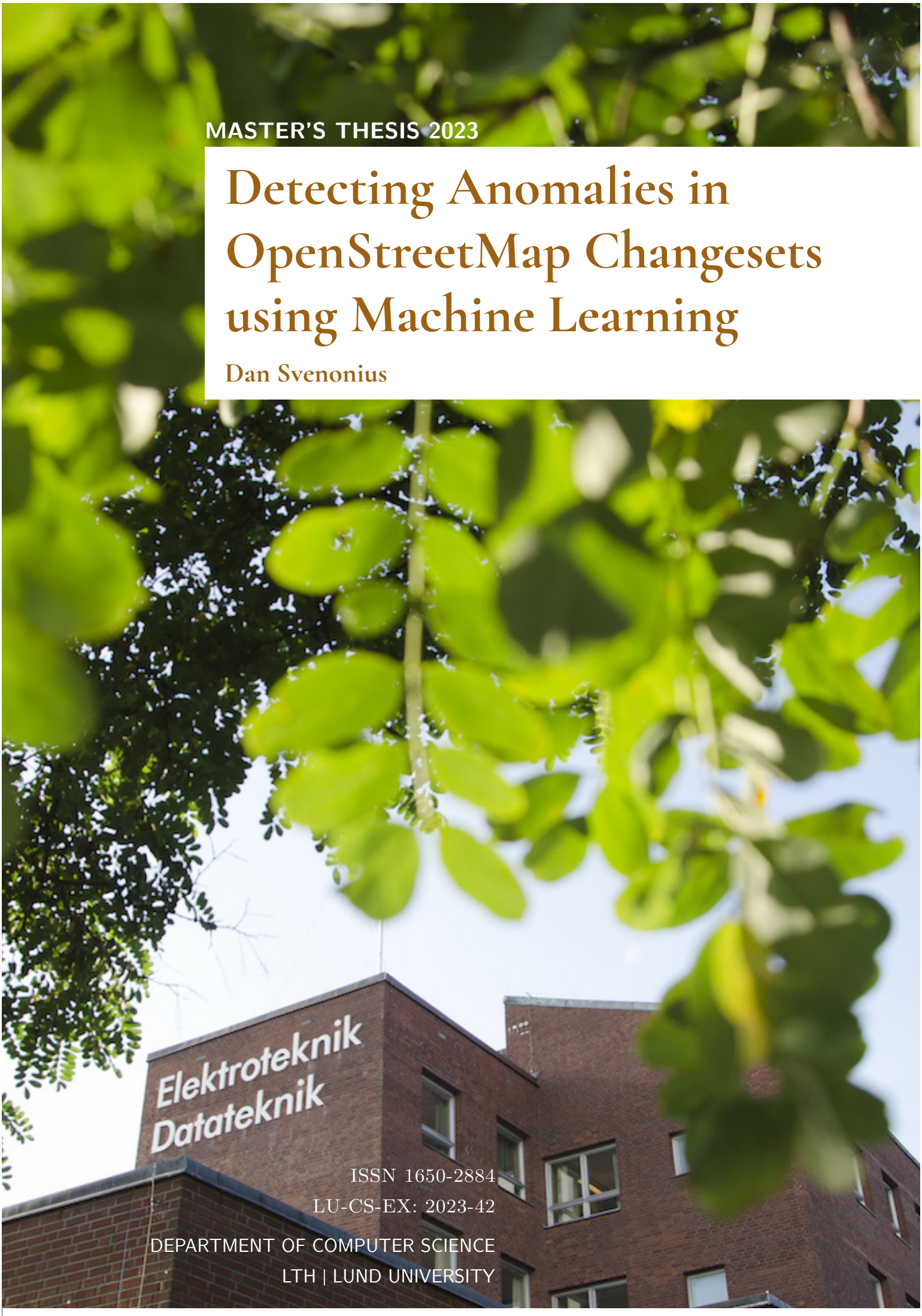


MASTER'S THESIS 2023

Detecting Anomalies in OpenStreetMap Changesets using Machine Learning

Dan Svenonius

ISSN 1650-2884
LU-CS-EX: 2023-42

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-42

Detecting Anomalies in OpenStreetMap Changesets using Machine Learning

Anomalidetektion i OpenStreetMap med Maskininlärning

Dan Svenonius

Detecting Anomalies in OpenStreetMap Changesets using Machine Learning

Dan Svenonius
da1881sv-s@student.lu.se

November 14, 2023

Master's thesis work carried out at AFRY AB.

Supervisors: Hampus Londögård, Hampus.Londogard@afry.com
Patrik Edén, patrik.eden@cec.lu.se

Examiner: Jacek Malec, Jacek.Malec@cs.lth.se

Abstract

OpenStreetMap is an open data set in which anyone can make contributions. This makes it prone to user errors in edits, and validation of these edits is required to make the data more trustworthy. As there are vast amounts of possible errors, it is difficult to create a number of heuristics for validation, making a machine learning approach interesting which could act as an initial filter, flagging potentially erroneous edits for further inspection.

This thesis investigates the potential of machine learning in validating OpenStreetMap changesets. It builds upon existing work in vandalism detection but attempts to generalize the problem from vandalism to unintentional errors. It compares neural networks to tree-based models and finds that vandalism detection methods with machine learning is effective in this task, where tree models perform significantly better than neural networks. In particular, gradient boosted trees performs best with regards to every metric, achieving 89.46% accuracy. Finally, an extensive feature importance analysis is performed.

Keywords: Machine learning, Anomaly detection, Validation, Explainable AI, Shapley values

Acknowledgements

I want to express my deepest gratitude to Patrik Edén at Lund University and Hampus Londögård at AFRY, without which this thesis would not have been possible. Your knowledge and feedback has truly been both inspiring and invaluable.

I also want to express my gratitude to Björn Pedersen for sharing vital ideas during the formulation the project and discussions throughout process.

Thank you Per Svensson, Pierre Laveklint and Rik Nauta for your guidance, and Jacek Malec for your help.

Finally, thank you Leo Westerberg, Simon Erlandsson and Tim Jangenfeldt for your great company throughout this thesis, and Clara Eklund for your support.

Contents

Chapter 1
Introduction

1.1 Background

In large and complex datasets, it can be advantageous to let anyone make contributions, also known as letting the dataset be *open*. This extracts individuals' knowledge in their own respective fields, as well as solves the problem of a central planner gathering all information, which quickly becomes unfeasible. Open datasets have potential for much better accuracy and detail, but also run a high risk for errors in contributions. Manually fact-checking individual contributions is very time consuming considering both the size and complexity of the dataset. This makes a machine learning approach interesting, which could act as a filter to narrow down the number of contributions that should be validated.

Two very large open data sets prone to incorrect contributions, intentional or not, are Wikipedia and OpenStreetMap. This project will work with geographical data, and hence use the OpenStreetMap data set. Previous work by Tempelmeier and Demidova (2022) has shown machine learning to be reasonably effective in flagging potential vandalism in contributions to OpenStreetMap, where vandalism is classified as an intentional, often dramatic, incorrect edit. This project will build upon their findings and attempt to generalize the problem. The aim is to investigate the effectiveness of machine learning methods in detecting unintentional errors. In addition, different types of machine learning models are compared and feature importance is analyzed.

1.2 Purpose

Having some kind of quality control in all data sets is important, and arguably the most important in open data sets. While many OpenStreetMap users correct other editor's mistakes, it is reasonable to assume that not all errors are detected and thus stay in the map. Additionally, a time gap between an error and its fix is inevitable. This time gap renders the affected

segment of the map unreliable, although a given map user might not identify the error and consequently use incorrect information. Furthermore, if another editor is unable to identify the error, it might propagate through future edits, making a clean revert non-trivial. Introducing some kind of automatic approach to detecting potential errors can significantly lower the downtime while possibly also flagging certain suspicious edits to be validated before rollout, combating these issues.

The purpose of this project is to investigate a machine learning approach to detecting a more general type of error than vandalism, however using existing methods based on vandalism detection by Tempelmeier and Demidova (2022). Vandalism detection is most likely easier than general error detection, as vandalism should be more dramatic in nature and thus easier to detect. Additionally, the project aims to investigate which types of models are well suited for this task and the importance of different features for the models.

1.3 Problem statement

This work was carried out together with AFRY and a client providing a map service. The project will attempt to answer the following questions:

1. How do existing vandalism detection methods in OpenStreetMap translate to general anomaly detection?
2. How do different types of machine learning models compare in anomaly detection in OpenStreetMap?
3. Which features are most important for the models in anomaly detection, and why?

Chapter 2
Theory

2.1 Classification in Machine Learning

In machine learning, a large set supervised (see 2.1.1) machine learning tasks fall into one of two categories - regression or classification. Regression aims to model a function with continuous outputs, for example how the price of a house will depend on a number of inputs like location and number of rooms. Classification on the other hand models a function with discrete outputs, like whether an image of an animal is a mouse, cat or a dog. Of course, the modeled function is never perfect as noise is always present in the data in addition to the sample size being limited, and as such the output of the models are referred to as *predictions*.

A classification problem could in theory contain any number of different classes, but in general two cases are distinguished: binary and multi-class classification. Binary classification refers to a problem with only two classes resulting in an ”either/or” answer, like given a set of images of cats or dogs, predicting which image contains cats and which contain dogs. Multi-class on the other hand refers to all classification problems with three or more classes, and does not answer an ”either/or” question, or at least not in a straight forward manner.

As this project aims to classify edits (or *changesets*, more about this below) in OpenStreetMap, the remainder of this section will handle the case of classification tasks in machine learning and thus omit related theory regarding regression.

2.1.1 Supervised vs. Unsupervised Learning

There are many machine learning algorithms, some of which require information about the ”correct” answer in order to learn how to accurately predict data. Data that has a correct answer attached to it is called *labeled* data. Algorithms that require labels are called *supervised* and include conventional artificial neural networks and decisions trees, among others. Conversely, algorithms that do not require labels are called *unsupervised* and include clustering algorithms such as k-means (Ahmed, Seraj, and Islam 2020). There are also other machine

learning methods that do not fit into a supervised *vs.* unsupervised paradigm, such as reinforcement learning (Qiang and Zhongli 2011).

2.1.2 Parameters vs. Hyperparameters

A *hyperparameter* in machine learning is a parameter that is defining how a model is trained, which is not to be confused with a model's *parameters* (sometimes referred to as *weights*), which are the model's learned rules or weights that in turn define the final model. The hyperparameters are only relevant when training the model, whereas the parameters define the model when training is finished. An example of a hyperparameter is how many times a data set is fed to the model during training, whereas a parameter is what the model learns when seeing this data set, for example a weight between two nodes in an artificial neural network.

2.1.3 Loss Functions

In supervised machine learning, in order to optimize a model's parameters, a loss function is minimized. The loss function quantifies how accurate the model's predictions are with respect to its actual value. The loss functions used in this project are presented in further detail below.

2.1.4 Training, Validation and Test Data

In supervised learning the full data set is split into subsets to use for different purposes. In general, either a train/test or a train/test/validation split is used. Their purposes are:

Training data: The data used to optimize the model's *parameters*. This is the data that is fed to the model during training and as such the data that determines the model's parameters through minimization of the loss function.

Validation: The data used to optimize the model's *hyperparameters*. A model will in general produce different results with the same training data when different hyperparameters are used. The validation data is then used to determine which model performs best, which in turn yields the best set of hyperparameters.

Test data: The data used to determine the performance of the model. This data set is not at all used when training the model, and the model is thus entirely independent of this data (assuming independent samples), which will indicate how well the model generalizes on new data.

2.1.5 Cross-validation

Cross-validation is a way to better gauge the general performance of a model based on training and validation data. When performing K-fold cross-validation, one partitions the data set into K equal parts, where $K - 1$ parts are used to train the model and the remaining part is used to test the model's performance. This is then done K times while changing the validation data. The figure below illustrates an example of 3-fold cross-validation:

3-fold cross validation.

2.2 Artificial Neural Networks

2.2.1 The Artificial Neuron

Artificial neural networks, or neural networks/ANNs in short, are a family of machine learning models. Their most basic component is a neuron (also referred to as a node), inspired by the biological neuron in brains. A neuron in an artificial neural network is a transformation $y : \mathbb{R}^p \to \mathbb{R}$:

$$y(x_1, ..., x_p) = \varphi(b + \sum_{i=1}^{p} w_i x_i) \tag{2.1}$$

where x_i are the input signals, w_i are the weights, b is the bias and φ is the activation function. The activation function generally has a non-decreasing monotone behaviour. Figure 2.1 taken from Ohlsson and Edén (2021a) below illustrates this:

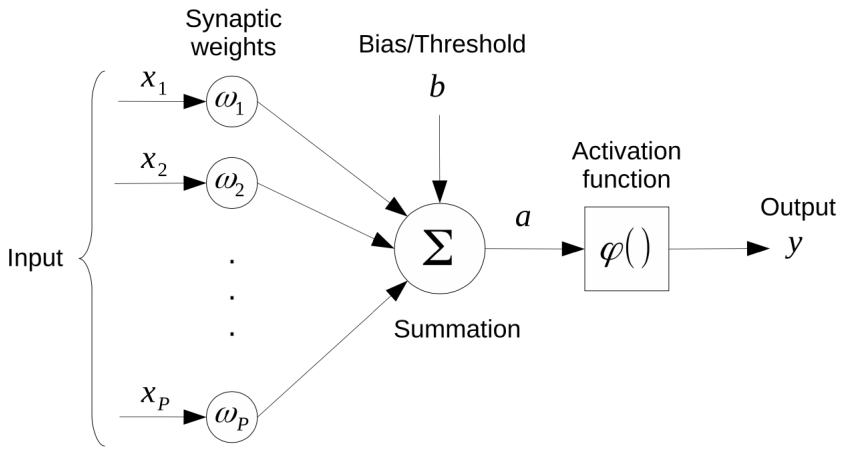


Figure 2.1: Illustration of an artificial neuron. The figure is taken from Ohlsson and Edén (2021a).

2.2.2 The Artificial Neural Network

An artificial neural network is composed of many neurons setup in a sequence of layers. Typically the layers are dense, meaning that every neuron from a given layer has a weight connected to every neuron in the next layer. Each neuron in a layer will consequently input its output into all neurons in the next layer, multiplied by different weights for each receiving neuron. The output dimension is equal to the number of neurons in the last layer. Figure 2.2 taken from Ohlsson and Edén (2021a) illustrates this:

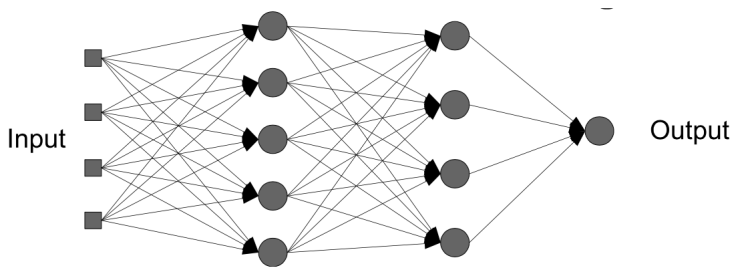


Figure 2.2: Illustration of how artificial neurons stacked in layers connect in an artificial neural network. The figure is taken from Ohlsson and Edén (2021a).

For a classification problem, the output of a given neural network is a discrete probability distribution over all the classes in the data set. This is achieved by applying a final activation function to output of the last layer, creating a probability in the binary case or a probability distribution in the multi-class case. In the binary case, this is the logistic function: (Ohlsson and Edén 2021b)

$$\varphi(a) = \frac{1}{1 + e^{-a}} \tag{2.2}$$

and softmax in the multi-class case, where φ_i corresponds to the probability that the sample belongs to class i: (Ohlsson and Edén 2021b)

$$\varphi_i(a_1, \ldots, a_p) = \frac{e^{a_i}}{\sum_{j=1}^{p} e^{a_j}} \tag{2.3}$$

2.2.3 Loss Function

The loss function E used in this project for neural network classification is the cross-entropy function. In the binary case it is given by:

$$E(\boldsymbol{\omega}) = -\frac{1}{N} \sum_{n=1}^{N} (d_n \ln(y_n) + (1 - d_n) \ln(1 - y_n))$$

where $\boldsymbol{\omega}$ is the vector of all weights $\omega_1, ..., \omega_k$, N is the size of the training data set, $d_n \in \{0, 1\}$ is the target (where $0, 1$ are representations of more meaningful class labels) and $y_n \in [0, 1]$ is the probability output of the model for sample n in the training data. In the multi-class case, the *categorical cross-entropy* loss function is given by: (Ohlsson and Edén 2021b)

$$E(\boldsymbol{\omega}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{c} d_{ni}\ln(y_{ni})$$

where the notation is the same as in the binary case, but for each sample in the training set, the loss function needs to account for the prediction for every individual class, hence summing over the classes c. However, no class should be given a larger weight than another, so $d_{ni} \in \{1, ..., c\}$ is replaced by a one-hot encoding, where a sample n belonging to class j_n:

$$d_{ni} = \begin{cases} 1, & \text{if } i = j_n, \\ 0, & \text{if } i \neq j_n, \end{cases}$$

$y_{ni} \in [0, 1]$ is the predicted probability that sample n belongs to class i (Ohlsson and Edén 2021b). Of course, the dependence on the weights $\boldsymbol{\omega}$ lies inside of the model's predictions y_n or y_{ni} as the predictions are determined by the weights.

2.2.4 Back-propagation

Back-propagation is an algorithm for finding all partial derivatives $\frac{\partial E}{\partial \omega_i}$ for a loss function E. Essentially, back-propagation starts from the output layer of the network (hence the name) and utilizes the chain rule to iteratively find $\frac{\partial E}{\partial \omega_i}$ by going backward through the network's layers.

