CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Using machine learning to improve a column generation framework for a tail assignment optimizer

Master's thesis in Mathematical Sciences

Lukas Enarsson and Karthikeyan Jeganathan

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Using machine learning to improve a column generation framework for a tail assignment optimizer

Lukas Enarsson and Karthikeyan Jeganathan

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Using machine learning to improve a column generation framework for a tail assignment optimizer

Lukas Enarsson and Karthikeyan Jeganathan

Supervisor: Emil Gustavsson, Fraunhofer-Chalmers Centre
Advisor: Mattias Grönkvist, Jeppesen Systems AB
Examiner: Ann-Brith Strömberg, Mathematical Sciences

Master's Thesis 2022
Department of Mathematical Sciences
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Using machine learning to improve a column generation framework for a tail assignment optimizer

Lukas Enarsson and Karthikeyan Jeganathan
Department of Mathematical Sciences
Chalmers University of Technology and University of Gothenburg

Abstract

While the research effort spent on algorithms for solving optimization problems appearing in an airline industry is vast, the use of machine learning for improving these algorithms is a new and under-explored topic. A popular and widely researched problem in the airline industry is the tail assignment problem. This problem aims to assign a set of aircrafts to a set of flights such that constraints like pre-planned maintenance activities, destination restrictions, curfews and turn time rules are respected and some objective function like fuel costs or robustness is optimized to get a schedule for an airline. At Jeppesen Systems, the tail assignment problem is solved using a column generation framework which uses a subgradient optimizer to solve the Lagrangean dual problem. In this thesis, machine learning is being used to determine whether to start each iteration of the column generation framework from the previous iteration solution (warm start) or from scratch/origin (cold start). The choice of starting procedure has a significant effect on the solution time and overall quality of the solution. The problem is investigated by proposing predefined strategies with different types of starts which can be used to solve a tail assignment problem. After studying the performance of the strategies, it turned out that some of them perform better than the current heuristic employed at Jeppesen Systems for certain problem instances, implying that the machine learning model could improve the column generation framework. Five supervised learning models were then implemented to learn the patterns for selecting the right strategy for a given tail assignment problem instance. The implemented models were then evaluated against a baseline model using accuracy and F_1-score as evaluation metrics. The results from the analysis of the models suggest that all the selected features of a problem instance allow four of the five models to perform better than just randomly guessing the strategy. In the future, the models can be improved by developing models on a larger data set with more problem instances, having more diverse, dynamic strategies and by analyzing the effect of the choice of input features for each tail assignment problem.

Keywords: tail assignment, column generation, subgradient optimization, airline optimization, aircraft routing, shortest path problem, machine learning, supervised learning, classification.

Acknowledgements

We would like to thank **Jeppesen Systems** for giving us this wonderful opportunity to do this thesis work. First of all, we would like to thank our supervisor, **Mattias Grönkvist**, for mentoring and guiding us in all technical aspects as well as in report writing. His expertise in the field of optimization has been a great support throughout this thesis. We would also like to thank our Chalmers supervisor **Emil Gustavsson**, who has guided us in each step of this thesis. We also thank our examiner **Ann-Brith Strömberg**, who has been a great help during this master thesis.

At Jeppesen, we would like to specially thank **Subhra Ghosh, Andreas Westerlund, Jens Kjerrström, Chi Thong Luong** and **Johan Kellerth Fredlund** for their constant support, guidance and suggestions in this thesis work. It was a nice learning experience from each and every one of you.

Finally, we would like to thank our **friends** and **families** for all their constant support and motivation.

Contents

List of Figures

List of Tables

1 Introduction

This chapter presents an overview of the problem that this thesis aims to solve.

1.1 Background

An important optimization problem in the airline industry is the tail assignment problem (TAP) [1]. This thesis will mainly focus on the tail assignment problem and has been done in collaboration with Jeppesen Systems. Jeppesen Systems offers flight planning products, operations planning tools and provide software solutions in airline industry. When planning the routes of aircraft for a given set of flights, each flight is first assigned to a particular aircraft type by solving what is known as the fleet assignment problem. After assigning the flights to aircraft types, each individual aircraft is assigned to a subset of the flights while minimizing some objective and considering certain constraints. This is the tail assignment problem which Jeppesen Systems usually solves separately for each unique aircraft type. The constraints that are usually considered include those of respecting flight maintenance and flight time, but other constraints can also be included. With the help of tail assignment, one can improve aircraft utilization by factoring in crew connections, maintenance, fuel usage, operational costs, and constraints for each aircraft in the fleet. By optimizing the objective, one can fly more before maintenance checks and align maintenance cycles to reduce the fleet maintenance costs.

At Jeppesen Systems, the tail assignment problem is formulated using a Dantzig–Wolfe reformulation [2] where the flight routes and the decision of leaving a flight unassigned are used as decision variables which helps in modelling constraints like regular maintenance, noise limitations, and the time needed between two flights. The major drawback of having flight routes as decision variables is that the number of variables in the problem grows exponentially with the number of flights. The tail assignment problem is solved using a column generation framework [3] which generates aircraft schedules. This helps in efficiently planning the available resources in an airline company. Also in case of disruptions, it can be rescheduled with less additional cost.

The column generation framework is solved by starting with an initial restricted master problem (RMP) formulated by leaving every flight unassigned [1]. After finding an optimal solution to the dual problem of the RMP, the dual values are used to generate potentially improving routes which are added to the restricted master problem. The updated RMP is then resolved and the process repeats until no improving routes can be found. The dual solution of the RMP itself is found by using subgradient optimization, since the RMP will be hard to solve optimally.

One step in the subgradient algorithm is to decide whether we let the initial dual solution be the dual solution from the previous restricted master problem (warm start) or if it should be the origin (cold start). Currently, a heuristic is employed at Jeppesen Systems to identify whether to use warm start or cold start when solving the current restricted master problem using this subgradient algorithm.

However, it has been found that this heuristic does not always seem to suggest a good choice. Therefore, an interesting question is if it is possible to improve the decision of using warm start or cold start by using machine learning.

1.2 Aim

The aim of this thesis is to investigate the possibility of using machine learning to decide whether warm start or cold start should be used while solving the restricted master problem in each iteration of the column generation framework for the tail assignment problem in order to improve the solution quality and the running time.

1.3 Limitations

During this thesis, the ethical aspects will not be taken into consideration as all the work is done with already collected test data of flights by Jeppesen Systems. The investigation of the tail assignment problem will not affect any real life solutions of the tail assignment problem until it is found that the solution quality and running time of the column generation framework improves, and the machine learning algorithm is implemented in the current framework.

Instead of selecting between warm start and cold start at every iteration, a decision on the strategy of how to select the starts is made before starting the column generation framework. While considering the time available for this thesis, only a few predefined strategies with various combinations of warm start and cold start which could potentially improve the solution quality will be used and decided between. This thesis will only revolve around improving the selection of warm start and cold start and will not be concerned about improving the performance of any other part of the subgradient optimizer or any other component of the overall Tail Assignment algorithm. With many machine learning algorithms available in usage, only five algorithms, namely decision tree, random forest, Naive Bayes, Support Vector Machine, and artificial neural networks will be investigated in this thesis.

1.4 Research questions

- Is it possible that only using warm start or only using cold start is a better strategy than the current heuristic?
- Could a choice between a set of predefined strategies based on the input characteristics of the problem before the first iteration of the column generation algorithm work by predicting a strategy from a trained machine learning model?
- Do the proposed strategies perform better than the current heuristic?
- Which ML model works the best for improving the subgradient optimizer?

1.5 Thesis outline

Chapter 2 is dedicated to some of the optimization theory behind Jeppesen Systems' tail assignment solver and the machine learning theory that is used during the rest of this thesis. Chapter 3 contains the various methods used in this thesis and Chapter 4 contains the results of the methods. Finally, the thesis is concluded in Chapter 5 with discussions and a conclusion.

2
Theory

In the following sections, the main theory that is used for the rest of the thesis will be presented.

2.1 Tail assignment problem

Within the airline industry, an important aspect is to plan what aircraft to use on each flight. According to Grönkvist [1], planning consists of three different steps. The first step of planning is to determine when flights will occur and at what airports. After that, aircraft types are assigned to each flight and the crew is planned according to the types of aircraft that are assigned to the flights. The final step of planning is to assign an individual aircraft to each flight, which is known as aircraft assignment, which looks very different for each airline. Grönkvist describes the *tail assignment problem* as a specific approach of aircraft assignment [1]. In this approach, the aircraft are assigned to the flights such that constraints applying to all aircraft as well as individual aircraft, e.g., maintenance and restrictions during flight are fulfilled while minimizing some objective function. The aircraft are assigned over a fixed period of time. The purpose of this is to be able to handle a large variety of constraints and objectives that occur during planning.

Some of the constraints that may occur as part of the TAP are described both by Grönkvist [1] and by Fuentes, et al. [4] and include maintenance activities for generic and individual aircraft, noise limitations preventing aircraft from being operated at some airports at certain times, or considering the time that it takes to switch crews between arrival to and departure from an airport. An example of an objective that can be optimized is one that encourages aircraft to stay at airports between flights and maintenance for either a long time or a short time so that they are either used well, or can be used between planned activities if an unexpected event occurs [4].

2.1.1 Formulating the tail assignment problem

The most intuitive way of formulating the tail assignment problem would be to formulate it as an integer multi-commodity network flow problem [5] with maintenance constraints on the path with each arc contributing to the objective. However, [1] instead formulates the TAP by using path-flows, in order to combine the constraints of individual aircraft into the decision variables. This means that entire flight routes become the variables. This can be considered as a form of *Dantzig–Wolfe decompo-*

sition [2]. Let $\mathcal{R}$ denote the set of possible individual aircraft routes with respect to the individual constraints and let $x_r = 1$ if route $r \in \mathcal{R}$ is used and $x_r = 0$ otherwise. Let c_r be the cost of using route $r \in \mathcal{R}$. While planning the various flights, only one aircraft should be used for each flight, meaning that only one route can include a given flight. Let $\mathcal{F}$ denote the set of all flights and other activities, and define $a_{f,r}$ to be one if the route $r \in \mathcal{R}$ covers activity $f \in \mathcal{F}$ and zero otherwise. Then the constraint making sure that each activity is included in exactly one of the used routes, can be formulated as $\sum_{r\in\mathcal{R}} a_{f,r}x_r = 1$ for every activity $f \in \mathcal{F}$. It is also possible to leave an activity unassigned if routes that cover a single activity each at a high cost are added to $\mathcal{R}$. The formulation of the problem is then:

$$
\begin{aligned}
z^* = \min \quad & \sum_{r\in\mathcal{R}} c_r x_r, && && \text{(2.1a)}\\
\text{s.t.} \quad & \sum_{r\in\mathcal{R}} a_{f,r} x_r = 1, && f \in \mathcal{F}, && \text{(2.1b)}\\
& x_r \in \{0,1\}, && r \in \mathcal{R}. && \text{(2.1c)}
\end{aligned}
$$

As the decision variables cover all the constraints for individual aircraft implicitly, the number of explicit constraints are linear with respect to the number of activities. However, the number of variables is exponential with respect to the number of activities and connections between activities, and the problem is a set partitioning problem, which is NP-Hard [1].

2.2 Lagrangean relaxation

As the optimization problem formulated in Equation (2.1) is a large-scale optimization problem, techniques which are used in optimization at a large scale are needed. One of these techniques include *Lagrangean relaxation*, where the idea is to relax some complicating constraints and add them as penalized terms in the objective function. In the following subsection some general theory and description regarding Lagrangean relaxation is presented (see [6]).

2.2.1 Lagrangean relaxation for linear programs

Consider a general *linear program* (LP) [7]:

$$
\begin{aligned}
z^* = \min \mathbf{c}^T\mathbf{x}, && \text{(2.2a)}\\
\text{s.t.} A\mathbf{x} \geq \mathbf{b}, && \text{(2.2b)}\\
x \in \mathcal{X}, && \text{(2.2c)}
\end{aligned}
$$

where $\mathbf{x}, \mathbf{c} \in \mathcal{R}^n$, $\mathbf{b} \in \mathcal{R}^m$, $A \in \mathcal{R}^{m\times n}$ and the set $\mathcal{X} \subset \mathcal{R}^n$ is polyhedral, nonempty and bounded. Assume also that the set $\{x \in \mathcal{X} : A\mathbf{x} \geq \mathbf{b}\}$ is nonempty, which since $\mathcal{X}$ is bounded implies that there exists an optimal solution to the problem. The problem is created in such a way that it would be easier to solve if the constraints $A\mathbf{x} \geq \mathbf{b}$ were removed.

In order to make the problem simpler, the constraints (2.2b) are relaxed by giving a cost to violations of the constraint instead of requiring the constraint to be satisfied. For this purpose, define the *Lagrangean function* $\mathcal{L} : \mathcal{R}^n \times \mathcal{R}^m \mapsto \mathcal{R}$ as [8]:

$$\mathcal{L}(\mathbf{x}, \mathbf{u}) = \mathbf{c}^T\mathbf{x} + \mathbf{u}^T(\mathbf{b} - A\mathbf{x}). \tag{2.3}$$

Note that in the case when $\mathbf{u} \in \mathcal{R}^m_+$, the minimization of $\mathcal{L}(\mathbf{x}, \mathbf{u})$ with respect to $\mathbf{x} \in \mathcal{X}$ is a relaxation of the original LP.
The minimization of the Lagrangean function w.r.t. $\mathbf{x}$ allows the definition of the *Lagrangean dual function* [7]:

$$h(\mathbf{u}) = \min_{\mathbf{x}\in\mathcal{X}} \mathcal{L}(\mathbf{x}, \mathbf{u}) = \mathbf{b}^T\mathbf{u} + \min_{\mathbf{x}\in\mathcal{X}} (\mathbf{c} - A^T\mathbf{u})^T\mathbf{x}. \tag{2.4}$$

The inner minimization problem $\min_{\mathbf{x}\in\mathcal{X}}(\mathbf{c} - A^T\mathbf{u})^T\mathbf{x}$ is called the *Lagrangean subproblem* [8]. Denote the optimal set of the Lagrangean subproblem for a given $\mathbf{u} \in \mathcal{R}^m$ as $X(\mathbf{u})$. The Lagrangean dual function provides some useful properties that can be used to find solutions to the original LP. One such property is that for any $\mathbf{u} \in \mathcal{R}^m_+$ and $\mathbf{x} \in \mathcal{X}$ such that $A\mathbf{x} \geq \mathbf{b}$, it holds that:

$$h(\mathbf{u}) \leq \mathbf{c}^T\mathbf{x}. \tag{2.5}$$

This property is known as *weak duality* [7]. Weak duality implies that every value of the Lagrangean dual function, for non-negative values of the multipliers $\mathbf{u}$, provides a lower bound on the objective value. Thus, it is of interest to try to maximize this function to get the best possible lower bound, leading to the definition of the *Lagrangean dual problem* to find:

$$h^* = \max_{\mathbf{u}\in\mathcal{R}^m_+} h(\mathbf{u}). \tag{2.6}$$

As the original optimization problem is linear and as such is convex, *strong duality* holds for the problem, where $h^* = z^*$, though if the problem is non-convex, the duality gap $z^* - h^* \geq 0$ can be non-zero [7]. Another property of the Lagrangean dual function is that it is concave, implying that the Lagrangean dual problem is always a convex problem [7], suggesting that it is relatively easy to solve, provided that evaluating $h(\mathbf{u})$ is easy.

There exists conditions on the optimal solution that makes it easy to check if a given primal-dual pair $\mathbf{x}$ and $\mathbf{u}$ is optimal in the primal and dual problem, respectively. Provided that there is no duality gap, the pair $\mathbf{x}^*$ and $\mathbf{u}^*$ is optimal in the primal and dual problem if and only if $\mathbf{x}^* \in \mathcal{X}$, $A\mathbf{x}^* \geq \mathbf{b}$, $\mathbf{u}^* \in \mathcal{R}^m_+$, and $(\mathbf{u}^*)^T(\mathbf{b} - A\mathbf{x}^*) = 0$ [6]. In other words, $\mathbf{x}^*$ and $\mathbf{u}^*$ have to be primal and dual feasible, respectively, and they have to satisfy what is known as complementary slackness.

2.2.2 Subgradient optimization

One method of solving the Lagrangean dual problem is called *subgradient optimization*, an inner point method that relies on the *subgradient* of the Lagrangean dual function. A subgradient of a concave function h at $\mathbf{u} \in \mathcal{R}^m$ is defined as any vector $\gamma \in \mathcal{R}^m$ where:

$$h(\mathbf{v}) \leq h(\mathbf{u}) + \gamma^T(\mathbf{v} - \mathbf{u}), \tag{2.7}$$

holds for all $\mathbf{v} \in \mathcal{R}^m_+$. The set of all subgradients at the point $\mathbf{u}$ is known as the *subdifferential* and is denoted as $\partial h(\mathbf{u})$ [9]. There is a proposition [9], Prop. 6.20, that states that for convex problems like problem (2.2), the subdifferential of the corresponding Lagrangean dual function, as defined in (2.4), is equal to the convex combination of the points $A\mathbf{x} - \mathbf{b}$ where $\mathbf{x} \in X(\mathbf{u})$. It can be shown that if there is a $\gamma \in \partial h(\mathbf{u})$ where $\gamma \leq \mathbf{0}$, $\mathbf{u}^T\gamma = 0$, and $\mathbf{u} \geq 0$ then $\mathbf{u}$ is the optimal dual point [8].

Subgradient optimization starts at a point $\mathbf{u}_0 \in \mathcal{R}^m_+$ and then searches for a new dual point in the direction of a subgradient, as it always points no more than 90 degrees away from the direction towards an optimal solution [6], thus it will always move closer to an optimal solution. In other words, $\mathbf{u}$ is updated with the formula:

$$\mathbf{u}_{i+1} = [\mathbf{u}_i + \alpha_i(A\mathbf{x} - \mathbf{b})]_{\mathcal{R}^m_+}, \tag{2.8}$$

where α_i are the step lengths and $[\mathbf{v}]_{\mathcal{R}^m_+}$ is the projection of $\mathbf{v} \in \mathcal{R}^m$ onto the set $\mathcal{R}^m_+$. These updates occur until $\mathbf{u}$ is optimal in the dual problem or when the subgradient is small enough. Given that $\lim_{i\to\infty} \alpha_i = 0$, $\sum_{i=1}^{\infty} \alpha_i = \infty$ and $\sum_{i=1}^{\infty} \alpha_i^2 < \infty$, the iterative scheme in (2.8) will converge to the optimal solution of $h(\mathbf{u})$ [10]. The choice of the initial point $\mathbf{u}_0$ affects how fast the subgradient optimization converges, which will be important when solving the tail assignment problem.

2.3 Column generation

This section provides an overview on *column generation* [3], a method used to solve linear programs with a large number of variables.

2.3.1 Dantzig–Wolfe decomposition and the master problem

In Section 2.1.1, it was mentioned that *Dantzig–Wolfe decomposition* was the basis for the formulation of the tail assignment problem. To explain what Dantzig–Wolfe decomposition is, consider the linear program (2.2). The set $\mathcal{X}$ can be redefined as a convex combination of its extreme points $\mathbf{x}_p$ for each p in the index set $\mathcal{P}$ [7] [9]. The variables may be rewritten as $\mathbf{x} = \sum_{p\in\mathcal{P}} \mathbf{x}_p\lambda_p$, where $\lambda_p \geq 0$ for all $p \in \mathcal{P}$ and $\sum_{p\in\mathcal{P}} \lambda_p = 1$. Substitute $\mathbf{c}$ and A for $c'_p = \{\mathbf{c}^T\mathbf{x}_p\}$ and $\mathbf{a}'_p = \{A\mathbf{x}_p\}$ for each $p \in \mathcal{P}$. This gives us the Dantzig–Wolfe decomposition of the original problem, which is known as the *master problem* (MP) [3]:

$$z^* = \min \sum_{p \in \mathcal{P}} c'_p \lambda_p, \tag{2.9a}$$

$$\text{s.t} \sum_{p \in \mathcal{P}} \mathbf{a'}_p \lambda_p \geq \mathbf{b}, \tag{2.9b}$$

$$\sum_{p \in \mathcal{P}} \lambda_p = 1, \tag{2.9c}$$

$$\lambda_p \geq 0, \quad p \in \mathcal{P}. \tag{2.9d}$$

2.3.2 The restricted master problem and pricing problem

As the size of the set $\mathcal{P}$ is likely to be exponential compared to the number of variables in the original problem, solving (2.9) is going to be very hard. This is mitigated by first solving the problem while only considering the variables in a subset of extreme points $\mathcal{P}' \subset \mathcal{P}$, add variables to $\mathcal{P}'$ that can improve the solution, solve it again with the new variables and repeat until the optimal solution is found. Through this, the *restricted master problem* (RMP) is defined as:

$$z^* \leq z^*_{RMP} = \min \sum_{p \in \mathcal{P}'} c'_p \lambda_p, \tag{2.10a}$$

$$\text{s.t} \sum_{p \in \mathcal{P}'} \mathbf{a'}_p \lambda_p \geq \mathbf{b}, \tag{2.10b}$$

$$\sum_{p \in \mathcal{P}'} \lambda_p = 1, \tag{2.10c}$$

$$\lambda_p \geq 0, \quad p \in \mathcal{P}'. \tag{2.10d}$$

Once the RMP has been solved, let z^*_{RMP} be the optimal value of the RMP for the primal solution $\mathbf{x}^*_{RMP} = \sum_{p \in \mathcal{P}'} \mathbf{x}_p \lambda_p$ and let π and ψ be the optimal dual values corresponding to the constraints (2.10b) and (2.10c), respectively. After this, the duals are used to find the point in $\mathcal{X}$ with the minimum reduced cost. This is also known as the *pricing problem* [11]:

$$\bar{c}^* = \min_{x \in \mathcal{X}} \{(\mathbf{c}^T - \pi^T A)\mathbf{x} - \psi\}. \tag{2.11}$$

Let $\mathbf{x}_{p'}$ be the solution to the pricing problem. Note that $\mathbf{x}_{p'}$ will be an extreme point to $\mathcal{X}$, since the pricing problem is still linear like the original master problem. If $\bar{c}^* < 0$, then adding the extreme point $p' = \mathbf{x}_{p'}$ to $\mathcal{P}'$ will improve the objective value of the master problem, so the cost $c'_{p'}$ and the column $\mathbf{a'}_{p'}$ is calculated and the variable $\lambda_{p'}$ is added to the RMP. The RMP is then solved again to find more improving extreme points. If $\bar{c}^* \geq 0$, no extreme point will improve the solution, so the optimal solution to the master problem is $\mathbf{x}^*_{RMP}$ with objective value $z^*_{RMP} = z^*$.

