Analysis of Financial Transactions using Machine Learning

Adam Wamai Egesa

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-05
Analysis of Financial Transactions using Machine Learning
(An Application to Compute the Socio-ecological Impact of Consumer Spending)

Adam Wamai Egesa
ada07awa@student.lu.se

February 8, 2016

Master’s thesis work carried out at the Department of Computer Science, Lund University.

Supervisors: Pierre Nugues, Pierre.Nugues@cs.lth.se
Marcus Klang, Marcus.Klang@cs.lth.se

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

Many people want to know the socio-ecological impact of the goods they purchase. In this thesis, we describe a system that computes the socio-ecological impact of those goods by analyzing uncategorized financial transactions. The computation is made possible by extending a system that can compute socio-ecological impact from categorized transactions. The extension further includes visualizations on the system’s web GUI using AngularJS and extension of the system’s Node.js API.

To compute the socio-ecological impact the report describes a categorization service. To connect the service to the core system a RabbitMQ message queue was used. The service trained supervised machine learning models using Apache Spark’s machine learning library (MLlib) on a dataset containing about 2.4 million categorized transactions. This achieved a categorization accuracy of 82.9%.

The main focus for future work is to increase accuracy by using named-entity recognition and splitting up the categorization into two steps using multiple categorizers.

Keywords: machine learning, apache spark, mllib, mcc, financial transactions
Acknowledgements

Special thanks to my supervisor Pierre Nugues, assistant supervisor Marcus Klang and Dennis Medved at the Computer Science Faculty in Lund for many valuable insights and help throughout the thesis’ duration. I would also like to thank my colleagues Christopher Olsson, Robin Undall-Behrend, Mikael Karlsson and Kristian Rönn at Meta Mind for their feedback and collaboration. In particular thanks to Kristian Rönn for providing the data needed for the thesis and collaboration on the first version of the Scikit-learn implementation. Further thanks to the corporation Meta Mind for financial support of the thesis.
# Contents

1 Introduction ........................................... 7
   1.1 Background ........................................ 7
   1.2 Related Work .................................... 8
   1.3 Contributions ..................................... 8

2 Data Sources and Data Sets .......................... 11
   2.1 Category Taxonomies ............................... 11
      2.1.1 UNSPSC ....................................... 11
      2.1.2 ProClass ..................................... 12
      2.1.3 Merchant Category Code (MCC) .............. 12
   2.2 Data ................................................. 13
   2.3 Data on the Marginal Impact of Expenses ......... 13

3 Algorithms and Tools ................................ 15
   3.1 Machine Learning ................................. 15
   3.2 Message Queue – RabbitMQ ....................... 16
   3.3 Data Analytics Tools ............................. 17
      3.3.1 Scikit-learn .................................. 17
      3.3.2 Apache Spark and MLLib .................... 18
   3.4 Server – Node.js .................................. 18
   3.5 Web Client – AngularJS Framework .............. 19
   3.6 Database – MongoDB .............................. 19

4 Implementation ......................................... 23
   4.1 System Architecture ............................... 23
   4.2 Web Dashboards .................................... 24
      4.2.1 Initialization .................................. 24
      4.2.2 User input .................................... 26
      4.2.3 Components ................................... 27
   4.3 App Server .......................................... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1</td>
<td>Initialization</td>
<td>32</td>
</tr>
<tr>
<td>4.3.2</td>
<td>API</td>
<td>32</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Modules</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Classification Server</td>
<td>35</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Training workflow</td>
<td>35</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Testing workflow</td>
<td>37</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Prediction workflow</td>
<td>39</td>
</tr>
<tr>
<td>4.4.4</td>
<td>MLlib Wrapper Classes</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Scikit-learn implementation</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td>Results</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Scikit-learn results</td>
<td>43</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Vectorizers</td>
<td>44</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Stopwords</td>
<td>44</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Lowercase</td>
<td>44</td>
</tr>
<tr>
<td>5.1.4</td>
<td>n-grams</td>
<td>44</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Classifier options</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>MLlib results</td>
<td>45</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Experimental setup</td>
<td>45</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Data</td>
<td>45</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Feature Selection</td>
<td>46</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Results from entire dataset</td>
<td>47</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Results from 10% sample</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Computation Results</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>53</td>
</tr>
<tr>
<td>6.1</td>
<td>MLlib Results</td>
<td>53</td>
</tr>
<tr>
<td>6.2</td>
<td>Time and memory complexity</td>
<td>54</td>
</tr>
<tr>
<td>6.3</td>
<td>Impact Computation</td>
<td>54</td>
</tr>
<tr>
<td>6.4</td>
<td>Future Work</td>
<td>55</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>57</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Metrics</td>
<td>63</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Many people want to know the socio-ecological impact of the goods they purchase. In this thesis, we describe a system that computes the socio-ecological impact of those goods by first analyzing uncategorized financial transactions. This is achieved by extending a system that computes socio-ecological impact from categorized consumption to also work for uncategorized transactions.

Computing a person’s or organization’s socio-ecological footprint (e.g. $CO_2$ emissions, land use and water use) is not something new. However, very few organisations and even fewer individuals do this on a regular basis to better inform their economic decisions and how they affect the planet. The main reason for this is that it used to be a very data intensive and repetitive task to do these kinds of computation. This thesis automates these computations and visualizes the socio-ecological impacts together with the financial transactions to help make it easier for each actor to make informed decisions.

To compute the socio-ecological impact of an actor over a period of time, we need to know what economical activities the actor has been engaged with during that period. For each economic activity, we need to know the following facts:

1. **Category** of the product or service purchased. For example milk or gasoline.

2. **Impact factors** list that describes the marginal impact per unit purchased in different variables, e.g. $CO_2$, land use, water, nitrogen, quality-adjusted life years etc.

3. **Amount** purchased in American dollars or equivalent currency. We then use other models to compute the equivalent value of the currency spent in actual amounts of the category. For instance 1 $ spent on milk could be approximately computed to equal 1 liter of milk. Whereas other units are used appropriately.

As an example if we have bought 50 USD (amount) of clothes (category) and the $CO_2$ emission is 0.15 kg $CO_2$ / USD (impact factor), the $CO_2$ impact of our purchase is
50 \times 0.15 = 7.5 \text{ kg } CO_2. \text{ However, doing all of these computations for any actor requires an enormous amount of time and energy for several reasons. Here are four of them:}

1. Some products have several hundreds of impact factors, rather than just the one in our example.

2. Even a small actor like an individual or a small or medium-sized enterprise (SME) have hundreds of purchases per year.

3. Finding the right impact factors are hard since they are often buried in the academic literature or behind paywalls.

4. Finding suggestions that are actionable is very hard since the effectiveness of different actions depends on your already existing consumption patterns.

### 1.2 Related Work

Categorization of data sets containing hierarchies have been done before, such as in [8], where two approaches are compared. In recent years, machine learning (ML) has been used to categorize private and corporate entities’ expenses by private companies [10] in order to present where or what the entities’ expenses have gone towards. [10] for instance builds on one such system using supervised machine learning. Other than that there have however not been many examples of published research dedicated to classifying financial transactions to merchant category codes. In contrast some research relies on more complete financial information for each transaction. This information is stored by banks and made available, such as [19], which relies on over 40 fields in each transaction to categorize fraudulent financial transactions.

Like in [10] this thesis relies on only the three fields shown to bank customers. This thesis however expands on this to further detail the connection with another system that computes socio-ecological impact and further visualize this. Another framework is also used for the machine learning in the form of Apache Spark’s MLlib. Additionally this thesis focuses on data obtained from US and UK governmental institutions rather like in [10] which uses Swedish data sets.

### 1.3 Contributions

The thesis consists of three independent contributions:

I We explored and implemented how the system should receive and store each user’s bank transactions manually uploaded by users.

II We explored and chose what features and models that were best suited for the classification task at hand. Classifying a user’s bank transactions into categories that can be linked to the multi-regional input-output analysis (MRIO).

III We finished the loop by connecting a user’s expenses with the MRIO database collected to give an approximate socio-ecological impact.
Chapter 2 describes the data while chapter 3 outlines the algorithms and tools used for the contributions. Chapter 4 continues by outlining the implementation details of these contributions. Chapter 5 follows this with the results gathered from the classification, while the final Chapter 6 introduces the conclusions drawn from these results.
1. Introduction
Chapter 2
Data Sources and Data Sets

2.1 Category Taxonomies

2.1.1 UNSPSC

The United Nations Standard Products and Services Code (UNSPSC) is a taxonomy of products and services classification developed by the United Nations Development Programme and Dun & Bradstreet Corporation. The intention of the taxonomy is to include every type of service and product and be available for all types of organisations, including non-governmental, for-profit businesses and governmental organisations. Every classification in the taxonomy is encoded as an eight-digit decimal number, coded to a maximum of four levels of granularity. The top level is segment, followed in sequence by family, class and commodity. Code “00” is treated specifically to give the higher levels of the taxonomy, i.e. segment, family and class their own eight-digit codes. Table 2.1 shows some example codes.

This project uses version 15 which contains exactly 54,051 entries, where all types of products and services are hoped to be represented in some form. At the time of writing, there are also versions 16 and 17; each focusing on updating different areas along with ad-hoc changes. Version 17 in particular has more than 77,000 codes.

Table 2.1: Some examples from the UNSPSC taxonomy.