2.2.5 Gradient Descent

Gradient descent is the method used for finding a minimum of the loss function. Put simply, given a gradient (calculated through the back-propagation method), taking a step in the opposite direction of the gradient will result in a new point further "down" the loss function. Iterating this procedure will result in a finding a minimum. Note that a minimum does not imply a global minimum, meaning that simply taking steps in the negative gradient direction can result in convergence yet poor performance. Several methods have been developed to solve issues with gradient descent such as stochastic gradient descent and Adam (LeCun et al. 2012; Yi, Ahn, and Ji 2020).

2.2.6 Attention

Attention is a way for a model to learn the most important elements out of a given set of elements. It consists of a query Q, keys K and values V. The attention function is given by

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the number of columns in the matrix K. Intuitively, from the similarity between the query Q and the keys K a probability distribution is created representing attention weights, and multiplying this with V results in a weighted sum of V. An extension of this mechanism is multi-head attention, where this mechanism is used multiple times and each

"head" focuses on different features. This way, the attention mechanism can output multiple objects that are important in different ways (Vaswani et al. 2017).

2.2.7 Hyperparameters

Only the hyperparameters tuned or otherwise relevant for the project will be stated.

Hidden layer sizes: The number of layers and the size of each layer in the artificial neural network, except for the input and output layers, which have to be consistent with the input and output dimensions.
Epochs: The number of times each sample in the training data is fed to the network during training. Occasionally, this is given by a maximum number of epochs, after which the network will stop training if it hasn't already converged according to a tolerance parameter.
L2 regularization: L2 adds the sum of the squared weights to the loss function so that $E \leftarrow E + \alpha \sum_i \omega_i^2$. The intuition is that large weights lead to overfitting, and penalizing this reduces overfitting. In hyperparameter tuning, the factor α is tweaked.
Batch size: The number of data points used for every iteration of updating the weights via gradient descent. This only applies if some version of stochastic gradient descent is used where a subset of the training data is used when calculating the gradient.
Learning rate: The length of the step taken in the negative gradient's direction during gradient descent.

Additionally, the model presented by Tempelmeier and Demidova (2022) uses a number of hyperparameters which are not tweaked in any of the models that are created in this project, as Ovid is used without any hyperparameter modifications. They are however given below:

Dropout: The fraction of nodes in layer that are randomly set to zero.
Number of attention heads: Number of heads used in multi-head attention, equaling the number of outputs from multi-head attention.
Patience: Number of epochs after which to stop training if validation loss has not improved.

2.3 Decision Trees

A decision tree is a supervised model which partitions the data. At every node in the tree, the data is split based on a condition for an individual feature. The conditions are decided by a heuristic (which can be interpreted as a sort of loss function) quantifying how well a given condition partitions the data into subsets that are as homogeneous as possible. Training is then finished when every leaf consists of only a single class and this class decides the prediction of a sample reaching this leaf (Almog 2022). Hyperparameters can affect the purity of a leaf, in which case the output probability of a sample belonging to class i is the fraction of samples of class i in that leaf. This corresponds to the predicted class being the majority class of the leaf (*sklearn.ensemble.RandomForestClassifier* 2023).

While several heuristics exist, this project only uses Gini impurity, where a Gini impurity G is defined on a set S with n different classes and is given by

$$G = 1 - \sum_{i=1}^{n} p_i^2 \tag{2.4}$$

where p_i is the relative frequency of class i in S. Intuitively, it quantifies the spread of classes within a set where a large G implies a diverse set containing significant numbers of more than one class. Conversely, a small G corresponds to a very homogeneous set (Almog 2022).

2.4 Random Forest

A random forest is an ensemble of independently trained decisions trees. The method of training a random forest is slightly different from a single decision tree, however. First, m subsets $S_1, ..., S_m$ are created through bagging, or bootstrap aggregating, where each subset draws n samples from the training data set with replacement. Here, m corresponds to the number of decision trees in the forest and n is the size of each subset, often equal to the size of the training data set. Secondly, a decision tree is trained for every S_i, however only a random subset of the features present in S_i are considered when finding the best split condition at a node. This causes individual trees in the forest to behave differently, reducing overfitting when considering the ensemble as a whole (Yiu 2019). Finally, the output of a random forest is a probability distribution over all classes computed by the mean probability output of every individual tree in the forest. The predicted class is then the class corresponding to the maximum of this probability distribution (*sklearn.ensemble.RandomForestClassifier* 2023).

2.4.1 Hyperparameters

Number of trees: The total number of decision trees in the forest.
Maximum depth: The maximum depth of each decision tree within the random forest.
Maximum features: The number of features to consider when deciding a best split in a node.

2.5 Gradient Boosted Trees

When using gradient boosting with decision trees, one uses an ensemble of decision trees to make predictions. However unlike random forest, trees are not trained independently of one another, but each tree is instead iteratively introduced to reduce the prediction error of the existing ensemble. To state the algorithm in an informal fashion:

Consider a series of samples $x_1, ..., x_n$ and targets $y_1, ..., y_n$ with k classes and an ensemble of m decision trees giving predicted probabilities $F_{mj}(x_i) = p_{ij}$ of sample i belonging to class j. Then consider the residuals for these predictions $r_{ij} = p_{ij} - y_i$ where $i \in 1, ..., n$ and $j \in 1, ..., k$. A new decision tree is trained to minimize the residuals r_{ij} and is then added to the ensemble to create a new model F_{m+1}. This process is then iterated until the desired number of trees are added to the model (Masui 2022).

In binary classification, the sigmoid function is applied to the sum of all trees' outputs, which in turn is decided by the value of the leaf that the sample reached in a tree. For multi-class, each class is trained separately and softmax is applied to the outputs of each individual model (Bùi 2023).

2.5.1 Loss Function

The loss function used in this project for gradient boosting with decision trees is the cross-entropy function, see section 2.2.3 for a detailed explanation.

2.5.2 Hyperparameters

Number of trees: The number of trees in the ensemble.
Depth: The depth of each tree.
L2 regularization: The inverse of the weight each leaf's output is multiplied by. A large L2 gives the leaf a smaller weight.

2.6 Evaluation Metrics

2.6.1 Confusion Matrix

A confusion matrix provides information about how the ground truth classes relate to the predicted classes in classification. The rows typically represent the ground truth and the columns represent the predictions, meaning that an element c_{ij} in a confusion matrix $\boldsymbol{C}$ is the number of samples that in reality belongs to class i but were classified as class j. Hence, confusion matrices with large counts along the central diagonal and small counts otherwise are indicative of a good model. An example of a confusion matrix is given in figure 2.3, using a simple classification model for `SKlearn`'s iris data set with three different classes:

2.6.2 Precision, Recall, F1 and Accuracy

Below, TP, FP, TN and FN refers to true positive, false positive, true negative and false negative, respectively.

For a classification problem, *precision* is defined as

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.5}$$

and gives the ratio between the number of samples correctly classified as class i against the total number of samples classified as i. It is a measure of how "precise" the model is in its classification of class i, as a high value yields few samples incorrectly classified as i. On the other hand, *recall* is defined as

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.6}$$

and quantifies the ratio of the number of samples correctly classified as i against the total number of samples of class i. It describes how well the model classifies all of the samples within a given class. Finally, F_1 (or *F1-score*) is defined as the harmonic mean between precision and recall and is given by

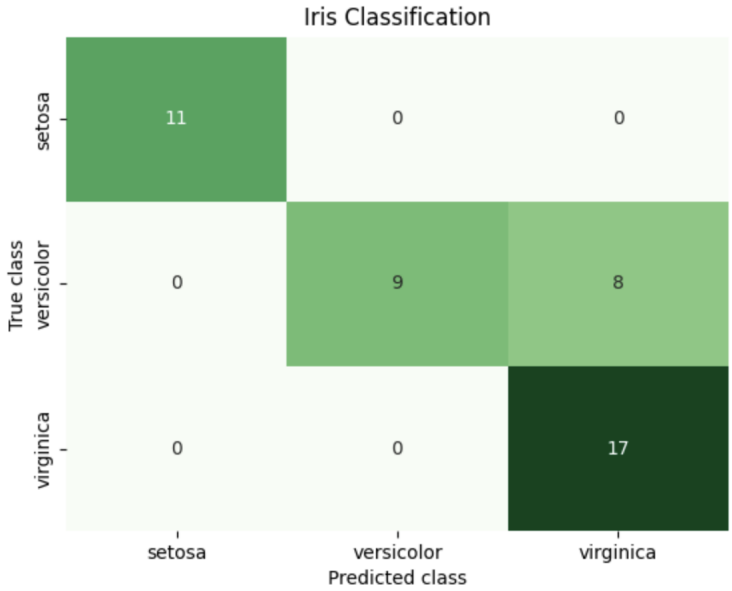


Figure 2.3: Example confusion matrix using a simple classification model for `SKlearn`'s iris data set with three classes - *setosa*, *versicolor* and *virginica*. The rows represent the ground truth class and the column the predicted class. The example model works well except for having a tendency to classify *versicolor* as *virginica*.

$$F_1 = 2\,\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2TP}{2TP + FP + FN} \tag{2.7}$$

A property of the harmonic mean is that it punishes extreme values, meaning that although one of precision or recall might be perfect (equal to one) while the other is zero, the F1-score in this situation would be equal to zero, maintaining the information that this would be a bad model. F1-score can be seen as a combination between precision and recall. Precision, recall and F1-score are given for every class in the classification problem.

Finally, *accuracy* is given by

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.8}$$

where all of TP, TN, FP, FN denotes the correctly/incorrectly classified samples for *all* classes. Intuitively, it is the fraction of correctly classified samples in the data set.

2.7 Shapley Values

Shapley values stem from game theory and provides one solution to a cooperative game, which is a game where players form coalitions to obtain utility, or "gain". Within a cooperative game, the Shapley values provide one answer as to how the utility should be shared between the players and has a number of desirable properties (additivity, symmetry, efficiency, null

player property and uniqueness). The formula for calculating the Shapley value for player i in a cooperative game is

$$\sigma_i = \sum_{S \in 2^{N \setminus \{i\}}} \frac{(|N| - |S| - 1)!|S|!}{|N|!} (v(S \cup \{i\}) - v(S)) \tag{2.9}$$

where $N = \{1, ..., n\}$, are the players, $S \subseteq N$ and $v(S)$ is the utility of coalition S. Intuitively, σ_i can be thought to represent the "power" of player i through a weighted sum of the *marginal* contributions of player i to each coalition, with a combinatorial factor to achieve the desirable properties. A large Shapley value means that the player gives large contributions to coalitions, meaning she should also receive a large utility (Lucchetti 2023).

Within machine learning, Shapley values are used to understand models by quantifying individual features' effects on a model's predictions. However in order to utilize the theory behind Shapley values, one first needs to rephrase the task of predicting with machine learning to cooperative game theory. Here, the input features are viewed as the players and the predictions are viewed as the utility (Rozemberczki et al. 2022). Equation (2.9) shows that calculating the Shapley values requires both $v(S)$ and $v(S \cup \{i\})$, which poses a problem as this in theory requires 2^n different models to be trained - one for each possible subset of N, which quickly becomes unfeasible. SHAP values as presented by Lundberg and Lee (2017) provides a solution to this.

2.7.1 SHAP Values

This project utilizes SHAP values, implemented in Python's `SHAP` library, which uses different algorithms to calculate the approximations of the Shapley values for different types of machine learning models. It differentiates between model-specific and model-agnostic methods, which either utilizes the specific architecture of certain models or makes no assumptions about the model's internal structure. Below, the model-agnostic `KernelExplainer` is described because of its generality, although the model-specific `TreeExplainer` is also used in the project.

To solve the problem of training a new model for all possible $S \subseteq N$, the idea is to use only one model trained on all features in combination with a conditional expectation to handle missing inputs. In `KernelExplainer`, these missing inputs are calculated by providing a background data set to sample from, replacing the removed input by the sampled. Then, the SHAP values for a sample x are calculated all at once using a connection between Shapley values and linear regression, explained below. The advantage of this method is that it requires fewer samples and model evaluations than merely approximating by sampling.

First, the concept of an *explanation model* g is introduced to explain a more complex model f, which in this case is the conditional expectation representation of the machine learning model. The model g uses simplified inputs x' and a mapping $x = h_x(x')$ which maps the simplified inputs to the original inputs x, where $g(x') = f(h_x(x'))$. Then, *additive feature attribution methods* are defined as having linear explanation models on the form

$$g(z') = \phi_0 + \sum_{i=1}^{N} \phi_i z'_i \tag{2.10}$$

where $z' \in \{0,1\}^N$ are binary inputs and each ϕ_i in the explanation model corresponds to the effect of feature i. $z'_i = 1$ means feature i is included in the coalition and vice versa for $z'_i = 0$. For additive feature attribution methods, there exists a unique explanation model which satisfy three properties (local accuracy, missingness and consistency), which is given by

$$\phi_i(f,x) = \sum_{z' \subseteq '} \frac{(|N| - |z'| - 1)!|z'|!}{|N|!} \left(f(h_x(z')) - f(h_x(z' \setminus i))\right) \tag{2.11}$$

where $|z'|$ are the number of non-zero elements in z'. Note that this is an equivalent formulation of the definition of Shapley values in equation (2.9). To find ϕ, the *LIME* method is used, which creates a local approximation around a model's prediction. This is an additive feature attribution method, making g follow equation (2.10). *LIME* establishes that ϕ is found through the following expression:

$$\arg\min_g L(f, g, \pi_{x'}) + \Omega(g) \tag{2.12}$$

and if

$$\begin{aligned} \Omega(g) &= 0, \\ \pi_{x'}(z') &= \frac{N-1}{\binom{N}{|z'|}|z'|(N-|z'|)}, \\ L(f,g,\pi_{x'}) &= \sum_{z' \in Z} (f(h_x(z')) - g(z'))^2 \pi_{x'}(z') \end{aligned}$$

the solution satisfies the three properties, causing ϕ to correspond to the Shapley values for f (note that f is the conditional expectation and not the true model, making these SHAP values for the model rather than the true Shapley values). L is weighted sum of squared errors, weights decided by π'_x and Ω is a complexity penalty. Because g is linear and L is a sum of squared errors, g can be calculated through weighted linear regression. (Lundberg and Lee 2017)

Chapter 3
OpenStreetMap Data

OpenStreetMap's geographical model is built from three different types of elements that form a hierarchical structure - nodes, ways and relations. The node is the most simple element, defined by a longitude and latitude and represent a point on the map. Nodes are used to either represent points of interest, for example a restaurant, or are used in sequence to build more complex elements. Secondly, a way is an ordered sequence of nodes which construct larger elements like roads or buildings. Finally, relations consists of nodes, ways and other relations which form the largest OSM elements, for example city borders. In addition, elements can hold *tags* further describing the element.

When a user makes a contribution to the database, a *changeset* is created containing these edits. Changesets are central to this project as machine learning classification will be performed on changesets - these are explained in detail below.

3.1 Tags

In OpenStreetMap, both elements and changesets can have tags to further explain the contents of these objects. Tags are key-value pairs of free text fields with a maximum of 255 characters, however there are conventions as to how common information is supposed to be represented in tags. The key describes a kind of category, while the value gives further information about the object regarding this specific category (*OpenStreetMap Wiki, Tags* 2022). The conventions on how to use a set of common tags are given in *OpenStreetMap Wiki, Map features* (2023).

3.2 Changesets

Every single contribution to OSM creates a changeset. Three different operations are possible when editing - create, modify and delete. Create introduces an entirely new element to OSM,

modify changes an existing element and delete removes an existing element.

A changeset contains all of the edits (meaning creates, modifies and deletes of nodes, ways and relations) made over a short period of time, limited either by a maximum time of 24 hours or 1 hour of inactivity. There is a convention regarding the geographical extent of a changeset in that it should be ”local”. What exactly this entices is however debated, as some argue that it should not span more than a city, whereas others argue that the maximum scope could be a continent. In general though, it is recommended to keep the changes to one country, and preferably as small as possible, and if making edits over several areas it is recommended to open and close several changesets so as to still make each individual changeset ”local” (*OpenStreetMap Wiki, Changeset* 2023).

A changeset consists of two XML files. One contains information about the changeset as a whole, referred to as the *metadata* file, and the second contains the edited elements, referred to as the *OsmChange* file. First, an example of the changeset metadata file is given below:

```
▼<osm version="0.6" generator="CGImap 0.8.8 (2341057 spike-06.openstreetmap.org)" copyright="OpenStreetMap and
 contributors" attribution="http://www.openstreetmap.org/copyright"
 license="http://opendatacommons.org/licenses/odbl/1-0/">
  ▼<changeset id="132463487" created_at="2023-02-12T21:20:41Z" closed_at="2023-02-12T21:20:41Z" open="false"
   user="Yves Labeeuw" uid="5818792" min_lat="50.8501315" min_lon="4.3615216" max_lat="50.8510046"
   max_lon="4.3624486" comments_count="0" changes_count="2">
     <tag k="comment" v="escalier relié au chemin"/>
     <tag k="created_by" v="iD 2.24.2"/>
     <tag k="host" v="https://www.openstreetmap.org/edit"/>
     <tag k="locale" v="fr"/>
     <tag k="imagery_used" v="AIV Flanders most recent aerial imagery"/>
     <tag k="changesets_count" v="37"/>
   </changeset>
 </osm>
```

Figure 3.1: An example of a changeset metadata file downloaded from OpenStreetMap. This contains information regarding the changeset as a whole, but not information about individual edits.

Note the ***comment***-tag which will become relevant later. Table 3.1 below describes the relevant attributes in the changeset metadata file.

Table 3.1: Descriptions of the changeset *metadata* attributes. Only the attributes relevant to this project are described.

id	The unique id of this specific changeset.
created_at	When the changeset was opened.
closed_at	When it was closed and uploaded to OSM.
uid	Unique id of changeset author.
min_lat	Latitude of the southernmost element subject to edits by the changeset.
min_lon	Longitude of the westernmost element subject to edits by the changeset.
max_lat	Latitude of of the northernmost element subject to edits in the changeset.
max_lon	Longitude of the easternmost element subject to edits in the changeset.
tag: comment	An optional but widely used tag describing the edits. The comment is written in natural language by the author.

