Improving the OpenStreetMap Data Set using Deep Learning

Hampus Londögård
dat13hlo@student.lu.se

Hannah Lindblad
elt13hli@student.lu.se

June 19, 2018

Master's thesis work carried out at ÅF Digital Solutions.

Supervisors: Pierre Nugues, Pierre.Nugues@cs.lth.se
Thomas Hermansson, thomas.hermansson@afconsult.com

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

Abstract

OpenStreetMap is an open source of geographical data where contributors can change, add, or remove data. Since anyone can contribute, the data set is prone to contain data of varying quality. In this work, we focus on three approaches for correcting WAY component name tags in the data set: Correcting misspellings, flagging anomalies, and generating suggestions for missing names.

Today, spell correction systems have achieved a high correction accuracy. However, the use of a language context is an important factor to the success of these systems.

We present a way for performing spell correction without context through the use of a deep neural network. The structure of the network also makes it possible to adapt it to a different language by changing the training resources. The implementation achieves an F_1 score of 0.86 (ACR 0.69) for WAY names in Denmark.

Keywords: MSc, Machine Learning, OpenStreetMap, Random Forest, Neural Network, Sequence-2-Sequence

Acknowledgements

We would like to thank Pierre Nugues for the feedback he has given us on this thesis.

We would also like to thank Marcus Klang for his invaluable input on neural networks and Daniel Palmqvist for his feedback and knowledge of algorithms and geographical information systems.

Last but not least, we would like to thank ÅF and Thomas Hermansson for giving us the opportunity to work on this thesis.

Contents

Chapter 1

Introduction

In this chapter, we introduce the motivations for this thesis. The first two sections present the background and purpose. We then continue to cover the research questions, related work, outline and work distribution.

1.1 Background

OpenStreetMap (OSM) is an initiative to create a free, editable map of the world. The project maintains an open data set of geographical data which can be used for any purpose as long as OpenStreetMap is credited. Anyone can become a contributor and add data to the project.

The OSM data set consists of three basic components: Nodes, Ways, and Relations. An example of a component is the Way component which is a list of between 2 and 2000 nodes. Way components are used to represent linear features like rivers and roads (OpenStreetMap, 2017).

The components can also have tags attached to them. A tag consists of key and value fields in free text format. The tags are used to describe the elements they are attached to. An example of a tag is the name tag which is used to convey the name of a component. The city of Copenhagen is for instance a node with the key-value name tag *name=København*.

Conventions for the meaning and use of tags have been agreed upon by the OSM community. These are however not enforced by any particular entity and thus the data contained in the OSM data set is of varying quality. In the name tag case, there are occurrences of misspelled, missing, and incorrect names. For Way components, the names should match the street name depicted on the street sign for the road in question (OpenStreetMap, 2018).

1.2 Purpose

In this thesis, we focus on name tags for Way components in the OSM data set of Denmark. Our goal is to predict if the instances are correct or incorrect given the name tag, potentially in combination with other tags like maximum speed, highway type, and surface. If possible, suggestions should also be generated in combination with the prediction of an incorrect instance. The suggestions could then be used for manual correction of the instance.

With the resulting algorithm, we aim to assist the ongoing work of improving the quality of the data contained in the OSM data set. We intend to integrate the end result into the open source tool Atlas Checks (Apple Inc, 2018), which tests Atlas file data integrity by traversing the OSM data set and applying different kinds of checks. An example of a check that is performed today is a test that finds duplicate Nodes. By integrating the work into Atlas Checks, the data set can be monitored for erroneous data without the need for manually searching the data.

1.3 Problem Formulation

We aim to improve the quality of the OSM data set by increasing the number of accurate Way components and hence smooth the data set. This should be achieved by flagging possibly incorrect instances for manual correction. By requiring manual correction instead of performing changes automatically, we lower the expectations on the system. The proposed solution should answer the following questions:

1. How to identify misspelled names in Way components?
2. How to generate suggestions for a misspelled name?
3. How to generate suggestions for Way components with missing names?
4. Is it possible to find anomalies in names in combination with other tags like maximal speed?

Short motivations for the different research questions are listed below.

- A quality issue in the data is the occurrence of misspelled names. In order to correct these instances, they first need to be identified.
- When a misspelled instance has been identified, a suggestion for a correction should also be proposed by the system to simplify the correction process.
- Another issue is missing names for Way instances. A Way is considered to have a missing name if neither the name tag nor the noname tag is present. These instances should be flagged and suggestions provided algorithmically.
- The last step in our attempt to smooth the data set is by identifying anomalies in the data. One relation could be the Danish word for street (“gade”) and the maximum speed limit of 50 km/h or less. If a Way name’s suffix is “gade”, it’s 25 times more probable to have a max speed limit of max 50 km/h in comparison to a Way with the suffix “vej”. By taking advantage of relations like these, anomalies could possibly be identified and flagged.

1.3.1 Challenges

Given related work and research, it is not reasonable to expect ideal performance from a spell corrector without annotated data nor context. Whitelaw et al. (2009) showed the possibility of creating a spell corrector without annotated data by using the frequency of words on the web. However, they used the context of a sentence to improve their performance. In our case of Way name tags, it is often not possible to find the name on the Web, except for in other map services such as Google Maps. This is especially true in countries like Denmark. Way names in Denmark are often compounds of two or more words which means that the name is often a non-existing word. In addition, Way names always lack the context of surrounding words, as opposed to spell correction of texts, which adds another layer of complexity to the problem.

1.3.2 Constraints

This thesis focuses on Way component names in the data set that are present. The cases where the name tag is missing has only been considered briefly.

The algorithm uses solely the OSM data set. There wasn't enough time to investigate conflation with other data sets. In addition, conflation would be difficult to extend to cover other countries besides Denmark.

1.4 Related Work

Whitelaw et al. (2009) designed and implemented an end-to-end system that included spell checking and auto correction without requiring manually annotated training data. In this proceeding, the system is shown to perform better than candidate corrections based on hand-curated dictionaries – all without a manually annotated corpus. The World Wide Web is used as a large noisy corpus from which they infer knowledge about misspellings. Their work is differentiated from ours by the fact that that they make use of the surrounding language context.

An example of why the context is important is the misspelling "goint". 'Goint" could refer to both "joint" or "going" depending on the context. The phrase

> Companies typically pursue **joint** ventures for one of four reasons.

is more probable than

> Companies typically pursue **going** ventures for one of four reasons.

Whitelaw et al. (2009) introduce a method that adds artificial noise to data. The core concept is to perform an experiment where participants copy a Wikipedia article with the addition of not being allowed to edit what they have written. No backspace or arrow keys can be used by the participants. The authors then base the artificial noise that they introduce to their training data on the errors generated from the experiment. We used this approach to introduce artificial errors to our training sets.

Max and Wisniewski (2010) present a study of naturally occurring errors in Wikipedia. They created the Wikipedia Correction and Paraphrase Corpus (WiCoPaCo), which is

a freely available resource built by automatic mining of the Wikipedia revision history. To create this detailed corpus, Max and Wisniewski show how to infer when a word is different. This includes not only misspellings but also synonyms, like how the word ideal is a different word for perfect. They also arrange their findings in a visually pleasing way that maintains a great detail of the errors. This arrangement of information and analysis of data inspired us on how to analyze the OSM data and where to look for misspellings.

Karpathy (2015) showed that a sequence to sequence (seq-2-seq) neural network delivers a powerful performance in the case of generating texts with a special style such as C code or Shakespeare. In addition, Chollet (2017) describes how to use the seq-2-seq architecture to translate texts from English to French. More about seq-2-seq networks can be found in the Theory section on recurrent neural networks.

1.5 Outline

In this report, we begin by introducing the theory behind the implementations. We then explain the method we applied when approaching the problem, and tools used during this process. After the approach, the implementations and results are presented, followed by a discussion of the results and improvements that were tried during the thesis work. Finally, we conclude what knowledge was gained in this thesis and suggest possible future work.

We also include an appendix with Danish text that was generated by the system together with figures showing missing names suggestions by the missing names algorithm.

1.6 Work Distribution

The thesis work has been completed in full collaboration.

Chapter 2

Theory

In this section, we introduce the theoretical background that is needed to understand our work. We begin with a short introduction to machine learning followed by a presentation of the machine algorithms used in this thesis. We continue with sections on neural networks where we describe central theoretical concepts that we have applied to our work. The following sections present natural language processing concepts used and OpenStreetMap tags that were utilized to extract information from the OSM data set. Lastly, the evaluation metrics are introduced.

2.1 Introduction to Machine Learning

Machine learning is a way for a computer to learn – i.e. progressively improve performance on a specific task – from data instead of being explicitly programmed how to do so.

In the grand scheme, there are two different types of machine learning: supervised and unsupervised. In the supervised case, the computer is presented with example inputs and their desired outcome, also known as a label. In the unsupervised case, no labels are given to the computer, leaving it to find a pattern in the data and to label it itself.

Another way to divide machine learning systems into different types is by categorizing the desired output from the system. In this thesis, we have decided to use a supervised classifier. In supervised classifiers, outputs are divided into two or more classes and the learning system has to produce a model that assigns the input to one of these classes.

Below follows two sections on central concepts. The first section details central concepts such as under- and overfitting and early stopping. The second section describes how data can be bootstrapped.

2.1.1 Central Concepts

When training a machine learning system, there is always a danger of the system excessively tailoring itself to the data. There is also a risk of the system ending up being too general. These two cases are known as overfitting and underfitting.

Underfitting means that the system over generalizes the data. By doing this, the system will produce a high error rate during both the training and test phase.

Overfitting means that the system excessively tailors the parameters to the training set which in turn lowers the ability of the algorithm to generalize on new data. Due to this specificity, the error rate during the training phase will be low. However, during the test phase, when new data is presented, the error rate will be significantly increased. This is one of the reasons to split the available data into two separate test and training sets. If testing is performed on the same data as the training, there will be no indication when the system is overfitted. A visualization of these concepts is shown in Figure 2.1.

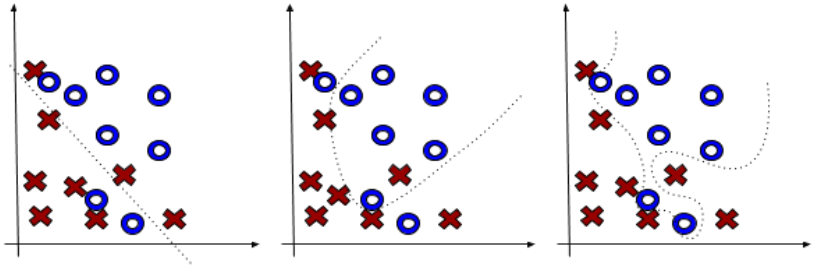


Figure 2.1: Underfitting and overfitting in machine learning. The leftmost figure shows a case of underfitting while the rightmost shows the case of overfitting. The middle figure shows a reasonable fit given the training data.

Early stopping is a form of regularization to combat overfitting when training with an iterative method, such as gradient descent (Patterson and Gibson, 2017). These types of methods update the learner to better fit the training data for each iteration. This indirectly improves the learner's performance on the test data set or any data outside of the training set. Past a point, however, further iterations come at the cost of reduced generalization and the algorithm becomes overfitted. See Figure 2.2.

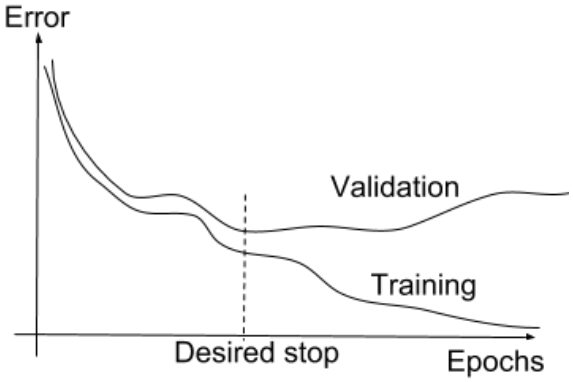


Figure 2.2: Visualization of when to perform an early stop as the validation error score starts to increase.

2.1.2 Bootstrapping Data

Bootstrapping data is a way of creating a balanced data set where the number of examples of each class is about the same. It is performed by drawing random samples from a data set in a systematic manner where each class is selected at the same probability. Bootstrapping does not only reduce the risk of overfitting the data during training but also improves the systems ability to solve edge cases.

2.2 Decision Trees

A decision tree is a non-parametric supervised learning approach used for classification and regression. The goal is to learn to classify the data based on rules that can be inferred from training on its features. The structure is flowchart-like and therefore easy to visualize and interpret. A visualization of a decision tree can be found in Figure 2.3, where each tree in the figure shows a decision tree.

2.3 Ensemble Learning

An ensemble is composed of a set of individually trained predictors. Maclin and Opitz (2011) show that ensemble learning models are often more accurate than any individual predictor in the ensemble. Two important ensemble methods are bagging and boosting. Bagging refers to models where independent predictors produce a prediction by an averaging process. Boosting methods instead focus on building a series of classifiers where each member is constructed based on the performance of the previous classifier in the series.

2.3.1 Random Forest

The Random Forest ensemble learning method is based on decision trees in the sense that the model consists of multiple decision trees that through a voting phase produce a final output. Hence, random forests are a bagging method. A simplified visualization of the

random forest structure is shown in Figure 2.3. Each subtree 1..N is a decision tree by itself. Because of this structure of using multiple decision trees and then having a voting phase, the random forest classifier reduces the behaviour of overfitting the data as decision trees tend to do (Breiman, 2001).