2.4 Column generation applied to the tail assignment problem

In this section, the application of column generation to the tail assignment problem is presented, which also provides an explanation of the *warm start* and *cold start* that will be controlled by machine learning later in the thesis.

The tail assignment problem as formulated in (2.1) is a binary linear optimization problem with an exponential number of variables. The solution method used at Jeppesen Systems is called linear optimization branch-and-price [11]. In this method, the binary requirements are first relaxed, turning the TAP into the linear program:

$$z^* = \min \sum_{r \in \mathcal{R}} c_r x_r, \tag{2.12a}$$

$$\text{s.t} \sum_{r \in \mathcal{R}} a_{f,r} x_r = 1, \quad f \in \mathcal{F}, \tag{2.12b}$$

$$0 \leq x_r \leq 1, \quad r \in \mathcal{R}. \tag{2.12c}$$

This LP is then solved using column generation. To provide a set of initial extreme points for the first restricted master problem, the extreme point corresponding to using all decision variables in $\mathcal{R}$ that correspond to leaving activities unassigned, as mentioned in Section 2.1.1 is added. The RMP is then solved using subgradient optimization to get a dual solution π for the pricing problem, corresponding to the constraints (2.12b). An example of using subgradient optimization along with column generation can be seen in [12].

When using subgradient optimization, the constraints (2.12b) are relaxed and an initial feasible solution in the dual space is selected. One initial solution that will always work is the origin. Using the origin as an initial solution will be called using *cold start*. Another possible initial solution that can be used after solving the RMP and having to re-solve it with more columns is the optimal dual solution from the previous RMP. Using this initial solution is known as using *warm start*.

The pricing problem can be interpreted as finding some new route that has a minimal reduced cost with respect to the dual vector $\pi = \{\pi_f\}_{f \in \mathcal{F}}$. The total reduced cost of a route is then:

$$\bar{c}_r = c_r - \sum_{f \in \mathcal{F}} a_{f,r} \pi_f. \tag{2.13}$$

Since $a_{f,r}$ is only equal to one if activity f is in route r, define the set $\mathcal{F}_r$ to be the subset of activities encountered during route r. Then $\sum_{f \in \mathcal{F}} a_{f,r} \pi_f = \sum_{f \in \mathcal{F}_r} \pi_f$. Under the assumption that the cost of a route c_r is the sum of the costs d_f of each activity $f \in \mathcal{F}$ within the route, this turns the reduced cost of a route r into:

$$\bar{c}_r = \sum_{f \in \mathcal{F}_r} (d_f - \pi_f). \tag{2.14}$$

This means that if the connections between activities are known, the pricing problem becomes a shortest path problem about finding the route in $\mathcal{R}$ with the lowest cost where the cost of traversing a connection leading into activity $f \in \mathcal{F}$ is $d_f - \pi_f$ [1]. Due to the route having to be valid with respect to the individual constraints to be in $\mathcal{R}$, (2.11) becomes a resource-constrained shortest path problem, which is

NP-Hard. A solution method to the pricing problem is outside the scope of the thesis, but it can be found in [1].

Note that when subgradient optimization is used for a fixed number of iterations, a non-optimal dual solution is likely found. However, close-to-optimal dual values do perform well when generating new routes [11] [12], and since subgradient optimization usually performs faster on these types of problems compared to the simplex method, which is another solution method for linear programs, it is still worth using subgradient optimization instead. However, due to the dual values not necessarily being optimal, using warm start or cold start might give different dual values to the pricing problem which may affect what columns are generated by the pricing problem and thus affect the solution quality and running time of the column generation.

As for which start is the best for a given iteration, it's difficult to say. Brevis [13] claims that while there are theoretical conditions where warm start is always better than cold start, there are also occurrences when warm start does not really affect the solution time compared to cold start. This motivates the thesis' aim of investigating if machine learning can at least improve the decision between warm start and cold start.

2.5 Machine learning

This section describes the machine learning theory and algorithms that will be used in the thesis.

2.5.1 Data sets

A *data set* [14] $\mathcal{D}$ is a collection of data or information. Each row i in a data set is called as an *instance* or *observation*. An observation i consists of an *input vector* $\mathbf{x}_i = \{x_{i,j}\}_{j \in F}$ where F is the set of *features*. The number of features $|F|$ will be denoted as m. In other words, $x_{i,j}$ contains the data corresponding to feature j in the observation i and $\mathbf{x}_i$ can be written as an m-dimensional vector. The observation $\mathbf{x}_i$ may also contain an associated *label* or *class* y_i chosen from a set $\mathcal{L}$. If $\mathcal{L}$ is discrete, the size of the set will be denoted by L. If none of the observations in the data set has a label associated with them, the data set is called an *unlabelled data set* [15]. If all observations in the data set have labels, it is called a *labelled data set*. In other words, if the size of the data set is n, then for an unlabelled data set, $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^{n}$ while for a labelled data set, $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$.

2.5.2 Types of machine learning

Machine learning [16] is a branch of artificial intelligence [17] that aims to let machines learn automatically and improve the solution of a task from experience without being explicitly programmed to deal with the task. All machine learning models take $\mathbf{x}_i$ as the input while the output of the model varies depending on the task which the machine learning algorithm is trying to solve. The two main types of machine learning are *supervised learning* [18] and *unsupervised learning* [19]. In supervised learning, a labelled data set is used to train the machine learning algorithm to predict y_i based on $\mathbf{x}_i$ for unknown $\mathbf{x}_i$. In unsupervised learning, an unlabelled data set is used. The model is then instead trained to find the underlying patterns in the data. This can sometimes be used to preprocess data for a supervised learning problem. There is also semi-supervised learning [20], a variation of supervised learning where only some of the data have labels, and reinforcement learning [21], where the machine is given rewards when it selects actions and it learns to select actions that give the maximum long-term reward. For the rest of the thesis, the focus will be on supervised learning, with a bit of unsupervised learning being used when preprocessing the data.

Supervised learning is divided into two types, namely *regression* [22] and *classification* [23], based on the type of label set y_i is in. In regression, the label set $\mathcal{L}$ is continuous and the model estimates what value each input $\mathbf{x_i}$ should have. On the other hand, if $\mathcal{L}$ is discrete, the problem is a classification problem, and the model is instead used to classify $\mathbf{x_i}$ as one or more of the existing labels. Classification is further divided into *binary classification* where there are only two class labels and $\mathbf{x_i}$ can only be of one class, *multi-class classification* where there are more than two class labels but $\mathbf{x_i}$ can still only be of one class, and *multi-label classification* where $\mathbf{x_i}$ can be of any number of classes.

2.5.3 Common terminologies in Machine learning

A *model* is a function which maps the input features with the output labels. It represents the relation between the input features and output labels of a data set. The model has *parameters*, which are internal configuration variables which decide the exact mapping of input features. The parameters are fit to the data by using a data set known as the *training data set.* How they are fitted depends on the model, but fitting the parameters with a training data set is known as training the model. *Hyperparameters* are external configuration variables which helps in controlling the learning process and to estimate the model parameters. They are often set by using heuristics prior to the training and not changed during the training which affects how quickly and reliably the model trains, but can also be fine tuned by a data set separate from the training data set known as the *validation data set.* Parameters and hyperparameters are unique for a specific machine learning algorithm.

A *testing data set* is a data set which is used to test whether a trained model generalises well for previously unseen data which is separate from the training and validation data set. The purpose of the testing data set is to see if the model generalizes well to observations it has not been trained on. An *outlier* is an observation which deviates largely from the other observations in the data set. Outliers may affect the training of the model in a way that goes against the purpose of the model.

The difference between observed value and the expected predicted value when training the model on different data sets is called the *bias*, which is simply the assumptions made by the model. A low bias model makes fewer assumptions whereas the high bias model makes more assumptions on the target function. *Variance* is defined by how much predictions change when trained by different data sets [24]. *Overfitting models* are models which are highly dependent on the training data and gives a good performance on training data but a poor generalization for new unknown data. Such models usually have a high variance and low bias. On the other hand *underfitting models* have poor performance on both training data and poor generalization for new unknown data. Usually underfitting models have a high bias and low variance. Bias and variance are inversely related to each other, so a good model balances the bias and the variance.

2.5.4 Machine learning workflow

The Machine learning workflow mainly consists of five steps. Defining the problem, building the data set, training the model, evaluating the model and using the model. Initially, the problem has to be clearly defined and has to be a specific question rather than a general question. For example, instead of defining the problem as 'How can the solution quality of the column generation framework be improved using machine learning?' Even though this is a valid question, it is a generic question and not a specific one. Instead specific tasks can be defined as 'How could formulating different strategies with different variations of start improve the solution quality of the column generation framework?'. This helps to solve the problem in an easier and faster way by identifying what machine learning task can be used to solve this problem. Also, it helps in better understanding the data which is needed for solving the problem. Along with this, the type of machine learning task has to be correctly identified as it also helps in understanding what kind of data is required for solving the defined problem. Building an appropriate data set is one of the most crucial steps in machine learning since the model can only solve a task well if the data represents the task well. Building the data set involves both collecting the data and preprocessing it. Preprocessing mainly consists of taking care of missing data and other things that will make the data easier to understand, like feature scaling or a principal component analysis. Understanding the data helps in determining suitable machine learning models which can give effective solutions.

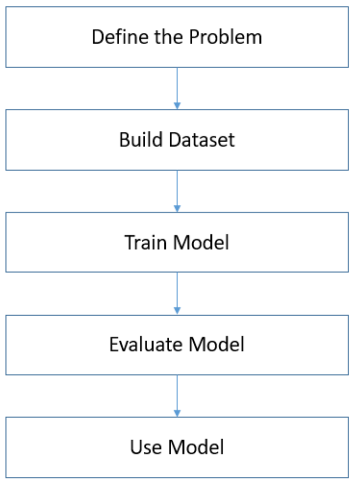


Figure 2.1: The workflow for any given machine learning problem.

The data set is split into a training data set and a testing data set after preprocessing. A machine learning algorithm is selected and the model is trained with the training data to learn the model parameters. The learned model is tested against the testing set to see how well the model generalizes for previously unseen new data. The model is evaluated using the different evaluation metrics to determine the quality of the model. Based on the results, the hyperparameters can be modified to improve the model training. Once a good quality model is created, the model can be used to solve the defined problem. The model is continuously monitored and improved by adding new data to the data set. A full view of the workflow can be seen in Figure 2.1.

2.5.5 Data preprocessing

2.5.5.1 Feature scaling

Most machine learning algorithms are dependant on the size of the range of values on each feature, so it's important to make sure that each feature is considered on the same scale. An example is that any algorithm that uses the Euclidean distance between data to classify will consider features that is on a larger scale more important than features on a smaller scale. To fix this, feature scaling [25] is implemented. Feature scaling is a method of putting features on different scales, onto the same scale. The main two feature scaling methods are *normalization* and *standardization.*

In normalization, the features are scaled to be within the interval $[0, 1]$ using the transformation

$$x_{i,j}^{\text{norm}} = \frac{x_{i,j} - x_{\text{min}j}}{x_{\text{max}j} - x_{\text{min}j}}, \tag{2.15}$$

where $x_{\text{min}j}$ and $x_{\text{max}j}$ is the minimum and maximum value of feature $j \in \mathcal{F}$ in the data, respectively.

While normalization could be considered useful since it puts features in a bounded interval, in practice, *standardization* is used more often. In standardization, the features are scaled to have mean 0 and variance 1. This is done with the transformation

$$x_{i,j}^{\text{stand}} = \frac{x_{i,j} - \mu_j}{\sigma_j}, \tag{2.16}$$

where μ_j and σ_j is the sample mean and standard deviation, repectively, of the data for feature $j \in \mathcal{F}$. The main advantage of standardization is that the overall shape of the distribution does not change.

2.5.5.2 Principal Component Analysis

When using machine learning, the number of input features m could be so large that the model takes too long to train and predict. As a result, it is sometimes desirable to lower the dimension of the input to reduce complexity. The goal is to reduce the dimension of the data while retaining as much information as possible. A technique that can be used to reduce the dimension of the vector is a statistical method known as a *principal component analysis* (PCA) [26]. PCA finds the principal components of the input data in $\mathcal{D}$. The first principal component is the unit vector α_1 within the m-dimensional space where the variance of:

$$\alpha_1^T \mathbf{x} = \sum_{j=1}^{m} \alpha_{1j} \mathbf{x}_j \tag{2.17}$$

is maximal, where $\mathbf{x}$ is a random observation $\mathbf{x}_i$ from $\mathcal{D}$. The second component is defined by being the unit vector α_2 which is uncorrelated with α_1 and where $\alpha_2^T \mathbf{x}$ has maximal variance. In the same vein, the p:th principal component is the unit vector α_p which is uncorrelated with the vectors $\alpha_1, \alpha_2, ..., \alpha_{p-1}$ where $\alpha_p^T \mathbf{x}$ has maximal variance. The reason why one would want to find the principal components of a data set is that by using the first couple of principal components as the inputs in the data set, the dimension of the input can become a lot smaller while keeping most of the variance of the data set.

To provide a visual example of this, Figure 2.2 shows data sampled from the multinomial Gaussian distribution with mean around the origin and the covariance matrix $\Sigma = \begin{bmatrix} 1 & 3 \\ 3 & 5 \end{bmatrix}$, along with the principal components of the data. In this case, $\alpha_1 = (0.47, 0.88)$ and $\alpha_2 = (-0.88, 0.47)$. It can be observed that α_1 points in the direction that the data varies most in, which is easier seen in Figure 2.3, which is the same data but transformed linearly so the axes are along the principal components. Another thing to note is that α_1 and α_2 are orthogonal to each other. This is a side effect of the vectors being uncorrelated. In fact, the first p principal components will always provide an orthogonal p-dimensional basis of the data.

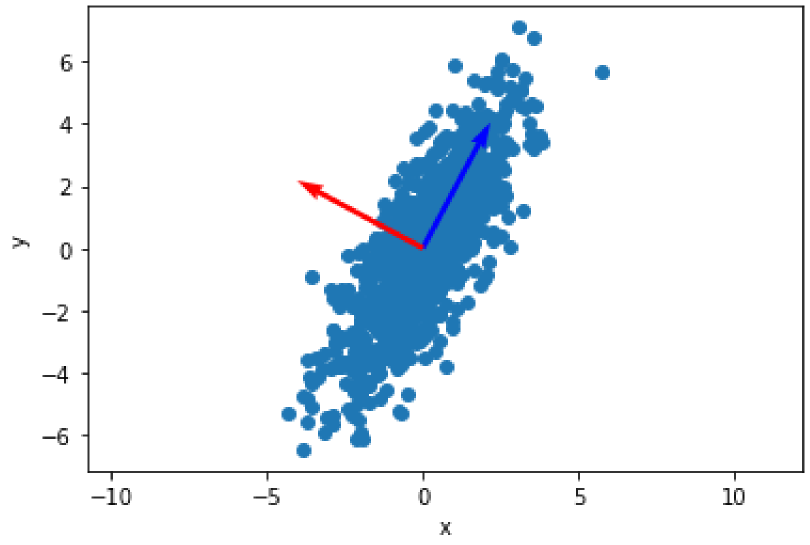


Figure 2.2: Plot of sampled example data with the first and second principal component in blue and red, respectively.

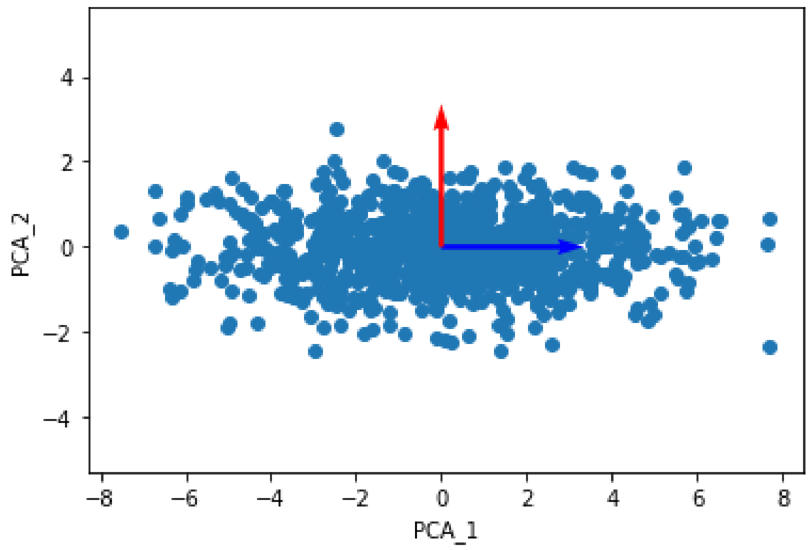


Figure 2.3: Plot of the same example data transformed with respect to the principal components.

In order to find the principal components, the covariance matrix Σ is defined as the $m \times m$ matrix where $\Sigma_{j,k}$ is the covariance between feature j and feature k. The p:th principal component turns out to be the eigenvector of Σ corresponding to the p:th largest of Σ's eigenvalues [26]. The proof that this applies to the first principal component is obtained by first noting that the variance of $\alpha_1^T \mathbf{x}$ can be rewritten as $\alpha_1^T \Sigma \alpha_1$. Thus, finding the unit vector that maximizes the variance is equivalent to solving the problem:

$$\text{maximize } \alpha_1^T \Sigma \alpha_1, \tag{2.18a}$$

$$\text{s.t } \alpha_1^T \alpha_1 = 1. \tag{2.18b}$$

This is solved by using Lagrangian relaxation to see that this is equivalent to maximizing:

$$h(\lambda) = \alpha_1^T \Sigma \alpha_1 - \lambda(\alpha_1^T \alpha_1 - 1), \tag{2.19}$$

which when derived with respect to α_1 gives the necessary local optimality condition:

$$\Sigma \alpha_1 - \lambda \alpha_1 = (\Sigma - \lambda \mathbf{I}_n)\alpha_1 = \mathbf{0}. \tag{2.20}$$

This equation is only satisfied when λ is an eigenvalue and α_1 is the corresponding eigenvector. That the eigenvalue has to be the largest eigenvalue in particular to have a global optimum can be seen by noting that:

$$\alpha_1^T \Sigma \alpha_1 = \alpha_1^T \lambda \alpha_1 = \lambda \alpha_1^T \alpha_1 = \lambda. \tag{2.21}$$

By induction, it is possible to show that this also holds for $p = 2, 3, ..., m$. A proof for $p = 2$ is found in [26], which can be generalized to hold for all $p \leq m$.

Usually, Σ is unknown, so the sampled covariance matrix S is calculated instead from:

$$S_{j,k} = \frac{1}{n-1} \sum_{i=1}^{n} (x_{i,j} - \bar{x_j})(x_{i,k} - \bar{x_k}), \tag{2.22}$$

where $\bar{x_j} = \frac{1}{n} \sum_{i=1}^{n} (x_{i,j})$. After calculating the sampled covariance matrix, the eigenvectors of it are found by using the single value decomposition of the matrix. More details about how this is obtained is found in [26].

2.5.6 Multi-class classification algorithms

This section describes the multi-class algorithms that will be used for analysing the data in Chapter 3.

2.5.6.1 Naive Bayes

Naive Bayes [27] is a multi-class classification algorithm which is based on the Bayes theorem [28]. Bayes theorem gives the probability of an event, based on the prior knowledge of probability of another event related to it. It states that the probability of an observation $\mathbf{x} = x_1, x_2, ..., x_n$ belonging to the class y is defined as below:

$$P(y|\mathbf{x}) = \frac{P(y)P(\mathbf{x}|y)}{P(\mathbf{x})}. \tag{2.23}$$

In (2.23), $P(y)$ is the probability of an event y, without any reference to the data known as the prior probability of y. This can be estimated by simply taking the percent of the training data belonging to class y. $P(\mathbf{x}|y)$ is the probability of the observation $\mathbf{x}$ given that it belongs to class y known as the likelihood. $P(y|\mathbf{x})$ is the probability of an observation belonging to the class y given an observation $\mathbf{x}$ known as the posterior probability.

This algorithm works mainly on a naive assumption that each feature in the data set contributes independently and equally to the class label. With $P(\mathbf{x})$ being a constant, it can be changed to a proportionality.

$$P(\mathbf{x}|y) = P(x_1, x_2, ..., x_n|y) = \prod_{i=1}^{n} P(x_i|y), \tag{2.24}$$

$$P(y|\mathbf{x}) \propto P(y) \prod_{i=1}^{n} P(x_i|y). \tag{2.25}$$

Here, $P(x_i|y)$ is the probability of x_i having its current value, given the label y. This can be estimated by making assumptions on the overall distribution of the data. For example, one can assume that $x_i|y$ is Gaussian, estimate the distributions using the sample mean $\mu_{j,l}$ and standard deviation $\sigma_{j,l}$ for each feature $j \in \mathcal{F}$ and label $l \in \mathcal{L}$.

In real world scenarios, it is quite difficult to get different features which are completely independent of each other, but still this naive assumption works well in practice. It is usually easy to build and performs well in multi-class predictions.

2.5.6.2 Support Vector Machine

A supervised learning algorithm which can be used for both regression and classification tasks is the *Support Vector Machine* (SVM) [29]. SVM finds a decision boundary which can separate the m-dimensional space into two classes. The decision boundary is in the form of a $(m-1)$-dimensional *hyperplane*. The hyperplane always tries to maximize the distance between the data points. The data points which are used to determine this hyperplane are called *support vectors*. The distance between the support vectors and the hyperplane is the *margin*. The *optimal hyperplane* is a hyperplane with the maximum possible margin and the algorithm aims to determine one such hyperplane.

Consider the data set $\mathcal{D}$ containing n observations $\{\mathbf{x_i}\}_{i=1}^{n}$ in a vector space of the features $\mathcal{F}$, $\mathbf{V} \in \mathcal{R}^m$, with associated output labels in the binary set $y_i \in \{-1, 1\}^n$. The support vector of class 1 will be denoted by $\mathbf{x}_+$ and the support vector of class -1 will be denoted by $\mathbf{x}_-$ [30].