<table>
<thead>
<tr>
<th>Level</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment</td>
<td>44000000</td>
<td>Office Equipment, Accessories and Supplies</td>
</tr>
<tr>
<td>Family</td>
<td>44120000</td>
<td>Office supplies</td>
</tr>
<tr>
<td>Class</td>
<td>44121900</td>
<td>Ink and lead refills</td>
</tr>
<tr>
<td>Commodity</td>
<td>44121903</td>
<td>Pen refills</td>
</tr>
</tbody>
</table>
2. Data Sources and Data Sets

### Table 2.2: Some examples from the ProClass taxonomy.

<table>
<thead>
<tr>
<th>Level</th>
<th>Code</th>
<th>Heading</th>
<th>Clarification Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>140000</td>
<td>Clothing Products</td>
<td></td>
</tr>
<tr>
<td>Level 2</td>
<td>141300</td>
<td>Uniforms Products</td>
<td></td>
</tr>
<tr>
<td>Level 3</td>
<td>141400</td>
<td>Body Armour Products, includes personal protective equipment</td>
<td></td>
</tr>
</tbody>
</table>

#### 2.1.2 ProClass

The name ProClass originates from PROcurement CLASSification and it was developed by British authorities as an easier and more flexible version of UNSPSC. There is thus an official ProClass to UNSPSC mapping that maps each ProClass entity to an UNSPSC entity. Rather than the broad ambitions for UNSPSC, the expressed use-case for ProClass by the British authorities was however instead on doing spending analysis of where British authorities spend the taxpayers money. One reason stated for this has been to increase transparency, following the British Prime Minister’s call to publish each financial transaction over £500 from January 2011 [4].

Specifically ProClass targeted having only around 300 categories [2]. That limitation however proved hard to keep with version C15.1 containing over 500 [3]. Each entity in ProClass can further be divided into up to three levels, further associating six decimal digits to each entity. There is however not always a level two or three associated to a top level entity. An example of the taxonomy is shown in Table 2.2. All the entities which are sub-levels of each level 1 generally share only one two digit pairing but sometimes several. In the example, it can also be seen that the last 4 digits are in a shared namespace rather than directly denoting whether they are level 2 or 3. Where in the example the classification Uniforms has the code 141300 while one of its sub-levels have 141400 rather than 1413xx.

#### 2.1.3 Merchant Category Code (MCC)

A merchant category code (MCC) is a four-digit number the credit card companies like Visa [23] assigns to a business when it starts using cards as a form of payment. The number represents which category a merchant belongs to, e.g. “a car dealer” or an “airline company”. Sequences of codes are furthermore divided into higher order categories, effectively making MCC a two-level taxonomy. The lower level are sometimes here specific larger companies, such as specific car dealers.

Although the MCC taxonomy has two levels, for our purposes these levels were however considered to be the same level. Where we further abstract specific companies to a more broader category present in the taxonomy. This reduced the taxonomy from over 500 entries to 296 entries. We then further tweaked the MCC taxonomy to also include categories of itself. I.e. we divided the MCC entries into different categories by adding a new level, we then considered some of these categories to be irrelevant for private persons. As private persons are observed to not require many of the categories organisations may use.

We labeled these new MCC categories as level 1 and the regular MCC entries as level 2. The new levels are however not used for machine learning in any way. In total there
2.2 Data

Table 2.3: An example of the MCC taxonomy.

<table>
<thead>
<tr>
<th>MCC</th>
<th>Level 1 code</th>
<th>Level 1 desc.</th>
<th>Level 2 desc.</th>
<th>Level 2 description (long)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4722</td>
<td>3</td>
<td>Transportation</td>
<td>Travel Agencies</td>
<td>Travel Agencies</td>
</tr>
<tr>
<td>4723</td>
<td>3</td>
<td>Transportation</td>
<td>Package Tours</td>
<td>Package Tour Operators</td>
</tr>
<tr>
<td>4815</td>
<td>4</td>
<td>Utilities</td>
<td>Telephone Service</td>
<td>Masterphone-Telephone Service</td>
</tr>
</tbody>
</table>

There were two different sets of data. Both data sources came from governmental institutions in the USA. Both used the same classification scheme (MCC) and currency (USD). Both data sets were also structured to have the same format in a comma-separated values file. Values listed as the following:

1. **Description.** Plain text with upwards of 30 characters, including only one or a few words.

2. **Amount.** A number of type float. Always in USD currency.

3. **Date.** Formatted slightly differently between and inside different data sets. Four specific patterns were discovered to be able to catch all the different variations.

4. **MCC.** The MCC code for the given transaction.

2.3 Data on the Marginal Impact of Expenses

To know the marginal impact (or the so called “impact factor”) of a given expense or purchase, e.g. 0.15 kg $CO_2$ per USD of textiles consumed, we relied on previous research that we have put into a database. The types of research we rely upon can be categorized based on the type of methodology used in their investigations, see Table 2.4 for a compact overview of this. The impact factors used during the project was MRIO utilizing UNSPSC-encoded categories, where the future plan is to incorporate the more granular investigation methods LCI, LCA and EPD. So as to allow the user a more accurate account of the actual soci-ecological impact of different choices but also differentiate between products of the same category.
Table 2.4: The different types of impact factors that can be gathered.

<table>
<thead>
<tr>
<th>Type of research</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRIO</td>
<td><strong>Multi-regional input-output analysis</strong> (MRIO) is a method to calculate the socio-ecological impact of an economic sector within a given country, taking inter-dependencies between different national sectors of the global economy into account. By following the flow of money between different sectors using so called input-output tables, the environmental impact of a sector is calculated [13]. For example, information on fossil fuel inputs from the Norwegian energy sector to the Swedish car manufacturing sector can be used to investigate flows of carbon dioxide between the different economies.</td>
</tr>
<tr>
<td>LCI</td>
<td><strong>A life cycle inventory database</strong> (LCI) typically contains empirical data on the marginal impact of a specific production processes [11] (e.g. melding iron) or sub-components (e.g. a CPU) that can be used to conduct a life cycle assessment (see below). As the categories are more specific it can be viewed as a more granular variant of MRIO.</td>
</tr>
<tr>
<td>LCA</td>
<td><strong>An life cycle assessment</strong> (LCA) is a method of calculating the impact of a product by adding up the environmental cost through all production steps (e.g., from raw material extraction to manufacture, distribution and disposal) [11]. As rather than specific categories of product a specific product is investigated, LCA is a more granular variant of LCI.</td>
</tr>
<tr>
<td>EPD</td>
<td><strong>An environmental product declaration</strong> (EPD) is a type of LCA that has been done by a company in order to disclose information about the environmental impacts of one of its products [11]. EPD being a form of LCA, it is a more granular variant of LCI.</td>
</tr>
<tr>
<td>CEA</td>
<td><strong>A cost-effectiveness analysis</strong> (CEA) is a method used in health economics to calculate the marginal health outcome per dollar of a specific charitable program or health intervention [14] (e.g. distributing malaria nets) quantified in quality-adjusted life-years (QALY), which is the equivalent of 1 year of perfect health.</td>
</tr>
</tbody>
</table>

Chapter 3
Algorithms and Tools

3.1 Machine Learning

To compute the socio-ecological impact, the financial transactions needed to be categorized into about 300 categories using a categorization service. Categorization is a common use-case for machine learning, often referred to as classification.

Given the non-trivial nature of implementing and optimizing each of these algorithms the libraries scikit-learn and Apache Spark’s MLlib were used.

Classification with Machine Learning

The project uses using supervised machine learning to create models that can classify a particular type of data. Taking data that has a set of labels which it can take and teaching an algorithm how to recognize which set of data points belong to what label. Often only binary classification is needed. As we have over 300 labels, we however require multiclass classification, sometimes also referred to as multinomial classification, rather than binary classification.

Binary classification only classifies data into two different categories or rather if it belongs to a certain category or not, whereas multiclass classification refers to classifying data into more than two classifications. Algorithms are often first established as binary classification solvers but sometimes also lend themselves towards being extended as multiclass classifiers, whereas others are by nature binary classifiers. Even algorithms, that by nature are binary, can however still solve multiclass classification problem by reducing the multiclass classification problem into multiple binary classification problems through one of two strategies.

The first strategy is called one-vs.-rest in which a binary classifier is trained for each label, which classifies between the label and the rest of the labels. Further requiring the binary classifier to output a probability so that the classifier with the highest probability
can be picked, possibly resulting in ambiguity when several classifiers report the same probability.

The second strategy is called one-vs.-one and instead employs \(K(K - 1)/2\) binary classifiers given \(K\) labels, also called a K-way multiclass problem, making a binary classifier for every pairing of labels possible. Once run against a data point the label that most classifiers predict is chosen. This strategy notably does not require the confidence score from an algorithm, which is hard to extract from some algorithms.

**Logistic Regression**

Apache Spark implements binary logistic regression along with a generalization of the multinomial logistic regression through the one-vs.-rest strategy presented in the previous section. \(K-1\) binary models are thus created, given \(K\) labels. Each model is then compared against the first label, choosing the model which yields the largest probability.

**Support Vector Machines**

Apache Spark implements only a binary model as shown below, while scikit-learn implements a multinomial variant of the model. Something Apache Spark also plans to do in the future.

\[
L(w; x, y) := \max\{0, 1 - yw^T x\}
\]

**Naive Bayes**

Naive Bayes is a family of probabilistic classifiers which are all based on applying Bayes’ theorem on a set of features assumed to have strong independence. The project uses the multinomial version of Naive Bayes formula expressed below.

\[
p(x|C_k) = \frac{(\sum_i x_i)!}{\prod_i x_i!} \prod_i p_{ki}^{x_i}
\]

### 3.2 Message Queue – RabbitMQ

**Why Message Queues**

As the categorization service was done using scikit-learn in Python and Apache Spark’s MLlib Java implementation Node-based app server could not communicate natively with the categorization services in an easy way. The easiest solution found instead was to use message queues to facilitate the communication between the two services, where specifically only remote procedural calls (RPCs) were made from the app server to classify transactions and receive a response.

