic Year 2022/2023

To my family: Filomena, Gaia and Vincenzo

Abstract

The shortest path problem is a fundamental and well-established challenge in computer science. In this paper, we investigate this problem in a novel setting where multiple experts provide real-time traffic information. Our primary objective is to identify the shortest travel time based on the aggregated expert advice. We consider two distinct scenarios: one involving directed graphs and the other incorporating undirected graphs. For the latter, we develop an algorithm with rigorously established performance guarantees under certain assumptions. Our main focus is the directed graph where we delve into the utilization of predictive techniques to overcome the worst-case performance limitations of existing approaches. To address this challenge, we devise a heuristic that surpasses the performance of the best-known online algorithm for the problem when provided with high-quality advice while maintaining its effectiveness even when the advice is inaccurate. We substantiate our findings by conducting an empirical evaluation of our heuristic on real-world data sets, demonstrating its practical applicability.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

Machine learning has achieved remarkable success in various domains, yet its practical deployment still faces significant challenges, particularly in handling inevitable errors. These errors may stem from the fact that most machine learning models are trained to optimize average performance, potentially leading to increased error on outliers. Machine learning aims to predict the unknown, while online algorithms guide decision-making without future knowledge. These algorithms strive to perform near-optimally under worst-case scenarios, but this focus on robustness can lead to conservative choices, resulting in high competitive ratios even for seemingly simple problems.
This paper explores the potential of combining machine learning with online algorithms to address the shortest traffic time problem. In this setting, a source, a destination, and k experts providing path suggestions are given. The goal is to find the shortest traffic time; for a more detailed discussion of the problem, please refer to Section 2. While the straightforward method is to follow the suggestion of the expert that suggests the minimum travel time, the performance of such an approach may be hindered by imperfect expert estimations. The expert's estimation of an edge's travel time may vary depending on the day of the week, time of day, and whether or not it is a holiday. This implies that the expert's advice may be inconsistent, making it challenging to rely solely on their recommendations. To address this, we investigate the effectiveness of incorporating machine learning predictions to improve the overall performance of online algorithms, distinguishing between bad and good advice.
We consider two scenarios:

- undirected graph: $w_{i,j} = w_{j,i;}$
- directed graph: It may happen that $w_{i,j} \neq w_{j,i.}$

In the directed setting, following a single expert consistently yielded approximate competitive ratios between 1 and 2, depending on various factors like the chosen expert, the time of the day and the date. However, by following the expert suggesting the minimum estimated time of arrival (ETA) or following the machine-learned predictions, the approximate competitive ratio could be reduced to 1. We analyzed the performances with different levels of traffic hours. Our proposed approach, which carefully integrates machine learning predictions into the online algorithm, demonstrated robust performance, achieving the best competitive ratio across all scenarios.
The reasons why we studied also the undirected setting can be found at Section 3.

The results of this study suggest that they are independent of the time of the day that is chosen and that, given enough data, achieves a better competitive ratio than following the single best expert in hindsight (Section 4). Nonetheless, this finding may be influenced by the methodology employed to determine the actual travel times (Section 2).
Our approach adheres to the principles of the OMLA model [1]:

- *Minimal Assumptions*: We avoid making restrictive assumptions about the machine learning predictor to use better predictors in the future;
- *Robustness*: The algorithm consistently outperforms traditional methods, even with imperfect predictions;
- *Worst-Case Competitiveness*: The algorithm maintains competitive performance even in the presence of inaccurate oracle predictions.

In summary, our study provides compelling evidence that combining machine learning with an online algorithm (Section 3) can significantly improve the performance of shortest-path algorithms, particularly for the crucial task of predicting traffic times. Our approach addresses the inherent challenges of using imperfect machine learning predictions and achieves robust performance across different traffic conditions. This work paves the way for more effective and adaptable navigation solutions in real-world scenarios.

1.1 Related Work

Augmenting Online Algorithms with Machine Learned Predictors. Our work builds upon previous efforts to enhance online algorithms with external information. Mahdian et al. [2] introduced a meta-algorithm that interpolates between the worst-case and optimistic competitive ratios, while Lykouris and Vassilvitskii [1] established the Online with Machine Learned Advice (OMLA) model, providing a theoretical foundation for combining online algorithms with machine learning predictions. Various other problems have been tackled in this model with encouraging outcomes, like scheduling with restricted assignments [3] and facility location [4]. Our work extends the OMLA model to the shortest traffic time problem, demonstrating the effectiveness of incorporating machine learning insights into online algorithms.

Learning from Expert Advice. This is a famous problem in the learning community. "Consider the following fundamental iterative decision-making problem: at each time step $t = 1, 2, ..., T$, the decision maker faces a choice between two actions A or B (i.e., buy or sell a certain stock). The decision maker has assistance in the form of N "experts" who offer their advice. After choosing between the two actions, the decision maker receives feedback in the form of a loss associated with each decision. For simplicity one of the actions receives a loss of 0 (i.e., the "correct" decision) and the other a loss of 1. We make the following elementary observations:

- A decision maker that chooses an action uniformly at random each iteration trivially attains a loss of $\frac{T}{2}$ and is *correct* 50% of the time.
- In terms of the number of mistakes, no algorithm can do better in the worst case!

We are thus motivated to consider a *relative performance metric*: can the decision maker make as few mistakes as the best expert in hindsight? The next theorem shows that the answer in the worst case is negative for a deterministic decision-maker.

Theorem 1. *Let $L \leq \frac{T}{2}$ denote the number of mistakes made by the best expert in hindsight. Then there is no deterministic algorithm that can guarantee less than $2L$ mistakes.*

Proof. Assume that there are only two experts and one always chooses the option A while the other always choose option B. Consider the setting in which an adversary always chooses the opposite of our prediction (she can do so, since our algorithm is deterministic). Then, the total number of mistakes the algorithm makes is T. However, the best expert makes no more than $\frac{T}{2}$ mistakes (at every iteration exactly one of the two experts is mistaken). Therefore, no algorithm can always guarantee less than $2L$ mistakes.