In the feature space $\mathbf{V}$, the algorithm identifies the optimal hyperplane $\mathbf{h}$ in the vector space $\mathbf{V}$, which can be represented by a normal vector $\mathbf{w}$ and a parameter $\mathbf{b}$ with the hyperplane residing at the point $\mathbf{w}^T\mathbf{x} + b = 0$. and with the support planes residing at $\mathbf{w}^T\mathbf{x} + b = 1$ when $\mathbf{x} = \mathbf{x}_+$ and $\mathbf{w}^T\mathbf{x} + b = -1$ when $\mathbf{x} = \mathbf{x}_-$.

Since the optimal hyperplane is supposed to maximize the margin, which is the distance between the support vectors projected onto the direction of $\mathbf{w}$, the goal is to maximize $\frac{1}{||\mathbf{w}||}(\mathbf{x}_+ - \mathbf{x}_-)^T\mathbf{w} = \frac{\mathbf{x}_+^T\mathbf{w} - \mathbf{x}_-^T\mathbf{w}}{||\mathbf{w}||}$. By the definitions of the support planes:

$$\mathbf{x}_+^T\mathbf{w} = 1 - b, \tag{2.26}$$
$$\mathbf{x}_-^T\mathbf{w} = -1 - b, \tag{2.27}$$

meaning that the margin can be rewritten as:

$$\frac{\mathbf{x}_+^T\mathbf{w} - \mathbf{x}_-^T\mathbf{w}}{||\mathbf{w}||} = \frac{1 - b - (-1 - b)}{||\mathbf{w}||} = \frac{2}{||\mathbf{w}||}. \tag{2.28}$$

So overall, the goal is to maximize $\frac{2}{||\mathbf{w}||}$. In order to simplify the problem, the problem is inverted and multiplied with the positive value $||\mathbf{w}||$ to make the equivalent problem to minimize $\frac{1}{2}||\mathbf{w}||^2 = \frac{1}{2}\mathbf{w}^T\mathbf{w}$.

Since the support vectors are supposed to be the closest of their respective class to the hyperplane, this means that all of them have to exist on one side of the supporting hyperplane of their class. For observations which belong to class 1, this means that $\mathbf{w}^T\mathbf{x} + b \geq 1$ and for the observations which belong to class -1 it means that $\mathbf{w}^T\mathbf{x} + b \leq -1$. This can be summarized as $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$ for all $i \in \{1, ..., n\}$. This, combined with the calculation of the margin leads to the optimization problem:

$$\min \quad \frac{1}{2}\mathbf{w}^T\mathbf{w}, \tag{2.29a}$$
$$\text{s.t} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \qquad i \in \{1, ..., n\}, \tag{2.29b}$$

where the solution gives the $\mathbf{w}$ and b defining the optimal hyperplane.

Sometimes however, the data does not allow all points to be outside the margin. To counter this, let $\zeta_i \geq 0$ denote the amount that data point i is allowed outside the margin and let $C \geq 0$ be the cost of a data point being outside the margin. Then the problem instead becomes:

$$\min \quad \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n}\zeta_i, \tag{2.30a}$$
$$\text{s.t} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \zeta_i, \qquad i \in \{1, ..., n\}, \tag{2.30b}$$
$$\zeta_i \geq 0, \qquad i \in \{1, ..., n\}. \tag{2.30c}$$

The description so far of SVM only constructs a linear boundary, which will not fit problems that are linearly separable. To fix this, one uses *kernels* [30]. A kernel is a function $k(\mathbf{x}, \mathbf{x}')$ that outputs some measure of similarity to them. Some examples are the dot product $\mathbf{x}^T\mathbf{x}'$, the polynomial kernel $(\mathbf{x}^T\mathbf{x}' + c)^d$ for some $c \geq 0$ and positive integer d and the radial basis kernel (RBF) $e^{-\gamma||\mathbf{x}-\mathbf{x}'||^2}$ for some $\gamma \geq 0$. The important thing about kernels is that they can be viewed as the dot product of $\mathbf{x}$ and $\mathbf{x}'$ after being transformed into some, possibly higher dimensional, vector space. In other words the kernel can be written as:

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T\phi(\mathbf{x}'). \tag{2.31}$$

Thus, nonlinear problems can be handled by replacing the dot product in Constraint (2.30b) by the function $w^T \phi(\mathbf{x})$ to create the problem:

$$\begin{aligned} \min \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n} \zeta_i, && && (2.32a) \\ \text{s.t} \quad & y_i(\mathbf{w}^T\phi(\mathbf{x_i}) + b) \geq 1 - \zeta_i, && i \in \{1, ..., n\}, && (2.32b) \\ & \zeta_i \geq 0, && i \in \{1, ..., n\}. && (2.32c) \end{aligned}$$

After deciding the separating hyperplane, new data points are classified by calculating their relation with the hyperplane using $\mathbf{w}^T\phi(x) + b$ and choosing the sign as the final class prediction.

Observe that unlike the supervised learning algorithm above and below, SVM is in its most basic form a binary classifier and not a multi-class classifier. However, it can be extended to handle multi-class problems through either a *one-versus-rest* or *one-versus-one* approach [31]. In the one-versus-rest approach, one trains L binary classifiers on each data point representing each class. Each classifier tries to separate its represented class from the rest. However, this may cause the binary classifiers to say that the data belongs to more than one class. The binary classifiers will also likely suffer from an imbalanced data set, as for example normally balanced data set with five classes gives all classifiers data consisting of $\frac{1}{5}$ positive and $\frac{4}{5}$ negative examples. In the one-versus-one approach, $L(L-1)/2$ classifiers are trained to separate data points of each pair of classes. Then, the classifiers vote which class in their pair the data should be classified as and the most voted on is picked as the final class. However, this takes a longer time to train and classify than the one-versus-rest approach and can still lead to the votes concluding that the data belongs to several classes.

2.5.6.3 Decision tree

A *decision tree* is a model based around recursively partitioning the input space and defining another model in the respective region in the form of a tree [31]. Consider that the input data $\{\mathbf{x}\}_{i=1}^{n}$ has feature values $\mathcal{T}_j,\ j \in \mathcal{F}$. Then a tree is constructed by splitting the data set based on whether the variable $j^* \in \mathcal{F}$ is above or below value $t^* \in \mathcal{T}_j$ where j^* and t^* minimizes:

$$\text{cost}(\{\mathbf{x}_i, y_i : x_{i,j^*} \leq t^*\}) + \text{cost}(\{\mathbf{x}_i, y_i : x_{i,j^*} > t^*\}), \tag{2.33}$$

where the cost function is a function that measures how well the split separates the different labels. In this case, either the *Gini impurity* [31] defined by:

$$G(\mathcal{D}) = 1 - \sum_{c=1}^{C} \hat{\pi}_c^2, \tag{2.34}$$

or the *entropy* defined by:

$$H(\mathcal{D}) = -\sum_{c=1}^{C} \hat{\pi}_c \ log(\hat{\pi}_c), \tag{2.35}$$

is used where $\hat{\pi}_c$ is the fraction of the data set $\mathcal{D}$ that is of class c. The Gini impurity encourages data sets where each label is either used often or used as little as possible, with the best value found when the data set is pure, that is, it only consist of one label. After splitting the data set into two, the same splitting is done on two new data sets recursively until either a fixed depth set as a hyperparameter beforehand is reached or until the data has been separated into only pure data sets at each leaf. An example of a decision tree can be seen in Figure 2.4 for a data set with four entries. The full data set contains $(\mathbf{x}_1, y_1) = ([1, 1], 1)$, $(\mathbf{x}_2, y_2) = ([-1, 1], 0)$, $(\mathbf{x}_3, y_3) = ([1, -1], 0)$ and $(\mathbf{x}_4, y_4) = ([-1, -1], 1)$. In the decision tree it first splits the data set in two at $x_{i,0} \leq 0$ so x_2 and x_4 end up in the left branch while x_1 and x_3 end up in the right branch, with both branches having one data point from each label. Both of these are then split at $x_{i,1} \leq 0$ to create a leaf for each of the data points where each leaf is completely pure.

When classifying a new input, one simply traverses the tree using the splits to determine which branch to take until a leaf node is reached, with the most common class in the leaf node being chosen as the class of the input. For example, if $\mathbf{x} = [3, -5]$ and the tree in Figure 2.4 is used, the first branch will see that $x_0 > 0$ and pick the right branch and then choose the left branch since $x_1 \leq 0$. It then ends up in a leaf node that only contains data with label 0, so the label chosen ends up being 0.

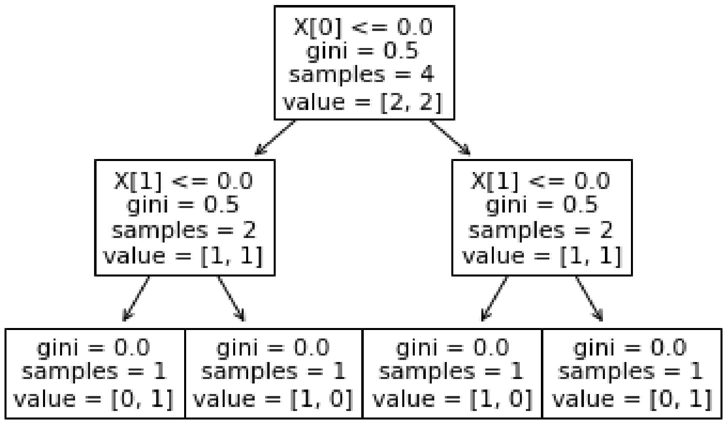
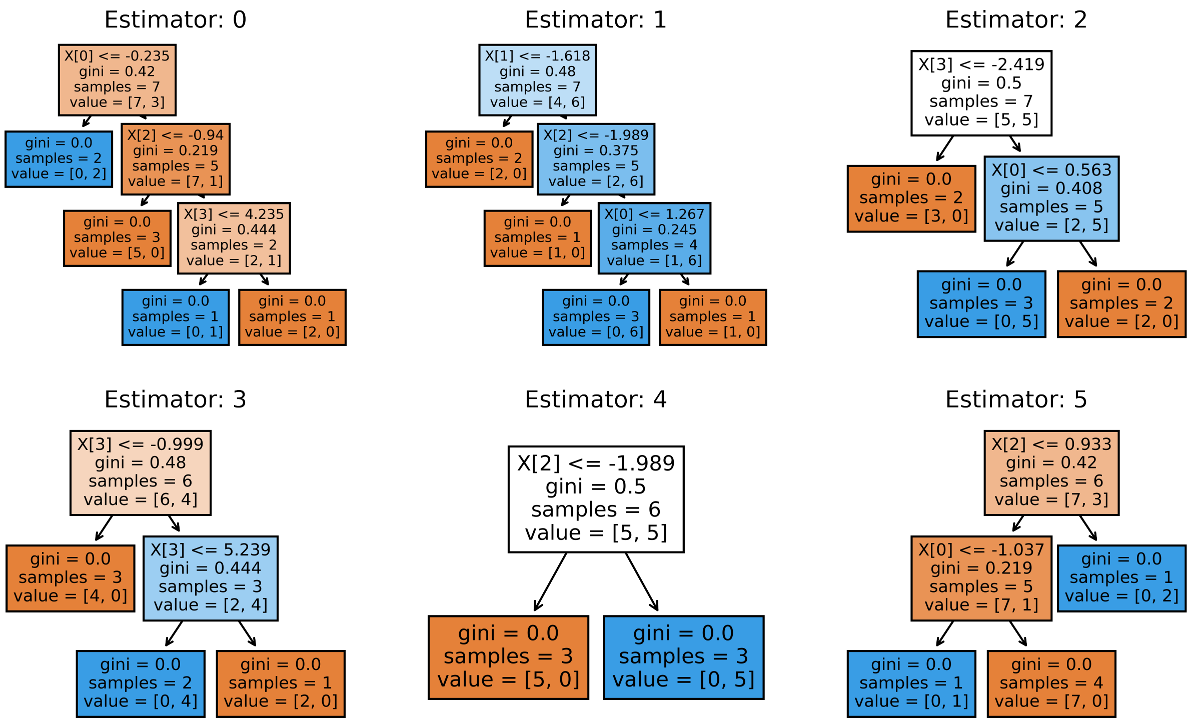


Figure 2.4: Example image of a decision tree taken from the sklearn Python package.

The main advantage to decision trees is that they are easy to interpret and they scale well with large data sets [31]. However, they are not as accurate as other models and decision trees have a higher variance than most models since small changes in the data can have a large impact on the construction of the tree.

2.5.6.4 Random forest

A *random forest* [31] is an ensemble method which aims to reduce the large variance of the decision trees by combining the results of many separate decision trees. This is done by training a set of M decision trees on different subsets of the original data which are chosen randomly with replacement and constructed by only using a random subset of the variables at each branch. The purpose of this is to try to make the decision trees as uncorrelated as possible. An example of this when $M = 6$ can be found in Figure 2.5. Since the data is chosen randomly with replacement, it leads to the different trees having different distributions of the labels and along with the random features to consider at each branch causes the splits to occur differently for each tree.



Estimator: 0
X[0] <= -0.235 gini = 0.42 samples = 7 value = [7, 3]
gini = 0.0 samples = 2 value = [0, 2]
X[2] <= -0.94 gini = 0.219 samples = 5 value = [7, 1]
gini = 0.0 samples = 3 value = [5, 0]
X[3] <= 4.235 gini = 0.444 samples = 2 value = [2, 1]
gini = 0.0 samples = 1 value = [0, 1]
gini = 0.0 samples = 1 value = [2, 0]
Estimator: 1
X[1] <= -1.618 gini = 0.48 samples = 7 value = [4, 6]
gini = 0.0 samples = 2 value = [2, 0]
X[2] <= -1.989 gini = 0.375 samples = 5 value = [2, 6]
gini = 0.0 samples = 1 value = [1, 0]
X[0] <= 1.267 gini = 0.245 samples = 4 value = [1, 6]
gini = 0.0 samples = 3 value = [0, 6]
gini = 0.0 samples = 1 value = [1, 0]
Estimator: 2
X[3] <= -2.419 gini = 0.5 samples = 7 value = [5, 5]
gini = 0.0 samples = 2 value = [3, 0]
X[0] <= 0.563 gini = 0.408 samples = 5 value = [2, 5]
gini = 0.0 samples = 3 value = [0, 5]
gini = 0.0 samples = 2 value = [2, 0]
Estimator: 3
X[3] <= -0.999 gini = 0.48 samples = 6 value = [6, 4]
gini = 0.0 samples = 3 value = [4, 0]
X[3] <= 5.239 gini = 0.444 samples = 3 value = [2, 4]
gini = 0.0 samples = 2 value = [0, 4]
gini = 0.0 samples = 1 value = [2, 0]
Estimator: 4
X[2] <= -1.989 gini = 0.5 samples = 6 value = [5, 5]
gini = 0.0 samples = 3 value = [5, 0]
gini = 0.0 samples = 3 value = [0, 5]
Estimator: 5
X[2] <= 0.933 gini = 0.42 samples = 6 value = [7, 3]
X[0] <= -1.037 gini = 0.219 samples = 5 value = [7, 1]
gini = 0.0 samples = 1 value = [0, 2]
gini = 0.0 samples = 1 value = [0, 1]
gini = 0.0 samples = 4 value = [7, 0]

Figure 2.5: Example of a random forest with six trees.

The trees then vote on what class a new input $\mathbf{x}$ should be classified as and the most commonly picked voted label is then picked as the label for $\mathbf{x}$. This model has the advantage of having a smaller variance compared to the decision tree. Unfortunately, since we use multiple, mostly uncorrelated trees, the decisions made by the trees collectively is hard to interpret, compared to the simple interpretation of the decision tree.

Another aspect of the random forest classifier is that it can be used to analyze the importance of features by looking at the average decrease in the Gini impurity for all the decision trees in the forest that is associated with each feature [25]. This helps in developing a model which will focus mainly on the relationship between the most important features and the output label.

2.5.6.5 Artificial neural networks

An *artificial neural network* is a computing model inspired by the neural network in the human mind [32]. The formulation of the model in our report comes from [32] and [33]. Artificial neural networks consists of elements known as *neurons* which are bound to each other by directed *connections* which have given weights. Some of the neurons will be *input neurons* that take in values from other data and the values are then propagated to the network until it reaches some neurons known as *output neurons* which output the final prediction.

A neuron is sent a variable *input* $\mathbf{x} = \{x_1, ..., x_m\}$ from all connections directed to it with the associated *weights* $\mathbf{w} = \{w_1, ..., w_m\}$, as well as a constant input $x_0 = 1$ with weight w_0. The constant input together with it's weight is known as the *bias* of the neuron. These inputs and weights are then aggregated by a function $f(\mathbf{x}, \mathbf{w})$ to a value u which is known as the *signal.* Usually this function is the weighted sum $f(\mathbf{x}, \mathbf{w}) = \sum_{i=0}^{m} w_i x_i$. The signal is then put through an *activation function* $s = g(u)$ to determine the activation level of the neuron which is then sent as the output to all outgoing connections from the neuron. The activation function can either be linear or nonlinear, but the most common are nonlinear. Some choices for the activation function include the sigmoid function $\sigma(u) = \frac{1}{1+e^{-u}}$, and $\tanh(u)$, since they limit the possible values of s to the intervals $[0, 1]$ and $[-1, 1]$, respectively. Another common activation function is the Rectified Linear Unit (ReLU) function:

$$ReLU(u) = \begin{cases} 0, & u \leq 0, \\ u, & u > 0. \end{cases} \tag{2.36}$$

There are many ways that the neurons can be connected to each other, but the architecture that will be the focus of this thesis is what is known as a *multi-layered perceptron.* In this network, the neurons are structured in *layers* $l \in \{0, ..., L\}$, where all connections from neurons within a layer are only connected to neurons within the next layer. Layer 0 consist of the input neurons and is known as the *input layer* while layer L consists of the output neurons and is called the *output layer.* The layers between the input and output layer are known as *hidden layers.* The goal is then to set the weights on the connections in such a way that data given to the input gives an output close to the desired output.

Before discussing how the multi-layered perceptron learns these weights, there are some extra notations that will be useful both to understand how the weights are learnt and how the perceptron itself predicts the output. Let n_l denote the number of nodes in layer $l \in \{0, ..., L\}$. The weights that connect nodes in layer l to nodes in layer $l-1$ will be put in the matrix W^l of size $n_l \times (n_{l-1} + 1)$. Element $w_{j,i}^l$ is then the weight of the connection between node j in layer l and node i in layer $l-1$ for $j \in \{1, ..., n_l\}$ and $i \in \{0, ..., n_{l-1}\}$. Note that the weights of the connections from the biases of the neurons in layer $l-1$ are included in the weight matrix. Let I_j^l be a vector consisting of the signal value given to nodes $j \in \{1, ..., n_l\}$ in layer $l \in \{1, ..., L\}$ and let Y_j^l be the vector with the output value of nodes $j \in \{1, ..., n_l\}$

in layer $l \in \{1, ..., L\}$. These can be defined recursively by:

$$I_j^1 = \sum_{i=0}^{n_0} W_{j,i}^0 x_i, \tag{2.37}$$

$$Y_j^l = g(I_j^l), , \quad l \in \{1, ..., L\}, \tag{2.38}$$

$$I_j^l = \sum_{i=0}^{n_l} W_{j,i}^l Y_i^{l-1}, \quad l \in \{2, ..., L\}, \tag{2.39}$$

where x_i is the input given in the input layer and $Y_0^{l-1} = 1$ for $l \in \{1, ...L-1\}$.

Figure 2.6 provides an example of a multilayered perceptron with $L = 3$, an input layer, two hidden layers and an output layer. First, the input data $\{x_i\}_{i=1}^4$ in layer 0 is sent alongside the bias x_0 through the connections to the nodes in layer 1 with the weights $W_{j,i}^0$. These are then summed up to calculate I_j^1 which is then put through the activation function g to obtain Y_j^1 for $j \in \{1, 2, 3\}$. These are then sent to layer 2 alongside the bias Y_0^1 which are then summed with the weights $W_{j,i}^1$ to obtain I_j^2 and Y_j^2 for $j \in \{1, 2\}$. This process is then repeated to get Y_j^3 for $j \in \{1, 2, 3\}$ which is the final output of the network.

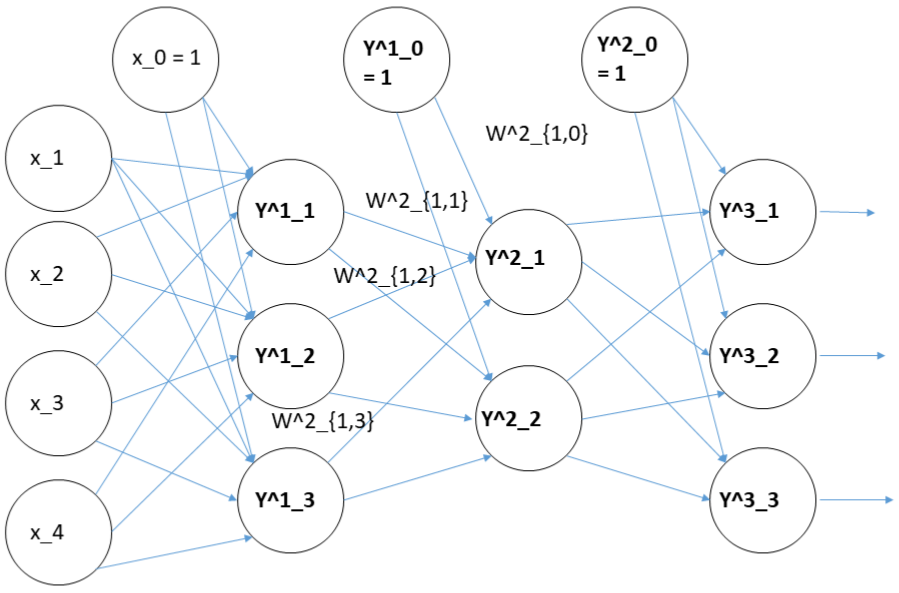


Figure 2.6: Picture of a multilayered perceptron with one hidden layer and its connections. The weights going into node 1 in the second hidden layer is explicitly mentioned as well.

In order to let the neural network learn the proper weights, a function that measures the deviation of the current prediction and the desired output is employed. This function is known as a *loss function* [30]. In this case, the square error is used for a single training instance. It is defined as:

$$E(k) = \frac{1}{2}\sum_{j=1}^{n_L}(y_j(k) - Y_j^L(k))^2, \tag{2.40}$$

where $y_j(k)$ is the desired output of node j and $Y_j^L(k)$ is the actual output from the neural network for the k:th training instance, respectively. In the case where a set of p training instances is used as a batch, the mean square error is used instead:

$$\mathcal{E} = \frac{1}{p}\sum_{k=1}^{p} E(k). \tag{2.41}$$

The goal of the training is to adjust the weights so that $\mathcal{E}$ is minimized. This can be done in many different ways, but it is usually based on algorithm known as *back-propagation* [33], which is based on gradient descent. Two that will be used in the thesis is *stochastic gradient descent* (SGD) [32] and *Adam* [34].