**Why RabbitMQ**

The message queue chosen was in the end RabbitMQ. This was primarily due to both recommendation and it being the most popular and established of the found alternatives.
The strongest alternatives were Redis and ZeroMQ. Where benchmarks seemed to generally favor RabbitMQ along with slightly more features. ZeroMQ on the other hand instead seemed to show much better benchmarks but radically harder to implement due to simply not providing many of the features RabbitMQ does provide, such as fault-tolerance, along with seeming to require more code to properly setup. Future iterations will however likely replace RabbitMQ with ZeroMQ in lieu of the possible performance gains.

**RabbitMQ Usage**

In RabbitMQ the following entities are of particular importance:

- **A producer** is a user application that sends messages.
- **A queue** is a buffer that stores messages.
- **A consumer** is a user application that receives messages.
- **An exchange** forwards messages from a producer to the appropriate queue.

RabbitMQ can be used to send and receive messages between a producers and consumers and by extension also perform RPCs.

Depending on the configuration, many producers can emit messages to the same exchange which in turn can emit messages to many different queues. Each queue can in turn also be sending the messages to one or more consumers.

In the case of RPCs, a client and server each serve as both producers and consumers. The client initiating a RPC by acting as producer to send a message to the server who in turn thus serves as a consumer. Upon receiving the message the server acts as a Producer and sends back another message to the client which thus has become a consumer.

There are no states built-in to RabbitMQ and messages can thus be thought to be forwarded in an idempotent manner. It is however possible to change the configuration of a RabbitMQ entity in run-time. [16]

## 3.3 Data Analytics Tools

To analyse the data, two machine learning libraries were used, scikit-learn and Apache Spark’s MLlib.

### 3.3.1 Scikit-learn

Scikit-learn is a Python wrapper library for machine learning algorithms introduced in [15] which uses the machine learning library LIBLINEAR, introduced in [5], for the machine learning algorithms used in this project. Scikit-learn was however primarily used as a smaller pilot project, consisting of only one large script that was slightly altered to make a proof-of-concept.
3. Algorithms and Tools

3.3.2 Apache Spark and MLlib

The second machine learning library used was MLlib [12], which is a machine learning library built upon the large-scale data processing engine Apache Spark version 1.5, introduced in [24]. Apache Spark and MLlib were written in Scala but also have Java, Python and R API’s. Through this API Apache Spark introduces Resilient Distributed Datasets (RDDs). These enables a user to associate data to the RDDs as they were stored locally, with the library then partitioning the RDDs into different partitions, where the partitions could be both locally and distributed over a cluster. This further enables relatively easy manipulation of big datasets that cannot exist on one machine. For MLlib, it specifically means that it can create models that are bigger than the available RAM on a computer, where it can either leak the models to disk or distribute the models over a cluster to continue working with them even if they do not fit to local RAM. Specifically meaning that the solution can scale horizontally over a large cluster for big data relatively easy, where big data was expected for an extension of the system when used to classify products or possibly when it scales to take in much more data from many more users. This was also the main reason it was chosen as the library to focus on over the more established and better performing alternative LIBLINEAR.

Accumulators and Broadcasters

To optimise these partition calculations, broadcasters and accumulators can be used. Broadcasters enables keeping read-only variables on each machine without copying it for each started task needlessly.

Accumulators instead enable a smooth way to accumulate values. Through a defined function the accumulators merge once the partitions are reduced to present a single result.

Pipelines using DataFrames

Since version 1.3 of MLlib and Apache Spark, DataFrames have been an alternative to simply using RDDs. Where data frames are a more abstract version of RDDs, following a specific tabular schema it allows operations to be scheduled on it. Using the schema, it then does an internal optimisation to the operations being done. To utilise this, MLlib created pipelines, transformers and estimators, where a DataFrame can be created and have a sequence of transforms and fitting of estimators scheduled through the pipeline. The pipeline can then be fitted itself. This fits all the estimators, turning them into transformers, creating a pipeline model. Internally the pipeline can use schema of the DataFrames to optimise how the operations are performed, where each operation is performed on a column basis. This further means that for instance feature operations like extractions can focus on specific parts of the data rather than all of it like scikit-learn does.

3.4 Server – Node.js

Node.js, or Node as it is often called, is server platform which is used to in the app server housing the API. Specifically Node is a JavaScript runtime built on the browser Google Chrome’s JavaScript engine called V8 [21]. As a side effect of running on JavaScript it is
single-threaded. As opposed to most other server frameworks and libraries which take on a multi-threaded approach to handle the various requirements a server typically has\[22\], such as being responsive to asynchronous requests made by many clients. What enable Node to still manage the requirements put on most servers is that it is also event-driven, using a non-blocking I/O model. Thus allowing it simply switch to doing something else once it has to either wait for input or output. I.e. input via incoming HTTP requests or both input and output from databases or message queues. This however also means that any processing-intense work does block the execution of any other tasks on the entire thread. The project uses the version of Node.js currently in Long-term support (LTS) for the app server, i.e. version 4.2.

Node being written in JavaScript leads to being able to share web client and server code more easily. While it can be easier for developers to switch between two as they use the same language. This is strengthened even further by utilising MongoDB as a database, where the objects already present in JavaScript can be stored, see Section 3.6.

3.5 Web Client – AngularJS Framework

The users’ web client is built using HTML, CSS and JavaScript. To help structure the code and add a lot of built-in features the framework AngularJS was used. AngularJS is a Model-View-Control framework in JavaScript for the browser [6]. The project uses version 1.4 for the web app while there is a release candidate for version 1.5 and a complete re-write of the framework labeled version 2 is in Beta. The framework enables routing in the browser. I.e. it can change the current URL of the browser and with it the view itself by manipulating the DOM using dependency injection to create isolated scopes which can be referred to as components. In these isolated scopes controllers are then further attached. Each controller then has access to other types of services such as $http for doing HTTP requests and custom services the developer can build to synchronize logic between the different components. There are further many other functions in the framework such as data filtering and two-way data binding.

3.6 Database – MongoDB

As the system needs to store and access data in real-time a database was used. Which in turn could index the data so that it could be easily retrieved once saved. The database used in the system was MongoDB, version 3.2 using the storage engine Wired Tiger. Version 4.3 of the node package Mongoose was used to explicitly model the data in MongoDB and make data validation upon insertion and updates easier, where Mongoose was further used to connect and query the MongoDB database from Node, though sometimes bypassed to access the official MongoDB driver for Node upon which Mongoose is built.

NoSQL – Document Database

As opposed to a regular relational, often called SQL, database which stores data as tables containing rows and columns, MongoDB stores data as collections containing documents,
where it can be classified as a document or NoSQL database. NoSQL only defining that it is not a SQL database whereas document databases in general store rich data similar to how MongoDB does it. Each document is stored as BSON (Binary Object Notation), a binary object format similar to, but more expressive than JSON (JavaScript Object Notation). Each query is however done using JSON which was derived from JavaScript and still mostly remain a subset of the JavaScript language. As the app server is further using JavaScript through Node.js there is no need to have a difference between the in-memory data structures and the data in the database. This is a key difference when compared to relational databases which store data only as tables and rows, or more formally, relations and tuples, leading to the object-relational impedance mismatch [7]. Though there like with MongoDB are many libraries that seek to remedy this it is possible to avoid the object-relational impedance mismatch even when using the MongoDB driver for Node directly (or bypassing Mongoose to do it). In combination with the other JavaScript technologies used it is easier for front-end and back-end developers to collaborate by sharing code and to switch between front-end and back-end development. The relative ease of development was the primary reason it was chosen.

### Performance and Scaling

In terms of performance there were various benchmarks showing wildly different claims for all of the different databases out there. As most had some kind of fault, such as an old version of MongoDB, strong bias or slightly particular use-case no consensus was made on definite performance trade-offs between databases other than the NoSQL database Redis likely being among the fastest with $O(1)$ insertion and access of data. With some other NoSQL databases such as HBase and Cassandra also generally perceived as faster, with SQL databases generally showing much slower or faster results depending on the benchmark. Though both SQL and MongoDB use B-trees to store data, resulting in $O(\log(n))$ access times, MongoDB handles relations completely differently, making for a harder time to compare the two technologies. HBase and Cassandra are further instead wide column databases while Neo4J is a graph database, all potentially with their own niche cases that fit certain datasets better, as explained in [17].

Common for NoSQL databases are however that they are generally considered better at horizontal scaling, i.e. scaling across many machines. Functionality that historically was either missing or very difficult on traditional SQL databases[17]. Newer versions of SQL are however quite much better at this. MongoDB in particular uses something called sharding to achieve this.

### CAP: ACID vs BASE

ACID–compliant databases conform to four principles:

**Atomic**: Everything in a transaction succeeds or the entire transaction is rolled back.

**Consistent**: A transaction cannot leave the database in an inconsistent state.

**Isolated**: Transactions cannot interfere with each other.

**Durable**: Completed transactions persist, even when servers restart etc.
There is however a spectrum to in which way a database may be ACID compliant. Typically SQL databases are fully ACID-compliant in that they can join queries to different tables or relations to ensure that the entire operation is ACID-compliant. MongoDB however is only ACID-compliant at the document level.

As opposed to this there are also BASE–compliant databases (Basic Availability Soft-state Eventual consistency), where MongoDB is thus BASE–compliant over multiple documents but ACID on single documents.

Related strongly to this is the CAP theorem[17]:

**Consistency:** All clients see the same data at all times.

**Availability:** If you can talk to a node in the cluster, it can read and write data.

**Partition-tolerance:** The database keeps it characteristics even should the cluster break into different partitions unable to communicate.