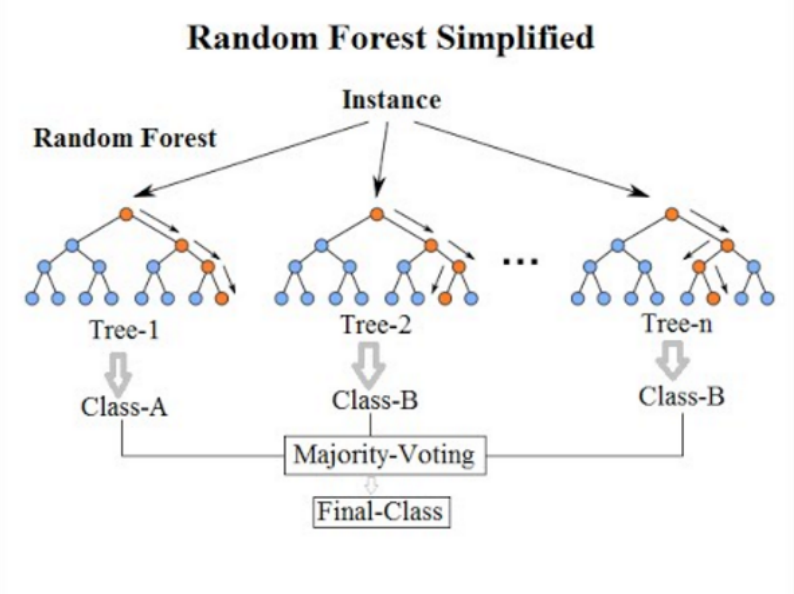


Figure 2.3: Random forest classifier simplified (Wikimedia Commons, 2017a)

2.3.2 XGBoost

In contrast to random forests, XGBoost uses the boosting ensemble technique. That means that the model of predictors (decision trees) is built sequentially instead of independently. The aim is that predictors created later in the learning process will learn from the previous predictors' mistakes. Maclin and Opitz (2011) mention that boosting methods aim to minimize the error in the bias term – how close the resulting model is to the target function – which may result in models not reproducible by the individual learning algorithms. That makes boosting methods ideal for cases where weaker learning algorithms are combined in one model.

More specifically, each tree x_i in the ensemble receives its input from the previous tree x_{i-1}. The sum of all outputs is the result. See Equation 2.1 where y_j is the output of tree x_j and N is the number of trees.

$$\sum_{j=0}^{N} y_j \tag{2.1}$$

2.4 Neural Networks

Neural networks is a subgroup of machine learning. The fundamental unit of a neural network is an artificial neuron which is loosely based on the biological neuron in the mammalian brain. Like the biological neuron with its synapses, the artificial neurons are connected to several other neurons. The neurons can be trained to pass along useful signals to its neighbors. The term deep neural networks refers to neural networks with more than two layers.

The activation function is the key to successful learning in machine learning. The artificial neuron, like the biological neuron, is stimulated by inputs. The connections in the network are made automatically as the network weights itself during training. The goal of the training phase is to let the artificial neurons know what signals to pass on so that they only pass along useful signals.

See Figure 2.4 on how the input is first transformed into a value which either will be sent forward or not depending on the result from the activation function.

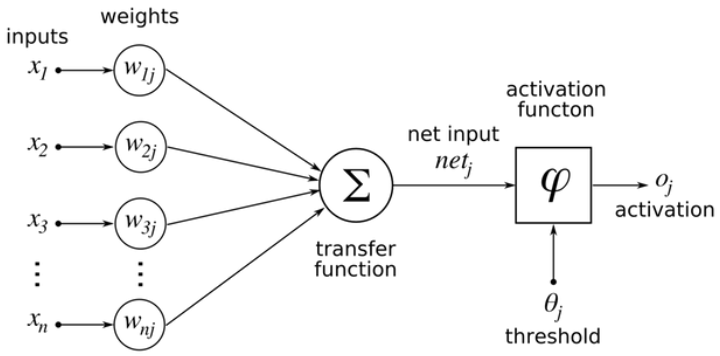


Figure 2.4: A diagram of a perceptron node where a perceptron is the most basic artificial neuron (Eclipse Deeplearning4j Development Team, 2018).

Feed forward neural networks (FFNN) are a class of artificial neural networks where the connections don't form cycles. Data is inputted to the network which then produces an output that is unconnected to earlier and later inputs. In some cases, such as with sequences of either images (video), text or audio, FFNNs do not perform well. To combat this, a new type of Neural Network, the **recurrent neural network** (RNN), was created. RNNs form cycles with the data (explained further in section 2.5 on RNNs) which means that they can handle an infinite input.

The following measurements are used during training of a neural network to change and improve the training process:

An **epoch** describes the number of times the network sees the entire data set. Each time the network has seen all examples, an epoch has passed.

In our case, the definition of an epoch is a bit redefined. An epoch is rather a subset of the complete data set. Because of this definition of an epoch, we can add more data on-the-fly and note how much it helps to improve our system. This also means that we can interactively test with new data – similar to K-folding but the knowledge of how much the increase of data improves the system is added.

A **batch** is the number of examples that are propagated through the network before a parameter (weight) update is performed. To optimize the system, it is important to have a suitable batch size. The smaller batch size, the longer time required for the system to converge to a local minima and the larger the less likely the minima is to be global. Running multiple epochs helps the system to learn more but if the number of epochs is too large the system will overfit the data.

An **iteration** describes the number of times that a batch of data has passed through the network. In the case of neural networks, this means the forward and backward pass. Every time you pass a batch of data through the network, an iteration has been completed.

2.4.1 Loss

The concept of loss is integral to machine learning. All types of machine learning algorithms have the primary goal of minimizing loss. Loss is calculated on the training and test set and is an interpretation of how well the model is doing for these two sets. It is a summation of the errors made for each example in the training or test set.

In the case of neural networks, the loss is usually negative log-likelihood for classification and the main objective in a learning model is to minimize the loss function's value with respect to the parameters of the model by changing the weight vector through different optimization methods, such as backpropagation.

The value of the loss function implies how well a certain model behaves after each iteration of optimization. Ideally, one would expect a reduction of loss after multiple iterations.

2.4.2 Gradient Descent

A gradient is defined as the generalization of the derivative of a function in one dimension to a function f in several dimensions. It is represented as a vector of n partial derivatives of the function f. To summarize, the gradient is important in optimization because the gradient points in the direction of the greatest rate of increase of the function. The magnitude of this is the slope of the graph in that direction. The **gradient descent** measures this slope, that is the change in error caused by a change in the weight, and takes the weight one step toward the bottom of the valley, see Figure 2.5. It does so by taking the derivative of the loss function to produce the gradient. As we want to minimize the loss we want to find the minimum of the loss function. Figure 2.5 shows a small visualization of this in a two dimensional space.

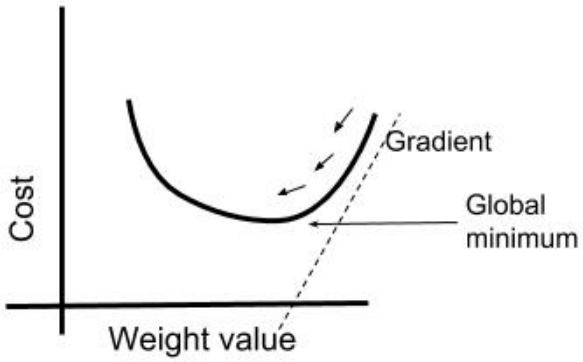


Figure 2.5: Gradient Descent

The **stochastic gradient descent** (SGD) is an implementation of gradient descent where the gradient is computed and then the parameter vector is updated after each training sample. Batches can also be used where a number of samples are processed before the update.

2.4.3 Hyperparameters

Hyperparameters can be tuned to make the network train more efficiently. Hyperparameter selection refers to the selection of the controlling optimization functions as well as the model selection during training with a chosen learning algorithm. The focus is on learning the structure of the data as quickly as possible, while making sure that the model neither underfits nor overfits to the training data.

The **learning rate** decides the scaling factor of the gradient which in turn will influence at which rate the parameters are adjusted during optimization. This adjustment of parameters is made in order to minimize the error in the network's predictions. The value scales the size of the steps (updates) a neural network takes to its parameter vector x as it crosses the loss function space.

A large learning rate (e.g. 1) will yield large leaps and a small learning rate (e.g. 0.00001) will result in slow progress. Large rates will save time initially but might become disastrous if they lead the system to overshoot the minimum. This follows from the fact that the algorithm will bounce back and forth on either side of the minimum and possibly escape the valley and move onto the next valley.

In contrast, a small learning rate should always lead to a minimum but can be incredibly slow. The time consuming part is not the only con of a small learning rate but also, the minimum found might be a local minimum rather than a global one.

The main purpose in training of machine learning algorithms is to control overfitting. This is done by helping the system with the effects of out-of-control parameters by using different methods to minimize parameter size over time.

Momentum helps the learning algorithm to not get stuck in a local minima in the search space. In other words, momentum does to the learning rate what the learning rate does to the weights. It gives a momentum to the learning rate based on current status and in the end this leads to a model of higher quality.

2.4.4 Optimization Algorithms and Updaters

DeepLearning4Java (DL4J), a framework for deep neural networks in Java, introduces two new terms – optimization algorithms and updaters – in addition to the previously mentioned hyperparameters. The optimization algorithm is the back propagation algorithm and the updater refers to the updater used during the back propagation when weights and learning rate are updated.

Optimization Algorithms

DL4J supports four different optimization algorithms out of the box. The first one is SGD, which was described earlier. DL4J also includes SGD with line search, conjugate gradient line search and LBFGS (Hinton and Salakhutdinov, 2006). DL4J notes that these three are more powerful than the vanilla SGD but much costlier during each update for the parameters due to the line search component (DL4J, 2018).

Updaters

There are five different updaters included in the DL4J package. The first and most basic one is **Stochastic gradient descent** (SGD). SGD only uses the learning rate.

The second updater is **Nesterov** which is a reference to Nesterov's momentum. Nesterov's momentum is a variant of the standard momentum. Standard momentum can be thought of as a ball that rolls down a hill, along the slope. Nesterov's momentum tries to make this ball smarter by giving it a notion of where it is going so it knows to slow down before the hill slopes up again. By knowing how the parameters will be updated, the approximate position of the parameters in the next iteration can be calculated.

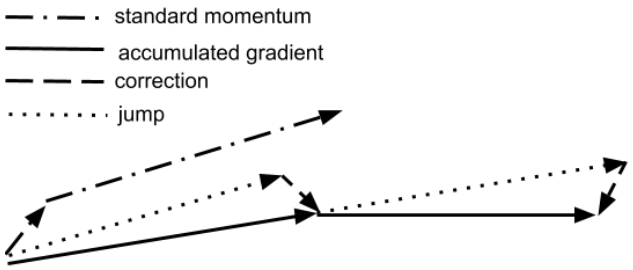
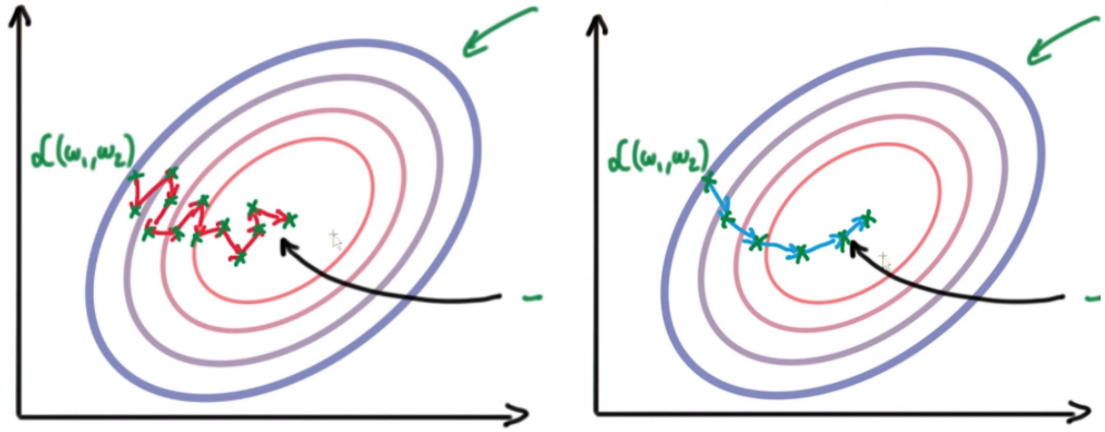


Figure 2.6: Image of Nesterov's Updater.

This is shown in Figure 2.6. Normally, momentum methods computes the gradient at the current location first and then takes a big leap in the direction of the updated accumulated gradient – like in Figure 2.6. The idea of Nesterov is to instead make a jump in the direction of the previous accumulated gradient and then measure the gradient where you end up and make a correction accordingly. The idea is that it's better to correct a mistake after you have made it. Figure 2.6 shows that the second jump (dotted line) does the same jump as the previous accumulated gradient and then corrects itself.

This approach of anticipatory update prevents the system from updating too fast and results in an increased responsiveness. According to Bengio et al. (2012), this significantly

increased performance of recurrent neural networks on a number of tasks. A simplified version of how the ball rolls down the slope can be seen in Figure 2.7.



(a) Momentum **(b)** Nesterovs Momentum

Figure 2.7: A comparison between Momentum and Nesterovs Momentum (Eclipse Deeplearning4j Development Team, 2018).

The third updater is **Adagrad**. Since Nesterov lets us adapt the update to the slope of the error function and speed up SGD, the idea is to adapt the update to each of the individual parameters in order to perform larger or smaller updates depending on their relative importance. In practice, this means that Adagrad updates less frequent parameters with a larger factor and more frequent parameters with a smaller weight. There exists a modification of Adagrad named Adadelta which seeks to reduce the aggressive and monotonical reduction of learning rate. Adadelta is also included as an updater in the DL4j package.

