A Certainty-Based Approach for Dynamic Hierarchical Classification of Product Order Satisfaction

Thomas Brink[a], Jim Leferink op Reinink[a], Mathilde Tans[a], Lourens Vale[a]

[a]*Erasmus University Rotterdam, P.O. Box 1738, 3000 DR, Rotterdam, the Netherlands*

Abstract

Online retailers collaborate more and more with partners to sell products via their platform, making it increasingly important to preserve platform quality. In this paper, we predict the quality of matches between sellers and customers associated with product orders, where we face a trade-off between the accuracy and timeliness of our predictions. To deal with this trade-off, we introduce the Hierarchical Classification Over Time (HCOT) algorithm, which dynamically classifies product orders using top-down, non-mandatory leaf-node prediction. To enforce a blocking approach with respect to the tree-based class hierarchy, we introduce the Certainty-based Automated Thresholds (CAT) algorithm, which automatically computes optimal thresholds at each node in the hierarchy. The resulting CAT-HCOT algorithm has the ability to provide both accurate and timely predictions while specifically tailoring the hierarchical classification approach to the domain of customer satisfaction in online retailing. CAT-HCOT obtains a predictive accuracy of 94%. In terms of timeliness, CAT-HCOT classifies 40% of product orders on the order date itself, 80% of product orders within five days after the order date, and 100% of product orders after 10 days. In contrast, a static classification approach is unable to capture the accuracy vs. timeliness trade-off. Also, CAT-HCOT outperforms a flat, one-level hierarchy baseline method in terms of hierarchical precision and recall scores.

Keywords: Dynamic Hierarchical Classification, Non-Mandatory Leaf-Node Prediction, Blocking Approach, Threshold Automation, Accuracy, Timeliness, Customer Satisfaction, Online Retailing

Availability: `https://github.com/thomas-brink/Seminar-QM-BA`

Email addresses: `455893tb@student.eur.nl` (Thomas Brink), `452358jl@student.eur.nl` (Jim Leferink op Reinink), `446685mt@student.eur.nl` (Mathilde Tans), `447398lv@student.eur.nl` (Lourens Vale)

Table of Contents

1. Introduction

Since the late 1990s and especially during the COVID-19 pandemic, online shopping has gained popularity as more and more consumers buy increasingly diversified products online [1, 2]. As a result, online retailers often collaborate with partners to sell products via their platform, making it increasingly important to preserve the platform quality. That is, a platform's partners must be reliable and deliver products of high quality to customers in order to maintain the platform's reputation. Therefore, online retailers want the match between seller, platform, and customer to be as good as possible. A *match* is constituted when a customer orders a seller's product via the online retailer's platform[1]. We define the *match quality* of a product order as the extent to which customers are satisfied with the order. A product order is a *Happy Match* if there is no cancellation, the product is delivered on time, there is no customer case, and there is no return. However, if any of these aspects goes wrong (e.g., the product is not delivered on time), then the order is an *Unhappy Match*. Depending on the type and number of aspects that go wrong, an Unhappy Match can be classified as a Mildly, Medium, or Heavily Unhappy Match. Finally, if not all information on product delivery is known and no other aspects go wrong, the order is labelled as an *Unknown Match*. The split of an order's match quality into the Happy, Unknown, and Unhappy classes is defined as the *general match classification*, while the split of match quality into the Happy, Unknown, and three gradations of Unhappy classes is defined as the *detailed match classification*. Whereas assessing the match quality of product orders usually takes multiple weeks, ideally, online retailers should be able to detect the match quality of an order directly on or shortly after the order date. For example, if a partner at the platform consistently delivers products that are being returned, the retailer wants to flag this partner as quickly as possible and, if necessary, take measures to preserve platform quality. Therefore, in this paper, we investigate the following research question:

How can we accurately predict the match quality of product orders shortly after the orders have been placed?

To answer this question, we predict the match quality of product orders for a large retailer in the Netherlands at different points in time after the order was placed and evaluate the quality and timeliness of our predictions. We refer to the labels that are assigned to product orders based on match quality simply as *match labels*. We specifically focus on assigning such match labels to product orders in a fast and accurate way, such that the online retailer is able to detect any problems quickly and confidently. However, we do not focus on the drivers of those match labels, that is, the reasons why a match has a particular match label. These drivers, which we refer to as *match determinants*, include cancellations, late deliveries, customer cases, and returns. This is a different research area that is relevant for the retailer in case a partner consistently generates unhappy matches and the retailer needs to find out why.

Clearly, the longer the period between the order date and the time of prediction, the more information will be available on the match determinants, and the more accurate we expect our prediction to be. On the

[1] Appendix A provides a comprehensive list of definitions.

other hand, the longer the period between the order date and the time of prediction, the less actionable our prediction will be for the retailer. For example, if a partner consistently delivers products for which customer cases are started, the retailer should know as soon as possible about this issue. We refer to this aspect as the *timeliness* of our predictions. Hence, overall, we face a trade-off between accuracy and timeliness.

In this paper, we deal with this trade-off by developing our own hierarchical classification algorithm to predict the match quality of product orders. In hierarchical classification (HC), the hierarchical structure in the classes to be predicted is explicitly taken into account, which may lead to better classification performance as opposed to a 'flat' classification approach, which does not consider this hierarchical structure. HC organises classes into a tree-based hierarchy, which in our case combines the general and detailed classification systems into one tree structure that captures the full hierarchical structure of the match labels. The leaf nodes in our tree, which form the main focus of our predictions, are represented by the detailed match classification.

To be able to predict these leaf nodes while taking into account the trade-off between accuracy and timeliness, we introduce the so-called Hierarchical Classification Over Time (HCOT) algorithm, which dynamically predicts instances by means of Non-Mandatory Leaf-Node Prediction (NMLNP) in an iterative fashion. That is, when we are not 'certain' enough about classifying a certain instance on a specific day, we block this instance in the hierarchy and try to assign a more detailed label to this instance the next day when, possibly, new information is available. In order to incorporate the concept of certainty, we develop the Certainty-based Automated Thresholds (CAT) algorithm, which automatically computes optimal probability thresholds per node in the hierarchy that an instance to be classified must exceed to continue to a deeper level in the tree. If an instance passes the thresholds on a specific day, we are certain enough about the classification of this instance and thus classify it according to the label corresponding to the leaf node that is reached. By combining the HCOT algorithm with thresholds computed by the CAT algorithm, we develop a classification procedure (CAT-HCOT) that is able to classify product orders with certainty over time, thus capturing the trade-off between accuracy and timeliness.

The relevance of this paper is threefold. First, we are, to the best of our knowledge, the first to apply hierarchical classification over time. Second, we contribute to the existing literature by means of the introduction of the CAT algorithm for dealing with uncertainty. Third, in contrast to most of the existing literature, we apply hierarchical classification to a shallow (3-level) tree with strongly imbalanced data that is tailored to the domain of customer satisfaction in online retailing.

We find that CAT-HCOT provides predictions with a global accuracy of 94%. Moreover, global and class-specific hierarchical precision, recall, and F_1-score almost always achieve values above 90%. Thus, CAT-HCOT exhibits very strong predictive performance. With respect to timeliness, CAT-HCOT is able to classify around 40% of orders on the order date itself, approximately 80% of orders within 5 days after the order date, and 100% of orders after 10 days. All in all, we find that CAT-HCOT nicely captures the accuracy vs. timeliness trade-off by using a certainty-based classification approach, while a static method does not allow for such a balanced trade-off. We further find that CAT-HCOT achieves major improvements over a flat baseline method in terms of hierarchical precision and recall scores on the most specific and least occurring Unhappy classes, which illustrates that taking into account the hierarchical class structure improves predictive performance.

This paper is structured as follows. In Section 2, we review the literature on hierarchical classification, where we focus on the most prominent techniques, the non-mandatory leaf-node prediction blocking approach, hierarchical performance measures, and application domains. In Section 3, we present our data set along with overall descriptive statistics. Consequently, in Section 4, we outline the directions of our research and present the corresponding methods used. In Section 5, we present our results. Finally, in Section 6, we draw our conclusions and discuss interesting directions for future work.

2. Literature Review

This paper aims at predicting the match quality of product orders shortly after the order has been placed, in a way that ensures predictive accuracy and timeliness. As the match labels consist of a finite number of categories, the problem boils down to a (multi-class) classification problem. Specifically, we model this problem as a hierarchical classification problem. In the current section, we start by reviewing the general concept of hierarchical classification, after which we dive deeper into the technique of non-mandatory leaf-node prediction. Consequently, we discuss hierarchical classification performance measures and, last, go through application domains in a hierarchical classification context.

2.1. Hierarchical Classification Techniques

Silla and Freitas [3] discuss the task of hierarchical classification in an excellent survey. They define hierarchical classification (HC) as a classification approach where the classes to be predicted are organised into a class hierarchy—typically a tree or a Direct Acyclic Graph (DAG). According to Freitas and Carvalho [4] and Sun and Lim [5], HC classification methods differ on a number of criteria. The first criterion is the type of hierarchical structure that is used, which can be either a tree or a DAG. In the latter, a node can have more than one parent node, which is not possible in the former. The second criterion is related to the depth of classification in the class hierarchy, which depends on whether or not the algorithm always assigns an instance to a leaf node. More specifically, *mandatory leaf-node prediction* (MLNP) forces the HC algorithm to always assign an instance to a leaf node, whereas *non-mandatory leaf-node prediction* (NMLNP) allows the algorithm to stop the classification in any node at any level of the hierarchy. Finally, the third criterion is related to how the hierarchical structure is explored. Silla and Freitas [3] distinguish three general approaches here, which include flat classification, local classification, and global classification.

The flat classification (FC) approach is the simplest hierarchical classification technique and ignores the class hierarchy—it typically only predicts the classes at the leaf nodes. Therefore, a flat classifier is equivalent to a traditional classifier during training and testing, and does not explore parent-child class relations. However, assignment of an instance to a particular leaf class still implies that this instance also belongs to all ancestor classes based on the original hierarchical structure [3].

The local classification (LC) approach takes into account the class hierarchy by using a local information perspective—that is, it trains classifiers locally: either per node, per parent node, or per level of the class hierarchy. Therefore, Silla and Freitas [3] identify three standard approaches to classifying locally: *local classification per node* (LCN), *local classification per parent node* (LCPN), and *local classification per level* (LCL). We prefer LCPN and LCL over local classification per node (LCN), since this leads to employing fewer classifiers and does not require a specific strategy for selecting training examples [3]. As stated, these three local HC algorithms differ mostly in their training phase, but are very similar in their testing phase. That is, all LC algorithms use a *top-down approach* in their testing phase, which means that for each test instance, they first predict its first-level, most generic class, and then use this predicted class to determine which classes are considered at the second level (i.e., the children of the predicted class at the first level), and so on, recursively, until they make the most specific prediction possible. A disadvantage of this top-down prediction approach is the occurrence of *error propagation*, which means that prediction errors at any class level are propagated downwards through the hierarchy. One way to reduce or stop this error propagation is to use a *blocking approach*, which blocks the process of passing down instances in the hierarchy if the prediction at the current level is not accurate enough according to some criterion. A downside of this blocking approach is that the user might end up with less specific (and therefore less useful) class predictions.

Last, the global classification (GC) approach tackles the problem of hierarchical classification by learning a single global model that naturally takes into account all parent-child relations between the different classes. As a result, the global classifier can assign an instance potentially to any class at every level of the hierarchy. In comparison to local classification, global classification has the advantages that the total size of the classification model is smaller, and the class hierarchy is taken into account in a single run of the classification algorithm. However, GC algorithms are often more complex than LC algorithms.

In hierarchical classification, similar classification methods can be used to those applied to non-hierarchical (multi-class) classification, albeit that different classifiers can be used at different levels. For example, in LCPN, the choice of classification algorithm can be made per parent node. In the existing literature, examples include Naïve Bayes [6], Logistic Regression [7], Support Vector Machine [8], Decision Tree [9], Random Forest [10], and Neural Network [11]. It should, however, be noted that some methods, including NMLNP, require the classifier to provide soft (probabilistic) predictions.

2.2. Blocking Techniques

In NMLNP, the HC algorithm can block the classification at any level of the hierarchy according to some criterion. As a result, the most specific class predicted for any instance can be either at an internal or a leaf node of the hierarchy. The simplest criterion to implement this blocking approach is to use a threshold at each node, and if the probability of the classifier for a test instance at a given node does not exceed this threshold, then the classification is blocked for that instance.

Various methods exist to determine the thresholds at the nodes in the hierarchy. The simplest approach is to set one user-specified threshold that is the same for all nodes in the hierarchy, such that instances are blocked if their probability for a given node does not exceed this threshold. Another approach is to use multiplicative thresholds [8], which compute the product of classifier probabilities over hierarchical levels and then compare this product to a user-specified threshold. Last, procedures for automated threshold determination exist. These procedures usually optimise some criterion in order to find the best threshold for any given node. For example, Andres et al. [12] minimise the fraction of misclassified instances, Chen et al. [13] optimise accuracy, and Ceci and Malerba [14] and Addis et al. [15] maximise the F_1-score. Last, Ceci and Malerba [16] have proposed an advancement of their earlier automated threshold algorithm. In this algorithm, instead of maximising the F_1-score, they find the threshold that minimises the dissimilarity-based tree distance between the predicted class and the true class for each instance.

2.3. Hierarchical Performance Measures

In their systematic analysis of performance measures for classification tasks, Sokolova and Lapalme [17] compare measures for binary, multi-class, multi-label, and hierarchical classification problems. They define *hierarchical performance measures* of predictive performance of a HC algorithm that explicitly incorporate the class hierarchy. Costa et al. [18] discuss several distance-based, depth-dependent, and semantics-based hierarchical performance measures, which make use of differences and similarities between classes. These measures often depend on choices specified by the user, which makes them rather subjective. Therefore, Sokolova and Lapalme [17] and Silla and Freitas [3] recommend using hierarchy-based measures instead, which use the concepts of ancestral and descendent classes. Kiritchenko et al. [19] suggest hierarchical extensions of 'flat' performance measures. More specifically, they define hierarchical precision (hP), hierarchical recall (hR), and the hierarchical F_1-score (hF_1) as the hierarchical extensions of precision, recall, and F_1-score, respectively. The big advantage of these hierarchical performance measures is that they can be effectively applied to any HC scenario—whether the problem is tree-structured, DAG-structured, MLNP, or NMLNP.

In the special case of HC algorithms with NMLNP, the application of the hP, hR, and hF_1 measures may be more complex [3]. That is, generalisation and specialisation errors complicate the evaluation of the HC algorithm. As a consequence of NMLNP, for any test instance, the predicted most specific class can be more generic (generalisation error) or more specific (specialisation error) than the true most specific known class. Basically, the hF_1 measure naturally penalises both generalisation and specialisation errors. However, in certain applications, over-generalisation or over-specialisation may not be considered as errors, in which case the hP and hR need to be modified [3].

Another complexity resulting from NMLNP is the blocking problem, which says that during the top-down process of classification of a test instance, the classifier may block the instance from passing down the hierarchy. This means that higher-level classifiers may block certain instances from being classified into their more specific true classes. Sun et al. [20] propose a measure to keep track of the proportion of test instances that is blocked by a certain classifier, despite belonging to a more specific class in the hierarchy. The authors define this measure as the *blocking factor* for a certain classifier.

2.4. Applications of Hierarchical Classification

Most research on hierarchical classification focuses on the area of text categorisation, which lends itself well for the application of class hierarchies. Practically any type of electronic text document can be organised into a hierarchy, and due to the increasing number of electronic documents, the potential use of hierarchical classification is obvious [4]. Besides text categorisation, hierarchical classification has also been applied in the areas of bioinformatics, music genre classification, phoneme classification, and emotional speech classification [3]. However, the use of hierarchical classification is certainly not limited to these areas. In this paper, we apply hierarchical classification in the area of customer satisfaction in online retailing, which is, to the best of our knowledge, a rather novel area. One paper that takes a similar approach is written by Vandic et al. [6], who propose a framework for hierarchical product classification in the area of e-commerce.

The structure of the class hierarchy logically depends on the application at hand. In most applications, especially in text categorisation and bioinformatics, the class hierarchy is quite large in terms of the number of nodes—internal nodes and leaf nodes. Silla and Freitas [3] report that most hierarchical classification approaches clearly outperform flat classification approaches, because the former explicitly accounts for the structure in the class hierarchy. However, one can imagine that this predictive advantage becomes less pronounced as the class hierarchy becomes smaller, thereby approaching the most simple flat hierarchy. In this research, we use a rather small class hierarchy, consisting of only seven nodes (internal and leaf), whereas the vast majority of class hierarchies in existing applications consist of ten or (much) more nodes [3].

As for the applications of NMLNP, the choice of blocking instances from reaching leaf nodes in the hierarchy often rests on instances not belonging to a leaf node, but rather to an internal class, e.g., in document text classification [14]. However, in our approach, NMLNP plays a different role: that of ensuring 'certainty' of predictions over time. Namely, if an instance gets blocked at a certain point in time, we try to predict it the next day, making use of the newly available information. Although making predictions over time has been subject to previous research [21], applying this concept to hierarchical classification has, to the best of our knowledge, not yet been investigated.

Lastly, our data is skewed across classes, which means that we face an imbalanced data problem in our hierarchical classification task. A common approach to deal with imbalanced data in classification is that of cost-sensitive learning, where larger costs are assigned to misclassifications of less represented classes [22]. As Zheng and Zhao [23] show, such cost-sensitive approaches can drastically improve the performance of hierarchical classification algorithms. The simplest approach herein is to assign class weights inversely proportional to the rate of occurrence in the data set, which is the approach we will take in this research.

3. Data

In this section, we discuss how we clean and validate our data, in addition to presenting important descriptive statistics. Our data set contains roughly 4.78 million observations on product orders at a large online retailer in the Netherlands from January 2019 to December 2020.

3.1. Data Procedures

As the data set contains some noisy observations, we first need to clean the data set. Noisy observations include product orders with conflicting characteristics, such as a return before the order has been placed, or a larger number of items returned than ordered. In nearly all cases, we decide to identify those noisy observations and remove them from the data set, since we want to train our model only on sensible observations. Only in the case of product orders for which more items are returned than originally ordered, we decide to manually modify those observation instead of removing them: we replace the quantity returned by filling in the quantity ordered. Section B.1 presents a complete description of the data cleaning process.

After data cleaning, we validate the data: using the retailer's business logic, we check whether match determinants and match labels are correctly encoded for product orders. For example, we check whether product orders that are cancelled within 10 days after the order date indeed are encoded as having a cancellation. Also, based on the retailer's match classification tree in Appendix C, we check whether product orders have been assigned the correct match label based on their match determinants. Section B.2 presents a complete description of the data validation process.

Last, in order to make our data set well-suited for prediction models, we select, transform, and create (new) variables. For example, we create dynamic product- and seller-related performance variables that indicate a product's number of units sold or a seller's average number of orders per day, taking into account the most recent information up until the time of prediction. We also create dummy variables corresponding to the match determinants, which indicate whether or not the information on the match determinant is known to the retailer. These dummy variables are also dynamic, because they change value from 0 to 1 once new information on a match determinant becomes available. For example, we create a dummy variable for product delivery, which indicates whether it is known at some point in time that a product was delivered late (regardless of how late it was delivered with respect to the order date). If the information on late product delivery becomes available 2 days after the order date, the dummy variable takes value 0 in the first 2 days after the order date, and then takes value 1 as soon as the information is known. If the product is delivered on time, there is no late delivery at all and the dummy variable keeps value 0 over time.

Once all information on the match determinants corresponding to a product order is known, one can determine the corresponding match label with respect to the decision tree in Appendix C. Because of this property, the match determinant dummy variables can be regarded as 'near-perfect predictors' for the match labels. Overall, as more and more days after the order date pass, more information on the match determinants is known, which results in more accurate predictions. As a result of this construction, on each point in time

Table 1
Distribution of product orders over the match labels.

Match label		Count	Percentage
Happy		2,688,910	56.29%
Unhappy	Mildly Unhappy	413,946	8.67%
	Medium Unhappy	95,576	2.00%
	Heavily Unhappy	60,699	1.27%
	Total	570,221	11.94%
Unknown		1,517,711	31.77%
Grand total		4,776,842	100%

after the order date, we can train a prediction model with the same (dynamic) variables, where the dynamic variables change over time as more information on the match determinants becomes available. Section B.3 presents a complete description of the selection of the final set of variables. In addition, Appendix D and Appendix E provide lists of the variables contained in the original data set and in our models respectively.

3.2. Descriptive Statistics

In this section, we discuss the most important descriptive statistics of our (cleaned) data set. Table 1 shows the distribution of product order observations over the match labels, according to the general and the detailed classification system. Clearly, the majority of product orders is a Happy Match (56%), about one-third is an Unknown Match (32%), and the minority is an Unhappy Match (12%). Using the detailed classification system, we find that the majority of unhappy product orders is a Mildly Unhappy Match (9%), and that a small fraction of all product orders is a Medium Unhappy Match (2%) or a Heavily Unhappy Match (1%). Thus, the data is obviously skewed with respect to the different match labels. Hence, in our classification task, we face an imbalanced data problem.

Table 2 gives us more insight in the values of the match determinant variables—that is, the occurrences of cancellations, late deliveries, customer cases, and returns. We find that 0.5% of product orders is cancelled, 3.4% of orders has a customer case related to it, and 5.9% of orders is returned. In addition, 3.5% of product orders is delivered late, while roughly 36% of product deliveries is unknown.

The information on cancellations, deliveries, customer cases, and returns is typically unknown on order date itself and only becomes available in the period afterwards. For example, the online retailer allows

Table 2
Occurrence of match determinants as percentage of total number of product orders.

	No cancellation	On-time delivery	No customer case	No return
True	99.50%	60.75%	96.56%	94.12%
False	0.50%	3.47%	3.44%	5.88%
Null	N.A.	35.78%	N.A.	N.A.

customers to start a case or a return anywhere in the 30 days after the order date. Similarly, a product can be cancelled within 10 days after the order date. In the end, the online retailer finalises product order matches only when 30 days after the order date have passed. Figure 1 shows the percentage of product orders that has information available on a specific match determinant per day after the order date. Particularly for cancellations and deliveries, most data already becomes available in the first five days after the order has been placed. For customer cases and returns, the vast majority of information is available within 10 days after the order date. For all match determinants, very little extra information becomes available in the 15 last days of the maximum period of 30 days. Therefore, in this research, we use a time window of 11 days to predict match labels: that is, on the order date itself (day 0) and in the 10 days afterwards (day 1 to 10).

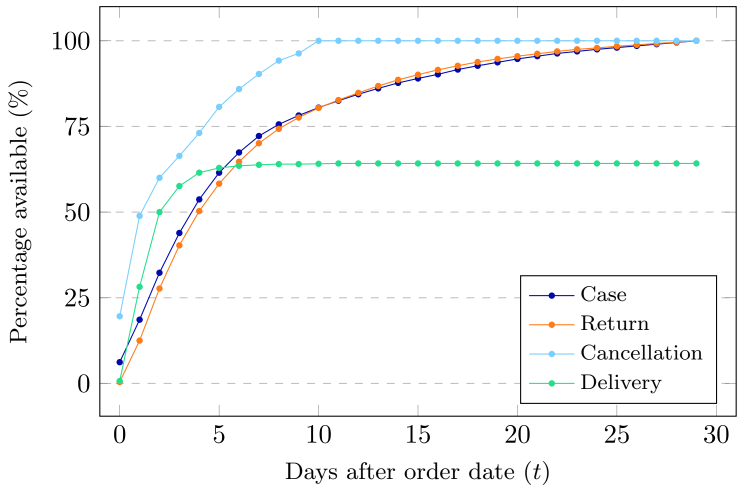


Figure 1. *Availability of match determinant information 0 to 30 days after the order date. The amount of information that is known on product delivery never reaches 100% due to the occurrence of unknown deliveries.*

4. Methodology

In this section, we introduce our hierarchical classification approach that classifies instances over time using an automated threshold procedure. We first provide some general notation and definitions, after which we present the structure of the tree-based class hierarchy. Next, we discuss the main components of the Hierarchical Classification Over Time (HCOT) algorithm as well as model and classifier selection. Afterwards, we present the Certainty-based Automated Thresholds (CAT) algorithm, discuss the testing procedure using the CAT-HCOT algorithm, and evaluate our model. Last, we propose two baseline methods which we use to compare our model, and discuss the software we use to implement our methods.