$\square$

This observation motivates the design of random decision-making algorithms.

Theorem 2. *Let $\varepsilon \in (0, \frac{1}{2})$. Suppose the best expert makes L mistakes. Then:*

- *There is an efficient deterministic algorithm that can guarantee fewer than $2(1+\varepsilon)L + \frac{2logN}{\varepsilon}$ mistakes;*
- *There is an efficient randomized algorithm for which the expected number of mistakes is at most $(1+\varepsilon)L + \frac{logN}{\varepsilon}$.*

The problem of prediction from expert advice was devised by Littlestone and Warmuth (1989, 1994)." [5]

2 Preliminaries

This section commences with a comprehensive definition of the problem at hand. Subsequently, we delve into the fundamental concepts of machine learning and online algorithms, providing a solid foundation for our subsequent discussions.

The Problem. We are provided with the source and destination locations specified in latitude-longitude coordinates, along with a specific date and time of day. The objective is to determine a path that minimizes the total travel time in retrospect. To achieve this, we employ the expertise of $k \geq 2$ experts, each capable of answering the following question:

- Provide the shortest path from the source to the destination, including an estimate of the arrival time in seconds and the distance in meters.

The provided information by the experts may not always be accurate. As we traverse the path from the source to the destination, we encounter actual travel times for each edge. This real-time information can be leveraged to refine our prediction model and adjust the weights assigned to each expert's recommendations. By incorporating this feedback, we can effectively penalize experts whose advice proves to be less reliable.

Technical Assumption. As we cannot directly measure the actual travel time between two points, we utilize the expert providing the lowest estimated arrival time for a specific path as our de facto ground truth. This approach allows us to evaluate the performance of our prediction model without relying on perfect real-time data. Thus, we need $k \geq 3$ experts.

As outlined in [6], a route comprises a series of *legs*, each of which entails a succession of discrete *steps*. The step constitutes the most elementary unit of a route. While the error associated with a single-step prediction may appear insignificant, accumulating such errors over a series of consecutive steps can lead to significant discrepancies in the overall estimated travel time.

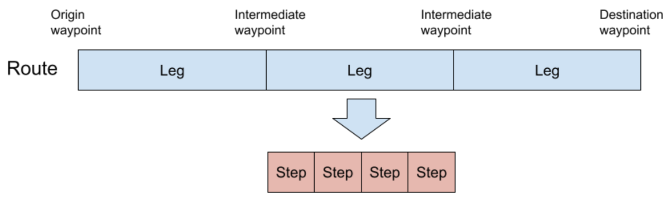


Figure 1: Composition of a route [6].

The objective is to identify a path that minimizes the total travel time, ensuring that it does not exceed $O(poly(k) \cdot T)$, where T is the optimal minimum travel time.

Machine Learning Basics. As described in [1], we are given a feature space X, describing the estimation of the information of each edge of the graph G and a set of observations Y. An example is a pair (x, y), where $x \in X$ describes the specific features of an edge e, and $y \in Y$ gives the corresponding observation. A hypothesis is a mapping $h : X \to Y$ and can be probabilistic in which case the output on $x \in X$ is some probabilistically chosen $y \in Y$. To measure the performance of a hypothesis, we first define a loss function $\ell : Y \times Y \to R^{\geq 0}$. When the labels lie in a metric space, we define absolute loss $\ell_1(y, \hat{y}) = |y - \hat{y}|$ and squared loss $\ell_2(y, \hat{y}) = (y - \hat{y})^2$.
To evaluate the performance of a predictor on a particular input, we consider a loss function. Similar to the prediction model, selecting a loss function involves trade-offs between the learnability of the predictor and resulting performance guarantees. For a given loss function ℓ, problem P, and prediction model H, the performance of the predictor $h \in H$ on input $\sigma \in P$ is evaluated by its error

$$\eta_\ell(h, \sigma) = \textstyle\sum_i \ell(y(\sigma_i, h(\sigma_i).$$

In full generality, this error can depend on the whole input in complicated ways. For the shortest path problem, the prediction model we consider predicts the subsequent time that a requested element will get requested again. In this case, a natural loss function such as absolute or squared loss decomposes the error across edges. In Section 4, we discuss the performance of the predictor and, based on our findings, the squared loss seems to be the best choice.

Competitive analysis. As mentioned in [1], to obtain worst-case guarantees for an online algorithm (that must make decisions when elements arrive), we compare its performance to that of an offline optimum (with the benefit of hindsight). Let σ be the input sequence of elements for a particular online decision-making problem, $cost_A(\sigma)$ be the cost incurred by an online algorithm A on this input, and $OPT(\sigma)$ be the cost incurred by the optimal offline algorithm. Then algorithm A has competitive ratio CR if, for all sequences σ,

$$CR = \tfrac{cost_A(\sigma)}{OPT(\sigma)}.$$

To bind the performance of an online algorithm on every input, we say that the algorithm has a x-approximation if, for all sequences and $0 < \varepsilon < 1$,

$$CR = x + \varepsilon.$$

Online Convex Optimization. "In Online Convex Optimization (OCO), an online player iteratively makes decisions. At the time of each decision, the outcome or outcomes associated with it are unknown to the player. After committing to a decision, the decision maker suffers a loss: every possible decision maker beforehand. The losses can be adversarially chosen, and even depend on the action taken by the decision maker. This frameworks have the following restrictions:

- The losses determined by an adversary should not be allowed to be unbounded. Otherwise, the adversary could keep decreasing the scale of the loss at each step and never allow the algorithm to recover from the loss of the first step. Thus, we assume that the losses lie in some bounded region.

- The decision must be somehow bounded and/or structured, though not necessarily finite. To see why this is necessary, consider decision-making with an infinite set of possible decisions. An adversary can assign high loss to all the strategies chosen by the player indefinitely while setting apart some strategies with zero loss. This precludes any meaningful performance metric.