The fourth updater is **RMSProp**. Hinton (2018) proposes RMSProp as an adaptive learning rate method. It was developed independently from Adadelta and seeks to solve the same issues as Adadelta. To summarize, RMSProp is a more advanced version of Adadelta with the difference that RMSProp divides the learning rate by an exponentially decaying average of squared gradients.

The fifth and final updater is **Adaptive Moment Estimation** (Adam). Adam is another method to compute an adaptive learning rate. Adam expands upon Adadelta and RMSprop by also storing an exponentially decaying average of past gradients. Kingma and Ba (2014) show empirically that Adam works well in practice and compares favorably to other adaptive learning method algorithms.

2.5 Recurrent Neural Networks

Unlike FFNNs, a **recurrent neural network** (RNN) forms cycles with the data. It retrieves context out of a time sequence and adjusts its decision on the current sequence depending on earlier input and output as shown in Figure 2.8. Connections between units form a directed graph along a sequence. This allows the input to be a sequence, or rather to behave like a time sequence since the network can form a sense of context by knowing

what the input has been before. Therefore, an RNN can be implemented as a sequence to sequence (seq-2-seq) neural network. In sequence to sequence learning, the aim is to train models from one domain to another. A common example is translation where for instance a text in English is converted into a text in French. In the context of this thesis, we can think of the conversion as being from a misspelled Danish word to a correct Danish word. In this case, we use a character level sequence to sequence, meaning that the sequence consists of characters instead of words.

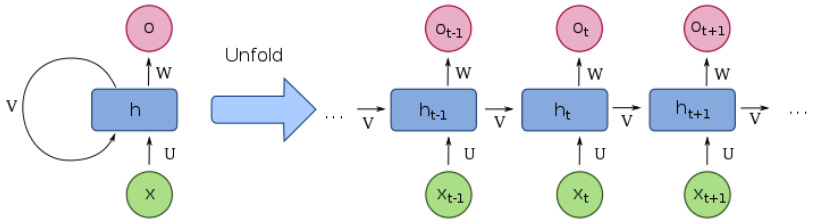


Figure 2.8: A recurrent neural network and the unfolding in time of the computation involved in its forward computation (Wikimedia Commons, 2017b).

Another property of sequence to sequence is the ability to have different input and output lengths. The approach is to determine a fixed length and pad the remaining characters with a null character for shorter words. In addition to the padding, we require a masking mechanism during training to determine whether there is an input/output present for a certain time step or if it is just padding. Using this approach, we can construct networks with different input and output lengths as shown in Figure 2.9.

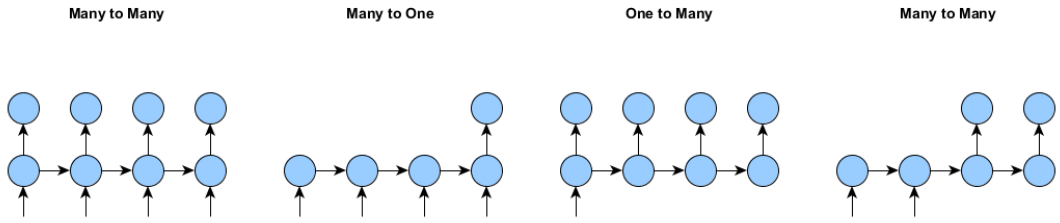


Figure 2.9: Different kind of RNNs where the lower row represents the input mask and the upper row represents the output mask. In the first case from the left, no masking is required whereas the second and third cases only require an output and input mask resepectively. The last case requires both an input and output mask (Eclipse Deeplearning4j Development Team, 2018).

2.5.1 Back Propagation

Back propagation is a problem that eluded a practical solution for decades. The problem is to calculate the partial derivative of error with respect to the weight parameters and

iteration round (Sathyanarayana, 2014). A solution was introduced by Rumelhart et al. (1986) where they explain it as

> The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure.

The reason why we want to be able to calculate the partial derivative at each iteration is to know the slope and find the minimum error, as explained with the example involving a ball earlier in this chapter.

In other words, the goal of back propagation is to update the weights in the network so that they cause the output to be closer the target output. By doing this, the error is minimized for each output neuron and the network as whole.

2.5.2 LSTM

Hochreiter and Schmidhuber (1997) introduced Long Short-Term Memory (LSTM) for the first time and later Gers et al. (2000) improved upon the concept by adding a forget gate to the LSTM cell. Before forget gates, an LSTM cell could grow indefinitely and eventually break down if not completely reset. Adding a forget gate enables the possibility of an LSTM cell to reset itself at appropriate times.

LSTM units were created to counter the issue of vanishing gradients that RNNs are prone to produce during training. The vanishing gradient problem is the problem of gradients growing too large or too small during training which makes it difficult to model long range dependencies, e.g. more than 10 time steps (Patterson and Gibson, 2017).

The critical part of the LSTM is the memory cell and its gates: the forget, output, and input gate. The contents of the memory cell is modulated by the input and forget gates. If both gates are closed, the memory cell will stay unmodified during that one time step. This gating structure allows information to be retained across multiple time steps and thereby allows gradients to flow across multiple time steps. This allows the LSTM model to overcome the vanishing gradient problem. Patterson and Gibson (2017) stress how important and central the LSTM unit is for RNNs as it enables the network to maintain a state over time.

2.5.3 Bidirectional RNNs and BiLSTM

For many sequence labelling tasks, it would be beneficial to not only have access to past context but also future context. In the case of letters and classifying a certain letter, it's useful to know both past and future letters. However, since standard RNNs process sequences as a time sequences, future context is ignored.

Graves (2008) discusses two approaches on how to implement the Bidirectional RNN (BiRNN). The obvious solution of adding a time window for future context to the network input suffers from intolerance of distortions and a fixed range of context which isn't compatible with the unfixed range of context required for RNNs.

A distortion is when the data differentiates itself to earlier data in other ways than added noise. In an image, a distortion would bend and change the form while noise would just make it harder to see the shape.

The other solution is to introduce a delay between the input and output and thereby give the network a few time steps of context. This method allows the RNN to retain the robustness towards distortion but the issue of fixed range of context remains. In addition, it also places an unnecessary burden on the network. Neither of these approaches remove the asymmetry between the past and the future information.

The BiRNN presented by Baldi and Pollastri (2003); Schuster and Paliwal (1997); Schuster (1999) offers a more elegant solution. The idea is that during training, each sequence is presented both backwards and forwards to two separate hidden layers, both of which are connected to the same output layer as shown in Figure 2.10. Graves (2008) goes on to say that "BiRNNs consistently outperform unidirectional RNNs on real-world sequence labeling tasks".

A Bidirectional LSTM (BiLSTM) network is an implementation of the LSTM cell that supports BiRNN as explained above.

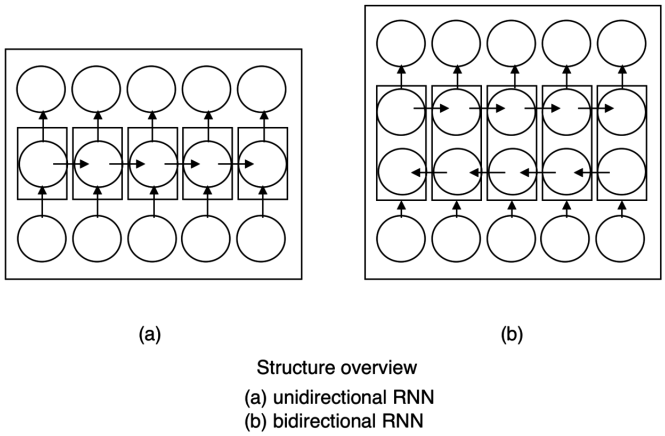


Figure 2.10: Standard and bidirectional RNNs (Wikimedia Commons, 2015).

2.6 Autoencoder

An autoencoder automatically encodes the input to classify it and hence is an unsupervised method of learning. As shown in Figure 2.11, the autoencoder is symmetrical with a bottleneck in the middle. The idea is that compressing the data will force the system to find patterns hidden in it. These patterns will automatically be found by the system as it evaluates its weights during training.

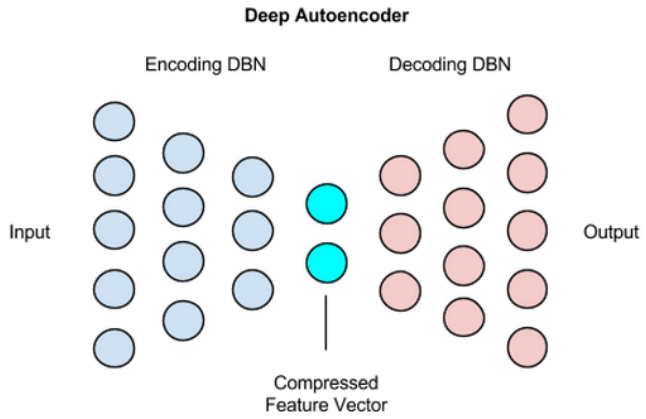


Figure 2.11: How an autoencoders layers are built (Eclipse Deeplearning4j Development Team, 2018).

2.7 Compound Words in Danish

In the Danish language, words are often a compound of words which means that two words are merged into one word. Two examples of this are:

anders + gade → andersgade
pant + brev → pantebrev

Note that in the latter, an e is applied as a binder between the words. The compounding means that these words include a suffix and a prefix that are complete words themselves.

As shown in Figure 2.12, almost 92% of the name tags for Way components in the OSM data set contain one of 24 different suffixes. The most common suffixes are "vej" and "gade". The label other spans 22 common suffixes - sti, boulevard, alle/allé, vang, bakk, park, bro, led, væng, plads, høj, passage, sted, bane, træde, promenad, spor, hav, torv and esplanad. The label missing denotes names that don't contain any of these 24 suffixes.

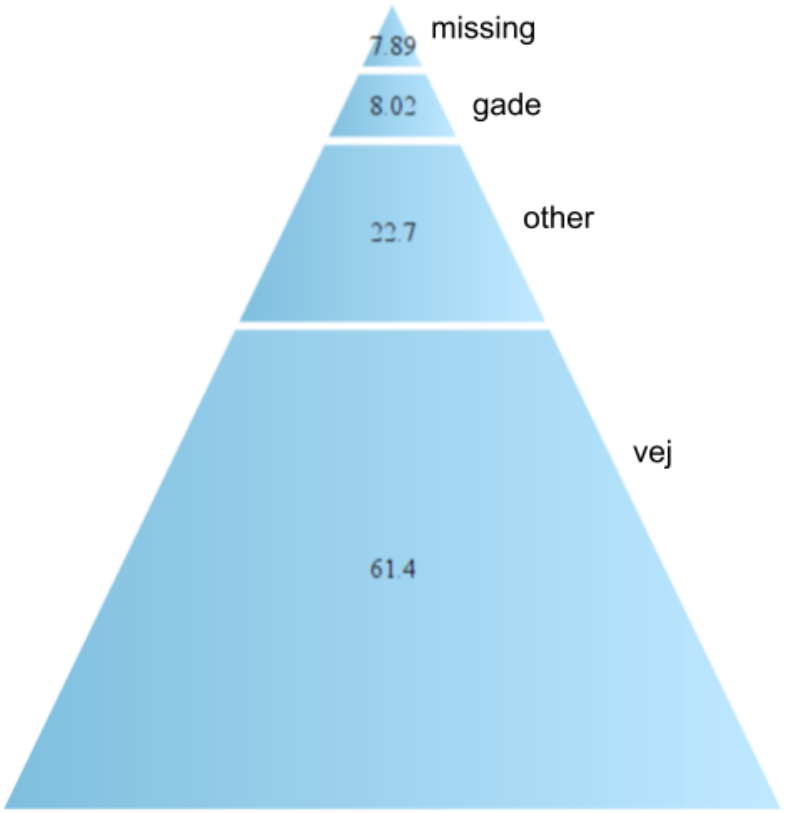


Figure 2.12: Pie chart displaying the different suffixes of name tags in the OSM dataset.

Sjöbergh and Kann (2004) explain how there are a few common ways to create a compound in the Nordic languages, or Danish in our case. See Table 2.1 for the operations with examples.

Type		**Example**
Null operation	*0*	Corselitze+vej
Addition	*+e*	Torv+e+gade
	+s	Shetland+s+gade

Table 2.1: The null and addition operations for compounding forms in Danish.

2.8 OpenStreetMap and Tags

In OpenStreetMap, each component can have multiple tags. These tags might contain information that is valuable to provide a greater understanding of the component. In this thesis we have used the following tags:

Name: The name tag contains the name of the Way component where the name should reflect what is denoted on the street sign.

Noname: The noname tag is a Boolean that is true if the Way has no name. If a Way has a name but it hasn't been provided yet, the noname tag shouldn't be used.

Surface: The surface tag explains the type of surface that the Way component has. This can be paved, cobbled or any of 27 other classes.

Highway: The highway tag explains the type of Way component. This can be motorway, pedestrian or any of 17 other classes.

Maxspeed: The maxspeed tag contains the maximum speed for the Way component. This can be numeric or text like for instance *signals* or *none*.

2.9 Damerau-Levenshtein Distance

The Damerau-Levenshtein distance is a string metric used to measure the edit distance between two sequences. The edit distance between two sequences is the minimum number of operations required to change one word into the other. The following operations are supported.

1. Delete a character.
2. Transpose, swap two characters next to each other.
3. Insert a character.
4. Replace a character with different one.

Damerau (1964) states that these four operations correspond to more than 80% of all human misspellings when considering misspellings that can be corrected with at most one edit operation. Throughout this document, the term edit distance will be a reference to the Damerau-Levenshtein distance.