4.1. Notation and Definitions

In this research, we use a tree-based structure to organise our classes into a hierarchy. Following the definitions provided by Silla and Freitas [3], a *tree-based class hierarchy* is defined as $\mathcal{H} = (\mathcal{N}, \prec)$, where $\mathcal{N}$ denotes the finite set of all nodes in the hierarchy (or classes in the application domain) and '$\prec$' represents the 'is-a' (or 'subclass-of') relation. The 'is-a' relation is characterised by the following properties:

1. **Anti-reflexivity**. $\forall p \in \mathcal{N}$, $p \nprec p$.

2. **Asymmetry**. $\forall p, q \in \mathcal{N}$, if $p \prec q$, then $q \nprec p$.

3. **Transitivity**. $\forall p, q, s \in \mathcal{N}$, if $p \prec q$ and $q \prec s$, then $p \prec s$.

We can classify a node $m \in \mathcal{N}$ according to different tree-based terminology systems:

- **Root/standard**. We define R as the root node of the hierarchy and $\mathcal{S} \subset \mathcal{N}$ as the set of all standard nodes, where $\mathcal{S} = \mathcal{N} \setminus \{R\}$.

- **Internal/leaf**. We define $\mathcal{I} \subset \mathcal{N}$ as the set of internal nodes and $\mathcal{L} \subset \mathcal{N}$ as the set of leaf (or external) nodes, where $\mathcal{I} \cup \mathcal{L} = \mathcal{N}$.

- **Parent/child**. We define $\mathcal{C}_m$ as the set of all child nodes of node m and $\mathcal{S}_m$ as the set of all sibling nodes of node m. Further, we define p_m as the parent node of node m.

- **Ancestor/descendant.** We define $\mathcal{A}(m)$ as the set containing node $m \in \mathcal{N}$ and all its ancestor nodes, and $\mathcal{D}(m)$ as the set containing node $m \in \mathcal{N}$ and all its descendant nodes.

A training instance is defined as $u \in \mathcal{U}$, where $\mathcal{U}$ is the set of all training instances and $|\mathcal{U}| = M$. We assign a test instance $r \in \mathcal{R}$ to a class with respect to the hierarchy, where $\mathcal{R}$ is the set of all test instances and $|\mathcal{R}| = N$. By definition of the tree-based hierarchical structure, an instance assigned to a particular class naturally belongs to all its ancestor classes. We define the i-th test instance by $r_i = (\mathbf{x}_i, l_i) \in \mathcal{R}$, where $\mathbf{x}_i \in \mathbb{R}^D$ is the vector of values on the D model features and $l_i \in \mathcal{L}$ is the true label of instance i.

4.2. Tree-Based Class Hierarchy Structure

In order to use our hierarchical classification algorithm, we define a tree-based class hierarchy structure based on business logic. We do not use a Directed Acyclic Graph (DAG) structure, since, as a consequence of business logic, any node in the tree cannot have more than one parent node. Figure 2 shows the 3-level tree representing the class hierarchy: first, product orders are classified either as Unknown (<0>) or Known (<1>). If the order is classified as Unknown, it means that we expect all service criteria to be met, but it is unknown whether or not the product is delivered on time. If the order is classified as Known, it means that we can predict the match quality of the order as either Happy (<1.1>) or Unhappy (<1.2>). The order is classified as Happy when we expect all service criteria to be met. In contrast, the order is classified as Unhappy when at least one service criterion is not met. If the order is classified as Unhappy, we can assign the order to a more specific Unhappy class, which reflects the degree of unhappiness associated with the order. Based on business logic, we classify the order either as Mildly Unhappy (<1.2.1>), Medium Unhappy (<1.2.2>), or Heavily Unhappy (<1.2.3>). Since the exact business rules for determining the degree of unhappiness are retailer-specific and rather complex, we do not discuss them in this paper.

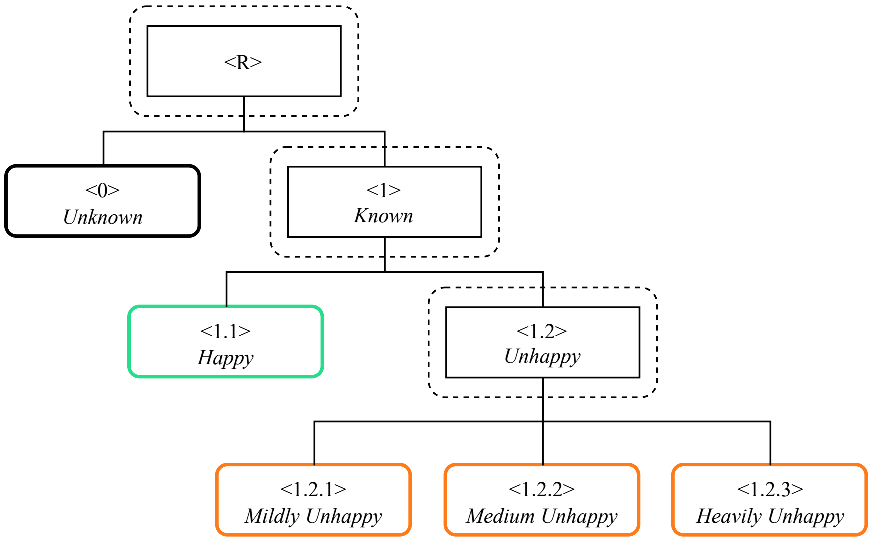


Figure 2. *Pre-defined class hierarchy based on business logic.*

4.3. Hierarchical Classification Over Time (HCOT)

In our problem setting, we face a trade-off between accuracy and timeliness. On the one hand, product orders should be classified as accurately as possible to ensure reliable predictions, but, on the other hand, orders should be classified as soon as possible to ensure actionable predictions. To handle this trade-off, we use hierarchical classification of order satisfaction for $t = 0, 1, \ldots, 10$ days after the order date. We refer to this approach as *Hierarchical Classification Over Time* (HCOT). After 10 days, each product order should be assigned to a label which corresponds to a leaf node $l \in \mathcal{L}$. Since any instance can be assigned only one path of predicted labels, HCOT makes use of *single path prediction* (SPP). For example, an order can follow the path <R> $\rightarrow$ <1> $\rightarrow$ <1.1>. The choice for SPP follows from business logic, since an order can have at most one label per hierarchy level. For example, an order cannot be classified as both Happy (<1.1>) and Unhappy (<1.2>). The HCOT algorithm consists of three main procedures: training, testing, and blocking.

Training approach. In HCOT, we train the hierarchy $\mathcal{H}_t$ independently per day using the available information on day t. This implies that each 'daily' hierarchy $\mathcal{H}_t$ is trained on the same (number of) instances and the same features, while the information contained in those features may change over time as more information about the match determinants comes in. For each daily hierarchy, we make use of *local classification per parent node* (LCPN), which means that we train a classifier at each parent node. The parent nodes in our class hierarchy (indicated by the dotted lines in Figure 2) include the root node <R>, node <1> (Known), and node <1.2> (Unhappy). At <R> and <1>, we need a binary classifier, as both nodes have only two child nodes. At <1.2> we need a multi-class classifier, as this node has three child classes. Since each level has only one parent node in our hierarchy, we note that LCPN is equivalent to local classification per level (LCL). We train the classifiers using the true training labels. That is, we train the classifier at <R> with all training data, the classifier at <1> with all instances that are truly Known, and the classifier at <1.2> with all instances that are truly Unhappy. In Section 4.4, we elaborate on the way we select our classifier methods and optimise the corresponding models.

Testing approach. Next, by applying the trained classifiers, we can assign test instances $r_i \in \mathcal{R}$ using HCOT. We do so by means of iterative top-down class-prediction over time with non-mandatory leaf-node prediction (NMLNP), which implies that a test instance is not necessarily assigned to a leaf node $m \in \mathcal{L}$ (e.g., <1.2.1>) on a certain day, but can also get blocked at an internal node $m \in \mathcal{I}$ (e.g., <1.2>). In general, for day $t = 0, 1, \ldots, 9$, an instance first goes through the hierarchy $\mathcal{H}_t$ and, if it is not classified into a leaf node, it is sent to $\mathcal{H}_{t+1}$. Each day, starting from the root node <R>, we first try to assign an instance to a label at the first level and then use that label to determine whether we continue to assign the instance to a label at a lower level. Hence, although an instance can be classified at an internal node on a given day, the next day we start the classification process again from the top of the tree. This overcomes the issue of *error-propagation*, which means that prediction error at any class level is propagated downwards with respect to the hierarchy. This is a clear disadvantage of the top-down class-prediction procedure. By re-starting the classification process again at the top of a tree on any given day, we avoid propagation of possible errors made on the previous day. If an instance is still not classified to a leaf node on day 9, we make sure these instances are necessarily classified into a leaf node on day 10. In this way, we make sure all instances are assigned to a leaf node within 10 days after the order date.

Blocking approach. We use a blocking approach to handle NMLNP and determine whether a test instance continues to the next level in the tree on a certain day. In a blocking approach, the process of passing down test instances in the hierarchy is blocked if the prediction at a given level is not certain enough according to some criterion. The simplest criterion is a threshold at each node $m \in \mathcal{S}$: if the probability for an instance to reach a given child node does not exceed its threshold, the classification is blocked at the corresponding parent node. When an instance does not get blocked at any of the parent nodes, it reaches a leaf node and gets classified. We refer to this process as *leaf-classification.* Since this instance passes all thresholds on its path, we are confident enough about our classification and thus expect to end up with accurate predictions. In addition, since our testing approach implies that all instances go through the hierarchy in an iterative fashion (over time), we make sure to classify our instance as soon as possible. Thus, the blocking approach in HCOT allows us to face the trade-off between accuracy and timeliness. In Section 4.5, we propose and discuss the algorithm for computing certainty-based automated thresholds.

In total, HCOT works as follows. Each day, starting on the day of ordering $t = 0$, an instance can either be assigned to a leaf class or be blocked at one of the parent (internal) nodes. If an instance gets classified into a leaf class, we consider its classification as final and we do not try to classify the instance again the next day, as we assume that a retailer wants to immediately act on a predicted match quality. Hence, from a business perspective, there is no need to classify a leaf-classified instance again on the next day. From a methodological perspective, it is possible to classify leaf-classified instances again on the next day, but we do not expect the classification of such instances to improve drastically over time. On the other hand, if an

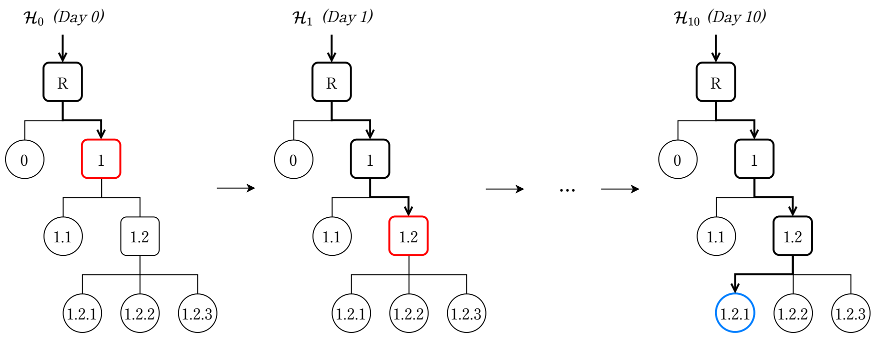


Figure 3. *Graphical representation of Hierarchical Classification Over Time (HCOT) with $\mathcal{H}_0, \ldots, \mathcal{H}_{10}$. Red colour indicates blocking at the corresponding parent node. Blue colour indicates leaf-node classification.*

instance gets blocked at an internal node, there is insufficient information available for the instance to be assigned to a leaf class and, as a result, we try to classify the same instance again the next day, starting all over again from the root node of the hierarchy. Still, by assigning internal labels, HCOT provides actionable insights even if there is not enough certainty of providing leaf-node predictions. Because we want all instances to be classified within 10 days, we drop our blocking approach from HCOT on day 10, so that all not-yet classified instances get assigned a leaf class on day 10 (using *mandatory* leaf-node prediction).

Figure 3 graphically illustrates the process of hierarchical classification over time with an example. On day 0, an instances gets assigned to node <1>, but is blocked from further hierarchical classification due to a lack of certainty. On day 1, we try to leaf-classify the instance again. This time, the instance gets assigned to node <1.2>, but is again blocked from further hierarchical classification. In this specific example, the instance gets blocked up until day 9, such that we need to use mandatory leaf-node prediction on day 10 to ensure that the instance is eventually assigned to a leaf class.

4.4. Classifier Selection and Model Optimisation

In HCOT, we construct our 'optimal' hierarchical classifier by optimising each daily hierarchy ($\mathcal{H}_t$) independently. To do so, there are three components we take into account: classifier selection, imbalanced data, and model tuning. All three components are applied to daily hierarchies, which are eventually put together over time to form our HCOT classifier. First, we consider the process of selecting classifiers. As noted, we train each $\mathcal{H}_t$ using LCPN, implying that we train classifiers per parent node. Therefore, different classifiers can be used at different parent nodes in the hierarchy. In addition to providing flexibility in classifier selection, this could also lead to improved predictions. Namely, different algorithms may prove to be better able to differentiate between observations in different 'regions' (parent-child splits) of our data. Therefore, each day, we optimise the combination between classifiers at the different parent nodes. To optimise such combinations, we perform an exhaustive search across all possibilities. Since our tree consists of three parent nodes (see Figure 2), considering K classifiers leads to evaluating $C = K^3$ combinations. To select our classification methods, it must be noted that our blocking approach requires probabilistic class predictions. The first classifier we choose is the Random Forest (RF) classifier, which, through averaging classifications over various decision trees, is known to possess strong predictive performance and provide specific class probabilities for instances. The second classifier we select is Logistic Regression (LR), which is a probabilistic classification method by nature. Taking these $K = 2$ classifiers, we evaluate $C = 2^3 = 8$ combinations on a daily basis. Clearly, C explodes with K, such that we decide to keep K small. However, in case more classifiers would be selected, we could use (greedy) heuristics to overcome the computational difficulty of an exhaustive search and still arrive at a near-optimal solution. Appendix F provides a detailed description of the RF and LR classification methods.

Second, in training the combinations of the above two classification methods for each $t = 0, 1, \ldots, 10$, we face an imbalanced data problem. That is, the distribution of data across the child nodes of all parent nodes in our hierarchy is skewed. If we define the class distribution as the relative occurrence of instances per child node in the data set, we find that the class distributions are roughly 2:1, 5:1, and 7:2:1, for the root parent node <R>, parent node <1>, and parent node <1.2>, respectively. Class imbalance may be a problem, as it biases classifiers towards predicting the most occurring classes [23]. In our application, we value instances from each class equally important and, therefore, apply a cost-sensitive imbalance approach at each parent node. This approach assigns larger weights to classifying instances from less prevalent classes. That is, assigned weights to classes are inversely proportional to the rate of occurrence of these classes. In this way, classifiers at each parent node are less biased towards assigning the most prevalent labels.

Third, having selected the combinations of classifiers to consider and accounting for imbalanced data, we turn to tuning each $\mathcal{H}_t$. This implies that, for each t, we need to find the best combination of classifiers and the corresponding optimal hyperparameters. To do so, we divide our training data into a training set and a validation set. For each combination of classifiers, we then turn to Bayesian hyperparameter optimisation (Tree-structured Parzen Estimation). Training our hierarchy using LCPN, we evaluate the validation performance by means of top-down mandatory leaf-node prediction, where we aim to maximise the validation hierarchical F_1-score (hF_1), which is defined in Appendix G. Having obtained a set of optimal hyperparameters for all eight classifier combinations, we compare the hF_1-scores for all these combinations and select the one with the best performance. Applying this procedure, we obtain daily optimal hierarchies. The process of Bayesian hyperparameter optimisation is elaborated on in Appendix H.

To illustrate the classifier selection and model optimisation procedure, we visualise our train-validation splits in Figure 4. First, we split the data into train (80%) and test (20%) using a time series split, so that all test data comes after train data in terms of order date (Step 1 and 2), which is most representative of the real-life situation. Next, we take a subsample of the train data containing 1.1 million instances. As some of our engineered features. i.e., our *historic* variables, require some time to stabilise, we decide to burn in the first 10% of observations, leaving us with a sample of 1 million observations to train and validate on (Step 3). After that, we split this sample into train (80%) and validation (20%), again using a time series split (Step 4). On this split, we apply the previously described procedure of finding the best daily combination of classifiers with optimal hyperparameters to this train-validation split to find optimal daily hierarchies.

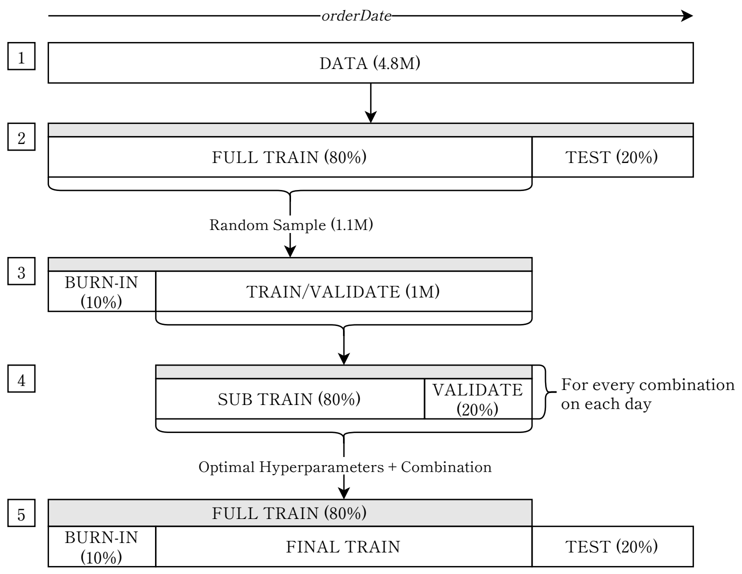


Figure 4. *Procedure for training, validation, and testing. Between brackets, we indicate the number of observations or the percentage of a certain set of observations. Capital M means 'million'.*

4.5. Certainty-Based Automated Thresholds (CAT)

In this section, we present the Certainty-based Automated Thresholds (CAT) algorithm, which is a blocking approach for handling NMLNP. In a given hierarchy, CAT independently computes thresholds for all child nodes $m \in \mathcal{S}$. When dealing with a time component (in HCOT), we have daily hierarchies, such that thresholds will be computed for each child node and each day t independently. The threshold decides whether an instance coming from the parent node of m, p_m, gets accepted at node m. We denote this threshold by $\text{Th}_t(m)$. Since our hierarchy consists of seven child nodes and HCOT has 10 days of non-mandatory leaf-node prediction, applying CAT to HCOT implies computing 10 (days) $\times$ 7 (nodes per day) = 70 thresholds. For a single threshold in hierarchy $\mathcal{H}$, we first obtain the class probabilities of the trained instances based on true data, i.e., only the instances truly belonging to p_m are considered for the threshold of node m, and then use the corresponding set of probabilities to compute the optimal threshold.

First, we introduce the crux of the CAT algorithm. This is the constant hyperparameter α, which we dub the 'certainty' parameter. The hyperparameter is used to locate a lower bound for the optimisation of the thresholds. The certainty α decides the trade-off between accuracy and timeliness. Since there is no mathematical optimum in this trade-off, the certainty represents user preferences where accuracy might be more or less important than timeliness. Hence, the hyperparameter α is user-specified.

Second, we introduce some terminology to formally define the CAT algorithm and context. We denote the set of instances that, during the training phase, is known to belong to parent node p_m as $\text{Training}(p_m)$. The posterior probability that train instance $u_j \in \text{Training}(p_m)$ belongs to node m, as assigned by the classifier at parent node p_m on day t is denoted as $\gamma_{p_m \to m,t}(u_j)$. Next, we define the set with majority vote probabilities, which are the probabilities for node m that exceed the probabilities to go to one of the sibling nodes of m, $m' \in \mathcal{S}_m$. Formally, this is given by

$$\mathcal{Z} = \left\{ \gamma_{p_m \to m,t}(u_j) \mid \gamma_{p_m \to m,t}(u_j) > \gamma_{p_m \to m',t}(u_j), \quad \forall m' \neq m,\, m' \in \mathcal{S}_m, \quad \forall u_j \in \text{Training}(p_m) \right\}. \tag{1}$$

Since instances can only belong to one class in this problem setting, they are assigned to exactly one node. Logically, node m is only presented with majority vote probabilities, where the threshold subsequently decides whether an instance is accepted. Therefore, we train $\text{Th}_t(m)$ solely on instances that have a majority vote probability for node m on day t.

Similarly to Training(p_m), we define Training(m) and Training$(\mathcal{S}_m)$. The threshold should block instances that do not belong to Training(m) and should accept instances that do. To distinguish between instances that should and should not be accepted, we first define the list of majority vote probabilities that should be accepted by the threshold as

$$\zeta_t(m) = \left\{ \gamma_{p_m \to m,t}(u_j) \in \mathcal{Z} \mid u_j \in \text{Training}(m) \right\}. \tag{2}$$

The list of majority vote probabilities that should be accepted to one of the sibling nodes of m, ergo, not to node m, is given by

$$\zeta_t(\neg m) = \left\{ \gamma_{p_m \to m,t}(u_j) \in \mathcal{Z} \mid u_j \in \text{Training}(\mathcal{S}_m) \right\}. \tag{3}$$

Section I.1 includes some visualisation of the sets and their respective relations to aid interpretation.

Now, the certainty parameter α comes in and its role is quite straightforward. The certainty is a number between 0 and 1 and it is used to compute the (100α)-th percentile of $\zeta_t(\neg m)$, denoted by $\zeta_t(\neg m)_\alpha$. The percentile serves as a lower bound on the search space for the optimal threshold. This search space is dubbed the 'allowed search space' $\mathcal{V}$, which is formally defined as

$$\mathcal{V} = \left\{ \gamma_{p_m \to m,t}(u_j) \in \mathcal{Z} \mid \gamma_{p_m \to m,t}(u_j) \geq \zeta_t(\neg m)_\alpha \right\}. \tag{4}$$

Essentially, CAT allows us to be certain that *at least* $(100\alpha)\%$ of $\zeta_t(\neg m)$ will be blocked by $\text{Th}_t(m)$ (for visualisation of this process, see Section I.2). When $\alpha = 0$, no restrictions are placed on the allowed search space. This is likely to yield relatively low thresholds and result in timely classifications. On the other hand, when $\alpha = 1$, we restrict $\mathcal{V}$ to contain probabilities that are only found in $\zeta_t(m)$, which is likely to result in thresholds approaching 1. Consequently, this will yield highly accurate but less timely classifications. Also, note that the relation between α and the relative importance of timeliness and accuracy is not necessarily linear but dependent on the data set.

Subsequently, within $\mathcal{V}$, we optimise threshold $\text{Th}_t(m)$ by looking for the separation point between $\zeta_t(m)$ and $\zeta_t(\neg m)$ that maximises F_1. To do so, CAT requires a classifier that outputs near-continuous probabilities. Otherwise, the instances will be concentrated in the few available probabilities, such that making a distinctive split is impossible. Given a near-continuous classifier, it is not necessary to consider each probability as potential threshold, as the number of distinct probabilities is often extremely high. Hence, we do a grid search using ξ evenly spaced steps within $\mathcal{V}$. The hyperparameter ξ is given as input to CAT. We denote the set containing the ξ grid points as Θ, where each $\theta \in \Theta$ is considered as potential threshold.

For each potential threshold, we determine the number of true positives (TP) and false positives (FP) by calculating the number of posterior probabilities in $\zeta_t(m)$ and $\zeta_t(\neg m)$ that exceed the threshold, respectively. Next, we calculate precision and recall, where the latter uses that the sum of true positives (TP) and false negatives (FN) equals the cardinality of $\zeta_t(m)$. Combining precision and recall, we compute the F_1-measure and select the threshold with the highest F_1 as $\text{Th}_t(m)$. Algorithm 1 presents the complete procedure of CAT. Additionally, Section I.3 includes a visualisation of the algorithm.