Define $\eta \in (0,1)$ as the learning rate, that is, the length of the gradient descent steps. Divide the training data into batches of size p and feed the data in the batch forward to the network to predict the output of the data. The backpropagation algorithm calculates the gradient $\frac{\partial \mathcal{E}}{\partial W_{j,i}^l}$ of the weights of each layer based on the error function and then update the weights according to

$$W_{j,i}^l(t+1) = W(t)_{j,i}^l - \eta\frac{\partial \mathcal{E}}{\partial W_{j,i}^l(t)} = W(t)_{j,i}^l - \eta\frac{1}{p}\sum_{k=1}^{p}\frac{\partial E}{\partial W_{j,i}^l(t)}. \tag{2.42}$$

For $l = L$, the chain rule is used to find the gradients:

$$\frac{\partial E}{\partial W_{j,i}^L(t)} = \frac{\partial E}{\partial Y_j^L}\frac{\partial Y_j^L}{\partial I_j^L}\frac{\partial Y_j^L}{\partial W_{j,i}^L(t)}. \tag{2.43}$$

Note that E, Y_j^L and I_j^L depend on both the instance k and the iteration t, since the weights update every iteration. The first factor is $(y_j(k) - Y_j^L)$ according to Equation (2.40), the second factor is $g'(I_j^L)$ according to (2.38) and the last factor is Y_i^{L-1} according to (2.39). In other words:

$$\frac{\partial E}{\partial W_{j,i}^L(t)} = Y_i^{L-1}\delta_j^L, \tag{2.44}$$

where $\delta_j^L = \frac{\partial E}{\partial I_j^L} = (y_j(k) - Y_j^L)g'(I_j^L)$.
The previous layers' gradient is calculated recursively by assuming that $\frac{\partial E}{\partial I_j^L(t)} = \delta_j^{l+1}(k)$ is known and using the chain rule again analogously to the calculation of the final layer:

$$\frac{\partial E}{\partial W_{j,i}^l(t)} = \frac{\partial E}{\partial Y_j^l}\frac{\partial Y_j^l}{\partial I_j^l}\frac{\partial Y_j^l}{\partial W_{j,i}^l}. \tag{2.45}$$

The latter two factors are the same as previously. The derivative of the first factor is obtained by using the chain rule again:

$$\frac{\partial E}{\partial Y_j^l} = \sum_{o=1}^{n_{l+1}} \frac{\partial E}{\partial I_o^{l+1}} \frac{\partial I_o^{l+1}}{\partial Y_j^l} = \sum_{o=1}^{n_{l+1}} \delta_o^{l+1} \frac{\partial \sum_{p=0}^{n_{l+1}} W_{p,j}^{l+1}(t)}{\partial Y_j^l} = \sum_{o=1}^{n_{l+1}} \delta_o^{l+1} W_{o,j}^{l+1}(t). \tag{2.46}$$

Thus, the gradient of the non-final layers are:

$$\frac{\partial E}{\partial W_{j,i}^l(t)} = Y_i^{l-1} \delta_j^l, \tag{2.47}$$

where $\delta_j^l = \sum_{o=1}^{n_{l+1}} \delta_o^{l+1} W_{o,j}^{l+1} g'(I_j^l)$.
By inserting this into Equation (2.42), the updates are then

$$W_{j,i}^l(t+1) = W(t)_{j,i}^l + \eta \delta_j^l Y_i^{l-1}. \tag{2.48}$$

One potential issue with backpropagation is that due to the heavy use of the chain rule, different values of g' are multiplied repeatedly. This runs the risk of either causing the gradients to explode or vanish if there are many layers, affecting the learning of the first few layers. An example of the gradient vanishing is the function $g(u) = \tanh(u)$, as $g'(u) = 1 - \tanh(u)^2 \in [0, 1]$, where the derivative is less than one unless $u = 0$ where it is equal to one. Thus, repeated multiplications of g' will lead to smaller and smaller values, meaning that the early layers barely learn anything. This problem is known as the *vanishing gradient problem* [24]. One possible solution to this is to use the ReLU activation function, as it can only have the derivative zero or one, depending on the sign of the signal.

While being able to predict a given output vector is nice, the question is how it can be used to determine which class to use. The answer is to let layer $L-1$ have as many nodes as the number of labels and let the final activation function be the softmax function [24]:

$$Y_j^L = Softmax(Y)_j = \frac{e^{Y_j^{L-1}}}{\sum_{i=1}^{n_{L-1}} e^{Y_j^{L-1}}}. \tag{2.49}$$

The reason why the softmax function should be the final layer is to let the output be an estimate of the probability of each label being chosen, as the sum of the nodes in the layer after the softmax function is one. To translate the labels of the training data into an output that the neural network can work with, the label is transformed into a "one-hot" vector where the entry corresponding to the chosen label is one and the rest of the entries are zero.

2.5.7 Grid search

Model tuning is an important task after developing a machine learning model. Each machine learning model works differently and has its own set of parameters and hyperparameters. The models can predict better results only when the right values are assigned to the parameters and hyperparameters for the problem being solved.

One such way to fine tune the models to find the better values for those parameters is by using a *grid search* method [25]. In this method, a set of parameters associated with the machine learning model is chosen and a list of values associated with each of the parameters is selected. An exhaustive search over all of the specified parameter values for an estimator is done in order to find the best values for a parameter which helps in making better predictions using that model. For example, if a machine learning model takes p hyperparameters for training, and each of the p hyperparameters get a list of three values each, 3^p combinations of the hyperparameters are tested and evaluated, and whatever combination is given the best performance according to the evaluation is used as the hyperparameter for the model.

2.5.8 Multi-label classification

While multi-class classification allows only one label to be given as the output for a piece of data, multi-label classification is a generalization of multi-class classification that allows a piece of data to have more than one label [35]. This can be represented by replacing the label y with a binary vector $\mathbf{y} \in \{0, 1\}^{|\mathcal{L}|}$, where each entry in the vector represents if a label in $\mathcal{L}$ is used for the data point or not.

In order to train a machine learning model, one can either transform the multi-label problem into one or more multi-class or binary classification problems and solve them with one of the algorithms above, or generalize the model to handle multi-label classification. Neural networks is an example that can already handle multi-label classification well, since normally the output is already a vector. The only difference between the multi-label method and the multi-class method described above is that the softmax layer for multi-class classification is replaced by a sigmoid layer to let the values of the output be between zero and one, representing the probability that the label is used. Another model that can be adapted is decision trees, where the Gini impurity for a label to be zero or one is calculated for each label and the cost is instead the average Gini impurity for each label.

There are four transformations of the problem that will be used in the thesis. The first is *binary regression* [35], where one assumes that the labels are independent and trains a model for each label to predict if the label is used or not for an observation and then combines the result for the final multi-label. The second is the *classification chain*, which also trains a model for each label, but instead of being trained independently on the observation, they are trained sequentially, with the prediction for label j using $\mathbf{x}$ and the prediction of the previous label as the training input. Next is the *label powerset*, which views the binary vector as a label set with 2^L different labels for each combination of labels in $\mathbf{y}$ and trains a machine learning model to predict the exact label set. Finally, a somewhat naive transformation is to let one of the labels of $\mathbf{y}$ be randomly selected and be considered the label of the input $\mathbf{x}$ which will then be solved as one multi-class problem with L labels.

2.5.9 Evaluation metrics

Once the model is trained, the quality of the model has to be evaluated to find out how well the model performs and generalises for previously unseen data. Some of the more common evaluation metrics [36] used in machine learning include *accuracy*, *precision*, *recall* and $\mathbf{F_1}$*–score*. Accuracy is the percentage of correct predictions made by a classification model. It can be defined as the ratio of number of correct predictions to the total number of predictions. It works well when there is a balanced data set where most classes have equally many observations, but might not work well if the distribution of classes is imbalanced. For example, if there are two classes, where one class is used 99% of the time, a model that only predicts that class has an accuracy of 99%, but it wouldn't be a good model if the purpose was to detect the minority class.

A *confusion matrix* is performance measurement for classification models which summarizes the prediction results of a binary classifier. It gives information about four possible prediction results, namely *true positive* (TP), *true negative* (TN), *false positive* (FP) and *false negative* (FN). True positives and true negatives are cases where the classifier correctly predicts the positive class and negative class, respectively. False positives and false negatives are cases where the classifier incorrectly predicts the positive class and negative class, respectively.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.50}$$

Using the values from the confusion matrix, other metrics such as precision and recall are calculated.

Precision is the measure of the model's performance to classify positive instances. It is the ratio of positive class predictions which really belongs to the positive class:

$$Precision = \frac{TP}{TP + FP}. \tag{2.51}$$

Recall is the measure of the model's performance to correctly identify positive instances. It is the ratio of instances in the positive class which are correctly predicted:

$$Recall = \frac{TP}{TP + FN}. \tag{2.52}$$

The so-called *F-score* is an evaluation metric which makes use of weighted average of precision and recall in binary classification systems. It uses both false positives and false negatives while evaluating the model and usually works better than accuracy in case of uneven class distribution:

$$F\beta - score = \frac{(1 + \beta^2)Precision \cdot Recall}{\beta^2 Precision + Recall}. \tag{2.53}$$

The β parameter used while calculating the F-score controls the extent which precision is weighed into the F-score. When $\beta = 1$, the score is known as the F_1-score, and is simply the harmonic mean of the precision and recall:

$$F_1 - Score = \frac{2(Precision \cdot Recall)}{Precision + Recall}. \tag{2.54}$$

While these metrics by themselves only work for binary classifiers, they can be generalized to both multi-class and multi-label classifiers [25]. This is done for a multi-class classifier by first considering for each class a "one vs rest" classification and calculating the number of true positives, false positives, etc. for each class. For a multi-label classifier this is done by considering each label a binary classifier like the binary regression and similarly calculating the true positives, etc. Afterwards, one can either use a *micro-average* by summing the prediction results over each class and calculating the metric from this sum, use a *macro-average* by calculating the metric for each class and then averaging over it or use a weighted macro-average by using a weighted average of the metric for each class where the weight comes from the number of observations with the given label. For multi-label classification in particular, one can instead use the *Hamming loss* [35], which is the number of mistakes made for each label in all instances divided by the number of labels times the number of instances. Unlike the previous metrics, where the quality is best when it is close to one, the Hamming Loss should be as close to zero as possible.

Cross validation is a method which is used to estimate how well a model generalizes to new data. In this method, a small part of the data set is not included in training the model which is then used to evaluate the performance. One of the main method of performing cross validation is *k–fold cross validation.* In k-fold cross validation, the training data set is divided into k parts and the model is trained and evaluated k times. The models are trained in such a way that each of the k-folds will be used as a test set while all other folds will be collectively used as a training set.

A *baseline model* is a model that can be used as a baseline for evaluating models [24]. These may be simple machine learning models or naive models with no intelligence. One such naive predictor for performing the multi-class prediction is to randomly guess a strategy label. The accuracy of such a naive predictor when using n classes is $\frac{1}{n}$. Another example would be a model that would always pick one label all the time. The main purpose of baseline models is to give some lower bound on the evaluation metric for the machine learning model that it should strive to improve.

3
Methods

This chapter presents the methods used within this thesis project. This includes the choices and motivations behind the proposed strategies, how the data is collected, how the collected data is then used in the machine learning algorithms and how the implemented algorithms are evaluated.

3.1 Selecting strategies

As mentioned in Section 1.3, instead of deciding between warm start and cold start, predefined strategies were proposed which the machine learning algorithms will learn to predict from a set of input features/structure of the problem instance. As a quick reminder, the decision was whether the initial dual solution of the subgradient method for the next iteration of column generation should be the last dual solution obtained in the previous iteration of column generation (warm start) or the origin (cold start). Some of the problem instances at Jeppesen Systems took around nine hours to run with the current heuristic and some of the proposed strategies took a lot longer than the current heuristic. For the cases where strategies were parameterized, a single parameter was chosen with a single reasonable value in order to minimize the number of strategies. In this thesis, nine such different strategies were selected including the current heuristic in use at Jeppesen Systems. For more clarity, Table 3.1 shows the list of the nine strategies which have been proposed and experimented within this thesis. As mentioned earlier, different strategies can be formulated with different variations, but the list of strategies from Table 3.1 was chosen as they potentially cover various different paths.

Index	**Strategy**
1	Current heuristic.
2	Only use warm start.
3	Only use cold start.
4	Start with warm start and switch starts every 10th iteration.
5	Cold start every 20th iteration, else warm start.
6	Cold start first 10 iterations, then switch every 10th iteration.
7	Cold start first 15 iterations, then use the current heuristic.
8	Warm start first 15 iterations, then use the current heuristic.
9	Dynamic strategy based on the lower bound.

Table 3.1: Table of the nine strategies investigated within the thesis project.

The current heuristic first performs one subgradient step for both warm start and cold start to see what objective value is obtained for the first step and selects the start with the highest objective value. Two other simple strategies that were proposed uses only warm start for all iterations and only cold start for all iterations, respectively. The next strategy proposed was to move in a zig-zag pattern, starting with one start and switching which start is used every couple of iterations. It was decided that this strategy would start with warm start and switch every tenth iterations. Next, a strategy which mainly used one start but on occasion used the other start was proposed. The decision was to use cold start every 20th iterations and otherwise use warm start. An example of how these look in practice can be found in Table 3.2.

Iteration	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**	**Strategy 5**
1	ws	ws	cs	ws	ws
2	cs	ws	cs	ws	ws
3	cs	ws	cs	ws	ws
4	cs	ws	cs	ws	ws
5	cs	ws	cs	ws	ws
6	cs	ws	cs	ws	ws
7	ws	ws	cs	ws	ws
8	ws	ws	cs	ws	ws
9	ws	ws	cs	ws	ws
10	ws	ws	cs	ws	ws
11	ws	ws	cs	cs	ws
12	ws	ws	cs	cs	ws
13	ws	ws	cs	cs	ws
14	ws	ws	cs	cs	ws
15	ws	ws	cs	cs	ws
16	ws	ws	cs	cs	ws
17	ws	ws	cs	cs	ws
18	ws	ws	cs	cs	ws
19	ws	ws	cs	cs	ws
20	ws	ws	cs	cs	cs

Table 3.2: The first twenty choices of warm start or cold start for the first five strategies for one of the problem instances.

The next three strategies was proposed based on an observation within the current heuristic where the cold start was initially used a lot before using warm start. An example can be seen in the first strategy in Figure 3.2 where after the first iteration, cold start is performed for the next five iterations, followed by warm start. The idea was to replicate part of the choices in the heuristic without having to solve the two subgradient steps for each iteration. If it worked, it would help in saving time and computing power. The proposed strategies were to use cold start for the first ten iterations followed by using cold start every 10th iteration and otherwise use warm start. Another strategy used cold start for the first 15 iterations and switches back to the current heuristic afterwards. Similar to the previous strategy, another strategy used warm start for the first 15 iterations and then switches back to the current heuristic.

The final strategy proposed was based on lower bound of the problem instance that is calculated by the framework before applying the subgradient optimization. The proposed idea was a variation of the current heuristic where the objective of warm and cold start for the first subgradient step is calculated and then the warm start objective is compared to the lower bound. If it is larger, the start with the lowest objective is selected, while if the objective is lower, the start with the highest objective is selected. These four strategies can be seen in Table 3.3.

Iteration	**Strategy 6**	**Strategy 7**	**Strategy 8**	**Strategy 9**
1	cs	cs	ws	ws
2	cs	cs	ws	cs
3	cs	cs	ws	cs
4	cs	cs	ws	cs
5	cs	cs	ws	cs
6	cs	cs	ws	cs
7	cs	cs	ws	ws
8	cs	cs	ws	ws
9	cs	cs	ws	ws
10	cs	cs	ws	ws
11	ws	cs	ws	ws
12	ws	cs	ws	ws
13	ws	cs	ws	ws
14	ws	cs	ws	ws
15	ws	cs	ws	ws
16	ws	ws	ws	ws
17	ws	ws	ws	ws
18	ws	ws	ws	ws
19	ws	ws	ws	ws
20	cs	ws	ws	ws

Table 3.3: The first twenty choices of warm start or cold start for the last four strategies for one of the problem instances.

3.2 Collecting data

After defining the nine strategies in Table 3.1, each of those strategies were implemented in the column generation framework so that they can be used to solve the tail assignment problem. The column generation framework code was modified to extract parts of a given problem instance as possible input features, along with the name of the problem instance in a separate XML file. Similarly, the output features which are the results for a given problem instance when using one of the nine strategies were collected in another XML file, along with the problem instance name and strategy used to solve that problem instance. The name of the problem instance was then used to match the input features with the same problem instances output features for each strategy. The problem instances that Jeppesen Systems had were then ran for each of the nine strategies to collect the data. For each problem instance, the input features were collected from one of the runs, as the input features are only dependent on the problem instances, while the output features were collected for each strategy.

Feature number	**Input Features**
1	Number of flight legs.
2	Number of aircraft.
3	Number of global constraints.
4	Number of soft cumulative rules.
5	Number of hard cumulative rules.
6	Number of unassigned tasks.
7	Number of tasks preassigned to flights.
8	Number of connections between tasks.
9	Number of starting tasks.
10	Number of end tasks.
11	Mean of the standard deviation of connection cost for each aircraft.
12	Number of connections of duration - 0–10 minutes.
13	Number of connections of duration - 10–20 minutes.
14	Number of connections of duration - 20–30 minutes.
15	Number of connections of duration - 30–45 minutes.
16	Number of connections of duration - 45–60 minutes.
17	Number of connections of duration - 60–90 minutes.
18	Number of connections of duration - 90–120 minutes.
19	Number of connections of duration - 2–3 hours.
20	Number of connections of duration - 3–4 hours.
21	Number of connections of duration - 4–6 hours.
22	Number of connections of duration - 6–8 hours.
23	Number of connections of duration - 8–12 hours.
24	Number of connections of duration - 12–16 hours.
25	Number of connections of duration - 16–24 hours.
26	Number of connections of duration - >24 hours.
27	The time period that the aircraft are optimized during.
28	Average length of flights.

Table 3.4: Input features extracted for each problem instance.

Feature number	Output Features
1	Number of unassigned flight legs.
2	Total running time.
3	Objective value.

Table 3.5: Output features extracted for each strategy and each problem instance.

The input and output features that were relevant to this thesis that were collected for all the problem instances for each of the strategy are shown in Tables 3.4 and 3.5 respectively. To explain what each of the features mean, the number of legs and vehicles are the number of flight legs and aircraft used in the problem instance, respectively. *Global constraints* [1] are constraints that are defined over multiple aircraft routes. For example, if one wanted a limit on the total amount of exhaust fumes of the aircraft to be more environmentally friendly, the exhaust fumes would be a global constraint. *Cumulative rules* are constraints on resources of a single aircraft which limits the use of the resource. An example of this would be a limit on the flying time or the number of landings an aircraft can do in the period optimized for. These can either be hard rules, which are not allowed to be broken and thus limit the set of valid routes, and soft rules, which can be broken but incur a penalty for the violation, which affect the cost of certain routes. A task is either a flight or some maintenance on an aircraft, where some are preassigned. Start tasks and end tasks are tasks which an aircraft can start and end a flight route with, respectively. Another way to look at the tasks is that they correspond to the nodes of a graph. The connections can be explained as arcs in the graph where the tasks are nodes. There is also a cost for each aircraft and a time associated with each connection. The time is the time an aircraft spends on the ground between the two tasks, and the cost is the cost for the corresponding flight or maintenance in Equation 2.14. In order to combine all the costs into a single feature, each aircraft had the standard deviation of the costs of all connections calculated, and then the mean over these standard deviations were calculated to provide the mean standard deviation mentioned in Table 3.4.

While collecting the data, there were two major challenges that were encountered, namely the number of problem instances available at Jeppesen Systems based on real life tail assignment problems that data could be collected from and the amount of time required to collect the data for those available instances. The number of available tail assignment problem instances which used subgradient methods were around 245 which is very small from a machine learning perspective. Another major issue was the time required for running the 245 problem instances with the nine different proposed strategies using the Tail Assignment optimizer. With the available tests being large-scale instances with up to 200 000 flights and 300 vehicles, as well as the preparations done by Jeppesen Systems' optimizer before running the column generation also taking time, most of the problem instances took a long time. In addition, while some of the strategies may work better than the current heuristic, others might be worse than the current heuristic. In any case, solving the 245 instances using the nine strategies took several weeks, with some instances taking around three days for non-optimal strategies.

If there was the risk of having too few problem instances, one could augment the input features of the given instances to create more problem instances. However, creating flight schedules that can satisfy a constraints of the aircraft routes while also being balanced is itself a non-trivial task, meaning that augmenting the input features could have easily make the new problems infeasible. This could have resulted in the machine learning model being trained to learn the patterns of an infeasible problem instance rather than improving the solution quality of the algorithm, meaning that this was not considered. In order to address the long running time, the 120 instances which takes less than ten minutes to run in total with the current heuristic were collected first in order to create a reasonable data set to develop models before having all the data. Since the time taken for collecting the problem instances was quite high, the data was collected only for the instances which takes at most nine hours to run the optimizer with the current heuristic. The distribution of the total solving time with the current heuristic for the instances collected are shown in Table 3.6.

Index	**Computation time of instances**	**Number of instances**
1	Less than 10 minutes.	120
2	10-60 minutes.	39
3	1-2 hours.	23
4	2-3 hours.	30
5	3-4 hours.	9
6	4-5 hours.	6
7	5-6 hours.	7
8	6-7 hours.	5
9	7-9 hours.	6

Table 3.6: Distribution of computation time of the problem instances using the current heuristic.

3.3 Labeling data

As mentioned in the previous section, the input features of each problem instance and output features for each instance and strategy pair were collected. In order to label the data set with which strategy works the best for each problem instance, the three output features, total running time, objective value and number of unassigned legs were used along with the combination of all three features. The four different ways in which the output features were analyzed using various methods to apply a label. These methods will hereby be called *labellers* and are listed below:

- Selecting strategies which has the minimum objective value cost.
- Selecting strategies which has the minimum total running time.
- Selecting strategies which has the minimum number of unassigned legs.