A practical example of two words that are one edit distance apart are "live" and "hive" since only one edit is required to change one into the other, replace l with h and "live" becomes "hive".

2.10 Evaluation Metrics

To evaluate our results, we have primarily used the F_1 score. The F_1 score is briefly presented in the first subsection below. Another important visualization tool that we have used, the confusion matrix, is explained in the following section. In order to further improve the understanding of how well the system is performing in the case of spell correction with the specific prerequisites for this thesis, we introduced an Accurate Correction Rate which is defined in the last section.

2.10.1 F-measure

The evaluation of the classifiers' performances have been partially made based on the precision, recall, accuracy, and F-measure metrics. These measurements are made for

each class and then weighted together into an average (given that the classification is a multi-class classification).

Precision, also called positive predictive value, is the fraction of relevant instances among retrieved instances. For example: If we have a system that predicts if an image contains a hot dog or not and our system classifies the image as a hot dog eight times out of twelve whereas only five of those were a hot dog in reality we would have had a precision of $\frac{5}{8}$. Precision can be seen as a measurement of exactness.

Recall, also called sensitivity, is the fraction of relevant instances that have been retrieved among all relevant instances. Using the example from precision, with the addition that only six out of the twelve images is of a hot dog, the recall would be $\frac{5}{6}$ as we did the correct classification of the hot dog in five out of six cases. Recall can be seen as a measurement of completeness.

One way of getting an overview of a system's performance is to use the harmonized mean created out of the precision and recall. This harmonic mean is called the F score. Throughout this report, the F_1 score will be used to measure how well the system is performing. The F_1 score is calculated using equation 2.1.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.2}$$

2.10.2 Confusion Matrix

A confusion matrix is a matrix which visualizes the accuracy of the predictions performed by an algorithm. Figure 2.13 shows a confusion matrix where the diagonal is highlighted. We can see that the values along the diagonal are greater than the surrounding values. The diagonal values are central and tell us how well a system has performed. The rows are the system's predictions and the columns are the true labels. In the case of Figure 2.13, the situation is reversed. If the diagonal value for class Y is 1.0, the system labels the input 100% correctly in the case of Y.

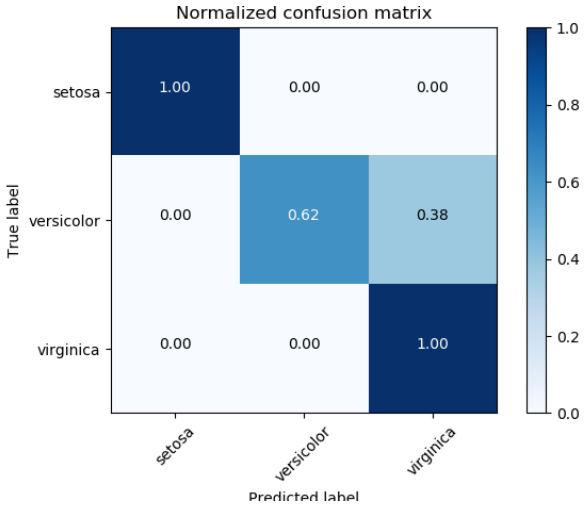


Figure 2.13: Example of a confusion matrix (Pedregosa et al., 2011).

2.10.3 Accurate Correction Rate

Accurate Correction Rate (ACR) is a normalized value/ratio of how many corrections that are introduced in comparison to errors. Below, the rest of the metrics used for the evaluation of the spell corrector are defined.

Errors Introduced (EI): Well spelled words that have been corrected to misspelled words.

Corrections Introduced (CI): Misspelled words that have been corrected to a well spelled words.

Erroneous Corrections Introduced (ECI): Misspelled words that have been corrected to misspelled words.

Unchanged words (U):

C: Number of correct corrections made by the system. This includes CI and correctly unchanged words i.e. words that shouldn't be corrected.

Accurate Correction Rate (ACR): A normalized value that gives an overview of how well the system performs in practice, calculated by

$$\mathbf{ACR} = \frac{\mathbf{CI}}{\mathbf{EI} + \mathbf{CI}}$$

.

Chapter 3
Approach

We have used machine learning to attempt to correct WAY names in the OSM data set for Denmark. The aim has been to develop a generalized method that can be adapted for different countries by changing the data used for training. This section presents the methodology that we used for the thesis work with subsections for the CRISP-DM process model, a method section for discussion of choices and considerations made during the thesis work as well as a section describing the tools used for the implementation.

3.1 CRISP-DM

We have applied the CRISP-DM (CRoss-Industry Standard Process for Data Mining) methodology to our thesis work. According to a poll by KDnuggets in 2014, CRISP-DM was the most common main methodology for analytics, data mining or data science projects (KDnuggets, 2014). The process consists of six phases – Business understanding, Data understanding, Data preparation, Modeling, Evaluation, Deployment – as shown in Figure 3.1 (Harper and Pickett, 2006).

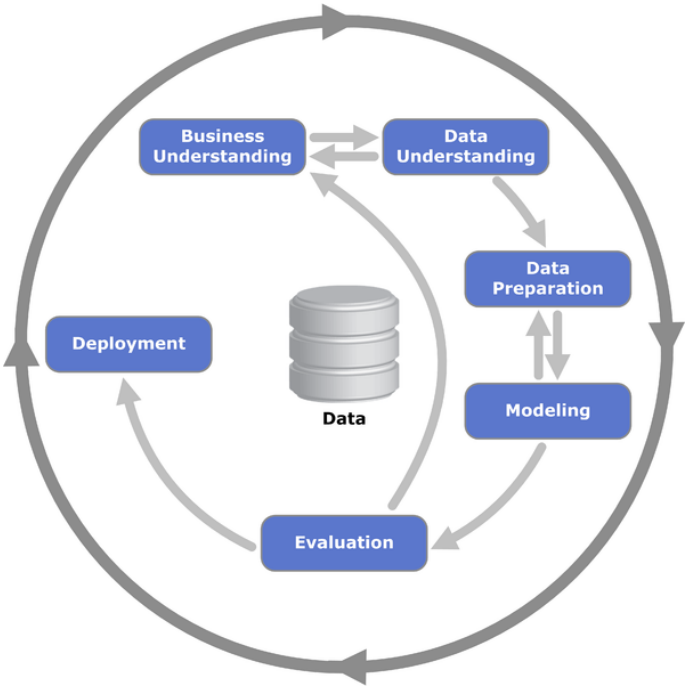


Figure 3.1: The CRISP-DM life cycle model (Wikimedia Commons, 2012).

In Figure 3.1, the arrows imply the main dependencies between phases. The phases don't have to be performed sequentially, instead most projects move between the phases according to the needs of the project (Chapman et al., 2000). Below follows a short description of the six different phases.

The **Business understanding** phase focuses on clarifying what the client expects to gain from the data mining. Project risks can be minimized by clarifying the business reasons that motivate the project.

The focus of the **Data understanding** phase is to collect and analyze data in order to gain a better understanding of the data and to facilitate the data preparation phase.

The **Data preparation** is one of the most important and time-consuming phases of CRISP-DM. It's estimated that data preparation usually takes 50-70% of a project's time and effort (Chapman et al., 2000, chap. 2). Tasks during the preparation can include sorting, cleaning, and merging data records.

The **Modeling** phase is usually done in multiple iterations with different parameters and focuses. Often the data can bring different things to light by applying different approaches.

The **Evaluation** phase is needed to determine the success of the mining and modeling. In order to do this, measurements and comparisons have to be made and if the results are not satisfactory, the above phases can be reiterated until a viable solution has been found.

The **Deployment** phase means that the results are integrated into the final system or that the results are used to make improvements to the organization.

3.2 Method

This section describes how we chose to investigate the problem and construct possible solutions. The first section covers pre-work made in accordance with the Data understanding and the Data preparation phase of CRISP-DM. The second part focuses on the Evaluation and Modeling phases and the choices and considerations made during the thesis work. The phase of Business understanding is however excluded as the customer of ÅF didn't wish to reveal the business reasons for this project. The last phase of Deployment can be made by the customer where the program can be converted into an Atlas Check. Atlas Checks is an automatic framework for automatic testing on OSM as described in the Introduction.

3.2.1 Pre-work

We began with investigating the corrections made to name tags in the OSM data set. The change data was extracted and analyzed by comparing an older (2015-01-01) and newer version (2018-01-18) of the data set. The data sets were retrieved from the Geofabrik organization. When comparing these two revisions of the data set, we assumed that the most recent one was correct.

To facilitate the retrieval of data used for creating files to be loaded into RAM by Atlas (read more on the Atlas tool in the section Experimental Setup), we divided the geographical area of Denmark into 900 rectangles. This resulted in about 500 usable files since some were filtered out for containing too little information (like areas covering sea for instance). Through the use of a custom filter, the Way segments that had its start and end nodes on separate sides of a boundary were filtered out. The "way sectioning" rules performed by Atlas were also omitted. The rules determine how to break ways at intersections in order to create Atlas Edges. Since we are interested in the Way names and not the structures this wouldn't have an effect on our results for the data.

We then analyzed the collected data. The result can be seen in Figure 3.2. The edits with an edit distance greater than three aren't considered edits but rather name changes as this was the case when some of the instances were manually inspected. We believe that even though this does not confirm that 80 percent of all misspellings are within one edit distance as Mays et al. and Damerau proclaimed; their statistic is true and our sample is too small to confirm this. The distinction that an edit distance greater than three is a change of name enables us to train a more careful algorithm which handles edit operations of less complexity.

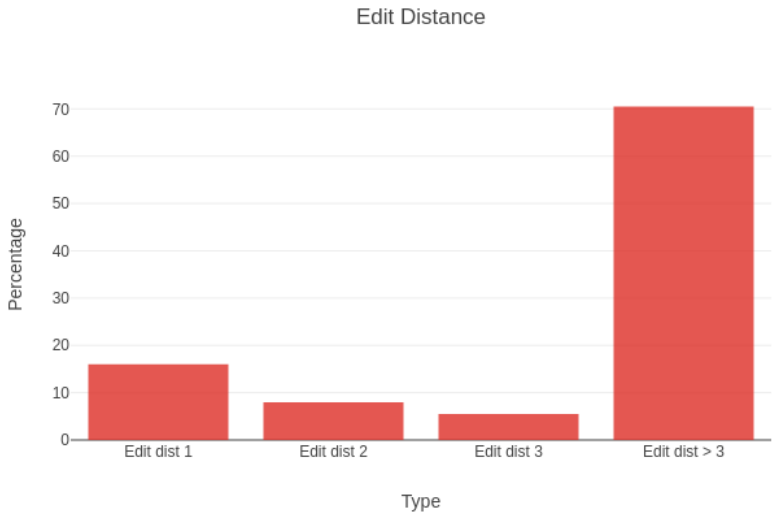


Figure 3.2: OSM change data when comparing the 2015-01-01 and 2018-01-18 data sets.

The total number of edits that could be extracted from the data set was 1,800 compared to the about 52,000 unique way names in the OSM data set for Denmark. Approximately 300 out of the 1,800 changes were of edit distance 3 or less. This means that of 52,000 unique names, 300 have been updated due to a misspelling of edit distance 3 or less between 2015 and 2018. To get sufficient data to train a machine learning algorithm, we constructed an experiment with inspiration from Whitelaw et al. (2009) where participants had to type Danish street names on a keyboard without using backspace or delete. Through this experiment, we gathered almost 1600 hand copied street names. The data was analyzed and the statistics used to introduce artificial noise to OSM data. The noised data was in turn used for training the machine learning algorithm. We used the original experiment data as test data.

The misspellings in the experiment data is not as natural as the OSM change data, but preferable to artificial noise based on a random function. If the machine learning algorithm performs well on this type of data, this indicates that the algorithm has learned from the data. However, to achieve better results when correcting real OSM data, training and test data of higher quality is required.

3.2.2 Implementation and Evaluation

This subsection covers the modeling and evaluation phases where we iteratively test different models and approaches to the problem and evaluate the progress.

Spelling Corrector

We investigated the most common edit operations of distance one in the OSM change data. Mays et al. (1991) and Damerau (1964) observe that 80 percent of misspellings are derived from single instances of insertion, deletion, or substitution – meaning an edit distance of

1 – and that words are usually spelled as intended. Our research showed that the most common edit distance 1 operation was the insertion of an *s*, see Figure 3.3. When manually inspected, the *s* was often inserted next to another *s* in the word. From this we constructed a baseline to the problem of spell correction where we trained an algorithm for insertion of *s* or double *s* in words. The results were encouraging for expanding the concept to develop a machine learning algorithm for general spell correction. By developing the baseline, we could ensure that there was a simple way to solve an isolated part of the problem. However, the low amount of change data in combination with the constraint of inserting or deleting an *s* next to another *s* in a name means that no real conclusions can be drawn regarding the efficiency of a solution for the baseline on the general problem.

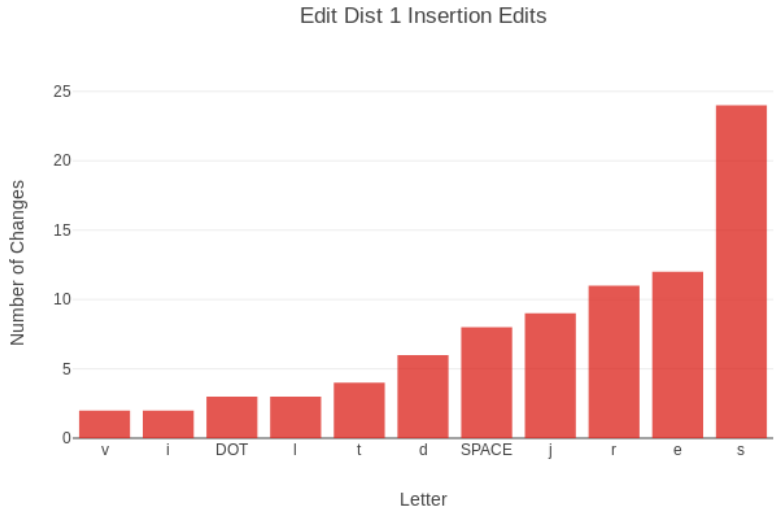


Figure 3.3: Insertion edits for changes of edit distance 1.