The role of α is to pre-select an allowed search space to find our optimal threshold. When α increases, this allowed search space will narrow down towards 1, which will likely lead to a higher threshold. Moreover, when applying CAT to HCOT, we deal with a time component. Over time, our classifier is based on more information, making it easier to separate probabilities. This means that the posterior probabilities gravitate towards 0 and 1. If a probability in $\zeta_t(\neg m)$ moves towards 1, it is more likely to be a majority vote probability. The other way around, if it moves towards 0, it becomes less likely that the probability will remain a majority vote probability. Consequently, the distributions of $\zeta_t(m)$ and $\zeta_t(\neg m)$ are more likely to shift towards 1. In general, this will increase the lower bound of $\mathcal{V}$ and hence, may push $\text{Th}_t(m)$ upward. Whether the threshold increases depends on the distributions of $\zeta_t(m)$ and $\zeta_t(\neg m)$, as well as on α.

Algorithm 1: CAT

```
Data: $\{\gamma_{p_m \to m,t}(u_j) \mid u_j \in \text{Training}(p_m)\}$; $\alpha$; $\xi$
Result: $\text{Th}_t(m)$
1  Initialise the following sets
       $\mathcal{Z} = \{\gamma_{p_m \to m,t}(u_j) \mid \gamma_{p_m \to m,t}(u_j) > \gamma_{p_m \to m',t}(u_j), \quad \forall m' \neq m,\ m' \in \mathcal{S}_m,\ \forall u_j\}$
       $\zeta_t(m) = \{\gamma_{p_m \to m,t}(u_j) \in \mathcal{Z} \mid u_j \in \text{Training}(m)\}$
       $\zeta_t(\neg m) = \{\gamma_{p_m \to m,t}(u_j) \in \mathcal{Z} \mid u_j \in \text{Training}(\mathcal{S}_m)\}$
2  Define the allowed search space, $\mathcal{V} = \{\gamma_{p_m \to m,t}(u_j) \in \mathcal{Z} \mid \gamma_{p_m \to m,t}(u_j) \geq \zeta_t(\neg m)_\alpha\}$
3  Define the set with potential thresholds $\Theta$ as $\xi$ evenly spaced points within $[\min\{\mathcal{V}\}, \max\{\mathcal{V}\}]$
4  Initialise $F_1^* = 0$ and $\text{Th}_t(m) = 0$
5  for $\theta \in \Theta$ do
6      $\text{TP} = |\{\gamma_{p_m \to m,t}(u_j) \in \zeta_t(m) \mid \gamma_{p_m \to m,t}(u_j) \geq \theta\}|$
7      $\text{FP} = |\{\gamma_{p_m \to m,t}(u_j) \in \zeta_t(\neg m) \mid \gamma_{p_m \to m,t}(u_j) \geq \theta\}|$
8      $recall = \frac{\text{TP}}{|\zeta_t(m)|}$;   $precision = \frac{\text{TP}}{\text{TP}+\text{FP}}$;   $F_1 = \frac{2 \times precision \times recall}{precision + recall}$
9      if $F_1 > F_1^*$ then
10         $F_1^* := F_1$
11         $\text{Th}_t(m) := \theta$
```

While constructing CAT, we considered the following alternatives. With respect to the time component as in HCOT, predicting a false negative is not as troublesome as predicting a false positive. Namely, if an instance is blocked, we can try again the next day, but if an instance is accepted and thereby classified to the wrong leaf node, there is no chance to rectify this mistake. While the predictions should be accurate, they should also be timely. Therefore, we solve this issue with CAT, where we use α to reach a certain standard for precision but at the same time try to recall as many instances as possible within the allowed search space. Hence, we optimise F_1 because precision and recall are equally important within $\mathcal{V}$. On the other hand, if one disregards the construction of $\mathcal{V}$, one may also believe precision should be valued over recall. In that case, one could solve this issue by adjusting β in the F_β-measure. For argument's sake, suppose we do. Then, if $\beta = \frac{1}{2}$, we imply that false positives are four times more important than false negatives (see Appendix J for the derivation). There are two problems with this approach: (1) there is no guarantee that this would reach our desired standard for precision, e.g., if $\beta = \frac{1}{2}$ is too high, and (2) if we do reach our desired standard, perhaps with a lower β, we might surpass this level to such an extent that we do not recall as much as possible. Hence, this alternative would not be as effective as CAT and would yield inferior results compared to the CAT algorithm.

Secondly, CAT can be divided into two parts. First we determine the allowed search space, after which we optimise the thresholds. The optimisation of the thresholds is separate from the model optimisation of HCOT. It would have been too computationally expensive to combine these procedures. Next to adding 70 dimensions (10 days $\times$ 7 nodes per day) to the hyperparameter optimisation for the HCOT algorithm, the thresholds would also add interactions between nodes and across days. This changes the optimisation to such a complex system that simultaneous hyperparameter optimisation is not feasible.

Last, the objective of the HCOT algorithm is to optimise the hierarchical F_1-score, following the popular example of Ceci and Malerba [14]. Naturally, the thresholds are optimised according to this measure as well. Alternatively, one might consider optimising the thresholds with respect to the total distance between the true and classified classes. Distance can be defined as, for instance, the weighted average of the edges between classes. This is however not suitable to this problem setting. Such an algorithm, as in Ceci and Malerba [16], is designed for a document classification setting, where some documents should *always* be classified to internal nodes. In contrast, in our case, the label of any instance always corresponds to a leaf node. Therefore, when we would apply Ceci and Malerba [16]'s algorithm, it would favour lower thresholds because it always wants instances to reach the leaf nodes.

4.6. CAT-HCOT

Having obtained the daily best classifier combination and respective optimal hyperparameters as well as a procedure to compute daily optimal thresholds, we can now predict the match quality of orders by combining CAT and HCOT, in short denoted by CAT-HCOT. First, we train the complete HCOT algorithm by independently training each $\mathcal{H}_t$, for $t = 0, 1, \ldots, 10$, with the obtained optimal hyperparameters and best classifier combinations. We do this using the previously explained LCPN approach. Note that for each hierarchy we thus train on the same (number of) instances and the same features, while the information contained in those features may change over time.

Second, we apply CAT with certainty parameter α independently to each $\mathcal{H}_t$ to obtain day- and node-specific thresholds $\text{Th}_t(m)$. Similar to the trained classifiers, the thresholds are computed based on the true training labels and their respective class probabilities. That is, only instances truly belonging to a specific parent node are considered when computing the thresholds belonging to its child nodes.

Third, combining the trained hierarchies and their optimal thresholds, we can start predicting instances in a top-down, non-mandatory leaf-node procedure over time. Algorithm 2 presents this procedure. In short, we only classify instances when we are confident enough about their classification and otherwise try again the next day when (possibly) new information is available. This leads to some instances being classified sooner than other instances, with the overarching goal of balancing accuracy and timeliness. In Appendix K, we provide an example of an instance going through the CAT-HCOT algorithm until it is classified.

In applying these three steps, we consider the full training set containing 80% of data points and burn in its first 10% of instances (sorted by order date) to stabilise our historic variables (Figure 4, Step 5). The withheld test set containing the most recent 20% of data is then run through Algorithm 2 in order to evaluate the overall performance of CAT-HCOT.

```
Algorithm 2: CAT-HCOT
  Data: Set $\mathcal{R}$ of all test instances $r_i = (\mathbf{x}_i, l_i)$ to be classified; Tree-based class hierarchy $\mathcal{H} = (\mathcal{N}, \prec)$;
        Trained hierarchy $\mathcal{H}_t$ at $t \in \{0, 1, \ldots, 10\}$; Thresholds $\text{Th}_t(m)$ for node $m$ at day $t$ from CAT
  Result: Set $\mathcal{P}$ of classified test instances $(\mathbf{x}_i, \hat{l}_i)$, where $\hat{l}_i \in \mathcal{L}$
1 Initialise $\mathcal{P} = \emptyset$ and $t = 0$
2 while $\mathcal{R} \neq \emptyset$ do
3     for $r_i \in \mathcal{R}$ do
4         if $t < 10$ then
5             Top-down, non-mandatory leaf-node prediction with respect to $\mathcal{H}_t$ assigns $\mathbf{x}_i$ a label
              $\hat{l}_i \in \mathcal{I} \cup \mathcal{L}$ if posterior probabilities exceed thresholds $\text{Th}_t(m)$ for the entire path to $\hat{l}_i$
6             if $\hat{l}_i \in \mathcal{L}$ then
7                 $\mathcal{R} := \mathcal{R} \setminus r_i$
8                 $\mathcal{P} := \mathcal{P} \cup (\mathbf{x}_i, \hat{l}_i)$
9         else
10            Top-down, mandatory leaf-node prediction with respect to $\mathcal{H}_{10}$ assigns $\mathbf{x}_i$ a label $\hat{l}_i \in \mathcal{L}$
11            $\mathcal{R} := \mathcal{R} \setminus r_i$
12            $\mathcal{P} := \mathcal{P} \cup (\mathbf{x}_i, \hat{l}_i)$
13    $t := t + 1$
```

4.7. Performance Measures

To evaluate the performance of CAT-HCOT, we use the measure of predictive accuracy alongside hierarchical performance measures as proposed by Kiritchenko et al. [19]. These latter measures are the hierarchical extensions of precision, recall, and F_1-score. Hierarchical performance measures explicitly incorporate the hierarchical path-distance between true and predicted labels, which means that the farther classifications are from the true label with respect to the hierarchy, the heavier the penalty. This property makes hierarchical performance measures preferable over flat performance measures in case of hierarchical classification.

Global predictive accuracy (Acc), *global hierarchical precision* (hP) and *global hierarchical recall* (hR) are defined as follows:

$$Acc = \frac{1}{N}\sum_{i=1}^{N}\left|\mathcal{A}(l_i) \cap \mathcal{A}(\hat{l}_i)\right|, \qquad hP = \frac{\sum_{i=1}^{N}\left|\mathcal{A}(l_i) \cap \mathcal{A}(\hat{l}_i)\right|}{\sum_{i=1}^{N}\left|\mathcal{A}(\hat{l}_i)\right|}, \qquad hR = \frac{\sum_{i=1}^{N}\left|\mathcal{A}(l_i) \cap \mathcal{A}(\hat{l}_i)\right|}{\sum_{i=1}^{N}\left|\mathcal{A}(l_i)\right|}, \tag{5}$$

where l_i and $\hat{l}_i$ denote the true and the predicted label of instance i, respectively, and $\mathcal{A}(l)$ denotes the set containing leaf node $l \in \mathcal{L}$ and all its ancestor nodes except the root node R. We use the term 'global' here, because these performance measures are calculated over all N test instances after the total period of 10 days.

The *global hierarchical F_1-measure* (hF_1) aggregates the hP and hR measures by taking their harmonic mean, and is defined as follows:

$$hF_1 = \frac{2 \cdot hP \cdot hR}{hP + hR}. \tag{6}$$

In the case of flat precision and recall, we can compute class-specific precision and recall. We extend this idea to our hierarchical measures by defining *class-specific hierarchical precision* (hP_c) and *class-specific hierarchical recall* (hR_c), except that we now sum over all predicted instances of class $c \in \mathcal{L}$ in hP_c, and all true instances of class $c \in \mathcal{L}$ in hR_c. We calculate the *class-specific hierarchical $F_{1,c}$-measure*, where we replace hP and hR by their class-specific counterparts hP_c and hR_c.

As our HCOT algorithm classifies instances over time, we do not only want to evaluate the algorithm's predictive performance globally, but also want to evaluate its performance per day. This means that we need daily hierarchical performance measures. Therefore, we define *daily accuracy* (Acc_t), *daily hierarchical precision* (hP_t), and *daily hierarchical recall* (hR_t), where we now sum over (and divide by) all N_t instances that are assigned a (leaf) class $c \in \mathcal{L}$ on day $t \in \{0, 1, \ldots, 10\}$. We calculate the *daily hierarchical $F_{1,t}$-measure*, where we replace hP and hR by their daily counterparts hP_t and hR_t.

Last, we notice that we can combine the class-specific and daily hierarchical performance measures to calculate *class-specific daily hierarchical precision* ($hP_{c,t}$) and *class-specific daily hierarchical recall* ($hR_{c,t}$). As before, these precision and recall measures can be combined to calculate the *class-specific daily hierarchical $F_{1,c,t}$-measure*. In Appendix G, we specify the formulas of all (hierarchical) performance measures.

To evaluate the extent to which instances are blocked at a certain classifier on a parent node, we use the *blocking factor*, which is defined as the fraction of instances that is blocked at a parent node relative to the total number of instances entering the corresponding classifier in the top-down class-prediction approach. The blocking factor gives us insight into the evolution of the blocking process at each parent node over time.

4.8. Baseline Methods

We compare the performance of CAT-HCOT against two baselines: static HCOT and flat CAT-HCOT.

4.8.1. Static HCOT

We first compare the performance of CAT-HCOT against *static HCOT*, which is a hierarchical classification method with mandatory leaf-node prediction (MLNP). More specifically, this baseline method does not use a blocking approach with thresholds, such that, each day, it classifies all instances. When using static HCOT in practice, one would choose a specific day and use the predictions obtained for that day. Considering the accuracy-timeliness trade-off, we expect CAT-HCOT to achieve higher daily accuracy than static HCOT for the first few days, since it only assigns instances to leaf nodes if there is enough confidence to do so. For the last few days, we expect static HCOT to be able to obtain higher daily accuracies, since CAT-HCOT at this point would only need to predict previously blocked and thus more uncertain instances, while static HCOT always classifies all instances on each day. In terms of timeliness, static HCOT will become relatively less timely across days, as CAT-HCOT classifies more and more instances over time.

4.8.2. Flat CAT-HCOT

We also compare the performance of CAT-HCOT against *flat CAT-HCOT*, which is a specific case of CAT-HCOT with the simplest hierarchy possible: one root node that has all leaf classes as its child nodes (illustrated in Appendix L). This flat baseline method uses the same classifier selection (with $C = 2^1 = 2$ combinations) and model optimisation approach, and applies the same (CAT) blocking approach as standard CAT-HCOT. As for the accuracy-timeliness trade-off, we expect the accuracy for flat CAT-HCOT to be slightly lower than that of CAT-HCOT, because the smaller hierarchy is likely to capture the hierarchical structure in the classes less accurately. Presumably, this would most affect the least represented classes in the data (i.e., Medium and Heavily Unhappy). Note that, even though the flat baseline uses a one-level hierarchy, we use the same set of ancestor nodes as in CAT-HCOT when evaluating hierarchical performance measures (such that, e.g., a Medium Unhappy match is also Known and Unhappy). In terms of timeliness, we expect flat CAT-HCOT to provide slightly more timely predictions than regular CAT-HCOT, since its one-level hierarchy causes instances to be classified already when a single threshold is passed.

4.9. Software

We use Python 3.8 to perform all analyses. To program the LR and RF classifiers, we use the *scikit-learn* package [24]. We perform hyperparameter optimisation using the *hyperopt* package [25]. To program the hierarchical structure, we use the framework as presented by Warshaw [26]. Our code is available on GitHub.

5. Results

In this section, we present and evaluate our results. First, we discuss the found best classifier combinations and optimal model parameters. Second, we consider the optimal thresholds as determined by our Certainty-based Automated Thresholds (CAT) algorithm. Third, we evaluate the predictive performance of CAT-HCOT and, fourth, compare it against two baseline methods. Last, we perform a sensitivity analysis on the certainty parameter α, which balances the trade-off between accuracy and timeliness.

5.1. *Classifier Selection and Model Optimisation*

Table H.3 and H.1 present the optimal validation hF_1-scores and the corresponding tuned hyperparameters for all $C = 8$ classifier combinations on all days $t = 0, ..., 10$, respectively. Overall, for each classifier combination, we find that the hF_1-score increases over time: the more days after the order date, the more information is available and, hence, the better the predictive performance. This predictive performance improves most in the first 5 days after the order date, with hF_1-scores reaching roughly 96%. From day 6 to day 9, the performance shows only minor improvements. On day 10, all hF_1-scores lie between 97% and 99%, thus nearly representing perfect scores. For day 0 until day 3, we find that combinations with an LR-classifier on the second parent node (<1>) clearly exhibit lower hF_1-scores than the other combinations, while these differences vanish afterwards. This is an indication that, given the available information, different classifiers may be better at distinguishing between different 'regions' of the data, which is the exact reason why we aim to find our best combination of classifiers and an important factor why hierarchical classification may be preferred over flat classification. After 4 days, all combinations seem to perform well, without extreme differences in hF_1-scores. Among all classifier combinations, we find that LR-RF-RF achieves the highest average hF_1-score (94.03%) measured over all days and forms the optimal combination from day 3 onwards. Also, on day 0 until day 2, the performance of LR-RF-RF is relatively close to the optimal hF_1-score. Since LR-RF-RF thus shows to be the dominant choice, we decide to keep our choice simple and work with this combination across all days instead of using optimal daily combinations.

5.2. *Certainty-Based Automated Thresholds (CAT)*

Next, we discuss the optimal thresholds as determined by the Certainty-based Automated Thresholds (CAT) algorithm. We set the certainty parameter $\alpha = 0.7$ and use $\xi = 100$ number of steps in the allowed search space. We point out that the choice of α is user-specific. However, we disregard $\alpha = 0$ and $\alpha = 1$ beforehand, because they oversimplify the accuracy-timeliness trade-off and therefore do not show the true working of CAT. Section 5.5 provides a sensitivity analysis of the effect of the certainty parameter α on the accuracy and timeliness of the predictions.

Table 3 shows the daily optimal thresholds per node as determined by CAT. In general, we would expect the thresholds to slightly increase over time. This is the case for almost all nodes, apart from some minor fluctuations. Notable is the dip around day 5 for node <1> (Known), which means there is some variability in the distributions of $\zeta_t(<1>)$ and $\zeta_t(\neg <1>)$. Additionally, we note that the thresholds tend to be higher for nodes at the top of the hierarchy (e.g., Unknown and Known) compared to nodes deeper in the hierarchy (e.g., Mildly, Medium, and Heavily Unhappy). A high threshold likely results from a high lower bound, which means the classifier had difficulty separating the distributions of $\zeta_t(m)$ and $\zeta_t(\neg m)$. We point out that one should be careful in comparing thresholds for the Unhappy classes with the other classes, since the range of majority vote probabilities of the Unhappy classes differs from that of the other classes.

Table 3
Optimal thresholds per node and day computed with the CAT algorithm where $\alpha = 0.7$ and $\xi = 100$.

	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
Unknown	95.8%	96.4%	96.5%	96.8%	96.6%	97.5%	98.1%	98.7%	99.0%	99.2%
Known	89.8%	89.3%	80.2%	73.2%	66.5%	64.2%	66.9%	88.2%	83.2%	77.5%
Happy	64.1%	66.1%	74.9%	78.0%	80.8%	84.0%	82.8%	82.9%	84.8%	85.8%
Unhappy	63.4%	59.2%	63.5%	63.5%	69.7%	68.0%	71.9%	71.8%	71.3%	71.7%
Mildly Unhappy	57.0%	58.3%	59.8%	62.0%	64.3%	66.3%	69.1%	70.4%	70.9%	71.0%
Medium Unhappy	51.9%	52.2%	52.4%	52.9%	53.5%	53.9%	55.6%	57.4%	59.6%	61.1%
Heavily Unhappy	50.3%	48.8%	48.2%	47.5%	47.9%	47.1%	47.6%	47.6%	48.1%	49.7%

5.3. CAT-HCOT

Using the best classifier combination with optimal hyperparameters and using the optimal thresholds from CAT ($\alpha = 0.7$), we use the CAT-HCOT algorithm to classify our test instances. In this section, we evaluate the predictive performance of CAT-HCOT. First, we discuss its global and class-specific performance. Second, we evaluate the use of the blocking approach. Last, we discuss feature importance.

5.3.1. General Performance

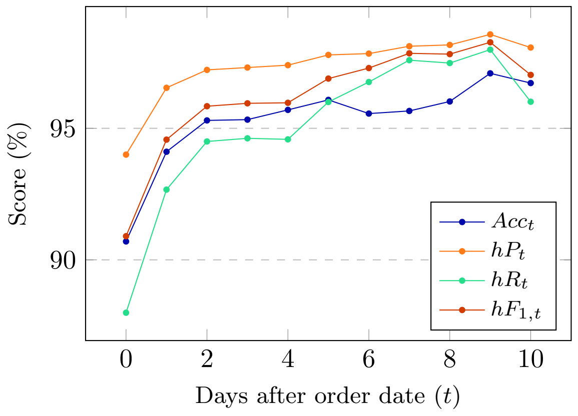
Overall, we find that CAT-HCOT achieves a global accuracy of 93.7%. Furthermore, Table 4 shows global and class-specific hierarchical performance measures. All performance measures in Table 4 are computed based on the entire set of leaf-node predictions that is made in the entire period of 0 to 10 days after the order date. Here, global measures are computed based on all classified instances, whereas class-specific measures are computed based on all instances that are assigned to a specific class. Global hierarchical precision, recall, and F_1-scores all reach values over 90%. If we consider class-specific measures, the scores remain high with values mostly above 90%. One particular class that deviates somewhat is Unknown, which achieves a relatively low hierarchical recall score of 83%. This finding could be explained by the fact that instances only need to pass a single threshold to be classified as Unknown. Hence, for each instance, hierarchical recall is either 0 or 1, whereas for other classes, it can achieve scores between 0 and 1. As a result, hierarchical recall (and thus hierarchical F_1) could be lower for Unknown than for other classes.

Figure 5 presents the performance of the CAT-HCOT algorithm in terms of daily accuracy (Acc_t), daily hierarchical precision (hP_t), daily hierarchical recall (hR_t), daily hierarchical F_1-score ($hF_{1,t}$), and cumulative percentage of classified instances per day. The daily hierarchical performance measures evaluate

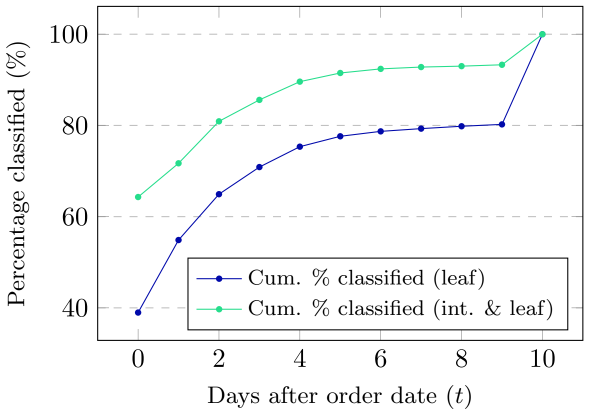
Table 4
Global and class-specific performance of CAT-HCOT.

	hP	hR	hF_1
Global	96.1%	92.2%	94.1%
Happy	96.8%	94.3%	95.6%
Unknown	93.6%	83.0%	87.9%
Mildly Unhappy	97.2%	97.7%	97.4%
Medium Unhappy	92.7%	92.9%	92.8%
Heavily Unhappy	95.2%	95.4%	95.3%

all leaf-node predictions that are made on a specific day. Considering daily accuracy (Figure 5a) and the cumulative percentage of leaf-node predictions (Figure 5b), we find that CAT-HCOT is able to classify roughly 40% of instances on the order date itself with an accuracy of almost 91%. As more information comes in over time, the cumulative percentage of leaf classifications increases further (with decreasing rate). On day 5, CAT-HCOT has leaf-classified almost 80% of instances, where the level seems to stabilise. After day 5, the cumulative percentage of leaf classifications almost stops increasing, which means that roughly 20% of instances cannot be classified with enough certainty. Hence, on day 10, we make sure all these instances are classified by means of mandatory leaf-node prediction. We point out that from day 2 onwards (including day 10), CAT-HCOT achieves a daily accuracy of 95% or higher on the daily leaf-classified instances.