A database that is only on one machine is a typical example of a CA system, where it is either available or not depending on if the one machine is up, while partition-tolerance cannot apply and consistency can be assumed as there is only one place to write and read the data. When scaling up the CAP theorem however predicts that one can only have two of the three, with the caveat that one can only really choose between consistency or availability. As losing partition-tolerance implies that one can never have network failures which in practise is very hard or expensive. The need for a choice between consistency and availability can be traced from a short example in where a network is broken in two. Should one enforce availability, such as ensuring the serialization of an order, an identical serialization could be made in another part of the system. This then breaks consistency. If on the other hand you force consistency you can only do the orders on one part of the system, thus breaking availability. There are further some more caveats where you can mix and match availability and consistency. For example ensuring reads available on all systems along with consistency but no writes always being available. In terms of the CAP theorem MongoDB, Redis and HBase are CP (consistent and partition-tolerant) while Neo4J and Cassandra are AP (available and partition-tolerant).

**Why MongoDB**

The original reason for choosing MongoDB for the system was for the expected programmer productivity it would garner from it’s relatively simple model and fit with the rest of the JavaScript technologies used. As suggested in [17] it is a good idea to try different databases with real data and measure both performance and developer productivity. The project did however not have an evaluation of this character in its scope while it was rather deemed that the by far easiest solution was to build on top of the existing database, with the design to be able to replace certain static data with Redis in the future. What also played into the consideration was the popularity of the databases, where the mentioned databases were observed to at least be in the top 25 most popular databases in the past year[20].
Chapter 4
Implementation

This chapter will outline all the implementation details of this project.

4.1 System Architecture

Figure 4.1 illustrates the high-level components in the system and how they communicate with each other. The direction of the arrows further illustrates in what direction the data flows. As is seen there exists two types of clients, mobile apps and web dashboards. Both types of clients are essentially presented as the same application to the users. Some
4. Implementation

functionality is however absent in the mobile apps, such as some graphs and filters. This project continued the development of the already existing web dashboards but was not involved in the development of the mobile apps. The implementation of the web dashboards is however further discussed in Section 4.2 Web Dashboards.

There were further two types of servers, the app server and the classification server. The app server functions to both serve the web dashboards themselves to the browsers as well as serve the API used by both the mobile apps and web dashboards. As the web dashboards were essentially a super set of the mobile apps in terms of functionality, they also used a super set of the API compared to the mobile apps.

The app server further has the responsibility to manage all queries done to the MongoDB database. Read more about the configuration for the database in Section 3.6. It further loaded some configuration files through the file system, in the form of CSV files. Finally the app server managed the communication with the classification server through remote procedural calls using messaging queues. Specifically RabbitMQ was used as a message queue. Read more about implementation details of the app server in section 4.3.

The classification server in turn handled the classification of transactions. Where it could load, parse and train on transactions stored in CSV files. It could then further output the test set that could be used to test the system through the client. Read more about implementation details of the app server in section 4.4.

4.2 Web Dashboards

As is also visible from Figure 4.2, there are in total four different kinds of users. Private persons and companies are exactly that and have access to different components in general. There are however a few components they share completely, such as transactions. While they have similar but different dashboards. Contributors are thought to have a higher level of access, further being able to change the content visible for all other users. Administrators in turn have access to change anything in the server, including users.

The web dashboards are essentially just one big web app built on the JavaScript framework AngularJS. Figure 4.2 illustrates both the initialization of the web dashboards and how user input and data flows in the web application. Sub-sections 4.2.1 and 4.2.2 go into more details of what happens in each arrow of the flowchart. As is apparent from the Figure, the web app is dependent on using many different components. All of the components available in the web app are however not in the scope of this project. The components that however are in the scope of the project are shown in Section 4.2.3.

4.2.1 Initialization

1 Get HTML Transmits the index.html file through a HTTP GET request. Though the same content is always fetched the URL of this HTTP request determines which route is later used in step 5.

2 The CSS and JavaScript are fetched asynchronously with the CSS HTTP requests sent first to the server.
4.2 Web Dashboards

Figure 4.2: Flowchart of the web apps’ initialization and use of the REST API. The initialization steps are numbered one through eight. The remaining steps include calling and receiving responses from the REST API in addition to how the data, DOM and routes are managed.

(a) Get CSS transmits both the project’s CSS files, "app.css" and the CSS files available from all the dependencies, "vendor.css". When in development mode all the original source files are transmitted, while when in production a preprocessed version is transmitted instead, namely "app.css" and "vendor.css". Transmission are done through HTTP GET requests.

(b) Get JavaScript transmits both the project’s JavaScript files and the JavaScript files available from all the dependencies added with bower, i.e. "vendor.js". When in development mode, all the original source files are transmitted, while when in production a preprocessed version is transmitted instead. The project’s own source code is then consolidated under "app.js" with the dependencies sent as "vendor.js". The latter part visualized in Figure 4.2. Transmission are done through HTTP GET requests.

3 Load CSS. Once the CSS has been fetched, it is loaded as the CSS to be used for styling the page and taken into account when the browser renders each page.

4 Load JavaScript. Once the JavaScript has been fetched, it is loaded together with the DOM. It then starts running all the JavaScript files, starting with all the dependencies, so that they are available for the application code.

5 Route. "app.js" initializes the Angular part of the app once it has received a DOMContentLoaded event from the DOM. Specifically all Angular scripts are bound to the global scope so that they can be dependency injected continuously during the
application life-cycle. This includes the components which are further discussed in section 4.2.3. The details of the many third-party dependencies called are however considered to be outside of the scope of this project.

One script of importance is however the $routeProvider in Angular. The $routeProvider configures which JavaScript file should be run with which HTML template file given a route. When an URL address is entered into the browser, Angular extracts a route and then injects the HTML template into the DOM and runs the JavaScript file specified by the $routeProvider. Given for example the URL http://normative.io/company/transactions, the route /company/transactions would be extracted as the given route. In our implementation we then again call the $routeProvider in "component.js".

The $routeProvider is initialized in app.js to the global Angular scope. This effectively enables "component.js" to use the $routeProvider from the global Angular scope to register which HTML template and controller script should be associated to the component’s route. I.e. "component.html" and "component.controller.js" respectively. Angular then can thus then later on inject the correct HTML template and JavaScript script to the scope.

6 HTML path. The component controller is through Angular bound to a specific part of the DOM.

7 Run scripts. The component controller is executed on the component’s template HTML file.

8 Inject. The component’s HTML template and its controller script is injected into index.html, enabling it to be rendered for the user.

4.2.2 User input

User Input. As seen in Figure 4.2, the user input is first entered into the browser. The input then first travels through the DOM created from "index.html" into the DOM of the current component into "component.html" which in turn puts this into the "component.controller". This in turn causes component specific logic to run in response to the input. This could for instance be changing the data presented following a change in filter input fields or changing the route and URL of the web app as the user clicks in the app menu.

Request (JSON) and Response (JSON). The "component.controller" sends HTTP requests both when the script is first executed and in response to user input. The HTTP requests are sent to the app server which in turn always responds with an HTTP response. The data format used in the communication is JSON.

DOM and data manipulation. The "component.controller" does both various DOM and data manipulation unique to each component. This may be done at any point during the application life-cycle but is generally done at the start of the script when it is initially executed, as a response to a user input from the DOM or following a HTTP request or response.
4.2.3 Components

Below are descriptions for each of the components relevant to the project. Where previous sections in this chapter outlines where components fit into the architecture and flow of the application.

**App menu.** This component controls navigation to the different routes available. I.e. clicking on a link sends a new route to Angular which then maps it to the component clicked. The component associated with each route is always highlighted in blue. The app menu further changes depending on which role the user has, as they are considered separate applications to the user. Figures 4.3 and 4.4 show how private consumer and business users respectively see the app menu.

![App Menu](carl.png)

**Figure 4.3:** The app menu seen by private consumers.

**Toolbar.** This component houses the navigational bar shown at the top of the screen everywhere in the application. As the users see different web applications depending on their specific role (private consumer, corporation, contributor or admin), the toolbar must manage changing its title. The toolbar also houses links to sign in, sign up and sign out as well as the notification view, which is not covered in this project. The component further links to putting out the app menu on smaller screens. These features can be observed in Figures 4.5, 4.6, 4.7 and 4.8.
Figure 4.4: The app menu seen by businesses.

Figure 4.5: The toolbar when logged out.

Figure 4.6: The toolbar on a small screen when logged in as a company.

Figure 4.7: The toolbar on a small screen when logged in as a private consumer.

Figure 4.8: The toolbar on a wide screen when logged in as a corporation.
Company Dashboard. The company dashboard is shown in Figure 4.9 and displays the general impacts the company’s transactions have generated. It also displays two bar graphs for the current transactions stored in the database. The first graph shows them sorted by the 27 level 1 categories added for the MCC taxonomy. The second shows the transactions sorted by each individual entry in the MCC taxonomy. The user currently needs to manually press a button to recalculate the data each time the statistics change.

Transaction Sources. This component can be seen in Figure 4.10 and shows all the different transaction sets the user has uploaded, including date, number of transactions and file name. Further enabling the user to remove specific transaction sets that have been uploaded.

Transactions. By far the biggest and most complex component. Here the user can see all of his or her transactions as a table. It is further possible to sort the transactions by
any of the available level 1 and 2 MCC categories, date and amount in addition to a
free text filter. A more detailed view of the impacts, as a table, is available as a tab
next to the transactions. Both the tables’ rows for transactions and impacts can be
sorted by any column, while the tables themselves are automatically paginated to 10
rows per page. These features can be observed in Figures 4.11 and 4.12.