Secondly, the *OsmChange* file contains information about the elements edited by the changeset. The OsmChange file corresponding to the example used in figure 3.1 is given below:

```
▼<osmChange version="0.6" generator="CGImap 0.8.8 (2341061 spike-06.openstreetmap.org)" copyright="OpenStreetMap and
 contributors" attribution="http://www.openstreetmap.org/copyright"
 license="http://opendatacommons.org/licenses/odbl/1-0/">
 ▼<modify>
    <node id="8268400439" visible="true" version="2" changeset="132463487" timestamp="2023-02-12T21:20:41Z"
    user="Yves Labeeuw" uid="5818792" lat="50.8507846" lon="4.3618443"/>
  </modify>
 ▼<modify>
  ▼<way id="889387301" visible="true" version="3" changeset="132463487" timestamp="2023-02-12T21:20:41Z" user="Yves
   Labeeuw" uid="5818792">
     <nd ref="8268400446"/>
     <nd ref="8268400447"/>
     <nd ref="8268400460"/>
     <nd ref="8268400448"/>
     <nd ref="8268400449"/>
     <nd ref="8268400450"/>
     <nd ref="8268400451"/>
     <nd ref="8268400439"/>
     <nd ref="8268400452"/>
     <nd ref="8268400453"/>
     <nd ref="8268400454"/>
     <nd ref="8268400455"/>
     <nd ref="8268400456"/>
     <nd ref="8268400457"/>
     <nd ref="8268400458"/>
     <nd ref="8268400459"/>
     <nd ref="8268400460"/>
     <tag k="highway" v="path"/>
     <tag k="layer" v="1"/>
    </way>
  </modify>
</osmChange>
```

Figure 3.2: OsmChange file corresponding to the changeset in figure 3.1. In this changeset, one node and one way was modified.

This file shows the state of the edited elements *after* the changeset was applied. In order to see the actual edits one would need to compare with the previous version of these elements. Figure 3.3 shows the *difference* between the XMLs for these two versions:

```
<node
    version="2"
    changeset="132463487"
    timestamp="2023-02-12T21:20:41Z"
    user="Yves Labeeuw"
    uid="5818792"
    lat="50.8507846"
    lon="4.3618443"
/>
```

```
<way
    version="3"
    changeset="132463487"
    timestamp="2023-02-12T21:20:41Z"
    user="Yves Labeeuw"
    uid="5818792">
  <nd ref="8268400439"/>
</way>
```

Figure 3.3: The difference between the element versions before and after the changeset was applied.

The difference is that the node was moved and also integrated into the edited way. The actual edit was to include the node representing a staircase to be a part of a park, represented by a way.

The relevant attributes for OSM elements are described in table 3.2 below:

Table 3.2: Descriptions of element attributes relevant to this project. See the <node>/<way>-tags in figure 3.2 for an example of how an element is represented in OSM.

id	The unique id of this specific element.
visible	Whether this element is visible on the OSM map. Deleted elements have $visible = "false"$ and deletes can be reverted by setting $visible = "true"$, meaning deleted elements still exist in the database.
version	The iteration of the element. $version = 1$ for a newly created element.
changeset	Id of the changeset creating this element version.
uid	Unique id of changeset author.

Chapter 4
Creating a Data set

The following section will present how the data set is created and how the changesets are preprocessed. A changeset history dump downloaded on February 27th 2023 constitutes the basis for the data set, and as such no changeset created at a later point in time is considered in this project. No changeset committed before 1st December 2011 is considered, which is motivated below. An element history dump is used in addition to the changeset history, also downloaded on February 27th 2023.

The finished data set contains 51825 data points, 24950 reverted changesets (ground truth positive) and 26875 non-reverted changesets (ground truth negative).

4.1 Finding Reverted Changesets

This project aims to use machine learning for detecting suspicious changesets, which will translate to the task of binary classification of changesets as being potentially erroneous or not. Thus, there will be no differentiation between changesets where all edits are potentially erroneous, as opposed to only a subset. The question posed will simply be "*does this changeset contain at least one potentially erroneous edit?*".

In order to do this with supervised learning, a number of potentially erroneous changesets have to be collected, where the most straight forward method is to find changesets known to contain errors. However, considering the wide range of possible edits and errors that can be made in OpenStreetMap, it is very difficult to find a set of rules that, if they apply, would indicate an erroneous edit. Instead, a method very similar to the one used by Tempelmeier and Demidova (2022) which utilizes the comment tag is adopted. The comment tag is widely used - a scan of the changeset history shows that 91.2% of all changesets between 1st December 2011 and 27th February 2023 use this tag - and these comments are scanned for four things:

1. The changeset comment contains the keyword "revert".

2. The changeset contains an ID, identified by a string of 8 or 9 characters in a row without blank spaces.

3. The ID is preceeded by any of the following three words: `"changeset"`, `"chgset"` or `"cs"`.

4. The keyword ”vandalism” is not included in the comment.

There were 27393 changesets satisfying all of the above conditions. These four conditions are used for the following reasons:

1. Searching for the ”revert” keyword suggests that a user detected some edits that were a detriment to the quality of the OSM data and reverted these elements back to their previous state, most likely indicating errors in the reverted edits. It need not be its own word, but the sequence of characters must exist in the comment such that ”reverting” and ”reverted” would pass as well.

2. The ID establishes a connection to the reverted changeset, which can then be retrieved. Restricting to IDs with 8 or 9 digits considers all changesets from December 1st 2011, equivalent to about 123 million changesets. This provides a compromise between size and quality in the data set, as conventions and behaviour in OSM editing may change over time and very old changesets may constitute poor data.

3. The ID *needs* to refer to a changeset, and not a node, way or relation. By demanding that the ID is preceeded by `"changeset"`, `"chgset"` or `"cs"` - three common words for ”changeset” in comments - will establish that the ID in fact refers to a changeset and not an OSM element.

4. The project aims to look at unintentional mistakes. Vandalism has an intentional connotation, and reverting changesets mentioning this word in the comment are thus discarded. Similarly to ”revert”, it need not be its own word but the sequence of characters cannot exist in the changeset comment.

Explained intuitively, the method searches the entire changeset history for *reverting* changesets where a connection to the *reverted* changeset is established through its ID in the reverting changeset’s comment. Then, the reverted changeset is retrieved and is labeled as containing errors, or equivalently in this case, as having been reverted. These two ways of referring to the same class - containing errors or having been reverted - will henceforth be used interchangeably. Figure 4.1 below illustrates this method.

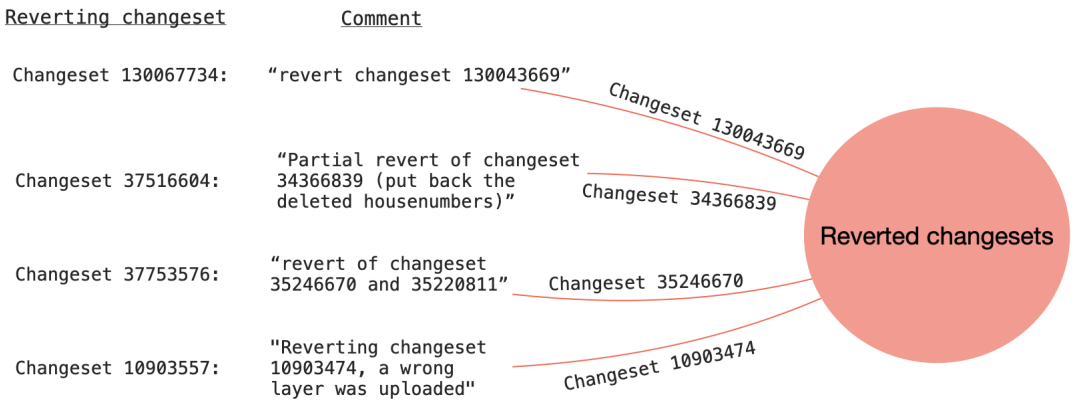


Figure 4.1: Figure illustrating how changesets' comments are scanned to find reverted changesets. Note in the third example that only changeset 35246670 is added to the data set while 35220811 is not, since the second ID is not preceeded by any of `{"changeset", "chgset", "cs"}`. Also note that the comment in changeset 10903557 contained the word ”reverting” rather than ”revert” but this still passes.

The key reason for using this method is that instead of trying to establish a number of heuristics to find erroneous changesets, this problem is circumvented by ”outsourcing” this judgement to the OSM community by using the changeset's comment tag. It should however be explicitly said that there is not a perfect match between erroneous changesets and suspicious changesets as some edits could be strange yet correct. The method is however deemed to be good enough as every erroneous changeset should be flagged for further inspection given a perfect model.

4.2 Finding Non-Reverted Changesets

To create a ground truth negative data set, an equal number of changesets were randomly sampled for the same time interval (December 1st 2011 to February 27th 2023), removing the changesets that were identified in section 4.1. This will be referred to as the *Not reverted* class. While this does not guarantee a data set free from reverts nor errors, it is not meant to constitute a data set entirely free from them either. Rather it represents the true distribution of OSM changesets as opposed to the reverted distribution, acting as a ”control group”.

However, it is still good to get some data point about the rate of correct *vs.* incorrect changesets in this data set. The method resulted in 27393 reverted changesets out of about 123 million total changesets, which is equivalent to a 0.022% probability of a changeset satisfying the requirements of the method. Of course, not every erroneous changeset is identified with the aforementioned method, but even a 100-fold increase yields an overwhelming majority of true negatives in the non-reverted data.

4.3 Preprocessing

The changesets found were then preprocessed into numerical features in a number of steps. The features used to represent a changeset are identical to the ones used by Tempelmeier and Demidova (2022), where a changeset's features are split into three parts - changeset, user and element data. The features, are given below as well as their variable names. All of the following data retrieving and processing was performed using either the OSM main API (*OpenStreetMap Wiki, API v0.6* 2023) or a combination of `PyOsmium` developed by Hoffmann (2015) and `Osmium Tool` by Topf (2013).

4.3.1 Changeset Data

The changeset data contains information solely about the changeset as a whole (and thus conceptually more similar to the changeset metadata file rather than the OsmChange file) and consists of the following:

Table 4.1: Features in the changeset data.

create_cs *modify_cs* *delete_cs* *edits*	Number of creates, modifies, deletes and total number of edits (their sum) in the changeset.
nnodes *nways* *nrelations*	Number of nodes, ways and relations that were subject to an edit in the changeset.
min_lon *min_lat* *max_lon* *max_lat*	Minimum/maximum longitude/latitude. These are defined by the eastern-, northern-, western- and southern-most elements subject to edits in the changeset.
box_size	The size of the bounding box as defined by minimum/maximum longitude/latitude.
comment_len	The number of characters in the comment.
imagery_used	If the source of the satellite imagery was given.
editor_app	Which editing application was used when creating the changeset. This is one-hot encoded to have seven levels: *josm*, *go map*!!, *osm go*!, *potlatch*, *streetcomplete*, *vespucci*, *other* where *other* includes making edits in the OSM web browser.

All of the above information was gathered using the changeset metadata and OsmChange files, downloaded from the OSM main API in Python. The OsmChange file contained information about the operations and elements while the changeset metadata file contained the rest of the features.

4.3.2 User Data

The user data contains information about all previous changesets committed to OSM by the user *prior* to the creation of the changeset. It gauges the experience level of the users as well as their editing patterns. It consists of the following features:

Table 4.2: Features in the user data.

create_u *modify_u* *delete_u* *contributions*	Total number of creates, modifies, deletes and total number of contributions (their sum) by the user to OSM.
create_nodes *create_ways* *create_relations*	Number of nodes, ways and relations that the user has created.
active_weeks	Number of unique active weeks. A user making many edits within one week will still only result in one active week, and is as such a measure of activity over time.
acc_created	Time of account creation.

This data was gathered by scanning both the changeset history and the element history files. Although no changesets before 1st December 2011 are eligible in this project, these are still included when aggregating the user history to obtain a fair estimate of users' experience. First, the changeset history file containing all previous OSM changesets was scanned to count the number of active weeks for all relevant users. Then, the full element history file containing every version of every OSM element was scanned to find all elements that were touched by the relevant users, giving the operations and element counts. These scans were performed with `PyOsmium`. Finally, the account creation time was retrieved from the OSM main API in Python.

4.3.3 Element Data

Finally, the element data contains information about the elements that were edited. As a changeset can contain edits to multiple elements, the following features were collected for every *individual* element subject to an edit in the changeset:

Table 4.3: Features in the element data.

operation	The changeset's operation (create/modify/delete) on the element. This is one-hot encoded.
type	The type of element (node/way/relation) that was edited. This is one-hot encoded.
version	Number of times this element has been edited.
ntags *ntags_added* *ntags_deleted*	Number of tags for the current version, and number of tags added and deleted.
nvalid_tags *nprev_valid_tags*	Number of valid tags for the current and previous version, see section 3.1.
weeks_to_prev	Weeks since last edit of the element.
name_changed	If the element's name tag was changed in the changeset.
nprev_auths	Number of unique users to edit the element before the changeset.

These features were gathered in two ways. First, the edit operation and element version number were retrieved by inspecting the OsmChange file of the changeset, which in turn was retrieved from the OSM main API in Python. While inspecting this file, all element IDs subject to edits in the changeset were stored. Second, the element history file was filtered using `Osmium Tool`, discarding all irrelevant element IDs. Third, using `PyOsmium`, the filtered element history file was scanned searching for all previous versions of the relevant elements. All of the past versions were used to count the number of previous authors, whereas only the previous version was used to calculate the rest of the features.

4.4 Final Dataset

Table 4.4 below shows the resulting number of data points of each class after preprocessing, which is slightly smaller than the initial size as some changesets were discarded due to raising errors during preprocessing.

Table 4.4: The final data set.

Class	Size
Reverted	24950
Not reverted	26875

Chapter 5
Models

One of the questions this project attempts to answer is how different types of models perform at anomaly detection in changesets. Because of this, four models are compared, two based on neural networks and two based on trees:

1. A multilayered perceptron (MLP). This is a simple neural network model with dense layers.
2. Random forest.
3. Gradient boosted trees.
4. Reduced Ovid. A slightly tweaked version of Ovid, the model presented by Tempelmeier and Demidova (2022).

These are crude, unrefined models meant to represent a lower bound of performance for a certain type of machine learning models. Because of this, the problem regarding inconsistent input dimensions of the element data (see section 4.3.3) is solved by simply circumventing it - the element data is omitted altogether for the MLP, random forest and the boosted trees model. Ovid however has a solution to this which is adopted in the Reduced Ovid model, explained in more detail in section 5.2.3.

5.1 Hyperparameter Tuning

This section presents the method used to find a decent set of hyperparameters for the MLP, random forest and boosted trees. Reduced Ovid's hyperparameters are not tuned, but instead set to be identical to those used in the original Ovid model to maintain maximum similarity with the results from Tempelmeier and Demidova (2022). The same method is used for all models - first, the data is split into a 80% training and 20% test data where only the training

data is used in the hyperparameter tuning. A grid search testing all possible hyperparameter combinations with 3-fold cross-validation is performed, where the the set of hyperparameters with the highest average validation score are considered best. All hyperparameters not mentioned are set to default values according to the corresponding package used: `SKLearn`'s `MLPClassifier` and `RandomForestClassifier` (Pedregosa et al. 2011) and `catboost`'s `CatBoostClassifier` (Dorogush, Ershov, and Gulin 2017).

5.1.1 Multilayered Perceptron

For the MLP, the data was scaled to zero mean and unit variance. Additionally, the grid search was performed in two steps as one search required an unfeasible amount of computational resources or time. The result from the first search was used in the second, which together form the final hyperparameters.

Table 5.1: Hyperparameters tested in the grid search for the MLP, using 3-fold cross validation. The best hyperparameters are marked in bold. The result from the first grid search (left table) was input into the second (right table).

Hyperparameter	Values tested
Hidden layer sizes	(10), (10, 10), (10, 10, 10), (100), **(100, 100)**, (100, 100, 100), (1000), (1000, 1000), (1000, 1000, 1000)
Maximum epochs	200, **400**, 600

$\xrightarrow{(100,100),\,400}$

Hyperparameter	Values tested
L2 regularization	0.001, 0.0001, **0.00001**
Batch size	100, **200**, 300
Learning rate	0.01, **0.001**, 0.0001

5.1.2 Random Forest

Random forest is an ensemble method that in general improves with larger ensembles until saturation (Probst and Boulesteix 2017). As such, it is good to ensure that the ensemble has approximately converged to the asymptote and will not improve considerably from more trees, which is done by testing three large numbers of trees and comparing the performance of the best set of hyperparameters with the other number of trees. If the difference in performance is small, convergence is achieved. The following hyperparameters were tested:

Table 5.2: Hyperparameters tested in the grid search with 3-fold cross-validation. The best set of hyperparameters are bolded.

Hyperparameter	Values tested
Number of trees	100, 250, **500**
Maximum depth	5, 10, 25, **50**
Maximum features	2, **5**, 10

With 500 trees performing best, it is sufficient to compare this model to one using identical hyperparameters but 250 trees instead of 500. The difference in accuracy for these models on the test data is 0.01 percentage points, showing that the ensemble is saturated.

5.1.3 Gradient Boosted Trees

Similarly to the random forest, with gradient boosted trees you want enough trees to approximately achieve convergence to a performance asymptote. Again, three large number of trees are compared to ensure saturation, however note that the number of trees used here are considerably larger than for the random forest. This is due to the nature of the model requiring more trees to saturate.

Table 5.3: Hyperparameters tested in the grid search with 3-fold cross-validation. The best set of hyperparameters are bolded.