(a) *Daily hierarchical performance measures.*

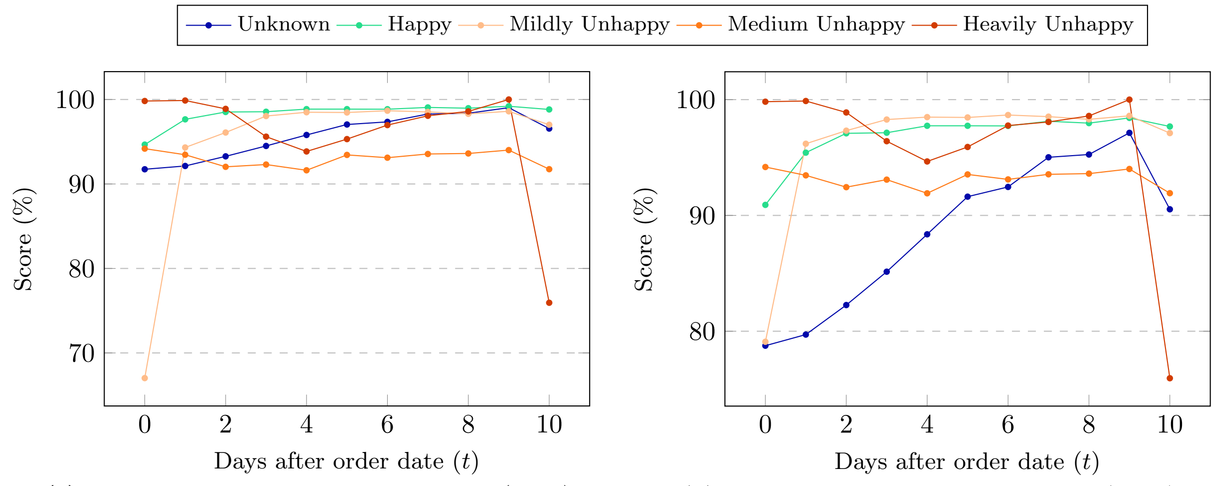


(b) *Cumulative percentage of classified instances for internal- and leaf-node predictions.*

Figure 5. *Daily performance of the CAT-HCOT algorithm.*

For instances that are blocked at a particular parent node and therefore not assigned a leaf class, CAT-HCOT may be able to assign them a predicted class by classifying them into an internal node (Known or Unhappy). It is interesting to evaluate the cumulative percentage of leaf-node *and* internal-node predictions, because this gives a better indication of the practical use of the algorithm. That is, online retailers do not necessarily need detailed labels (leaf nodes) to act upon predictions, but can in some cases also rely on more general predictions (internal nodes). For example, if a product order is predicted to be Unhappy, this already may be valuable enough for the online retailer to act upon. From Figure 5b, we find that roughly 64% of instances is assigned an internal- or leaf-node label on the order date itself, which further increases towards roughly 93% on day 9. Here, we recall that the internal-node predictions are never finalised. Instead, the associated instances are again classified in a top-down manner on the next day, until they reach a leaf node.

Last, we evaluate the performance of CAT-HCOT on a daily, class-specific basis. Figure 6 presents daily class-specific hierarchical precision ($hP_{c,t}$) and recall ($hR_{c,t}$) evaluated for all five leaf classes. We find that almost all daily class-specific hierarchical scores exceed 90%, which indicates that CAT-HCOT achieves very strong predictive performance for all classes. There are two exceptions to this general conclusion. First, hierarchical recall for Unknown shows a deviating pattern over time: it starts relatively low, just below 80%, and increases slowly over time, eventually exceeding 90%. Again, this can be explained by the fact that Unknown is positioned on top of the hierarchy, such that instances only need to pass one threshold to be classified as Unknown. As a result, for each instance, hierarchical recall is either 0 or 1, leading to lower scores on the first days when clearly not all true Unknown instances have been correctly predicted. The second exception is related to the classes Mildly Unhappy on day 0 and Heavily Unhappy on day 10: in both cases, hierarchical precision and recall are relatively low, achieving values below 80%. A possible explanation for this is the low occurrence of these classes and availability of determinants in the data, which makes accurate classification more difficult, even if we impose certainty thresholds and allow for classification over time. For a full overview of all results in this and the following section, we refer to Table M.1.

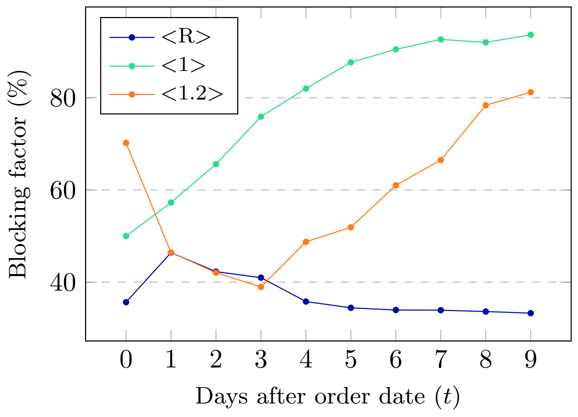


(a) *Daily class-specific hierarchical precision ($hP_{c,t}$).* **(b)** *Daily class-specific hierarchical recall ($hR_{c,t}$).*

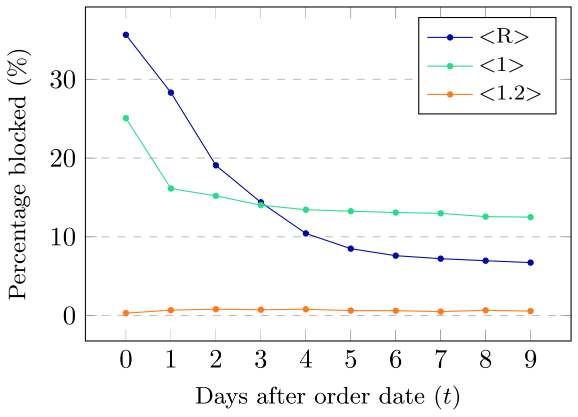
Figure 6. *Daily class-specific hierarchical performance measures.*

5.3.2. Blocking at Parent Nodes

In this section, we give more insight into the CAT blocking approach, which is used to stop classification of instances if the posterior probability at a given level does not exceed a threshold. Figure 7a shows the daily blocking factor for the different three parent nodes in our hierarchy. Over time, we find that the blocking factor decreases for the root node <R>, while it increases for parent nodes <1> and <1.2>. This could be explained by the notion that the same instances get blocked at parent nodes <1> and <1.2> over time, while the total number of instances reaching these parent nodes decreases over time. That is, in absolute terms, the number of blocked instances stays rather constant, but the number of instances entering the node decreases, such that the blocking factor increases. This can also be observed from Figure 7b, where we plot the percentage of blocked instances relative to the (fixed) total number of instances that need to be classified. Over time, this percentage stays relatively constant for nodes <1> and <1.2>, while the daily total fraction of blocked instances, given by the sum of the percentages at the three parent nodes, decreases over time. This is mostly due to the steady decrease in blocked instances at root node <R>. However, we do note that up until day 9, at least 6.7% of instances is blocked at this node. These instances are not assigned an internal classification and therefore do not provide us with actionable business insights. Logically, since we apply MLNP on day 10, the fraction of blocked instances in the end reaches 0 for all parent nodes.



(a) *Daily blocking factor per parent node.*



(b) *Blocked instances as a percentage of all instances.*

Figure 7. *Analysis of the blocking procedure at the parent nodes.*

Even if there is not enough certainty of providing leaf-node predictions, CAT-HCOT still provides actionable insights by assigning internal labels. We do not finalise these internal-node predictions, meaning that the next day the instances are again classified in a top-down manner and may end up in a different (internal) node. This procedure helps us overcome the issue of error propagation. Figure 8 shows the classification performance at the two internal nodes <1> (Known) and <1.2> (Unhappy). First, we observe that hierarchical recall is relatively low for both nodes. This is not surprising, since, by definition, internal-node predictions have not yet reached the most specific node possible, making it impossible to achieve perfect recall scores. More specifically, daily class-specific hierarchical recall ($hR_{c,t}$) can reach maximums of 50% and 66.7% for <1> and <1.2>, respectively, since at most 1 out of 2 and 2 out of 3 nodes in the path can be assigned correctly. Second, we find that hierarchical precision is close to 100% on all days for <1>, while predictions are almost always perfectly precise for <1.2> from day 5 onwards. Still, a significant fraction of internal predictions is incorrect for <1.2> on day 0 until day 4. This shows that, indeed, it may not be a good idea to finalise internal predictions, as this would propagate the error to the next day by continuing classification from an incorrect node. However, as the internal-node predictions are nearly always correct after day 4, it could be beneficial to finalise the internal classifications from this day onwards.

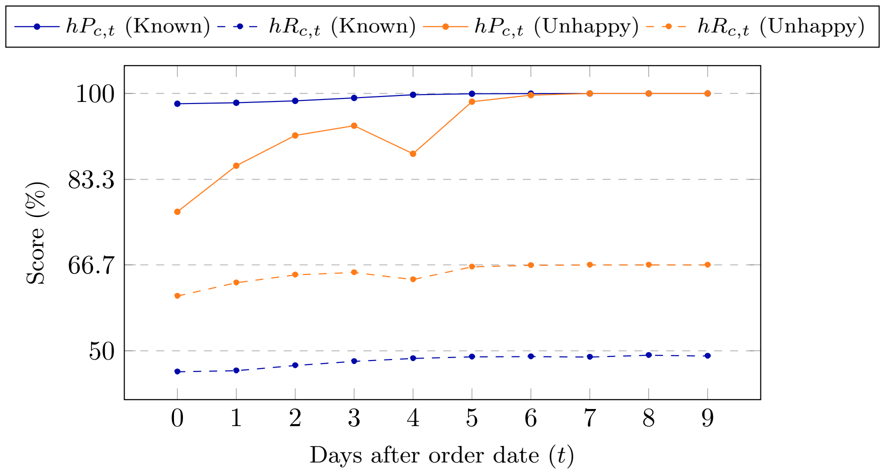


Figure 8. *Daily internal-node hierarchical performance measures. Daily class-specific hierarchical recall ($hR_{c,t}$) can reach maximum values of 50% and 66.7% for <1> (Known) and <1.2> (Unhappy), respectively.*

5.3.3. Feature Importance

To get more insight into the variables that drive hierarchical classification of product orders, we analyse the importance of our model features. Appendix N shows a summary of the most important features per parent node (where a classifier is trained) and per day. It turns out that the dummy variables based on the match determinants are generally most important in product order classification. For example, it is important to know whether a customer case is opened or whether a product is returned. Over time, their importance increases, since more and more information on cancellations, late deliveries, customer cases, and returns becomes available. Other important variables are those indicating the historic performance of transporters based on the fraction of Happy, Unhappy, and Unknown matches.

5.4. Baseline Methods

In this section, we evaluate the performance of CAT-HCOT against two baselines: static HCOT, which does not use a blocking approach, and flat CAT-HCOT, which uses a one-level hierarchy. Figure 9 plots daily accuracy scores and cumulative percentages of classified instances for CAT-HCOT and the two baselines.

We first compare CAT-HCOT to the static baseline. For static HCOT, we achieve an accuracy of 81.7% when predicting all instances on day 0. This accuracy steadily increases until reaching a value of 98.2% on day 10. This increase makes sense, as with each day more information becomes available, which in turn leads to more accurate predictions. In contrast to static HCOT, CAT-HCOT does not assign leaf-node predictions for all instances at each day, but only classifies those instances for which it is certain enough. This leads to higher daily accuracy than static HCOT for the first 5 days after (and including) the order date. In contrast, from day 6 to day 10, static HCOT attains higher daily accuracy than CAT-HCOT. This is mainly because in the later days, CAT-HCOT only needs to classify those instances that were previously blocked and are thus difficult to classify to a leaf node, whereas static HCOT classifies the full set of instances each day. This makes it easier to attain higher daily accuracy for static HCOT than for CAT-HCOT from day 6 until day 10. All in all, we find that CAT-HCOT nicely balances the accuracy vs. timeliness trade-off by using a certainty-based classification approach, while static HCOT does not allow for such a balanced trade-off.

Second, we compare CAT-HCOT to the flat baseline ($\alpha = 0.7$). From Figure 9, we find that CAT-HCOT achieves higher accuracy scores than flat CAT-HCOT, while it provides less timely predictions. However, these differences are small. Instead, CAT-HCOT and flat CAT-HCOT mostly differ in terms of class-specific hierarchical performance. In particular, we find that flat CAT-HCOT is not able to achieve similarly high hierarchical precision and recall scores as CAT-HCOT on the Unhappy classes. While CAT-HCOT achieves hierarchical precision and recall scores of above 90% for Medium and Heavily Unhappy (Table 4), flat CAT-HCOT only achieves scores between 50% and 80%, where the performance gap is most pronounced in the first days after the order date. Hence, especially for classes deep in the hierarchy which have low representations in the data and are difficult to predict soon after the order date, CAT-HCOT achieves major improvements in predictive performance over flat CAT-HCOT, because it takes into account the full hierarchical class structure. This underlines the added value of hierarchical classification in a setting where classes are hierarchically structured, even if the tree-based hierarchy is shallow (only 7 nodes over 3 levels). Tables M.2 and M.3 provide more detailed evaluation results of the flat baseline method.

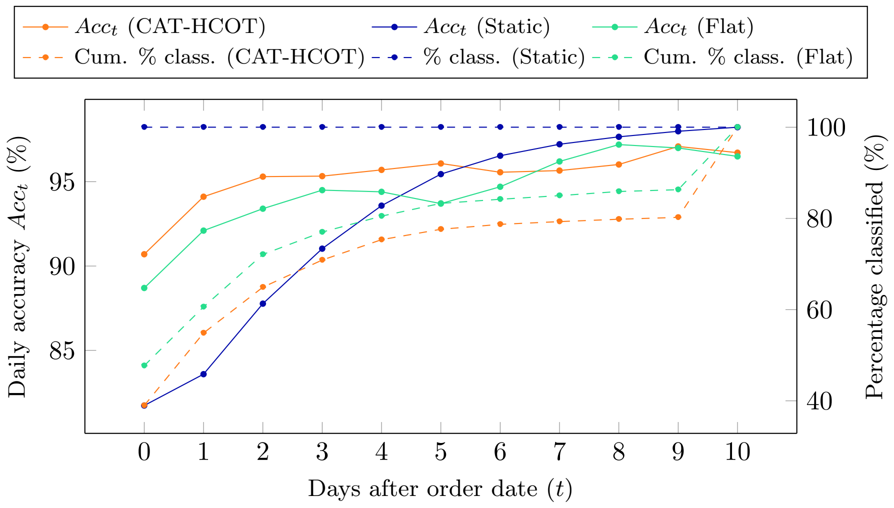
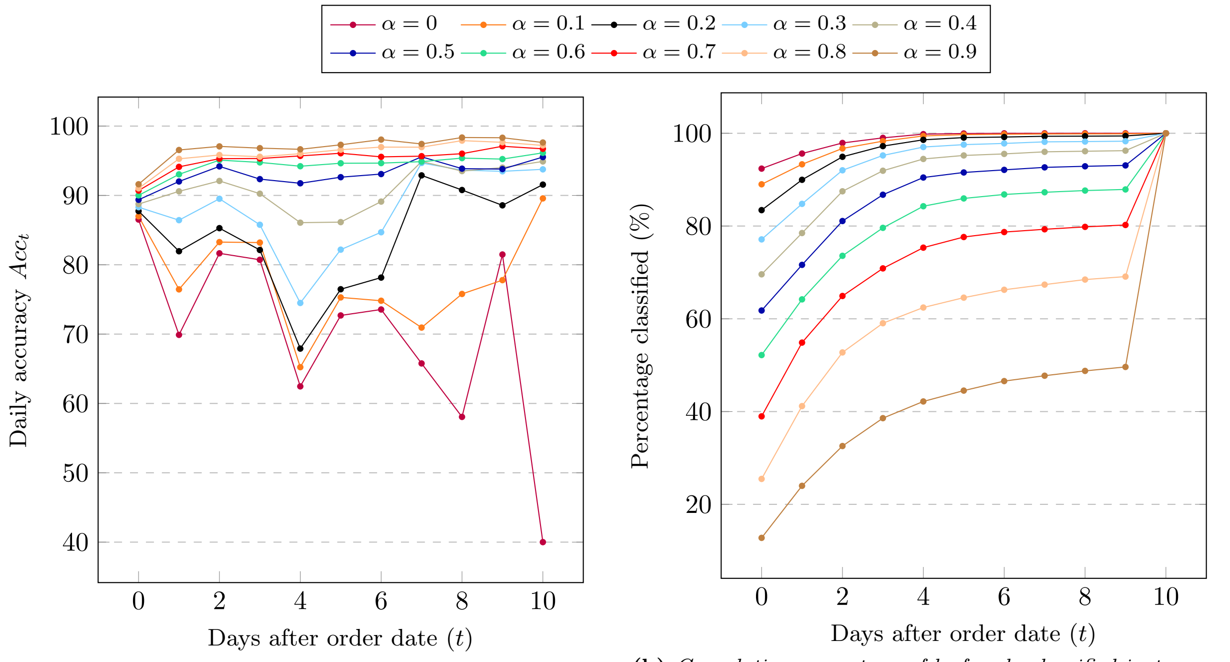


Figure 9. *Comparison of CAT-HCOT against static and flat baseline methods.*

5.5. Sensitivity Analysis of Certainty Parameter

To analyse the effect of the certainty parameter α, which balances the trade-off between accuracy and timeliness, we perform a sensitivity analysis. We evaluate the CAT-HCOT algorithm for values of α ranging from 0 to 1 with step size 0.1. Figure 10 shows the $\alpha-$specific accuracy and the cumulative percentage of leaf-node predictions over time. We mention that for $\alpha = 1$, the CAT-HCOT algorithm fails, because it only gives 0.04% of instances a leaf-node prediction in the first nine days after (and including) the order date. Therefore, the results for $\alpha = 1$ are omitted from the figure. First, Figure 10a shows that higher values of α lead to higher leaf-node prediction accuracy. This make sense, because a higher certainty implies that we put more weight on accuracy than on timeliness. Second, Figure 10b shows that a higher α leads to a lower cumulative percentage of leaf-classified instances in the first nine days after the order date. This makes sense too, because, following the accuracy-timeliness trade-off, higher accuracy comes at the expense of timeliness.

Inspecting specific values of α, we find that for $\alpha < 0.5$, accuracy scores are highly volatile on the first few days after the order date. On the other hand, high fractions of instances get assigned to a leaf node on those days. The opposite story goes when $\alpha > 0.5$, for which fewer instances get leaf-classified in the first few days, but the accuracy scores of these instances reach much higher levels. Hence, in applications where timeliness is more important than accuracy, lower values of α are more suitable, while higher values of α are preferable in case accuracy is considered to be more important than timeliness.



(a) *Daily accuracy Acc_t of leaf-node predictions.* **(b)** *Cumulative percentage of leaf-node classified instances.*

Figure 10. *Sensitivity analysis for certainty parameter α in CAT.*

6. Conclusions and Future Work

In this paper, we have designed a method to accurately predict the match quality of product orders at a large online retailer in the Netherlands shortly after the orders have been placed. In this prediction task, we face a clear trade-off between accuracy and timeliness: on the one hand, product orders should be classified as accurately as possible to ensure reliable predictions, and on the other hand, product orders should be classified as soon as possible to ensure actionable predictions. We use our own hierarchical classification method that classifies instances over time in a top-down, non-mandatory leaf-node prediction procedure with certainty-based automated thresholds.

Our results suggest that the Hierarchical Classification Over Time (HCOT) algorithm combined with the Certainty-based Automated Thresholds (CAT) algorithm, in short CAT-HCOT, achieves highly promising predictive performance. CAT-HCOT obtains a global predictive accuracy of 93.7%, with daily accuracies consistently exceeding 90%. In addition, both global and class-specific hierarchical performance measures almost always achieve scores above 90%. With respect to timeliness, CAT-HCOT is able to classify around 40% of orders on the same date these orders are placed, approximately 80% of orders within 5 days after the order date, and 100% of orders after 10 days. All in all, we find that CAT-HCOT nicely captures the accuracy vs. timeliness trade-off by using a certainty-based classification approach, while a static method does not allow for such a balanced trade-off. We further find that CAT-HCOT achieves major improvements over a flat baseline method in terms of hierarchical precision and recall scores on the most specific and least occurring Unhappy classes, which illustrates that taking into account the hierarchical class structure improves predictive performance. Last, CAT-HCOT is widely applicable and highly flexible, because the certainty parameter α can be user-specified to balance the trade-off between accuracy and timeliness. We outline further business implications of our research in Appendix O.

The main contribution of our research lies in three areas. First, we propose a new method of using hierarchical classification in a dynamic way. Second, we introduce a new certainty-based automated thresholds algorithm and combine this algorithm with hierarchical classification over time. Last, we add to the literature on hierarchical classification by applying this technique in a shallow-tree setting with imbalanced data, specifically tailored to the field of customer satisfaction in online retailing.

In the future, this research may be extended into various interesting directions. First, we point out that, although this research is tailored to the domain of customer satisfaction in online retailing, the CAT-HCOT framework can be applied to any other dynamic hierarchical classification problem. Furthermore, the CAT algorithm does not require the presence of a time component and can thus be applied to any non-mandatory leaf-node hierarchical classification problem. Second, we can implement the CAT-HCOT algorithm by using different classification methods that are able to produce near-continuous class prediction probabilities, such as Naive Bayes, Neural Networks, or Support Vector Machines. Third, it can be interesting to use a different hierarchical structure in the match labels that is 'optimal' according to some algorithm or heuristic. In this research, the tree-based hierarchy is defined based on business logic, but it might be that different splits at different tree levels lead to better classification performance. Last, instead of predicting match labels, it can be interesting to predict the match determinants that determine the match quality of a product order, which include cancellations, late delivery, customer cases, and returns. This would give more insights into the quality of matches from a business perspective, providing online retailers with the reasons why a particular product order is happy or unhappy.

References

[1] L. Zhou, L. Dai, D. Zhang, Online shopping acceptance model - A critical survey of consumer factors in online shopping, Journal of Electronic Commerce Research 8 (2007) 41–62.

[2] N. Donthu, A. Gustafsson, Effects of covid-19 on business and research, Journal of Business Research 117 (2020) 284–289.

[3] C. N. Silla, A. A. Freitas, A survey of hierarchical classification across different application domains, Data Mining and Knowledge Discovery 22 (2011) 31–72.

[4] A. Freitas, A. Carvalho, A tutorial on hierarchical classification with applications in bioinformatics, Research and Trends in Data Mining Technologies and Applications (2007) 175–208.

[5] A. Sun, E.-P. Lim, Hierarchical text classification and evaluation, in: Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001), IEEE, 2001, pp. 521–528.

[6] D. Vandic, F. Frasincar, U. Kaymak, A framework for product description classification in e-commerce, Journal of Web Engineering 17 (2018) 1–27.

[7] S. Guo, H. Zhao, Hierarchical classification with multi-path selection based on granular computing, Artificial Intelligence Review 54 (2021) 2067–2089.

[8] S. Dumais, H. Chen, Hierarchical classification of web content, in: Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2000), ACM, 2000, pp. 256–263.

[9] A. Clare, R. D. King, Knowledge discovery in multi-label phenotype data, in: Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2001), Lecture Notes in Computer Science, vol. 2168, Springer, 2001, pp. 42–53.

[10] I. Dimitrovski, D. Kocev, S. Loskovska, S. Džeroski, Hierarchical annotation of medical images, Pattern Recognition 44 (2011) 2436–2449.

[11] Z. Xiao, E. Dellandrea, W. Dou, L. Chen, Automatic hierarchical classification of emotional speech, in: Proceedings of the 9th IEEE International Symposium on Multimedia Workshops (ISMW 2007), IEEE, 2007, pp. 291–296.

[12] B. Andres, U. Köthe, M. Helmstaedter, W. Denk, F. A. Hamprecht, Segmentation of SBFSEM volume data of neural tissue by hierarchical classification, in: Proceedings of the 30th Annual Symposium of the German Association for Pattern Recognition (DAGM 2008), Lecture Notes in Computer Science, vol. 5096, Springer, 2008, pp. 142–152.

[13] Y. Chen, S. Zhao, Z. Xie, D. Lu, E. Chen, Mapping multiple tree species classes using a hierarchical procedure with optimized node variables and thresholds based on high spatial resolution satellite data, GIScience & Remote Sensing 57 (2020) 526–542.