After implementing the s/ss baseline, we decided to look into algorithmic solutions to spell correction. Read more on this in Section 3.3.3. By combining the algorithmic approaches with the final machine learning algorithm in a two-headed approach, we hoped to achieve higher accuracy. It is also beneficial to be able to compare results from the spell checker with established spelling correction approaches.

In order to increase the validity of our final solution, we investigated the performance when the system was trained and tested on a different data set. For this evaluation, we chose OSM data for Estonia as we wanted a smaller data set that didn't require much work when extracting relevant information. Like for the main case of OSM data for Denmark however, the lack of sufficient data is a concern when interpreting the results. It would be preferable to evaluate on a country with a larger data set of updates so that the artificial noise introduced to the training data is more representative of real change data.

The data for the Estonian spell checker was created in the same way as in the Danish case. The exception is that there was no file with manual, crowdsourced data. We made use of the edits from 2015 to 2018 to generate new misspellings. Out of the generated data we shuffled and selected 2000 names to test on – note that these names are a combination of noised words from both OSM data and Wikipedia.

During the training of the neural networks, tweaking was required to optimize the results. This includes regularization such as l2, dropout, and early stopping. Also, the learning rate and updaters were changed and finally, the network architecture was also experimented with through different setups where the width and depth of the network were changed. The tweaking was made on a trial and error basis based on what we deemed probable to be effective.

Missing Names

The missing names algorithms are baselines to simple problems observed in the data. The correctness of the algorithms were manually validated since no automatic evaluation framework was developed. The aim is to integrate the algorithms to the Atlas checks framework and send the suggested changes to MapRoulette were they can be manually edited. MapRoulette is a gamified approach to fixing incorrect OSM data by breaking problems into micro tasks (OpenStreetMap, 2018). Statistics regarding the number of edges that are missing names are presented in the Results Section.

Anomalies

For anomalies in the data, we decided to work with the name, maxspeed, surface, and highway tags. We created a data set of artificial anomalies from the OSM data since there was no data set readily available. By creating a set of all tags that a street name has in the OSM data, we could determine anomalies from all other possible tags in OSM, see Figure 3.4. Note that $(0, 30..)$ denotes custom classes created for speeds in order to reduce dimensions and the complexity of the problem.

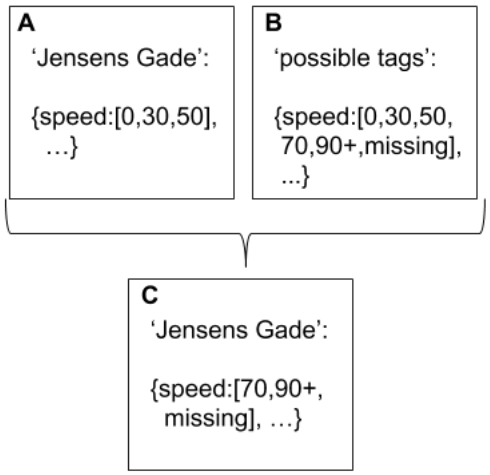


Figure 3.4: Visualization of how we create the tags that will later be randomized into anomalies. *A* denotes all tags that have been connected with "Jensens Gade", *B* denotes all possible tags throughout OSM and *C* denotes the difference betweeen these two. We will generate anomalies by randomizing the tags in *C* with the associated Way name "Jensens Gade".

This method of creating data is simple and time effective but not ideal since some of the anomalies might never occur in the OSM data.

Another possible approach would be to look at what tags coexist in the data, such as asphalt and 120 km/h, and connect these to a Way name that has never occurred together with these tags. For instance, a path would typically not have asphalt and 120 km/h attached. A path such as "Stenstien" would therefore be combined with these two tags to create an anomaly.

A third approach would be to manually create the anomaly instances, however, this would introduce a bias towards what we want to believe is an anomaly.

3.3 Tools

In this section we introduce the different tools used in this master thesis.

3.3.1 Atlas

Atlas is a tool written in Java that is open source on Github under OSM Lab. The main advantage of Atlas is that it loads OSM data directly into the memory. Having OSM structured in this way creates the possibility of traversing the OSM data like a routing algorithm would. This facilitates the process of exploring surroundings in the data set such as what edges and nodes are connected to a particular edge. The following types are available in Atlas when loading a map (Apple Inc, 2017).

1. Edges and Nodes are navigable items.
   (a) Edges contain a start and an end Node.
   (b) Nodes contain out- and in-going edges.
   (c) Edges are uni-directional, a two-way road from OSM is represented as two edges in reverse directions.
2. Area, Line, and Point is used for non-navigable features such as Parks and Points Of Interests.
3. Relations are a true mirror of relations in OSM. They link features together.
4. ComplexEntity are entities built on the fly for Atlas. These are used for concepts that are more complex than a simple feature. E.g. ComplexBuilding with multipolygon and parts.

In our report, we focus on Edges since they correspond to the elements that constitute Ways.

3.3.2 DeepLearning4Java

The DeepLearning4Java or DL4J framework has been used as Java was a language requirement. DL4J is a group of deep learning tools that is packaged together as a suite to perform functions such as

- Integration
- Vectorization
- Modeling
- Evaluation

DL4J is an Open Source project with an active core group of developers that work daily with DL4J. DL4J is designed to have modern execution platforms in mind from the beginning and does not suffer from parallellization issues that some other libraries have had. DL4J is focused on enterprise-grade functionality and is targeted at practitioners who need a Java Virtual Machine (JVM) option in deep learning but also want the speed of C++ and the power of Spark for parallel computation.

3.3.3 Algorithmic Spell Correction

To understand how spell correction works and how to do it well we implemented two different algorithmic solutions to the problem, Norvig's Spelling Corrector and SymSpell, where the latter is applied to the end-system.

Norvig's Spelling Corrector

Norvig (2007) introduced one of the most famous baseline spelling correctors. Norvig makes use of the Damerau-Levenshtein distance operations and a corpus to identify which words that exist and how common they are.

To correct a word W with edit distance one, the algorithm iterates the characters $c_0..c_n$ of W and applies four different operations at each character c_i, creating new versions of W, where each version is one edit distance from W.

For all the versions of W, the probability of the version being correct is calculated by dividing the amount of times the word is mentioned by how many words there is in the corpus. Norvig also applies a heuristic that if W exists in the corpus, that implies that it is the correct one. This creates a few situations where a word won't be corrected even though it's incorrect. The same applies if one searches with an edit distance of two and a word exists with edit distance of one – then the edit distance of one is chosen.

We implemented Norvigs spelling corrector in Java but it was lacking performance for edit distance of two on longer words as the naive approach of Norvig grows exponentially in time complexity.

SymSpell

SymSpell is a C# library that utilizes a Symmetric Delete Spelling Correction (SDSC) algorithm. The SDSC algorithm makes the spell correction and fuzzy search about 1 million times faster than the spelling correction algorithm by Peter Norvig when using an maximum edit distance of 3, as was shown by Garbe (2015).

The SDSC algorithm reduces the complexity of both edit candidate generation and dictionary look up for a given edit distance. Since the algorithm only deletes characters from the word, it is language independent.

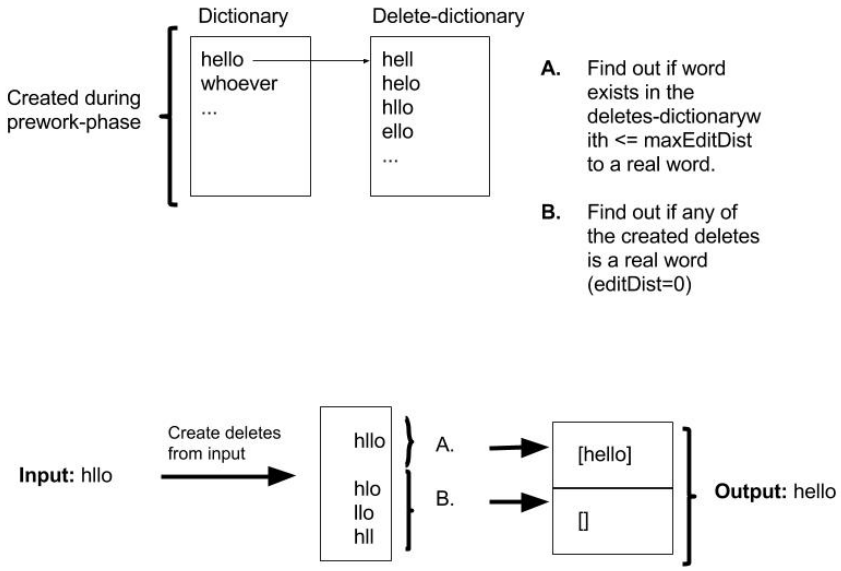


Figure 3.5: Example of SymSpell with a maximum edit distance of one.

Transposes, replaces, and inserts of the input word are transformed into deletes of the dictionary term. Replaces and inserts are not only expensive to do in comparison to deletes but also language dependent. This is especially notable in languages such as Chinese with 70,000 Unicode Han characters. A lot of the combinations never create an actual word. By creating the deletes from the dictionary term it is possible to only create the expansions that lead to an existing word.

Because of the architecture of the algorithm, the pre-work is the hard work. An average 5 letter word has about 3 million possible spelling errors within a maximum edit distance of 3 and with SymSpell you only need to pre-calculate and store 25 deletes to cover them all.

Using the example in Figure 3.5, we correct the misspelling “hllo” to “hello” by first reducing “hllo” to “hlo”, “llo”, and “hll”. Note that if we were looking for a correction within two edit distances we would’ve applied the same delete to all four versions again. We then take these versions and try to find the same in the dictionary, where “hllo” is found. “Hllo” is created in the dictionary by the deletion of “hello” during the pre-work phase. The dictionary then identifies “hello” within one edit distance from “hllo” by backtracking itself. This means that we have done an insert of an “e” between the “h” and “l” by backtracking our deletion.

Below are descriptions of the customizations we made to the SymSpell framework.

Frequency Dictionary

The quality of the dictionary is of greatest weight to retain a high correction quality. In the beginning of the project, a dump of the Danish Wikipedia was used. To improve the performance, it was however decided to instead use a corpus from `korpus.dsl.dk/`. This corpus is maintained by Det Danske Sprog- og Litteraturselskab (The Danish Society for Language and Literature) and is therefore deemed to be of good quality.

SymSpell Specialized Look Up

We created our own look up method that is specialized to be cautious in finding corrections. This specialized look up only finds edits within one edit distance and the method can freely remove characters either from the front or from behind until a correctly spelled word is found. When a word with edit distance is found on either side the algorithm will try to correct the leftover word. It then merges the removed characters with the corrected word and gives this correction a scoring depending on edit distance, excluding the removal of characters, and term frequency. This technique was derived from the knowledge of compounding in Danish as was shown in Section 2.8.

An example of this would be the name “Andersvej”. If we misspell it as “Andrsvej”, the system would remove characters from the right until it found “vej” and then attempt to correct “Andrs”. The algorithm wouldn’t find any correction removing characters from left. The name “Andersvj” would instead be corrected by removing characters until “Anders” is found and then “vj” would be corrected into “vej” as the system wouldn’t find anything when removing characters from the right.

Chapter 4
Implementation

The implementation supports finding missing names, correcting misspellings, and finding anomalies. We decided to split the solution into three different subsystems where each focuses on one of these different parts. This section covers the s/ss spell checking baseline together with the three subsystems called WAY name spell checking, missing names and anomalies. In each section, there is a subsection describing the experimental setup, a short description of the different approaches and results from each evaluation. The training and testing was performed on data from the area of Denmark. For the spell checking system, training and testing was also performed on data of Estonia.

4.1 S/SS Spell Checking

A common edit in OSM involves the insertion and deletion of the letter *s*. A spell checker which corrects instances of single or double *s* in names was therefore developed as a simple baseline to the problem of name correction. An example is "Korsbakken" which is correctly spelled with one s. An example of a misspelling is "Haselbo" which should be spelled "Hasselbo". This section presents three implementations to the problem: a baseline, a random forest, and a feed forward neural network (FFNN) implementation.

4.1.1 Experimental Setup

The random forest and FFNN implementation uses the character before and after the s/ss characters as features in a name and predicts the appropriate s form. By not including the s/ss classes as input, neither true-negative nor false-positive examples was required. Instead, the s/ss characters could be used as labels for the input. Hence, we didn't need to generate an artificial training set but could instead use the WAY names from the current OSM data set.

The training data was undersampled in order to minimize the effects of the more common single s case on the classifier. The ratio was set to 50 percent after experimenting with different values. See Table 4.1 for details.

Data	s	ss	Total
Original	386,001	25,601	411,602
Bootstrapped	25,600	25,601	51,201

Table 4.1: Data on *s* cases in Way names from the OSM data set.

4.1.2 Baseline Implementation

We implemented the first baseline in the easiest possible way. The baseline randomly replaces instances of either single or double *s* in names with a random *s* form (single or double *s*). The performance of this baseline is shown in the results section.