**Figure 4.11:** The view with a data table containing transactions in the first tab, along with filters for the transactions and a second tab containing the impacts. This view shows the first page.

**Figure 4.12:** The view with a data table containing transactions in the first tab, along with filters for the transactions and a second tab containing the impacts. This view shows the second page.
4.3 App Server

Settings. As seen in Figure 4.13, the user can change its password, country and currency. Changing the country further impacts how the impact calculation is done. The currency is considered to affect which currency the transactions are considered to be in.

Figure 4.13: The settings view available for all users.

4.3 App Server

At its core built a Node.js server. With the library Mongoose as the Object Data Model (ODM) used to connect to the database. The library ExpressJS was used to manage the routing of the application.
4. Implementation

4.3.1 Initialization

The scripts running during the initialisation of the app server can be seen in steps 1-8 seen in Figure 4.14. Below follows a more detailed explanation of what happens in each of those steps.

1 **Initialization.** The server is initialized, directly continuing to load configuration.

2 **Express configuration + middleware** The configuration such as whether it is in production or development mode is loaded and passed back to the app script.

3 **Init connection** The database connection is established.

4 **Dev or prod** If the application is determined to be in production mode a "prod" flag is passed on to production.seed. Otherwise a "dev" flag is passed on to development.seed. It further means choosing a specific database in storage. This means that development and production data are kept separate even locally.

5 (a) **dev** The "production.seed" file is started.
   (b) **prod** The "development.seed" file is started.

6 **Run seed** A seed file specific for production or development is executed. Meaning that the database is filled with mock data for the development version and without it for the production version. Along with using different database storage points.

7 **CSV file** Clears old mock data and parses new mock data from predefined CSV files.

8 **Parsed model (JSON)** Sends parsed model to "module.model" to save to database.

4.3.2 API

This section highlights the steps taken when the server receives a HTTP request, seen in Figure 4.14

R1 **Request (JSON)** A HTTP request is received from the endpoint through ExpressJS, allowing the route, data (in JSON format) and response callback to be forwarded. The request is thus forwarded to routes.

R2 (a) **Request (JSON)** In routes it is determined whether the request is calling the API. If so it is forwarded to the particular index.js file for the module the request is associated with.

   (b) **HTML file** when routes is determined not to be an API call, i.e start with "/api", then the HTML file "index.html" or other static content (CSS and client JavaScript files).

R3 (a) **Request (JSON)** When the request hits the index file for a module the corresponding route is found and the authentication middleware is triggered. Meaning that the request is passed along to the authentication middleware.
(b) **HTML file** If the API was not called static content described in 2b is instead forwarded.

**R4 User (JSON).** In the authentication middleware, the user’s object is fetched from the database. The header’s token is then compared to JSON Web Token stored for the user in the database. If they can be matched the user is authenticated, given that the user has the corresponding access right to access that route. A regular user can for instance not delete or add other users. If authentication fails a HTTP code 403 is instead sent.

**R5 Request (JSON).** When the authentication is successful the request, together with the fetched user object, is passed along to the appropriate function in the controller for that module that fits the given route.

**R6 Request (JSON).** In the controller, any sent parameter or data is extracted into their own variables. This is followed by a call to the service function of the module to do whatever needs done for the request.

**R7 - R10. Response (JSON).** Once the service function completes its action a response will be sent back. If an error occurred due to an error by the client or user a 4xx (i.e. 400-499) error code along with an explanation is sent back. If an error occurred due to some kind of apparent server bug a 5xx error code is sent back instead. Otherwise a 2xx code is sent along with the expected JSON data. This is first passed from the controller function all the way to the endpoint, at which point ExpressJS sends it back as an HTTP response.

### Database operations

This section highlights how database operations are done on the app server, see context in Figure 4.14.

**A1 DBM query** The service for a module saves or fetches data from the database through a query using the DBM Mongoose. Due to Node.js being non-blocking for I/O the function doesn’t block while it waits for the database to return an answer.

**A2 Driver query** The module.model is only a specification and thus the DBM query is essentially just translated into a Node.js driver query by Mongoose itself.

### CSV file uploads

This section highlights how CSV file uploads are done on the app server, see context in Figure 4.14.

**B1 Uploaded CSV file** The CSV file gets uploaded using a Node.js stream. For every 1000 objects parsed, as well as when the streaming has completed, are sent forward to be saved to the database.

**B2 Driver query** All objects are simply immediately sent to the database to be saved directly through the native Node.js to MongoDB driver. This process is currently only
done when uploading transactions but is implemented generally enough to easily be extended to other types of models.

Classification

This section highlights how classification is done through the app server, see context in Figure 4.14.

C1 JSON (AMQP) The service sends unclassified transactions in JSON format over the AMQP protocol to the RabbitMQ message queue.

C2 JSON (AMQP) The service receives the previously sent transactions classified by the classification server over the AMQP protocol in the same format as they were sent.

4.3.3 Modules

As the main part of the functionality on the app server are the actual modules seen in Figure 4.14, they are listed in this section for reference. Note that though there are more modules available in the project they were excluded as they are not part of the scope for this project.

CategoryData. Contains all the impact and consumption data for each UNSPSC category. Have service functions to calculate impact given a distribution of UNSPSC and values, i.e. how much of each UNSPSC consumed. Region is also required.

MccTaxonomy. A module for saving all the MCC taxonomy information as well as mapping between UNSPSC and the MCC taxonomy.

Region. A list of all the worlds countries and corresponding currencies.

Statistic. Saves any type of statistic in a format that fits nicely with a certain graph library in the web app. Currently only transactions over different categories is able to be calculated and fetched.

Transaction. The central point for operating on transactions. Primary concern of the module is to classify and save transactions. Which includes supporting uploading of CSV files.

TransactionImpact. This module takes transactions and maps their UNSPSC distribution and region using the MCCTaxonomy and Region module respectively. It then uses the CategoryData module to calculate the impact of all of these transactions. Where it is calculated in both a more detailed form and a more general form which only supports six types of impacts. These are then either summed before they are to a more total amount, or sent in its raw form to be sorted manually on the client side.

TransactionSource. This module essentially saves a timestamp and id that all transactions added at the same time by the same user gets associated with. Through this module it is then possible to delete and fetch transactions using only which transaction source they belong to.
4.4 Classification Server

The classification server is written in Java and uses Apache Spark’s MLLib to do classification of transactions. There are three different workflows with main methods that can be executed on the classification server. The first two are to train and test machine learning models, see sect. 4.4.1 and sect. 4.4.2 for more details. The third is to use trained machine learning models to dynamically call and using machine learning models to dynamically predict incoming transactions through the messaging queue RabbitMQ and return the transactions classified, so that it becomes the server in a remote procedural call. The app server acting as client, for more on predictions see sect. 4.4.3.

4.4.1 Training workflow

The steps in the training workflow can be seen in Figure 4.15, where either the data or event is stated in each labeled arrow. Below is a more intimate description of what happens in each step of the system.

1 Initialization Starts the main method in TransactionTrainer using the spark-submit script from the Apache Spark library, which also starts the Apache Spark run-time
with the system, loading the configuration set in the default configuration file. A singleton in the form of the enum SparkFields is then used to further configure and get a reference to the Apache Spark run-time. Where all the configuration set at run-time is also stored. Such as registering all classes that need to be serialized to the KryoSerializer, the serializer recommended by Apache Spark. Following this an instance of TransactionTrainer is created.

2 **Path** Once the TransactionTrainer has been initialized the hard-coded path to all the data is sent to CSVLoader.

3 **Data (CSV)** CSVLoader uses the path to the data to read it and parse it using the Transaction class. Creating a DataFrame containing columns description, payment date, amount, mccCode, year, month, day of year, day of month and day of week. The four first columns being the original data, while parsing an Apache Spark accumulator is used to gather all the labels into a HashMap.

4 **Parsed data (DataFrame)** When the data has been fully read and then parsed using the Transaction class, it is divided into training and test data and sent back to the TransactionTrainer class.

5 **Training data (DataFrame)** The TransactionTrainer class sends the training data to the FeatureExtractor.

6 **Feature pipeline** The FeatureExtractor takes all the columns of the transactions (except the label) and puts it through the feature pipeline. The feature pipeline is a MLlib pipeline, i.e containing different transformers and estimators. Specifically for the feature pipeline is that when it is fitted, it results into a vector of features for each transaction in the DataFrame. Other than the aforementioned columns already present in the DataFrame, the possible features to be included are TF-IDF or token counts from the description possibly in combination with n-grams and stop word removal. The FeatureExtractor further indexes the string codes of the labels, i.e. the MCC codes, but in a pipeline separate from the feature pipeline. As the feature pipeline needs to be used on unlabeled transactions as well, to gather the same type of features, it cannot also contain an operation that depends on the transactions having labels. Once the feature pipeline has been sent and the string indexing model created they are sent back to the TransactionTrainer.

7 **Params** The TransactionTrainer sends parameters to the ModelBuilder to configure the model(s) it wants along with their actual configuration.

8 **Models** One or more configured models are sent back to the TransactionTrainer.

9 **Feature pipeline** The TransactionTrainer uses the models received in the previous step, the feature pipeline received in an earlier step and the training data received even earlier to train the models, or rather fit the Estimator versions of the machine learning models so that they turn into transformer, i.e. fitted models. As transformers, or fitted models, they can then be used to classify transactions, after they have been put through the feature pipeline. Once the models have been fitted the feature pipeline is sent to the ModelSerializer.
10 **Fitted models** After the feature pipeline has been sent to the ModelSerializer the fitted models are sent as well.