The OCO framework models the decision set as a convex set in Euclidean space denoted as $\mathcal{K} \subseteq \mathbb{R}^n$. The costs are modeled as bounded convex functions over $\mathcal{K}$. In stochastic optimization, the optimizer attempts to minimize a convex function over a convex domain as given by the mathematical program, as follows. At each iteration t, the online player chooses $x_t \in \mathcal{K}$. After the player has committed to this choice, a convex cost function $f_t \in \mathcal{F} : \mathcal{K} \rightarrow \mathbb{R}$ is revealed. Here, $\mathcal{F}$ is the bounded family of cost functions available to the adversary. The cost incurred by the online player is $f_t(x_t)$, the value of the cost function for the choice x_t. Let T denote the total number of game iterations. Let $\mathcal{A}$ be an algorithm for OCO, which maps a certain game history to a decision in the decision set:

$$x_t^{\mathcal{A}} = \mathcal{A}(f_1, ..., f_{t-1}) \in \mathcal{K}.$$

We define the *regret* of the decision maker to be the difference between the total cost it has incurred and that of the best-fixed decision in hindsight. Formally, the regret of $\mathcal{A}$ after T iterations is

$$Regret_T(\mathcal{A}) = sup_{(f_1,...,f_T \subseteq \mathcal{F})}(\textstyle\sum_{t=1}^{T} f_t(x_t^{\mathcal{A}}) - min_{x \in \mathcal{K}} \sum_{t=1}^{T} f_t(x)).$$

Intuitively, an algorithm performs well if the regret is sublinear as a function of T (i.e., $Regret_T(\mathcal{A}) = o(T)$), since this implies that on average, the algorithm performs as well as the best-fixed strategy in hindsight.
The running time of an algorithm for OCO is defined to be the worst-case expected time to produce x_t, for an iteration $t \in [T]^2$ in a T-dimensional repeated game. Typically, the running time will depend on n (the dimensionality of the decision set $\mathcal{K}$), T (the total number of game iterations), and the parameters of the cost functions and underlying convex set." [5]

The Predictor. To instantiate the framework to the shortest path problem, we need to specify the features space X to consider, the prediction model H (and thereby the observation space Y), as well as the loss function ℓ. Each edge corresponds to one request σ_i and its feature $x(\sigma_i) \in X$ encapsulates any information available to the machine learning algorithm.
The main design choice of the prediction model is the question of what to predict which is captured in our framework by the choice of the observation space. For the shortest traffic time problem, a natural candidate is predicting the reliability of an expert. Thus, we evaluate each expert on their estimation of the time of arrival. If such estimations are perfect, the online algorithm can achieve the best competitive ratio. We could also use as a feature the estimation of the distance. However, during our practical experiments (Section 4), adding the estimation of the distance to the feature space X did not improve the performance of the predictor. Formally, the observation space Y we consider is the set of real times that it took to cross an edge. In our experiments, we tried to use both ℓ_1 and ℓ_2 and we found out that the best performance was achieved by using ℓ_2. The prediction model H employed was the *Adaptive Gradient Descent* (AdaGrad) [5] algorithm with weights initially fixed to 1 and breaking ties randomly. It is an *Online Gradient Descent* (OGD) algorithm based on the OCO model where we learn the optimal regularization online. "That is, a *regularizer* that adapts to the sequence of loss functions and is in a sense the "optimal" regularization to use in hindsight" [5]. For more information on AdaGrad, OGD, Gradient Descent (GD) and Regularization we refer to chapters 2, 3 and 5 from the book of Hazan [5], one of the inventors of the AdaGrad algorithm.
To bound the weights updated by AdaGrad into the constraint between 0 and 1, we used the *softmax* function [7]:

$$softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}.$$

3 Main Results

In this section, we present our main contribution: an heuristic that fulfills the three desiderata outlined in Section 1. Before delving into the details of our heuristic, we first examine alternative approaches and demonstrate that blindly adhering to the predictor's recommendations can lead to poor performance in terms of both robustness and competitiveness. Also, we analyze an algorithm for an undirected graph that gives an $O(k)$-approximation under certain assumptions. Finally, we analyze our heuristic.

Blindly Following the Predictor. Simply selecting the expert that the predictor deems most reliable is not always the optimal strategy. Although the experts are not infallible, there exist experts with superior performance compared to others. However, as the predictor must evaluate the efficacy of each expert to determine the most reliable, it may inadvertently follow less trustworthy sources. Consequently, the predictor's success hinges not on a single expert but rather on the overall quality of the expert pool. Additionally, if outliers arise among the most reliable experts, the predictor may excessively penalize these experts, necessitating a prolonged period to recalibrate the weights to favor the superior experts. This delay can significantly impact competitiveness and robustness. Our proposed predictor achieves an approximation between 1 and 2 in the case of a directed graph.

Blindly Following a Randomized Algorithm. We also explored a randomized approach, selecting an expert uniformly at random at each step. However, this approach consistently underperformed compared to the predictor and the online algorithm, only occasionally matching their performance. Consequently, combining it with the predictor would be inconsequential, as it does not yield improved competitive ratios.

Blindly Following The Online Algorithm. The online algorithm is a straightforward approach that involves identifying the nearest edge with the lowest travel time value. In our context, we seek the expert whose estimated arrival time for a specific destination d is the minimum. However, blindly adhering to the online algorithm's recommendations is not always optimal due to the inherent inaccuracies of expert estimations. It attains a 1-approximation in the case of a directed graph.

3.1 The Algorithm

Before presenting the heuristic for a directed graph, we present an algorithm for the undirected game that has an $O(k)$-approximation. The game has the following assumptions:

- There are no one-way streets;
- Traffic is independent of time;
- The length of an edge is constant.