4.1.3 Random Forest Implementation

In order to further boost the F_1 score of the classifier, a limit is set on the level of certainty required in order to make a prediction. The final version requires a prediction certainty of 90% or above. From this follows a lower number of predictions for the data set with a new total of 4,392 examples (a 67% decrease).

In Table 4.2, confusion matrices for the random forest classifier with and without a certainty requirement are shown. The certainty requirement was chosen by comparing different requirements between 50 and 95 percent and thereafter using the best one.

ss	**267**	2,547	2,814
s	49	**10,846**	10,895
	316	13,393	**13,709**

ss	**59**	20	79
s	0	**4,313**	4,313
	59	5,628	**4,392**

Table 4.2: Confusion matrix for the Random Forest classifier without a certainty requirement to the left. To the right, the same classifier but with a certainty requirement of 90% or above.

Initially, the classifier model was a decision tree but the random forest model was tried and yielded better results and was therefore chosen. A gradient boosting classifier was also developed using the Python library XGBoost (eXtreme Gradient Boosting). XGBoost did not offer an improvement compared to the decision tree classifier with the certainty requirement. The results are shown in Table 4.3.

ss	**126**	92	218
s	2	**5628**	5630
	128	5720	**5848**

Table 4.3: Confusion matrix for XGBoost classifier.

4.1.4 Feed Forward Neural Network Implementation

Stochastic gradient descent (SGD) was used as optimization algorithm. The updater was chosen to RMSProp. RMSProp gave, tied with Adam, the best performance when compared to Nesterov's momentum, Adagrad, and SGD – where SGD means that you only use the learning rate. Multiclass cross entropy was used since it meant the output could be in the format $[X, Y]$ where X is probability of first class and Y is the probability of the second class. Since we only have two classes, the network is a binary classifier.

The results are shown in Table 4.4.

ss	**108**	208	316
s	288	**13,106**	13,394
	396	13,314	**13,214**

Table 4.4: Confusion matrix for the feed forward neural network.

4.1.5 Results

As shown in Table 4.5, the Random Forest classifier resulted in the best score and the Feed Forward Neural Network performed the worst out of the three.

	Class	Precision	Recall	F_1	Support
Random Baseline	ss	0.02	0.50	**0.04**	316
	s	0.98	0.50	**0.66**	13,393
Random Forest	ss	0.75	1.0	**0.86**	59
	s	1.0	1.0	**1.0**	4,333
Feedforward NN	average	0.63	0.66	**0.3**	13,710

Table 4.5: Results for s/ss spelling correctors.

4.2 Way Name Spell Checking

The spell checker predicts misspelled Way names and generates suggestions. If the suggestion is the same as the label in the test set, the name is assumed to be correct and otherwise the name is considered to be incorrect. The section follows the structure of the previous section with subsections of experimental setup, a description of the implementations and a results section.

4.2.1 Experimental Setup

The Way name spell checker was trained on the corpus from the Danish Society for Language and Literature and the complete OSM data set. These two data set was artificially noised (misspelled) to fifty percent and then shuffled. The artificial noise was based on data from the crowdsourcing experiment that was described in the Method section (3.1)

and the differences found in OSM data set from 2015 and 2018. In other words, we applied the same type of edits that were made historically.

The evaluation was made in three parts.

- **Comparison with s/ss baseline:** Evaluation on Way name corrections to see if s/ss parts in names have been correctly idenfified.
- **Crowdsourced data:** Correction of data from the experiment for collecting spelling mistakes.
- **OSM data:** OSM data sets from 2014 and 2015 were evaluated on and the results compared to the latest (2018) data set.

The spell checker was also trained and tested for Estonian way names using the BiRNN from the Danish spell checker. A dump of the Estonian Wikipedia as well as two different OSM files of Estonia were used as data. The OSM files were retrieved from Geofabrik where the first was from 2014-01-01 and the second from 2018-04-01.

4.2.2 RNN and BiRNN Implementations

The baseline RNN was implemented as a sequence to sequence (seq-2-seq) network where each time step is a character and the input/output can be a sequence of undecided length. The implementation was naive but surprisingly effective.

We improved the baseline by implementing an autoencoder structure and creating a bottleneck in the layers to compress information and keep the most important features. As shown in Table 4.8, this significantly improved the system.

We later reiterated on the structure of the system and converted it to a bi-directional recurrent neural network (BiRNN). The baseline BiRNN works just like the baseline RNN except that it is bi-directional, meaning that the input is seen both forward and backwards before the output is produced.

Following from the previous success of using the autoencoder structure – i.e. bottlenecking the network – it was the first improvement applied to the BiRNN.

We also improved the system by applying SymSpell on uncertain predictions. Read more on this in the discussion.

4.2.3 Results

Comparison with S/SS Baseline

The confusion matrices for the s/ss spell checker and the Way name spell checker on the test data is shown in Table 4.6. The s/ss spell checker uses a certainty requirement of 90% or more. This is the reason why the support for its confusion matrix is 461 instead of the total number of examples, 1184.

ss	**13**	9	22
s	0	**439**	439
	13	448	**461**

ss	**26**	5	31
s	6	**1147**	1153
	32	1152	**1184**

Table 4.6: Confusion matrices for the s/ss spell checker to the left and the BiRNN to the right.

Evaluation on crowdsourced Data

Table 4.7 shows the results for the F_1 score evaluation for the different implementations of the Way name spell checker on the crowdsourced test data.

	Accuracy	Precision	Recall	F_1
RNN Baseline	0.8391	0.7788	0.7235	0.7958
RNN Autoencoder	0.8613	0.8515	0.7495	0.8285
BiRNN Autoencoder	0.8826	0.8789	0.8105	0.8634

Table 4.7: RNN and BiRNN performance.

Table 4.8 shows the Accurate Correction Rate (ACR) for the different implementations of the Way name spell checker. In this table we also included the ACR score for SymSpell as a benchmark score.

	ACR	EI	CI	ECI	U	C
SymSpell	**0.1348**	231	36	128	1087	888
RNN Baseline	**0.0086**	231	2	245	1058	620
RNN Autoencoder	**0.5143**	17	18	118	1382	843
BiRNN Autoencoder	**0.6363**	20	35	153	1274	903
BiRNN Autoencoder + SymSpell	**0.6949**	18	41	153	1270	911

Table 4.8: EI: Errors Introduced, CI: Corrections Introduced, ECI: Erroneous Correction Introduced, U: Unchanged, C: Number of correct predictions made by the system.

Evaluation on OSM Data

Table 4.9 shows the results for the Way name spell checker 2014 and 2015 OSM data sets. *ET* denotes the Estonian Way name spell corrector introduced in the next section.

	F_1	ACR	EI	CI	FC
BiRNN Autoencoder	0.97	0.004	1653	7	73
ET BiRNN Autoencoder	0.94	0.002	506	1	5

Table 4.9: Performance of the system on the OSM change data. FC = Failed Corrections, correctly identifying a misspelled word but applying an incorrect correction. Explanation of labels in text above.

Estonia

Table 4.10 shows results from evaluating the spell checker on OSM data from Estonia.

	ACR	F_1
SymSpell	0.36	-
BiRNN Autoencoder (dk_alphabet)	0.85	0.983
BiRNN Autoencoder (et_alphabet)	0.88	0.986

Table 4.10: Results when trained on Estonian data for Estonia.

4.3 Speed Limit Anomalies

This network was aimed at detecting anomalies by predicting speed limits from names. The spell checker network structure was used, but the output was changed to instead be a speed limit class at the end of the time sequence (name). Also, uni-, bi-, and trigrams of characters from the name were tried as inputs to the network. First, the amount of dimensions, or classes, was decreased by filtering the n-grams by occurrence. This reduced the number of classes to about 100. The other n-gram classes were assigned the class *unknown*. Further, an embedding layer was used to reduce the dimensionality of the input in an unsupervised fashion. We tried four different intervals for speed classes as output from the network, these can be seen in Table 4.11.

X classes of speed (one class for each possible maxspeed tag)
5 classes of speed (0-30, 40-50, 50-70, 70+)
4 classes of speed (0-30, 40-50, 60+, missing)
3 classes of speed (0-50, 50+, missing)

Table 4.11: Different intervals for speed classes.

This approach did not converge during training and was therefore abandoned.

4.4 Name and Tag Anomalies

The purpose of the anomalies implementations are to identify anomalies in the data given a number of inputs. The following subsections describe the implementations which identify anomalies in the data from the last five characters of a Way name tag together with the maxspeed, highway and surface tags. This includes a section on the experimental setup, shorter descriptions of the implementations and a final section presenting the results.

4.4.1 Experimental Setup

There's a large number of classes in the OSM syntax for the different tags used as input and these were therefore manually divided into a number of categories. The maxspeed tag has five categories, the highway tag has eight categories and the surface tag has seven

categories. The maxspeed categories are listed in Table 4.12. The OSM highway classes are labelled as shown in Table 4.13 and the OSM surface classes as shown in Table 4.14.

Numeric-0:	Numeric speed limit below 30.
Numeric-30:	Numeric speed limit of 30 or above but below 70.
Numeric-70:	Numeric speed limit of 70 and above.
Missing:	Missing speed tag.
Other:	Other than the above.

Table 4.12: Maxspeed categories.

Motorway:	motorway, motorway_link, trunk. trunk_link
Primary:	primary, primary_link, secondary, secondary_link, tertiary, tertiary_link, unclassified
Residential:	residential
Pedestrian:	pedestrian, living_street
Paths:	footway, bridleway, steps, path
Cycleway:	cycleway
Missing:	Missing highway tag.
Other:	Other than the above.

Table 4.13: Highway categories.

Paved:	paved, asphalt, concrete, concrete lanes, concrete plates
Paving_stones:	paving_stones, sett, unhewn_cobblestone, cobblestone
Metal:	metal
Wood:	wood
Unpaved:	funpaved, compacted, fine_gravel, gravel, pebblestone, dirt, earth, grass, grass_paver, gravel_turf, ground, mud, sand, woodchips, salt, snow, ice
Missing:	Missing surface tag
Other:	Other than the above

Table 4.14: Surface categories.

The training and test sets used were generated according to the methodology presented in the Method chapter in section 3.2.2.

4.4.2 RNN Implementation

The implementation uses the spell checker network structure. However, the five last characters of a name together with the maxspeed, highway and surface tags are used as input. The output is also changed to a Boolean to indicate an anomaly or not.

4.4.3 XGBoost Implementation

The implementation uses the same input and output structure as the RNN version except for the name. The name is instead classified as one of 23 suffix classes depending on the type of Way name. These classes have been tailored to Danish according to the data presented in Section 2.9 on compound words in Danish.

4.4.4 Results

The XGBoost implementation achieves an F_1 score of **0.91** on the test data. The RNN implementation achieves an F_1 score of **0.9**. However, when applied to the current OSM data set, the XGBoost implementation flags **20,531**/327,069 (about 6%) instances as anomalies whereas the RNN implementation flags **5,939**/327,069 (about 2%). These results were achieved by setting certainty requirements on the predictions required to flag an anomaly. For the XGBoost implementation, this limit was set to **0.95**. The limit for the RNN implementation was set to **0.75**. The certainty requirements were chosen by comparing different requirements between 50 and 95 percent and using the best one.

The confusion matrix for the XGBoost implementation is shown in Table 4.15 and the confusion matrix for the RNN implementation is shown in Table 4.16.

T	**71,070**	10,735	81,805
F	3,984	**77,746**	81,730
	75,054	88,481	**163,535**

Table 4.15: Confusion matrix for the XGBoost implementation.

T	**3**	81,802	81,805
F	2	**81,728**	81,730
	5	163,530	**163,535**

T	**72,212**	9,593	81,805
F	6,339	**75,391**	81,730
	78,551	84,984	**163,535**

Table 4.16: Confusion matrix for the RNN implementation. The left matrix shows results without suffix extraction and the right shows results with suffix extraction.

4.5 Missing Names

For nameless entities, the name tag should be omitted and the noname tag attached (noname=yes). The idea is to indicate that the street doesn't have a name with the noname tag. The absence of a name tag is increasingly used to indicate areas which need to be surveyed still.

To solve the smoothing scenario of a missing name – defined as a Way that is missing both a name and a noname tag – a basic graph-continuity-checker was implemented. The finder was implemented using algorithms/checks. The algorithm was never improved further than the baseline implementation due to time constraints.

The algorithm finds the missing name in two different scenarios where the first is a missing name between two edges if the two edges has the same name. This is often found in motorway-links for instance. The second scenario is the “small-tail”-case which means that we have a small tail that returns to the same node that it exited from. This is often found in urban areas where a small tail on a road leads to 1-4 houses. An example of this is shown in Figure 4.1.

Figure 4.1: Example of the small-tail case. The image shows a way with the name Damsgårdsvej. From this way, there are smaller ways leading up to buildings along the road that could possible also be tagged with the name Damsgårdsvej.

4.5.1 Results

Figure 4.2 shows the ratios of present and missing edges in the data set. The total number of edges with missing names in the data set is 6,066,646. The algorithm identifies 756,245 instances which corresponds to about 12,5%.

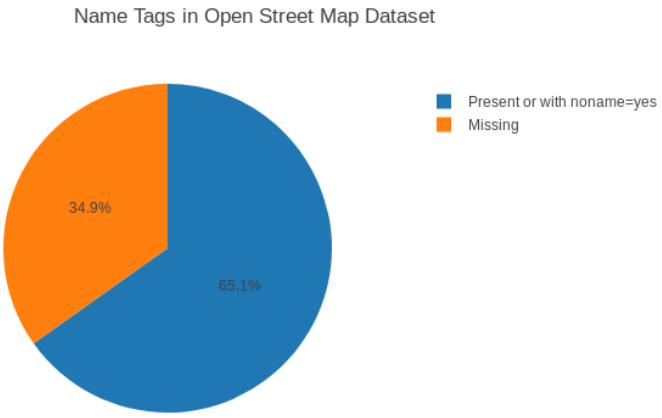


Figure 4.2: Ratio of present and missing names in the OSM data set from 2018.