[14] M. Ceci, D. Malerba, Hierarchical classification of HTML documents with WebClassII, in: Proceedings of the 25th European Conference on Information Retrieval (ECIR 2003), Lecture Notes in Computer Science, vol. 2633, Springer, 2003, pp. 57–72.

[15] A. Addis, G. Armano, E. Vargiu, Assessing progressive filtering to perform hierarchical text categorization in presence of input imbalance, in: Proceedings of the 2nd International Conference on Knowledge Discovery and Information Retrieval (KDIR 2010), SciTePress, 2010, pp. 14–23.

[16] M. Ceci, D. Malerba, Classifying web documents in a hierarchy of categories: A comprehensive study, Journal of Intelligent Information Systems 28 (2007) 37–78.

[17] M. Sokolova, G. Lapalme, A systematic analysis of performance measures for classification tasks, Information Processing & Management 45 (2009) 427–437.

[18] E. Costa, A. Lorena, A. Carvalho, A. Freitas, A review of performance evaluation measures for hierarchical classifiers, in: Proceedings of the 2nd AAAI Evaluation Methods for Machine Learning Workshop (EMML 2007), AAAI, 2007, pp. 1–6.

[19] S. Kiritchenko, S. Matwin, R. Nock, A. F. Famili, Learning and evaluation in the presence of class hierarchies: Application to text categorization, in: Proceedings of the 9th Conference of the Canadian Society for Computational Studies of Intelligence (Canadian AI 2006), Lecture Notes in Computer Science, vol. 4013, Springer, 2006, pp. 395–406.

[20] A. Sun, E.-P. Lim, W.-K. Ng, Performance measurement framework for hierarchical text classification, Journal of the American Society for Information Science and Technology 54 (2003) 1014–1028.

[21] C. Meiring, A. Dixit, S. Harris, N. S. MacCallum, D. A. Brealey, P. J. Watkinson, A. Jones, S. Ashworth, R. Beale, S. J. Brett, et al., Optimal intensive care outcome prediction over time using machine learning, PLOS One 13 (2018) e0206862.

[22] C. Elkan, The foundations of cost-sensitive learning, in: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), Morgan Kaufmann Publishers Inc., 2001, pp. 973–978.

[23] W. Zheng, H. Zhao, Cost-sensitive hierarchical classification for imbalance classes, Applied Intelligence 50 (2020) 2328–2338.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[25] J. Bergstra, D. Yamins, D. D. Cox, Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures, in: Proceedings of the 30th International Conference on Machine Learning (ICML 2013), JMLR.org, 2013, pp. 115–123.

[26] D. Warshaw, Decision tree hierarchical multi-classifier, https://github.com/davidwarshaw/hmc, 2017. Accessed: February 2021.

[27] T. Hastie, R. Tibshirani, J. Friedman, The elements of statistical learning: Data mining, inference, and prediction, Springer Science & Business Media, 2009.

[28] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, C.-J. Lin, Liblinear: A library for large linear classification, Journal of Machine Learning Research 9 (2008) 1871–1874.

[29] R. Martinez-Cantin, Bayesopt: a bayesian optimization library for nonlinear optimization, experimental design and bandits, Journal of Machine Learning Research 15 (2014) 3735–3739.

[30] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, in: Proceedings of the 25th Annual Conference on Neural Information Processing Systems (NIPS 2011), Curran Associates, Inc., 2011.

Appendix A List of Definitions

- **Accuracy versus timeliness trade-off**. Trade-off between the quality of our predictions and the time that it takes to make these predictions. Generally, the more time we take, the more accurate we can be and vice versa.
- **Blocking approach**. Procedure that prohibits instances from reaching lower levels of the hierarchy if some criteria are not met.
- **Blocking factor**. Measure that represents the fraction of test instances blocked at a hierarchical classifier relative to the number of instances entering the classifier.
- **Cancellation**. A product order is cancelled by either the seller or the customer before delivery takes place. We consider only cancellations within 10 days after the order date.
- **CAT**. Certainty-based Automated Thresholds, defined via the CAT algorithm. These thresholds are used to determine whether we are certain enough about orders to be put through to the next level of the hierarchy.
- **CAT-HCOT**. Combination of the CAT and HCOT algorithms that makes use of automated thresholds to apply hierarchical classification over time.
- **Certainty parameter.** Parameter that defines how certain we are about our predictions, ranging between 0 (least confident) and 1 (most confident). The specific value of the certainty parameter indicates what fraction of points that should not be accepted with a certain threshold would be blocked on the train set (and are expected to be blocked in the test set).
- **Customer case**. A customer case is started when the customer has a complaint about the order process or the product itself. We consider only customer cases that are started within 30 days after the order date.
- **Detailed match classification**. Classification of product orders into the Happy, Unknown, Mildly Unhappy, Medium Unhappy, and Heavily Unhappy classes.
- **Error propagation**. Principle indicating that classification errors made at a higher level of the hierarchy are passed on through to lower levels of the hierarchy.
- **FC**. Flat Classification: simples hierarchical classification technique that assumes a one-level hierarchy such that all classes are predicted simultaneously in a single tree split.
- **Flat CAT-HCOT**. Specific case of CAT-HCOT that makes use of a flat hierarchy, i.e., a hierarchy with a single parent node (the root node). Used as one of the baseline methods in this research.
- **General match classification**. Classification of product orders into the Happy, Unknown, and Unhappy classes.

- **Happy Match**. A match is Happy when all service criteria (or match determinants) are met, i.e., there is no cancellation, the product is delivered on time, there is no customer case, and no return.
- **HC**. Hierarchical Classification: technique taking into account the hierarchical structure in the classes to be predicted.
- **HCOT**. Hierarchical Classification Over Time, defined via the HCOT algorithm. Orders are classified over time based on their prediction certainty at different points in time.
- **Heavily Unhappy Match**. Detailed match label denoting the most severe gradation of the Unhappy Match class.
- **Hierarchical performance measures**. Measures to evaluate the predictive performance of hierarchical classifiers that incorporate the hierarchy-based tree structure directly.
- **LCPN**. Local Classification per Parent Node: procedure that applies a binary or multi-class classifier at each parent node in the hierarchy to train the classifier.
- **Leaf-classification.** Classification of an instance to a leaf node of the tree-based class hierarchy corresponding to a detailed match label.
- **Match**. A customer orders a product from a seller via the online retailer's platform. Following this order, a match between customer and seller is constituted.
- **Match determinants**. The factors that determine the match quality of a product order, i.e., what type of match label is assigned to a product order. The match determinants include cancellations, on-time delivery, customer cases, and returns. See also 'service criteria'.
- **Match label**. The label that is assigned to a product order based on the quality of the match between seller and customer (Figure C.1). We distinguish between a general match label classification (Happy / Unhappy / Unknown) and a detailed classification (Happy / Mildly Unhappy / Medium Unhappy / Heavily Unhappy / Unknown).
- **Match quality**. The extent to which seller, customer and online retailer are satisfied following a product order by the customer.
- **Medium Unhappy Match**. Detailed match label denoting the middle gradation in terms of severity of the Unhappy Match class.
- **Mildly Unhappy Match**. Detailed match label denoting the least severe gradation of the Unhappy Match class.
- **MLNP**. Mandatory Leaf-Node Prediction: type of HC that, in contrast to NMLNP, forces the classification algorithm to always assign an instance to a leaf node.

- **NMLNP**. Non-Mandatory Leaf-Node Prediction: type of HC that, in contrast to MLNP, allows instances to be blocked at internal nodes in the hierarchy during classification.
- **On-time delivery**. A product is delivered on time when the product is delivered on or before the promised delivery date and within 13 days after the order date.
- **Product order**. A single order of a particular product (encoded in the data set as the combination of a product, quantity ordered, price, date and customer).
- **Return**. A product can be returned by the customer within 30 days after the order has been placed.
- **Service criteria**. The criteria that are used to evaluate the match quality of a product order. The service criteria include cancellations, on-time delivery, customer cases and returns. See also 'match determinants'.
- **SPP**. Single Path Prediction: indicates that an instance can be assigned to at most one path of predicted labels at any given level of the hierarchy.
- **Static HCOT**. Hierarchical classification method that applies MLNP to classify all instances on each prediction day. One of the baseline methods used in this research.
- **Threshold**. Node-specific level of confidence that a to-be classified instance should exceed to be passed on through to the corresponding node in the hierarchy.
- **Top-down class prediction approach**. Testing procedure that determines the flow in which instances are classified. Here, the most generic level is first classified, after which an instance moves to the next level in the hierarchy until it reaches a leaf node or gets blocked.
- **Tree-based class hierarchy**. Type of hierarchical structure that defines the relation between classes, where a node can have only one parent node.
- **Unhappy Match**. A match is Unhappy when one or more of the service criteria (or match determinants) is/are not met, i.e., the order is cancelled, the product is not delivered on time, there is a customer case, and/or the product is returned.
- **Unknown Match**. A match is Unknown when all service criteria are met but product delivery is unknown.

Appendix B Data and Variable Procedures

From the online retailer, we obtained two data sets: one containing data from 2019, the other from 2020. We loaded these into one table, containing a combined total of 4,779,466 orders. The records do not contain an order ID. Still, the online retailer assured us that the data does not have multiple records for one product order. However, there is no way to be absolutely certain, since we cannot verify this ourselves. Note that there are duplicate rows because it is possible that the same order was placed by a different customer with matching timelines.

B.1 Data Cleaning

First, we list a number of observations that do not make any sense and remove these from our data set. The removed types of observations are listed in Table B.1 alongside their number of occurrences in the data set. In addition to removing observations, we also solve for a particular type of noisy observation by means of transformation.

We elaborate on three types of nonsensical observations. The first are orders from sellers for which the registration date is unknown. Since the time a seller has been active on the platform can provide insightful information (e.g., about reliability), we use the registration date of sellers in our models. Most algorithms do not accept 'null' values and because the frequency of this aberration is relatively low (234 records), we choose to remove records where the registration date of the seller is unknown. Another option would have been to estimate the registration date, which would not have been an easy task due to the fact that we have little unique information on specific sellers.

Secondly, we observe several orders for which the promised delivery date is unknown, a phenomenon

Table B.1
Noisy orders and the number of times the observation is found in the orders. There is some overlap: these numbers sum to 2,627, when in reality this pertains a total of 2,624 orders. Firstly, there is one instance where the order is registered for return before it is delivered and the promised delivery date is unknown. Secondly, there is one order where a case is opened before the order date and the promised delivery date is unknown. Thirdly, there is one order where a case is opened before the order date and the product is registered for return before the first delivery moment.

Noisy observation	Number of occurrences
Order is cancelled before the order date	0
Order has promised delivery date before the order date	76
Order is shipped before the order date	0
Order is delivered before the order date	34
Order is marked for return before the order date	0
Order has a case started before the order date	170
Seller was not registered on the order date	0
Order is cancelled after delivery	19
Order is cancelled after it is marked for return	1
Order is marked for return before it is delivered	1152
Order has an unknown registration date of seller provided	234
Order has an unknown promised delivery date	941

that occurs 941 times. The same reasons apply as before: the number of occurrences is relatively small and estimating the promised delivery date is difficult. An unknown promised delivery date results in an unknown value for *onTimeDelivery*. If this value would have been known, we could roughly deduce the promised delivery date using *onTimeDelivery* and *dateTimeFirstDeliveryMoment*. But since this is not the case, removing the records seems to be our best option.

The third type of strange observation is that of orders where more products are returned than were originally ordered. This happens in 14,722 product orders. Here, we choose to adjust the value for the quantity returned because (i) removing all records would result in significant information loss, and (ii) it is relatively easy to adjust the value in a reliable way. There are two options: either one equates the ordered quantity to the returned quantity, or the other way around. Since we assume the quantity ordered to be more prominent than the quantity returned, we opt for the second approach: if $quantityReturned > quantityOrdered$, then $quantityReturned = quantityOrdered$. This might not be the actual quantity that is returned, but it provides us with a reliable estimate. Furthermore, the impact of being slightly off is negligible, since the value is changed minimally, i.e., it is decreased by 1 in 87% of the cases. The largest difference that occurs is 50, which happens once.

B.2 Data Validation

After having cleaned the data, we make sure that the drivers of match determinants are consistent with the observed values of the match determinants and that the match determinants are consistent with the observed match labels.

The allocation of match determinants only takes a pre-specified period into consideration. If an event happens outside of that time frame, it will be treated as if it did not happen. The logic for the determinants is as follows, and can also be found in Figure B.1.

- **Cancellations**. A cancellation is taken into account if the cancellation was made within 10 days after the order date. Additionally, if the customer cancelled the order, the cancellation is also required to have been made after the promised delivery date. This requirement is based on business logic provided by the retailer.

- **Deliveries**. A delivery should be known to be on time or late within 13 days after the order date, otherwise it will be treated as unknown. That is, a delivery that is on time and within 13 days of the order date is considered on-time. If a delivery is known to be late within 13 days after the order date, it is marked as a late delivery. If the delivery is unknown or becomes known only after 13 days, it is considered to be unknown.

- **Returns**. If an order is requested to be returned within 30 days, it is classified as returned.

- **Cases**. If a case is opened within 29 days after the order date, it is classified as having one or more cases. If there are multiple cases, it is unknown to us when the subsequent cases are opened.

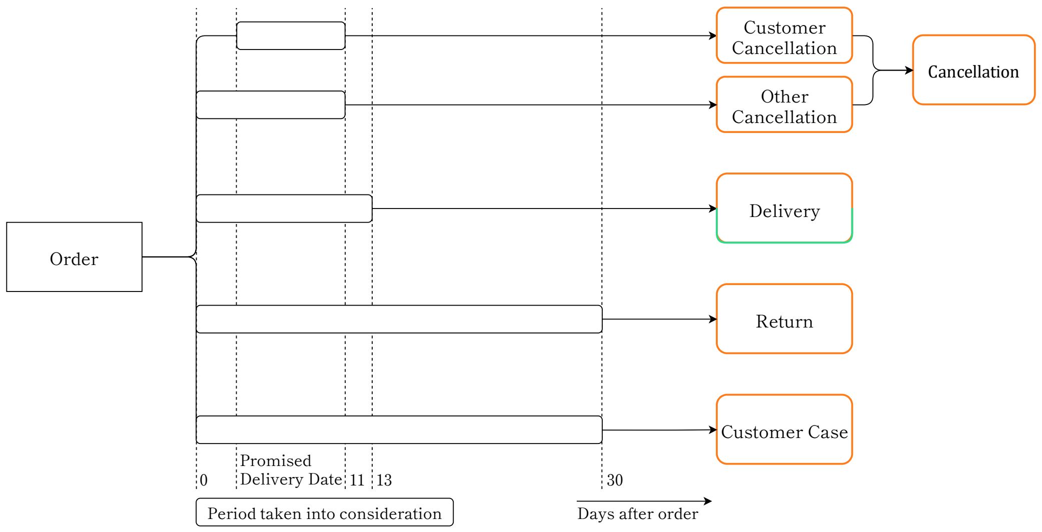


Figure B.1. *Timeline of consideration for determinants. Events that happen within the marked time frame are taken into consideration. Events that happen outside of the time frame are treated as if they did not happen. Note that a delivery event within the considered period can be a late delivery as well as an on-time delivery, which could result in unhappy and happy matches, respectively.*

As for cancellations, business logic is only specified for cancellations requested by sellers or by customers. However, there are other types of cancellations. According to our most up-to-date knowledge, there are some orders which should be classified as having a cancellation, even though they are classified as having no cancellation. This pertains 277 orders, mostly cancelled due to a technical issue. We suspect there might be a more complicated logic behind the classification of cancellations, hence our starting point is that the variable *noCancellation* is correct.

Furthermore, we adjust the values for the drivers of the determinants. If an order is said to have no cancellation according to *noCancellation*, the values for *cancellationDate* and *cancellationReasonCode* are set to 'null', even if they are originally filled in. The driver for an on-time delivery is the first delivery moment relative to the promised delivery date. If an order is classified as having an unknown delivery, in which case the promised delivery date is 'null', we also set the value of the first delivery moment to 'null'. If an order is classified by *noReturn* as having no return, the values for *returnDate*, *quantityReturned*, and *returnCode* are adjusted to be 'null'. Similarly, if an order is classified by *noCase* as having no case(s), the values for *startDateCase* and *cntDistinctCaseIds* are set to 'null'.

Lastly, the consistency of the linkage between the match determinants and the match labels is checked. It is confirmed that every order has the correct label assigned given its values for the match determinants and applying the decision tree as in Figure C.1.

B.3 Variable Procedures

In order to use our prediction models, we select, transform, and create variables such that we get accurate and well-interpretable results. Below, we outline the procedure of how we modifythe set of original variables in our data set into the set of variables that we use for prediction purposes.

1. **Dropping irrelevant variables.** From the original set of variables, we select all relevant variables and drop some unnecessary variables.

    - We do not consider *transporterName* and *transporterNameOther*, since *transporterCode* alone gives sufficient information about the type of transport company that takes care of product delivery.
    - We do not consider *brickName*, *chunkName*, *productSubGroup*, and *productSubSubGroup*, since they contain very detailed product classifications with many labels. In prediction, this would lead to many dummy variables in our models. We only use *productGroup*, since it gives a sufficiently accurate description of the various product groups.
    - We do not consider *cancellationCode*, as the reason behind a cancellation does not provide us with additional information: knowing that an order has been canceled (which we can infer through *cancellationDate*) is sufficient.
    - We do not consider *cntDistinctCaseIds*, as the number of distinct cases is final only after 30 days from the order date. However, we want to make our predictions earlier, so we cannot use this variable.
    - We do not consider *returnCode*, as, similar to *cancellationCode*, the reason behind a return does not provide us with additional information: knowing that an order has been returned (which we can infer through *returnDateTime*) is sufficient.
    - We do not consider *currentCountryAvailability*, as this variable gives a snapshot of the current availability of partners, but does not give the availability of a partner at specific order dates in the past. As we cannot use current information when predicting using past data, this variable is thus not used.
    - We do not consider *calculationDefinitive*, since this variable does not give us extra information. More specifically, it indicates for every order whether 30 days after the order date have passed, which is always the case in our data set.

2. **Creating dummy variables from categorical variables.** In our prediction models, we cannot use variables with multiple categories and, therefore, we transform some of our categorical variables into dummy variables, each representing one category of the original categorical variable. This concerns the variables *countryCode*, *transporterCodeGeneral*, *fulfilmentType*, *productGroupGeneral*, and *countryOriginSeller*.

    - As *transporterCode* contains many labels and some labels cover only a tiny fraction of observations, we manually cluster the labels into groups to improve interpretability. We reduce the number of labels from 28 to 5, obtaining three labels for the three transport companies with most orders in the data set, one label for all orders which are sent by mail, and one label for all other (smaller) transport companies. In this way, we produce the new variable *transporterCodeGeneral*. This variable is again transformed into a set of 5 dummy variables to use in our prediction models. We note that the transporter code is only known after the shipment date is confirmed. Hence, the dummy variables related to *transporterCodeGeneral* are always 0 if the shipment date is not yet known at the time of prediction.

- As *productGroup* contains many labels and some labels only cover few observations, we manually cluster the labels into groups to improve interpretability. We use the general product categories listed on the online retailer's website to cluster the product orders. For each of the 60 original product group categories, we decide manually to which of the 14 general product categories it belongs. In this way, we cluster product orders into 14 instead of 60 groups and produce the new variable *productGroupGeneral*. This variable is again transformed into a set of 14 dummy variables to use in our prediction models.

3. **Creating date-related variables.** For all of our time-related variables, we transform the dates to either date differences or dummies indicating whether information is known at a specific point in time.

   - At the time of ordering (0 days after the order date), we can only use the variables *registrationDate-Seller* and *promisedDeliveryDate*, as the other date variables may still be unknown (e.g., at the time of ordering, it is not yet known whether a case is or will be made). For both variables, we calculate the date difference from the order date and use the resulting variables in our prediction models.
   - At $x > 0$ days after the order date, if a service event is known to have happened, we can calculate the date difference from the order date. However, if a service event is not (yet) known to have happened, we cannot calculate the date difference. Hence, for every service event (cancellation, on-time delivery, customer case, and return), we create a dummy variable that indicates whether or not the service event is known to have happened at $x > 0$ days after the order date. If the dummy variable is 1, this means that the service event is known to have happened. If the dummy variable is 0, this means that the service event has not happened (yet).

4. **Creating time-related dummy variables.** For each order, we create dummy variables indicating on which day of the week, in which month, and in which year the order was made. Also, we create a dummy variable which indicates whether or not the order was made during the COVID-19 pandemic.

5. **Creating product- and seller-related performance variables.** We create variables that give the total number of orders and units sold of a product, the total number and fraction of returns of a product, and average number of orders per day of a seller. These variables always take into account the most recent information up until the time of prediction. In addition, we create historic performance variables for *transporterCode*, *sellerId* and *productGroup* that indicate the fraction of orders that is known to be Happy, Unhappy, or Unknown for a specific value of these variables at a certain point in time.

6. **Other variables.** We create a variable *productTitleLength*, which gives the character length of the string containing the title of the product that was ordered. We suspect that more concise product titles might belong to more reputable, well-known products, which could correspond with higher order satisfaction.

In Appendix D, a list of all variables contained in the original data set from our retailer is provided. A complete overview of all variables used in our prediction models is given in Appendix E.

Appendix C Happy Match Decision Tree

Figure C.1 presents our proposed decision tree for classification of product orders according to the classification system with happy, unhappy, and unknown matches. Although this tree slightly differs from the one that was provided to us by the online retailer (through the position of the Unknown class in the tree), our proposed decision tree is in perfect accordance with the data set.

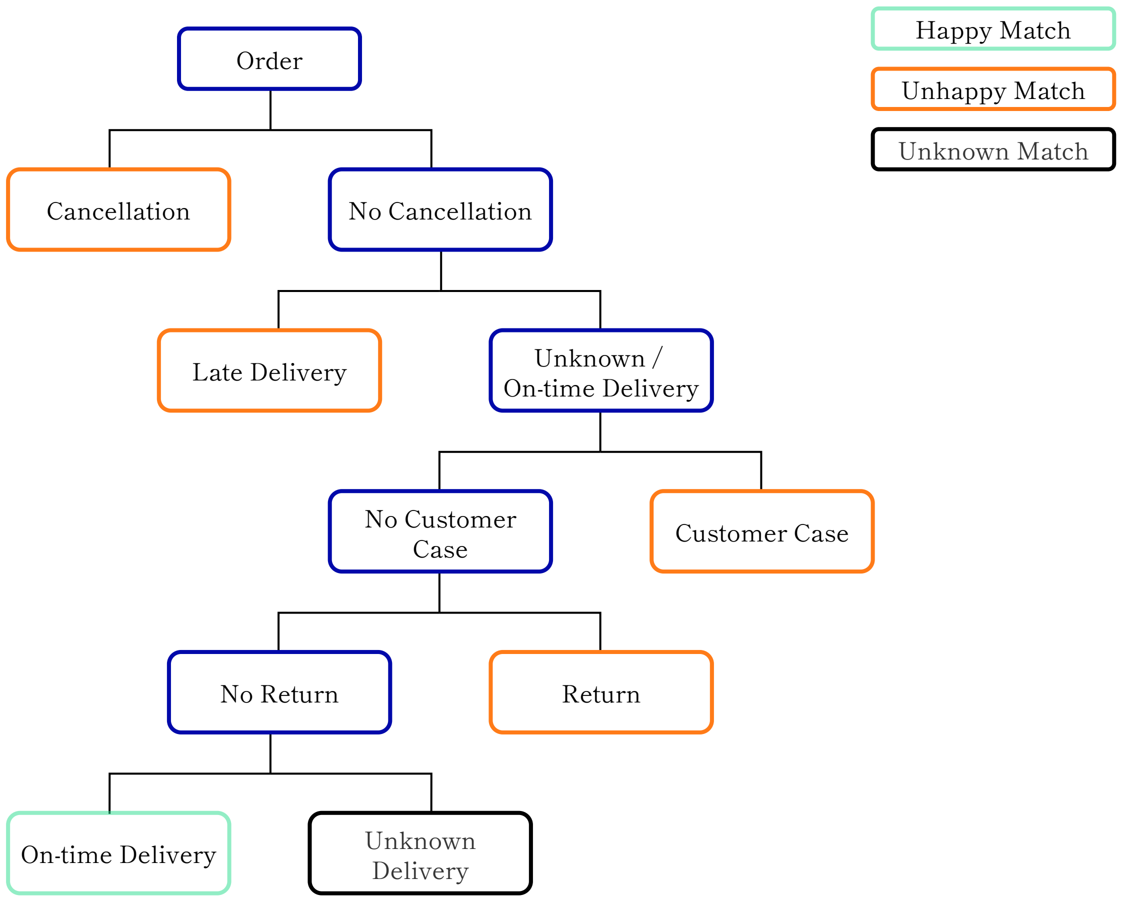


Figure C.1. *Happy Match Decision Tree.*

Appendix D List of Original Data Set Variables

Below, we present a list of all variables from the original data set with their corresponding descriptions. This list consists of (i) descriptive variables, (ii) match determinants, and (iii) match labels (output). We present these variables in the same order as they are listed in the original data set.