- Selecting strategies which uses a combination of all three output features by prioritizing unassigned legs, followed by running time and finally objective value.

In order to develop good machine learning models, the training data set should have enough data for each of the classes. The distribution of the problem instances across the different strategies chosen were analyzed to see if the training data set is good for developing machine learning models. After analyzing the output features for some of the problem instances, it was found that there were multiple strategies which were considered good for the same problem. This could be because when a problem instance only needs a few iterations, some strategies may lead to the same decisions. For example, using the strategy of using only warm start leads to the same decisions as using cold start every 20th iteration if the instance requires less than 20 iterations.

Due to this, the problem of selecting a strategy would be considered as a multi-label classification problem, where each label represents each of the strategies. To create a set with only one label per problem instance for multi-class classification, the strategies that were suggested for a problem instance was randomly selected several times to create multiple multi class data sets. The Gini impurity, which was described in Section 2.5.6.3 was then calculated for each of the data sets, and the one with the highest Gini score was chosen as the final data set. This was done in order to balance the distribution of data across the different classes, as having the classes balanced is more likely to give good results. The multi-label vector was also kept for each of the labellers as well to use for multi-label classification.

3.4 Creating the data set

When creating the data set, all the input features and output features mentioned in Tables 3.4 and 3.5 were extracted from the XML files and placed in separate csv files. These csv files were then used to create separate data frames for the input and output features, respectively.

In the input data frame, the soft and hard cumulative rules were divided by the total number of cumulative rules, if there were any, in order to instead show what percentage of the rules were soft and hard, respectively. Similarly, the number of connections of each duration were divided by the total number of connections to have them written as a percentage as well. This was done to make both sets of features independent on the size of the problem instance. This especially holds for the connections, as the number of connections is already a feature, meaning that having the number of connections of each duration contains redundant information. The active period was also converted into a duration in minutes, as the active period was originally written as the start of the period and the end of the period. The main changes to the features in Table 3.4 are then that features 4–5 and 12–26 had their total number replaced with a percentage while the time period optimized during is renamed to active period time:

Similarly, the output data frame was analyzed as mentioned in Section 3.3 to get the multi-class output labels y and the multi-label labels $\mathbf{y}$. The multi-label labels were also analyzed to see as how often each strategy was recommended as well as how many best strategies each problem instance contained. The input data frame and the labels were then combined based on the instance name to create a data set of 245 problem instances. This data set was then randomly split into a training set and a test set where 80% of the data was in the training set and 20% was in the testing set [24]. Since the data set only consisted of 245 observations, which is very small for developing a machine learning model, the decision was made to not use a validation set for hyperparameters in order to have more data for testing and training.

Finally, in order to let every feature be of equal importance, the features of both data sets were standardized based on the mean and standard deviation of the training set. Similarly, two new data sets were also created in order to check the effects of PCA and feature importance and how efficient dimension reduction can be. The first new data set contained only the data of the five most important features according to a random forest classifier run beforehand on the training set. The method for this was previously discussed in Section 2.5.6.4 For the multi-labelled data, the trees in the random forest uses the average Gini impurity for each label to split the tree. The second data set has the data transformed into the first five principal components of the training data. The two new data sets are expected to provide a lower accuracy due to some information being lost, but will generally be faster and may even make some of the information in the input more clear.

3.5 Implementing the models

The main focus of this thesis is to investigate how machine learning models can help to improve the solution quality of the column generation framework and not to implement or develop new machine learning models, meaning that the thesis will mainly use pre-implemented models. The *sklearn* library in Python which contains efficient tools for machine learning and statistical modelling, along with *tensorflow* which contains tools for neural networks. With these libraries, the five models in Table 3.7 were chosen and analyzed in this thesis. Each of the models were trained for each of the different labellings and data sets by using the *GridSearchCV* method from the *sklearn* library to fine tune the hyperparameters on the training set before fitting the model on the testing set. This method uses a combination of Grid Search and five-fold cross-validation to determine the best hyperparameters based on a given metric, which for this thesis was the F_1-score, in order to balance precision and recall. Unfortunately, this method does not work on the neural network, so instead, all the hyperparameter combinations were tested on the training and test data, and the combination that gave the best F_1-score was the one that was used. The parameters and their corresponding values used in the grid search for the various models are listed in Table 3.8.

For the multi-label classification problems, $\mathbf{y}$ was transformed into multi-class problems through binary relevance, classification chain and label powerset before training using *GridSearchCV*. The exception to this was the neural network, which was instead trained directly without any transformation, as that already has a clear multi-label implementation. Besides the F_1-score, the accuracy was calculated for the multi-class labellings, and the Hamming loss was calculated for the multi-label labellings, in order to see how often the correct label is picked and how often a label is correctly chosen as good or not, respectively.

In order to have a benchmark to compare the other models to, two baseline models was employed for both multi-class and multi-label classification. One randomly selects among the nine different labels for the multi-class classification uniformly, and uniformly chooses among each of the possible multi-label vectors for multi-label classification. The other baseline instead picks only the current heuristic for multi-class classification and picks the multi-label vector of only ones for multi-label classification. Both the baseline models were evaluated with the entire data set as the test set in order to have a good benchmark.

Index	**Machine Learning Model**
1	Decision tree
2	Random forest
3	Naive Bayes
4	Support Vector Machine
5	Neural network

Table 3.7: The machine learning models implemented and analyzed in the thesis.

Model	**Parameter**	**Values tested**
Decision tree (Section 2.5.6.3)	Cost function (2.33)	[Gini impurity (2.34), entropy (2.35)]
	Max depth	[3,4,5,6,7, No limit]
Random forest (Section 2.5.6.4)	Cost function	[Gini impurity, entropy]
	Max depth	[4,5,6]
	Number of trees	[50,100,150,200,250]
SVM (Section 2.5.6.2)	C (2.30)	[1, 10, 100]
	Kernel (2.31)	[Linear, Polynomial, RBF, Sigmoid]
Neural network (Section 2.5.6.5)	Number of hidden layers	[1,2,3]
	Number of neurons	[16,32,64]
	Optimizer	[Adam, SGD]
	Activation function	[ReLU (2.36), tanh]
	Number of epochs	[100,150,200]

Table 3.8: Machine learning models along with different parameters tested in grid search and references to the relevant sections and equations.

,

3.6 Predicting with 120 problem instances

The machine learning model is as good as the data used in training the model. It depends on both the quantity as well as the quality of the data which is used to train the machine learning model. Since this thesis mainly revolves around investigating how machine learning can be used to improve the solution quality of a tail assignment problem which basically has a major drawback of less number of data points. In order to validate the use of such a small data set to do the machine learning, an idea of training the model with problem instances which takes less than 10 minutes is used to predict the strategies for instances which takes hours to run. The main idea behind this approach was to test whether the model trained with instances which usually takes less than 10 minutes to run could help in predicting the right strategy for larger instances.

A training data set comprising of 120 problem instances which takes less than 10 minutes to run was used to train the machine learning models. Similarly, the instances which usually take more hours to run were selected as the testing data set. The instances in the testing data set were selected as mentioned in Table 3.9. Then, the results were compared with the baseline models.

Index	**Computation time**	**Number of instances**
1	1–2 hours	23
2	3–4 hours	9
3	5–6 hours	7

Table 3.9: Computation time using the current heuristic for the 39 problem instances used in the testing data set.

4 Results and discussion

This chapter presents the results within this thesis project. This includes how the data is distributed across the different classes and the results from the various machine learning models with the data set and variations of it.

As mentioned in Section 3.3, four different labellers, namely the running time labeller, the cost labeller, the unassigned legs labeller and the combination labeller that uses all 3 output features have been used to determine the best multi-label label for each problem instance in the data set which were then turned into a multi-class label using a random choice between the candidate strategies for some iterations to get a balanced data set. The five different machine learning models were then trained on the input features paired with each of the four labellers for both multi-class and multi-label classification on three different sets of the features, namely using all 28 input features, using the five first principal components according to a PCA, and using the top five most important features according to a random forest (represented as feature selection, or FS for short). This section will only show the results of the best models and not for every model. More detailed results can be found in Appendix A.

4.1 Data distribution

Figures A.1 and A.2 represent the data distributions of all the input features as histograms. It can be seen clearly from the images that the data distribution of multiple features like the number of legs, the global constraints, the pre-assigned tasks, the mean standard deviation of connection costs, and the average length of flights contains data which can be considered as outliers. Even though they have huge deviations, such values are not treated as outliers because they might correspond to values which are derived from large-scale tail assignment problem instances, which are the more important problem instances to determine good strategies for.

4.1.1 Feature correlation

When having data with a lot of features, some of the features may correlate with each other and thus contain very similar information. To examine this, the correlation of all pairs of features according to the data set were computed with the results found in Figure 4.1

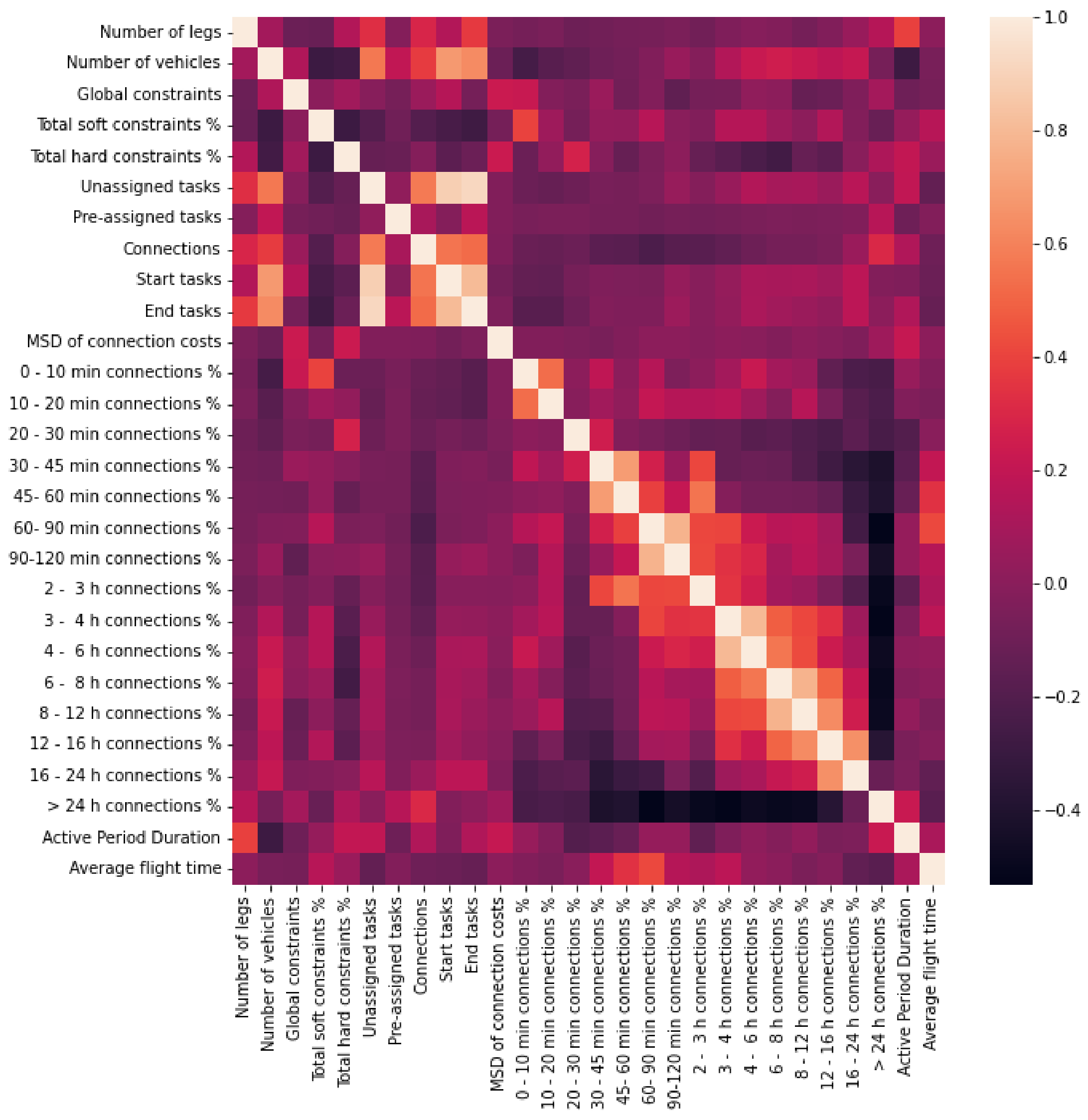


Figure 4.1: Heat map of the correlation between the various features.

Figure 4.1 suggests that some of the input features are correlated with each other. The number of flight legs had some correlation with both the unassigned tasks and the active period of the flights, which makes sense as the flights make up a large part of the tasks and more flights will eventually increase the period the flights end up on. The active period is also negatively correlated with the number of aircraft, which is likely due to more aircraft being able to fit more flights in the same period of time. The number of aircraft also affects the number of tasks and connections along with having a large number of connections being 3–12 hours. It was also found that all the task-based features were heavily correlated with each other.

The number of global constraints affected the percentage of soft cumulative constraints, likely due to soft constraints being more common than hard constraints. Both the total number of constraints and the soft constraint percentage were correlated with the percent of the connections take under ten minutes, which might be how soft constraints affect the connections. A similar thing can be seen with the hard constraints, which was correlated with the percent of 20–30 minute connections.

Finally, the average flight time was correlated with the length active period, and interestingly enough the percentage of connections between 60 and 90 minutes and the variance of the connections costs. The former could be expected since a longer period of flights could lead to some longer flights being able to be put in the period, but the other ones are a bit more surprising.

4.1.2 Principal components

After performing a PCA on the training set of the data that was used for the rest of the machine learning, the first five principal components can be found in Figure 4.2. The signs of the most influencing features of the first five principal components is found in Table 4.1.

Table 4.1 shows that the first principal component ended up having large positive values for the number of vehicles, most of the task-based features and percentage of connections between three and 24 hours. As most of the task features are correlated with the number of vehicles, it could be that this component represents the overall size of the problem instance. The second component is very negative towards the percentage of connections between one and twelve hours while it was more positive towards the amount of percentage over 24 hours, this could be perhaps be interpreted as a contrast of the lengths of the connections.

The third principal component seems to contrast the number of soft constraints and longer connections with the number of tasks and short connections. The fourth component is along the number of flights and percent of cumulative rules along with 90–120 minute connections and active period duration. Since the active period duration is correlated with the number of legs, this component is mainly alongside the number of legs. The fifth component have large values for of the number of global constraints and really short connections, which have some correlation with the global constraints, meaning that the global constraints might be the main contributor to the principal component.

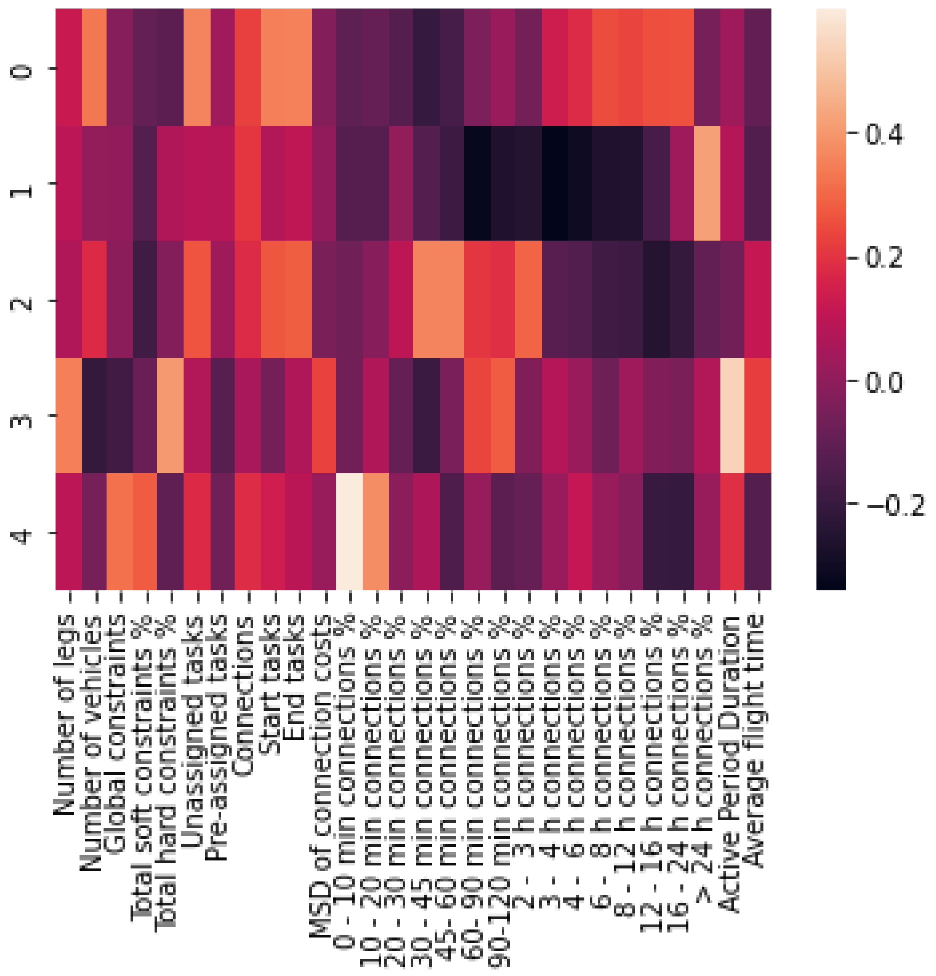


Figure 4.2: Heat map of the values of the first five principal components vs the original features for the training set.

Feature	α_1	α_2	α_3	α_4	α_5
Number of legs	(+)			+	
Number of vehicles	+		(+)	(-)	
Global constraints				(-)	+
Soft cumulative rules %	(-)	(-)	-		(+)
Hard cumulative rules %	(-)			+	
Unassigned tasks	+		+		(+)
Pre-assigned tasks					
Connections	+	(+)	+		(+)
Start tasks	+		+		
End tasks	+		+		
MSD of connection costs				(+)	
0–10 minute connections %	(-)	(-)			+
10–20 minute connections %	(-)	(-)			+
20–30 minute connections %	(-)		(+)		
30–45 minute connections %	-	(-)	+	(-)	
45–60 minute connections %	(-)	(-)	+		
60–90 minute connections %		-	+	(+)	
90–120 minute connections %		-	+	+	
2–3 hour connections %		-	+		
3–4 hour connections %	(+)	-	(-)		
4–6 hour connections %	(+)	-	(-)		
6–8 hour connections %	+	-	-		
8–12 hour connections %	+	-	-		
12–16 hour connections %	+	(-)	-		(-)
16–24 hour connections %	+		-		(-)
>24 hour connections %		+	(+)		
Active period duration				+	(+)
Average length of flights	(-)	(-)	(+)	(+)	

Table 4.1: Table of the signs of the first five principal components $\{\alpha\}_{i=1}^{5}$. A feature having + or - means that the absolute value is over half the maximum absolute value of the component and is positive or negative, respectively, while a feature has (+) or (-) if the absolute value is between 25% and 50% of the maximum and is positive or negative, respectively.

4.1.3 Label distribution

As the number of problem instances used for training the machine learning models were only 245 and there are nine different strategies, the problem instances distributed across each label plays a crucial role in machine learning. The number of labels that exist in each problem instance for each of the labellers found in Section 3.3 can be seen in Table 4.2

# labels	Running time	Cost	Unassigned legs	All 3 features
1	195	126	17	201
2	14	21	9	12
3	9	15	20	5
4	1	11	16	2
5	6	5	5	5
6	5	8	11	5
7	9	3	8	9
8	1	3	17	1
9	5	53	142	5

Table 4.2: Number of possible labels for each problem instance in the data set based on four different ways of labelling.

Table 4.2 shows that the running time labeller and the so-called combination labeller that uses all three output features has most of the problem instances with exactly one single label, with a few instances having more than one label and has only five instances having all nine labels. By a simple observation it can be seen that the combination labeller has fewer problem instances with more than one label for every number of labels except the last one. This is likely due to the running time playing a role in the combination labeller, meaning that it is more specific about what labels to give each problem instance compared to the running time labeller. The other two labellers have more labels per problem instance, with the cost labeller having around 70 fewer instances than them with only one label and 53 instances have all nine labels. The unassigned legs labeller gives all labels for 142 observations and only 17 observations has exactly one label. Due to this, the data sets made from running time labeller and combination labeller using all three features should be given more priority when evaluating the models.

While labelling the multi-class classification data set, the number of possible labels for each problem instance gives an indication of how the model will perform. Whenever the problem instances have exactly one label, the model tends to be deterministic where it will be evaluated fairly based on the true positive values. But whenever the instance has more than one label, there is a high chance that the model predicts any one of the labels and it is highly possible that the model is evaluated without considering this possibility. Due to this nature of the data set used for multi-class classification, the trained models tend to have a smaller accuracy where the model is unfairly penalised in spite of making a right prediction in most of the cases.

The number of times each strategy is used as a label for each labeller can be found in Table 4.3. Refer back to Table 3.1 for each of the strategies.

Strategy	**Running Time**	**Cost**	**Unassigned legs**	**All 3 features**
1	77	108	213	75
2	55	109	211	49
3	56	73	173	50
4	40	84	176	37
5	50	97	199	56
6	28	84	171	28
7	30	99	193	33
8	60	128	217	61
9	34	70	167	30

Table 4.3: Data distribution of 245 problem instances across nine classes for each of the four labellers.

Table 4.3 suggests that the distribution of the running time labeller and combination labeller are quite similar to each other. They are equally distributed compared to the other labellers. For both of these labellers, the current heuristic is the most frequent label followed by starting with cold start and then using the current heuristic (Strategy 8). On the other hand, the strategy that starts with cold start and then mostly uses warm start (Strategy 6) has the fewest problem instances. With each label being assigned to less than a third of the total problem instances, any model will be likelier to not predict a label that should be given.

The data obtained from the unassigned legs labeller has notably more problem instances that recommends all strategies with 142 problem instances. In the case of multi-label classification, due to the large number of labels for each problem instance, models trained on this data set will likely have an easier time deciding if an instance has a label, but a harder time deciding if an instance should not have a label in the label vector. Since the unassigned legs labeller gives labels to problem instances that give the minimum number of unassigned legs regardless of the running time, it might easily assign labels to instances that run for a really long time when there are strategies that gives a similar solution faster. The same can be said to a lesser extent for the data using cost as the labeller as it has 53 observations with all nine labels. However, the overall number of observations that recommends each strategy is around half that of the unassigned legs labeller. The overall number points towards the cost labeller considering a strategy good more often than the running time or combination labeller, but not as often as the unassigned legs labeller.