Hyperparameter	Values tested
Number of trees	2000, 4000, **6000**
Depth	3, **7**, 10
L2 regularization	0.5, **3**, 10

Comparing the performance of the best set of hyperparameters with an identical model but with 4000 trees, the resulting difference in accuracy is 0.23 percentage points. The ensemble is estimated to be saturated, as a 50% increase of number of trees resulted in only a marginal gain in accuracy.

5.1.4 Final Hyperparameters

The final hyperparameters used for all models are the following:

MLP	
Hidden layer sizes	(100, 100)
Maximum epochs	400
L2 regularization	0.00001
Batch size	200
Learning rate	0.001

Random Forest	
Number of trees	500
Depth	50
Max features	5

Boosted trees	
Number of trees	6000
Depth	7
L2 regularization	3

5.2 Reduced Ovid

Because this project builds upon previous work by Tempelmeier and Demidova (2022), it is interesting to compare the performance of the vandalism detection model presented in that project - Ovid - on the data set used in this project, which is intended to contain more unintentional mistakes and should represent a somewhat different distribution.

In order to make Ovid compatible with the data used in this project, some minor tweaks had to be made. First, because of difficulty in gathering information about tags for objects in a reasonable time span, the top 12 keywords variable used in Ovid has been omitted. Secondly, for the element data, Ovid uses a feature representing the distance an element has been moved for an edit, which is also omitted in this project as its usage was not mentioned in the paper but rather became evident when reviewing Ovid's source code. This became an issue as all changesets were preprocessed on the fly, making this information unavailable given the time constraints for this project. For this reason, we call the model in this project *Reduced Ovid*.

5.2.1 Architecture of Ovid

Ovid is a complex architecture compared to the other models presented in this project, utilizing multi-head attenion and a number of fully connected (FC), concatenation (concat) and normalization (norm) layers. Figure 5.1 is taken from Tempelmeier and Demidova (2022), which illustrates the model.

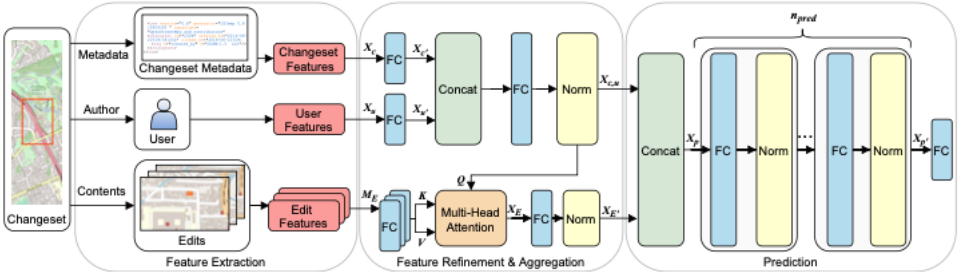


Figure 5.1: A figure taken from Tempelmeier and Demidova (2022) describing the architecture of Ovid.

The fully connected layer is a dense neural network layer, the normalization layer normalizes every individual sample, and a concatenation simply joins two output vectors of size m and n into one vector of size $m + n$. Finally, the multi-head attention is used to extract the most important edits in the changeset, more on this in section 5.2.3.

5.2.2 Hyperparameters

In order to minimize the difference between Ovid and Reduced Ovid, this project will not optimize Reduced Ovid's hyperparameters but rather use the same set of hyperparameters as used in the source code for Ovid. These are however tuned for the vandalism data set, which could result in some performance loss when applying the model to the data set based on reverts. The following hyperparameters are used (Tempelmeier and Demidova 2021):

Table 5.4: Hyperparameters used for Reduced Ovid.

Max edits	20
n_{pred}	1
$hidden_{pred}$	24
Dropout	0.5
Patience	10
Number of attention heads	10
Batch size	2000

Max edits define a maximum threshold for the number of edits in a changeset, before the entire element data ($X_{E'}$) is set to zero and only the user and changeset data is used. n_{pred} is the number of repeating FC → Norm blocks in the rightmost section of Ovid's architecture in figure 5.1, and $hidden_{pred}$ is the number of nodes in some of the hidden layers, explained further in the following table below. Each FC layer will be identified by the input/output or previous/next layer name:

Table 5.5: Number of nodes in the FC layers of Reduced Ovid. The layers are identified by the input/output or previous/next layer names as defined by figure 5.1.

FC layer	Number of nodes
$X_c \rightarrow FC \rightarrow X_{c'}$	$n_{changeset\ features} + n_{user\ features}$
$X_u \rightarrow FC \rightarrow X_{u'}$	$n_{changeset\ features} + n_{user\ features}$
Concat → FC → Norm → $X_{c,u}$	$n_{changeset\ features} + n_{user\ features}$
$M_E \rightarrow FC \rightarrow$ Multi-Head Attention	$n_{edit\ features}$
$X_E \rightarrow FC \rightarrow$ Norm	$hidden_{pred}$
$X_p \rightarrow FC \rightarrow ... \rightarrow X_{p'}$	$hidden_{pred}$

5.2.3 Element Data in Ovid

Because of Reduced Ovid utilizing multi-head attention, it does in fact have access to the element data unlike the MLP, random forest and boosted trees models. In addition to multi-head attention, it introduces an upper bound of 20 edits per changeset, and if this threshold is exceeded the element data is entirely omitted, setting $X_{E'} = \mathbf{0}$. Their reasoning is that one individual edit is negligible if $n_{edits} > 20$ (Tempelmeier and Demidova 2022).

Explaining the details of Reduced Ovid's element data implementation will become relevant when analyzing feature importance in section 6. Every feature in the element data is prefixed by an index in the interval $[0, 19]$ which corresponds to an edit. Hence element data features are repeated 20 times, each time for a different edit. Additionally, a ***mask*** feature is used for each edit, which determines whether the multi-head mechanism has access to it. Manipulating ***mask*** is how $X_{E'}$ can be set to zero - by "hiding" every edit in input vector from the multi-head attention, which is also how non-existent edits are made unavailable for changesets with fewer than 20 edits. The formal logic for ***mask*** for edit i for a changeset with n_{edits} edits is given in the equation below (i is 0-indexed):

$$mask(i, n_{edits}) = \begin{cases} \text{True}, & \text{if } i < n_{edits} \text{ and } n_{edits} \leq 20 \\ \text{False}, & \text{otherwise} \end{cases}$$

5.3 Performance

Figure 5.2 shows the confusion matrices and tables 5.6, 5.7 shows the precision, recall, F1 and accuracy. All metrics are calculated on the test data.

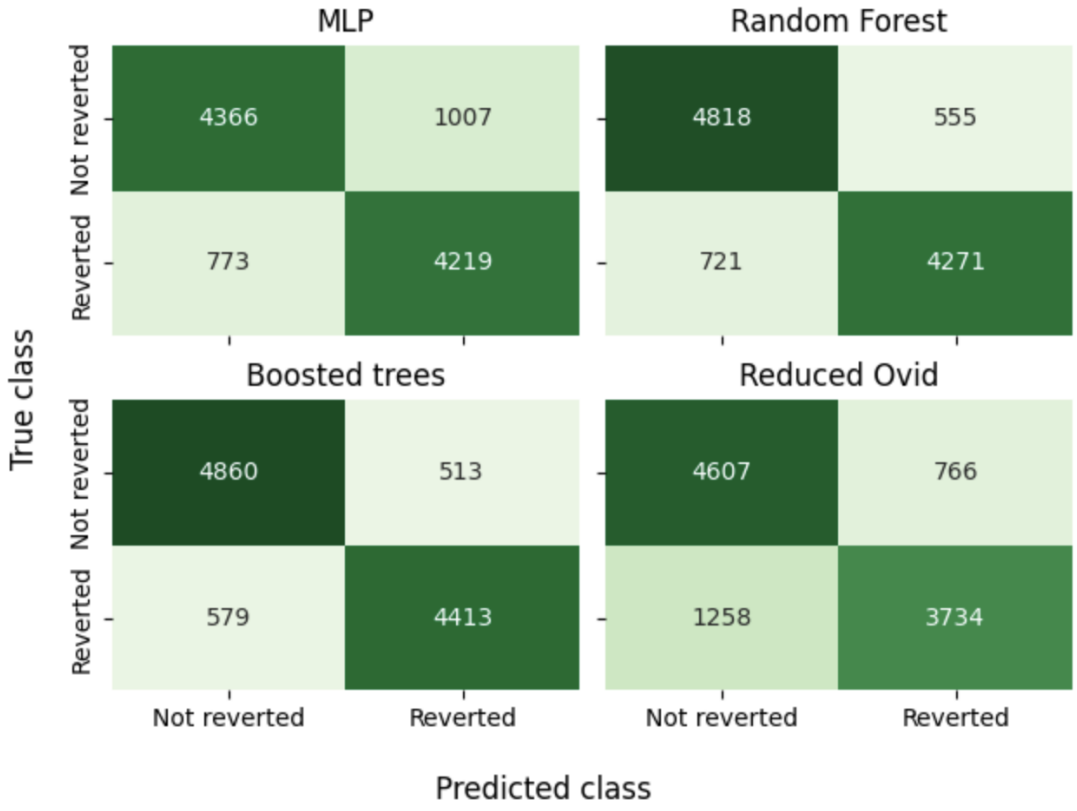


Figure 5.2: Confusion matrices.

Table 5.6: Precision, recall and F1-score.

MLP			
	Precision	Recall	F1
Not reverted	0.8496	0.8126	0.8307
Reverted	0.8073	0.8452	0.8258

Random forest			
	Precision	Recall	F1
Not reverted	0.8698	0.8967	0.8831
Reverted	0.8850	0.8556	0.8700

Boosted trees			
	Precision	Recall	F1
Not reverted	0.8935	0.9045	0.8990
Reverted	0.8959	0.8840	0.8899

Reduced Ovid			
	Precision	Recall	F1
Not reverted	0.7855	0.8574	0.8199
Reverted	0.8298	0.7480	0.7868

Table 5.7: Accuracy.

Accuracy			
MLP	Random forest	Boosted trees	Reduced Ovid
0.8283	0.8769	0.8946	0.8047

Chapter 6
Model Analysis

Because the performance in section 5 was very good, in particular for the tree models, what follows is an extensive analysis of how the models perform so well. To do this, first the SHAP values will be presented and analyzed briefly. Secondly, the models will be retrained on a reduced set of features to find more information about them, and the features containing strong signals will be analyzed further to find the reason for this signal. The SHAP values will complement this in depth feature analysis. Throughout this analysis, all models presented in section 5 will be trained on the reduced set of features to find potential differences between them, however the main focus of this section is not to make comparisons between the models, but rather an investigation of feature importance - only in the initial SHAP analysis will the models be compared thoroughly, otherwise only major differences between models will be commented.

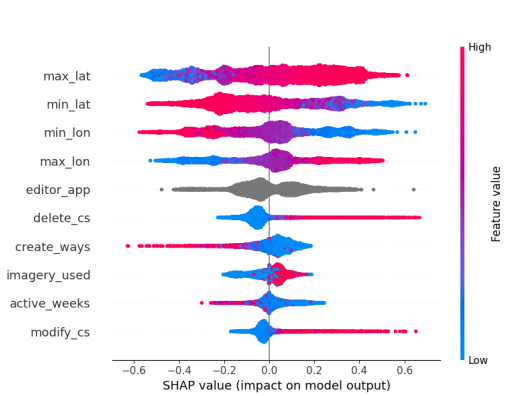
Only the changeset and user data will be analyzed in detail for two reasons. First, only Reduced Ovid actually uses the element data, and considering that this model performs worst with regards to most metrics, the element data does not seem be very effective in this case. Second, there is a strong signal in the user and changeset data which is interesting to investigate further, especially considering that these features do not contain information about individual edits, which is causal in deciding if an edit is correct.

Below, the model hyperparameters and architectures as well as the data split will be identical to the one used in section 5. Reduced Ovid uses 10% of the training data as validation for early stopping. Note that when omitting features the hyperparameters found may not be as effective. This is ignored as finding a new set of hyperparameters for each reduced model is unfeasible, however this may affect results.

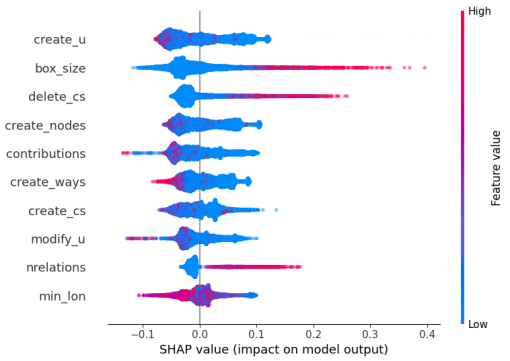
6.1 SHAP Values

In figure 6.1 the 10 most important features (out of 24 total in the changeset and user data or 344 if also using the element data) and their SHAP values are presented in descending order,

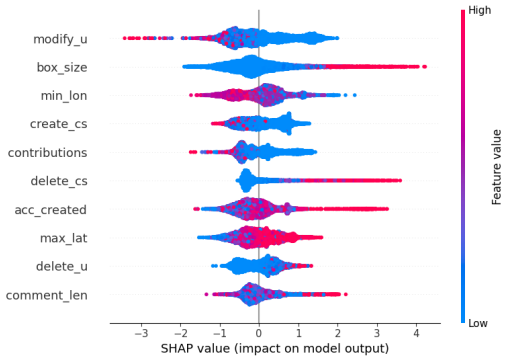
decided by the mean absolute SHAP value.



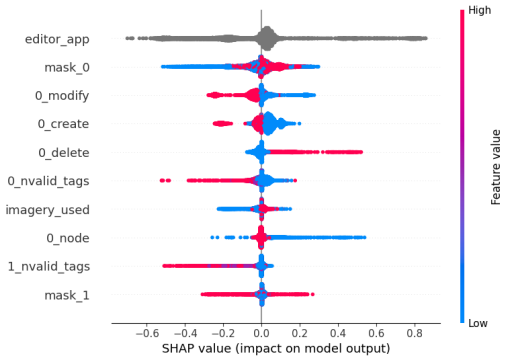
(a) MLP.



(b) Random forest.



(c) Boosted trees.



(d) Reduced Ovid.

Figure 6.1: The 10 most important features ordered descendingly according to the SHAP values for each model. A red and blue data point indicates a high and low relative feature magnitude, respectively. A large distance from $x = 0$ indicates a large SHAP value which implies a large impact on the model's prediction. The features are ordered according to their mean absolute value, placing an emphasis on high *average* impact.

A number of observations can be made from figure 6.1. One, there is a considerable overlap between the ten most important features for both tree models - they share ***delete_cs***, ***box_size***, ***contributions***, ***min_lon***, ***create_cs*** and ***modify_u***, and the SHAP values look similar for these features. Second, the tree models seem to give larger importance to features related to user experience than the ANN models, as the random forest and boosted trees have four and five out of ten features from the user data, while the MLP has 2 and Reduced Ovid has 0.

Third, the MLP places a very large emphasis on the location of the edit, as ***min/max_lon/lat*** constitute the four most important features, where it is surprising that ***min*** and ***max*** have reversed behaviours - this is discussed further in section 6.3.2. Fourth, both ANN models weigh ***editor_app*** very heavily whereas the tree models do not. Fifth, eight out of ten features for Reduced Ovid are from the element data, and in particular only the first two edits, as can be seen by the prefixed index 0 and 1.