Algorithm 1 Algorithm

```
Require: $k \geq 3$, $s$, $d$
  $dist = \varepsilon$
  For every expert, initialize a weight equal to 1
  Pick the expert $i$ with maximum weight, breaking ties randomly
  Query $i$ to obtain a path from $s$ to $d$
  Follow the suggested path until you reach a node that has distance $\geq dist$
  Compute the loss function between the estimated value and the truth value
  The machine learning algorithm updates the weight of $i$
  if we did not reach the destination then
      $dist = dist \cdot 2$
      Go back to the source and try another expert
  end if
```

Theorem 3. *Given an undirected graph where there are no one-way streets and assuming that traffic is independent of time and that the length of an edge is constant, we can get an $O(k)$-approximation.*

Proof. First, we pick an expert at random and go to the next node that has the smallest distance from s. If we did not reach d, we go back and try another expert but, this time, we double the distance required. We do this until we reach d. The lower bound of such an approach is the case where we follow the nodes from the shortest path but, by our approach, we go back to the source every time until we reach d. Thus, we take at least double the minimum time required. Then, what is the upper bound? We follow the recommendations from a pool of k experts, where $k-1$ experts are bad. It means that their suggestions are bad. Thus, our algorithm, before understanding which is the best expert to follow, obeys the suggestions of every other expert until understanding which is the best expert for that destination d. Thus, we follow at most $k-1$ different paths before following the path suggested by the best expert, which is the optimal path that requires time $2Tc$, where $c \geq 1$ is a constant that represents the uncertainty with respect to the minimum time required. If such an expert is perfect, $c = 1$; otherwise it suggest a path which is the closest to T.

$\square$

3.2 The Heuristic

Our main technical contribution lies in the development of a prediction-based adaptation of an online algorithm that follows the expert suggesting the minimum travel time, treating both approaches as black boxes. This heuristic produces a 1-approximation, outperforming either individual algorithm. By combining the online algorithm algorithm with our predictor, we aim to incorporate the feedback from both approaches. At each node along the path, we determine whether to follow the recommendations of the online algorithm or the predictor. Initially, we start by following the predictor's guidance and evaluate its performance based on the error it generates. The error is calculated as the absolute difference between the estimated travel time and the actual travel time. We accumulate the sum of these errors and compute their average over time. A threshold T is set, defined as the mean of the collected errors plus a variable c. The variable c is crucial as relying solely on the mean may lead to inaccurate feedback. Generally, we prefer to follow the predictor's recommendations, as it reflects our perception of the expert's reliability; this is controlled by c. If the error is lower than T, then we follow the most reliable expert and we increase c, meaning that the advice of the predictor was good. If the error is greater than T and we followed the predictor, we will follow the expert picked by the online algorithm and we lower c, meaning that the advice of the predictor was bad. We decrease c because, if the quality of the overall experts is poor but the performance of the expert picked by the online algorithm is good, we will follow more and more the expert picked by online algorithm. If the error is greater than T and we followed the expert picked by online algorithm, it means that the advice of the expert picked by the online algorithm was bad. Hence, we follow the predictor and we increase c. In this way, for each destination, the algorithm tries to understand which is the best method between the two and, even with a few destinations, our algorithm beats the performance of the online algorithm.

Algorithm 2 Heuristic

```
Require: $k \geq 3$, $s$, $d$, $m$, $n$, $c$, $decision$
  Find the expert $i$ that suggests the minimum estimated time of arrival
  while $v \neq d$ do
      if $decision == 1$ then
          The predictor selects the expert
      else
          Select $i$
      end if
      Query the selected expert on ($s$, $d$) and obtain:
      - the next node $v$
      - its estimated time of arrival $x$
      Go to node $v$ and find out the real-time $y$ that it took to go to $v$
      Compute the error $e = |x - y|$
      The predictor updates the weight of the selected expert
      Update the sum of the collection of the errors $m = m + e$
      Update the number of the collected errors $n = n + 1$
      Compute the threshold $T = \frac{m}{n} + c$
      if $e \leq T$ then
          $decision = 1$
          $c = c + 3$
      end if
      if $decision == 1$ and $e > T$ then
          $decision = 2$
          $c = c - 1$
      end if
      if $decision == 2$ and $e > T$ then
          $decision = 1$
          $c = c + 3$
      end if
      $s = v$
  end while
```

3.3 Heuristic Analysis

Let us bind the performance of our heuristic. It cannot take less time than the minimum travel time required to go from a source s to a destination d since, otherwise, this implies that there exists another shorter path.
Consider the case where the quality of the estimations from the expert that suggest the minimum estimated time of arrival is poor. Our heuristic starts

following the decisions of the predictor and we assume is not perfect because, if it is, it will mean that the overall quality of the experts is very good. By exploiting the estimations from every expert, we will obtain an equal or better competitive ratio than following only a single expert as in the online algorithm because there is at least one edge where there is a better estimation from one of the other experts than from the expert picked by the online algorithm. Assuming is not perfect, there will be at least one episode where the distance between the expected value and the ground truth will be greater than T. Hence, we will try to follow more the suggestions from the expert picked by the online algorithm. If such suggestions are wrong, our heuristic penalizes the weight of such an expert and increases T. As the number of destinations considered increases, we will only follow the suggestions of the predictor because T will tend to infinity.
Consider the case where the quality of the estimations from the expert that suggest the minimum estimated time of arrival is good and assuming that the predictions are bad. This means that the overall quality of the experts is poor.
Our heuristic starts following the decisions of the predictor and tries to follow every expert before understanding which is the best. Since the quality of the overall experts is poor, at a certain point, we will incur a greater error than T. Such an error may be produced by the bad quality of the expert chosen or because of an outlier error produced by one of the best experts. When it happens, it will follow the expert picked by the online algorithm and, since its predictions are good, the weight of this expert is increased. In the long-term the weight of such an expert will always be the maximum weight and, in case of an outlier error, it will follow again this expert, re-adjusting its weight. As the number of destinations considered increases, we will only follow the suggestions of the expert chosen by the online algorithm.
Consider the case where both approaches are bad. As described above, we will still follow the slightly better approach.
Consider the case where both approaches are good, assuming neither is perfect. We will mostly follow the predictor but, whenever we follow an expert which gives poor estimations about a destination d, it will produce an error which is greater than T. Thus, we will follow the suggestion of the expert picked by the online algorithm, re-calibrating the weights. Assuming such estimation was good, we will follow again this expert otherwise we will explore and discover, through the good predictions of the predictor, the best expert for d.