We checked 50 random edges that were corrected by the algorithm and as shown in Table 4.17, around 70 percent of all changes were made correctly. We show a few of these cases visually in the appendix .1.

Actual Error	34
No error	16

Table 4.17: Manually controlled edges.

Chapter 5
Discussion

In this chapter, we interpret the results presented in the Implementation chapter and describe different approaches that were tried in order to improve those results. The first subsection briefly discusses the metrics used for evaluation. The following four sections cover the results for the WAY name spell checker together with improvements that were tried. Finally, the last two sections discuss the knowledge gained from implementing the subsystem for detecting anomalies.

5.1 Interpretation of Results

We have used two different metrics to evaluate the performance of the spelling correction: F_1 score and ACR. ACR is a metric for indicating how well the system is at predicting a correction for a word.

The F_1 score is, in our case, skewed since the score is averaged on each individual character predicted – not if the proposed word is correct. If we predict nine out of ten characters correctly in a word, the system scores a 90 percent correctness for this name when in fact, the system didn't perform a correct spell correction. We therefore invented the ACR score, which takes the resulting word from the spelling corrector into account and measures the ratio of correct predictions.

The F_1 score is useful as well however. The ACR is a quite rigid metric where information regarding how close the system was at predicting a correct word is lost.

5.2 Spell Checking Results

The spell checker performs better than the s/ss baseline with a higher total number of predictions as well as accurate such predictions. The evaluation on the crowdsourced test also show that the network has developed a language model since it corrects 41 instances.

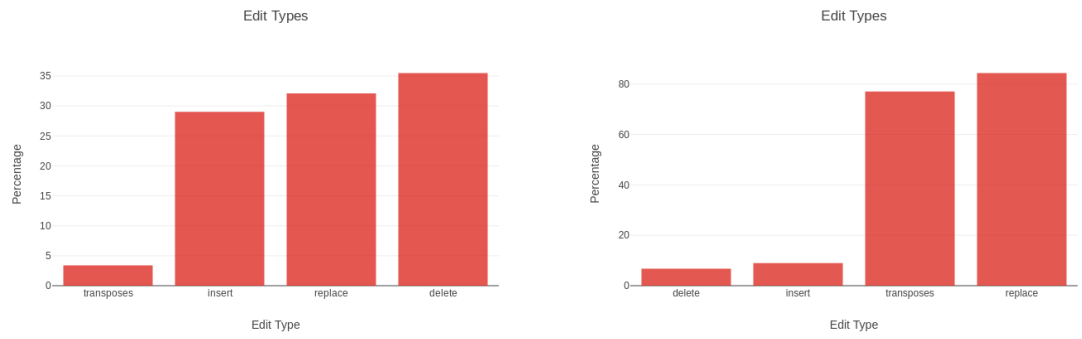
However, it also misspells 18 instances and fails to correct 153 misspelled instances. It's possible that this result could be improved with more and better data but it is also hard to draw any conclusions since the training data was created by inserting artificial noise. The network could have modeled the structure behind the insertion of artificial noise.

The test on OSM data show rather depressing results. The spell checker introduces 1653 errors while only correcting 7 instances and failing to correct 73. When comparing the errors in the OSM data with the training set, there are differences in the types of spell errors.

5.2.1 Difference Between Test and Reality

When analyzing the edits, we can see that transposes especially have a clear over representation in the crowdsourced data. We believe this is due to instructions for the participants to not edit their text. A transpose is an easy error to spot yet is never edited in our case. This would probably be the case in a real situation when a contributor adds the name to the OSM data set. The noise is therefore not as close to true spelling mistakes and the system will be weighted towards correcting a spelling error that is not as common as others.

See Figure 5.1 for a visualization of the errors contained in respective sets. If there for instance were more examples of deletion edits in the training data, this could possibly improve the performance of spell correction algorithm.



(a) OSM data

(b) Crowdsourced data

Figure 5.1: Edit type frequency.

5.2.2 Prediction Errors

In order to get an overview of the types of mistakes that the spell correction network makes, we investigated some erroneous corrections made by the system. The results are shown in Table 5.1. They indicate that some rare characters have not been present in enough examples. The examples with characters **'** and **-** show overfitting since they always produce the same output. We believe that bootstrapping the data set by the number of characters present during input and output would improve and stabilize the performance.

Our investigation shows the complexity of spell correction without context. In comparison, when correcting a word in a sentence, the surrounding words can be used to infer

more information about the word in question. Without context and with several spelling suggestions of the same edit distance available, a spelling correction system can't determine which suggestion is more probable. Table 5.1 shows two erroneous corrections made by the system that are both correct in the sense that it's correct Danish.

Type	Original	Erroneous Correction
Both correct versions	ovrevej	overvej
	emil bojsens gade	emil bossens gade
Failed corrections	hndelsvej	hlndelsvej
'→ j	christian ii's allé	christian iijs allé
- → b	nørre-herlevvej	nørrebherlevvej
. → random	i.g. smiths alle	img. smiths alle

Table 5.1: Errors introduced by the spell correction network.

In regular auto correction systems, more common terms are often misspelled and corrected. Davies and Gardner (2011) explain how frequency information has a central role in learning a language since the most frequent words are enough to understand the context and meaning of written and spoken sentences. Nation (1990) shows that the 4,000-5,000 most frequent words account for up to 95 percent of written text and the 1,000 most frequent words account for up to 85 percent of speech. This means that such a spell checker could achieve good scoring from being successful at correcting the most common words.

When it comes to Way names in Denmark, the most frequent words are not as occurring since the names are compounds of multiple words. This means that Way names are rare and consequently, it's hard to find data on where misspellings happen.

5.2.3 Misspelled Names in Estonia

The results for the misspellings in Estonia outdid the Danish by a sizeable margin. The F_1 score is a slightly more than 10 percent larger and the ACR is almost 20 percent better.

Analyzing the data, we found that there were only 19 edits of edit distance 1 made between 2014 and 2018 in the data set for Estonia. Out of these, 15 were unique. In the data set for Denmark there was around 300 edits of edit distance 1 and also 384 new ones from the crowdsourced data. Since the amount of different types of edits is much fewer in Estonian, it is probable that our Estonian spell checker had an easier time to learn correction of these misspellings. Another reason could be that the Estonian language might be less complex. These findings confirm our goal of having a modular spell corrector that easily changes language.

5.3 Improving Uncertain Predictions

This section covers the two approaches we implemented in order to improve the accuracy of the Way name correction system by targeting uncertain predictions.

5.3.1 Propagation

We experimented with creating edit distance 1 edits of a name if a character in the name proposed by the system was below a certain threshold in certainty. This was done for all character positions in the name where the uncertainty was below a threshold. These newly created versions of the name was then used as a new input to the system. The strategy didn't offer any improvements, the uncertain predictions even became even less accurate. Some instances that were correctly spelled became misspelled.

We believe that the explanation to this failed attempt is that the misspelling is a character that the system believes, with certainty, is correct. When manually investigating a few words, this was true. We believe that this strategy is almost impossible to do correctly as we can't know which character is incorrect when the uncertainty is not at the location of the misspelling.

5.3.2 SymSpell

Instead of propagating different versions of the name to the system, we applied SymSpell with the specialized look up to the name. The result was successful and often improved the F_1 score by a net of circa 8-12 percent in the number of corrections introduced. As mentioned, the system gets an additional hint on correctness from SymSpell. However, the specialized look up method relies heavily on heuristics and should be further worked upon in order to improve the accuracy of the spell correction.

5.4 Saturation of the Network

This section describes an approach to introduce a form of context to the spell checking network in addition to the name. We tried to add the contents of the max speed tag to the Way name spell checking network input in hopes of improving the results. Our hypothesis however, was that the experiment would result in no improvement since the network is built as a seq-2-seq. Each character is often followed by the correct character and the max speed tag would therefore be ignored.

Despite this, we wanted to try the approach. The results are shown in Table 5.2.

In conclusion, we believe that it is debatable if the system actually performed worse or if the difference is small enough to be considered a variance. We choose to consider the difference non-existent and believe that the network learned to ignore the max speed tag.

	ACR	ECI	F_1
BiRNN Autoencoder	0.6363	153	0.867
BiRNN Autoencoder + speed tag	0.5912	159	0.858

Table 5.2: Comparison of system with and without an appended max speed tag.

5.5 Reduction of Aggression

This section describes different ways of training the network to improve the score on the OSM data set.

We trained the same BiRNN Autoencoder network on different data sets to investigate what kind of impact the training sets have on the result.

BiRNN Autoencoder 1 is trained on the data that was presented in the Results chapter. The data is a combination of the corpus and the OSM data set and was artificially noised as described in the Method section. We introduced noise to fifty percent of the names in the data set.

BiRNN Autoencoder 2 is trained on the OSM data set without any artificially inserted noise. The data set was split in 2 parts, 2/3 to train on and 1/3 to test on.

BiRNN Autoencoder 3 is a combination of BiRNN Autoencoder 1 and 2. We combined the training data used in BiRNN Autoencoder 2, the crowdsourced data, and the corpus. We then applied noise to this training data. However, we did not apply as much noise as in BiRNN Autoencoder 1, we only applied five percent noise in comparison to fifty percent.

Table 5.3 shows that both BiRNN Autoencoder 2 and 3 improved their scores and we believe that the data set is one of the more important features to experiment on.

	F_1	ACR	EI	CI	FC
BiRNN Autoencoder 1	0.97	0.004	1653	7	73
BiRNN Autoencoder 2	0.98	0.011	93	1	42
BiRNN Autoencoder 3	0.99	0.167	5	1	3

Table 5.3: Performance of the different versions of the BiRNN Autoencoder on the OSM change data. FC = Failed Corrections, correctly identifying a misspelled word but applying an incorrect correction. EI = Errors Introduced, CI = Corrections Introduced, ACR = Accurate Correction Rate.

5.6 Weight of Suffix Extraction

This section covers the reason behind the failed speed limit network attempt. It also discusses the importance of suffix extraction for achieving better results in the case of the anomalies network.

We believe that the reason for the poor performance of the speed limit anomalies network is difficulties in finding a structure in a name with n-grams. By using suffix extraction, a form of tokenization, we managed to improve the results immensely in the case of the name and tag anomalies network. Our idea of suffix extraction, where we retrieve a suffix of five characters, originates from the information presented in Section 2.8 – in the Danish language, compound words are often used and street names are even more likely to be a compounds. This was shown to be true in Figure 2.12 where 61.4% of Danish road names end in “vej”.

In addition to the difficulties in finding connections between n-grams and speeds, we also realized that the first approach used speed and the full name as input while the second

approach used multiple tags and the name suffix. The conclusion we draw from this is that reduction of dimensionality is important for training the network. By extracting the suffix of the Way name and thereby reducing the input size by a order of 10, we improved our scoring and the system converged faster.

5.7 Name and Tag Anomalies

There was no time to manually retrieve name and tag anomalies from the OSM data set. Instead, we created an artificial data set where we introduced anomalies as described in the Method section.

Because of how this data set is generated, the credibility of the results is questionable, at least in a practical sense. We applied the system on today's OSM data set without any generated noise and noted that the RNN seemed more selective in its approach. Although, when inspecting the results further we found that many of the flagged anomalies probably weren't anomalies at all.

We want to put emphasis on the issue that this is a neural network – we don't know why the network has made a certain prediction. The system might actually not make use of the street name at all but learn to only find an anomaly by using the highway, surface and, max speed tags.

Chapter 6
Conclusions

The first conclusion that we draw from this thesis is that the missing names problem can be solved to a decent level using basic algorithms. We suggest corrections for 12.5 percent of all missing names using two basic algorithms which are correct in approximately 70 percent of the cases. We don't believe deep learning is required to solve the problem of missing names.

The second conclusion is that deep neural networks work well for finding anomalies in data. We show that even though the data is artificially generated, the network can identify anomalies. We believe that with more and better data, the system's performance on today's OSM data set could be improved.

The third conclusion we draw from this thesis is that the WAY name spell corrector is restrictive yet effective as it corrects more cases accurately than it misspells on the crowd-sourced data. Also, we improved the performance on the OSM data set immensely by reducing the noise in the training data. The spell checker is on par with our baseline S corrector as well. We have shown that the spell checker language is exchangeable by applying it to Estonian data.

Our implementation's main contribution is that while the system incorporates several known techniques, it does not require any manually annotated resources and is focused on the difficult instances of compound WAY names. We have not seen this implemented anywhere else.

Finally, we would like to emphasize that the results show that the neural network can learn a language model and correct misspellings. We also believe that data is a huge problem for training neural networks to do these kinds of corrections. Especially WAY data is difficult to retrieve in the vast amounts needed to train a network. Deep neural networks require a huge amount of data with both erroneous and correct instances where the former simply is not available in our case.

To summarize, we have shown the following

- Missing names can be filled to an extent using basic algorithms.

- Anomalies can be found using deep learning.
- A Way name spell corrector was implemented without language context and with the ability to change language by changing the training data. The performance never achieves the goal but with better, and more, data it could prove very useful as it has shown the ability to learn how to correct misspellings.

Chapter 7
Future Work

In this section, future improvements that can be made to the implementations are presented.

7.1 History log

By looking at the editing log for a WAY component, we believe that there's a possibility to know if the component has a greater chance of being correct. A component that has been edited more than x times should be more trustworthy since it has been under scrutiny several times. We could not include this in the thesis work because the calls to the REST API of OSM was too slow and our time was limited.