11 **Object file** The ModelSerializer uses the KryoSerializer to serialize the feature pipeline object into an object file that is persisted to disk.

12 **Object file** The ModelSerializer uses the KryoSerializer to serialize the fitted model objects into an object file that is persisted to disk.

13 **Test data (RDD)** Once the serialization is complete the remaining data that was not used for training, i.e. the test data, is sent as an RDD to CSVWriter.

14 **Test data (CSV)** The CSVWriter writes all of the test data as labeled data in the most ideal form the transactions can have. Where dates for instance are set to have a uniform ISO format rather than one of four formats used in the original data.

### 4.4.2 Testing workflow

![Flowchart for how the data flows in the system when testing trained machine learning models.](image)

**Figure 4.16:** Flowchart for how the data flows in the system when testing trained machine learning models.
The steps in the testing workflow can be seen in Figure 4.16, where either the data or event is stated in each labeled arrow. Below is a more intimate description of what happens in each step of the system.

1 **Initialization** Starts the main method in PipelineTester using the spark-submit script from the Apache Spark library, which also starts the Apache Spark run-time with the system, loading the configuration set in the default configuration file. A singleton in the form of the enum SparkFields is then used to further configure and get a reference to the Apache Spark run-time, where all the configuration set at run-time is also stored. Such as registering all classes that need to be serialized to the KryoSerializer, the serializer recommended by Apache Spark. Following this an instance of PipelineTester is created.

2 **Path** Once the PipelineTester has been initialized the hard-coded paths to the persisted feature pipeline and models are sent to the ModelSerializer.

3 **Object file** The ModelSerializer deserializes the object file of the feature pipeline using the same library used for serialization, the KryoSerializer.

4 **Object file** The ModelSerializer deserializes the object file of the models using the same library used for serialization, the KryoSerializer.

5 **Feature pipeline** Once the feature pipeline has been deserialized by the ModelSerializer it is sent to the PipelineTester.

6 **Models** Once the fitted models have been deserialized by the ModelSerializer they are sent to the PipelineTester.

7 **Path** The hard-coded path to the location of the test data is sent to CSVLoader.

8 **CSV** The test data is loaded by CSVLoader and parsed using the Transaction class resulting in a DataFrame of the test data.

9 **Parsed data (DataFrame)** The parsed data in the shape of a DataFrame is sent back to PipelineTester.

10 **Metrics** Using the feature pipeline the features are extracted from the test data. The features are then sent through the fitted model, resulting in all of the test data getting a column of both the indexed and string form of a predicted label for each model. This is possible as the fitted model is actually a MLlib pipeline as well, including which had the string indexing model that was originally used to index the labels for training. That same model can then be used to return the label indexes to their original string form.

   Metrics are then calculated from the test data DataFrame, comparing the original label to the predicted label. Thus calculating f1-scores, precision and recall for each label in addition to an average of each of the mentioned metrics. All the metrics are then persisted or just printed to the system console.
4.4 Classification Server

4.4.3 Prediction workflow

The steps in the prediction workflow can be seen in Figure 4.17. Where either the data or event is stated in each labeled arrow. Below is a more intimate description of what happens in each step of the system.

1 Initialization Starts the main method in Predictor using the spark-submit script from the Apache Spark library, which also starts the Apache Spark run-time with the system, loading the configuration set in the default configuration file. A singleton in the form of the enum SparkFields is then used to further configure and get a reference to the Apache Spark run-time. Where all the configuration set at run-time is also stored. Such as registering all classes that need to be serialized to the KryoSerializer, the serializer recommended by Apache Spark. Following this an instance of Predictor is created.

2 Path Once the Predictor has been initialized the hard-coded paths to the persisted feature pipeline and models are sent to the ModelSerializer.
3 **Object file** The ModelSerializer deserializes the object file of the feature pipeline using the same library used for serialization, the KryoSerializer.

4 **Object file** The ModelSerializer deserializes the object file of the models using the same library used for serialization, the KryoSerializer.

5 **Feature pipeline** Once the feature pipeline has been deserialized by the ModelSerializer, it is sent to the Predictor.

6 **Models** Once the fitted models have been deserialized by the ModelSerializer, they are sent to the Predictor. Predictor then further creates an instance of RPCServer which starts up a connection to the RabbitMQ message queue while the program then starts listening after requests to the RabbitMQ queue.

7 **Unlabeled data (JSON)** Unlabeled data is received in the format JSON from the message queue RabbitMQ.

8 **Unlabeled data (List)** The unlabeled data is deserialized from JSON to a Java Collections List using the library GSON.

9 **Labeled data (JSON)** The predictor uses the previously fetched feature pipeline to transform the unlabeled DataFrame to add a feature column for each data point. The unlabeled data is then put through the fitted model pipeline, predicting each unlabeled data point based on the features. The result is two new columns, an indexed and one unindexed column of predicted labels. The labeled data is then serialized by getting a JSON representation of each transaction from Apache Spark methods and manually creating a JSON array. The serialized labeled data is then sent back to the RPCServer.

10 **Labeled data (JSON)** The RPCServer sends back the now labeled data as JSON to the same RabbitMQ queue, using the appropriate id so that the client making the call (typically the app server) can properly wait and receive a reply on the other side of the queue, creating a full remote procedural call loop. Following this step the RPCServer goes back to listening for new classification requests from RabbitMQ.

### 4.4.4 MLlib Wrapper Classes

To use all the algorithms in MLlib and at the same time build for the new DataFrame API, wrapper classes were made for the old MLlib API which used RDDs directly rather than DataFrames. SVMs, logistic regression and naive Bayes were wrapped like this. While decision trees and random forest are used using the new DataFrames API. See Figure 4.18 for the class hierarchy, connecting the MLlib API classes. Note that all of these models are defined and called by the ModelBuilder class. Where it is the models themselves that manage their own serialization.
Early on in the project a Python variant was tried out as an alternative solution. Using machine learning algorithms from Scikit-learn instead. This consisted of just a single script file that first trained the data, tested it and finally made it available as a predictor through the RabbitMQ message queue. One limitation was however that this implementation was limited to only using the description in transactions.

There were however in turn more features gathered from the descriptions. Specifically the tried features were:

**Word counter** Using the CountVectorizer the descriptions were turned into a sparse matrix holding the count of word.

**TF-IDF** Using the TfidfVectorizer the words in each description were turned into a sparse matrix of tf-idf values.

**Word hash counter** Using the HashingVectorizer the counts stored were instead stored against the hashed values of the String representation of the words.

**Stemming** This was added through parameters to the Vectorizers.

**Stop words** This was added through parameters to the Vectorizers.

**n-grams** Up 6-grams were tried. This was added through parameters to the Vectorizers.

Additionally the Scikit-learn implementation used three different models. Multinomial naive Bayes, support vector machines and logistic regression.
Chapter 5

Results

This chapter will discuss the test results from both the Scikit-learn implementation and the MLlib implementation. Both explained more fully in the Implementation chapter.

The metrics measured in the project follow from the definition made in MLlib for multiclass classification\([1]\). See the Appendix A for a detailed definition or \([1]\).

5.1 Scikit-learn results

The scikit-learn implementation was carried out as a pilot study, using only 10% of the final dataset used for the MLlib implementation. The size of the dataset was 11.9 MB containing about 250 thousand transactions. Hence it was about 10% the size of the final dataset. The best results obtained for each algorithm using scikit-learn on this smaller dataset is shown in Table 5.1.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Weighted f1-score</th>
<th>Weighted precision</th>
<th>Weighted recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>support vector machine</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
</tr>
<tr>
<td>naive Bayes</td>
<td>84%</td>
<td>85%</td>
<td>85%</td>
</tr>
<tr>
<td>logistic regression</td>
<td>81%</td>
<td>86%</td>
<td>80%</td>
</tr>
</tbody>
</table>

Table 5.1: The best results for each algorithm in terms of weighted f1-score, precision and recall. Using regular counts of words with HashingVectorizer, stemming and bigrams of the description as features.
5. Results

Table 5.2: N-grams results when using support vector machines.

<table>
<thead>
<tr>
<th></th>
<th>words</th>
<th>bigrams</th>
<th>trigrams</th>
<th>quadgrams</th>
<th>5-grams</th>
<th>6-grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>89.5 %</td>
<td>90.0 %</td>
<td>89.4 %</td>
<td>90.1 %</td>
<td>90.1 %</td>
<td>90.1 %</td>
</tr>
</tbody>
</table>

5.1.1 Vectorizers

CountVectorizer, simply keeping a simple count of all the words performed better than both TfidfVectorizer and HashingVectorizer by at least 1% for each of the three models. I.e. SVM, LR and MNB. TfidfVectorizer meanwhile yielded the same results as HashingVectorizer.

5.1.2 Stopwords

Adding stopwords for English only served to slightly decrease the accuracy.

5.1.3 Lowercase

Not converting data to lowercase did not change the overall accuracy at all. It did however also seem to produce a less varied response where the 10 data point example that previously contained many different mappings all got stuck with the same. The assumption was thus made that using lowercase would scale better and as all else seemed equal that was chosen.

5.1.4 n-grams

Adding bigrams improved performance while trigrams decreased the results slightly for the test data. Adding quadgrams balanced test results performance and increased it from bigrams with about 0.5%, from 89.5% to 90.0%. Increasing to n-grams with n to five or six yielded no noticeable improvement in accuracy (less than 0.1%), it only increased the training time substantially. See Table 5.2 for a table of the results.

5.1.5 Classifier options

Increasing the iterations from 5 to 50 increased the training time tenfold and increased accuracy by about 1% for SVM but decreased just as much for Logistic Regression. Penalty options l1 and elasticnet dramatically decreased accuracy by over 5% for both SVM and Logistic Regression.