4 Experiments

In this section, we validate our approach to real-world data sets, empirically demonstrating its sensitivity to the predictor errors, and comparing it to established baselines.

4.1 Collecting the Data

To gather the data we drew inspiration from *routingpy* [8], a Python library designed for retrieving routing information from various APIs. However, our code diverges significantly from routingpy and utilizes different APIs:

- ArcGIS;
- BingMaps;
- Mapbox;
- MapQuest;
- TomTom.

Each API comes with its free plan, imposing usage restrictions ranging from daily to monthly or annual limits. To our knowledge, we identified only two other APIs (Google Maps and Apple Maps) that offer unique routing data. Moreover, some APIs rely on the data provided by others, such as the case of HERE and MapQuest. MapQuest utilizes HERE's services to generate routing information. We chose MapQuest over HERE solely due to its free plan and more liberal API usage guidelines. Google Maps and Apple Maps were not considered due to their lack of a free plan or the requirement to provide a credit card for free usage.
Every API can be accessed through either a GET or POST HTTP request, and individual APIs have varying configurations that can be modified. To address these complexities, we developed dedicated classes for each API, each handling the necessary URL parameters and ensuring adherence to the prerequisites outlined earlier. The exception is Mapbox, which utilizes a POST request. The parsed JSON response is returned as a Python object called *Direction*, encompassing the estimated time of arrival, estimated distance, and suggested path. The path starts from the source node and concludes at the destination node. The tolerance for the distance between consecutive nodes differs across APIs, leading to variations in the suggested nodes and path lengths. Unfortunately, not all APIs provide the option to modify this tolerance.

A single client class handles the HTTP requests, parsing the URL into the correct format and utilizing the parameters and URLs configured within the respective API class. Instead of terminating the program, if a request fails, the program prints the stack traceback on the console and returns a None object.
Each coordinate comprises two dimensions (latitude and longitude) and every API requires as inputs the coordinates in this format. To streamline data handling, we employed the FlexPolyline library [9] for encoding the coordinate pair into a single-string representation known as a polyline. Every time we want to perform a query we decode the single polyline.
To extract the needed values from the JSON response, we need to explain briefly the structure of a JSON file. It is represented as a Python dictionary and contains curly brackets ({ }) and brackets ([ ]). If we want to access an element e inside a curly bracket, we just write [“e”] while, instead, we write [0][“e”]. For example, in BingMaps:

$$distance = response[\text{“}resourceSets\text{”}][0][\text{“}resources\text{”}][0][\text{“}travelDistance\text{”}]$$

In the next pages, we show the algorithm of the client and the algorithms of each API used, to highlight the different settings each API requires. We don't show the exact lines of code on how to extract the values from the JSON response due to lack of space. Also, we note that these services may change their keywords and, if the reader wants to verify these experiments, we refer him or her to the documentation of each service.

Algorithm 3 Client

```
Require: $url$, $header$, $params$
  Encode $params$ into the URL format
  Merge the $url$ with the encoded $params$ using the symbol ? as separator
  Perform a GET or a POST HTTP request and obtain the $response$
  return $response$
```

Algorithm 4 ArcGIS

```
Require: $key$, $source$, $destination$, $departure$
  $url$ = “https://route.arcgis.com/arcgis/rest/services”
  $url$ = Merge $url$ with “/World/Route/NAServer/Route_World/solve”
  $header$ = {“Content-Type”: “application/json”}
  $params$ = {}
  $params$[“token”] = $key$
  $c$ = Merge $source$ and $destination$ using the symbol ; as separator
  $params$[“stops”] = $c$
  $params$[“f”] = “json”
  $params$[“impedanceAttributeName”] = “TravelTime”
  $params$[“accumulateAttributeNames”] = “Kilometers”
  $params$[“returnDirections”] = “false”
  $params$[“returnRoutes”] = “true”
  $params$[“outputLines”] = “esriNAOutputLineTrueShape”
  $params$[“outputGeometryPrecision”] = “0.1”
  $params$[“outputGeometryPrecisionUnits”] = “esriMeters”
  if $departure$ then
      $params$[“startTime”] = $departure$
  end if
  Perform a GET HTTP request to the $Client(url, header, params)$
  $distance$ = Pick the value of “Total_Kilometers” from $response$
  Convert $distance$ in meters
  $duration$ = Pick the value of “Total_TravelTime” from $response$
  Convert $duration$ in seconds
  $path$ = Pick the nodes of “paths” from $response$
  return $Direction(distance, duration, path)$
```

Algorithm 5 BingMaps

```
Require: $key$, $source$, $destination$, $departure$
  $url$ = “http://dev.virtualearth.net/REST/V1/Routes/Driving”
  $header$ = {“Content-Type”: “application/json”}
  $params$ = {}
  $params$[“key”] = $key$
  $params$[“wp.0”] = $source$
  $params$[“wp.1”] = $destination$
  $params$[“optimize”] = “timeWithTraffic”
  $params$[“routeAttributes”] = “routePath”
  if $departure$ then
      $params$[“timeType”] = “Departure”
      $params$[“dateTime”] = $departure$
  end if
  Perform a GET HTTP request to the $Client(url, header, params)$
  $distance$ = Pick the value of “travelDistance” from $response$
  Convert $distance$ in meters
  $duration$ = Pick the value of “travelDurationTraffic” from $response$
  $path$ = Pick the nodes of “coordinates” from $response$
  return $Direction(distance, duration, path)$
```