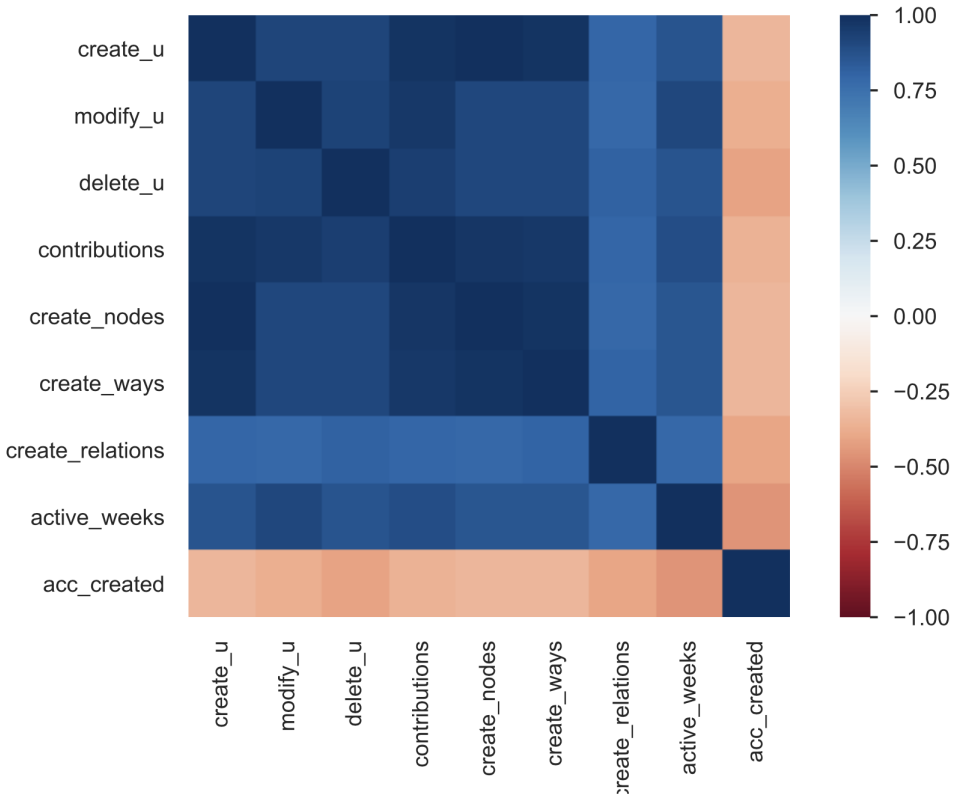


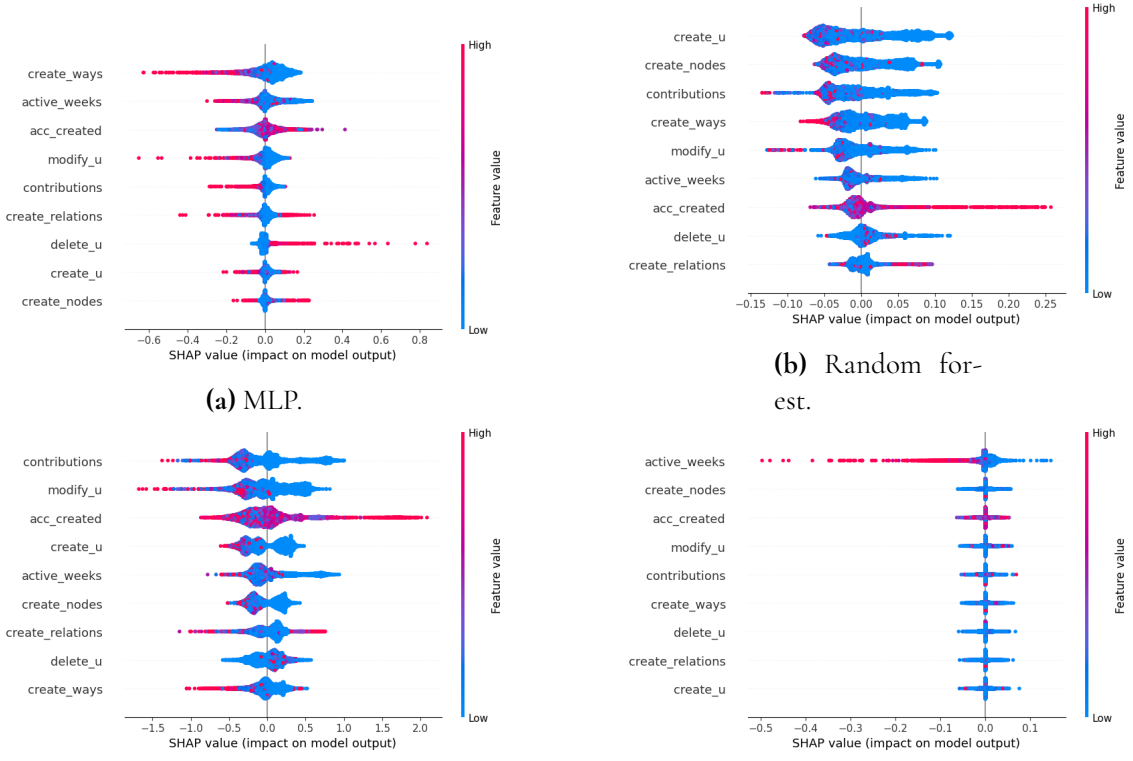
Figure 6.2: Heat map of the correlations between variables in the user data.

These observations and the SHAP values rank orderings will be referred to during the further analysis below. The changeset and user data will be analyzed separately because of similarity in the information contained in the features within each category.

6.2 User Data

To begin the analysis of the user data, the correlation matrix is presented in figure 6.2 to find potential redundancy in the features. It is presented as a heat map to give a better overview. The correlations between all variables except *acc_created* are very high. This is not unreasonable, considering that all of these variables increase monotonically as a user commits more changesets to OSM. *acc_created* correlating negatively is also expected as newer accounts should on average have made fewer edits.

To gain a better understanding for these features, their SHAP values are given in figure 6.3. The SHAP values indicate a trend of low user experience leading to higher probability of predicting a sample as reverted. They also introduce a rank ordering between these features, but considering the high redundancy in the features it is not unreasonable to think that any one feature in the user data is not significantly more important than another. To investigate this, a univariate analysis using only one feature from the user data is performed to see if any one feature gives a higher contribution than another. Table 6.1 gives the accuracy metrics for the univariate models, first including the changeset data, and then omitting it. It is omitted



(a) MLP.

(b) Random forest.

(c) Boosted trees.

(d) Reduced Ovid.

Figure 6.3: The SHAP values for the user data features. The models are trained on all features.

to find potential overlap in information between the user and changeset data, to amplify the signals in these features and to give a point of comparison.

Two specific things should be noted before making a general analysis of these results. First, Reduced Ovid has the same performance for all features when omitting the changeset data. This is most likely due to the signal in the single feature used being lost among the large number of element data features in combination with the models being trained with a fixed random seed. Secondly, random forest trained on *acc_created* performs significantly better than any other model. This could be due to data leakage, since the account creation date is translated to a timestamp which could be used as an ID, as some users occur more than once in the data set. It is however unclear why the other models did not exploit this leakage, if that is the case.

As for the general analysis, two things can be observed when including the changeset data. First, most features achieve fairly similar accuracy metrics with the different features, although all models agree that *create_relations* is a slightly worse predictor than most other features. Secondly, using one user data feature as compared to none gives a significant performance boost in most cases, but the same is also true for using all features rather than one. This indicates that the user's experience level is important when predicting, and while there is a lot of overlap in the user data features, they do not all simply gauge general user experience but also provides slightly different information. Comparing the results with and without the changeset data, it can be seen that all models perform well on the *active_weeks* feature (arguably the most general user experience heuristic) when omitting the changeset

Table 6.1: Accuracy on the test data for the univariate models. The row represent the single user data feature used. All and None represents two baseline models where all features in the user data are included and omitted, respectively. Bolded values are the best performing feature for each model.

Accuracy, with changeset data				
	MLP	Random forest	Boosted trees	Reduced Ovid
create_u	0.8020	0.8644	0.8719	0.7807
modify_u	0.7945	**0.8699**	0.8727	0.7657
delete_u	0.7852	0.8680	0.8724	0.7483
contributions	0.8071	0.8663	0.8725	0.7777
create_nodes	0.8014	0.8659	**0.8752**	0.7716
create_ways	0.8032	0.8669	0.8733	0.7719
create_relations	0.7922	0.8528	0.8642	0.7485
active_weeks	**0.8146**	0.8636	0.8700	**0.7862**
acc_created	0.8028	0.8521	0.8648	0.7727
All	0.8283	0.8769	0.8946	0.8047
None	0.7745	0.8334	0.8408	0.7338

Accuracy, without changeset data				
	MLP	Random forest	Boosted trees	Reduced Ovid
create_u	0.6787	0.6736	0.7004	**0.6771**
modify_u	0.6348	0.6408	0.6774	**0.6771**
delete_u	0.6515	0.6758	0.6829	**0.6771**
contributions	0.6774	0.6403	0.6869	**0.6771**
create_nodes	0.6747	0.6685	0.6964	**0.6771**
create_ways	0.6824	0.6687	0.6907	**0.6771**
create_relations	0.6304	0.6532	0.6526	**0.6771**
active_weeks	**0.6881**	0.7007	**0.7011**	**0.6771**
acc_created	0.6053	**0.7565**	0.6455	**0.6771**
All	0.7436	0.8160	0.8167	0.7339

data, but the same is not true for the tree models when including it. This suggests that there could be some patterns between the changeset data and user data which the tree models find, perhaps that the committed changeset follows previous editing behaviour.

There may be some behaviour where the models are aggressive against lower experienced users, as indicated by the SHAP values. This may be problematic on a general data set, as it will almost surely not have a 50/50-split of good and suspicious edits. This behaviour could be identified if the predictions are partitioned based on the users' experience levels. To do this, the *active_weeks* feature is used as a heuristic for general user experience. The thresholds are set so that the low experienced category is set strictly to ensure a large majority of beginners, while the threshold between the medium and high experienced users are set so that the number of reverted samples in these categories are approximately equal.

Table 6.2: The test data was partitioned based on experience levels according to *active_weeks*.

Interval	Experience level	Reverted samples	Non-reverted samples
$active\_weeks = 0, 1$	Low	1916	596
$active\_weeks = 2, ..., 24$	Medium	1537	1770
$active\_weeks \geq 25$	High	1539	3007

Figure 6.4 gives the precision and recall metrics for both classes for all models. The full set of user and changeset data features are used.

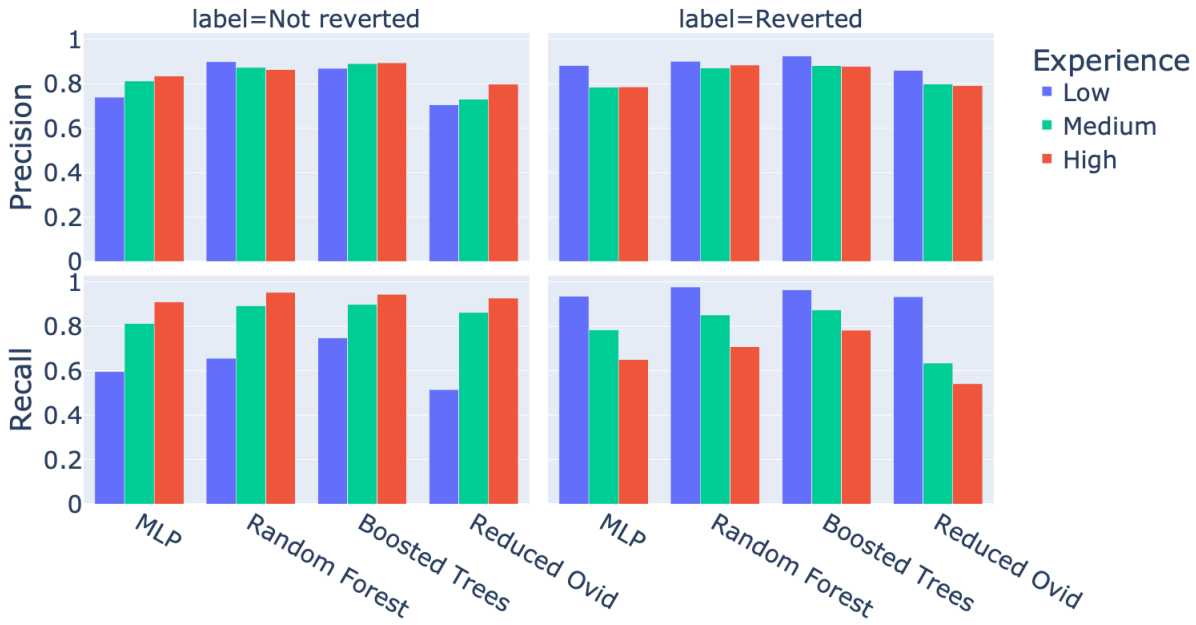


Figure 6.4: Precision and recall when partitioning the test data set based on experience levels according to table 6.2 for all models. As a reminder, $\text{Precision} = TP/(TP+FP)$ and $\text{Recall} = TP/(TP+FN)$

While precision is reasonably balanced, there is a clear pattern in recall - the number of false negatives decrease with experience level for the non-reverted data, and increases in the

reverted data. Intuitively, this means that all models are more aggressive for changesets where the user has low experience and more lenient for those with high experience. While this still leads to high accuracy metrics due to the imbalanced data set in these categories, one would expect this to cause generalization problems as a general distribution would most likely contain a large majority of correct changesets regardless of experience level. This problem would be most apparent for beginners where the models are most aggressive, but might cause similar problems albeit probably to a smaller extent in the other categories as well. How the models could be expected to generalize is discussed further in section 8.

6.3 Changeset Data

To begin the analysis of the changeset data, the correlations are presented in figure 6.5. They are presented as a heat map to give a better overview. Some clusters in the correlations can be observed, such as in the top left and bottom right corners, as well as two along the diagonal. Based on the correlations and intuitive similarity in information in the features, three feature categories will be introduced where the attempt is to minimize the overlap in information between these categories to gauge the importance of each type of information. Table 6.3 presents these categories.

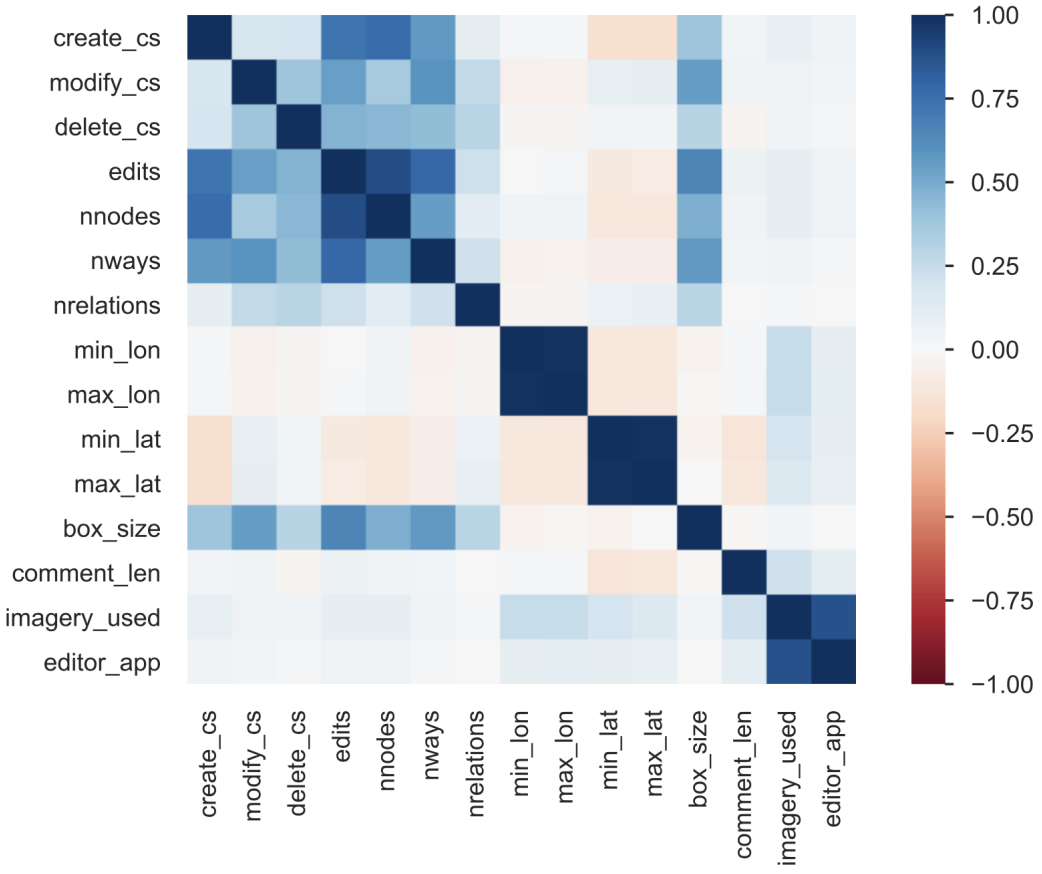


Figure 6.5: Correlations in the changeset data.

Table 6.3: The three categories that the changeset data features are split into.

Category	Features
Edit	*create_cs*, *modify_cs*, *delete_cs*, *edits*, *nnodes*, *nways*, *nrelations*, *box_size*
Location	*min_lon*, *min_lat*, *max_lon*, *max_lat*
Miscellaneous	*imagery_used*, *editor_app*, *comment_len*

The motivation for the categories are the following:

- **Edit**: There is a cluster of higher correlations (top left in the heatmap), and these variables have a comparatively strong correlation with *box_size*. The information in this category represent what kind and how many edits were made in the changeset, to which *box_size* was added because of the high correlation. This is intuitive, as when *box_size* increases, so does most likely the number of edits, which in turn increase the other features.
- **Location**: Minimum and maximum longitude and latitude give the location of the edits. It could be argued that *box_size* should belong to this category as it can be inferred from these features, but considering that *box_size* not only correlate very weakly with *min/max_lon/lat*, but also correlate considerably more with the features in the Edit category, *box_size* was instead included in the Edit category to minimize information overlap.
- **Miscellaneous**: This category contains the three remaining variables with no clear common denominator. *editor_app* and *imagery_used* do however have a very high correlation (0.869) and is thus reasonable to place in the same category. *comment_len* is also placed in this category despite a low correlation and no obvious overlapping information. The motivation for this is that the SHAP values in figure 6.1 indicate that *comment_len* is not a very important feature as it is only among the boosted trees 10 most important features, and creating a new category for this is deemed excessive.

A similar analysis to the user data is performed, but now using the categories instead of individual features. With only three categories however, all possible combinations can be tested. Table 6.4 shows the accuracy for the reduced models with and without the user data. Some observations can be made from table 6.4. First, all models agree that Edit + Location performs best with some margin when including the user data. Secondly, when omitting the user data, there is a clear pattern that the tree models find a significantly stronger signal in the Location category than any other, and that this pattern does not exist for the ANN models. This strong signal is somewhat unintuitive and will be analyzed further in section 6.3.2. Thirdly, all models predict better on only the changeset data rather than user data, as can be seen by comparing None in the upper table with All in the lower.

What follows is an analysis of the information contained in and a discussion of their importances.