The alpha value constant that multiplies the regularization term was 0.0001 by default for all three models. Results of changing the regularisation term can be seen in table 5.3.
Table 5.3: Results for support vector machines when tuning the regularization term.

<table>
<thead>
<tr>
<th>Regularization term</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0001</td>
<td>90</td>
</tr>
<tr>
<td>0.0003</td>
<td>89</td>
</tr>
<tr>
<td>0.001</td>
<td>87</td>
</tr>
<tr>
<td>0.003</td>
<td>85</td>
</tr>
<tr>
<td>0.01</td>
<td>83</td>
</tr>
<tr>
<td>0.03</td>
<td>82</td>
</tr>
<tr>
<td>0.1</td>
<td>68</td>
</tr>
<tr>
<td>0.3</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 5.4: The hardware used to get results from the MLlib implementation.

<table>
<thead>
<tr>
<th>Type</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-1620 v3, Quad Core, 3.5GHz, 10MB, 22nm</td>
</tr>
<tr>
<td>RAM</td>
<td>32 GB DDR4 ECC</td>
</tr>
<tr>
<td>SSD</td>
<td>60 GB space available</td>
</tr>
</tbody>
</table>

5.2 MLlib results

5.2.1 Experimental setup

The results from MLlib were run on a server running the hardware in table 5.4. Apache Spark was configured to use 24 partitions by default while the DataFrames were repartitioned to use 24 partitions precisely prior to fitting and training to ensure this was the case.

5.2.2 Data

The data used to evaluate the MLlib implementation was 120.9 MB and contained exactly 2,423,016 transactions. Evaluating the data however showed that it was both skewed and completely missing some of the available categories. Where only 267 of the 296 categories could be found in the data. Due to the data being so skewed sampling smaller samples of the data also lead to losing even more labels. Table 5.5 shows the decline of labels found with smaller sampling of the data. While Figures 5.1 and 5.2 shows how all the transactions look when uploaded to the company dashboard and thus partitioned over level 1 and level 2 categories respectively.
5. Results

Figure 5.1: The transaction data used for training and testing uploaded and visualized in the company dashboard as level 1 categories.

Figure 5.2: The transaction data used for training and testing uploaded and visualized in the company dashboard as level 1 categories.

5.2.3 Feature Selection

**Description** The baseline features used the "description" input to create features TF-IDF. Stopwords and n-grams were then further tested individually but yielded no difference in the results. As such they were not used.

**Amount** Further tests also used the "amount" input of each transaction, tested both with and without scaling and normalization using the $L^1$ norm respectively. All variations damaged f1-score, recall and precision heavily, reducing by more than 20 points.

**Payment date** All the date features were extracted and tested together, i.e. year, month, day of year, day of month and day of week. This however also heavily hurt the results.
Table 5.5: The metrics weighted f1-score, recall and precision from testing with 10% of the entire dataset on the model produced from training on 90% of the entire dataset. "-" mean the test failed, while training time and testing time is also included.

<table>
<thead>
<tr>
<th>Model</th>
<th>F1-score</th>
<th>Recall</th>
<th>Precision</th>
<th>Training time</th>
<th>Testing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NB</td>
<td>82.92%</td>
<td>82.65%</td>
<td>84.27%</td>
<td>30.23 seconds</td>
<td>41.56 seconds</td>
</tr>
<tr>
<td>RF</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

by reducing f-score, recall and precision to less than 40%.

5.2.4 Results from entire dataset

Four models from MLlib were tried to be trained on the full dataset, yielding the results of table 5.5. There it is seen that the naive Bayes model was the only model the hardware managed to run through the entire dataset. Specifically the three other models failed due to running out of available memory. Where they used up all the RAM and then leaked the remaining memory needed to create the models on to the disk space on the SSD until it got filled up, where upon they all crashed. The results from training the naive Bayes model can further be gleamed from the following three Figures 5.3, 5.4, 5.5 which shows the count of each of the categories along the x-axis. While each Figure further shows how high its corresponding metric is along the y-axis. Showing the metrics f1-score, precision and recall respectively.

5.2.5 Results from 10% sample

A 10% sample of the dataset was also tried on all the models, yielding the results in table 5.6. Again here only naive Bayes completed its execution. The remaining crashing after between 30 minutes to three hours.

The models, that were not naive Bayes, were thus ultimately only successfully executed on a data set with a mere 1000 transactions. This small subset however contained only 37 of the total categories and as such proved only as a proof-of-concept that they could actually train on data.

The results from training the naive Bayes model can further be gleamed from the following three Figures 5.6, 5.7, 5.8 which shows the count of each of the categories along the x-axis. While each Figure further shows how high its corresponding metric is along the y-axis. Showing the metrics f1-score, precision and recall respectively.

5.3 Computation Results

A randomised sample of 1000 transactions was selected from the test data. Worth noting is that the dataset sampled turned contained 64 unique categories (MCC). Computing the socio-ecological impacts using the real labels of the dataset and comparing that to the
Figure 5.3: The count and f1-score of each category for training part of the entire dataset.

Figure 5.4: The count and precision of each category for training part of the entire dataset.

impacts generated from using the labels generated from using the categorization service. Specifically the client was slightly modified to be able to display representations of the
5.3 Computation Results

Figure 5.5: The count and recall of each category for training part of the entire dataset.

Figure 5.6: The count and f1-score of each category for training part of the 10% sample of data.

impacts from different transaction sets in CSV format. The result was that the transactions still generated the same 78 impacts but in total generated an average of 85.5% of
Table 5.6: The metrics weighted f1-score, recall and precision from testing with 10% of a 10% large sample of entire dataset on the model produced from training on 90% of the aforementioned sample. "-" mean the test failed, while training time and testing time is also included. NB stands for multinomial naive Bayes, RF for random forests, DT for decision trees and LR for multinomial logistic regression.

<table>
<thead>
<tr>
<th>Model</th>
<th>F1-score</th>
<th>Recall</th>
<th>Precision</th>
<th>Training time</th>
<th>Testing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NB</td>
<td>77.53%</td>
<td>75.55%</td>
<td>84.76%</td>
<td>19.73 secs</td>
<td>10.11 secs</td>
</tr>
<tr>
<td>RF</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 5.7: The count and precision of each category for training part of the 10% sample of data.

the original impact values, where the variance between how much each impact output in comparison to the original landed on 4.73%. See figure 5.9 for more details on how the different impacts fared.
Figure 5.8: The count and recall of each category for training part of the 10% sample of data.

Figure 5.9: The value in percent for all of the impacts generated from the 1000 transactions sampled using their real categories compared to when using the categorization service instead.
5. Results
Chapter 6
Conclusions

Scikit-learn Results

The results received from scikit-learn was not completely comparable to MLlib as they used slightly different datasets. However as the difference was so significant both at comparable sizes of data and even when MLlib had remarkably more data the results can likely be attributed to scikit-learn using a better model.

Though the best model by far was SVM, having some kind of measurement on when the model thinks it’s uncertain fits a good business case. As the application does want to avoid showing faulty predictions to a degree and rather probe for the user to categorize if there is a certain level of uncertainty. As such SVM would at the very least need to be used in tandem with either or both of LR and NB.

Regardless of model selection a number of features would however be useful. Where CountVectorizer rather than TF-IDF from the words in the description is the starting point. Avoid stopwords but use n-grams. Specifically quad-grams would be used if efficiency was not a factor. However, as the difference between bi-grams and quad-grams was only 0.1% it is not worth the extra added processing that would require. Both in training but also as a pre-process step when doing predictions.

6.1 MLlib Results

For MLlib, the only classifier that could actually run the data without further configuration was Multionomial Naive Bayes at a weighted f1-score of 82.92%. It was thus without a doubt the best classifier. Further tuning might however enable the other classifiers to at least work for smaller amounts of data with the current hardware. It is for example possible to tune the convergence and maximum iterations of logistic regression to be slightly lower than what it was. Otherwise it is also possible to either add a bigger hard drive or more RAM to the current hardware, replace the current server with a better one or setup a cluster.
with more servers. The latter option is one of the greater pros for Apache Spark MLlib as opposed to scikit-learn. As the data grows it might be the most economical way to go about it, regardless.

The results for the various categories served to further prove what benchmarks have shown before, that even though more data leads to higher f1-scores, eventually there comes some diminishing returns on that [9]. Also specifically that more than 100 instances, preferably many thousands are needed for good results. Whereas many categories in the data have much less.

There were however a few categories that did not perform well despite larger amounts of data, thus further bringing down the curve. Exactly what did depended upon was not directly visible from the results gathered. It may however for instance be that it gets confused with one or more other categories a whole lot for some reason.

### 6.2 Time and memory complexity

As explained briefly in Sect. [3.1] all models used in the project increase linearly in time complexity as more labels are added. What is more alarming is however that the memory usage also scales at the same rate. This is assuming a linear increase in time complexity for binary models and defining the data as "n" and the categories regular level 2 MCC as "K" see Eq. [6.1] given the further assumption that the level 2 MCC are evenly distributed across the level 1 MCC. This thus yields the time complexity seen in Eq. [6.2].

Thus looking at the memory complexity it is easy to see why so much memory could be used even though only 120 MB was used. As given around 300 labels, one thus needs upwards of 300 times the space of the data in memory and disk on top of the overhead already in place.

\[
\begin{align*}
    n &= \text{data}, \quad c = \text{level 1 MCC}, \quad K = \text{level 2 MCC} \quad (6.1) \\
    O(Kn) &= \quad (6.2)
\end{align*}
\]

### 6.3 Impact Computation

Ideally the impact computation would have compared the exact socio-ecological impact from every single transaction available and compared that to how this compared to what the system actually produced. Comparing the actual socio-ecological impact from all transactions would however require either more extensive research such as doing actual LCA or EPD investigations on all the products purchased or finding data which also happened to have this. That was however not available for the project and was ultimately out of scope because it would primarily have served to evaluate the impact computation service itself.