Algorithm 6 Mapbox

```
Require: $key$, $source$, $destination$, $departure$
  $url$ = “https://api.mapbox.com/directions/v5/mapbox/driving-traffic”
  $header$ = {“Content-Type”: “application/x-www-form-urlencoded”}
  $params$ = {}
  $params$[“access_token”] = $key$
  $c$ = Merge $source$ and $destination$ using the symbol ; as separator
  $params$[“coordinates”] = $c$
  $params$[“geometries”] = “geojson”
  $params$[“overview”] = “full”
  if $departure$ then
      $params$[“depart_at”] = $departure$
  end if
  Perform a POST HTTP request to the $Client(url, header, params)$
  $distance$ = Pick the value of “distance” from $response$
  $duration$ = Pick the value of “duration” from $response$
  $path$ = Pick the nodes of “coordinates” from $response$
  return $Direction(distance, duration, path)$
```

Algorithm 7 MapQuest

```
Require: key, source, destination, departure
  url = “https://www.mapquestapi.com/directions/v2/route”
  header = {“Content-Type”: “application/json”}
  params = {}
  params[“key”] = key
  params[“from”] = source
  params[“from”] = destination
  params[“unit”] = “k”
  params[“routeType”] = ”fastest”
  params[“shapeFormat”] = “raw”
  params[“useTraffic”] = “true”
  params[“ambiguities”] = “ignore”
  params[“doReverseGeocode”] = “false”
  params[“narrativeType”] = “none”
  params[“timeType”] = “1”
  if departure then
      params[“timeType”] = “2”
      params[“isoLocal”] = departure
  end if
  Perform a GET HTTP request to the Client(url, header, params)
  distance = Pick the value of “distance” from response
  Convert distance in meters
  duration = Pick the value of “realTime” from response
  path = Pick the nodes of “shapePoints” from response
  return Direction(distance, duration, path)
```

Algorithm 8 TomTom

```
Require: key, source, destination, departure
  url = “https://api.tomtom.com/routing/1/calculateRoute/”
  header = {“Content-Type”: “application/json”}
  params = {}
  params[“key”] = key
  Merge url with source and destination using the symbol : as separator
  Append “/json” to url
  params[“travelMode”] = “car”
  params[“routeType”] = “fastest”
  params[“traffic”] = “true”
  params[“routeRepresentation”] = “polyline”
  if departure then
      params[“departAt”] = departure
  end if
  Perform a GET HTTP request to the Client(url, header, params)
  distance = Pick the value of “lengthInMeters” from response
  $t_1$ = Pick the value of “travelTimeInSeconds” from response
  $t_2$ = Pick the value of “trafficDelayInSeconds” from response
  duration = $t_1 + t_2$
  path = Pick the nodes of “points” from response
  return Direction(distance, duration, path)
```

Our experiments focused on two data sets taken from two cities to demonstrate the wide applicability of our approach.

- Rome is the data set containing 39 points within and around Rome;
- Paris is the data set containing 34 points within and around Paris.

In the following tables, we show the data sets. We note that, due to lack of space, we removed the locations Parioli, San Lorenzo, Selva Nera, Bufalotta and Quarticciolo from the table.

Location	**Latitude**	**Longitude**
Piazza Bologna	41.91394911573667	12.518699879396078
Villa Ada	41.923769235641124	12.494524115637745
Pigneto	41.89081617201369	12.534451552221077
Tufello	41.948880022671304	12.532275448140739
EUR	41.83027635665154	12.47294778524916
Ostiense	41.85574300608863	12.48142495456904
Monteverde	41.88019452993852	12.462169171556846
Tor Lupara	41.99647740608383	12.618687989376179
Ciampino	41.80478253692648	12.600693362046337
Ostia	41.73371644895152	12.289888685982373
Ariccia	41.720285156352105	12.670189673472319
Latina	41.46232124632789	12.895826577821909
Formello	42.08121639190518	12.405565490368899
Monterotondo	42.05201066151658	12.622590289190619
Tivoli	41.95831645850654	12.802547056606027
Gaeta	41.21466168005052	13.564101516804692
Monte Mario	41.94357961440761	12.42264495980253
Tuscolano	41.87249756156089	12.528667030151022
Trastevere	41.88522318022662	12.468883347187345
Gianicolense	41.877412945279154	12.450976625612535
Prati	41.90935423403128	12.460469319955317
Tor Bella Monaca	41.86667712718105	12.634636853520762
Frascati	41.807712557653986	12.675267978753801
Castel Gandolfo	41.74853318293775	12.648413064151798
Cinecittà	41.852295178123256	12.565798149369764
San Basilio	41.94163270665697	12.581806015984965
Montespaccato	41.9057947040496	12.391704555365187
Fiumicino	41.79577463217675	12.275551036996516
Fregene	41.85470771154881	12.198934136371578
Monti	41.894562657145336	12.492746494574103
Roma 70	41.83354241061384	12.506077942400793
La Giustiniana	41.983470354596626	12.410649588566303
Corso Trieste	41.92346148325291	12.511060622302532
Lungotevere	41.903512007921016	12.476551520686014
Casale Nei	41.970243262678935	12.53540737633203

Table 1: Rome data set.