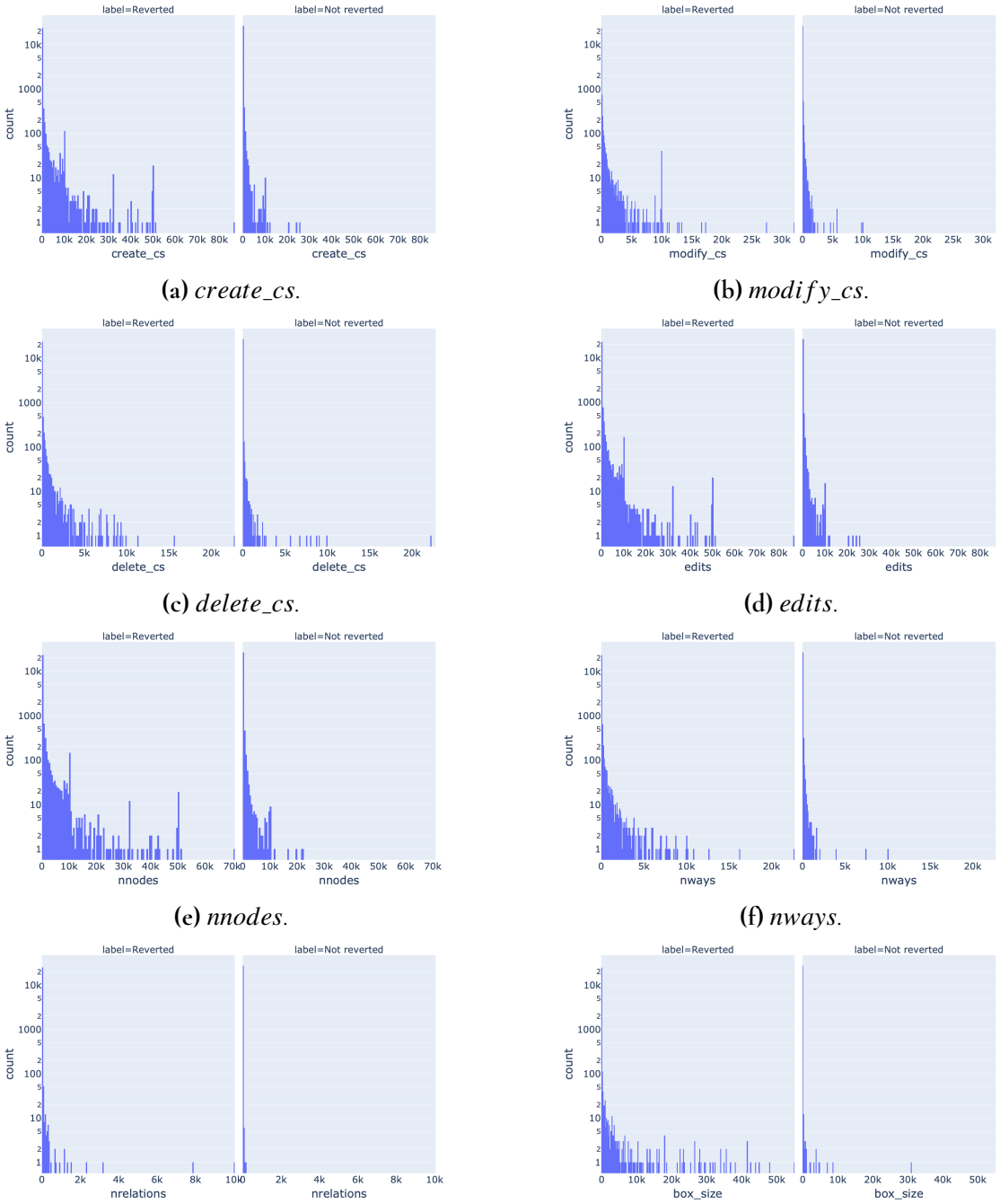
6.3.1 Edit Category

Figure 6.6 presents the univariate histograms are presented for the Edit category features, split on ground truth label. The most apparent difference between the reverted and non-

Table 6.4: The accuracy on the test data for the reduced models. Each row represents the categories trained upon. All and None are given as baselines, where all categories are included and omitted, respectively. None is not given in the second table, as this corresponds to an empty model for all but Reduced Ovid. Bolded values are the best performing category for each model, which are given twice - once for two category models, and once for one category models.

Accuracy, with user data				
	MLP	Random forest	Boosted trees	Reduced Ovid
Edit + Location	**0.8159**	**0.8687**	**0.8836**	**0.8041**
Edit + Miscellaneous	0.8024	0.8542	0.8597	0.7914
Location + Miscellaneous	0.8006	0.8576	0.8620	0.7878
Edit	**0.7899**	0.8439	**0.8538**	0.7575
Location	0.7819	**0.8502**	0.8537	0.7231
Miscellaneous	0.7559	0.8304	0.8227	**0.7605**
All	0.8283	0.8769	0.8946	0.8047
None	0.7436	0.8160	0.8167	0.7339

Accuracy, without user data				
	MLP	Random forest	Boosted trees	Reduced Ovid
Edit + Location	0.7152	**0.8059**	**0.8094**	0.7109
Edit + Miscellaneous	0.7082	0.7465	0.7617	0.7119
Location + Miscellaneous	**0.7248**	0.8038	0.7945	**0.7279**
Edit	0.6520	0.6694	0.6870	**0.7021**
Location	**0.6740**	**0.7678**	**0.7512**	0.6904
Miscellaneous	0.6259	0.6403	0.6410	0.6943
All	0.7638	0.8347	0.8408	0.7431



(a) *create_cs*.

(b) *modify_cs*.

(c) *delete_cs*.

(d) *edits*.

(e) *nnodes*.

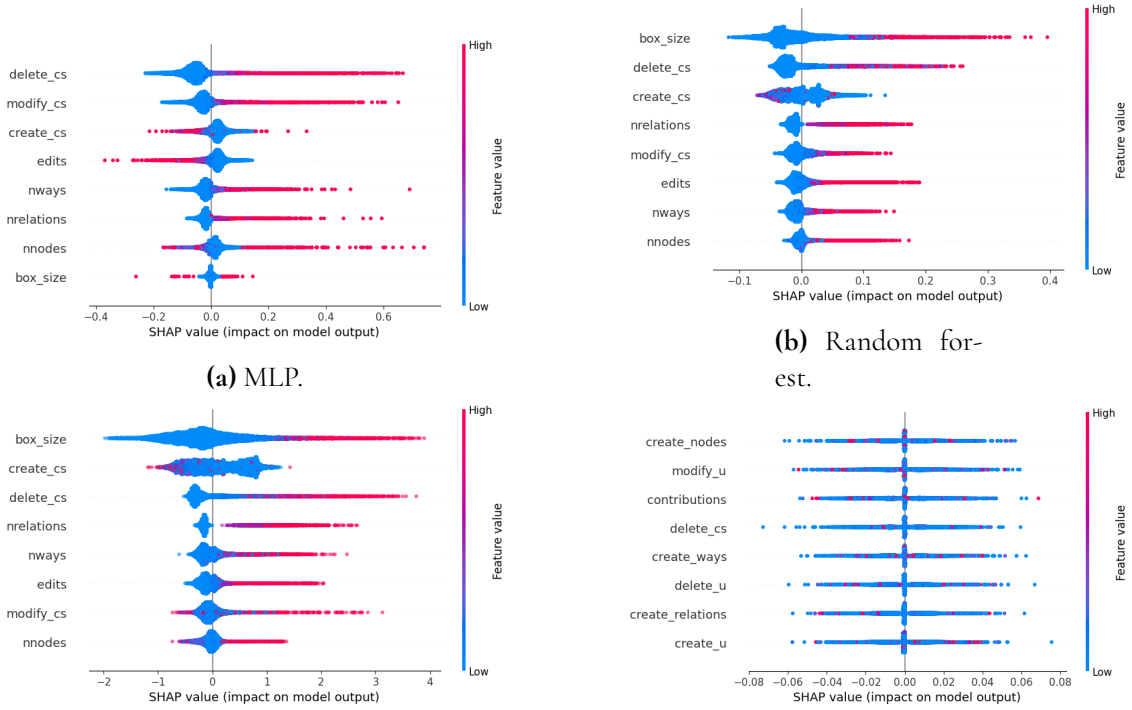
(f) *nways*.

(g) *nrelations*.

(h) *box_size*.

Figure 6.6: Histograms for all features in the Edit category split on ground truth label. Note the logarithmic *y*-axis. A clear pattern can be observed, where the reverted data has more large values.

reverted data can be seen in the tails of the distributions, showing that changesets containing a very large number of edits are more prevalent in the reverted data. This is not unreasonable as when the number of edits increase, so does most likely also the probability that at least *one* of them is incorrect and is consequently reverted. Additionally, the large amount of edits within one changeset indicate that there could be vandalism in the dataset despite making an attempt to exclude it.



(a) MLP.

(b) Random forest.

(c) Boosted trees.

(d) Reduced Ovid.

Figure 6.7: The SHAP values for the Edit category features. The models are trained on all features.

The SHAP values shown in figure 6.7 correspond well with the histograms, where a sample falling into the tail of the histogram has a considerably higher probability to be classified as reverted. This holds for all models but Reduced Ovid which has a strange behaviour, most likely due to the model giving a much heavier weight to the element data than other the features in the Edit category.

To further investigate the Edit category's outlier detection properties, the predictions of the models trained on all features is compared to an identical model, but where the Edit category is omitted. Table 6.5 gives the confusion matrices for the samples in the test data lying above the 99th percentile of the *non-reverted* distributions of any Edit category feature, corresponding to 877 reverted and 237 non-reverted data points.

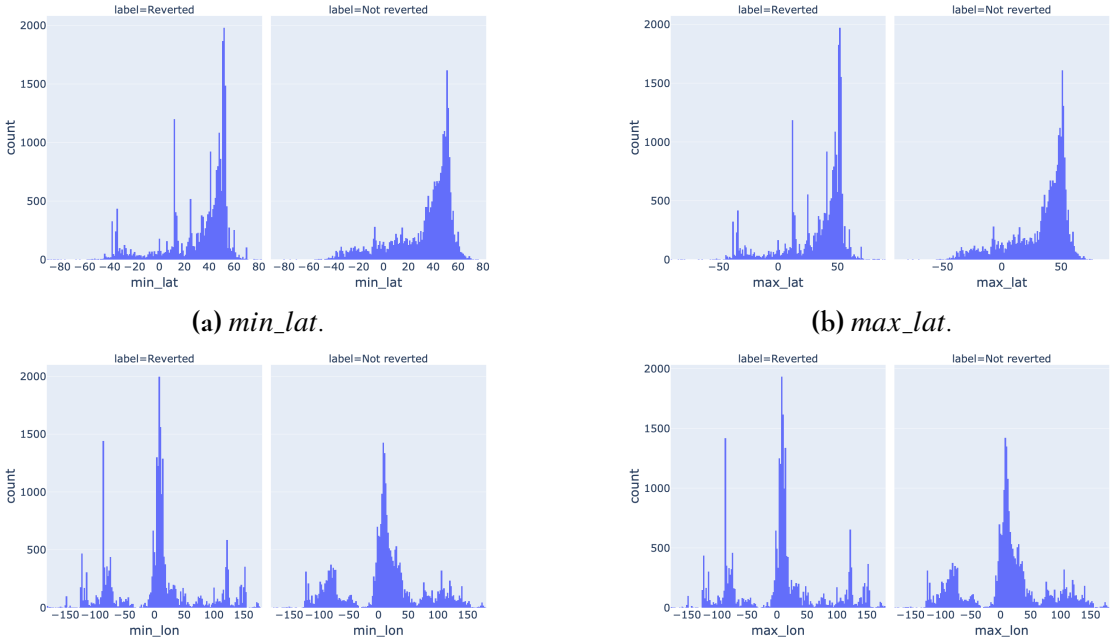
Table 6.5: Confusion matrices for the samples in the test data above the 99th percentile of the *non-reverted* distributions with respect to any of the features in the Edit category. The reduced model uses all features except the Edit category. The rows represent the ground truth and columns the predictions.

MLP				
	Reduced		All	
	Not reverted	Reverted	Not reverted	Reverted
Not reverted	176	61	129	108
Reverted	248	629	106	771
Random Forest				
Not reverted	219	18	127	110
Reverted	267	610	49	828
Boosted Trees				
Not reverted	192	45	134	103
Reverted	210	667	47	830
Reduced Ovid				
Not reverted	152	85	105	132
Reverted	256	621	108	769

While including the Edit category provides an improvement in terms of overall accuracy, table 6.5 shows that classifying the non-reverted class for these samples is more difficult when introducing the Edit category and vice versa for the reverted class. This is not unexpected, but highlights that these features introduce a trade-off between overall and edge case (in the not reverted class) accuracy that could be relevant depending on the use case.

6.3.2 Location Category

The univariate histograms of the features in the Location category split on ground truth label are given in figure 6.8. Some thin peaks in the histograms are seen in the reverted data which does not exist in the non-reverted data.



(a) *min_lat*.

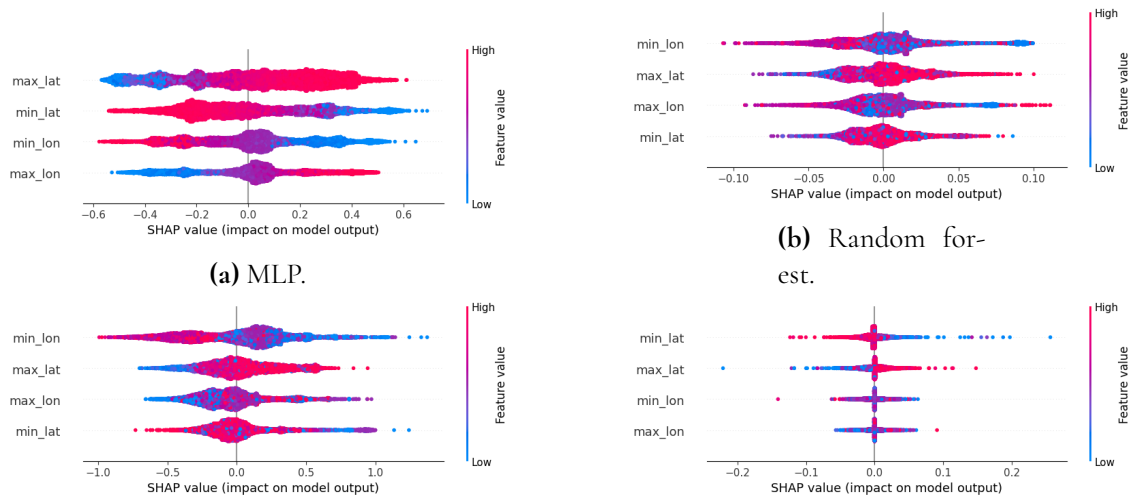
(b) *max_lat*.

(c) *min_lon*.

(d) *max_lon*.

Figure 6.8: Histograms for the Location category features split on label.

Figure 6.9 shows the SHAP values for these features to give further insight.



(a) MLP.

(b) Random forest.

(c) Boosted trees.

(d) Reduced Ovid.

Figure 6.9: The SHAP values for the Location category features. The models are trained on all features.

The SHAP values look relatively similar for the tree models, with a rough pattern that a large *lat* and small *lon* increases the probability to classify as reverted and vice versa. The MLP however shows a interesting trend that was mentioned in section 6.1 - *min* and *max* have opposite behaviour. This pattern appears in Reduced Ovid as well however to a smaller extent and and only for *lat*. This may be due to the MLP mainly using these features to infer

box_size, as a small ***box_size*** is less suspicious than a large one (see the histogram in figure 6.6h). This idea is supported by ***box_size*** being a weak feature for the MLP as seen in figure 6.7a but a very important feature for the better performing tree models, according to the SHAP values.

To briefly test this, table 6.6 presents two models, one trained on *only* the Location category, and the same model but where ***box_size*** is included.

Table 6.6: The accuracy of the models trained on only the Location category vs. the Location category and ***box_size***. No other features are used.

	MLP	Random forest	Boosted trees	Reduced Ovid
Location	0.6740	0.7678	0.7512	0.6904
Location + ***box_size***	0.6757	0.7645	0.7644	0.7217

Indeed, the MLP barely improves when introducing ***box_size***, indicating that ***min/max_lon/lat*** are used to find outliers with regards to it. Interestingly, the random forest does not improve and the boosted trees improves little despite the SHAP values ranking it as a very important feature, perhaps suggesting multivariate relations with other features. Nonetheless, the tree models' accuracies are very high considering that the location of the edited elements should intuitively not be important. Additionally, table 6.4 shows that out of the models trained on no user data and only one changeset data category, the best were the tree models by a significant margin, finding a strong signal in these features.

To inspect this further, a scatter plot of the test data in figure 6.10 is presented. Here, ***min_lon*** is plotted against ***min_lat*** and split on the ground truth label, where the colors correspond to the predictions of the random forest model trained on only the Location category. This paints something akin to a world map. The choice of ***min*** instead of ***max*** is arbitrary - both illustrate the data well and the plot looks essentially the same. It can be observed that both in terms of the predicted and ground truth data, there is a larger spread of the non-reverted data over the continents and conversely the reverted data is more clustered along densely populated coasts and Europe - perhaps areas with a more active OSM community. A possible explanation could simply be that areas with active OSM communities are more likely to inspect existing elements and find editing errors, while also putting in effort into reverting it.

Furthermore, this "map" highlights a difference between the distributions of reverted and erroneous changesets, as editing errors are probably not more frequent in Europe and densely populated areas as the map suggests, but they rather have larger OSM communities finding and reverting these errors, which causes this skewed distribution toward densely populated areas and creates the strong signal in latitude and longitude.