In order to understand the proper distribution of the strategies before randomly selecting a strategy from the possible candidates to create a multi-class classifictaion data set, the number of times a strategy is uniquely identified as the best strategy can be found in Table 4.4.

Strategy	Running Time	Cost	Unassigned legs	All 3 features
1	48	13	0	49
2	19	10	0	16
3	26	13	1	21
4	17	9	1	14
5	20	24	9	31
6	9	6	1	9
7	11	17	5	16
8	30	24	0	33
9	15	10	0	12

Table 4.4: Number of times a strategy is the only best label for the 245 observations.

Table 4.4 suggests that the running time labeller and the combination labeller has more problem instances listed than the other two labellers. This table also suggests that for these two labellers, the current heuristic (Strategy 1) is in general a good choice, followed by starting with cold start and then using the heuristic (Strategy 8). The least used strategy for both is the strategy to initially use cold start and then switch to mostly warm start (Strategy 6). Since the unassigned legs labeller had very few observations with exactly one label, there isn't a lot that can be gained from that column except that most of the strategies will be picked randomly when creating the multi-class labels. On the other hand, the cost labeller has a much smaller number of problem instances where the current heuristic, using only warm start or using only cold start are the best strategies compared to the running time labeller, while the other strategies has approximately the same number of problem instances where they are the best for.

Table 4.5 represents the final distribution of data points for the multi-class classification problem with the nine strategies where one of the labels given by the labeller is randomly selected several times. The most balanced labelling was then picked as the final labelling.

Strategy	Running Time	Cost	Unassigned legs	All 3 features
1	57	32	30	55
2	27	30	30	25
3	33	25	25	28
4	20	22	30	18
5	31	30	39	35
6	12	16	16	12
7	13	29	27	18
8	34	43	30	39
9	18	18	18	15

Table 4.5: Data Distribution of 245 problem instances across the nine classes after selecting a random label from the candidate labels.

A quick look shows that for all labellers, strategy six is the least picked label for a problem instance with slightly more than 5% for the cost and unassigned legs labellers, and slightly less than 5% for the other two labellers. The running time and combination labellers have over 20% of the observations which has the current heuristic as the label, which makes the data set unbalanced. On the other hand, the other two labellers have almost equal distribution of problem instances across all the nine strategies. The imbalanced data set might affect the results of the machine learning models later.

4.1.4 Feature importance

In order to see which features the models would consider important, a random forest classifier was trained on the data set using the four different multi-class labellers to find which features minimized the Gini impurity the most. The results for each labeller are found in Figures A.3 – A.6.

It can be seen that for all of the figures, the mean standard deviation of the connection costs were the most important for all of the labellers. This could be because this is one of the few features that describes the mathematical formulation of the tail assignment problem beyond just the dimension of the problem instance. All of the labellers also either had the active period duration as an important feature, or the number of legs, which is correlated with the period duration. There were also connection percentages in all the labellers, aside from the labeller that used all the output features, as well as some feature related to the number of tasks like the connections, unassigned tasks or end tasks in all labellers except the cost labeller.

However, it should be noted that the five most important features only make up about 25% of the weight in the random forest classifier for all the labellers. Since the five most important features make up $\frac{5}{28} \approx 18\%$ of the total features, it points towards most of the features being similar in importance. This theory is supported by the second to fifth feature having almost equal importance, with only the most important feature deviating from them for all labellers except the combination labeller where the most important feature is also similar to the other four.

As for the features of each labeller, certain features seem to be related to what is used as the label. The running time labeller has the number of connections and the active period duration among the important features, both features that would increase the complexity of the problem, and thus the running time of the column generation framework. For the cost labeller, a lot of the features revolve around shorter connections. The unassigned legs labeller has the number of legs as a feature, along with the unassigned tasks, which are correlated to the number of legs. The combination labeller seems to have a combination of the features from the other three labellers, with the active period duration from the cost and running time, connections and average flight time from the running time, and while the end tasks is not in any of the others, it is correlated with the unassigned tasks from the unassigned legs labeller. The reason why most of the features are also found in the running time labellers important features is likely because as shown in Tables 4.2 – 4.3, the two labellers are very similar to each other.

4.2 Hyperparameters

Before showing the optimal hyperparameters for the models, an example of the importance of good hyperparameters is found in Figure 4.3, which shows the different values which are assigned for the five different parameters for one of the neural network models.

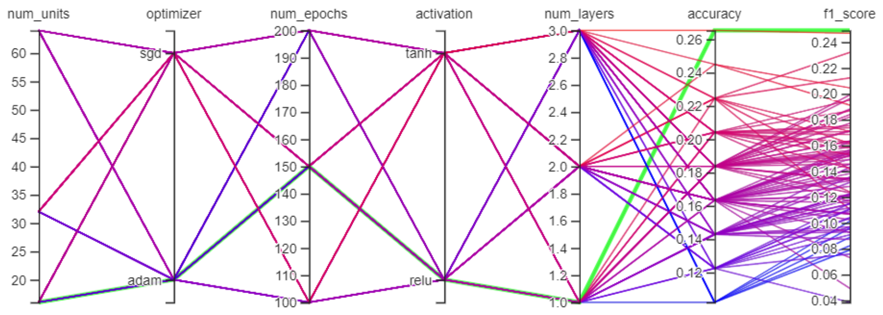


Figure 4.3: TensorBoard showing the accuracy and F_1-score for various neural network parameters for multi-class classification using running time labeller and all features.

The last two columns in that image gives the information about the accuracy score and the F_1-score of the neural network models when they are trained using the different parameter values as shown in the first five columns in that image for a neural network. From the figure, it is clear that different choices of the hyperparameters give different F_1-score, which is why grid search is used to find the best hyperparameters among several instead of just using one set of hyperparameters.

The best hyperparameter values for each of the machine learning models using the four different labellers can be found in Tables A.1 – A.8. By looking at the tables it can be seen that when it comes to creating the models, it seems that for decision trees and random forests, most of the decision trees have a depth of six or seven, while the random forest uses a depth of five or six. This suggests that deeper trees being better for these two models. Another thing to note is the cost function, which is the function used to split the trees that is suggested for the decision trees and random forests uses both entropy and Gini impurity for about the same number of models. It could mean that neither of the two cost functions work best for every labeller so both need to be considered when constructing the classifiers.

For the Support Vector Machine models, all of the models suggest the radial basis function, pointing towards the relation between the strategies and input features to be more complex than some polynomial function, if there is such a relation at all. It also generally recommends a high cost on the slack variables, which also suggests that separating the data set by some metric is difficult to do. Finally, the neural network model always recommends the Adam optimizer, suggesting that it is a good choice for the problem instances, while also suggesting 150 or 200 epochs often. This implies that the training might not be enough in the cases of 200 epochs or it is around the right number of epochs. It also recommends more than one layer for all but one model, which could point towards the model being complex.

4.3 Results from the model using entire dataset

4.3.1 Multi-class classification

The F_1-score and accuracy from testing the machine learning models on the data set for the optimal hyperparameters with the different labellers when transformed into a multi-class classification problem can be found in Tables A.9 − A.12. The best model results for each of the three feature sets can be found in Tables 4.6 − 4.8. The F_1-score and accuracy plots of the best models can be found in Figures 4.4 and 4.5, respectively.

Labeller	**Model**	**Baseline Accuracy** (Random, Heuristic)	**Baseline F_1–score** (Random, Heuristic)	**Accuracy**	**Best model F_1–score**
Running time	Neural network	0.0776 0.2327	0.0814 0.0878	0.2653	0.2490
Total cost	Neural network	0.1265 0.1306	0.1232 0.0302	0.2857	0.3026
Unassigned legs	Neural network	0.1224 0.1224	0.1206 0.0267	0.2040	0.2267
Combination	Neural network	0.1061 0.2245	0.1134 0.0823	0.2244	0.1952

Table 4.6: Best machine learning models for multi-class classification using all features.

Labeller	**Model**	**Baseline Accuracy** (Random, Heuristic)	**Baseline F_1–score** (Random, Heuristic)	**Accuracy**	**Best model F_1–score**
Running time	Neural network	0.0776 0.2327	0.0814 0.0878	0.2449	0.2421
Total cost	Decision tree	0.1265 0.1306	0.1232 0.0302	0.2653	0.2611
Unassigned legs	SVM	0.1224 0.1224	0.1206 0.0267	0.1837	0.2035
Combination	Random forest	0.1061 0.2245	0.1134 0.0823	0.2041	0.2101

Table 4.7: Best machine learning models for multi-class classification using PCA.

Labeller	**Model**	**Baseline Accuracy** (Random, Heuristic)	**Baseline F_1–score** (Random, Heuristic)	**Accuracy**	**Best model F_1–score**
Running time	Decision tree	0.0776 0.2327	0.0814 0.0878	0.2653	0.2265
Total cost	Random forest	0.1265 0.1306	0.1232 0.0302	0.3265	0.2806
Unassigned legs	Random forest	0.1224 0.1224	0.1206 0.0267	0.1633	0.1781
Combination	Random forest	0.1061 0.2245	0.1134 0.0823	0.2041	0.1942

Table 4.8: Best machine learning models for multi-class classification using the five most important features.

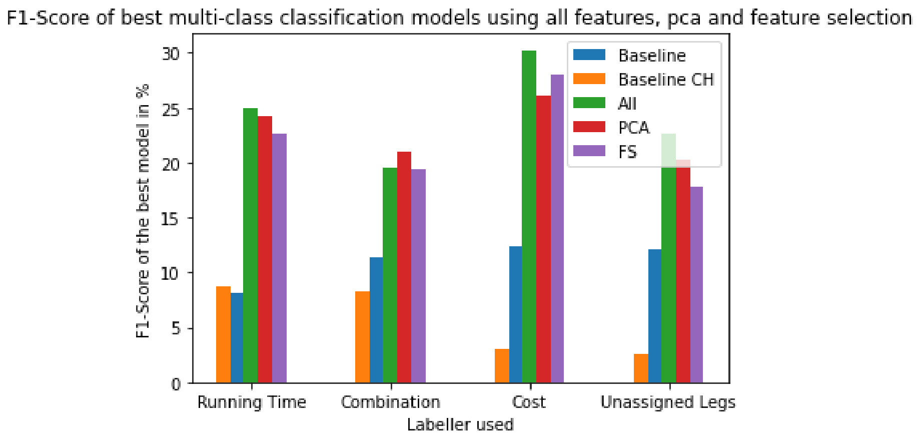


Figure 4.4: F_1-Score of the best multi-class models.

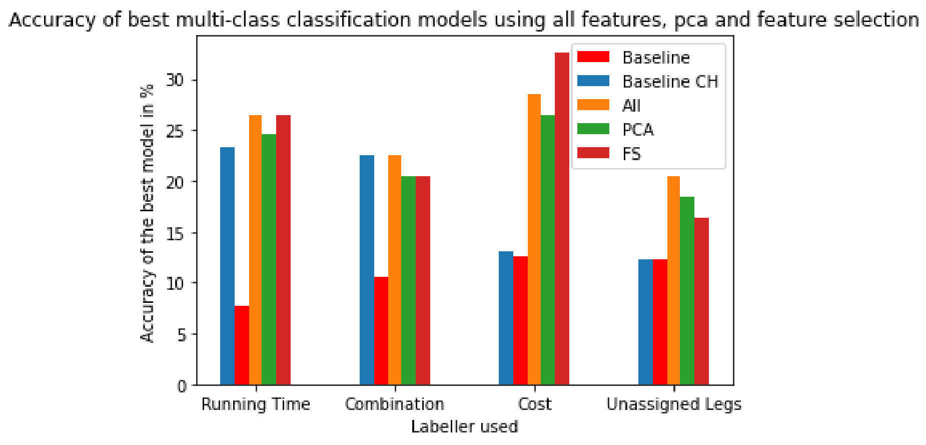


Figure 4.5: Accuracy Score of the best multi-class models.

By looking at the accuracies and F_1-scores of the models in Tables A.9 – A.12 and comparing them with the baseline models, most of the models perform at least better than the baseline models. Overall, the random forest classifier with feature selection gave the highest accuracy of 32.65% when using cost as the labeller. When F_1-score was considered as the main evaluation metric, the neural network gives the highest score of 30.26% when all features are used with cost as the labeller. These results might point towards machine learning at least being able though not by too much, which could be stem from the data being smaller. The decision trees, random forests and neural networks generally perform the best out of the models, while SVM is generally a bit worse but can reach similar levels occasionally. Naive Bayes on the other hand tends to perform worse than all the other models, and even gets worse results than the two baselines sometimes. In particular, with the cost labeller and all features, it has an accuracy and F_1-score of zero.

Another trend that can be seen by comparing the same model with different features, it can be seen that for both Naive Bayes and SVM, using PCA gives better results than using all the features. This makes sense, as naive Bayes assumes that all features are independent, and the principal components are independent with each other, meaning that the features fit the model a lot better. A theory as to why it gives better results for the SVM is that using PCA might orient the data in a way that makes it easier to solve. For the other models, the F_1-score tend to be higher when using all of the features, as using PCA and feature selection does remove information from the data, though there are exceptions like for the cost labeller where the decision tree performs much better for PCA and feature selection than with all features and the random forest has much better results with feature selection.

Finally, while it is a small thing, it seems that most of the models perform very similarly regardless of what label set is used, which could be a result of all sets having similar distributions of labels as seen in Table 4.5. The low F_1-score could come from the number of problem instances being small while still having nine classes to choose from. This means that the classes might not get enough data points to learn the patterns correctly. For example, some of the classes might be assigned to more problem instances and some may be assigned to fewer problem instances as can be seen in Table 4.5. The separation of the classes could probably be better if only three or four classes were used for classification instead of all nine or if more problem instances were added.

4.3.2 Multi-label classification

The F_1-score and the value of 1 minus the Hamming loss from the other multi-label transformations can be found in Tables 4.9 − A.16. The result of the multi-label neural network models can be found in Table A.17. The best model and transformation for each of the three different types of features can be found in the Tables 4.10 − 4.12.

Labeller	**Model**	**(1 - Hamming loss)**	**F_1–score**
Running time	Random model	0.4939	0.3059
	All strategies	0.1950	0.3493
Total cost	Random model	0.4980	0.4531
	All strategies	0.3864	0.5675
Unassigned legs	Random model	0.5016	0.6105
	All strategies	0.7800	0.8791
Combination	Random model	0.5048	0.2985
	All strategies	0.1900	0.3423

Table 4.9: Results of the baseline models for multi-label classification.

Labeller	**Model**	**Transformation**	**(1 - Hamming loss)**	**F_1–score**
Running time	Decision tree	Binary relevance	0.7755	0.4456
Total cost	Random forest	Binary relevance	0.8390	0.7628
Unassigned legs	Random forest	Label powerset	0.8776	0.9294
Combination	Decision tree	Classifier chain	0.7778	0.4252

Table 4.10: Best machine learning models for multi-label classification using all features.

Labeller	**Model**	**Transformation**	**(1 - Hamming loss)**	F_1**–score**
Running time	Decision tree	Binary relevance	0.7732	0.4185
Total cost	Random forest	Classifier chain	0.8299	0.7377
Unassigned legs	Random forest	Binary relevance	0.8639	0.9197
Combination	SVM	Classifier chain	0.7664	0.4044

Table 4.11: Best machine learning models for multi-label classification using PCA.

Labeller	**Model**	**Transformation**	**(1 - Hamming loss)**	F_1**–score**
Running time	Decision Tree	Classifier chain	0.8322	0.5283
Total cost	Random Forest	Binary relevance	0.8503	0.7776
Unassigned legs	Random Forest	Label powerset	0.8776	0.9291
Combination	Decision Tree	Label powerset	0.7937	0.5140

Table 4.12: Best machine learning models for multi-label classification using top 5 features.

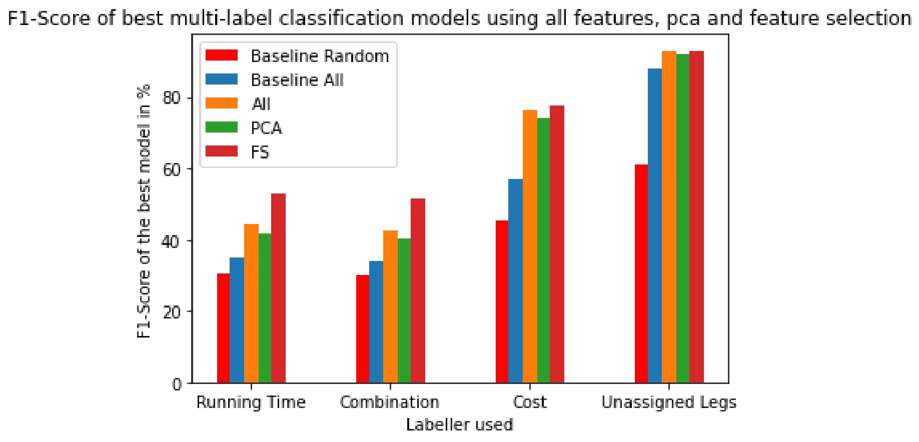


Figure 4.6: F_1-Score of the best multi-label models.

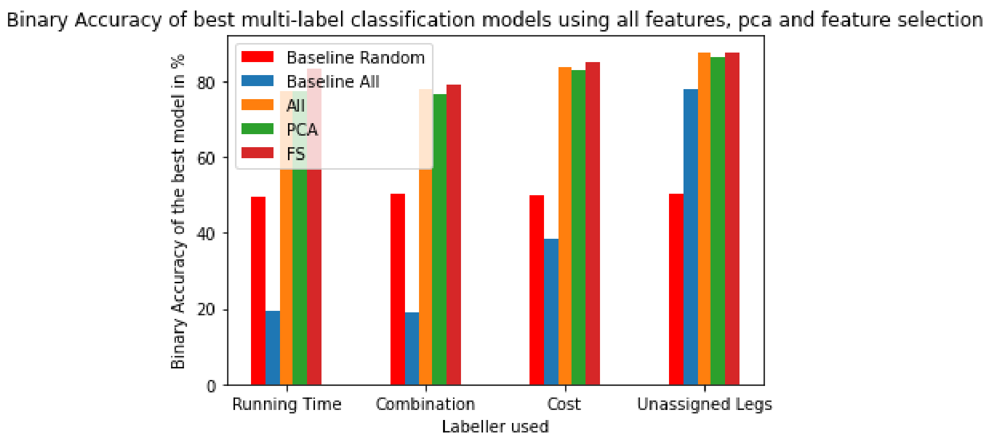


Figure 4.7: 1 - Hamming loss for the best multi-label models.

If one looks at Tables A.13 – A.16, one generally sees that the Hamming loss and the F_1-score is widely different across the the labellers, unlike the previous transformation. In particular, the unassigned legs labellers baseline F_1-score is around twice that of the running time and combination labeller. This is likely due to the more plentiful recommended strategies causing the rate of possible false positives to be much lower, which would increase the precision during a random choice, while not affecting the rate of true positives and false negatives, which would increase the F_1-score. This shows the usefulness of the F_1-score, as the Hamming loss between the four labellers are much closer to each other compared to the F_1-score.

Otherwise, when comparing the scores within a labeller, a lot of the conclusions about how the models perform are similar to the single label case, including that most of the models aside from naive Bayes on occasion is better than the baseline models, though there are some exceptions. For example, the naive Bayes classifier has almost twice the F_1-score with all features compared to when it uses the PCA for both binary relevance and label powerset when the running time or all three output features determine the label.

Comparing the performance of all the machine learning models, the decision tree gives the best F_1-scores of 44.56% and 42.52% when running time and combination of all features were used as the labellers respectively. Similarly, random forest gives the best F_1-scores of 76.28% and 92.94% when total cost and unassigned legs were used as the labellers respectively. Interestingly, the results from the models when using PCA and feature selection tend to give good results which are quite close to the results from using all the features. This suggests that when any of those transforms are done on the data set, the models tend to learn the patterns which they learn when all the features are used. It also helps in making faster predictions and saves significant computations.

Also, as can be seen in Figure A.17, the F_1-score for the neural network models, are much lower than the other models for the multi-label set. In fact, it is much closer to the multi-class data sets instead. This might be because it's method of predicting the labels are much closer to the multi-class counterpart compared to the other models. This holds especially true to the running time and combination labelled data sets, which only uses one label for most of the data already.

4.4 Predictions from the trained models on data of a different scale

After training the model on the 120 problem instances which were collected initially, the five machine learning models were trained using this data and were tested against the 39 problem instances which were collected later. The prediction results of the model for new unseen data are found in Tables A.18 – A.20. The best results of the predictions are found in Tables 4.13 and 4.14. Figures 4.8 and 4.9 contains the prediction results of multi-class classification and multi-label classification, respectively.

Labeller	**Model**	F_1**-score**
Running time	Decision Tree	0.2007
Total cost	Neural Network	0.1537
Unassigned legs	Neural Network	0.0978
Combination	Decision Tree	0.2339

Table 4.13: Best machine learning models prediction for multi-class classification using all features.

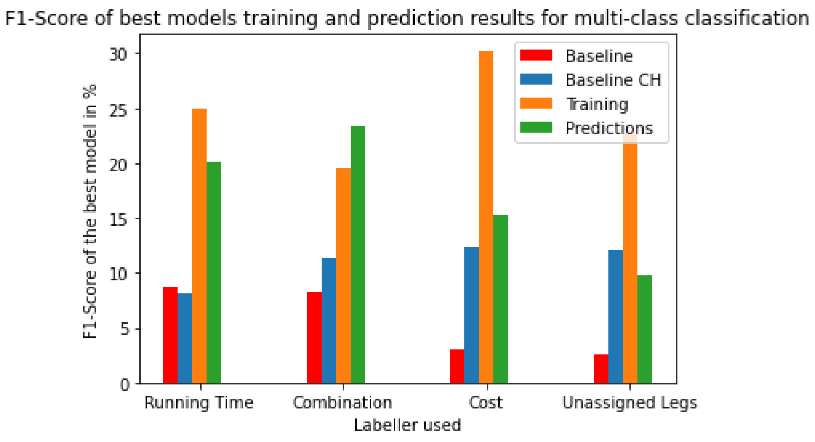


Figure 4.8: F_1-Score when predicting large-scale problem instances with small-scale problem instances using multi-class classification.