Instead, the effect of the categorization service on the impact computation service was evaluated. The evaluation was however limited in scope, covering only 1000 transactions with only about 20% of the categories. Further coloring the impact computation was the amount of each transaction. As the impact computation itself is quite complex, involving many impacts with highly varying values, the results are insufficient to reach a definite conclusion to actually explain them. It is however evident that for some reason the overall
trend among all the impacts from the transactions is a slightly downward trend of about 20%.

Regardless the overall goal of the project was achieved. Contributing with integrating a categorization service and an UI where a user can see an approximation of the socio-ecological impacts of their transactions in addition to filtering the transactions to distinguish the impacts of individual transactions.

6.4 Future Work

Multi-level Approach

An alternative to the current approach would however be to split up the categorization into more steps, like described as one approach in [8]. Specifically one could split it up into the two levels we have already defined for the MCC taxonomy. Given the prior assumptions we would then get the time and memory complexity in Eq. (6.3) for level 1 and (6.5) for level 2 when training models. Further Eq. (6.4) when predicting. As with multiple binary models used each one of them need to be when doing predictions. Predictions would thus go faster, while it would allow the system to utilize hardware that is bottle-necked by memory as one could simply train less models at a time.

\[
O(cn) \quad (6.3)
\]

\[
O(Kn/c) \Rightarrow O(Kn) \quad (6.4)
\]

\[
O(cKn/c) \Rightarrow O(Kn) \quad (6.5)
\]

Splitting up the model into two layers might also help in the cases where model is insecure about the second level, or even about the first level. As then the user can still get the level 1 categories the model think are rated most likely and start from there. Rather than having to go through all of the possible close to 300 categories.

It is also possible that there is some kind of similarity between codes grouped under a level 1 category, thus perhaps compensating slightly for categories that just don’t exist in the data. Thus causing the model to misinterpret it as another category. Luckily there are also no level 1 categories missing completely and only one with less than 1000 at 824.

As described in [8] we could instead introduce more labels, where we would in this case simply double the data used, using the level 2 categories for the new half.

Named Entity Recognition

As many transactions and even some MCC codes contain named entities it would be likely be a good approach to do named entity recognition (NER). Where POS-tagging could also be used to both add more information and aid with pointing out what words actually are named entities. There would secondly also need to be some kind of database to compare and actually recognize these named entities. Here a wide variety of sources could be used, WikiData [https://www.wikidata.org] is one strong contender.
Database integration

Given that the system does come into use, integrating the users transactions through the database would be a natural step to getting more data. Where the thought is for the users to actively participate in categorizing data. Or rather verifying that transactions are labeled correctly through active learning \cite{18}. The correctly labeled transactions can then later be looped back and be used for training the model and potentially filling the missing category gaps.

More Advanced Analytics

Due to the sheer volume of labels there was no confusion matrix made. A next step could be to find ways to more easily visualise very large confusion matrices. Splitting the training into several smaller models would however also make the confusion matrices much smaller and thus alleviate much of these issues. Using a confusion matrix would in turn enable activities like deciphering why a few of the categories performed badly. Where one could then observe what labels the model was labeling the transactions instead, from which a new feature could possibly be used do disambiguate the overlapping categories.

Other analytics that needs to improve in the future are how the impact computation results compare to the actual results, where one would ideally like to see how the impact computation vary depending on what category the transaction is and not just for a total set of transactions. This can then be combined with the confusion matrix to actually see how each category is likely to swing rather than have to use an actual test set. Thus further reducing the bias of the test set, where the amounts could be completely disconnected from the comparison.

Usability Testing

Usability testing on users are planned for the future to actually determine the usability this subset of the system has and thus determine to what degree this projects contributions could actually help a user with analysing their financial transactions.
Bibliography


Appendices
Appendix A

Metrics

This appendix defines the various metrics used to specify the results.
Define the class, or label, set as

\[ L = \{\ell_0, \ell_1, \ldots, \ell_{M-1}\} \]

The true output vector \( y \) consists of \( N \) elements

\[ y_0, y_1, \ldots, y_{N-1} \in L \]

A multiclass prediction algorithm generates a prediction vector \( \hat{y} \) of \( N \) elements

\[ \hat{y}_0, \hat{y}_1, \ldots, \hat{y}_{N-1} \in L \]

For this section, a modified delta function \( \delta(x) \) will prove useful

\[ \delta(x) = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{otherwise.} \end{cases} \]

Confusion Matrix:

\[
C_{ij} = \sum_{k=0}^{N-1} \delta(y_k - \ell_i) \cdot \delta(\hat{y}_k - \ell_j) \left( \begin{array}{cccc}
\sum_{k=0}^{N-1} \delta(y_k - \ell_1) \cdot \delta(\hat{y}_k - \ell_1) & \ldots & \sum_{k=0}^{N-1} \delta(y_k - \ell_1) \cdot \delta(\hat{y}_k - \ell_N) \\
\vdots & \ddots & \vdots \\
\sum_{k=0}^{N-1} \delta(y_k - \ell_N) \cdot \delta(\hat{y}_k - \ell_1) & \ldots & \sum_{k=0}^{N-1} \delta(y_k - \ell_N) \cdot \delta(\hat{y}_k - \ell_N) 
\end{array} \right)
\]

Overall Precision:

\[
PPV = \frac{TP}{TP + FP} = \frac{1}{N} \sum_{i=0}^{N-1} \delta(\hat{y}_i - y_i)
\]
Overall Recall:

\[
TPR = \frac{TP}{TP + FN} = \frac{1}{N} \sum_{i=0}^{N-1} \delta(\hat{y}_i - y_i)
\]

Overall F1-measure:

\[
F1 = 2 \cdot \left( \frac{PPV \cdot TPR}{PPV + TPR} \right)
\]

Precision by label:

\[
PPV(\ell) = \frac{TP}{TP + FP} = \frac{\sum_{i=0}^{N-1} \delta(\hat{y}_i - \ell) \cdot \delta(y_i - \ell)}{\sum_{i=0}^{N-1} \delta(y_i - \ell)}
\]

Recall by label:

\[
TPR(\ell) = \frac{TP}{P} = \frac{\sum_{i=0}^{N-1} \delta(\hat{y}_i - \ell) \cdot \delta(y_i - \ell)}{\sum_{i=0}^{N-1} \delta(y_i - \ell)}
\]

F-measure by label:

\[
F(\beta, \ell) = \left(1 + \beta^2\right) \cdot \left( \frac{PPV(\ell) \cdot TPR(\ell)}{\beta^2 \cdot PPV(\ell) + TPR(\ell)} \right)
\]

Weighted precision:

\[
PPV_w = \frac{1}{N} \sum_{\ell \in L} PPV(\ell) \cdot \sum_{i=0}^{N-1} \delta(y_i - \ell)
\]

Weighted recall:

\[
TPR_w = \frac{1}{N} \sum_{\ell \in L} TPR(\ell) \cdot \sum_{i=0}^{N-1} \delta(y_i - \ell)
\]

Weighted F-measure:

\[
F_w(\beta) = \frac{1}{N} \sum_{\ell \in L} F(\beta, \ell) \cdot \sum_{i=0}^{N-1} \delta(y_i - \ell)
\]
Med växande miljöhot så blir det allt mer viktigt att vi alla minimerar vår miljöpåverkan. För att förenkla detta har vi utvecklat en app som kan räkna ut miljöpåverkan genom en användares bankutdrag.

Dagens konsumtion bidrar till den globala uppvärmningen, och alla möjliga typer av miljöföroreningar. Detta gäller även för varor som vi tar för givet, såsom vatten. Vatten är en bristvara och 1,2 miljarder människor i världen lever i områden med brist på vatten. Detta leder till lidande för både människor, djur och natur. Allt detta gör att det finns många anledningar till varför både företag och vanliga konsumenter bör minska sin miljöpåverkan.

Vi som tillhör den 10 % rikaste delen av världen har dock också störst möjlighet att göra någonting åt detta. Mycket p.g.a. våra ekonomiska muskler, men också till följd av vår höga konsumtion. Exempelvis så genererar vi direkt och indirekt 50 % av världens koldioxidutsläpp.


Specifikt så läggs först bankutdraget med alla inköp in i systemet. Inköpen kategoriseras sedan utifrån förklädningskategori, såsom elektronikaffär eller restaurang, varpå systemet sedan approximerar vilka specifika typer av produkter och tjänster som har inköpts. Genom att hämta vad dessa produkter och tjänster har för påverkan från vår databas så kan en estimering av inköpens miljöpåverkan och andra etiska konsekvenser räknas ut. Detta kan bl.a. vara miljöpåverkan som land- och vattenanvändning eller koldioxid men även mer etisk påverkan som antalet människor (i antal livsår) eller djurliv som fått sina liv förkortade till följd av användarens konsumtion.

Mitt fokus för examensarbetet var att kategorisera alla inköp med hjälp av en form av artificiell intelligens, kallad maskininlärning. Under examensarbete färde jag programvaran att känna igen inköp utefter vad som står i beskrivningen för varje bankutdrag.

Appen är i nuläget i alfastadiet men kommer att finnas tillgänglig på både webben, mobilen och surfplattan. Betaversionen släpps snart! Håll utkik på Meta Minds hemsida (www.metamind.se) för mer information.