6.3.3 Miscellaneous Category

The univariate histograms for the features in the Miscellaneous category are presented in figure 6.11, split on the ground truth label. Some differences can be observed between the reverted and non-reverted distributions, such as a number of "spikes" in the reverted data, most notably when ***comment_len*** assumes its maximum value. Additionally, a higher probability

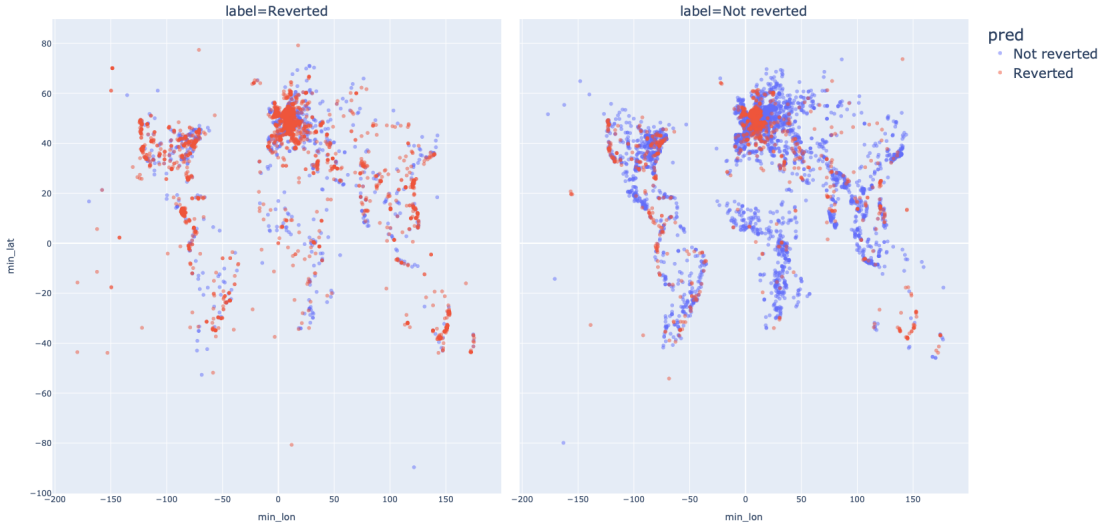
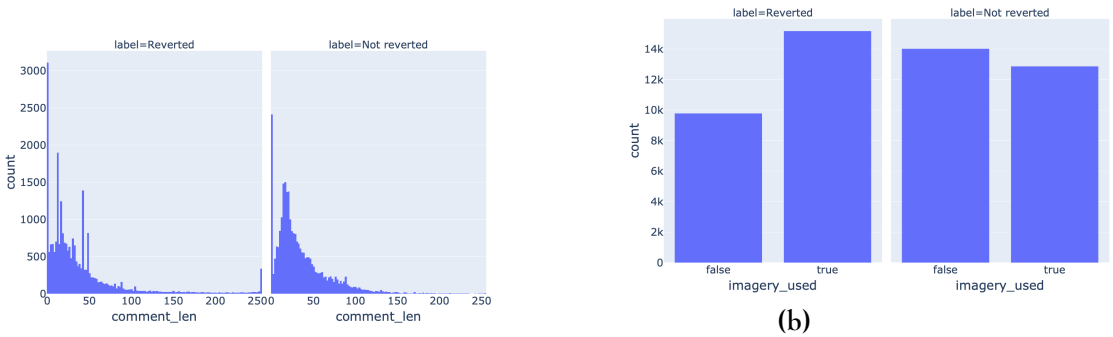


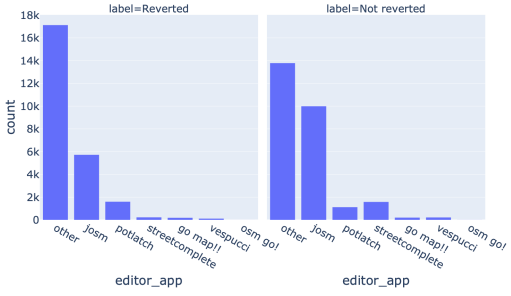
Figure 6.10: A scatter plot of the test data set, where *min_lon* is plotted against *min_lat*, split on the ground truth label. The color corresponds to the predictions of a random forest model's prediction trained only on the Location category features.

that *imagery_used* is true in the reverted data and some differences for *editor_app = other* and *editor_app = josm*.



(a) *comment_len*.

(b) *imagery_used*.

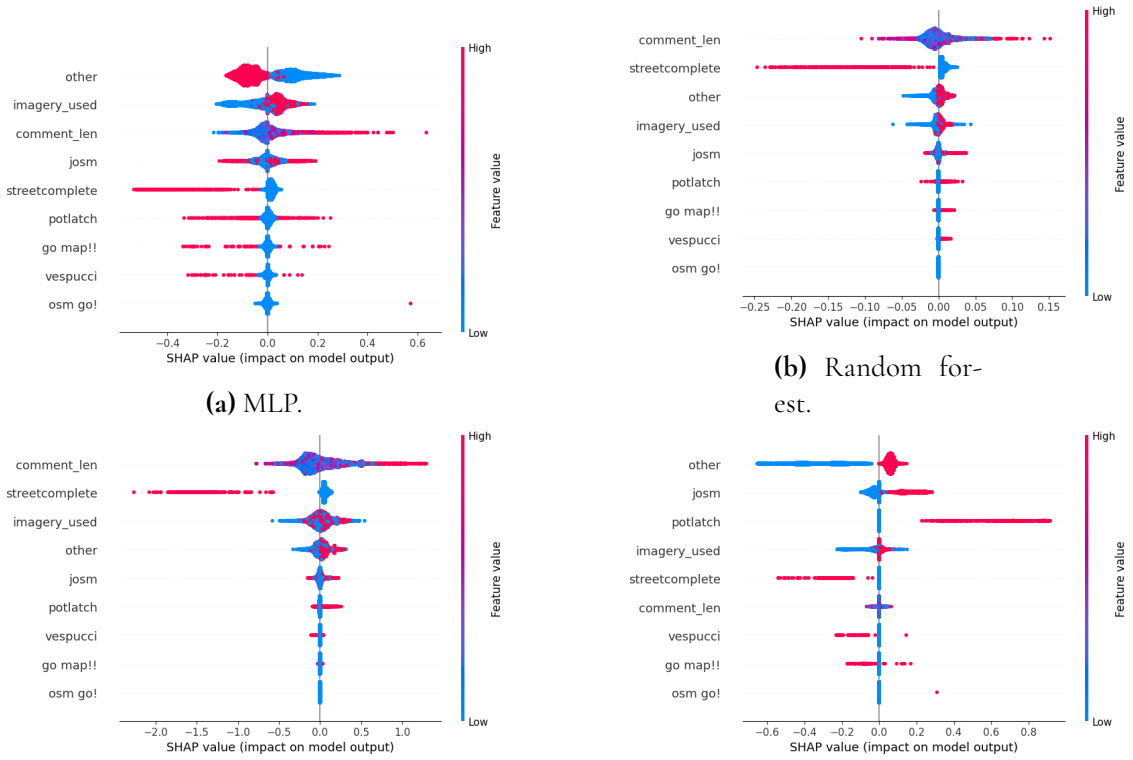


(c) *editor_app*.

Figure 6.11: Histograms for the Miscellaneous category features.

Furthermore, the SHAP values for these features is given in figure 6.12. Here, *editor_app*

is not aggregated as in figure 6.1 but instead split into each individual level to gain more information. These show that all models do not agree on how to interpret most levels of *editor_app*. Interestingly, some SHAP values even reverses the intuitive relation between the histograms and predictions, such as the MLP's *other* and Reduced Ovid's *josm*. Furthermore, a large *comment_len* is associated with an increased probability to classify a changeset as reverted.



(a) MLP.

(b) Random forest.

(c) Boosted trees.

(d) Reduced Ovid.

Figure 6.12: The SHAP values for the Miscellaneous category features. The models are trained on all features.

While there are no obvious patterns to analyze, some insight can be given into these features. A manual inspection of the 54 changesets in the test data where *comment_len* assumed its maximum value, 87% of these were reverted due to spamming OSM with commercial for their business in the comment tag. In addition, editing applications vary in their difficulty to navigate, causing some levels of *editor_app* to correlate with user experience as can be seen by table 6.7, providing a potential explanation for *josm* and *other* having different distributions for the classes.

Table 6.7: Correlations between *active_weeks* and the values in *editor_app*. *active_weeks* is seen as a heuristic for general user experience.

other	*josm*	*potlatch*	*streetcomplete*	*go map!!*	*vespucci*	*osm go!*
-0.227	0.243	-0.030	0.014	0.020	0.024	0.008

In general, table 6.4 show that omitting the Miscellaneous category does not impact the

models much, causing a drop of about 1 percentage point of accuracy for the model trained on all features except for Reduced Ovid in which the impact is negligible. However, intuitively these features should not contain strong signals and as such these results are not unexpected. The results also show that while SHAP values are valuable in explaining a model, they may not always give the entire picture, as seen by omitting ***editor_app*** having a small effect on the models despite them being very important in Reduced Ovid and the MLP's SHAP values. It also shows that reduced model analyses complement the SHAP values well. In conclusion though, these features may be more important in vandalism detection which do not transfer well into a general anomaly detection paradigm.

Chapter 7
Suggestions for Model Improvements

7.1 Element Data

Out of the models in this project, only Reduced Ovid actually uses the element data while also performing worst overall out of all models, making improvements to the element data and incorporating it into the other models a natural next step.

There are two main components to incorporating the element data in a successful way. First one would need to handle the issue of inconsistent dimensionality stemming from changesets in general containing different numbers of edits, which in Reduced Ovid is solved by using multi-head attention. Secondly, one would need to have a representation of the edits in such a way that important information is retained.

7.1.1 Element Data Features

7.1.1.1 Tags

Briefly, a tag is represented as a key-value pair, and a valid tag is a key-value pair that follows OSM convention. Manual inspection of some reverted changesets shows that it is not uncommon to revert a changeset for tagging errors. Common occurrences is using a tag in a way that it is not intended for, like giving general information in the ***name***-tag or giving an incorrect value to tag according to OSM convention.

A lot of information is lost when using simple counting mechanisms for the number of tags in OSM elements as is done in the element data in this project. To investigate potential improvements to this, the tag-keys for all elements touched by the changesets in this project's data set are counted. Both the current and previous element versions are counted, corresponding to the version before and after the changeset was applied. This is done to give an idea of the content of the edits. Table 7.1 below presents the 10 most frequent keys as decided by the *current* element. A caveat however is that the Current and Previous columns

are not inherently connected - this is simply counting the occurrences of keys in the current and previous versions of elements over the entire set of elements subject to edits in *any* of the changesets. It is in theory possible that the set of elements containing these tags in the Current and Previous columns are entirely disjunct.

Table 7.1: The 10 most common tag-keys in the data set as decided by the Current column, split by label. This corresponds to the element version *after* the changeset was applied, which in the reverted data is the version that was then reverted. Previous refers to the element version before the changeset was applied.

Reverted data			Non-reverted data		
Key	Current	Previous	Key	Current	Previous
name	438062	450747	highway	71485	74613
highway	352893	392102	name	52895	45107
crop	238407	130	source	42986	47657
source	150741	274074	building	41406	50697
addr:housenumber	150005	170307	addr:street	19817	14794
addr:street	148329	162403	surface	19406	15836
addr:city	143322	143097	addr:housenumber	19064	16143
building	139050	183740	addr:city	15381	11656
addr:postcode	127788	139444	addr:postcode	14395	12164
addr:country	108624	68325	oneway	13456	13403

The reverted data contains significantly more tags than the non-reverted data - about 5.5 million vs. 600 thousand, or around 9 times more. A reason for this can simply be that elements with more tags have more activity, causing errors to be identified and reverted to a greater degree than for elements containing fewer tags. Furthermore, the key `crop` has many occurrences in the reverted data - this is probably due to a small number of OSM users incorrectly adding this tag to *many* elements and is discarded as noise. Additionally, the `source`- and `addr:country`-tags have considerable differences as compared to the previous versions, which is a difference not present in the non-reverted data. (`addr:country` is not among the top 10 in the non-reverted data, and is instead at place number 14 with `Current = 7800` and `Previous = 6166`.)

There could potentially be that some tags are significantly more common than others to be used in an incorrect way, like `name` or `addr:country`. Some may also be suspicious to remove, such as `source`. One possible way to improve the tagging information is include features identifying suspicious use of the most common tags, like a character count for the `name`-tag or whether a `source`-tag was removed in the edit.

A more ambitious method is to have a one-hot encoding for every possible valid tag (and perhaps a final category representing a non-valid tag), and iterate over every past version of the element to find all occurrences of all valid tags. This could provide tagging context based on past element versions which could identify tagging anomalies.

7.1.1.2 Geographical Features

For nodes, the distance moved in the edit could be added as a feature. As nodes, ways and relations can be part of larger elements, parent element status could also be added as two features holding boolean values. Additionally, the size of the parent elements could also be added. This may give information of suspicious behaviour of performing dramatic edits on a part of an established element, which could be suspicious. A more challenging method would be to add features based on validating elements part of a parent against each other.

7.1.2 Inconsistent Dimensions

In changesets, the edits within a changeset are often similar, especially if performing many edits in a single changeset. An easy way to reduce the number of rows in the element data is to introduce a counting mechanism for repeating rows. Keeping only the unique row in the element data for a changeset, along with a counter giving the number of occurrences of this row in the changeset reduces the number of rows without any loss of information. Figure 7.1 shows that this simple addition drastically reduces the number of rows - the left shows a histogram of the total number of edits for each changeset, and the right a histogram of the total number of unique rows in the element data for each changeset.

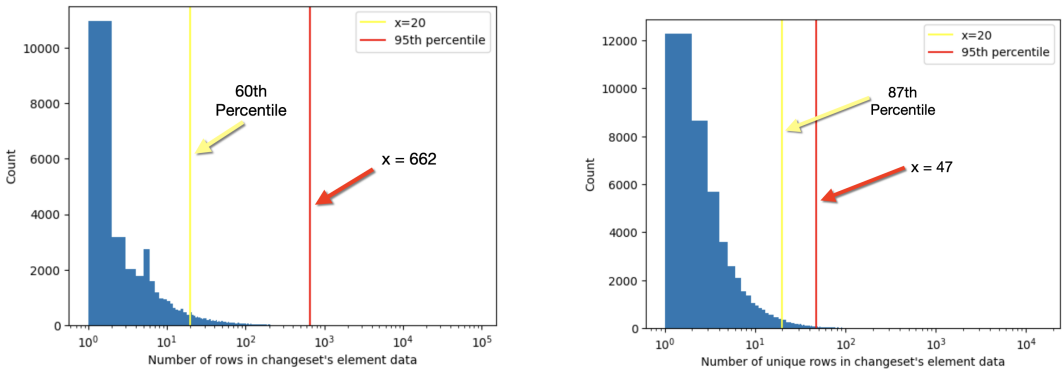


Figure 7.1: Histogram over the number of rows (left) and number of unique rows (right) for the data set in this project. The yellow line at $x = 20$ represents the maximum number of edited elements before Reduced Ovid omits the element data, while the red line represents the $95th$ percentile.

As can be seen by $x = 20$ corresponding to the 87th percentile rather than the 60th when using the counting mechanism, a reasonably low threshold like Reduced Ovid's 20 results in considerable more changesets not omitting the element data. Furthermore, if the threshold is instead used as a cutoff and the rows are sorted by descending counts, the most common edits are guaranteed to be used, giving a decent representation of the edits for most changesets even if the number of unique rows exceeds the cutoff value.

Another way to create consistent dimensionality is to use embeddings for the element data, but investigating this is however beyond the scope of this project.

7.2 Three classes

A fundamental flaw of using binary classification in this case is that the models do not distinguish between ”the changeset is believed to contain at least one error” and ”the changeset is believed to contain only errors”. However, when using this kind of model in practice, this is of course relevant information - given that a user trusts the model, a changeset that is predicted to *only* contain errors can be instantly reverted, perhaps even automatically, whereas a changeset where only a subset of the edits are suspicious require further inspection to determine which edits that ought to be reverted. Additionally, some methods to find the potentially erroneous edits within a changeset are only natural to use if one assumes that only a subset of the edits are incorrect, like outlier detection. Hence, a natural extension of the model is to introduce three classes instead of two that would represent whether a changeset contains only errors, some errors or no errors, and in turn should be fully, partially or not reverted, respectively.

To test the performance of introducing these three classes, the reverting changeset comments were scanned for the keyword ”part”, resulting in 2309 hits - considerably smaller than any other class, which is discussed in section 7.2.1 below. The keyword ”part” is used instead of ”partial” as it includes cases of ”partly”, ”parts” and ”partiel” in french, for example. The reverted changesets are now classified as partially reverted, introducing these three classes. Conversely, the remaining reverted changesets are labeled as full reverts.

Table 7.2: Number of samples for the three different classes.

Category	Number of samples	Fraction of dataset
Not reverted	26875	51.9%
Fully reverted	22974	44.3%
Partially reverted	1976	3.8%

Table 7.3 below presents the F1-scores for each class for all models.

Table 7.3: F1-scores for each class when using three classes. Bolded values are the best performing model for each class.

	MLP	Random forest	Boosted trees	Reduced Ovid
Not reverted	0.8202	0.8854	**0.8932**	0.7627
Fully reverted	0.7894	0.8489	**0.8586**	0.8185
Partially reverted	0.3688	0.3423	**0.4314**	0.0000

As can be seen, the F1-scores for the partially reverted class are bad for all models. With such an unbalanced dataset however, this is not unreasonable. If a large majority of the fully and not reverted samples are discarded to balance the classes to 1976 samples each, the retrained models on the balanced data set obtain the following F1-scores:

Table 7.4: F1-scores for each class when using three classes with a balanced dataset of 1976 samples per class. Bolded values are the best performing model for each class.

	MLP	Random forest	Boosted trees	Reduced Ovid
Not reverted	0.6964	0.7687	**0.8005**	0.6160
Fully reverted	0.6191	0.6391	**0.6828**	0.3133
Partially reverted	0.5973	0.7126	**0.7195**	0.5765

While the results are not great, the F1-scores are both fairly balanced and decently high considering the small sample size. The models can more easily distinguish between non-reverted samples and reverted samples, indicating that the average partial revert is closer to a full revert than a non-revert. Boosted trees perform best in all three classes, and in general, the tree models do a significantly better job in the non-reverted and partially reverted classes. Given more data, in particular in the partially reverted class, this extension to three classes could be promising.