We trust that this could improve the system significantly. Perhaps a simple heuristic like "if the element has been edited more than five times, it is assumed to be correct" would be helpful to remove some uncertain predictions.

7.2 Improving the Data Sets

A better data set with more realistic errors and a larger variety of changes would increase the accuracy of the system. As mentioned, the difference between the crowdsourced data and OSM data was significant. We also showed that the system learns a language model and corrects errors that is presented to it. By having a large number of natural misspelled WAY names, the system should be able to correctly identify and correct more errors. The same applies to the data supplied to the anomalies network.

7.3 Meta Information

We believe that with more resources, planning, and time, a network could make use of meta information such as geographical knowledge and named entity-recognition to improve its correction rate. We found that when looking at street names, many streets - especially in towns and cities - are named after famous events, cities, or persons.

An example of this is "Jens Lillelunds Vej" where Jens Lillelund is a famous director in the automotive business. This way exists in Charlottenlund which belongs to Gentofte – where Jens Lillelund died. In this case, both the geographical location and the name belongs to a person that could have been identified with named entity-recognition.

Another example is "Mariebjergvej" which is located next to the graveyard "Mariebjerg Kirkegård". This means that by using the location of the way, we could find the graveyard and thereby cross-reference the name.

7.4 Alphabet

We also believe that the accuracy of the spell checker could be improved by using a full alphabet, especially if time is available to train the system over a long time. Including information such as uppercase letters means more information and possibly also more accuracy. Streets often capitalize the first letter in each word and this is sometimes missed by the editors. The system could also learn to difference capitalized and non-capitalized a and thereby differentiate the probability of a certain character to come thereafter.

More characters, such as numbers. which is currently not included would probably also benefit the system as the system currently performs poorly if a number is included in the Way name.

It has to be noted however that in this case of adding more characters, one has to find large amounts of training data with these characters and also bootstrap the data set. We have noted that the system underperforms for more rare characters such as ü. Also, some characters are always predicted when the input is a certain another character which indicates severe overfitting.

7.5 Data Set Balancing

We managed to improve the score by training the BiRNN Autoencoder network with data sets that had less noise introduced than the original version. It has to be noted that this also leads to reduced correction rate which means that the total amount of corrections are also reduced.

We believe that further balancing with the data set could lead to further improvements.

Bibliography

Apple Inc. 2017. Atlas - osm in memory. https://github.com/osmlab/atlas.

Apple Inc. 2018. Atlas checks - osm data integrity checks with atlas. https://github.com/osmlab/atlas-checks.

Pierre Baldi and Gianluca Pollastri. 2003. The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem. *Journal of Machine Learning Research* 4:575–602.

Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. 2012. Advances in optimizing recurrent networks. *CoRR* abs/1212.0901. http://arxiv.org/abs/1212.0901.

Leo Breiman. 2001. Random forests. *Machine Learning* 45(1):5–32. https://doi.org/10.1023/A:1010933404324.

Pete Chapman, Julian Clinton, Randy Kerber, Thomas Khabaza, Thomas Reinartz, Colin Shearer, and Rudiger Wirth. 2000. Crisp-dm 1.0 step-by-step data mining guide. Technical report, The CRISP-DM consortium. https://www.the-modeling-agency.com/crisp-dm.pdf.

Francois Chollet. 2017. https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html.

Fred J. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7(3):171–176. https://doi.org/10.1145/363958.363994.

Mark Davies and Dee Gardner. 2011. A frequency dictionary of contemporary american english: Word sketches, collocates and thematic lists. *Reference Reviews* 25(1):36–36. https://doi.org/10.1108/09504121111103191.

DL4J. 2018. Troubleshooting neural networks: Updater and optimization algorithm. https://deeplearning4j.org/troubleshootingneuralnets#updater.

Eclipse Deeplearning4j Development Team. 2018. Deeplearning4j: Open-source distributed deep learning for the jvm, apache software foundation license 2.0. http://deeplearning4j.org.

Wolf Garbe. 2015. Fast approximate string matching with large edit distances in big data. http://blog.faroo.com/2015/03/24/fast-approximate-string-matching-with-large-edit-distances/.

Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. 2000. Learning to forget: Continual prediction with lstm. *Neural Comput.* 12(10):2451–2471. https://doi.org/10.1162/089976600300015015.

Alex Graves. 2008. *Supervised Sequence Labelling with Recurrent Neural Networks*. PhD dissertation, Technische Universität München.

Gavin Harper and Stephen D. Pickett. 2006. Methods for mining hts data. *Drug Discovery Today* 11(15):694 – 699. https://doi.org/https://doi.org/10.1016/j.drudis.2006.06.006.

Geoffrey Hinton and Ruslan Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *ScienceMag* 313:504 – 507. http://www.cs.toronto.edu/~hinton/science.pdf.

Geoffry E. Hinton. 2018. Overview of mini-batch gradient descent. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735.

Andrej Karpathy. 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

KDnuggets. 2014. Poll: What main methology are you using? https://www.kdnuggets.com/polls/2014/analytics-data-mining-data-science-methodology.html.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR* abs/1412.6980. http://arxiv.org/abs/1412.6980.

Richard Maclin and David W. Opitz. 2011. Popular ensemble methods: An empirical study. *CoRR* abs/1106.0257. http://arxiv.org/abs/1106.0257.

Aurélien Max and Guillaume Wisniewski. 2010. Mining naturally-occurring corrections and paraphrases from wikipedia's revision history. In *LREC*.

Eric Mays, Fred J. Damerau, and Robert L. Mercer. 1991. Context based spelling correction. *Information Processing & Management* 27(5):517 – 522. http://www.sciencedirect.com/science/article/pii/030645739190066U.

I.S.P. Nation. 1990. *Teaching and Learning Vocabulary*. Teaching Methods. Heinle & Heinle. https://books.google.se/books?id=IQd_QgAACAAJ.

Peter Norvig. 2007. How to write a spelling corrector. http://norvig.com/spell-correct.html.

OpenStreetMap. 2017. Elements. https://wiki.openstreetmap.org/wiki/Elements.

OpenStreetMap. 2018. Maproulette. https://wiki.openstreetmap.org/wiki/MapRoulette.

OpenStreetMap, 2018. 2018. Editing standards and conventions. https://wiki.openstreetmap.org/wiki/Editing_Standards_and_Conventions.

Josh Patterson and Adam Gibson. 2017. *Deep Learning: A practitioner's approach*. O'Reilly, United States of America.

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

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323(6088):533–536. http://dx.doi.org/10.1038/323533a0.

Shashi Sathyanarayana. 2014. A gentle introduction to backpropagation .

M. Schuster and K.K. Paliwal. 1997. Bidirectional recurrent neural networks. *Trans. Sig. Proc.* 45(11):2673–2681. https://doi.org/10.1109/78.650093.

Michael Schuster. 1999. *On supervised learning from sequential data with applications for speech recognition*. Ph.D. thesis, Nara Institute of Science and Technology.

Jonas Sjöbergh and Viggo Kann. 2004. Finding the correct interpretation of swedish compounds, a statistical approach. In *In Proc. 4th Int. Conf. Language Resources and Evaluation (LREC*. pages 899–902.

Casey Whitelaw, Ben Hutchinson, Grace Young, and Ellis Gerard. 2009. Using the web for language independent spellchecking and autocorrection. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*. ACL and AFNLP, pages 890–899.

Wikimedia Commons. 2012. Crisp-dm process diagram. https://sv.wikipedia.org/wiki/Fil:CRISP-DM_Process_Diagram.png.

Wikimedia Commons. 2015. Rnn brnn. https://commons.wikimedia.org/wiki/File:RNN_BRNN.png.

Wikimedia Commons. 2017a. Random forest diagram complete. https://commons.wikimedia.org/wiki/File:Random_forest_diagram_complete.png.

Wikimedia Commons. 2017b. Recurrent neural network unfold. https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg.

Appendices

Appendix A
Missing Name Figures

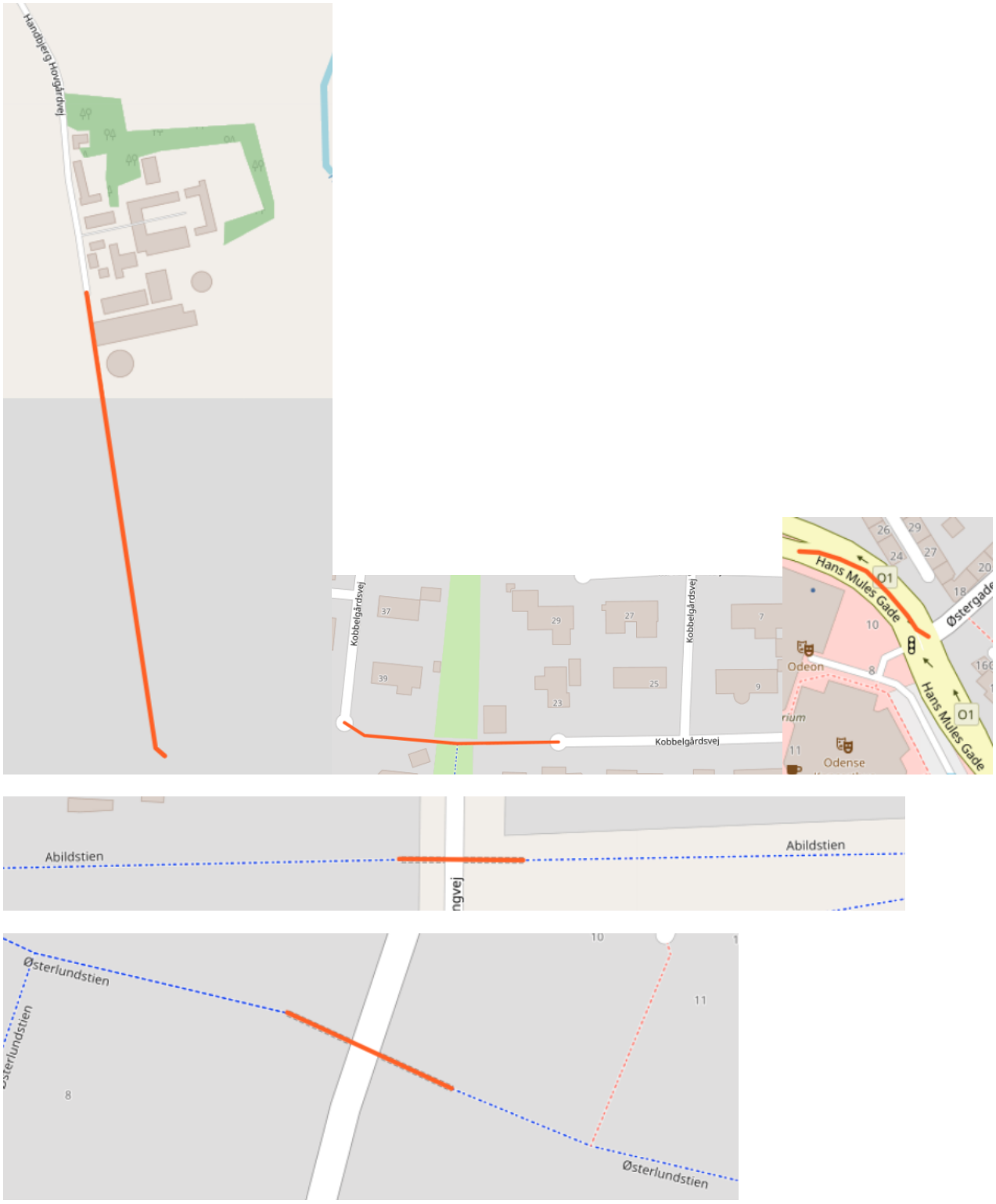


Figure A.1: Example of correct naming by Missing Names.

Figure A.2: Example of correct naming by Missing Names.

Figure A.3: Example of incorrect naming by Missing Names.

Appendix B
Generating Danish

As a side-experiment we learned the system to generate Danish. It was interesting to follow how the danish was generated and learned. To begin with the system overused the enter character, that is new line, but later learned to form sentences such as in

> 1tag iarti–t%e!oL! smtugu fn m9ødod i r deunennB

Further training it learned how to use common charcters instead of just randomly throwing them out

> gpnr,inao mssk ijd–emCtui tsaln e rr1 ae genoe dgvicedredeg gaafrasv r Eeadrro mori nnrr en Gdt

After a while it learned to spell danish but still couldn't use paranthesis and such correctly.

> Adugig Anhæ!jc hygge kåenn mid storfi!g kag 1545 bever !eg fag so! M!tek er neski åv Reg tar mer 1025 itbad ges 1Fkemaudsredersenis Fans, "deget Grandet ske!,ns tøn!r 115 jeg Ik!i!sig!en fid 2928) at 1..

And in the end the system got really well and actually wrote danish sentences correctly.

> Allerede har private udgave af Sveriges Universitet startede på den tyske hendes mor ved Kulturituativet af Arsham og hendes hovedvandgrundssøersJacki Bletvei fra alle baner ville gå fra dem af ..

It even learned to use paranthesis and citation correctly such as in.

> Ad Disservation (naturlig omfang) Niels Høyens and the Plantetysk Introduction Donago (1731–1912). Nr 4 blev åbnet i 1689-06 i Tromsdau, modelen fra 1917 og i 1830'erne og 1905 åbnede..
>
> Alubeney (født 7 februar 1955 i Værken University, England) er en dansk politiker fra Portugal i Karl Curvell Holstein Lars at CIA (1418–1755) var brudt til kuppet Feach Mendeloven 21metre i Kristiana..