Location	**Latitude**	**Longitude**
9th Arr	48.879298	2.337563
16th Arr	48.857653	2.274405
13th Arr	48.827352	2.358919
5th Arr	48.840708	2.351110
8th Arr	48.869624	2.310589
12th Arr	48.841549	2.430783
2nd Arr	48.8676846341558	2.342440799264235
6th Arr	48.85003755420804	2.328922142069705
15th Arr	48.84230696329063	2.2900730906335434
14th Arr	48.83086525801601	2.3227677479234576
11th Arr	48.859343489080544	2.3828740733240408
20th Arr	48.86509366761952	2.403839606953001
18th Arr	48.89234605710981	2.345468302116872
7th Arr	48.85501322133151	2.309346931176439
3rd Arr	48.86099203939442	2.3603527663106365
10th Arr	48.87830453747832	2.350969988578582
1st Arr	48.864621295963275	2.3317489977768906
4th Arr	48.854550560132154	2.3507762
17th Arr	48.88531117693295	2.2997553603787413
19th Arr	48.893516091742896	2.3832400959638464
Versailles	48.80300670481262	2.1452996142462464
Levallois-Perret	48.89662097048963	2.2923203822686666
Saint Denis	48.934469538408464	2.3553799773078707
6th Arr	48.84316424900635	2.33008644099978
16th Arr	48.861684307463875	2.279819467935261
Charles de Gaulle Airport	49.007837155759624	2.5507247929158066
Arc de Triomphe	48.87532007071821	2.2974249924524064
Tour Eiffel	48.8617495430599	2.29816966634968
Place d'Italie	48.83272985059296	2.353267852283001
Boulogne-Billancourt	48.838191302692486	2.2377453295792162
Creteil	48.79262471710494	2.4529334802677725

Table 2: Paris data set.

This data sets offers a comprehensive representation of the cities, encompassing both urban and suburban areas. Our testing focused on intra-city journeys, particularly those susceptible to traffic variations, as well as inter-city trips. For example, we designed the Rome data set to use a lot of paths that cross the GRA or Grande Raccordo Anulare (Great Ring Junction),

“which is a toll-free, ring-shaped 68.2 kilometers long orbital motorway that encircles Rome. GRA is one of the most important roads in Rome, and traffic reaches 160,000 vehicles per day as of 2011” [10]. All tests were conducted in driving mode.
For Rome, we used three sources: Piazza Bologna, Parioli, Formello. Piazza Bologna is a central location within Rome while Parioli is less centered and Formello is in its suburbs. For Paris, we used only its 9th arrondissement as the source. It is a location similar to what is the Parioli for Rome.
Tests were done with different departure times, some of which have usually have high levels of traffic and some that normally have low levels of traffic.
MapQuest was not used since we reached too soon its monthly API rate limit. Thus, 4 experts were used.

4.2 Results

To evaluate our proposed approach, we considered multiple algorithms for evaluation.

- B: This baseline represents the results of following the best expert in hindsight;
- SS: This baseline represents the results of following the expert that suggest the minimum travel time;
- ML: This baseline represents the results of following the machine learned predictions (i.e., the AdaGrad algorithm [5]) to understand which is the best expert;
- H: This algorithm is our proposed heuristic.

Our main metric for evaluation will be the *competitive ratio* of the algorithm, defined as the time taken by the selected approach divided by the minimum time that it takes. Nonetheless, we note that these findings may be influenced by the methodology employed to determine the actual travel times (see Technical Assumptions in Section 2) and by the experts used for these tests. Even though is rare, we may have a competitive ratio that is slightly lower than the ground truth. This suggests that, strangely, the expert who is working as the ground truth did not suggest the shortest path, even if every API is fixed to give us the fastest path.
As we can see in the figures below, our heuristic achieve the best of both worlds. Occasionally, it achieves a competitive ratio slightly above the best competitive ratio found by running the single methods. This may be biased by the noise generated by the number of tries used for getting an expected

competitive ratio, since, for each try, the order of the destinations was randomly shuffled. Most probably, if we do more tries, we will achieve the same lower bound.
We were limited by the number of queries that could be used. Hence, we used a variable *step* that every time decides which node to select from the suggested path. In Fig. 2 and Fig. 3 we fixed *step* = 9, meaning that we pick the 10th node from the suggested path. In Fig. 4, *step* = 29 while in Fig. 5 and Fig. 6 we fixed it to be proportionally to the length of the path, to always pick 20 nodes for each destination. We described what a step is in Section 2.
Our heuristic, beyond understanding which is the best method for that input, in most cases, beat the best competitive ratio of the online algorithm or of the predictor with a gap of 0.1. The number of tries used was 3. We could not do more tries or use more destinations due to the limitations on the number of queries imposed by every API. We note that in some tries H was the only one to beat also B in the directed setting and in Fig. 6 our solution on average beat B. This suggests that, with more data, our algorithm on average will probably beat also following the single best expert in hindsight. Our conclusion is that, with low level of traffic and enough data, the predictor will beat the online algorithm but, if we have high level of traffic and the expert that outputs the minimum travel time is reliable, the online algorithm will beat the predictor.

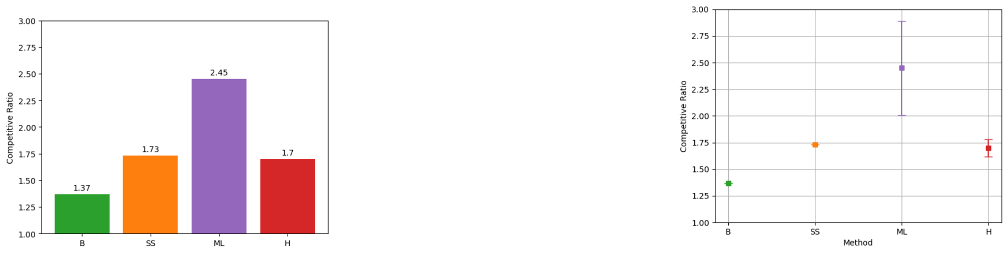


Figure 2: Average estimated competitive ratio (left) and the relative standard deviation (right) for the directed setting with 5 destinations from Rome and departure time fixed at 08/01/2024 (Monday), 8:00 with Piazza Bologna as the source.