7.2.1 Data Set Evaluation

Table 7.2 shows that the data set is heavily biased toward full rather than partial reverts. As the method for gathering data in this project is based on the method for vandalism detection in OSM changesets used by Tempelmeier and Demidova (2022), it is relevant to discuss how the method may be biased towards creating a data set similar to the vandalism data set. The method to find reverted changesets places a bias toward finding changesets that were *fully* reverted, as reverting changeset comments containing any of `"node"`, `"way"`, `"relation"` or `"part"`, all of which suggesting a partial revert, only constitute 13.4% of the reverting changesets found. Two examples would be *"reverting a way in changeset 12345678"* or *"partial revert of changeset 23456789"*. In vandalism detection, it is natural to use a method biased toward finding fully reverted changesets, as many cases of vandalism will be intentional, motivating a full revert. However in general anomaly detection, it is reasonable to believe that there is more balance between changesets that should be fully reverted and those that only require a partial revert. For example, edits based on incorrect information such as outdated satellite imagery or lack of local knowledge would motivate a full revert, while an incorrect tag on an otherwise correctly modified element would motivate a partial revert. The data set in this project probably has considerably more changesets that should be fully reverted as opposed partially reverted than the true distribution of OSM changesets, where these two classes are most likely more balanced or perhaps even skewed toward partial reverts.

7.2.2 Method to Find More Reverted Changesets

A way to find more changesets that were reverted either fully or in part would be to perform a tweaked and slightly more complicated data gathering technique as compared to the method in 4.1. First, the entire changeset history is scanned for comments containing the keyword "revert" and an ID, however the ID does *not* refer to a changeset, but instead a node, way or relation. This way one finds reverted elements rather than changesets. Knowing the ID of the reverting changeset, one could scan the history of the reverted element and find the element

version previous to the reverting changeset's edit, thus finding the version being reverted. This element version has a changeset ID corresponding to the changeset making the reverted edit. Finally, iterating over that changeset's edits to find if other edits were reverted or not by checking if previous and succeeding element versions are identical provides the information needed to determine if this was a full or partial revert.

Chapter 8
Discussion

8.1 Vandalism Detection Methods for General Anomaly Detection

The best performing model according to every metric is the boosted trees, achieving an accuracy of 0.8946 with balanced F1-scores, which is high. This indicates that the features optimized for vandalism detection translate well to the task of general anomaly detection. The best point of comparison is Reduced Ovid's anomaly detection performance in this project against the original Ovid's performance on the vandalism data set as presented by Tempelmeier and Demidova (2022) which have accuracies of 0.8047 and 0.8137, respectively - a small difference. This suggests that vandalism detection work about equally as well on both vandalism and non-vandalism data. However, these results may be somewhat biased to work better on this particular data set as the methods used to collect the data places a bias on finding full reverts rather than partial reverts, as discussed in section 7.2.1.

Furthermore, this bias may also cause the features optimized for vandalism detection to perform better than they would on a general data set. The task of finding changesets that should be fully reverted is considerably closer to vandalism detection than the task of finding changesets that only should be partially reverted. This may be a reason for the models' impressive performance. If the models were instead trained on the true distribution of changesets partially *vs.* fully containing errors, the performance using these features optimized for vandalism detection could be worse.

The introduction of three classes in section 7.2 experimented with changing the class balances, where a balanced data set did however show that the best model (boosted trees) was reasonably effective at differentiating fully, partially and not reverted samples as well. The F1-scores are however quite low, but this may simply stem from the small sample size. While it is not possible to make a clear judgement regarding the effectiveness of these features in a general distribution or the more nuanced task of using three classes before more data is used, the results in this project does however indicate that the features optimized for vandalism

detection used by Tempelmeier and Demidova (2022) are also effective in general anomaly detection.

8.2 Model Comparison

The performance metrics in section 5.3 clearly show that the boosted trees performs best, followed by random forest, the MLP and finally Reduced Ovid. The tree models in general perform better than the ANN models with a significant margin, and this pattern in general holds when using three classes rather than two. It should also be noted that Reduced Ovid performs worst despite being the only model to utilize the element data, although using hyperparameters tuned for the vandalism data set may have affected this.

One explanation for the tree models outperforming the ANN models could be the size of the data set - random forests seem to give better predictions given a relatively small data set, whereas neural networks in general needs larger amounts of data to achieve the same level of prediction accuracy. However, the neural network models generally continue to improve as the size of the data set increases, giving it a larger potential given that the data set can be expanded (Roßbach 2018). It is not unreasonable that this dynamic could extend from random forests to boosted trees as well, as the results support this. Another reason for the tree models performing best could be that the data set is tabular, meaning it is well structured and not "chaotic" like images or natural language. Grinsztajn, Oyallon, and Varoquaux (2022) have shown that over a number of medium sized data sets (defined by containing about 10 thousand samples), tree-based models perform best at tabular data, even without accounting for hyperparameter tuning being more computationally expensive in ANNs than trees.

Despite this, it should be noted that finding the optimal set of hyperparameters for neural networks is more demanding than for the tree-based models. As there was limited time for hyperparameter tuning, the tree models could have a more optimal set of hyperparameters than the ANN models, as they were not tuned in two steps like the MLP, and are tuned for this specific data set rather than vandalism data set, like Reduced Ovid. Providing more time for hyperparameter tuning may benefit the ANN models more than the tree models, which could narrow the performance gap between them.

With more data though, ANN models (probably more complex models than the simple MLP presented in this project, and perhaps different/more complex than Reduced Ovid as well) could perhaps perform better than the tree models. This may also provide an explanation for Reduced Ovid performing poorly in relation to the other, simpler models presented in this project. If neural networks require more data than tree models to achieve the same performance, more complex models like Ovid probably require *even more* data. This could also explain the MLP outperforming Reduced Ovid, as less complex architectures probably require less data to become "decent". Reduced Ovid's complexity may in this case hinder it, but given an order of magnitude of more data, this complexity may instead be to its benefit rather than its detriment.

8.3 Feature Importance

Section 6 provides a substantial analysis and some discussion of the changeset and user data following the surprisingly good performance on just these two sets of features. Below is only a brief discussion on the findings and machine learning in this task in general.

The models predict better on only the changeset data features rather than the user data. This provided an interesting result as before starting this analysis, we expected the opposite - if a human was given the task of predicting if a changeset was reverted or not without access to its' individual edits, we imagined that the easiest and intuitively strongest predictor to use would be overall user experience, and although this is effective, the changeset data proved to be more effective. While section 6.3 sheds some light as to why this may be, this highlights the incredible potential of machine learning in this case. In the changeset data, there are signals which are hard for humans to see, but which nonetheless are good predictors. Considering the wide array of potential editing errors that can occur in OSM, using machine learning to recognize complex patterns that humans struggle to see or understand can not only be effective in itself, but also complement manually determined validation heuristics very well as these, of course, are reserved for patterns that humans can understand and implement. In this way, machine learning models have a natural spot alongside heuristics in creating a more secure system.

8.4 Thoughts on Generalization

Throughout the report, there has been discussions on the generalization of these models and potential issues with regards to this. Section 6.2 shows that the models seem to be very aggressive in classifying beginners' changesets as potentially erroneous and vice versa, which could cause generalization issues but could hopefully be mediated by the element data, if introduced effectively. The balance between classes is discussed in section 7.2 and how it most likely does not represent the true distribution of OSM changesets, however this could potentially be solved by tuning the classification probability thresholds. Third, sections 7.2.1 and 6.3.2 discuss how the data gathering method seems to skew the reverted data to something similar to a vandalism data set as well as placing a bias toward areas with active OSM communities. To make a judgement to what extent this affects the models, one may need a less biased data set, perhaps even using a fundamentally different data gathering technique to avoid the issues of errors having to be found by another user. A way to circumvent these problems is to restrict usage of these models to areas that already have a large OSM activity.

While the models are not perfect and more work is required to get a clear picture of their generalized performance, they do not have to be perfect to provide value. No set of heuristics or any extent of manual inspection are going to be perfect, but are rather at some point deemed "good enough" to be used, and this perspective is relevant in this case as well. Having a model act as an initial filter could provide value given that there is a process to handle the flagged changesets. In fact, even if there is not, these models could be initially deployed to flag changesets but without any action taken, instead collecting data on the flagged/not flagged samples which can be analyzed further to give an idea of its generalization performance and the scale of changesets being flagged, despite them not being labeled.

8.5 Conclusions

The main conclusions of this project are:

1. The vandalism detection methods as presented by Tempelmeier and Demidova (2022) translate well to general anomaly detection. There may however be some bias toward the method working better on this particular data set, as the data gathering algorithm is biased toward finding changesets that were fully reverted, making the task more closely resemble vandalism detection.

2. Among the models tested, the boosted trees performs best with regards to every metric and achieves an accuracy of 89.46%. The tree models perform significantly better than the neural network models at this task, although this may change with a larger data set.

3. Both the changeset and user data are important and give considerable contributions to the models. A changeset from a beginner is more likely to be flagged as potentially erroneous than others, and the converse for experienced users. Location of the edits plays a large role in predictions, however this is most likely due to the data gathering method placing a bias on the reverted data to areas with active OSM communities. Information about the edits in the changeset as a whole is important in detecting outliers. Good performance was achieved while not utilizing the element data due to inconsistent dimensionality and Reduced Ovid performed worst despite accessing the element data, suggesting it could be improved.

8.6 Future Work

8.6.1 Incorporating Element Data

A large potential for improvement is to incorporate the element data effectively. Improving tagging information and removing redundancy is discussed in section 7.1.1, however these two suggested methods are mutually exclusive. Increasing the complexity in edit descriptions lowers the probability that two edits would have identical representations. A possible middle ground is to use some kind of feature engineering, adding a small number of important features. Embeddings could also be explored, as well as validating individual edits rather than entire changesets. Adding a large number of features could however cause overtraining issues, which ties this to the next point...

8.6.2 Increasing the Data Set Size

Creating a larger data set shows the models more types of errors, which becomes increasingly important if the element data is used. Additionally, a larger data set can set aside more test data to test the models' performance on the true distribution of reverted/not reverted changesets, which is most likely far from 50/50. The method in section 7.2.2 could be used, for example.

8.6.3 Three Classes

Gathering more data and expanding the models to classify changesets as only, partially or not containing errors will probably make them more useful, if successful. Section 7.2 shows that with balanced data sets, the F1-scores are also reasonably balanced. Furthermore, additional data in the partially reverted class would better show the models' abilities to distinguish between similar changesets.

8.6.4 Applicability

While many improvements and ways to build upon the results in this project are possible, this project aimed to investigate the potential in machine learning methods to flag potentially erroneous changesets for further inspection. Boosted trees very good performance suggests that without much further work it could be valuable as a filter in edit validation in OpenStreetMap.

References

Ahmed, Mohiuddin, Raihan Seraj, and Syed Mohammed Shamsul Islam (2020). "The k-means Algorithm: A Comprehensive Survey and Performance Evaluation". In: *Electronics* 9.8. ISSN: 2079-9292. DOI: 10.3390/electronics9081295. URL: https://www.mdpi.com/2079-9292/9/8/1295.

Almog, Uri (2022). *Decision Trees, Explained.* [Accessed 22/5-2023]. URL: https://towardsdatascience.com/decision-trees-explained-d7678c43a59e.

Bùi, Bào (2023). *How Boosted Trees Inference Works.* [Accessed 4/7-2023]. URL: https://medium.com/@bobi_29852/how-boosted-trees-inference-works-f161b03d5f5b.

Dorogush A., V., V. Ershov, and A. Gulin (2017). "CatBoost: gradient boosting with categorical features support". In: *Conference on Neural Information Processing Systems*.

Grinsztajn, Léo, Edouard Oyallon, and Gaël Varoquaux (2022). *Why do tree-based models still outperform deep learning on tabular data?* arXiv: 2207.08815 [cs.LG].

Hoffmann, Sarah (2015). *pyosmium.* [Accessed 18/5-2023]. URL: https://github.com/osmcode/pyosmium.

LeCun, Yann A. et al. (2012). "Efficient BackProp". In: *Neural Networks: Tricks of the Trade: Second Edition.* Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 9–48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3.

Lucchetti, Roberto (2023). *Game Theory: An Introduction*, pp. 102, 117–121.

Lundberg, Scott M. and Su-In Lee (2017). "A unified approach to interpreting model predictions". In: *CoRR* abs/1705.07874. arXiv: 1705.07874. URL: http://arxiv.org/abs/1705.07874.

Masui, Tomonori (2022). *All You Need to Know about Gradient Boosting Algorithm Part 2. Classification.* [Accessed 23/5-2023]. URL: https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-2-classification-d3ed8f56541e.

Ohlsson, Mattias and Patrik Edén (2021a). "Introduction to Artificial Neural Networks and Deep Learning". In: Lund University. Chap. 1 - Introduction, pp. 2–8.

— (2021b). "Introduction to Artificial Neural Networks and Deep Learning". In: Lund University. Chap. 2 - Feed-forward Neural Networks, pp. 9–46.

OpenStreetMap Wiki, API v0.6 (2023). [Accessed 18/5-2023]. URL: https://wiki.openstreetmap.org/wiki/API_v0.6.

OpenStreetMap Wiki, Changeset (2023). [Accessed 17/5-2023]. URL: https://wiki.openstreetmap.org/wiki/Changeset.

OpenStreetMap Wiki, Map features (2023). [Accessed 18/5-2023]. URL: https://wiki.openstreetmap.org/wiki/Map_features.

OpenStreetMap Wiki, Tags (2022). [Accessed 18/5-2023]. URL: https://wiki.openstreetmap.org/wiki/Tags.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Probst, Philipp and Anne-Laure Boulesteix (2017). *To tune or not to tune the number of trees in random forest?* arXiv: 1705.05654 [stat.ML].

Qiang, Wang and Zhan Zhongli (2011). "Reinforcement learning model, algorithms and its application". In: *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, pp. 1143–1146. DOI: 10.1109/MEC.2011.6025669.

Rozemberczki, Benedek et al. (2022). *The Shapley Value in Machine Learning*. arXiv: 2202.05594 [cs.LG].

Roßbach, Peter (2018). *Neural Networks vs. Random Forests – Does it always have to be Deep Learning?* [Accessed 9/6-2023]. URL: https://blog.frankfurt-school.de/wp-content/uploads/2018/10/Neural-Networks-vs-Random-Forests.pdf.

sklearn.ensemble.RandomForestClassifier (2023). [Accessed 1/6-2023]. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.

Tempelmeier, Nicolas and Elena Demidova (2021). *Ovid Repository*. [Accessed 2/6-2023]. URL: https://github.com/NicolasTe/Ovid.

— (2022). "Attention-Based Vandalism Detection in OpenStreetMap". In: *Proceedings of the ACM Web Conference 2022*. WWW '22. Virtual Event, Lyon, France: Association for Computing Machinery, 643–651. ISBN: 9781450390965. DOI: 10.1145/3485447.3512224. URL: https://doi.org/10.1145/3485447.3512224.

Topf, Jochen (2013). *Osmium Command Line Tool*. [Accessed 18/5-2023]. URL: https://github.com/osmcode/osmium-tool.

Vaswani, Ashish et al. (2017). "Attention Is All You Need". In: *CoRR* abs/1706.03762. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

Yi, Dokkyun, Jaehyun Ahn, and Sangmin Ji (2020). "An Effective Optimization Method for Machine Learning Based on ADAM". In: *Applied Sciences* 10.3. ISSN: 2076-3417. DOI: 10.3390/app10031073. URL: https://www.mdpi.com/2076-3417/10/3/1073.

Yiu, Tony (2019). *Understanding Random Forest*. [Accessed 23/5-2023]. URL: https://towardsdatascience.com/understanding-random-forest-58381e0602d2.

Appendices

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-08-21

EXAMENSARBETE Detecting Anomalies in OpenStreetMap Changesets using Machine Learning
STUDENT Dan Svenonius
HANDLEDARE Patrik Edén (LU), Hampus Londögård (AFRY AB)
EXAMINATOR Jacek Malec (LTH)

Fungerar maskininlärning för att hitta fel i stora mängder geografisk data?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Dan Svenonius**

OpenStreetMap är som Wikipedia för kartor, där vem som helst kan bidra. Eftersom databasen är så stor är det svårt att ha en bra kvalitetskontroll, vilket gör validering av bidrag med maskininlärning intressant, och det visar sig att det funkar ganska bra.

Tänk er att ni får i uppgift att skapa ett system för kvalitetskontroll av bidrag till hela Wikipedia. Svårt? Det är vad vi försökt göra i det här examensarbetet, fast med OpenStreetMap (OSM) och geografisk data. Ett bidrag är en samling individuella redigeringar, som att ändra ett adressnamn eller lägga till ett hus. När det är så många sorters fel som kan uppstå blir det orimligt att skapa ett antal regler, exempelvis att ett hus inte kan ligga i havet, för att göra denna kontroll. Då blir det naturligt att istället testa om maskininlärning kan fungera, då det hade kunnat flagga misstänksamma bidrag som får genomgå en mer ordentlig validering.

Här behöver vi dock veta om ett bidrag var bra eller dåligt, så vi är till synes är tillbaka på ruta ett. Istället hittar vi bidrag som av en annan användare har dragits tillbaka (utan att vi vet varför), och säger att dessa borde innehålla något konstigt. För de bra bidragen så drar vi slumpmässiga bidrag och antar att en stor majoritet av bidragen är bra.

Vi testar detta i fyra olika maskininlärningsmodeller (två baserade på neurala nätverk och två på beslutsträd) och får oväntat bra resultat - den bästa modellen har rätt på 9 av 10 bidrag i genomsnitt. Det visar sig även att modellerna baserade på beslutsträd med råge fungerar bäst.

Något väldigt intressant är att modellerna fungerar så bra *trots* att de inte har tillgång till information om specifika redigeringar, utan snarare om bidraget som helhet och dessutom användarens erfarenhet. Man hade kunnat tänka sig att den går mest på användarerfarenhet, men det är relativt balanserat mellan dessa två kategorier.

Detta visar på den otroliga potentialen i maskininlärning - att algoritmen hittar regler och mönster snarare än vi - både i denna uppgift, men egentligen överallt där stora datamängder finns.

Även LTH:s Sjön Sjøn finns med i OpenStreetMap, inlagd här som ”Lake Lake”.