Labeller	Model	Transformation	F_1-score
Running time	Decision Tree	Label Powerset	0.1792
Total cost	Decision Tree	Label Powerset	0.4912
Unassigned legs	Random Forest	Label Powerset	0.8898
Combination	Random Forest	Label Powerset	0.2231

Table 4.14: Best machine learning models prediction for multi-label classification using all features.

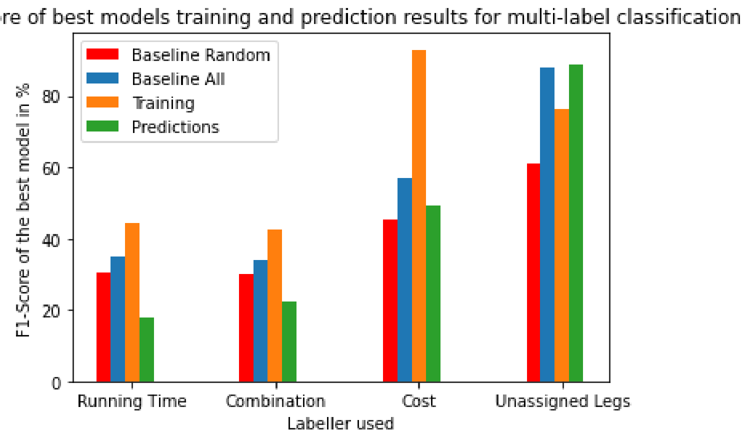


Figure 4.9: F_1-Score when predicting large-scale problem instances with small-scale problems using multi-label classification.

By comparing the results of these models with the counterparts for the full data set, one can see that the results are generally a lot worse. For the multi-class models, the decision tree with the combination labeller has the best result of 23.39% and the neural network with the unassigned labeller has the worst result of 9.78%. Interestingly, the results from the neural network using the combination labeller and all the data points has a lower F_1-score of 19.52% than the decision tree using the 120 problem instances. This suggests that the model is able to generalize to unseen instances to some extent. All multi-class models except the models using the unassigned labellers give better results than the random baseline model. However for the multi-label models, only the unassigned legs labeller performs better than the two baselines.

From Table A.19, it can be seen that some of the models using the running time labeller has an F_1-score of zero in the multi-labelled case. This is likely due to the input features being very different for large-scale problem instances in test set compared to the small-scale instances used for training. Furthermore, in the multi-label case, instances at a smaller scale are easier to solve, meaning that they are likely to have more strategies that are considered good under the labelling conditions. For example, it would be easy for a multi-label model to learn and consider a strategy good, when that does not fit the larger scale problem instances. Also, the training data set used is much smaller, which also makes it harder for the models to predict the labels correctly.

From Table 4.14 and Figure 4.9, it is clear that the label powerset transformation works better for the multi-label classification. The results from the running time labeller and the combination labeller are slightly lower than the baseline models. This might be due to the fact that the input features for the training set and testing set differs significantly because of the size of the problem instances.

4.5 Answers to the research questions

To answer the question of which model works best, there are different answers depending on if multi-class classification or multi-label classification is used. In case of the multi-class classification using all features, the neural networks has an edge over the other machine learning models. While performing multi-label classification, both the decision tree and random forest perform better than other models.

To answer if only using warm start or cold start is a better strategy than the current heuristic, it is quite evident that there are instances for which the current heuristic is not an optimal strategy for labellers using running time and total objective cost, as shown in Tables 4.3 and 4.4. These tables also show that sometimes the best strategy was to use only warm start (Strategy 2) or to use only cold start (Strategy 3).

Finally, to answer if a choice between predefined strategies based on the input features is possible with machine learning, the answer is not quite clear. The results in Section 4.3 show that most machine learning models learn the strategy choice better than just a random guess, but with the data given, it is not yet enough reliable to be used in practice.

5 Conclusion

5.1 Future work

The results point towards machine learning at least being better than both of the baseline models. There should be further investigation on using machine learning for this problem. Some things that can be investigated are to try the same problem with fewer strategies or to formulate new strategies that work better. Some ideas for the new strategies could be heuristics that perform better than the current heuristic on some of the problem instances. One can also use only some of the better strategies from the experimented nine strategies in this thesis. By having fewer strategies, it should be easier to tell if there are patterns to be found that determine the best strategy among those investigated. This would also reduce the time needed to collect the data in the first place, as there are fewer output features that need to be collected.

The results of training on smaller problem instances and testing on larger problem instances which was implemented implies that it was not a fair idea as the machine learning models has not seen the larger instances yet. This is likely because the larger problem instances have a different input features compared to the smaller problem instances. Because of this, if the goal is to improve the strategy choice for larger problem instances, only problem instances that are of larger scale should be used. This could also make it easier to formulate the machine learning as a multi–class classification problem for larger problem instances. This is mainly due to more decisions between warm start and cold start, leading to more problems having only one of the strategies as the optimal strategy. This is good because multi–label problems are generally more complex than multi–class problems. Due to using a combination of labels instead of a single label, the results are a bit harder to interpret and that it is much easier to have the data set be imbalanced.

An aspect that wasn't considered in the results was how machine learning would affect Jeppesen Systems' column generation framework when implemented into it. An idea of how to implement the machine learning into the column generation framework is to have a pretrained machine learning model that has learnt patterns from the previous data from the framework. When running the new framework, the input features are collected and then sent to the pretrained model to determine the strategy to use. Of course, some of the input features used in this thesis were extracted from the data and not originally a part of the code, implying that extracting the input features will take some extra time. However, it would still be a good idea to implement machine learning to decide the starting point since it might save a lot of computation time. The overhead time of collecting features and running the model should be negligible compared to possible gains in computational time.

Finally, this thesis did not investigate what start to use at each iteration, but instead looked what strategy to use for the entire problem instance based on features of the instance before solving it. If one were to determine what start to use at each iteration, the choice within the problem instance would be a lot more complicated. This is due to only being able to know the best choice by exhaustively looking at every combination of choices until the optimizer terminates which needs to be considered. Here, reinforcement learning could be a good approach as it learns from the decisions it makes. However, this will still be a hard problem to solve.

5.2 Conclusion

Our results show that there might be something in the input data that the machine learning models learn from. However, judging from the complexity of the optimal hyperparameters of the Support Vector Machine and the feature importance giving similar importance to all features, the decision is very complicated. If there is a reliable way to determine the correct strategy, the decision is very complex. With more analysis regarding other input features, other types of machine learning models and their hyperparameters, the choice of what strategies to look at and more collected data, machine learning models can be an applicable approach for choosing starting strategies in the column generation scheme. If reliable and robust methods would be found and implemented these might reduce the computational time for Jeppesen Systems considerably.

Bibliography

[1] M. Grönkvist, *The tail assignment problem.*, ser. PHD at Chalmers University of Technology. New series: 2327. Chalmers tekniska högskola, 2005.

[2] G. B. Dantzig and P. Wolfe, "Decomposition principle for linear programs." *Operations Research*, vol. 8, no. 1, pp. 101–111, 1960.

[3] J. Desrosiers and M. E. Lübbecke, "A primer in column generation," in *Column Generation*, G. Desaulniers, J. Desrosiers, and M. M. Solomon, Eds. Boston, MA: Springer US, 2005, pp. 1–32.

[4] M. Fuentes, L. Cadarso, V. Vaze, and C. Barnhart, "The tail assignment problem: A case study at vueling airlines," *Transportation Research Procedia*, vol. 52, pp. 445–452, 2021, 23rd EURO Working Group on Transportation Meeting, EWGT 2020, 16-18 September 2020, Paphos, Cyprus.

[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: theory, algorithms and applications.* Prentice Hall, 1993.

[6] E. Gustavsson, M. Patriksson, and A.-B. Strömberg, "Primal convergence from dual subgradient methods for convex optimization." *Mathematical Programming*, vol. 150, no. 2, pp. 365–390, 2015.

[7] J. Lundgren, M. Rönnqvist, and P. Värbrand, *Optimization.* Studentlitteratur, 2010.

[8] A.-B. Strömberg, T. Larsson, and M. Patriksson, *Mixed-Integer Linear Optimization: Primal–Dual Relations and Dual Subgradient and Cutting-Plane Methods.* Cham: Springer International Publishing, 2020, pp. 499–547.

[9] N. Andréasson, A. Evgrafov, and M. Patriksson, *An introduction to continuous optimization: Foundations and fundamental algorithms.*, 3rd ed. Studentlitteratur, 2016.

[10] K. M. Anstreicher and L. A. Wolsey, "Two "well-known" properties of subgradient optimization." *Mathematical Programming*, vol. 120, no. 1, pp. 213–220, 2009.

[11] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, "Branch-and-price: Column generation for solving huge integer programs." *Operations Research*, vol. 46, no. 3, pp. 316–329, 1998.

[12] D. Huisman, R. Jans, M. Peeters, and A. P. Wagelmans, "Combining column generation and Lagrangian relaxation," in *Column Generation*, G. Desaulniers, J. Desrosiers, and M. M. Solomon, Eds. Boston, MA: Springer US, 2005, pp. 247–270.

[13] P. G. Brevis, "Advances in interior point methods and column generation," University of Edinburgh, 2013.

[14] C. Sammut and G. I. Webb, Eds., *Data Set.* Boston, MA: Springer US, 2017, pp. 327–327.

[15] ——, *Unlabeled Data.* Boston, MA: Springer US, 2017, pp. 1304–1304.

[16] M. J. Haber, *Machine Learning.* Berkeley, CA: Apress, 2020, pp. 361–365.

[17] M. Kac, G.-C. Rota, and J. T. Schwartz, *Artificial Intelligence.* Boston, MA: Birkhäuser Boston, 1992, pp. 183–190.

[18] G. Dougherty, *Supervised Learning.* New York, NY: Springer New York, 2013, pp. 75–98.

[19] ——, *Unsupervised Learning.* New York, NY: Springer New York, 2013, pp. 143–155.

[20] M. F. A. Hady and F. Schwenker, *Semi-supervised Learning.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 215–239.

[21] S. Whiteson, *Reinforcement Learning.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 7–15.

[22] Srinivasa, K. G., Siddesh G. M., and Srinidhi H., *Regression.* Cham: Springer International Publishing, 2018, pp. 139–154.

[23] U. R. Hodeghatta and U. Nayak, *Supervised Machine Learning—Classification.* Berkeley, CA: Apress, 2017, pp. 131–160.

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016.

[25] S. Raschka, Y. Liu, V. Mirjalili, and D. Dzhulgakov, *Machine Learning with PyTorch and Scikit-Learn : Develop Machine Learning and Deep Learning Models with Python.* Birmingham, UK: Packt Publishing, 2022.

[26] I. Jolliffe, *Principal Component Analysis.*, ser. Springer Series in Statistics. Springer New York, 2002.

[27] G. I. Webb, *Naïve Bayes.* Boston, MA: Springer US, 2017, pp. 895–896.

[28] C. Sammut and G. I. Webb, Eds., *Bayes' Theorem.* Boston, MA: Springer US, 2017, p. 100.

[29] *Support Vector Machines for Classification.* New York, NY: Springer New York, 2008, pp. 285–329.

[30] R. Fernandes de Mello and M. Antonelli Ponti, *Machine Learning: A Practical Approach on the Statistical Learning Theory.* Springer International Publishing, 2018.

[31] K. P. Murphy, *Machine learning : a probabilistic perspective.*, ser. Adaptive computation and machine learning series. MIT Press, 2012.

[32] S. Shanmuganathan, *Artificial Neural Network Modelling: An Introduction.* Cham: Springer International Publishing, 2016, pp. 1–14.

[33] I. N. da Silva, D. Hernane Spatti, R. Andrade Flauzino, L. H. B. Liboni, and S. F. dos Reis Alves, *Artificial Neural Networks : A Practical Course.* Cham: Springer International Publishing, 2017, pp. 3–19.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: https://doi.org/10.48550/arXiv.1412.6980

[35] F. Herrera, F. Charte, A. J. Rivera, and M. J. del Jesus, *Multilabel Classification: Problem Analysis, Metrics and Techniques.* Springer International Publishing, 2016.

[36] A. Panesar, *Evaluating Machine Learning Models.* Berkeley, CA: Apress, 2021, pp. 189–205.

A

Appendix - Tables of results

This appendix will contain more detailed tables of the results from Section 4

A.1 Data distribution

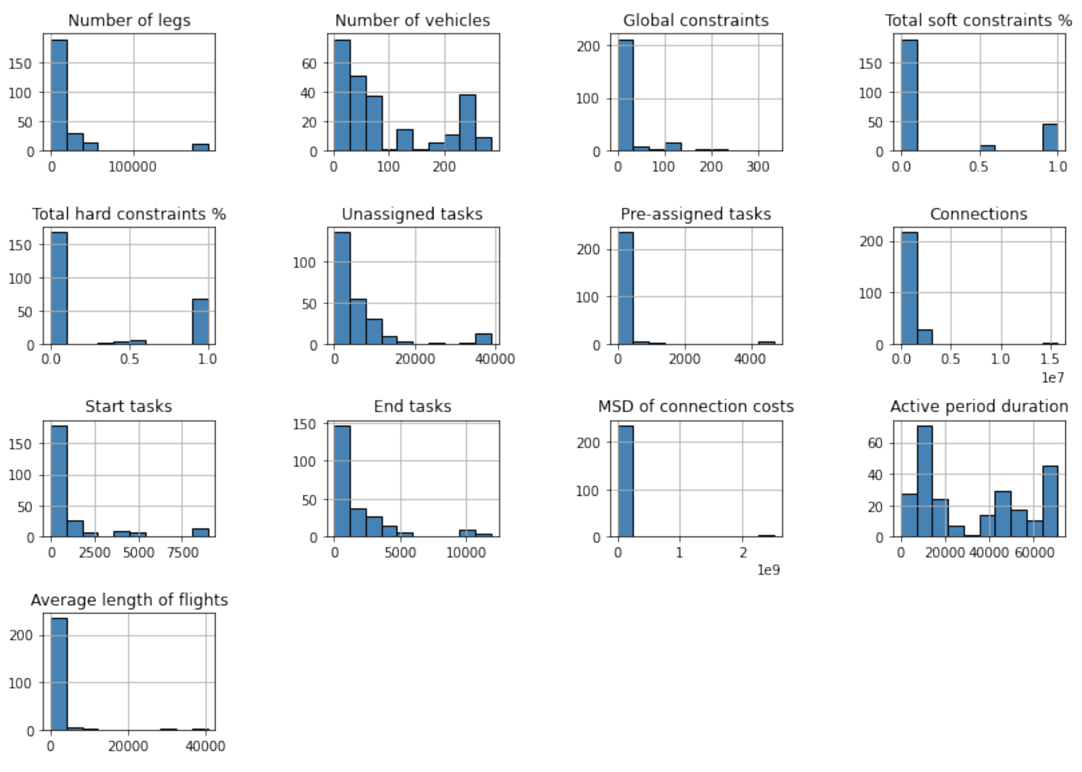


Figure A.1: Data distribution of input features - set 1.

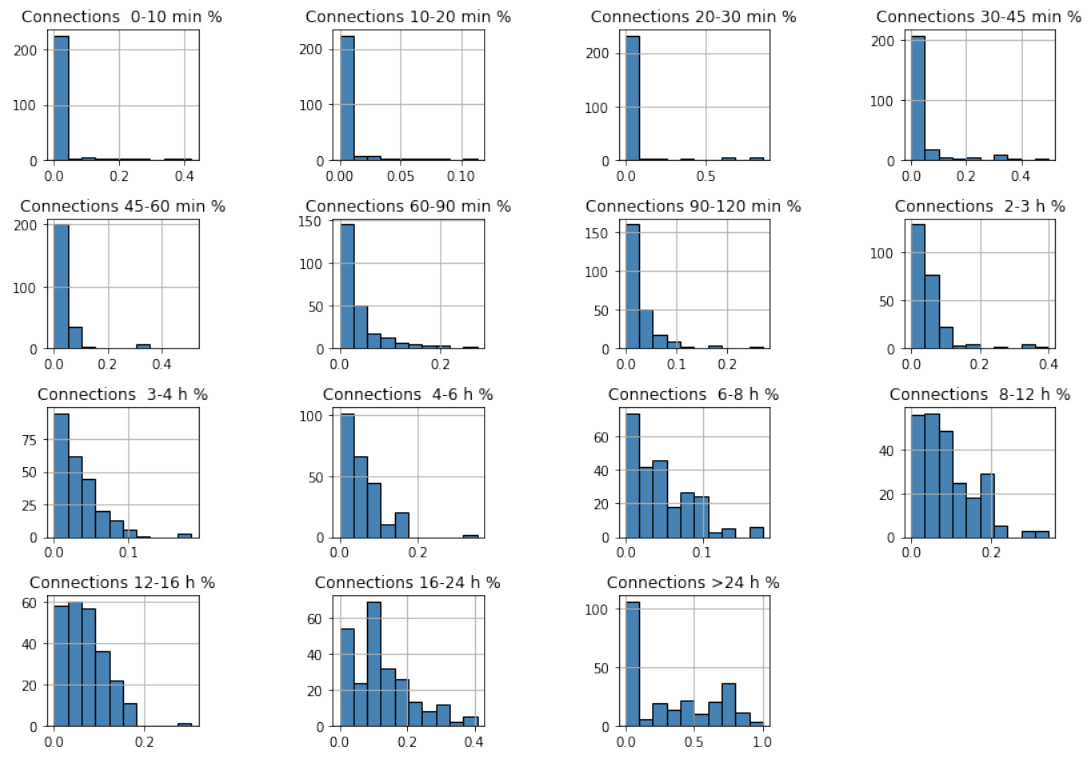


Figure A.2: Data distribution of input features - set 2.

A.2 Feature Importances

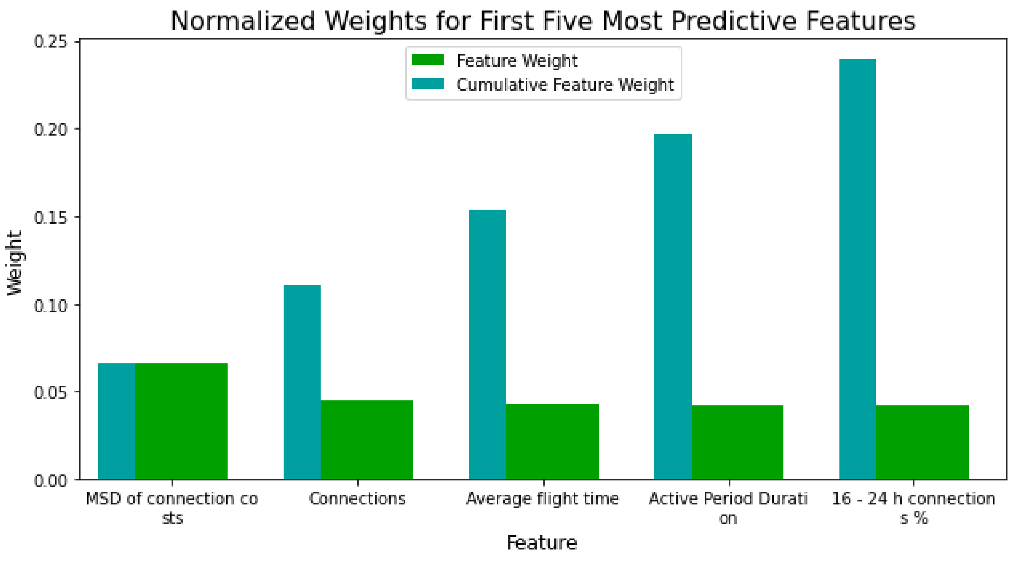


Figure A.3: Histogram of the importance of the most important features according to a random forest classifier when using the time as a label.

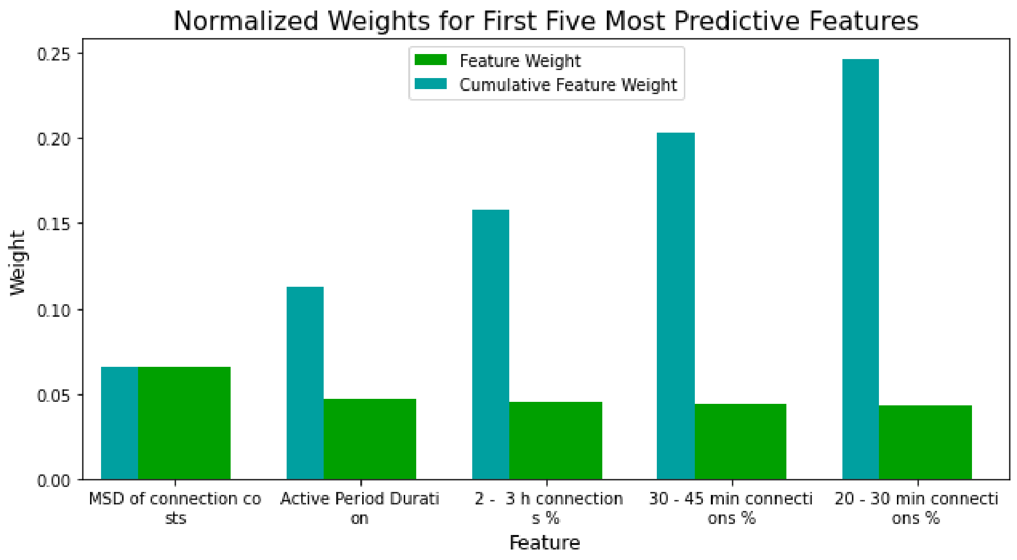


Figure A.4: Histogram of the importance of the most important features according to a random forest classifier when using the cost as a label.