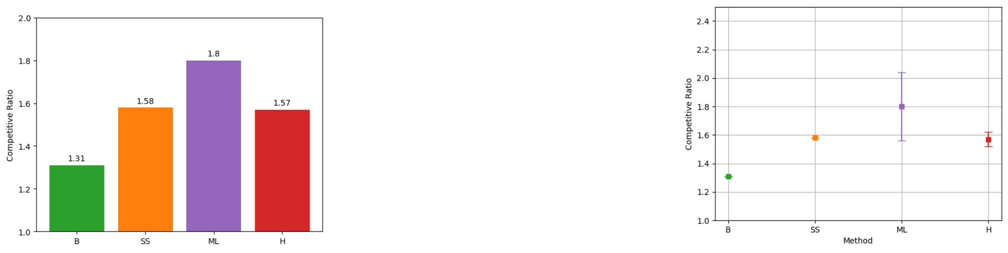


Figure 3: Average estimated competitive ratio (left) and the relative standard deviation (right) for the directed setting with 5 destinations from Rome and departure time fixed at 08/01/2024 (Monday), 15:00 with Piazza Bologna as the source.

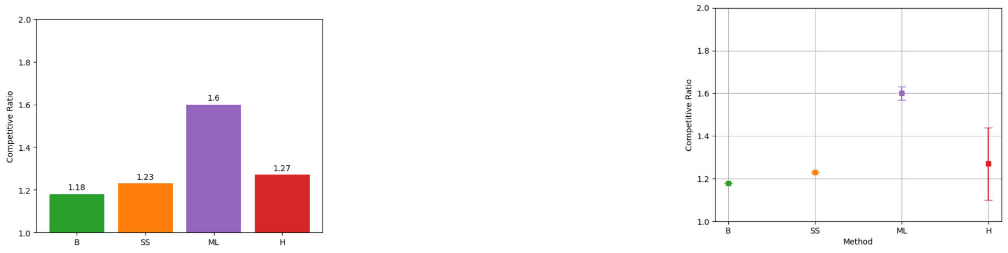


Figure 4: Average estimated competitive ratio (left) and the relative standard deviation (right) for the directed setting with 17 destinations from Rome and departure time fixed at 08/01/2024 (Monday), 15:00 with Formello as the source.

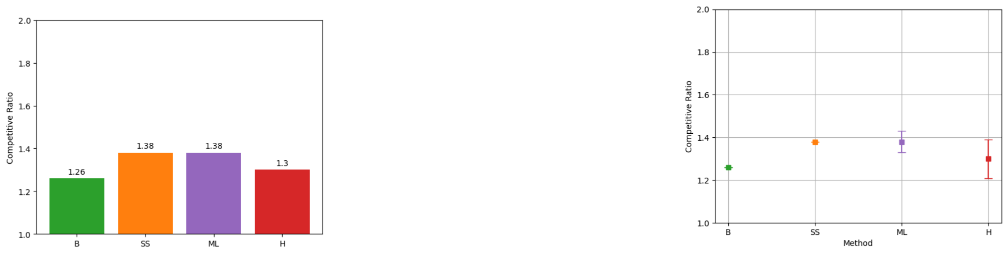
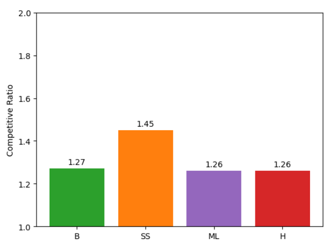


Figure 5: Average estimated competitive ratio (left) and the relative standard deviation (right) for the directed setting with 34 destinations from Paris and departure time fixed at 09/01/2024 (Tuesday), 10:30 with 9th arrondissement as the source.



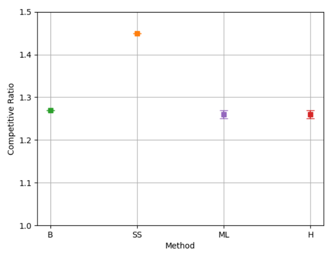


Figure 6: Average estimated competitive ratio (left) and the relative standard deviation (right) for the directed setting with 39 destinations from Rome and departure time fixed at 08/01/2024 (Monday), 18:00 with Parioli as the source.

5 Conclusion

In this study, we continue the research on online algorithms using machine learning predictors. This combines the practical successes of machine learning with the theoretical guarantees of online algorithms. We set up a model for the shortest traffic time problem and developed a machine learning algorithm whose performance is directly related to the accuracy of its predictions. Our work has laid the foundation for new avenues of future research, particularly in the pursuit of more capable predictors. From a practical standpoint, our algorithm shows how we can use machine learning safely, avoiding problems caused by occasional big errors in the predictions. At the same time, our experimental results show that even with simple predictors, our algorithm is better than following simply the expert that suggests the minimum travel time. In essence, we have reduced the worst-case performance of the shortest traffic time problem to the problem of finding a good (on average) predictor.

References

[1] Lykouris T. Vassilvitskii S. "Competitive Caching with Machine Learned Advice". In: *Journal of the ACM* 68.4 (2021). DOI: 10.1145/3447579.

[2] Mahdian M. Nazerzadeh H. Saberi A. "Online Optimization with Uncertain Information". In: *ACM Transactions on Algorithms* 8.1 (2012). DOI: 10.1145/2071379.2071381.

[3] Lattanzi S. Lavastida T. Moseley B. Vassilvitskii S. "Online Scheduling via Learned Weights". In: *SIAM* (2020). DOI: 10.1137/1.9781611975994.114.

[4] Almanza M. Chierichetti F. Lattanzi S. Panconesi A. Re G. "Online Facility Location with Multiple Advice". In: *NeurIPS* (2021). URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/250473494b245120a7eaf8b2e6b1f17c-Paper.pdf.

[5] Hazan E. *Introduction to Online Convex Optimization.* 2022.

[6] *About routes, legs, and steps.* URL: https://developers.google.com/maps/documentation/routes/understand-route-response#about_routes_legs_and_steps.

[7] *Softmax function.* URL: https://en.wikipedia.org/wiki/Softmax_function.

[8] *routingpy.* URL: https://github.com/gis-ops/routingpy.

[9] *FlexPolyline.* URL: https://github.com/heremaps/flexible-polyline.

[10] *Grande Raccordo Anulare.* URL: https://en.wikipedia.org/wiki/Grande_Raccordo_Anulare.