1. Original set of descriptive variables

- *orderDate*. The date on which a product order is placed (e.g. 2019-11-30).
- *productId*. An identifier for the product that is ordered.
- *sellerId*. An identifier for the seller from which the ordered product originates.
- *totalPrice*. Total price of the products ordered (product price times quantity ordered), represented in euros.
- *quantityOrdered*. Number of times the product is contained in the order.
- *countryCode*. Country from which the order is placed, either the Netherlands (NL) or Belgium (BE).
- *cancellationDate*. Date on which an order is cancelled, NULL if not (yet) cancelled.
- *cancellationReasonCode*. Reason for the cancellation of the order (e.g., TECH_ISSUE), NULL if not (yet) cancelled.
- *promisedDeliveryDate*. Date on which the order should be delivered to the customer.
- *shipmentDate*. Date on which the partner confirms the shipment of the order. If NULL, the shipment date is unknown.
- *transporterCode*. Code for the transporter that handles the order (e.g., DHL).
- *transporterName*. Full name of the transporter handling the order (e.g., DHL Global Mail).
- *transporterNameOther*. Column indicating whether *transporterCode* = 'Anders' (Other), NULL when this is not the case.
- *dateTimeFirstDeliveryMoment*. Timestamp on which the order is delivered.
- *fulfilmentType*. Code indicating whether the platform or the retailer handled the order (FBB or FBR, respectively).
- *startDateCase*. Date on which the first customer case belonging to the product order is started. Can be NULL if no case was started.

- *cntDistinctCaseIds.* Number of unique customer cases belonging to the product order, NULL if no case was started.
- *returnDateTime.* Date (not a timestamp) on which the return of a product order has been registered, NULL if product is not returned.
- *quantityReturn.* Quantity that is returned by the customer.
- *returnCode.* Reason for return that is logged internally (e.g., WRONG_SIZE_ORDERED).
- *productTitle.* Title of the product that is ordered.
- *brickName.* Name of the brick (internal categorization) to which the product belongs.
- *chunkName.* Name of the chunk (internal categorization) to which the product belongs.
- *productGroup.* Category to which the product belongs (level 1).
- *productSubGroup.* Category to which the product belongs (level 2).
- *productSubSubGroup.* Category to which the product belongs (level 3).
- *registrationDateSeller.* Date on which the seller was registered at the platform.
- *countryOriginSeller.* Country from which the seller originates.
- *currentCountryAvailabilitySeller.* Current countries the seller ships to. Can be NL (the Netherlands), BE (Belgium), or ALL (both).
- *calculationDefinitive.* Boolean representing whether 30 days after the order date have passed.

2. Match determinants

Note that these variables cannot be used for classification as they directly relate to the match labels.

- *noCancellation.* Boolean indicating whether the product order was cancelled (0) or not (1) within 10 days of ordering, 1 if cancelled and 0 otherwise.
- *onTimeDelivery.* Variable indicating whether the product order was delivered on time (*promisedDelivery-Date* on or before *dateTimeFirstDeliveryMoment*), NULL if delivery is unknown. Based on the first 13 days from the order date.
- *noCase.* Boolean indicating whether a customer case was started (0) or not (1) within 30 days after ordering.
- *hasOneCase.* Boolean indicating whether there was a customer case (1) or not (0) within 30 days after ordering.

- *hasMoreCases.* Boolean indicating whether there were multiple customer cases (1) or not (0) within 30 days after the order date.
- *noReturn.* Boolean indicating whether the product order was registered for return (0) or not (1) within 30 days after the order date.

3. Match labels

- *generalMatchClassification.* Classification system of product orders, which includes the labels (i) Happy, (ii) Unhappy, and (iii) Unknown.
- *detailedMatchClassification.* Classification system of product orders, which includes the labels (i) Happy, (ii) Mildly Unhappy, (iii) Medium Unhappy, (iv) Heavily Unhappy, and (v) Unknown.

Appendix E List of Model Variables

Below, we present a list of all variables that we use in our classification methods along with their corresponding descriptions.

1. Dependent variables

We focus on the *detailedMatchClassification*, but in fact two types of match labels are defined for product orders.

- *generalMatchClassification.* Classification system of product orders, which includes the labels (i) Happy, (ii) Unhappy, and (iii) Unknown.
- *detailedMatchClassification.* Classification system of product orders, which includes the labels (i) Happy, (ii) Mildly Unhappy, (iii) Medium Unhappy, (iv) Heavily Unhappy, and (v) Unknown.

2. Fixed explanatory variables

- *totalPrice.* Total price of the products ordered (product price times quantity ordered), represented in euros.
- *quantityOrdered.* Number of times the product is contained in the order.
- *fulfilmentByPlatform.* Dummy indicating whether the product order is fulfilled by the platform (1) or not (0).
- *countryCodeNL*: Dummy indicating whether the customer originates from the Netherlands (1) or Belgium (0).
- *countryOriginNL.* Dummy indicating whether the partner originates from the Netherlands (1) or elsewhere (0).
- *countryOriginBE.* Dummy indicating whether the partner originates from Belgium (1) or elsewhere (0).
- *countryOriginDE.* Dummy indicating whether the partner originates from Germany (1) or elsewhere (0).
- *productTitleLength.* The character length of the *productTitle* belonging to a product.
- *partnerSellingDays.* The number of days between *registrationDateSeller* and *orderDate.*
- *promisedDeliveryDays.* The number of days between *orderDate* and *promisedDeliveryDate.*
- *orderMonth.* Month-specific dummy for the 12 months (*orderJanuary*, *orderFebruary*, ...), taking on value 1 if an order date falls in the corresponding month and taking on value 0 otherwise.
- *orderWeekday.* Weekday-specific dummy for the 7 weekdays (*orderMonday*, *orderTuesday*, ...), taking on value 1 if an order date falls on the corresponding day of the week and taking on value 0 otherwise.

- *orderYear2020*. Dummy indicating whether or not an order date falls in 2020 (value 1 if it does, value 0 otherwise).
- *orderCorona*. Dummy indicating whether the order is placed in the pre- (0) or post-corona period (1). The post-corona period is taken to have started on 21 March 2020.
- *productGroupDummy*. Dummy for a specific product group (from *productGroupGeneral*) belonging to the product order, 1 if the order belongs to the product group and 0 otherwise.

3. Dynamic explanatory variables

- *cancellationKnownX*. Dummy indicating whether there was a known cancellation in the first X days after the date of ordering, taking value 1 if this is the case and 0 otherwise.
- *onTimeDeliveryKnownX*. Dummy indicating whether it is known whether a product was delivered on time X days after the date of ordering, taking value 1 if this is the case and 0 otherwise.
- *lateDeliveryKnownX*. Dummy indicating whether it is known if a product was delivered late X days after the date of ordering, taking value 1 if this is the case and 0 otherwise.
- *caseKnownX*. Dummy indicating whether there was a known case started in the first X days after the date of ordering, taking value 1 if this is the case and 0 otherwise.
- *returnKnownX*. Dummy indicating whether there was a known return of the product order in the first X days after the date of ordering, taking value 1 if this is the case and 0 otherwise.
- *transporterGroupX*. Dummy indicating whether it is known to what group of transporters (from *transporterGroupGeneral*) a product order belongs X days after the order date. The groups are (i) POSTNL, (ii) DHL, (iii) DPD, (iv) Briefpost, and (v) Other. For each group, a specific dummy is created, taking on value 1 if the order is known to belong to that group X days after the order date and 0 otherwise.
- *productOrderCountX*. Number of orders for which a certain product has been included, as measured X days after a specific order date.
- *productTotalCountX*. Number of ordered items for a certain product, as measured X days after a specific order date.
- *productTotalReturnedX*. Number of returned items for a certain product, as measured X days after a specific order date.
- *productReturnFractionX*. Fraction of ordered items that have been returned for a certain product, as measured X days after a specific order date.
- *sellerDailyOrdersX*. Average daily number of orders for which a product from a certain seller is included, as measured X days after a specific order date.

- *transporterCodeHistoricHappyX*. Fraction of orders for a transporter group for which the *generalMatchClassification* is known to be happy, as measured X days after a specific order date.
- *transporterCodeHistoricUnhappyX*. Fraction of orders for a specific transporter group for which the *generalMatchClassification* is known to be unhappy, as measured X days after a specific order date.
- *transporterCodeHistoricUnknownX*. Fraction of orders for a specific transporter group for which the *generalMatchClassification* is known to be unknown, as measured X days after a specific order date.
- *sellerIdHistoricHappyX*. Fraction of orders for a certain seller for which the *generalMatchClassification* is known to be happy, as measured X days after a specific order date.
- *sellerIdHistoricUnhappyX*. Fraction of orders for a certain seller for which the *generalMatchClassification* is known to be unhappy, as measured X days after a specific order date.
- *sellerIdHistoricUnknownX*. Fraction of orders for a certain seller for which the *generalMatchClassification* is known to be unknown, as measured X days after a specific order date.
- *productGroupHistoricHappyX*. Fraction of orders for a certain product group for which the *generalMatchClassification* is known to be happy, as measured X days after a specific order date.
- *productGroupHistoricUnhappyX*. Fraction of orders for a certain product group for which the *generalMatchClassification* is known to be unhappy, as measured X days after a specific order date.
- *productGroupHistoricUnknownX*. Fraction of orders for a certain product group for which the *generalMatchClassification* is known to be unknown, as measured X days after a specific order date.

Appendix F Classification Methods

The methodology discussed and notation used in this section to outline the logistic regression and random forest classifiers is based on Hastie et al. [27]. In addition, note that, for both classification methods, we account for imbalanced data by training the classifiers such that larger weights are assigned to classes that are less frequent in the data set. More specifically, we implement so-called *balanced* class weights, indicating that the weights assigned to a specific class are inversely proportional to the rate of occurrence of this class.

F.1 Logistic Regression

In the following, we assume that we need to classify instances based on only one predictor variable x. Of course, the presented methodology generalises to the case of multiple predictor variables. Our predictor $G(x)$ takes values in a discrete set $\mathcal{G} = \{1, 2, \dots, K\}$. In other words, there are K classes, and we construct decision boundaries to separate the input space into a collection of K regions labelled according to the classification. The set of classification procedures with linear decision boundaries is called *linear methods of classification*. In particular, for each class $k \in \mathcal{G}$, we fit a linear model, denoted by $f_k(x) = \beta_{0k} + \beta_{1k}x$. Then, the linear decision boundary between any two classes k and l is given by the set of points for which $f_k(x) = f_l(x)$, that is, the affine set $\{x : (\beta_{0k} - \beta_{0l}) + (\beta_{1k} - \beta_{1l})x = 0\}$. Methods that model linear discriminant functions $\delta_k(x)$, or methods that model posterior probabilities $\Pr(G = k|X = x)$ that are linear in x both belong to the linear methods of classification.

The *linear logistic regression model* is well-suited to model the posterior probabilities of the K classes via linear functions in x, while simultaneously ensuring that the probabilities remain in $[0, 1]$ and sum to one. Actually, for the decision boundaries to be linear, all we require is that some monotone transformation of the posterior probabilities $\Pr(G = k|X = x)$ be linear. For example, if $K = 2$, then a popular model for the posterior probabilities is a logit model, which is given by

$$\Pr(G = 1|X = x) = \frac{\exp(\beta_0 + \beta_1 x)}{1 + \exp(\beta_0 + \beta_1 x)}, \tag{F.1}$$

$$\Pr(G = 2|X = x) = \frac{1}{1 + \exp(\beta_0 + \beta_1 x)}. \tag{F.2}$$

A logit transformation of the posterior probabilities can be used to get linear decision boundaries. If we apply this logit transformation, we find the *log-odds*, which is given by

$$\log \frac{\Pr(G = 1|X = x)}{\Pr(G = 2|X = x)} = \beta_0 + \beta_1 x. \tag{F.3}$$

The resulting linear decision boundary is the set of points for which the log-odds is zero, and this is a hyperplane defined by $\{x : \beta_0 + \beta_1 x = 0\}$.

For convenience and simplicity, in the following derivations, we continue with the two-class case ($K = 2$). However, all derivations easily generalise to the multi-class case. We denote the posterior probabilities $\Pr(G = k|X = x) = p_k(x; \theta)$, for $k = 1, 2$, where $\theta = \{\beta_0, \beta_1\}$ is the set of parameters. We fit logistic regression models by maximum likelihood using the conditional likelihood of G given X. The log-likelihood function for N observations is given by

$$\ell(\theta) = \sum_{i=1}^{N} \log p_{g_i}(x_i;\theta), \tag{F.4}$$

where $p_k(x;\theta) = \Pr(G = k|X = x_i;\theta)$. For convenience, we code the class variable G as a binary response variable Y, where $y_i = 1$ when $g_i = 1$, and $y_i = 0$ when $g_i = 2$. If we let $p_1(x;\theta) = p(x;\theta)$, and $p_2(x;\theta) = 1 - p(x;\theta)$, then we can write the log-likelihood as follows:

$$\ell(\beta) = \sum_{i=1}^{N} \left\{y_i \log p\left(\mathbf{x}_i;\beta\right) + (1-y_i)\log\left(1-p\left(\mathbf{x}_i;\beta\right)\right)\right\} \tag{F.5}$$

$$= \sum_{i=1}^{N} \left\{y_i\beta'\mathbf{x}_i - \log\left(1+e^{\beta'\mathbf{x}_i}\right)\right\}, \tag{F.6}$$

where $\beta = (\beta_0, \beta_1)'$ and $\mathbf{x}_i = (1, x_i)'$. To maximise the log-likelihood, we set its derivatives to zero, which gives us the *score equations*

$$\frac{\partial \ell(\beta)}{\partial \beta} = \sum_{i=1}^{N} \mathbf{x}_i\left[y_i - p\left(\mathbf{x}_i;\beta\right)\right] = 0, \tag{F.7}$$

which are in this case two equations nonlinear in β. To solve the score equations, we use a coordinate descent algorithm that is implemented using the *liblinear* solver [28].

In some cases, it may be desirable to shrink the linear regression model to a model with only a subset of the p predictors. The shrunk model may be better interpretable and have lower prediction error than the full model. This process is typically called *shrinkage* or *regularisation*. Two well-known shrinkage techniques are L_1- and L_2-regularisation. In L_1-regularisation, an L_1-penalty is used to shrink the linear regression model. For logistic regression, we maximise the L_1-penalised version of (F.6), which is given by

$$\max_{\beta}\left\{\sum_{i=1}^{N}\left[y_i\beta'\mathbf{x}_i - \log\left(1+e^{\beta'\mathbf{x}_i}\right)\right] - \lambda\sum_{j=1}^{p}|\beta_j|\right\}, \tag{F.8}$$

where $\beta = (\beta_0, \beta_1, \ldots, \beta_p)'$ and $\mathbf{x}_i = (1, x_{1i}, x_{2i}, \ldots, x_{pi})'$. In addition, $\lambda \geq 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of λ, the larger the amount of shrinkage.

In L_2-regularisation, an L_2-penalty is used to shrink the linear regression model. For logistic regression, we maximise the L_2-penalised version of (F.6), which is given by

$$\max_{\beta}\left\{\sum_{i=1}^{N}\left[y_i\beta'\mathbf{x}_i - \log\left(1+e^{\beta'\mathbf{x}_i}\right)\right] - \lambda\sum_{j=1}^{p}\beta_j^2\right\}. \tag{F.9}$$

Typically, in both regularisation methods, we do not penalise the intercept term, and standardise the predictor variables for the penalty to be meaningful. Both regularisation methods are similar, in that they impose a penalty in the log-likelihood function to shrink the linear regression model. However, they have some subtle differences, since the L_1-penalty constraints the sum of the absolute values of the coefficients and the L_2-penalty constraints the sum of the squares of the coefficients. As a result, L_1-regularisation forces the coefficients of some predictor variables towards zero, whereas L_2-regularisation only lowers the sizes of coefficients, without forcing any of them to become zero. That is, L_1-regularisation essentially drops some predictor variables from the model, whereas L_2-regularisation only lowers coefficient sizes to avoid overfitting. Hence, L_1-regularisation may do well in models with many predictor variables (it makes the model better interpretable), whereas L_2-regularisation performs well in models with less, but highly correlated predictor variables (it makes the model more efficient).

F.2 Random Forest

Tree-based methods partition the predictor space into M rectangular regions $R_1, \ldots, R_M$ using specific splitting rules, and then fit a simple model in each region. We again assume that we face a multi-class classification problem, where the data is given by $(\mathbf{x}_i, y_i)$ for $i = 1, 2, \ldots, N$, with predictor variables $\mathbf{x}_i = (x_{1i}, x_{2i}, \ldots, x_{pi})'$ and response variable $y_i \in \{1, 2, \ldots, K\}$. In order to divide the predictor space into regions, the algorithm needs to decide on splitting variables, split points, and the shape of the tree. Starting with all data, for each splitting variable, an optimal split point according to some criterion. Among all predictor variables, the best split is chosen and the data is partitioned in two regions. This splitting process is again repeated on the two regions, and eventually on all of the resulting regions. In this way, the tree grows and achieves a certain *depth*. Tree size is a hyperparameter that determines the mode's complexity. The optimal tree size should be chosen from the data. However, in general, a large tree tends to overfit the data, while a small tree has the risk of not capturing the most important data structures.

In a particular node m of the tree, which represents a certain region R_m with N_m observations, we let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} I(y_i = k), \tag{F.10}$$

denote the proportion of observations from class k in node m, where $I(A)$ is an indicator function that is 1 if A is true and 0 otherwise. We classify the observations in node m to the class with the highest proportion of observations in node m, that is, to the majority class $k(m) = \arg\max_k \hat{p}_{mk}$. The selection of the optimal split for each predictor variable can be based on different measures of node impurity, for which we apply the Gini index (G), which is given as follows:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}). \tag{F.11}$$

In *random forests*, multiple trees are grown on bootstrapped samples from the training data, and the idea is to reduce the variance of the estimator by decorrelating the trees [27]. This decorrelation between trees is achieved through random selection of the input variables: before each split, a random selection of $m \leq p$ input variables is considered as candidates for splitting. Typically, we use $m = \lfloor \sqrt{p} \rfloor$. Hence, each time a random-forest tree T_b is grown on a bootstrapped sample of the training data, the following procedure is repeated for each terminal node of the tree, until some stopping condition is met (e.g., a minimum node size is reached):

1. Randomly select m predictor variables from the set of p variables. Usually, we set $m = \lfloor \sqrt{p} \rfloor$.
2. Pick the best split point among the m predictor variables based on some measure of node impurity (e.g., G).
3. Split the node into two child nodes.

The above procedure is repeated for all nodes in B random-forest trees, so that eventually the random forest consists of an ensemble of B (decorrelated) trees, which we denote by $\{T_b\}_{b=1}^{B}$. The trained random forest can then be used to make a prediction at a new point $\mathbf{x}$. Each random-forest tree outputs an own class prediction, and the final random-forest prediction is the majority class among the B individual-tree predictions. That is, if $\hat{C}_b(\mathbf{x})$ denotes the class prediction of the b-th random forest, then $\hat{C}_{RF}(\mathbf{x}) = \text{majority vote}\{\hat{C}_b(\mathbf{x})\}_{b=1}^{B}$ denotes the final random-forest class-prediction at the point $\mathbf{x}$.

Appendix G Specification of Hierarchical Performance Measures

Global predictive accuracy (Acc), *global hierarchical precision* (hP) and *global hierarchical recall* (hR) are defined as follows:

$$Acc = \frac{1}{N}\sum_{i=1}^{N}\left|\mathcal{A}(l_i)\cap\mathcal{A}(\hat{l}_i)\right|, \qquad hP = \frac{\sum_{i=1}^{N}\left|\mathcal{A}(l_i)\cap\mathcal{A}(\hat{l}_i)\right|}{\sum_{i=1}^{N}\left|\mathcal{A}(\hat{l}_i)\right|}, \qquad hR = \frac{\sum_{i=1}^{N}\left|\mathcal{A}(l_i)\cap\mathcal{A}(\hat{l}_i)\right|}{\sum_{i=1}^{N}\left|\mathcal{A}(l_i)\right|}, \tag{G.1}$$

where l_i and $\hat{l}_i$ denote the true and the predicted label of instance i, respectively, and $\mathcal{A}(l)$ denotes the set containing leaf node $l \in \mathcal{L}$ and all its ancestor nodes except the root node R. We use the term 'global' here, because these performance measures are calculated over all N test instances. The *global hierarchical F_1-measure* (hF_1) is defined by the harmonic mean of the hP and hR measures, or:

$$hF_1 = \frac{2\cdot hP\cdot hR}{hP+hR}. \tag{G.2}$$

We define *class-specific hierarchical precision* (hP_c) and class-specific hierarchical recall (hR_c) as follows:

$$hP_c = \frac{\sum_{i\in\mathcal{P}_c}\left|\mathcal{A}(l_i)\cap\mathcal{A}\left(\hat{l}_i\right)\right|}{\sum_{i\in\mathcal{P}_c}\left|\mathcal{A}\left(\hat{l}_i\right)\right|}, \qquad hR_c = \frac{\sum_{i\in\mathcal{T}_c}\left|\mathcal{A}(l_i)\cap\mathcal{A}\left(\hat{l}_i\right)\right|}{\sum_{i\in\mathcal{T}_c}\left|\mathcal{A}\left(l_i\right)\right|}, \tag{G.3}$$

where $\mathcal{P}_c$ denotes the set of predicted instances of class $c \in \mathcal{L}$, and $\mathcal{T}_c$ denotes the set of true instances of class $c \in \mathcal{L}$. The *class-specific hierarchical $F_{1,c}$-measure* aggregates hP_c and hR_c by taking their harmonic mean and is defined as follows:

$$hF_{1,c} = \frac{2\cdot hP_c\cdot hR_c}{hP_c+hR_c}. \tag{G.4}$$

We define *daily accuracy* (Acc_t), *daily hierarchical precision* (hP_t) and *daily hierarchical recall* (hR_t) as follows:

$$Acc_t = \frac{1}{N_t}\sum_{i=1}^{N_t}\left|\mathcal{A}(l_i)\cap\mathcal{A}(\hat{l}_i)\right|, \qquad hP_t = \frac{\sum_{i=1}^{N_t}\left|\mathcal{A}(l_i)\cap\mathcal{A}\left(\hat{l}_i\right)\right|}{\sum_{i=1}^{N_t}\left|\mathcal{A}\left(\hat{l}_i\right)\right|}, \qquad hR_t = \frac{\sum_{i=1}^{N_t}\left|\mathcal{A}(l_i)\cap\mathcal{A}\left(\hat{l}_i\right)\right|}{\sum_{i=1}^{N_t}\left|\mathcal{A}\left(l_i\right)\right|}, \tag{G.5}$$

where N_t denotes the number of instances that is assigned a (leaf) class $c \in \mathcal{L}$ on day $t \in \{0, 1, \ldots, 10\}$. The *daily hierarchical $F_{1,t}$-measure* aggregates hP_t and hR_t by taking their harmonic mean and is defined as follows:

$$hF_{1,t} = \frac{2\cdot hP_t\cdot hR_t}{hP_t+hR_t}. \tag{G.6}$$

We define *class-specific daily hierarchical precision* ($hP_{c,t}$) and *class-specific daily hierarchical recall* ($hR_{c,t}$) as follows:

$$hP_{c,t} = \frac{\sum\limits_{i \in \mathcal{P}_{c,t}} \left| \mathcal{A}(l_i) \cap \mathcal{A}\left(\hat{l}_i\right) \right|}{\sum\limits_{i \in \mathcal{P}_{c,t}} \left| \mathcal{A}\left(\hat{l}_i\right) \right|}, \qquad hR_{c,t} = \frac{\sum\limits_{i \in \mathcal{T}_{c,t}} \left| \mathcal{A}(l_i) \cap \mathcal{A}\left(\hat{l}_i\right) \right|}{\sum\limits_{i \in \mathcal{T}_{c,t}} \left| \mathcal{A}\left(l_i\right) \right|}, \tag{G.7}$$

where $\mathcal{P}_{c,t}$ denotes the set of predicted instances of class $c \in \mathcal{L}$ that is assigned a leaf class on day $t \in \{0, 1, \ldots, 10\}$, and $\mathcal{T}_{c,t}$ denotes the set of true instances of class $c \in \mathcal{L}$ that is assigned a leaf class on day $t \in \{0, 1, \ldots, 10\}$. The *class-specific daily hierarchical $F_{1,c,t}$-measure* aggregates $hP_{c,t}$ and $hR_{c,t}$ by taking their harmonic mean and is defined as follows:

$$hF_{1,c,t} = \frac{2 \cdot hP_{c,t} \cdot hR_{c,t}}{hP_{c,t} + hR_{c,t}}. \tag{G.8}$$

Appendix H Results of Hyperparameter Optimisation

In this research, we use Bayesian hyperparameter optimisation on the train/validation data to find our best classifiers, a technique which we present by means of background, implementation, and results.