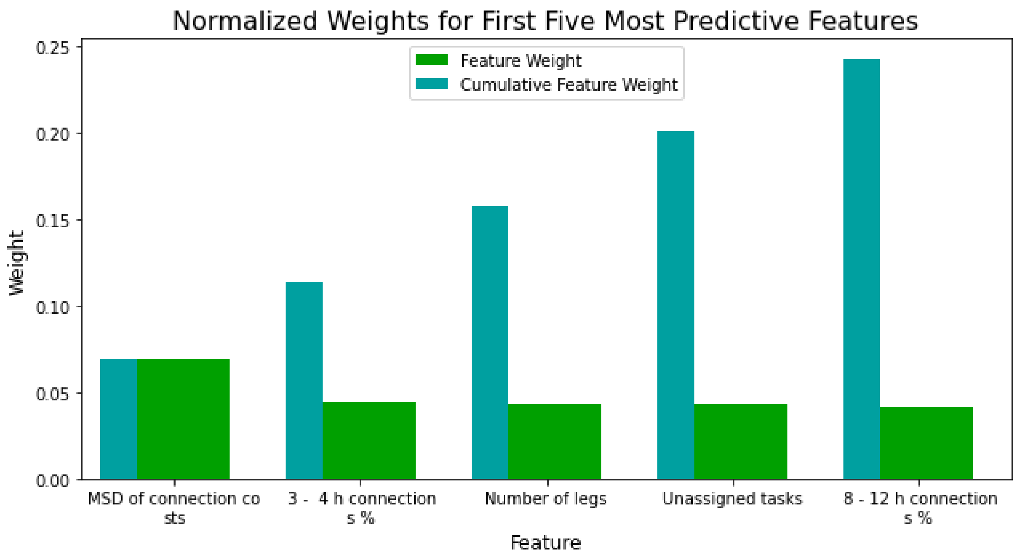


Figure A.5: Histogram of the importance of the most important features according to a random forest classifier when using the unassigned legs as a label.

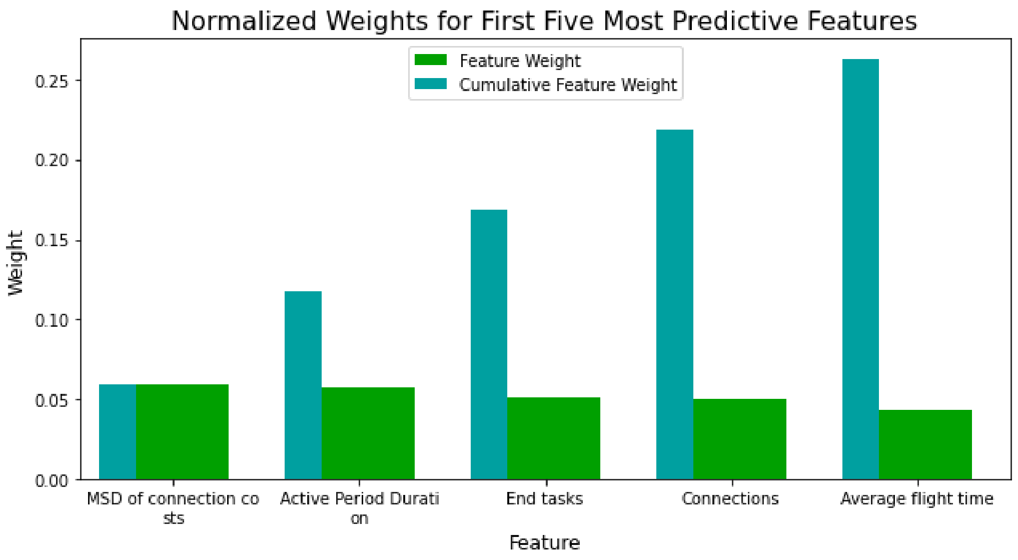


Figure A.6: Histogram of the importance of the most important features according to a random forest classifier when using a combination of all three output features as the label.

A.3 Optimal hyperparameters

Model	**Features**	**Parameter**	**Optimal value**
Decision Tree	All	Cost function Max depth	Entropy 7
Random Forest	All	Cost function Max depth Number of trees	Entropy 6 150
Support Vector Machine	PCA	C Kernel	100 RBF
Neural network	All	Hidden layers Neurons per layer Optimizer Activation function Number of epochs	1 16 Adam ReLU 150

Table A.1: Optimal hyperparameters and feature sets using the running time labeller for the multi–class classifiers.

Model	**Features**	**Parameter**	**Optimal value**
Decision Tree	PCA	Cost function	Gini
		Max depth	No limit
Random forest	FS	Cost function	Gini
		Max depth	4
		Number of trees	150
Support Vector Machine	PCA	C	100
		Kernel	RBF
Neural network	All	Hidden layers	3
		Neurons per layer	32
		Optimizer	Adam
		Activation	ReLU
		Number of epochs	200

Table A.2: Optimal hyperparameters and feature sets using the cost labeller for the multi–class classifiers.

Model	**Features**	**Parameter**	**Optimal value**
Decision Tree	All	Cost function	Entropy
		Max depth	7
Random forest	All	Cost function	Gini
		Max depth	5
		Number of trees	100
Support Vector Machine	PCA	C	100
		Kernel	RBF
Neural network	All	Hidden layers	3
		Neurons per layer	64
		Optimizer	Adam
		Activation function	tanh
		Number of epochs	200

Table A.3: Optimal hyperparameters and feature set using the unassigned legs labeller for the multi–class classifiers.

Model	**Features**	**Parameter**	**Optimal value**
Decision Tree	PCA	Cost function	Entropy
		Max depth	7
Random forest	PCA	Cost function	Entropy
		Max depth	6
		Number of trees	150
Support Vector Machine	PCA	C	100
		Kernel	RBF
Neural network	PCA	Hidden layers	2
		Neurons per layer	32
		Optimizer	Adam
		Activation function	tanh
		Number of epochs	100

Table A.4: Optimal hyperparameters and feature sets using the combined output labeller for the multi–class classifiers.

Model	**Transfomation**	**Features**	**Parameter**	**Optimal value**
Decision Tree	Classifier chain	FS	Cost function	Entropy
			Max depth	5
Random Forest	Classifier chain	FS	Cost function	Gini
			Max depth	5
			Number of trees	200
SVM	Classifier chain	PCA	C	100
			Kernel	RBF
Neural network	-	All	Hidden layers	2
			Neurons per layer	16
			Optimizer	Adam
			Activation function	ReLU
			Number of epochs	200

Table A.5: Optimal hyperparameters, transformations and feature sets using the running time labeller for the multi–label classifiers.

Model	**Transfomation**	**Features**	**Parameter**	**Optimal value**
Decision Tree	Binary relevance	FS	Cost function	Gini
			Max depth	No limit
Random Forest	Binary relevance	FS	Cost function	Gini
			Max depth	6
			Number of trees	50
SVM	Label powerset	All	C	1
			Kernel	RBF
Neural network	-	All	Hidden layers	2
			Neurons per layer	64
			Optimizer	Adam
			Activation function	tanh
			Number of epochs	200

Table A.6: Optimal hyperparameters, transformations and feature sets using the cost labeller for the multi–label classifiers.

Model	**Transfomation**	**Features**	**Parameter**	**Optimal value**
Decision Tree	Label powerset	All	Cost function	Gini
			Max depth	6
Random Forest	Classifier chain	FS	Cost function	Entropy
			Max depth	6
			Number of trees	50
SVM	Classifier chain	PCA	C	10
			Kernel	RBF
Neural network	-	PCA	Hidden layers	2
			Neurons per layer	32
			Optimizer	Adam
			Activation function	ReLU
			Number of epochs	200

Table A.7: Optimal hyperparameters, transformations and feature sets using the unassigned legs labeller for the multi–label classifiers.

Model	**Transfomation**	**Features**	**Parameter**	**Optimal value**
Decision Tree	Classifier chain	FS	Cost function	Entropy
			Max depth	No limit
Random Forest	Classifier chain	FS	Cost function	Gini
			Max depth	4
			Number of trees	150
SVM	Classifier chain	PCA	C	100
			Kernel	RBF
Neural network	-	FS	Hidden layers	2
			Neurons per layer	16
			Optimizer	Adam
			Activation function	tanh
			Number of epochs	150

Table A.8: Optimal hyperparameters, transformations and feature sets using the combination labeller for the multi–label classifiers.

A.4 Multi–class scores

Model	**Features**	**Accuracy**	**F_1-score**
Random baseline	-	0.0776	0.0814
Current heuristic baseline	-	0.2327	0.0878
Decision tree	All	0.2857	0.2348
	PCA	0.2041	0.1804
	FS	0.2653	0.2265
Random forest	All	0.2041	0.1883
	PCA	0.1837	0.1369
	FS	0.2245	0.1605
Naive Bayes	All	0.0612	0.0787
	PCA	0.1224	0.0820
	FS	0.1224	0.0684
SVM	All	0.0612	0.0456
	PCA	0.1633	0.1467
	FS	0.1633	0.1131
Neural network	All	0.2653	0.2490
	PCA	0.2449	0.2421
	FS	0.2244	0.2005

Table A.9: Results on testing set using the running time labeller for all the 245 observations in data set for the multi–class classification.

Model	Features	Accuracy	F_1-score
Random baseline	-	0.1265	0.1232
Current heuristic baseline	-	0.1306	0.0302
Decision tree	All	0.2041	0.2042
	PCA	0.2653	0.2611
	FS	0.2449	0.2459
Random forest	All	0.2449	0.2281
	PCA	0.2449	0.2263
	FS	0.3265	0.2806
Naive Bayes	All	0.0000	0.0000
	PCA	0.1020	0.1103
	FS	0.0816	0.0734
SVM	All	0.2041	0.1899
	PCA	0.2653	0.2354
	FS	0.1633	0.1664
Neural network	All	0.2857	0.3026
	PCA	0.2449	0.2433
	FS	0.2449	0.2381

Table A.10: Results on testing set using the cost labeller for all the 245 observations in data set for the multi–class classification.

Model	Features	Accuracy	F_1-score
Random baseline	-	0.1224	0.1206
Current heuristic baseline	-	0.1224	0.0267
Decision tree	All	0.1837	0.1656
	PCA	0.1429	0.1375
	FS	0.0816	0.0923
Random forest	All	0.1633	0.1795
	PCA	0.1429	0.1218
	FS	0.1633	0.1781
Naive Bayes	All	0.0816	0.0837
	PCA	0.1633	0.1518
	FS	0.1020	0.0734
SVM	All	0.1429	0.1515
	PCA	0.1837	0.2035
	FS	0.1429	0.1304
Neural network	All	0.2040	0.2267
	PCA	0.1632	0.1890
	FS	0.2040	0.1650

Table A.11: Results on testing set using the unassigned legs labeller for all the 245 observations in data set for the multi–class classification.

Model	**Features**	**Accuracy**	**F$_1$-score**
Random baseline	-	0.1061	0.1134
Current heuristic baseline	-	0.2245	0.0823
	All	0.1633	0.1504
Decision tree	PCA	0.1837	0.1932
	FS	0.1837	0.1883
	All	0.2041	0.1906
Random forest	PCA	0.2041	0.2101
	FS	0.2041	0.1942
	All	0.0612	0.0529
Naive Bayes	PCA	0.1837	0.1455
	FS	0.1429	0.1020
	All	0.1633	0.1720
SVM	PCA	0.1837	0.1780
	FS	0.1633	0.1532
	All	0.2244	0.1952
Neural network	PCA	0.2449	0.1994
	FS	0.2040	0.1892

Table A.12: Results on testing set using the combination labeller for all the 245 observations in data set for the multi–class classification.

A.5 Multi–label scores

Model	**Transformation**	**Features**	**(1 - Hamming loss)**	**F_1-score**
Random baseline	-	-	0.4939	0.3059
All baseline	-	-	0.1950	0.3493
Decision tree	Binary relevance	All	0.7755	0.4456
		PCA	0.7732	0.4185
		FS	0.7914	0.4818
	Classifier chain	All	0.7528	0.3983
		PCA	0.7483	0.3737
		FS	0.8322	0.5283
	Label powerset	All	0.7823	0.4341
		PCA	0.7710	0.4048
		FS	0.7868	0.4863
Random forest	Binary relevance	All	0.8141	0.4062
		PCA	0.8163	0.3803
		FS	0.8231	0.4581
	Classifier chain	All	0.8118	0.3904
		PCA	0.8186	0.3762
		FS	0.8413	0.4937
	Label powerset	All	0.7823	0.3841
		PCA	0.7800	0.3978
		FS	0.7982	0.4960
Naive Bayes	Binary relevance	All	0.5782	0.4066
		PCA	0.7551	0.0851
		FS	0.5193	0.3726
	Classifier chain	All	0.5442	0.3944
		PCA	0.7710	0.1614
		FS	0.4535	0.4483
	Label powerset	All	0.7052	0.1780
		PCA	0.7415	0.2490
		FS	0.7188	0.4142
SVM	Binary relevance	All	0.7846	0.3777
		PCA	0.7846	0.3706
		FS	0.7370	0.2742
	Classifier chain	All	0.7959	0.3622
		PCA	0.7619	0.4213
		FS	0.6916	0.2870
	Label powerset	All	0.7528	0.3576
		PCA	0.7460	0.3602
		FS	0.7279	0.0955

Table A.13: Results on the testing set for the running time labeller for the full 245 observation data set with the other transformations.

Model	**Transformation**	**Features**	**(1 - Hamming loss)**	**F_1-score**
Random baseline	-	-	0.4980	0.4531
All baseline	-	-	0.3864	0.5675
Decision tree	Binary relevance	All	0.8186	0.7435
		PCA	0.7460	0.6598
		FS	0.8322	0.7654
	Classifier chain	All	0.7392	0.6868
		PCA	0.7415	0.6536
		FS	0.7574	0.6897
	Label powerset	All	0.7642	0.6788
		PCA	0.7166	0.6197
		FS	0.7755	0.7276
Random forest	Binary relevance	All	0.8390	0.7628
		PCA	0.7891	0.6743
		FS	0.8503	0.7776
	Classifier chain	All	0.8277	0.7518
		PCA	0.8299	0.7377
		FS	0.8526	0.7739
	Label powerset	All	0.7959	0.7417
		PCA	0.7710	0.7044
		FS	0.8186	0.7571
Naive Bayes	Binary relevance	All	0.6599	0.5478
		PCA	0.7211	0.5193
		FS	0.5374	0.5700
	Classifier chain	All	0.7256	0.5437
		PCA	0.7483	0.6130
		FS	0.5147	0.5667
	Label powerset	All	0.7166	0.5581
		PCA	0.7324	0.6320
		FS	0.6667	0.5247
SVM	Binary relevance	All	0.7438	0.6208
		PCA	0.7528	0.6424
		FS	0.6803	0.5889
	Classifier chain	All	0.7619	0.6289
		PCA	0.7755	0.6610
		FS	0.6077	0.5742
	Label powerset	All	0.6825	0.6703
		PCA	0.6009	0.6108
		FS	0.6145	0.5948

Table A.14: Results on the testing set for the cost labeller for the full 245 observation data set with the other transformations.

Model	**Transformation**	**Features**	**(1 - Hamming loss)**	**F_1-score**
Random baseline	-	-	0.5016	0.6105
All baseline	-	-	0.7800	0.8791
Decision tree	Binary relevance	All	0.8322	0.8973
		PCA	0.8163	0.8835
		FS	0.8345	0.8986
	Classifier chain	All	0.8390	0.9024
		PCA	0.8435	0.9023
		FS	0.8050	0.8788
	Label powerset	All	0.8662	0.9228
		PCA	0.8299	0.9003
		FS	0.8141	0.8898
Random forest	Binary relevance	All	0.8617	0.9185
		PCA	0.8639	0.9197
		FS	0.8639	0.9190
	Classifier chain	All	0.8753	0.9285
		PCA	0.8458	0.9103
		FS	0.8594	0.9193
	Label powerset	All	0.8776	0.9294
		PCA	0.8594	0.9193
		FS	0.8776	0.9291
Naive Bayes	Binary relevance	All	0.6440	0.7440
		PCA	0.7710	0.8625
		FS	0.7710	0.8560
	Classifier chain	All	0.5488	0.6204
		PCA	0.6553	0.7666
		FS	0.8345	0.8978
	Label powerset	All	0.7642	0.8474
		PCA	0.7234	0.8219
		FS	0.7914	0.8655
SVM	Binary relevance	All	0.8254	0.8989
		PCA	0.8073	0.8903
		FS	0.8118	0.8868
	Classifier chain	All	0.7891	0.8690
		PCA	0.8322	0.9066
		FS	0.8141	0.8886
	Label powerset	All	0.8186	0.8858
		PCA	0.8481	0.9090
		FS	0.7959	0.8729

Table A.15: Results on the testing set for the unassigned legs labeller for the full 245 observation data set with the other transformations.

Model	**Transformation**	**Features**	**(1 - Hamming loss)**	**F_1-score**
Random baseline	-	-	0.5048	0.2985
All baseline	-	-	0.1900	0.3423
Decision tree	Binary relevance	All	0.7755	0.4061
		PCA	0.7732	0.3865
		FS	0.8141	0.5065
	Classifier chain	All	0.7778	0.4252
		PCA	0.7755	0.3736
		FS	0.8073	0.4536
	Label powerset	All	0.7642	0.3780
		PCA	0.7619	0.4009
		FS	0.7937	0.5140
Random forest	Binary relevance	All	0.8141	0.3992
		PCA	0.8186	0.3679
		FS	0.8254	0.4587
	Classifier chain	All	0.8163	0.4110
		PCA	0.8141	0.3630
		FS	0.8322	0.4784
	Label powerset	All	0.7823	0.3932
		PCA	0.7800	0.3919
		FS	0.7664	0.4490
Naive Bayes	Binary relevance	All	0.6213	0.4128
		PCA	0.7687	0.1500
		FS	0.5420	0.3404
	Classifier chain	All	0.5805	0.4063
		PCA	0.7755	0.1673
		FS	0.4966	0.3791
	Label powerset	All	0.7279	0.1823
		PCA	0.7551	0.2962
		FS	0.7574	0.4481
SVM	Binary relevance	All	0.7710	0.3421
		PCA	0.7732	0.3520
		FS	0.7370	0.2925
	Classifier chain	All	0.7619	0.3507
		PCA	0.7664	0.4044
		FS	0.7438	0.1123
	Label powerset	All	0.7483	0.3272
		PCA	0.7460	0.3437
		FS	0.7506	0.2256

Table A.16: Results on the testing set for the combination labeller for the full 245 observation data set with the other transformations.

Labeller	Features	(1 - Hamming loss)	F_1-score
Running time	All	0.8185	0.2550
	PCA	0.7845	0.2448
	FS	0.7959	0.2387
Total cost	All	0.7528	0.2328
	PCA	0.6734	0.2263
	FS	0.6644	0.1962
Unassigned legs	All	0.7845	0.2095
	PCA	0.8004	0.2129
	FS	0.7324	0.1877
Combination	All	0.8185	0.2312
	PCA	0.7936	0.2355
	FS	0.7936	0.2450

Table A.17: Results on the testing set for the multi–label neural network model using the full 245 observation data set.

A.6 Predictions from the trained models on data of a different scale

Labeller	Model	Accuracy	F_1-score
Running time	Decision tree	0.2564	0.2007
	Random forest	0.1795	0.1364
	Naive Bayes	0.2051	0.1424
	SVM	0.2308	0.1616
	Neural network	0.1282	0.1576
Total cost	Decision tree	0.1282	0.0570
	Random forest	0.1538	0.0829
	Naive Bayes	0.1538	0.1120
	SVM	0.0769	0.0318
	Neural network	0.2051	0.1537
Unassigned legs	tree	0.1795	0.0811
	Random forest	0.1282	0.0444
	Naive Bayes	0.1026	0.0636
	SVM	0.1282	0.0815
	Neural network	0.1282	0.0978
Combination	Decision tree	0.2821	0.2339
	Random forest	0.2051	0.1645
	Naive Bayes	0.2308	0.1924
	SVM	0.2564	0.1896
	Neural network	0.1795	0.1537

Table A.18: Prediction results of machine learning models using the 120 problem instances on 39 unseen problem instances - Multi–class classification.

Labeller	**Model**	**Transformation**	**(1 - Hamming loss)**	**F_1-score**
Running time	Decision Tree	Binary relevance	0.7777	0.0399
		Classifier chain	0.8062	0.0000
		Label powerset	0.8091	0.1792
	Random Forest	Binary relevance	0.8547	0.0373
		Classifier chain	0.8575	0.0000
		Label powerset	0.8290	0.1752
	Naive Bayes	Binary relevance	0.8319	0.1639
		Classifier chain	0.8433	0.1183
		Label powerset	0.8347	0.1221
	SVM	Binary relevance	0.7065	0.1596
		Classifier chain	0.7806	0.1386
		Label powerset	0.8233	0.1611
	Neural network	-	0.8063	0.1002
Total cost	Decision Tree	Binary relevance	0.6381	0.4387
		Classifier chain	0.5897	0.3685
		Label powerset	0.5099	0.4912
	Random Forest	Binary relevance	0.6125	0.4509
		Classifier chain	0.5811	0.3986
		Label powerset	0.4188	0.4302
	Naive Bayes	Binary relevance	0.5754	0.4441
		Classifier chain	0.5783	0.4366
		Label powerset	0.4501	0.4476
	SVM	Binary relevance	0.2877	0.4322
		Classifier chain	0.2905	0.4363
		Label powerset	0.2763	0.4485
	Neural network	-	0.7236	0.0810

Table A.19: Prediction results of machine learning models for the running time and cost labeller using the 120 problem instances on 39 unseen problem instances - Multi–label classification.

Labeller	**Model**	**Transformation**	**(1 - Hamming loss)**	**F_1-score**
Unassigned legs	Decision Tree	Binary relevance	0.6467	0.7494
		Classifier chain	0.7122	0.8206
		Label powerset	0.6752	0.7558
	Random Forest	Binary relevance	0.7635	0.8643
		Classifier chain	0.7521	0.8561
		Label powerset	0.8091	0.8898
	Naive Bayes	Binary relevance	0.6980	0.7718
		Classifier chain	0.6866	0.7549
		Label powerset	0.7293	0.8569
	SVM	Binary relevance	0.6809	0.8207
		Classifier chain	0.6894	0.8271
		Label powerset	0.7008	0.8369
	Neural network	-	0.3447	0.1905
Combination	Decision Tree	Binary relevance	0.7179	0.1342
		Classifier chain	0.6923	0.1720
		Label powerset	0.8034	0.1162
	Random Forest	Binary relevance	0.8575	0.0755
		Classifier chain	0.8660	0.0353
		Label powerset	0.8376	0.2231
	Naive Bayes	Binary relevance	0.8404	0.1880
		Classifier chain	0.8575	0.1446
		Label powerset	0.8347	0.1446
	SVM	Binary relevance	0.7008	0.1234
		Classifier chain	0.7777	0.1158
		Label powerset	0.8005	0.1171
	Neural network	-	0.8177	0.1354

Table A.20: Prediction results of machine learning models for the unassigned leg and combination labeller using the 120 problem instances on 39 unseen problem instances - Multi–label classification.