H.1 Background

In hyperparameter tuning, we aim to find the combination of hyperparameters that minimises some objective function or maximise some score on our validation set. However, the way in which this score depends on our hyperparameters is unknown. Brute force techniques (e.g., grid search) therefore extensively try out combinations of hyperparameters and select the combination yielding the lowest error (highest score). In contrast, Bayesian optimisation techniques tune hyperparameters by taking into account the results of past evaluations, potentially leading to better outcomes and faster optimisation [29].

Bayesian techniques aim to approximate the function that represents the dependence between hyperparameters and error by a so-called *surrogate function*. This function is built up based on past evaluations of hyperparameters and their respective error and updated with each validation iteration. The function that we use as surrogate is the Tree-structured Parzen Estimator (TPE), which models different distributions of hyperparameters based on some split in the value of the objective function [30]. In addition to a surrogate function, Bayesian optimisation techniques apply a so-called *selection function*, which selects the set of hyperparameters to be evaluated in the next iteration based on the information provided by the surrogate. By in turn evaluating a set of hyperparameters, updating the surrogate, and applying the selection function, we can iteratively find 'better' hyperparameters and optimise our validation error.

H.2 Implementation

To implement TPE Bayesian hyperparameter optimisation, we use Python's *hyperopt* package. As our selection function, we apply the *expected improvement* criterion, while the error that we aim to minimise is $-1 \times hF$ (such that we maximise the hF-score). In terms of hyperparameters and their parameter space, for LR we solely optimise over the regularisation type, i.e., either L1- or L2-regularisation. As for RF, we tune the number of tree estimators (*N estimators*) and the maximum depth (*max depth*) of those trees. As prior for both hyperparameters, we choose to apply a uniform integer distribution. The number of estimators is chosen to range between 10 and 50 (with stepsize 5), while the maximum depth of our trees is set between 5 and 15 (with stepsize 1). For CAT-HCOT algorithm, we implement hyperparameter optimisation on all 8 possible classifier combinations. This means the number of parameters to optimise ranges between 3 (LR-LR-LR) and 6 (RF-RF-RF). In the flat baseline, we either need to optimise 1 (LR) or 2 (RF) parameters. We set the maximum number of optimisation iterations to find our best hyperparameters to 20. Note, however, that in case we only evaluate LR, we do not need 20 iterations to evaluate all possible hyperparameters.

H.3 Results

Below, we present the hyperparameter optimisation results for CAT-HCOT and the static baseline method. First, we show the optimal sets of hyperparameters. Second, we provide the optimal hF-scores for all classifier combinations over time.

Table H.1
Optimal hyperparameters for classifier combination LR-RF-RF in CAT-HCOT.

	LR	**RF**		**RF**	
	Penalty[a]	Max depth	N estimators	Max depth	N estimators
Day 0	0	9	35	14	20
Day 1	0	10	45	14	45
Day 2	1	12	30	14	30
Day 3	1	12	30	14	30
Day 4	1	12	30	14	30
Day 5	1	12	30	14	30
Day 6	0	10	45	14	45
Day 7	0	10	45	14	45
Day 8	0	10	45	14	45
Day 9	0	10	45	14	45
Day 10	0	10	45	14	45

[a] 0 indicates L1-regularisation, 1 indicates L2-regularisation.

Table H.2
Optimal hyperparameters for the flat CAT-HCOT baseline.

	LR	**RF**	
	Penalty[a]	Max depth	N estimators
Day 0	0*	14	45
Day 1	0*	14	45
Day 2	0*	14	40
Day 3	0*	14	45
Day 4	0*	14	40
Day 5	0	14*	45*
Day 6	1	14*	40*
Day 7	0	14*	45*
Day 8	0	14*	45*
Day 9	0	14*	45*
Day 10	0	14*	45*

[a] 0 indicates L1-regularisation, 1 indicates L2-regularisation.
[*] indicates that the respective classifier is the optimal choice on the respective day.

Table H.3

Hierarchical F_1-scores for eight combinations of classifiers with optimal hyperparameters, as evaluated on the validation set.

	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10	Average
LR-LR-LR	79.02%	83.08%	88.89%	92.15%	94.56%	95.94%	96.73%	97.30%	97.69%	97.96%	98.18%	92.86%
LR-LR-RF	79.04%	83.17%	89.07%	92.42%	94.87%	96.31%	97.12%	97.70%	98.07%	**98.34%**	**98.53%**	93.15%
LR-RF-LR	83.46%	86.18%	90.32%	92.84%	94.89%	96.04%	96.76%	97.32%	97.70%	97.96%	98.18%	93.79%
LR-RF-RF	83.55%	86.05%	90.30%	**93.02%**	**95.19%**	**96.40%**	**97.15%**	**97.71%**	**98.08%**	**98.34%**	**98.53%**	**94.03%**
RF-LR-LR	79.75%	83.44%	88.98%	92.08%	94.31%	95.82%	96.55%	97.23%	97.57%	97.84%	98.05%	92.87%
RF-LR-RF	79.66%	83.50%	88.94%	92.25%	94.50%	95.95%	96.80%	97.19%	97.75%	97.85%	98.17%	92.96%
RF-RF-LR	84.49%	**87.03%**	**90.33%**	92.70%	94.62%	95.73%	96.50%	96.86%	97.40%	97.56%	97.85%	93.73%
RF-RF-RF	**84.62%**	86.80%	90.10%	92.70%	94.73%	96.02%	96.80%	97.29%	97.64%	97.97%	98.10%	93.89%

Note: Bold numbers indicate the best classifier combination(s) on a specific day based on validation hF_1-score.

Table H.4

Hierarchical F_1-scores for the flat CAT-HCOT baseline LR and RF classifiers with optimal hyperparameters, as evaluated on the validation set.

	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10	Average
LR	**81.16%**	**84.15%**	**88.55%**	**91.37%**	**93.37%**	94.62%	95.46%	96.05%	96.45%	96.73%	96.94%	92.26%
RF	80.39%	83.44%	88.02%	91.05%	93.36%	**94.92%**	**95.92%**	**96.84%**	**97.48%**	**97.85%**	**98.10%**	92.49%

Note: Bold numbers indicate the best classifier combination(s) on a specific day based on validation hF_1-score.

Appendix I Visualisations for the CAT Algorithm

I.1 Terminology

We present Figure I.1 to ease the interpretation of the terminology used in the CAT algorithm. We show the same tree with on the left the relations between the ancestry sets and on the right the relations between the training sets.

The ancestry sets for node m used in CAT are p_m and $\mathcal{S}_m$. The set p_m contains the parent node of m and always has a cardinality of 1 in our problem setting. The set $\mathcal{S}_m$ contains all sibling nodes of m, which are all nodes different from m that are on the same level and have the same parent node. We show an example for the ancestry sets of node <1>. The parent node is the root node, <R>, and the set of siblings contains only one node in this case, which is <0>. Then, the probability $\gamma_{p_m \to m,t}(u_j)$ given by the classifier at the root node for instance u_j on day t is $\gamma_{R\to 1,t}(u_j)$. The probability $\gamma_{R\to 1,t}(u_j)$ is included in the majority vote probabilities set $\mathcal{Z}$ if $\gamma_{R\to 1,t}(u_j) > \gamma_{R\to 0,t}(u_j)$.

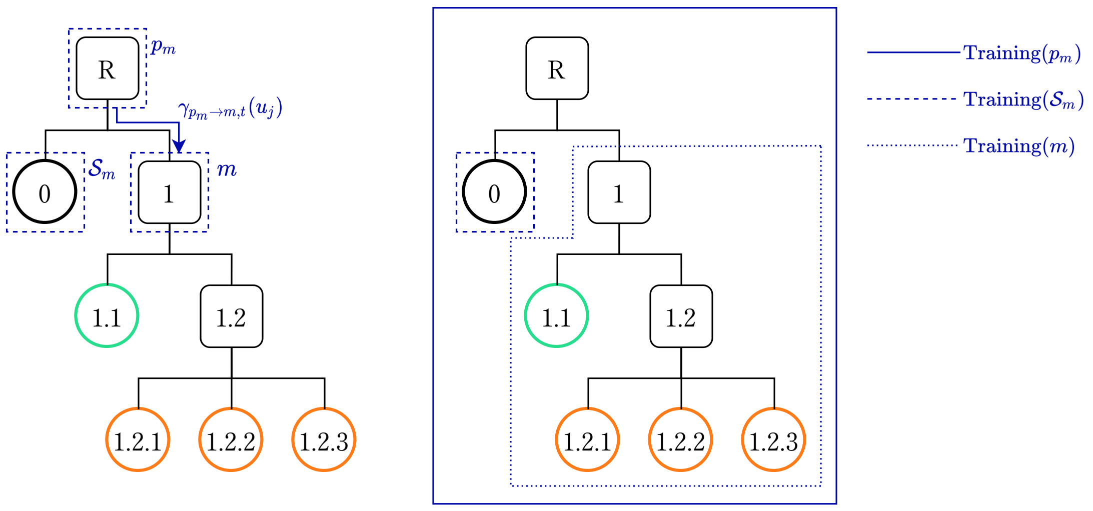


Figure I.1. *Relations between ancestry sets and between training sets.*

Next, we define the training sets Training(m), Training(p_m), and Training($\mathcal{S}_m$) for the respective ancestry sets m, p_m, and $\mathcal{S}_m$. The set Training($\mathcal{S}_m$) contains all training instances $u_j \in \mathcal{U}$ for which the true class $l_j \in \mathcal{L}$ belongs to one of the classes at the leaf nodes in the set containing the nodes in $\mathcal{S}_m$ and their descendants. Formally, this is defined as

$$\text{Training}(\mathcal{S}_m) = \bigcup_{m' \in \mathcal{S}_m} \left\{ \bigcup_{m'' \in (m' \cup \mathcal{D}_{m'}) \cap \mathcal{L}} \{u_j = (l_j, \mathbf{x}_j) \in \mathcal{U} \mid l_j = m''\} \right\}. \tag{I.9}$$

In our example, where m = <1>, the considered leaf nodes for Training(m), denoted by $(m \cup \mathcal{D}_m) \cap \mathcal{L}$, are given by the set {<1.1>, <1.2.1,>, <1.2.2>, <1.2.3>}. Thus, Training(<1>), the training set for the node Known, contains all instances for which the true class is either Happy or any of the Unhappy classes. Similarly, the considered leaf node for Training($\mathcal{S}_m$) = Training(<0>) is the only element in {<0>}, and for Training(p_m) = Training(<R>), the considered leaf nodes are all leaf nodes $\mathcal{L}$. Note that the union of Training(m) and Training($\mathcal{S}_m$) is equal to Training(p_m).

I.2 Effect of the Certainty Parameter on Thresholds

In Figure I.2, we present a visualisation of the effect of certainty parameter α on the thresholds. For three values of α, we show the streams of the majority vote probabilities in both $\zeta_t(m)$ and $\zeta_t(\neg m)$. These might affect the optimisation of the thresholds, which we depict by means of a simplified representation of the threshold optimisation with imaginary data. Again, for the example where the node in question is <1>, we note that the stream of majority vote probabilities in ζ_t(<1>) is never altered. Instead, the certainty decides which of the majority vote probabilities in $\zeta_t(\neg$ <1>) are disregarded when optimising the threshold on day t. Note that majority vote probabilities in $\zeta_t(\neg$ <1>) belong to instances that *should* be classified to <0>, because they have the label Unknown. If α increases, the disregarded part of $\zeta_t(\neg$ <1>) increases. This may push the threshold Th_t(<1>) upwards, dependent on the data at hand and the value of α.

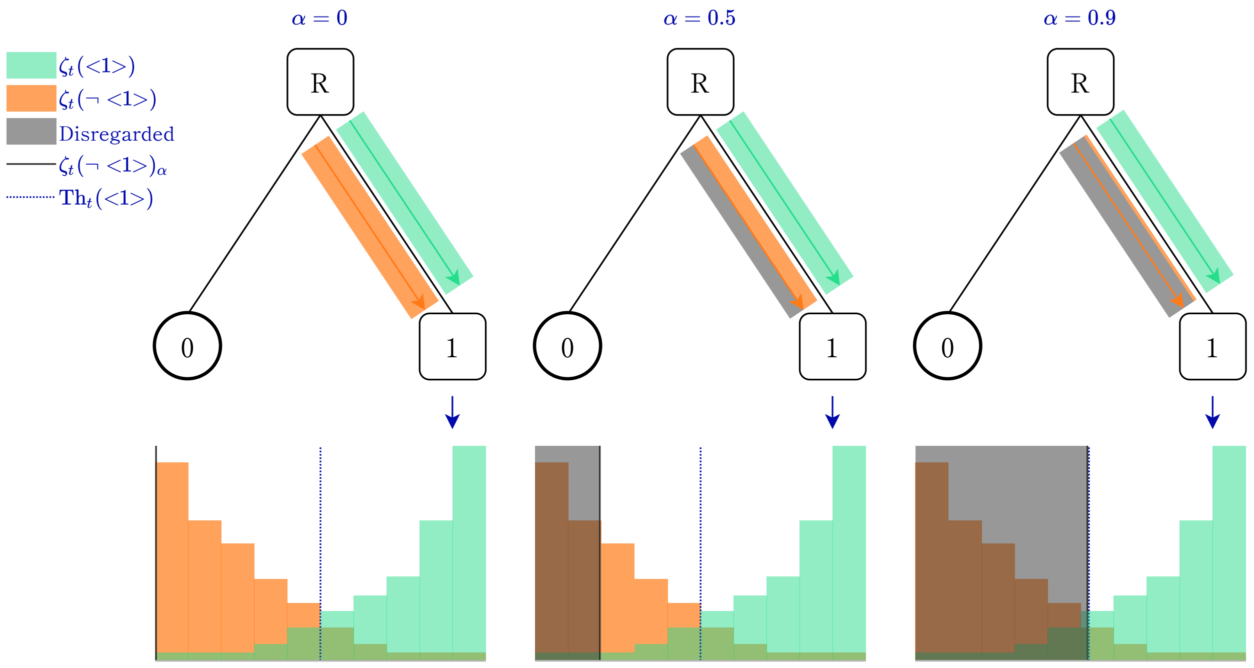


Figure I.2. *Visualisation of the effect of the certainty parameter α on the obtained threshold.*

I.3 Visual Example of the CAT Algorithm

To visualise the working of the CAT algorithm, we present Figures I.3-I.5. Each of these figures shows the distribution of the majority vote probability sets $\zeta_t(m)$ and $\zeta_t(\neg m)$. A probability in $\zeta_t(m)$ should be accepted, because the instance belongs to class m. In contrast, a probability in $\zeta_t(\neg m)$ should not be accepted, because the instance does not belong to m. The allowed search space $\mathcal{V}$ is represented by the shaded region in the figures. A small circle shows the F_1-measure evaluated in a point in the grid. The optimal threshold is presented by the bigger circle. The plots show the procedure of the algorithm for node

<1> (Known) as we move from day 0 (Figure I.3) to day 4 (Figure I.4) and, finally, to day 9 (Figure I.5). Note that, as we base the plots on a small sample of the data set, the plotted thresholds do not equal those reported in Section 5.

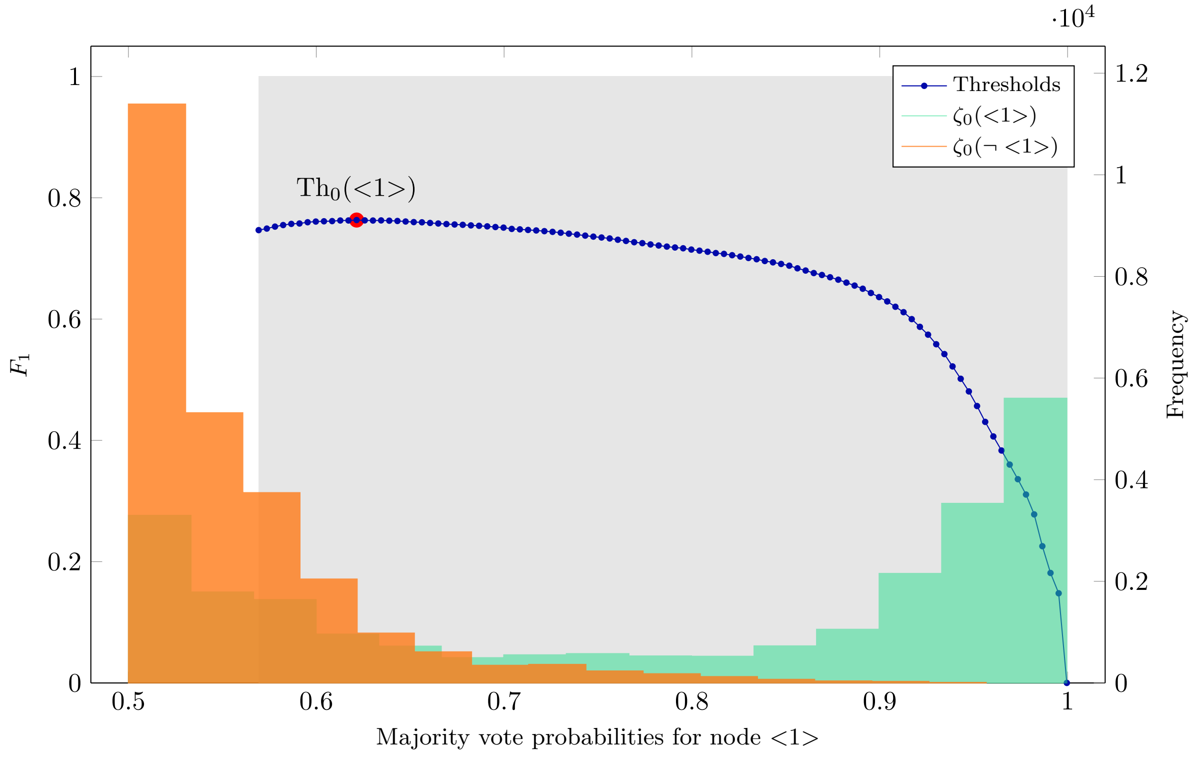


Figure I.3. *Visualisation of the distributions of $\zeta_0(<1>)$ and $\zeta_0(\neg <1>)$ alongside the optimisation of F_1 within $\mathcal{V}$.*

The lower bound is likely to increase over time, which often effectuates a similar pattern for the threshold. As more days pass, more information (potentially) comes in, making it easier to separate probabilities. In other words, the probabilities gravitate towards 0 and 1. A probability that moves towards 0 is not likely to remain a majority vote probability. However, a probability that moves towards 1 is likely to remain or become a majority vote probability. Since $\zeta_t(m)$ and $\zeta_t(\neg m)$ only contain majority vote probabilities, we expect their respective distributions to move towards 1. Next to posterior probabilities becoming more distinct, the predictive performance also increases. Therefore, we also expect that $\zeta_t(m)$ will increase in size, while $\zeta_t(\neg m)$ will decrease in size. Since the behaviour of the lower bound depends on the distribution of $\zeta_t(\neg m)$, the lower bound is likely to increase. This may push the threshold upwards as well. Whether the lower bounds increase depends on the data at hand and the certainty parameter α. The distributions can be subject to variability, as the remaining instances are classified the next day with an updated trained hierarchy. Furthermore, $\zeta_t(\neg m)$ is likely to thin out and cause the lower bound to become more sensitive. Indeed, the figures show that the lower bound increases as we move over time.

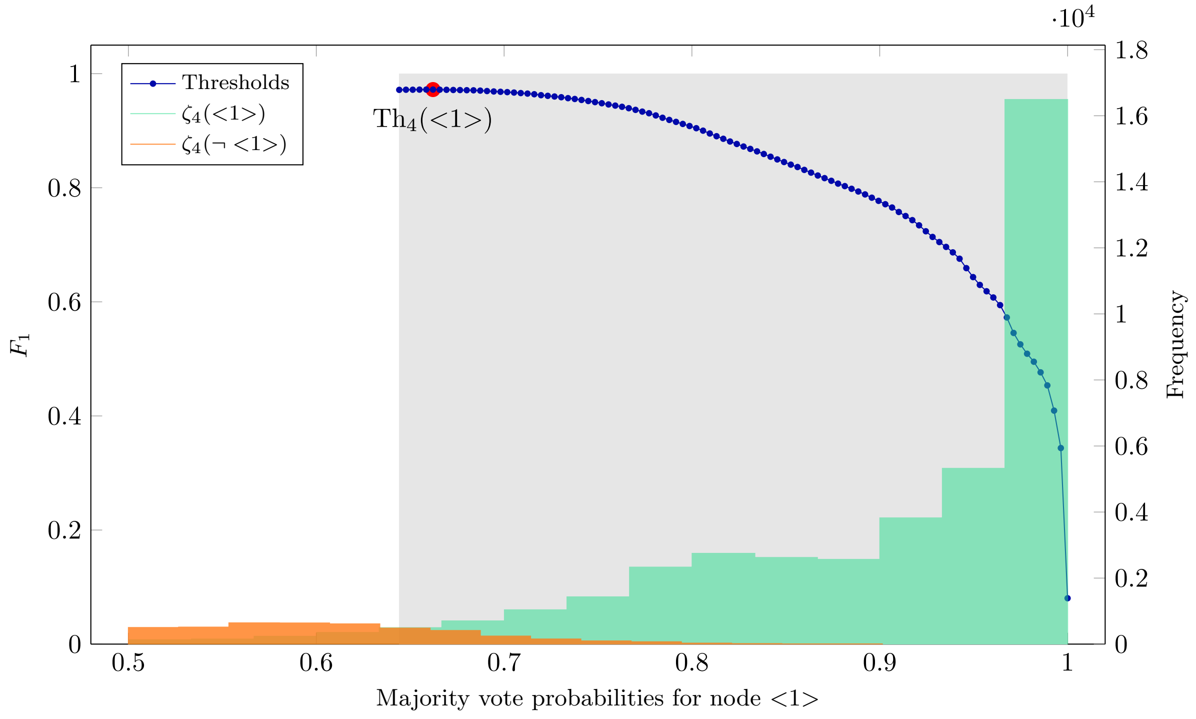


Figure I.4. *Visualisation of the distributions of $\zeta_4(<1>)$ and $\zeta_4(\neg <1>)$ alongside the optimisation of F_1 within $\mathcal{V}$.*

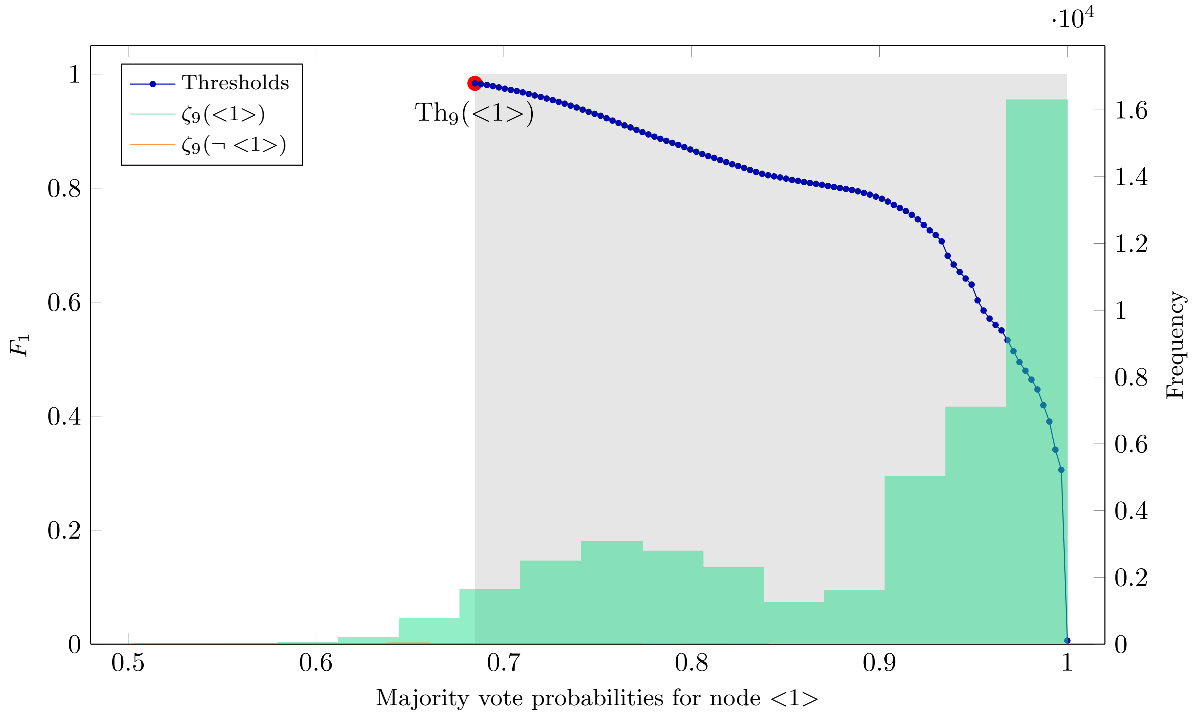


Figure I.5. *Visualisation of the distributions of $\zeta_9(<1>)$ and $\zeta_9(\neg <1>)$ alongside the optimisation of F_1 within $\mathcal{V}$.*

Appendix J Derivation of Importance Weights in F-measure

Here, we derive the effect that β has on the relation between false positives (FP) and false negatives (FN) in the F-measure F_β. With precision and recall defined as

$$precision = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}\,, \tag{J.10}$$

$$recall = \frac{\mathrm{TP}}{\mathrm{TP}+FN}\,, \tag{J.11}$$

F_β can be computed as

$$F_\beta = \frac{(1+\beta^2)\cdot precision \cdot recall}{\beta^2 \cdot precision + recall} \tag{J.12}$$

$$= \frac{(1+\beta^2)\cdot \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}} \cdot \frac{\mathrm{TP}}{\mathrm{TP}+FN}}{\beta^2 \cdot \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}} + \frac{\mathrm{TP}}{\mathrm{TP}+FN}} \tag{J.13}$$

$$= \frac{(1+\beta^2)\cdot \mathrm{TP}^2}{(\mathrm{TP}+\mathrm{FP})\cdot(\mathrm{TP}+FN)} \cdot \frac{(\mathrm{TP}+\mathrm{FP})\cdot(\mathrm{TP}+FN)}{\beta^2\cdot\mathrm{TP}\cdot(\mathrm{TP}+FN)+\mathrm{TP}\cdot(\mathrm{TP}+\mathrm{FP})} \tag{J.14}$$

$$= \frac{(1+\beta^2)\cdot\mathrm{TP}}{\beta^2\cdot(\mathrm{TP}+FN)+(\mathrm{TP}+\mathrm{FP})} \tag{J.15}$$

$$= \frac{(1+\beta^2)\cdot\mathrm{TP}}{(1+\beta^2)\cdot\mathrm{TP}+\beta^2\cdot FN+\mathrm{FP}}\,. \tag{J.16}$$

To express the importance, i.e., effect, that FN and FP have on F_β, we compute the first partial derivatives

$$\frac{\partial}{\partial FN}\left[\frac{(1+\beta^2)\cdot\mathrm{TP}}{(1+\beta^2)\cdot\mathrm{TP}+\beta^2\cdot FN+\mathrm{FP}}\right] = \left[-\frac{(1+\beta^2)\cdot\mathrm{TP}}{\left((1+\beta^2)\cdot\mathrm{TP}+\beta^2\cdot FN+\mathrm{FP}\right)^2}\right]\cdot\beta^2 \tag{J.17}$$

$$\frac{\partial}{\partial \mathrm{FP}}\left[\frac{(1+\beta^2)\cdot\mathrm{TP}}{(1+\beta^2)\cdot\mathrm{TP}+\beta^2\cdot FN+\mathrm{FP}}\right] = \left[-\frac{(1+\beta^2)\cdot\mathrm{TP}}{\left((1+\beta^2)\cdot\mathrm{TP}+\beta^2\cdot FN+\mathrm{FP}\right)^2}\right]. \tag{J.18}$$

The difference in importance between FN and FP is expressed by the ratio

$$\frac{\partial[F_\beta]}{\partial FN} : \frac{\partial[F_\beta]}{\partial FP} \Rightarrow \beta^2 : 1\,. \tag{J.19}$$

Hence, when $\beta = 1$, the ratio is $1 : 1$ and FN and FP are equally important. If $\beta = 2$, the ratio becomes $4 : 1$ and FN is four times more important than FP. The other way around, when $\beta = \frac{1}{2}$, the ratio is $\frac{1}{4} : 1$ and FP is four times more important than FN. Generally, FN is β^2 times as important as FP.

Appendix K HCOT Classification Example

In Figure K.1, we provide an example of the HCOT testing procedure from Algorithm 2. For simplicity, we illustrate this procedure using a constant threshold of 0.9 for each node and each day. Note that by using CAT, one can construct node- and day-specific thresholds. Below, we go through the complete testing procedure and explain the steps taken in the algorithm together with the reasoning behind these steps.

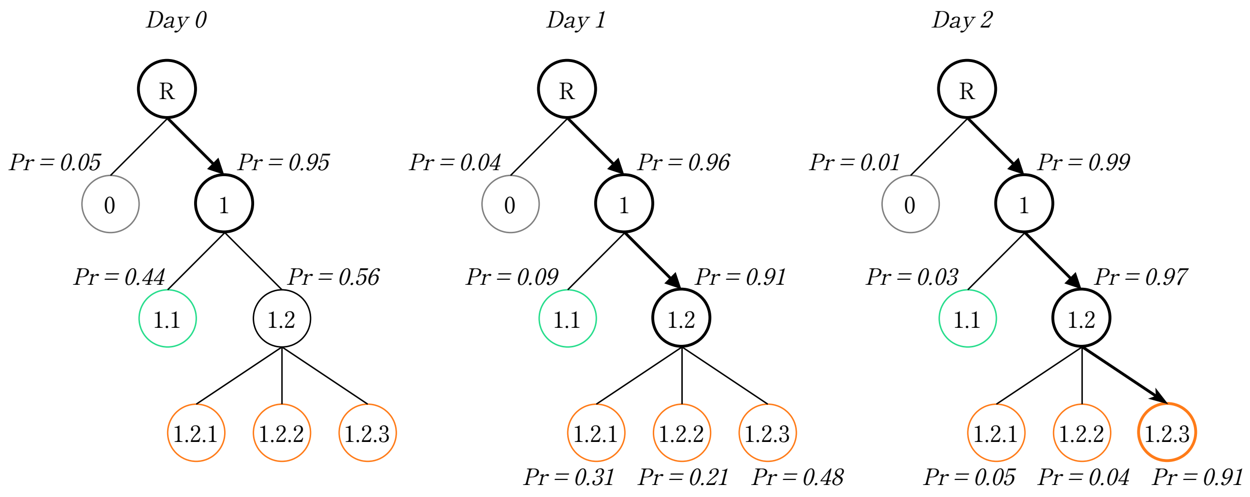


Figure K.1. *Example of the HCOT testing procedure (Algorithm 2) with a constant threshold of 0.9.*

- **Day 0**: An order X is placed and obtains a probability of 0.05 for the Unknown class and 0.95 for the Known class. As the majority vote is bigger than the threshold of 0.9, the order moves on to (parent) node <1>, which corresponds to the Known class. Next, we try to assign the order either to Happy or Unhappy, for which we obtain probabilities 0.44 and 0.56, respectively. Now, the threshold probability is not exceeded and therefore the order is blocked at node <1>. This means that on day 0, we only have sufficient information to classify the order as Known with enough certainty. In order to get a more detailed (leaf) label, we try again the next day. Nevertheless, the fact that X reached node Known already provides us with some insight on order X.

- **Day 1**: On day 0, order X did not reach a leaf node. Therefore, we try again on day 1, when (probably) more information about the order is available. Even though X reached node <1> (Known) with enough certainty on day 0, we start again from the top of the hierarchy on day 1. We do so to prevent error-propagation. Because we (most likely) had less information on day 0 as compared to day 1, it is possible that the classification to Known on day 0 was inaccurate. Hence, on day 1, we start again from the top of the hierarchy to include the newly available information. This is also the reason why we see different probabilities than those on day 0 for the first split in the hierarchy (0.04 and 0.96 compared to 0.05 and 0.95). Again, we reach node <1> with enough certainty. Now, we find probabilities 0.09 and 0.91 for the Happy and Unhappy class, respectively. This shows that the new information on day 1 (e.g., a return or cancellation) makes sure we can classify the order as Unhappy.

However, as the probabilities belonging to the different gradations of the Unhappy class do not exceed the corresponding thresholds, order X gets blocked at node <1.2> and moves on to day 2.

- **Day 2**: On day 2, we follow the exact same procedure as on day 1. Incorporating the newly available information, the probabilities for each class can slightly differ. This time we see that we can assign X with enough certainty to node <1.2.3> (Heavily Unhappy). Because this is a leaf node, the classification for X gets fixed and the testing procedure is finished.

For the previous example, we saw that X got classified at a leaf node after 2 days. In our HCOT algorithm, we make use of a total period of 10 days. During the first 9 days, we try to assign an order to a leaf node with enough certainty by taking into account the pre-defined thresholds. On the last day, we want each order to be assigned a leaf label and do not make use of thresholds to simply classify the remaining orders by following the path of majority votes.

Appendix L Flat Hierarchy

Figure L.1 presents the hierarchy that is used for flat classification. This flat hierarchy consists of a single parent node (the root node) and leaf nodes that are all on the same level.

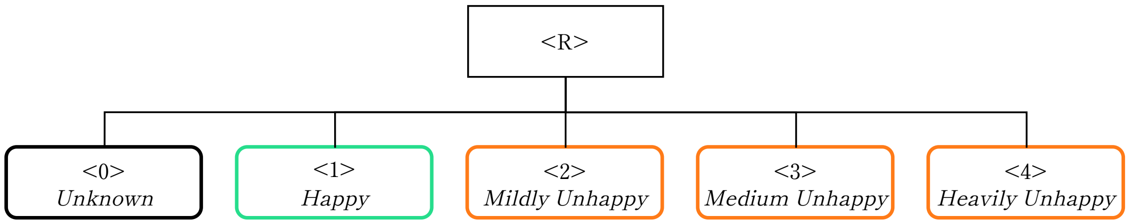


Figure L.1. *Tree-based structure of the flat hierarchy.*

Appendix M Results of CAT-HCOT and Baselines

In Table M.1, an extensive overview of the results from CAT-HCOT is given, as presented in Section 5.3. Table M.3 contains the results from the two baselines methods, static HCOT and flat CAT-HCOT, and Table M.2 contains the global measures of flat CAT-HCOT, as presented in Section 5.4.

Table M.1
Evaluation results of the CAT-HCOT algorithm ($\alpha = 0.7$) as discussed in Section 5.

	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Cum. classified (leaf)	39.0%	54.9%	64.9%	70.9%	75.3%	77.6%	78.7%	79.3%	79.8%	80.2%	100.0%
Cum. classified (int. & leaf)	64.3%	71.7%	80.9%	85.6%	89.6%	91.5%	92.4%	92.8%	93.0%	93.3%	100.0%
Acc_t	90.7%	94.1%	95.3%	95.3%	95.7%	96.1%	95.6%	95.7%	96.0%	97.1%	96.7%
hP_t	94.0%	96.5%	97.2%	97.3%	97.4%	97.8%	97.8%	98.1%	98.2%	98.6%	98.1%
hR_t	88.0%	92.7%	94.5%	94.6%	94.6%	96.0%	96.8%	97.6%	97.5%	98.0%	96.0%
$hF_{1,t}$	90.9%	94.6%	95.8%	95.9	96.0%	96.9%	97.3%	97.9%	97.8%	98.3%	97.0%
Blocking factor <R>	35.7%	46.4%	42.3%	41.0%	35.8%	34.4%	34.0%	33.9%	33.6%	33.3%	
Blocking factor <1>	50.0%	57.3%	65.6%	75.9%	82.0%	87.7%	90.5%	92.6%	92.0%	93.6%	
Blocking factor <1.2>	70.2%	46.4%	42.0%	39.0%	48.8%	51.9%	61.0%	66.5%	78.4%	81.2%	
$hP_{t,Unknown}$	91.7%	92.1%	93.3%	94.5%	95.8%	97.0%	97.4%	98.3%	98.4%	99.0%	96.6%
$hR_{t,Unknown}$	78.7%	79.7%	82.3%	85.1%	88.4%	91.6%	92.5%	95.0%	95.3%	97.1%	90.5%
$hP_{t,Happy}$	94.7%	97.7%	98.5%	98.6%	98.9%	98.9%	98.9%	99.1%	99.0%	99.2%	98.8%
$hR_{t,Happy}$	90.9%	95.4%	97.1%	97.1%	97.7%	97.7%	97.7%	98.1%	98.0%	98.4%	97.7%
$hP_{t,MildlyUnhappy}$	67.0%	94.3%	96.1%	98.0%	98.5%	98.5%	98.7%	98.5%	98.3%	98.6%	97.0%
$hR_{t,MildlyUnhappy}$	79.1%	96.2%	97.3%	98.3%	98.5%	98.5%	98.7%	98.5%	98.3%	98.6%	97.1%
$hP_{t,MediumUnhappy}$	94.2%	93.5%	92.0%	92.3%	91.6%	93.4%	93.1%	93.6%	93.6%	94.0%	91.8%
$hR_{t,MediumUnhappy}$	94.2%	93.5%	92.4%	93.1%	91.9%	93.5%	93.1%	93.6%	93.6%	94.0%	91.9%
$hP_{t,HeavilyUnhappy}$	99.8%	99.9%	98.9%	95.6%	93.8%	95.3%	97.0%	98.1%	98.6%	100.0%	75.9%
$hR_{t,HeavilyUnhappy}$	99.8%	99.9%	98.9%	96.4%	94.7%	95.9%	97.8%	98.1%	98.6%	100.0%	75.9%
$hP_{t,Known}$	98.0%	98.2%	98.6%	99.1%	99.7%	99.9%	100.0%	100.0%	100.0%	100.0%	
$hR_{t,Known}$	45.9%	46.1%	47.1%	47.9%	48.5%	48.8%	48.9%	48.8%	49.1%	49.0%	
$hP_{t,Unhappy}$	77.0%	86.0%	91.8%	93.7%	88.3%	98.4%	99.7%	100.0%	100.0%	100.0%	
$hR_{t,Unhappy}$	60.6%	63.2%	64.7%	65.2%	63.8%	66.3%	66.6%	66.7%	66.7%	66.7%	

Note: The blocking factors and internal performance measures are not available on day 10 since, on this day, we do not make use of a blocking approach and therefore no instances end up in internal nodes.

Table M.2
Global and class-specific hierarchical performance of the flat CAT-HCOT baseline ($\alpha = 0.7$).

	hP	hR	hF_1
Global	93.7%	90.9%	92.2%
Happy	96.3%	93.4%	94.8%
Unknown	93.9%	83.9%	88.6%
Mildly Unhappy	89.2%	93.7%	91.4%
Medium Unhappy	63.8%	77.7%	70.1%
Heavily Unhappy	55.5%	77.9%	64.8%

Table M.3
Daily evaluation results of the two baseline methods, static HCOT and flat CAT-HCOT ($\alpha = 0.7$), as discussed in Section 5.4.

		Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Static HCOT	Acc_t	81.7%	83.6%	87.8%	91.0%	93.6%	95.5%	96.5%	97.2%	97.7%	98.0%	98.2%
	% classified (leaf)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Flat CAT-HCOT	Acc_t	88.7%	92.1%	93.4%	94.5%	94.4%	93.7%	94.7%	96.2%	97.2%	97.0%	96.5%
	Cum. classified (leaf)	47.7%	60.6%	72.1%	77.0%	80.5%	83.3%	84.2%	85.0%	85.9%	86.3%	100.0%
	$hP_{t,Unknown}$	92.3%	92.2%	92.2%	92.5%	92.7%	92.8%	92.8%	92.9%	93.0%	93.1%	93.9%
	$hR_{t,Unknown}$	80.4%	80.1%	80.2%	80.7%	81.3%	81.4%	81.5%	81.7%	82.0%	82.1%	83.9%
	$hP_{t,Happy}$	94.6%	95.2%	95.7%	95.8%	95.8%	95.9%	95.9%	96.0%	96.0%	96.0%	96.3%
	$hR_{t,Happy}$	90.9%	91.7%	92.5%	92.7%	92.7%	92.8%	92.9%	92.9%	92.9%	92.9%	93.4%
	$hP_{t,MildlyUnhappy}$	54.0%	68.5%	79.9%	85.2%	87.4%	87.8%	88.4%	88.7%	88.9%	89.0%	89.2%
	$hR_{t,MildlyUnhappy}$	69.6%	80.6%	88.3%	91.4%	92.6%	92.9%	93.3%	93.4%	93.5%	93.6%	93.7%
	$hP_{t,MediumUnhappy}$	18.7%	33.2%	39.6%	43.9%	47.0%	57.6%	60.3%	61.9%	62.7%	63.3%	63.8%
	$hR_{t,MediumUnhappy}$	37.9%	53.6%	59.1%	62.6%	65.1%	73.4%	75.3%	76.4%	76.9%	77.3%	77.7%
	$hP_{t,HeavilyUnhappy}$	30.4%	40.4%	44.3%	46.3%	48.2%	50.7%	51.9%	53.1%	54.2%	54.6%	55.5%
	$hR_{t,HeavilyUnhappy}$	56.2%	66.5%	70.1%	71.7%	73.2%	75.1%	76.0%	76.9%	77.6%	77.9%	77.9%

Appendix N Feature Importance

Tables N.1–N.3 show the three most important features for each parent node in our hierarchy with respect to classification on day 0, 1, 2, 3, 5, and 10 after the order date.

Table N.1
Feature importance for classification on day 0 and day 1 after the order date.

		Day 0	Day 1
Feature 1	Node <R>	caseKnownX	onTimeDeliveryKnownX
	Node <1>	transporterCodeHistoricHappyX	onTimeDeliveryKnownX
	Node <1.2>	transporterCodeHistoricUnhappyX	transporterCodeHistoricUnhappyX
Feature 2	Node <R>	cancellationKnownX	returnKnownX
	Node <1>	transporterCodeHistoricUnhappyX	transporterCodeHistoricHappyX
	Node <1.2>	transporterCodeHistoricHappyX	transporterCodeHistoricHappyX
Feature 3	Node <R>	returnKnownX	caseKnownX
	Node <1>	transporterCodeHistoricUnknownX	transporterCodeHistoricUnknownX
	Node <1.2>	transporterCodeHistoricUnknownX	transporterCodeHistoricUnknownX

Table N.2
Feature importance for classification on day 2 and day 3 after the order date.

		Day 2	Day 3
Feature 1	Node <R>	onTimeDeliveryKnownX	onTimeDeliveryKnownX
	Node <1>	onTimeDeliveryKnownX	onTimeDeliveryKnownX
	Node <1.2>	transporterCodeHistoricUnhappyX	transporterCodeHistoricUnhappyX
Feature 2	Node <R>	returnKnownX	returnKnownX
	Node <1>	returnKnownX	returnKnownX
	Node <1.2>	caseKnownX	caseKnownX
Feature 3	Node <R>	caseKnownX	caseKnownX
	Node <1>	transporterCodeHistoricHappyX	lateDeliveryKnownX
	Node <1.2>	transporterCodeHistoricHappyX	transporterCodeHistoricHappyX

Table N.3
Feature importance for classification on day 5 and day 10 after the order date.

		Day 5	Day 10
Feature 1	Node <R>	returnKnownX	onTimeDeliveryKnownX
	Node <1>	onTimeDeliveryKnownX	onTimeDeliveryKnownX
	Node <1.2>	caseKnownX	caseKnownX
Feature 2	Node <R>	returnKnownX	returnKnownX
	Node <1>	returnKnownX	returnKnownX
	Node <1.2>	transporterCodeHistoricUnhappyX	returnKnownX
Feature 3	Node <R>	lateDeliveryKnownX	caseKnownX
	Node <1>	lateDeliveryKnownX	lateDeliveryKnownX
	Node <1.2>	returnKnownX	cancellationKnownX

Appendix O Business Implications

In this section, we discuss the implications of our research to business practice. In general, online retailers can use our classification algorithm to classify incoming product orders in an accurate and timely manner. An interesting feature of our algorithm is that it can classify product orders over time, and only finalises the classification if the prediction is sufficiently accurate and well-timed.

In our problem, we classify product orders with respect to a three-level hierarchy with seven nodes (excluding root node). We use a time period of ten days after (and including) the order date in which product orders need to be classified. A certainty parameter controls the trade-off between accuracy and timeliness: the higher this certainty parameter, the more accurate predictions will be, and, consequently, the less 'timely' they will be. However, regardless of the value of this certainty parameter, we ensure that all product orders are classified within ten days after the order date.

In principle, our algorithm can handle any hierarchical classification problem of product orders, since the tree-based hierarchy can be tailored to any problem setting. Although we have used a rather shallow tree, our algorithm also suits problems with larger trees containing more classification labels. In addition, the time window that is used to classify product orders can be adapted to the problem setting as well. We have used a maximum period of ten days after the order date in which product orders need to be classified, but this period can also be made shorter or longer, if necessary. Finally, in problem settings where accurate predictions are vital, the certainty parameter can be set to a larger value at the expense of timeliness. However, if the online retailer needs to evaluate the quality of product orders very quickly after their placement, it may be desirable to lower the certainty parameter at the expense of some accuracy in the predictions.

In our problem setting, we have found that the algorithm is able to classify around 40% of product orders on the order date itself, and more than 80% of product orders within five days after the order date. Here, we have set the certainty parameter to 0.7, which guarantees accurate predictions, but also allows us to provide relatively timely predictions (most of them within five days after ordering). This is very useful for online retailers, because it enables them to accurately detect the quality of product orders shortly after orders have been placed. If there are any problems, e.g., a partner consistently delivers products of bad quality, the online retailer is able to quickly flag this partner and take any measures if necessary. Also, during the 10-day prediction period, HCOT is able to assign test instances to internal nodes as well. Therefore, a retailer can 'learn' over time about the match quality of a particular order since the match quality becomes more and more specific once new information is available. For example, at some day HCOT might predict an order only as Known, while the next day it reaches the Unhappy node with enough certainty, and a few days later it is assigned to Mildly Unhappy. This is clearly an advantage of HCOT, which makes sure that retailers do not have to wait a fixed period of time to get reliable, detailed, predictions but rather gain more and more information about the match quality over time once there is enough certainty.